Document Number: P1037
Date: 2018-05-06
Project: C++ Extensions for Ranges, Library Evolution Working Group
Authors: Eric Niebler
Casey Carter
Reply to: Eric Niebler
eric.niebler@gmail.com

Deep Integration of the Ranges TS

Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad formatting.

Contents

1 Scope [intro.scope]

> “Design is not making beauty, beauty emerges from selection, affinities, integration, love.”
>
> —*Louis Kahn*

1 This document proposes changes to the components of namespace `std` and namespace `std::ranges` to deepen the integration of the components of the Ranges TS into the Standard Library.

2 Normative References [intro.refs]

1 The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

(1.1) — ISO/IEC 14882, *Programming Languages - C++*

(1.2) — ISO/IEC TS 21425:2017, *Technical Specification - C++ Extensions for Ranges*

ISO/IEC 14882 is herein called the *C++ Standard* and ISO/IEC TS 21425:2017 is called the *Ranges TS*.

3 General Principles [intro]

3.1 Goals [intro.goals]

1 The goal of this paper is to reduce the needless duplication of functionality that would come from naively dropping the Ranges TS into the `std::ranges` namespace. We also aim to simplify the job of authoring an iterator type that works with both existing generic code, which expects types to conform to the iterator requirements tables ([iterator.requirements]) as well as to satisfy the requirements of the iterator concepts as defined in the Ranges TS.

This paper builds a bridge between the pre-concepts STL and the newer constrained facilities.

3.2 Methodology [intro.methedology]

3.2.1 Background [intro.background]

1 When the STL was first written, there was no way test an expression to see if it was well-formed without causing a hard error at compile-time. As a result, iterators needed to explicitly tell generic code what concept they satisfied via an `::iterator_category` typedef, either within a specialization of `std::iterator_traits` or nested within the iterator type itself.

2 With the addition of generalized SFINAE for expressions ([3]), and especially with the addition of concepts, generic code no longer strictly needs the iterator to declare which concept is satisfied; instead, generic code can inspect which required expressions the type supports. Iterator tag types are needed only to disambiguate between concepts that differ only in semantics or to opt out of accidental conformance.

3 For legacy reasons, in both the working draft and in the Ranges TS, generic algorithms still require iterator types to declare their category via tag. This paper proposes doing away with that requirement, using syntactic conformance to infer the tag type for legacy code that still requires it. New code will simply use *requires-clauses* and the iterator concepts from the Ranges TS to constrain algorithms and select implementations.

4 A great deal of the machinery in the Ranges TS duplicates existing functionality in Standard Library. This made sense when the Ranges TS was seen as the beginning of a fork of the Standard Library itself. That duplication now seems like bad design.

5 By leveraging the compiler's new ability to infer concept satisfaction and by reusing existing library functionality, we can make it easier for users to author iterator types that meet the expectations of old generic code, while also addressing the shortcomings of the STL that the Ranges TS aimed to address.

3.2.2 Implementation strategy [intro.strategy]

3.2.2.1 Iterator tags [intro.iterator_tags]

1 The Ranges TS defines a parallel set of iterator tag types, from `input_iterator_tag` and `output_iterator_tag`, through `random_access_iterator_tag`. This paper proposes to remove those tag types and simply reuse the existing ones, with the addition of a new `contiguous_iterator_tag` type that represents a refinement of random-access iterators.

3.2.2.2 `iterator_concept` [intro.iterator_concept]

1 There are several differences between the iterator concepts as defined in the C++ Standard and the Ranges TS. To accomodate the fact that a single type might satisfy different iterator concepts in the different standards – and the fact that there will always be times when it is necessary to explicitly specify conformance

with a tag type – this paper proposes to add an additional optional member to `std::iterator_traits` named `::iterator_concept`.

2 As always, `std::iterator_traits<I>::iterator_category` specifies which of the requirements tables in [iterator.requirements] the type `I` purports to satisfy, whereas `std::iterator_traits<I>::iterator_concept`, when present, is used to opt-in or -out of satisfaction of concepts as defined in the Ranges TS.

3.2.2.3 Specifying associated types [intro.spec_assoc_types]

1 The Ranges TS defines three different customization points – `ranges::value_type<>`, `ranges::difference_type<>`, and `ranges::iterator_category<>` – for specifying the associated types for the `Readable`, `WeaklyIncrementable`, and `InputIterator` concepts, respectively.

2 With the addition of the optional `iterator_concept` nested typedef of `std::iterator_traits<>`, the `ranges::iterator_category<>` customization point is no longer necessary. Should users need to non-intrusively specify an iterator's tag, they may simply specialize `std::iterator_traits` as they always have.

3 For the sake of clarity and consistency with the naming of `std::iterator_traits`, we suggest renaming `std::ranges::value_type<I>::type` to `std::ranges::readable_traits<I>::value_type`. Likewise, `std::ranges::difference_type<I>::type` is renamed to `std::ranges::incrementable_traits<I>::difference_type`.

4 The primary `std::iterator_traits<>` template uses `std::ranges::readable_traits<>` and `std::ranges::incrementable_traits<>` for computing `::value_type` and `::difference_type`, respectively. That way, users need only specify these traits once.

3.2.2.4 Using associated types [intro.use_assoc_types]

1 To specify an iterator's `value_type`, users may specialize either `std::iterator_traits<>`, `std::ranges::readable_traits<>`, or both. If they specialize `iterator_traits<>` and not `readable_traits<>`, then `readable_traits<I>::value_type` will not reflect the value type they specified in `iterator_traits`. Generic code that uses `typename std::ranges::readable_traits<I>::value_type` directly is most likely wrong when `I` is an iterator type.

2 One possible solution would be to simply remove `std::ranges::readable_traits` and tell people to use `iterator_traits` to specify the value type of their `Readable` types. However, there are readable types that are not iterators; for example, `std::optional<int>`. Needing to specialize something called "`iterator_traits`" and specify a `difference_type` for something that is not incrementable would be strange, as would telling users to create a specialization of `iterator_traits` that lacks some of the traditional typedefs.

3 Instead, we propose to use the alias template `value_type_t<>` – renamed to `iter_value_t` and promoted to namespace `std` – from the Ranges TS, and to make it smarter. Rather than immediately dispatching to either `iterator_traits<I>` or `readable_traits<I>`, `iter_value_t<I>` first tests whether `iterator_traits<I>` has selected the primary template. If so, `iter_value_t<I>` is an alias for `std::iterator_traits<I>::value_type`; otherwise, it is an alias for `std::ranges::readable_traits<I>::value_type`.

4 The `difference_type_t` alias from the Ranges TS, renamed to `std::iter_difference_t`, gets the same treatment.

5 Testing whether a particular instantiation has selected the primary template is a matter of giving the primary template some testable property that specializations would lack. For instance, the primary template `std::iterator_traits<I>` might define a hidden nested typedef `__unspecialized` that is an alias for `I`. Any instantiation `std::iterator_traits<I>` that either lacks that typedef or has one that is not an alias for `I` is necessarily not the primary template.

3.2.2.5 Names of iterator associated types [intro.naming]

1 As already mentioned above, the aliases `std::ranges::value_type_t` and `std::ranges::difference_type_t` are renamed to `std::iter_value_t` and `std::iter_difference_t`. Here is the complete list of iterator associated types, and their old and new names as suggested by this paper:

Table 1 — Iterator associated types

Old name	New name
`std::ranges::difference_type_t`	`std::iter_difference_t`
`std::ranges::value_type_t`	`std::iter_value_t`
`std::ranges::reference_t`	`std::iter_reference_t`
`std::ranges::rvalue_reference_t`	`std::iter_rvalue_reference_t`

2 The new names correct two problems with the old names: the term “`type`” and the suffix “`_t`” both carry the same semantic information, and names like “value type” and “reference” are overly general. The prefix “`iter_`” already means “relating to iterators”, as in `iter_swap`.

3.2.3 Implementation experience [intro.impl]

1 The design described in this document has been prototyped and is available online for analysis and experimentation here: `https://gist.github.com/ericniebler/d07bfb0d8ebf2e25f94f2111f893ec30`.

2 Given the scope of this work, it makes sense to insist on a full implementation of this within an existing Standard Library implementation. That work has not yet begun.

3.2.4 Method of specification [intro.spec]

1 The changes suggested by this paper are presented as a set of diffs against the working draft as amended by P0896R1 ([2]) and P0944R0 ([1]).

2 Where the text of the Ranges TS or the working draft needs to be updated, the text is presented with change markings: ~~red strikethrough~~ for removed text and blue underline for added text.

28 Iterators library [iterators]

28.1 General [iterators.general]

1 This Clause describes components that C++ programs may use to perform iterations over containers26, streams30.7, ~~and~~ stream buffers30.6, and other ranges (Clause 29).

2 The following subclauses describe iterator requirements, and components for iterator primitives, predefined iterators, and stream iterators, as summarized in Table 2.

Table 2 — Iterators library summary

Subclause		Header(s)
28.3	~~R~~Iterator requirements	`<iterator>`
28.3.6	Indirect callable requirements	
28.3.7	Common algorithm requirements	
28.4	Iterator primitives	~~`<iterator>`~~
28.5	Predefined iterators	
28.6	Stream iterators	

[Editor's note: Move the section "Header `iterator` synopsis" to immediately follow [iterators.general] and precede [iterator.requirements], and change it as follows:]

28.2 Header `<iterator>` synopsis [iterator.synopsis]

```
#include <concepts>

namespace std {
  [Editor's note: Relocated from the header <range> synopsis, and dereferenceable is promoted to
namespace std::.]
  template <class T> concept dereferenceable  // exposition only
    = requires(T& t) { {*t} -> auto&&; };

  [Editor's note: The following through the definition of indirect_result_t is also relocated from
<range> but promoted to namespace std::.]
  // 28.3.2, associated types:
  // 28.3.2.1, ~~difference_type~~incrementable traits:
  template <class> struct ~~difference_type~~incrementable_traits;
  template <class T>
    using ~~difference_type_t~~iter_difference_t = see below;
      ~~= typename difference_type<T>::type;~~

  // 28.3.2.2, ~~value_type~~readable traits:
  template <class> struct ~~value_type~~readable_traits;
  template <class T>
    using ~~value_type_t~~iter_value_t = see below;
      ~~= typename value_type<T>::type;~~

  ~~// ??, iterator_category:~~
  ~~template <class> struct iterator_category;~~
  ~~template <class T> using iterator_category_t~~
    ~~= typename iterator_category<T>::type;~~
```

```
// 28.3.2.3, Iterator traits [Editor's note: Moved here from below.]
template<class Iterator> struct iterator_traits;
template<class T> struct iterator_traits<T*>;

template <dereferenceable T>
  using iter_reference_t = decltype(*declval<T&>());

template <Readable T>
  using iter_common_reference_t = common_reference_t<iter_reference_t<T>,
                                                      ~~value_type_t~~iter_value_t<T>&>;

template <dereferenceable T>
    requires ~~see below~~requires (T& t) {
      { ranges::iter_move(t) } -> auto &&;
    }
  using iter_rvalue_reference_t = decltype(ranges::iter_move(declval<T&>()));

namespace ranges {
  // 28.3.3, customization points:
  inline namespace unspecified {
    // 28.3.3.0.1, iter_move:
    inline constexpr unspecified iter_move = unspecified ;

    // 28.3.3.0.2, iter_swap:
    inline constexpr unspecified iter_swap = unspecified ;
  }
}

// 28.3.4, iterator ~~requirements~~concepts:
// 28.3.4.1, Readable:
template <class In>
concept Readable = see below;

// 28.3.4.2, Writable:
template <class Out, class T>
concept Writable = see below;

// 28.3.4.3, WeaklyIncrementable:
template <class I>
concept WeaklyIncrementable = see below;

// 28.3.4.4, Incrementable:
template <class I>
concept Incrementable = see below;

// 28.3.4.5, Iterator:
template <class I>
concept Iterator = see below;

// 28.3.4.4, Sentinel:
template <class S, class I>
concept Sentinel = see below;

// 28.3.4.7, SizedSentinel:
```

```
template <class S, class I>
constexpr bool disable_sized_sentinel = false;

template <class S, class I>
concept SizedSentinel = see below;

// 28.3.4.8, InputIterator:
template <class I>
concept InputIterator = see below;

// 28.3.4.9, OutputIterator:
template <class I>
concept OutputIterator = see below;

// 28.3.4.10, ForwardIterator:
template <class I>
concept ForwardIterator = see below;

// 28.3.4.11, BidirectionalIterator:
template <class I>
concept BidirectionalIterator = see below;

// 28.3.4.12, RandomAccessIterator:
template <class I>
concept RandomAccessIterator = see below;

// 28.3.4.13, ContiguousIterator:
template <class I>
concept ContiguousIterator = see below;

// 28.3.6, indirect callable requirements:
// 28.3.6.2, indirect callables:
template <class F, class I>
concept IndirectUnaryInvocable = see below;

template <class F, class I>
concept IndirectRegularUnaryInvocable = see below;

template <class F, class I>
concept IndirectUnaryPredicate = see below;

template <class F, class I1, class I2 = I1>
concept IndirectRelation = see below;

template <class F, class I1, class I2 = I1>
concept IndirectStrictWeakOrder = see below;

template <class, class...> struct indirect_result { };

template <class F, class... Is>
  requires (Readable<Is> &&...) && Invocable<F, iter_reference_t<Is>...>
struct indirect_result<F, Is...>;

template <class F, class... Is>
using indirect_result_t = typename indirect_result<F, Is...>::type;
```

```
// 28.3.6.3, projected:
template <Readable I, IndirectRegularUnaryInvocable<I> Proj>
struct projected;

template <WeaklyIncrementable I, class Proj>
struct ~~difference_type~~incrementable_traits<projected<I, Proj>>;

// 28.3.7, common algorithm requirements:
// 28.3.7.2 IndirectlyMovable:
template <class In, class Out>
concept IndirectlyMovable = see below;

template <class In, class Out>
concept IndirectlyMovableStorable = see below;

// 28.3.7.3 IndirectlyCopyable:
template <class In, class Out>
concept IndirectlyCopyable = see below;

template <class In, class Out>
concept IndirectlyCopyableStorable = see below;

// 28.3.7.4 IndirectlySwappable:
template <class I1, class I2 = I1>
concept IndirectlySwappable = see below;

// 28.3.7.5 IndirectlyComparable:
template <class I1, class I2, class R = ranges::equal_to<>, class P1 = identity,
    class P2 = identity>
concept IndirectlyComparable = see below;

// 28.3.7.6 Permutable:
template <class I>
concept Permutable = see below;

// 28.3.7.7 Mergeable:
template <class I1, class I2, class Out,
    class R = ranges::less<>, class P1 = identity, class P2 = identity>
concept Mergeable = see below;

template <class I, class R = ranges::less<>, class P = identity>
concept Sortable = see below;
```

[Editor's note: ranges::iterator_traits from P0896 is intentionally omitted.]

```
// 28.4, primitives
// 28.4.1, iterator tags
struct input_iterator_tag { };
struct output_iterator_tag { };
struct forward_iterator_tag: public input_iterator_tag { };
struct bidirectional_iterator_tag: public forward_iterator_tag { };
struct random_access_iterator_tag: public bidirectional_iterator_tag { };
struct contiguous_iterator_tag: public random_access_iterator_tag { };
```

[Editor's note: The clones of the iterator tags in the ranges namespace from P0896R1 are intentionally omitted here.]

```
// 28.4.2, iterator operations
template<class InputIterator, class Distance>
  constexpr void
    advance(InputIterator& i, Distance n);
template<class InputIterator>
  constexpr typename iterator_traits<InputIterator>::difference_type
    distance(InputIterator first, InputIterator last);
template<class InputIterator>
  constexpr InputIterator
    next(InputIterator x,
         typename iterator_traits<InputIterator>::difference_type n = 1);
template<class BidirectionalIterator>
  constexpr BidirectionalIterator
    prev(BidirectionalIterator x,
         typename iterator_traits<BidirectionalIterator>::difference_type n = 1);

// 28.4.3, range iterator operations
namespace ranges {
  // 28.4.3, Range iterator operations:
  // 28.4.3.1, ranges::advance:
  template <Iterator I>
    constexpr void advance(I& i, iter_difference~~_type~~_t<I> n);
  template <Iterator I, Sentinel<I> S>
    constexpr void advance(I& i, S bound);
  template <Iterator I, Sentinel<I> S>
    constexpr iter_difference~~_type~~_t<I> advance(I& i, iter_difference~~_type~~_t<I> n, S bound);

  // 28.4.3.2, ranges::distance:
  template <Iterator I, Sentinel<I> S>
    constexpr iter_difference~~_type~~_t<I> distance(I first, S last);
  template <Range R>
    constexpr iter_difference~~_type~~_t<iterator_t<R>> distance(R&& r);

  // 28.4.3.3, ranges::next:
  template <Iterator I>
    constexpr I next(I x);
  template <Iterator I>
    constexpr I next(I x, iter_difference~~_type~~_t<I> n);
  template <Iterator I, Sentinel<I> S>
    constexpr I next(I x, S bound);
  template <Iterator I, Sentinel<I> S>
    constexpr I next(I x, iter_difference~~_type~~_t<I> n, S bound);

  // 28.4.3.4, ranges::prev:
  template <BidirectionalIterator I>
    constexpr I prev(I x);
  template <BidirectionalIterator I>
    constexpr I prev(I x, iter_difference~~_type~~_t<I> n);
  template <BidirectionalIterator I>
    constexpr I prev(I x, iter_difference~~_type~~_t<I> n, I bound);
}
```

```
// 28.5, predefined iterators @and sentinels@
template<class Iterator> class reverse_iterator;

template<class Iterator1, class Iterator2>
  constexpr bool operator==(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
template<class Iterator1, class Iterator2>
  constexpr bool operator<(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
template<class Iterator1, class Iterator2>
  constexpr bool operator!=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
template<class Iterator1, class Iterator2>
  constexpr bool operator>(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
template<class Iterator1, class Iterator2>
  constexpr bool operator>=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
template<class Iterator1, class Iterator2>
  constexpr bool operator<=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
  constexpr auto operator-(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y) -> decltype(y.base() - x.base());
template<class Iterator>
  constexpr reverse_iterator<Iterator>
    operator+(
  typename reverse_iterator<Iterator>::difference_type n,
  const reverse_iterator<Iterator>& x);

template<class Iterator>
  constexpr reverse_iterator<Iterator> make_reverse_iterator(Iterator i);

template<class Container> class back_insert_iterator;
template<class Container>
  back_insert_iterator<Container> back_inserter(Container& x);

template<class Container> class front_insert_iterator;
template<class Container>
  front_insert_iterator<Container> front_inserter(Container& x);

template<class Container> class insert_iterator;
template<class Container>
  insert_iterator<Container> inserter(Container& x, typename Container::iterator i);
[Editor's note:  The insert iterators from P0896R1 are intentionally omitted.]

template<class Iterator> class move_iterator;
```

```
template<class Iterator1, class Iterator2>
  constexpr bool operator==(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template<class Iterator1, class Iterator2>
  constexpr bool operator!=(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template<class Iterator1, class Iterator2>
  constexpr bool operator<(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template<class Iterator1, class Iterator2>
  constexpr bool operator<=(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template<class Iterator1, class Iterator2>
  constexpr bool operator>(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template<class Iterator1, class Iterator2>
  constexpr bool operator>=(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
  constexpr auto operator-(
  const move_iterator<Iterator1>& x,
  const move_iterator<Iterator2>& y) -> decltype(x.base() - y.base());
template<class Iterator>
  constexpr move_iterator<Iterator> operator+(
    typename move_iterator<Iterator>::difference_type n, const move_iterator<Iterator>& x);
template<class Iterator>
  constexpr move_iterator<Iterator> make_move_iterator(Iterator i);
```

[Editor's note: `move_sentinel` is taken from the `<range>` synopsis of P0896R1.]

```
template <Semiregular S> class move_sentinel;
```

[Editor's note: `common_iterator` and associated types specializations are taken from the `<range>` synopsis of P0896R1.]

```
// 28.5.4, common iterators:
template <Iterator I, Sentinel<I> S>
  requires !Same<I, S>
class common_iterator;

template <Readable I, class S>
struct ~~value_type~~readable_traits<common_iterator<I, S>>;

template <InputIterator I, class S>
struct iterator_~~category~~traits<common_iterator<I, S>>;

template <ForwardIterator I, class S>
struct iterator_~~category~~traits<common_iterator<I, S>>;

template <class I1, class I2, Sentinel<I2> S1, Sentinel<I1> S2>
bool operator==(
  const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);
template <class I1, class I2, Sentinel<I2> S1, Sentinel<I1> S2>
  requires EqualityComparableWith<I1, I2>
bool operator==(
  const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);
```

```
template <class I1, class I2, Sentinel<I2> S1, Sentinel<I1> S2>
bool operator!=(
  const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);

template <class I2, SizedSentinel<I2> I1, SizedSentinel<I2> S1, SizedSentinel<I1> S2>
iter_difference~~_type~~_t<I2> operator-(
  const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);

// 28.5.5, default sentinels:
class default_sentinel;

// 28.5.6, counted iterators:
template <Iterator I> class counted_iterator;

template <Readable I>
  struct readable_traits<counted_iterator<I>>;

template <InputIterator I>
  struct iterator_traits<counted_iterator<I>>;

template <class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator==(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
// ... as in P0896R1, but relocated into namespace std
```

[Editor's note: P0896R1's versions of reverse_iterator and move_iterator are intentionally omitted.]

[Editor's note: unreachable is from P0896R1 and promoted to namespace std.]

```
class unreachable;

template <~~Iterator~~WeaklyIncrementable I>
  constexpr bool operator==(const I&, unreachable) noexcept;
template <~~Iterator~~WeaklyIncrementable I>
  constexpr bool operator==(unreachable, const I&) noexcept;
template <~~Iterator~~WeaklyIncrementable I>
  constexpr bool operator!=(const I&, unreachable) noexcept;
template <~~Iterator~~WeaklyIncrementable I>
  constexpr bool operator!=(unreachable, const I&) noexcept;

// 28.6, stream iterators
template<class T, class charT = char, class traits = char_traits<charT>,
         class Distance = ptrdiff_t>
class istream_iterator;
template<class T, class charT, class traits, class Distance>
  bool operator==(const istream_iterator<T,charT,traits,Distance>& x,
          const istream_iterator<T,charT,traits,Distance>& y);
template<class T, class charT, class traits, class Distance>
  bool operator!=(const istream_iterator<T,charT,traits,Distance>& x,
          const istream_iterator<T,charT,traits,Distance>& y);

template<class T, class charT = char, class traits = char_traits<charT>>
    class ostream_iterator;
```

```
    template<class charT, class traits = char_traits<charT>>
      class istreambuf_iterator;
    template<class charT, class traits>
      bool operator==(const istreambuf_iterator<charT,traits>& a,
              const istreambuf_iterator<charT,traits>& b);
    template<class charT, class traits>
      bool operator!=(const istreambuf_iterator<charT,traits>& a,
              const istreambuf_iterator<charT,traits>& b);

    template<class charT, class traits = char_traits<charT>>
      class ostreambuf_iterator;
    [Editor's note:  The stream iterators from P0896R1 are intentionally omitted.]

    // , range access
    template<class C> constexpr auto begin(C& c) -> decltype(c.begin());
    template<class C> constexpr auto begin(const C& c) -> decltype(c.begin());
    template<class C> constexpr auto end(C& c) -> decltype(c.end());
    template<class C> constexpr auto end(const C& c) -> decltype(c.end());
    template<class T, size_t N> constexpr T* begin(T (&array)[N]) noexcept;
    template<class T, size_t N> constexpr T* end(T (&array)[N]) noexcept;
    template<class C> constexpr auto cbegin(const C& c) noexcept(noexcept(std::begin(c)))
      -> decltype(std::begin(c));
    template<class C> constexpr auto cend(const C& c) noexcept(noexcept(std::end(c)))
      -> decltype(std::end(c));
    template<class C> constexpr auto rbegin(C& c) -> decltype(c.rbegin());
    template<class C> constexpr auto rbegin(const C& c) -> decltype(c.rbegin());
    template<class C> constexpr auto rend(C& c) -> decltype(c.rend());
    template<class C> constexpr auto rend(const C& c) -> decltype(c.rend());
    template<class T, size_t N> constexpr reverse_iterator<T*> rbegin(T (&array)[N]);
    template<class T, size_t N> constexpr reverse_iterator<T*> rend(T (&array)[N]);
    template<class E> constexpr reverse_iterator<const E*> rbegin(initializer_list<E> il);
    template<class E> constexpr reverse_iterator<const E*> rend(initializer_list<E> il);
    template<class C> constexpr auto crbegin(const C& c) -> decltype(std::rbegin(c));
    template<class C> constexpr auto crend(const C& c) -> decltype(std::rend(c));

    // , container access
    template<class C> constexpr auto size(const C& c) -> decltype(c.size());
    template<class T, size_t N> constexpr size_t size(const T (&array)[N]) noexcept;
    template<class C> [[nodiscard]] constexpr auto empty(const C& c) -> decltype(c.empty());
    template<class T, size_t N> [[nodiscard]] constexpr bool empty(const T (&array)[N]) noexcept;
    template<class E> [[nodiscard]] constexpr bool empty(initializer_list<E> il) noexcept;
    template<class C> constexpr auto data(C& c) -> decltype(c.data());
    template<class C> constexpr auto data(const C& c) -> decltype(c.data());
    template<class T, size_t N> constexpr T* data(T (&array)[N]) noexcept;
    template<class E> constexpr const E* data(initializer_list<E> il) noexcept;
  }
```

[Editor's note: For the remainder of Clauses [iterator] and [ranges], textually replace `value_type<>`, `difference_type<>`, `value_type_t<>`, `difference_type_t<>`, `reference_t<>`, and `rvalue_reference_t<>` with `readable_traits<>`, `incrementable_traits<>`, `iter_value_t<>`, `iter_difference_t<>`, `iter_reference_t<>`, and `iter_rvalue_reference_t<>`, respectively.]

28.3 Iterator requirements [iterator.requirements]

28.3.1 In general [iterator.requirements.general]

1 Iterators are a generalization of pointers that allow a C++ program to work with different data structures (for example, containers and ranges) in a uniform manner. To be able to construct template algorithms that work correctly and efficiently on different types of data structures, the library formalizes not just the interfaces but also the semantics and complexity assumptions of iterators. An input iterator `i` supports the expression `*i`, resulting in a value of some object type `T`, called the *value type* of the iterator. An output iterator `i` has a non-empty set of types that are *writable* to the iterator; for each such type `T`, the expression `*i = o` is valid where `o` is a value of type `T`. ~~An iterator `i` for which the expression `(*i).m` is well-defined supports the expression `i->m` with the same semantics as `(*i).m`.~~ For every iterator type `X` ~~for which equality is defined~~, there is a corresponding signed integer type called the *difference type* of the iterator.

2 Since iterators are an abstraction of pointers, their semantics is a generalization of most of the semantics of pointers in C++. This ensures that every function template that takes iterators works as well with regular pointers. This document defines six categories of iterators, according to the operations defined on them: *input iterators*, *output iterators*, *forward iterators*, *bidirectional iterators*, *random access iterators*, and *contiguous iterators*, as shown in Table 3.

Table 3 — Relations among iterator categories

Contiguous	→ **Random Access**	→ **Bidirectional**	→ **Forward**	→ **Input**
				→ **Output**

3 Forward iterators satisfy all the requirements of input iterators and can be used whenever an input iterator is specified; Bidirectional iterators also satisfy all the requirements of forward iterators and can be used whenever a forward iterator is specified; Random access iterators also satisfy all the requirements of bidirectional iterators and can be used whenever a bidirectional iterator is specified; Contiguous iterators also satisfy all the requirements of random access iterators and can be used whenever a random access iterator is specified.

4 Iterators that further satisfy the requirements of output iterators are called *mutable iterators*. Nonmutable iterators are referred to as *constant iterators*.

5 In addition to the requirements in this subclause, the nested *typedef-name*s specified in 28.3.2.3 shall be provided for the iterator type. [*Note:* Either the iterator type must provide the *typedef-name*s directly (in which case `iterator_traits` pick them up automatically), or an `iterator_traits` specialization must provide them. — *end note*]

6 Iterators that further satisfy the requirement that, for integral values `n` and dereferenceable iterator values `a` and `(a + n)`, `*(a + n)` is equivalent to `*(addressof(*a) + n)`, are called *contiguous iterators*. [*Note:* For example, the type "pointer to `int`" is a contiguous iterator, but `reverse_iterator<int *>` is not. For a valid iterator range [`a`,`b`) with dereferenceable `a`, the corresponding range denoted by pointers is [`addressof(*a)`,`addressof(*a) + (b - a)`); `b` might not be dereferenceable. — *end note*] [Editor's note: TODO: The term "contiguous iterator" is now *extremely* confusing in that it can easily be conflated with "types that satisfy the `ContiguousIterator` concept." Consider removing it.]

7 Just as a regular pointer to an array guarantees that there is a pointer value pointing past the last element of the array, so for any iterator type there is an iterator value that points past the last element of a corresponding sequence. These values are called *past-the-end* values. Values of an iterator `i` for which the expression `*i` is defined are called *dereferenceable*. The library never assumes that past-the-end values are dereferenceable. Iterators can also have singular values that are not associated with any sequence. [*Example:* After the declaration of an uninitialized pointer `x` (as with `int* x;`), `x` must always be assumed to have a

singular value of a pointer. — *end example*] Results of most expressions are undefined for singular values; the only exceptions are destroying an iterator that holds a singular value, the assignment of a non-singular value to an iterator that holds a singular value, and, for iterators that satisfy the *CppDefaultConstructible* requirements, using a value-initialized iterator as the source of a copy or move operation. [*Note:* This guarantee is not offered for default-initialization, although the distinction only matters for types with trivial default constructors such as pointers or aggregates holding pointers. — *end note*] In these cases the singular value is overwritten the same way as any other value. Dereferenceable values are always non-singular.

8 An iterator `j` is called *reachable* from an iterator `i` if and only if there is a finite sequence of applications of the expression `++i` that makes `i == j`. If `j` is reachable from `i`, they refer to elements of the same sequence.

9 Most of the library's algorithmic templates that operate on data structures have interfaces that use ranges. A *range* is a pair of iterators that designate the beginning and end of the computation. A range `[i,i)` is an empty range; in general, a range `[i,j)` refers to the elements in the data structure starting with the element pointed to by `i` and up to but not including the element pointed to by `j`. Range `[i,j)` is valid if and only if `j` is reachable from `i`. The result of the application of functions in the library to invalid ranges is undefined.

10 Most of the library's algorithmic templates that operate on data structures have interfaces that use ranges. A *range* is an iterator and a *sentinel* that designate the beginning and end of the computation, or an iterator and a count that designate the beginning and the number of elements to which the computation is to be applied.[1]

11 An iterator and a sentinel denoting a range are comparable. ~~The types of a sentinel and an iterator that denote a range must satisfy `Sentinel` (28.3.4.4).~~ A range `[i,s)` is empty if `i == s`; otherwise, `[i,s)` refers to the elements in the data structure starting with the element pointed to by `i` and up to but not including the element pointed to by the first iterator `j` such that `j == s`.

12 A sentinel `s` is called *reachable* from an iterator `i` if and only if there is a finite sequence of applications of the expression `++i` that makes `i == s`. If `s` is reachable from `i`, `[i,s)` denotes a range.

13 A counted range `[i,n)` is empty if `n == 0`; otherwise, `[i,n)` refers to the `n` elements in the data structure starting with the element pointed to by `i` and up to but not including the element pointed to by the result of incrementing `i` `n` times.

14 A range `[i,s)` is valid if and only if `s` is reachable from `i`. A counted range `[i,n)` is valid if and only if `n == 0`; or `n` is positive, `i` is dereferenceable, and `[++i,--n)` is valid. The result of the application of functions in the library to invalid ranges is undefined.

15 All the categories of iterators require only those functions that are realizable for a given category in constant time (amortized). Therefore, requirement tables for the iterators do not have a complexity column.

16 Destruction of an iterator whose category is weaker than forward may invalidate pointers and references previously obtained from that iterator.

17 An *invalid* iterator is an iterator that may be singular.[2]

18 Iterators are called *constexpr iterators* if all operations provided to satisfy iterator category operations are constexpr functions, except for

(18.1) — `swap`,

(18.2) — a pseudo-destructor call, and

(18.3) — the construction of an iterator with a singular value.

---

[1] The sentinel denoting the end of a range may have the same type as the iterator denoting the beginning of the range, or a different type.

[2] This definition applies to pointers, since pointers are iterators. The effect of dereferencing an iterator that has been invalidated is undefined.

[*Note:* For example, the types “pointer to `int`” and `reverse_iterator<int*>` are constexpr iterators. — *end note*]

19 In the following sections, `a` and `b` denote values of type `X` or `const X`, `difference_type` and `reference` refer to the types `iterator_traits<X>::difference_type` and `iterator_traits<X>::reference`, respectively, `n` denotes a value of `difference_type`, `u`, `tmp`, and `m` denote identifiers, `r` denotes a value of `X&`, `t` denotes a value of value type `T`, `o` denotes a value of some type that is writable to the output iterator. [*Note:* For an iterator type `X` there must be an instantiation of `iterator_traits<X>` (28.3.2.3). — *end note*]

[Editor’s note: Relocate [ranges.iterator.assoc.types] from P0896R1 here and change its name to “Associated types”.]

28.3.2 Associated types [iterator.assoc.types]

[Editor’s note: Changed as follows:]

1 To implement algorithms only in terms of iterators, it is often necessary to determine the value and difference types that correspond to a particular iterator type. Accordingly, it is required that if `WI` is the name of a type that satisfies the `WeaklyIncrementable` concept (28.3.4.3), `R` is the name of a type that satisfies the `Readable` concept (28.3.4.1), and `II` is the name of a type that satisfies the `InputIterator` concept (28.3.4.8) concept, the types

```
difference_type_t<WI>
value_type_t<R>
iterator_category_t<II>
```

be defined as the iterator’s difference type, value type and iterator category, respectively.

[Editor’s note: Change the name of [iterator.assoc.types.difference_type] from “`difference_type`” to “`Incrementable traits`” and change its stable name to [incrementable.traits].]

28.3.2.1 Incrementable traits [incrementable.traits]

1 To implement algorithms only in terms of incrementable types, it is often necessary to determine the difference type that corresponds to a particular incrementable type. Accordingly, it is required that if `WI` is the name of a type that satisfies the `WeaklyIncrementable` concept (28.3.4.3), the type

```
iter_difference_t<WI>
```

be defined as the incrementable type’s difference type.

2 ~~`difference_type_t<T>`~~`iter_difference_t` is implemented as if:

```
namespace std { namespace ranges {
  struct __empty { }; // exposition only

  template <class T> struct __with_difference_type { // exposition only
    using difference_type = T;
  };

  template <class> struct ~~difference_type~~incrementable_traits { };

  template <class T>
  struct ~~difference_type~~incrementable_traits<T*>
    : ~~enable_if~~conditional_t<is_object_v<T>~~::value~~,
        __with_difference_type<ptrdiff_t>, __empty> { };

  template <class I>
  struct ~~difference_type~~incrementable_traits<const I>
```

```
    : ~~difference_type~~incrementable_traits<decay_t<I>> { };

  template <class T>
    requires requires { typename T::difference_type; }
  struct ~~difference_type~~incrementable_traits<T> {
    using difference_type = typename T::difference_type;
  };

  template <class T>
    requires !requires { typename T::difference_type; } &&
      requires(const T& a, const T& b) { { a - b } -> Integral; }
  struct ~~difference_type~~incrementable_traits<T>
    : __with_difference_type< make_signed_t< decltype(declval<T>() - declval<T>()) >> {
  };

  template <class T>
    using ~~difference_type_t~~iter_difference_t = see below;
      ~~= typename difference_type<T>::type;~~
}}
```

3 If `iterator_traits<I>` does not name an instantiation of the primary template, then `iter_difference_t<I>` is an alias for the type `iterator_traits<I>::difference_type`; otherwise, it is an alias for the type `incrementable_traits<I>::difference_type`.

4 Users may specialize ~~`difference_type`~~`incrementable_traits` on user-defined types.

[Editor's note: Change the name of [iterator.assoc.types.value_type] from "`value_type`" to "`Readable traits`" and change its stable name to [readable.traits].]

28.3.2.2 Readable traits [readable.traits]

1 A `Readable` type has an associated value type that can be accessed with the `value_type_t` alias template.

1 To implement algorithms only in terms of readable types, it is often necessary to determine the value type that corresponds to a particular readable type. Accordingly, it is required that if `R` is the name of a type that satisfies the `Readable` concept (28.3.4.1), the type

```
iter_value_t<R>
```

be defined as the readable type's value type.

2 `iter_value_t` is implemented as if:

```
  template <class T> struct __with_value_type { // exposition only
    using value_type = T;
  };

  template <class> struct ~~value_type~~readable_traits { };

  template <class T>
  struct ~~value_type~~readable_traits<T*>
    : ~~enable_if~~conditional_t<is_object_v<T>~~::value~~,
      __with_value_type< remove_cv_t<T> >, __empty > { };

  template <class I>
    requires is_array_v<I>~~::value~~
  struct ~~value_type~~readable_traits<I>
    : ~~value_type~~readable_traits<decay_t<I>> { };
```

```
template <class I>
struct ~~value_type~~readable_traits<const I>
  : ~~value_type~~readable_traits<decay_t<I>> { };

template <class T>
  requires requires { typename T::value_type; }
struct ~~value_type~~readable_traits<T>
  : ~~enable_if~~conditional_t<is_object_v<typename T::value_type>~~::value~~,
      __with_value_type< typename T::value_type >, __empty > { };

template <class T>
  requires requires { typename T::element_type; }
struct value_type<T>
  : ~~enable_if~~conditional_t<
      is_object_v<typename T::element_type>~~::value~~,
      __with_value_type< remove_cv_t<typename T::element_type> >,
      __empty>
  { };

template <class T> using ~~value_type_t~~iter_value_t = // see below;
  ~~= typename value_type<T>::type;~~
```

3 If `iterator_traits<I>` does not name an instantiation of the primary template, then `iter_value_t<I>` is an alias for the type `iterator_traits<I>::value_type`; otherwise, it is an alias for the type `readable_traits<I>::value_type`.

4 If a type `I` has an associated value type, then ~~value_type<I>::type~~readable_traits<I>::value_type shall name the value type. Otherwise, there shall be no nested type ~~type~~value_type.

5 The ~~value_type~~readable_traits class template may be specialized on user-defined types.

6 When instantiated with a type `I` such that `I::value_type` is valid and denotes a type, ~~value_type<I>::type~~ readable_traits<I>::value_type names that type, unless it is not an object type (6.7) in which case ~~value_type<I>~~readable_traits<I> shall have no nested type `value_type`. [*Note:* Some legacy output iterators define a nested type named `value_type` that is an alias for `void`. These types are not `Readable` and have no associated value types. — *end note*]

7 When instantiated with a type `I` such that `I::element_type` is valid and denotes a type, ~~value_type<I>::type~~ readable_traits<I>::value_type names the type `remove_cv_t<I::element_type>`, unless it is not an object type (6.7) in which case ~~value_type<I>~~readable_traits<I> shall have no nested type `value_type`. [*Note:* Smart pointers like `shared_ptr<int>` are `Readable` and have an associated value type. But a smart pointer like `shared_ptr<void>` is not `Readable` and has no associated value type. — *end note*]

[Editor's note: Remove section "`iterator_category`" [iterator.assoc.types.iterator_category] from P0896R1.]

[Editor's note: From P0896R1, strike [iterator.traits], [iterator.stdtraits], and [std.iterator.tags]. Relocate the working draft's [iterator.traits] from [iterator.primitives] to here and change it as follows:]

28.3.2.3 Iterator traits [iterator.traits]

1 To implement algorithms only in terms of iterators, it is ~~often~~sometimes necessary to determine the ~~value and difference types~~iterator category that corresponds to a particular iterator type. Accordingly, it is required that if ~~Iterator~~I is the type of an iterator, the type~~s~~

```
~~iterator_traits<Iterator>::difference_type~~
~~iterator_traits<Iterator>::value_type~~
```

`iterator_traits<`~~`Iterator`~~`I>::iterator_category`

be defined as the iterator's ~~difference type, value type and~~ iterator category~~, respectively~~. In addition, the types

`iterator_traits<`~~`Iterator`~~`I>::reference`
`iterator_traits<`~~`Iterator`~~`I>::pointer`

shall be defined as the iterator's reference and pointer types~~,~~; that is, for an iterator object `a`, the same type as the type of `*a` and `a->`, respectively. The type `iterator_traits<I>::pointer` shall be `void` for a type `I` that does not support `operator->`. Additionally, i~~I~~n the case of an output iterator, the types

`iterator_traits<`~~`Iterator`~~`I>::difference_type`
`iterator_traits<`~~`Iterator`~~`I>::value_type`
`iterator_traits<`~~`Iterator`~~`I>::reference`
~~`iterator_traits<Iterator>::pointer`~~

may be defined as `void`.

2 The member types of the primary template are computed as defined below. The definition below makes use of several exposition-only concepts equivalent to the following:

```
template <class I>
concept _Cpp98Iterator =
  Copyable<I> && requires (I i) {
    { *i } -> auto &&;
    { ++i } -> Same<I>&;
    { *i++ } -> auto &&;
  };

template <class I>
concept _Cpp98InputIterator =
  _Cpp98Iterator<I> && EqualityComparable<I> && requires (I i) {
    typename common_reference_t<iter_reference_t<I> &&,
                                typename readable_traits<I>::value_type &>;
    typename common_reference_t<decltype(*i++) &&,
                                typename readable_traits<I>::value_type &>;
  } && SignedIntegral<typename incrementable_traits<I>::difference_type>;

template <class I>
concept _Cpp98ForwardIterator =
  _Cpp98InputIterator<I> && Constructible<I> &&
  Same<remove_cvref_t<iter_reference_t<I>>, typename readable_traits<I>::value_type> &&
  requires (I i) {
    { i++ } -> I const &;
    requires Same<iter_reference_t<I>, decltype(*i++)>;
  };

template <class I>
concept _Cpp98BidirectionalIterator =
  _Cpp98ForwardIterator<I> && requires (I i) {
    { --i } -> Same<I>&;
    { i-- } -> I const &;
    requires Same<iter_reference_t<I>, decltype(*i--)>;
  };

template <class I>
```

```
concept _Cpp98RandomAccessIterator =
  _Cpp98BidirectionalIterator<I> && StrictTotallyOrdered<I> &&
  requires (I i, typename incrementable_traits<I>::difference_type n) {
    { i += n } -> Same<I>&;
    { i -= n } -> Same<I>&;
    requires Same<I, decltype(i + n)>;
    requires Same<I, decltype(n + i)>;
    requires Same<I, decltype(i - n)>;
    requires Same<decltype(n), decltype(i - i)>;
    { i[n] } -> iter_reference_t<I>;
  };
```

(2.1) — If ~~`Iterator`~~`I` has valid (17.9.2) member types `difference_type`, `value_type`, ~~`pointer,`~~ `reference`, and `iterator_category`, `iterator_traits<`~~`Iterator`~~`I>` shall have the following as publicly accessible members:

```
using difference_type   = typename ~~Iterator~~I::difference_type;
using value_type        = typename ~~Iterator~~I::value_type;
using pointer           = ~~typename Iterator::pointer~~see below;
using reference         = typename ~~Iterator~~I::reference;
using iterator_category = typename ~~Iterator~~I::iterator_category;
```

If `I` has a valid member type `pointer`, then `iterator_traits<I>::pointer` names that type; otherwise, it is `void`.

(2.2) — Otherwise, if `I` satisfies the exposition-only concept *`_Cpp98InputIterator`*, `iterator_traits<I>` shall have the following as publicly accessible members:

```
using difference_type   = typename incrementable_traits<I>::difference_type;
using value_type        = typename readable_traits<I>::value_type;
using pointer           = see below;
using reference         = see below;
using iterator_category = see below;
```

If `I::pointer` is well-formed and names a type, `pointer` is an alias for that type. Otherwise, if `decltype(declval<I&>().operator->())` is well-formed, then `pointer` names that type. Otherwise, if `iter_reference_t<I>` is an lvalue reference type, `pointer` is `add_pointer_t<iter_reference_t<I>>`. Otherwise, `pointer` is `void`.

If `I::reference` is well-formed and names a type, `reference` names that type. Otherwise, `reference` is `iter_reference_t<I>`.

If `I::iterator_category` is well-formed and names a type, `iterator_category` names that type. Otherwise, if `I` satisfies *`_Cpp98RandomAccessIterator`*, `iterator_category` is `random_access_iterator_tag`. Otherwise, if `I` satisfies *`_Cpp98BidirectionalIterator`*, `iterator_category` is `bidirectional_iterator_tag`. Otherwise, if `I` satisfies *`_Cpp98ForwardIterator`*, `iterator_category` is `forward_iterator_tag`. Otherwise, `iterator_category` is `input_iterator_tag`.

(2.3) — Otherwise, if `I` satisfies the exposition-only concept *`_Cpp98Iterator`*, `iterator_traits<I>` shall have the following as publicly accessible members:

```
using difference_type   = see below;
using value_type        = void;
using pointer           = void;
using reference         = void;
using iterator_category = output_iterator_tag;
```

If `incrementable_traits<I>::difference_type` is well-formed and names a type, then `difference_type` names that type; otherwise, it is `void`.

(2.4) — Otherwise, `iterator_traits<`~~`Iterator`~~`I>` shall have no members by any of the above names.

3 Additionally, user specializations of `iterator_traits` may have a member type `iterator_concept` that is used to opt in or out of conformance to the iterator concepts defined in 28.3.4. If specified, it should be an alias for one of the standard iterator tag types (28.4.1), or an empty, copy- and move-constructible, trivial class type that is publicly and unambiguously derived from one of the standard iterator tag types.

4 ~~It~~`iterator_traits` is specialized for pointers as

```
namespace std {
  template<class T> struct iterator_traits<T*> {
    using difference_type   = ptrdiff_t;
    using value_type        = remove_cv_t<T>;
    using pointer           = T*;
    using reference         = T&;
    using iterator_category = random_access_iterator_tag;
    using iterator_concept = contiguous_iterator_tag;
  };
}
```

5 [*Example:* To implement a generic `reverse` function, a C++ program can do the following:

```
template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last) {
  typename iterator_traits<BidirectionalIterator>::difference_type n =
    distance(first, last);
  --n;
  while(n > 0) {
    typename iterator_traits<BidirectionalIterator>::value_type
     tmp = *first;
    *first++ = *--last;
    *last = tmp;
    n -= 2;
  }
}
```

— *end example*]

28.3.3 Customization points [iterator.custpoints]

[Editor's note: Relocated from [ranges.iterator.custpoints] in P0896R1.]

28.3.3.0.1 `iter_move` [iterator.custpoints.iter_move]

[Editor's note: Relocated from [ranges.iterator.custpoints.iter_move] in P0896R1 and changed as follows:]

1 The name `iter_move` denotes a *customization point object* (20.4.2.1.6). The expression `ranges::iter_move(E)` for some subexpression `E` is expression-equivalent to the following:

(1.1) — ~~`static_cast<decltype(iter_move(E))>(`~~`iter_move(E)`~~`)`~~, if that expression is well-formed when evaluated in a context that does not include `ranges::iter_move` but does include the lookup set produced by argument-dependent lookup (6.4.2).

(1.2) — Otherwise, if the expression `*E` is well-formed:

(1.2.1) — if `*E` is an lvalue, `std::move(*E)`;

(1.2.2) — otherwise, ~~`static_cast<decltype(*E)>(`~~`*E`~~`)`~~.

(1.3) — Otherwise, `ranges::iter_move(E)` is ill-formed.

2 If `ranges::iter_move(E)` does not equal `*E`, the program is ill-formed with no diagnostic required.

28.3.3.0.2 `iter_swap` [iterator.custpoints.iter_swap]

[Editor's note: ...as in P0896R1 with updates for the new associated type alias names.]

[Editor's note: The concept definitions `Readable` through `RandomAccessIterator` from subsection [ranges.iterator.requiremen (and P0944's `ContiguousIterator` concept) all get moved into a new section [iterator.concepts].]

28.3.4 Iterator concepts [iterator.concepts]

28.3.4.1 Concept `Readable` [iterator.concept.readable]

[Editor's note: ... as in P0896R1 with updated associated type aliases.]

28.3.4.2 Concept `Writable` [iterator.concept.writable]

[Editor's note: ... as in P0896R1 with updated associated type aliases.]

28.3.4.3 Concept `WeaklyIncrementable` [iterator.concept.weaklyincrementable]

[Editor's note: ... as in P0896R1 with updated associated type aliases.]

28.3.4.4 Concept `Incrementable` [iterator.concept.incrementable]

[Editor's note: ... as in P0896R1.]

28.3.4.5 Concept `Iterator` [iterator.concept.iterator]

[Editor's note: ... as in P0896R1.]

28.3.4.6 Concept `Sentinel` [iterator.concept.sentinel]

[Editor's note: ... as in P0896R1.]

28.3.4.7 Concept `SizedSentinel` [iterator.concept.sizedsentinel]

[Editor's note: ... as in P0896R1.]

28.3.4.8 Concept `InputIterator` [iterator.concept.input]

[Editor's note: Change as follows:]

1 Let *`ITER_TRAITS`*`(I)` be `I` if `iterator_traits<I>` names an instantiation of the primary template; otherwise, `iterator_traits<I>`.

2 Let *`ITER_CONCEPT`*`(I)` be defined as follows:

(2.1) — If *`ITER_TRAITS`*`(I)::iterator_concept` is valid and names a type, then *`ITER_TRAITS`*`(I)::iterator_concept`.

(2.2) — Otherwise, if *`ITER_TRAITS`*`(I)::iterator_category` is valid and names a type then *`ITER_TRAITS`*`(I)::iterator_category`.

(2.3) — Otherwise, if `iterator_traits<I>` names an instantiation of the primary template, then `random_access_iterator_tag`.

(2.4) — Otherwise, *`ITER_CONCEPT`*`(I)` does not name a type.

3 The `InputIterator` concept is a refinement of `Iterator` (28.3.4.5). It defines requirements for a type whose referenced values can be read (from the requirement for `Readable` (28.3.4.1)) and which can be both pre- and post-incremented. [*Note:* Unlike the input iterator requirements in 28.3.5.2, the `InputIterator` concept does not require equality comparison. — *end note*]

```
template <class I>
concept InputIterator =
  Iterator<I> &&
  Readable<I> &&
  requires { typename ~~iterator_category_t<I>~~ITER_CONCEPT(I); } &&
  DerivedFrom<~~iterator_category_t<I>~~ITER_CONCEPT(I), input_iterator_tag>;
```

28.3.4.9 Concept `OutputIterator` [iterator.concept.output]

[Editor's note: ... as in P0896R1.]

28.3.4.10 Concept `ForwardIterator` [iterator.concept.forward]

[Editor's note: Change the definition of the `ForwardIterator` concept as follows:]

```
template <class I>
concept ForwardIterator =
  InputIterator<I> &&
  DerivedFrom<~~iterator_category_t<I>~~ITER_CONCEPT(I), forward_iterator_tag> &&
  Incrementable<I> &&
  Sentinel<I, I>;
```

28.3.4.11 Concept `BidirectionalIterator` [iterator.concept.bidirectional]

[Editor's note: Change the definition of the `BidirectionalIterator` concept as follows:]

```
template <class I>
concept bool BidirectionalIterator =
  ForwardIterator<I> &&
  DerivedFrom<~~iterator_category_t<I>~~ITER_CONCEPT(I), bidirectional_iterator_tag> &&
  requires(I i) {
    { --i } -> Same<I>&;
    { i-- } -> Same<I>&&;
  };
```

28.3.4.12 Concept `RandomAccessIterator` [iterator.concept.random.access]

[Editor's note: Change the definition of the `RandomAccessIterator` concept as follows:]

```
template <class I>
concept RandomAccessIterator =
  BidirectionalIterator<I> &&
  DerivedFrom<~~iterator_category_t<I>~~ITER_CONCEPT(I), random_access_iterator_tag> &&
  StrictTotallyOrdered<I> &&
  SizedSentinel<I, I> &&
  requires(I i, const I j, const iter_difference~~_type~~_t<I> n) {
    { i += n } -> Same<I>&;
    { j + n }  -> Same<I>&&;
    { n + j }  -> Same<I>&&;
    { i -= n } -> Same<I>&;
    { j - n }  -> Same<I>&&;
    j[n];
    requires Same<decltype(j[n]), iter_reference_t<I>>;
  };
```

28.3.4.13 Concept `ContiguousIterator` [iterator.concept.contiguous]

[Editor's note: Change the definition of the `ContiguousIterator` concept as follows:]

```
template <class I>
concept ContiguousIterator =
  RandomAccessIterator<I> &&
  DerivedFrom<~~iterator_category_t<I>~~ITER_CONCEPT(I), contiguous_iterator_tag> &&
  is_lvalue_reference_v<iter_reference_t<I>> &&
  Same<~~value_type_t~~iter_value_t<I>, remove_cvref_t<~~remove_reference_t<~~iter_reference_t<I>~~>~~>>;
```

28.3.5 C++98 iterator requirements [iterator.cpp98]

1 In the following sections, `a` and `b` denote values of type `X` or `const X`, `difference_type` and `reference` refer to the types `iterator_traits<X>::difference_type` and `iterator_traits<X>::reference`, respectively, `n` denotes a value of `difference_type`, `u`, `tmp`, and `m` denote identifiers, `r` denotes a value of `X&`, `t` denotes a value of value type `T`, `o` denotes a value of some type that is writable to the output iterator. [*Note:* For an iterator type `X` there must be an instantiation of `iterator_traits<X>` (28.3.2.3). — *end note*]

28.3.5.1 Iterator [iterator.iterators]

1 The *Cpp98Iterator* requirements form the basis of the iterator taxonomy; every iterator satisfies the *Cpp98Iterator* requirements. This set of requirements specifies operations for dereferencing and incrementing an iterator. Most algorithms will require additional operations to read (28.3.5.2) or write (28.3.5.3) values, or to provide a richer set of iterator movements (28.3.5.4, 28.3.5.5, 28.3.5.6).

2 A type `X` satisfies the *Cpp98Iterator* requirements if:

(2.1) — `X` satisfies the *Cpp98CopyConstructible*, *Cpp98CopyAssignable*, and *Cpp98Destructible* requirements20.5.3.1 and lvalues of type `X` are swappable?.3.11, and

(2.2) — the expressions in Table 4 are valid and have the indicated semantics.

Table 4 — Iterator requirements

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`*r`	unspecified		*Requires:*`r` is dereferenceable.
`++r`	`X&`		

28.3.5.2 Input iterators [input.iterators]

1 A class or pointer type `X` satisfies the requirements of an input iterator for the value type `T` if `X` satisfies the *Cpp98Iterator* (28.3.5.1) and *Cpp98EqualityComparable* (20) requirements and the expressions in Table 5 are valid and have the indicated semantics. [*Note:* Since input iterators (and refinements thereof) are equality comparable, an input iterator can serve as a sentinel for another input iterator of the same type. — *end note*]

2 In Table 5, the term *the domain of* `==` is used in the ordinary mathematical sense to denote the set of values over which `==` is (required to be) defined. This set can change over time. Each algorithm places additional requirements on the domain of `==` for the iterator values it uses. These requirements can be inferred from the uses that algorithm makes of `==` and `!=`. [*Example:* The call `find(a,b,x)` is defined only if the value of `a` has the property *p* defined as follows: `b` has property *p* and a value `i` has property *p* if (`*i==x`) or if (`*i!=x` and `++i` has property *p*). — *end example*]

Table 5 — Input iterator requirements (in addition to Iterator)

Expression	**Return type**	**Operational semantics**	**Assertion/note pre-/post-condition**
`a != b`	contextually convertible to `bool`	`!(a == b)`	*Requires:*`(a,b)` is in the domain of `==`.
`*a`	`reference`, convertible to `T`		*Requires:*`a` is dereferenceable. The expression `(void)*a, *a` is equivalent to `*a`. If `a == b` and `(a,b)` is in the domain of `==` then `*a` is equivalent to `*b`.
`a->m`		`(*a).m`	*Requires:*`a` is dereferenceable.
`++r`	`X&`		*Requires:*`r` is dereferenceable. *Postconditions:*`r` is dereferenceable or `r` is past-the-end; any copies of the previous value of `r` are no longer required either to be dereferenceable or to be in the domain of `==`.
`(void)r++`			equivalent to `(void)++r`

`*r++`	convertible to `T`	`{ T tmp = *r;`
		`++r;`
		`return tmp; }`	

3 [*Note:* For input iterators, `a == b` does not imply `++a == ++b`. (Equality does not guarantee the substitution property or referential transparency.) Algorithms on input iterators should never attempt to pass through the same iterator twice. They should be *single pass* algorithms. Value type `T` is not required to be a *Cpp98CopyAssignable* type (). These algorithms can be used with istreams as the source of the input data through the `istream_iterator` class template. — *end note*]

28.3.5.3 Output iterators [output.iterators]

1 A class or pointer type `X` satisfies the requirements of an output iterator if `X` satisfies the *Cpp98Iterator* requirements (28.3.5.1) and the expressions in Table 6 are valid and have the indicated semantics.

Table 6 — Output iterator requirements (in addition to Iterator)

Expression	**Return type**	**Operational semantics**	**Assertion/note pre-/post-condition**
`*r = o`	result is not used		*Remarks:* After this operation `r` is not required to be dereferenceable. *Postconditions:*`r` is incrementable.
`++r`	`X&`		`&r == &++r`. *Remarks:* After this operation `r` is not required to be dereferenceable. *Postconditions:*`r` is incrementable.

`r++`	convertible to `const X&`	`{ X tmp = r;`
		`++r;`
		`return tmp; }`
			Remarks: After this operation `r` is not required to be dereferenceable.
			Postconditions:`r` is incrementable.

`*r++ = o`	result is not used		*Remarks:* After this operation `r` is not required to be dereferenceable. *Postconditions:*`r` is incrementable.

2 [*Note:* The only valid use of an `operator*` is on the left side of the assignment statement. *Assignment through the same value of the iterator happens only once.* Algorithms on output iterators should never attempt to pass through the same iterator twice. They should be *single pass* algorithms. Equality and inequality might not be defined. Algorithms that take output iterators can be used with ostreams as the destination for placing data through the `ostream_iterator` class as well as with insert iterators and insert pointers. — *end note*]

28.3.5.4 Forward iterators [forward.iterators]

1 A class or pointer type `X` satisfies the requirements of a forward iterator if

(1.1) — `X` satisfies the requirements of an input iterator (28.3.5.2),

(1.2) — `X` satisfies the *Cpp98DefaultConstructible* requirements20.5.3.1,

(1.3) — if `X` is a mutable iterator, `reference` is a reference to `T`; if `X` is a constant iterator, `reference` is a reference to `const T`,

(1.4) — the expressions in Table 7 are valid and have the indicated semantics, and

(1.5) — objects of type `X` offer the multi-pass guarantee, described below.

2 The domain of `==` for forward iterators is that of iterators over the same underlying sequence. However, value-initialized iterators may be compared and shall compare equal to other value-initialized iterators of the same type. [*Note:* Value-initialized iterators behave as if they refer past the end of the same empty sequence. — *end note*]

3 Two dereferenceable iterators `a` and `b` of type `X` offer the *multi-pass guarantee* if:

(3.1) — `a == b` implies `++a == ++b` and

(3.2) — `X` is a pointer type or the expression `(void)++X(a), *a` is equivalent to the expression `*a`.

4 [*Note:* The requirement that `a == b` implies `++a == ++b` (which is not true for input and output iterators) and the removal of the restrictions on the number of the assignments through a mutable iterator (which applies to output iterators) allows the use of multi-pass one-directional algorithms with forward iterators. — *end note*]

Table 7 — Forward iterator requirements (in addition to input iterator)

Expression	**Return type**	**Operational semantics**	**Assertion/note pre-/post-condition**

`r++`	convertible to `const X&`	`{ X tmp = r;`
		`++r;`
		`return tmp; }`	

`*r++`	`reference`		

5 If `a` and `b` are equal, then either `a` and `b` are both dereferenceable or else neither is dereferenceable.

6 If `a` and `b` are both dereferenceable, then `a == b` if and only if `*a` and `*b` are bound to the same object.

28.3.5.5 Bidirectional iterators [bidirectional.iterators]

1 A class or pointer type `X` satisfies the requirements of a bidirectional iterator if, in addition to satisfying the requirements for forward iterators, the following expressions are valid as shown in Table 8.

Table 8 — Bidirectional iterator requirements (in addition to forward iterator)

Expression	**Return type**	**Operational semantics**	**Assertion/note pre-/post-condition**
`--r`	`X&`		*Requires:* there exists `s` such that `r == ++s`. *Postconditions:*`r` is dereferenceable. `--(++r) == r`. `--r == --s` implies `r == s`. `&r == &--r`.

`r--`	convertible to `const X&`	`{ X tmp = r;`
		`--r;`
		`return tmp; }`	

`*r--`	`reference`		

2 [*Note:* Bidirectional iterators allow algorithms to move iterators backward as well as forward. — *end note*]

28.3.5.6 Random access iterators [random.access.iterators]

1 A class or pointer type `X` satisfies the requirements of a random access iterator if, in addition to satisfying the requirements for bidirectional iterators, the following expressions are valid as shown in Table 9.

Table 9 — Random access iterator requirements (in addition to bidirectional iterator)

Expression	**Return type**	**Operational semantics**	**Assertion/note pre-/post-condition**

`r += n`	`X&`	`{ difference_type m = n;`
		`if (m >= 0)`
		`while (m--)`
		`++r;`
		`else`
		`while (m++)`
		`--r;`
		`return r; }`	

`a + n` `n + a`	`X`	`{ X tmp = a;` `return tmp += n; }`	`a + n == n + a`.
`r -= n`	`X&`	`return r += -n;`	*Requires:* the absolute value of `n` is in the range of representable values of `difference_type`.
`a - n`	`X`	`{ X tmp = a;` `return tmp -= n; }`	
`b - a`	`difference_type`	`return n`	*Requires:* there exists a value `n` of type `difference_type` such that `a + n == b`. `b == a + (b - a)`.
`a[n]`	convertible to `reference`	`*(a + n)`	
`a < b`	contextually convertible to `bool`	`b - a > 0`	`<` is a total ordering relation
`a > b`	contextually convertible to `bool`	`b < a`	`>` is a total ordering relation opposite to `<`.
`a >= b`	contextually convertible to `bool`	`!(a < b)`	
`a <= b`	contextually convertible to `bool`.	`!(a > b)`	

28.3.6 Indirect callable requirements [indirectcallable]

28.3.6.1 General [indirectcallable.general]

1 There are several concepts that group requirements of algorithms that take callable objects (23.14.3) as arguments.

28.3.6.2 Indirect callables [indirectcallable.indirectinvocable]

1 The indirect callable concepts are used to constrain those algorithms that accept callable objects (23.14.2) as arguments.

```
namespace std {
  template <class F, class I>
  concept IndirectUnaryInvocable = // ... as in P0896R1 with new associated type names

  template <class F, class I>
  concept IndirectRegularUnaryInvocable = // ... as in P0896R1 with new associated type names

  template <class F, class I>
  concept IndirectUnaryPredicate = // ... as in P0896R1 with new associated type names

  template <class F, class I1, class I2 = I1>
  concept IndirectRelation = // ... as in P0896R1 with new associated type names

  template <class F, class I1, class I2 = I1>
  concept IndirectStrictWeakOrder = // ... as in P0896R1 with new associated type names

  template <class F, class... Is>
    requires (Readable<Is> &&...) && Invocable<F, iter_reference_t<Is>...>
  struct indirect_result<F, Is...> :
    invoke_result<F, iter_reference_t<Is>...> { };
}
```

28.3.6.3 Class template `projected` [projected]

[Editor's note: Change "Class template `projected`" as follows:]

1 The `projected` class template is intended for use when specifying the constraints of algorithms that accept callable objects and projections (). It bundles a `Readable` type `I` and a function `Proj` into a new `Readable` type whose `reference` type is the result of applying `Proj` to the `iter_reference_t` of `I`.

```
namespace std {
  template <Readable I, IndirectRegularUnaryInvocable<I> Proj>
  struct projected {
    using value_type = remove_cvref_t<~~remove_reference_t<~~indirect_result_t<Proj&, I>~~>~~>;
    indirect_result_t<Proj&, I> operator*() const;
  };

  template <WeaklyIncrementable I, class Proj>
  struct ~~difference_type~~incrementable_traits<projected<I, Proj>> {
    ~~using type = difference_type_t<I>;~~
    using difference_type = iter_difference_t<I>;
  };
}
```

2 [*Note:* `projected` is only used to ease constraints specification. Its member function need not be defined. — *end note*]

28.3.7 Common algorithm requirements [commonalgoreq]

28.3.7.1 General [commonalgoreq.general]

1 There are several additional iterator concepts that are commonly applied to families of algorithms. These group together iterator requirements of algorithm families. There are three relational concepts that specify how element values are transferred between `Readable` and `Writable` types: `IndirectlyMovable`, `IndirectlyCopyable`, and `IndirectlySwappable`. There are three relational concepts for rearrangements: `Permutable`, `Mergeable`, and `Sortable`. There is one relational concept for comparing values from different sequences: `IndirectlyComparable`.

2 [*Note:* The ranges::`equal_to<>` and ranges::`less<>` (23.14.8) function object types used in the concepts below impose ~~additional~~ constraints on their arguments ~~beyond~~in addition to those that appear explicitly in the concepts' bodies. ranges::`equal_to<>` requires its arguments satisfy `EqualityComparableWith` (?.4.3), and ranges::`less<>` requires its arguments satisfy `StrictTotallyOrderedWith` (?.4.4). — *end note*]

28.3.7.2 Concept `IndirectlyMovable` [commonalgoreq.indirectlymovable]

1 The `IndirectlyMovable` concept specifies the relationship between a `Readable` type and a `Writable` type between which values may be moved.

```
template <class In, class Out>
concept IndirectlyMovable =
  Readable<In> &&
  Writable<Out, rvalue_reference_t<In>>;
```

2 The `IndirectlyMovableStorable` concept augments `IndirectlyMovable` with additional requirements enabling the transfer to be performed through an intermediate object of the `Readable` type's value type.

```
template <class In, class Out>
concept IndirectlyMovableStorable =
  IndirectlyMovable<In, Out> &&
  Writable<Out, value_type_t<In>> &&
  Movable<value_type_t<In>> &&
  Constructible<value_type_t<In>, rvalue_reference_t<In>> &&
  Assignable<value_type_t<In>&, rvalue_reference_t<In>>;
```

28.3.7.3 Concept `IndirectlyCopyable` [commonalgoreq.indirectlycopyable]

1 The `IndirectlyCopyable` concept specifies the relationship between a `Readable` type and a `Writable` type between which values may be copied.

```
template <class In, class Out>
concept IndirectlyCopyable =
  Readable<In> &&
  Writable<Out, reference_t<In>>;
```

2 The `IndirectlyCopyableStorable` concept augments `IndirectlyCopyable` with additional requirements enabling the transfer to be performed through an intermediate object of the `Readable` type's value type. It also requires the capability to make copies of values.

```
template <class In, class Out>
concept IndirectlyCopyableStorable =
  IndirectlyCopyable<In, Out> &&
  Writable<Out, const value_type_t<In>&> &&
  Copyable<value_type_t<In>> &&
  Constructible<value_type_t<In>, reference_t<In>> &&
  Assignable<value_type_t<In>&, reference_t<In>>;
```

28.3.7.4 Concept `IndirectlySwappable` [commonalgoreq.indirectlyswappable]

1 The `IndirectlySwappable` concept specifies a swappable relationship between the values referenced by two `Readable` types.

```
template <class I1, class I2 = I1>
concept IndirectlySwappable =
  Readable<I1> && Readable<I2> &&
  requires(I1&& i1, I2&& i2) {
    ranges::iter_swap(std::forward<I1>(i1), std::forward<I2>(i2));
```

```
    ranges::iter_swap(std::forward<I2>(i2), std::forward<I1>(i1));
    ranges::iter_swap(std::forward<I1>(i1), std::forward<I1>(i1));
    ranges::iter_swap(std::forward<I2>(i2), std::forward<I2>(i2));
  };
```

2 Given an object `i1` of type `I1` and an object `i2` of type `I2`, `IndirectlySwappable<I1, I2>` is satisfied if after `ranges::iter_swap(i1, i2)`, the value of `*i1` is equal to the value of `*i2` before the call, and *vice versa*.

28.3.7.5 Concept `IndirectlyComparable` [commonalgoreq.indirectlycomparable]

1 The `IndirectlyComparable` concept specifies the common requirements of algorithms that compare values from two different sequences.

```
template <class I1, class I2, class R = ranges::equal_to<>, class P1 = identity,
  class P2 = identity>
concept IndirectlyComparable =
  IndirectRelation<R, projected<I1, P1>, projected<I2, P2>>;
```

28.3.7.6 Concept `Permutable` [commonalgoreq.permutable]

1 The `Permutable` concept specifies the common requirements of algorithms that reorder elements in place by moving or swapping them.

```
template <class I>
concept Permutable =
  ForwardIterator<I> &&
  IndirectlyMovableStorable<I, I> &&
  IndirectlySwappable<I, I>;
```

28.3.7.7 Concept `Mergeable` [commonalgoreq.mergeable]

1 The `Mergeable` concept specifies the requirements of algorithms that merge sorted sequences into an output sequence by copying elements.

```
template <class I1, class I2, class Out,
    class R = ranges::less<>, class P1 = identity, class P2 = identity>
concept Mergeable =
  InputIterator<I1> &&
  InputIterator<I2> &&
  WeaklyIncrementable<Out> &&
  IndirectlyCopyable<I1, Out> &&
  IndirectlyCopyable<I2, Out> &&
  IndirectStrictWeakOrder<R, projected<I1, P1>, projected<I2, P2>>;
```

28.3.7.8 Concept `Sortable` [commonalgoreq.sortable]

1 The `Sortable` concept specifies the common requirements of algorithms that permute sequences into ordered sequences (e.g., `sort`).

```
template <class I, class R = ranges::less<>, class P = identity>
concept Sortable =
  Permutable<I> &&
  IndirectStrictWeakOrder<R, projected<I, P>>;
```

28.4 Iterator primitives [iterator.primitives]

1 To simplify the task of defining iterators, the library provides several classes and functions:

28.4.1 Standard iterator tags [std.iterator.tags]

[Editor's note: Amend the section "Standard iterator tags" as follows:]

1 It is often desirable for a function template specialization to find out what is the most specific category of its iterator argument, so that the function can select the most efficient algorithm at compile time. To facilitate this, the library introduces *category tag* classes which are used as compile time tags for algorithm selection. They are: `input_iterator_tag`, `output_iterator_tag`, `forward_iterator_tag`, `bidirectional_iterator_tag` ~~and~~, `random_access_iterator_tag` and contiguous_iterator_tag. For every iterator of type ~~`Iterator`~~I, `iterator_traits<`~~`Iterator`~~I`>::iterator_category` shall be defined to be the most specific category tag that describes the iterator's behavior. Additionally and optionally, `iterator_traits<`~~Iterator~~I>::iterator_category may be used to opt in or out of conformance to the iterator concepts defined in section 28.3.4.

```
namespace std {
  struct input_iterator_tag { };
  struct output_iterator_tag { };
  struct forward_iterator_tag: public input_iterator_tag { };
  struct bidirectional_iterator_tag: public forward_iterator_tag { };
  struct random_access_iterator_tag: public bidirectional_iterator_tag { };
  struct contiguous_iterator_tag:  public random_access_iterator_tag { };
}
```

2 [*Example:* For a program-defined iterator `BinaryTreeIterator`, it could be included into the bidirectional iterator category by specializing the `iterator_traits` template:

```
template<class T> struct iterator_traits<BinaryTreeIterator<T>> {
  using iterator_category = bidirectional_iterator_tag;
  using difference_type   = ptrdiff_t;
  using value_type        = T;
  using pointer           = T*;
  using reference         = T&;
};
```

— *end example*]

3 [*Example:* If `evolve()` is well-defined for bidirectional iterators, but can be implemented more efficiently for random access iterators, then the implementation is as follows:

```
template<class BidirectionalIterator>
inline void
evolve(BidirectionalIterator first, BidirectionalIterator last) {
  evolve(first, last,
    typename iterator_traits<BidirectionalIterator>::iterator_category());
}

template<class BidirectionalIterator>
void evolve(BidirectionalIterator first, BidirectionalIterator last,
  bidirectional_iterator_tag) {
  // more generic, but less efficient algorithm
}

template<class RandomAccessIterator>
void evolve(RandomAccessIterator first, RandomAccessIterator last,
  random_access_iterator_tag) {
  // more efficient, but less generic algorithm
}
```

— *end example*]

28.4.2 Iterator operations [iterator.operations]

1 Since only random access iterators provide `+` and `-` operators, the library provides two function templates `advance` and `distance`. These function templates use `+` and `-` for random access iterators (and are, therefore, constant time for them); for input, forward and bidirectional iterators they use `++` to provide linear time implementations.

```
template<class InputIterator, class Distance>
  constexpr void advance(InputIterator& i, Distance n);
```

2 *Requires:* `n` shall be negative only for bidirectional and random access iterators.

3 *Effects:* Increments (or decrements for negative `n`) iterator reference `i` by `n`.

```
template<class InputIterator>
  constexpr typename iterator_traits<InputIterator>::difference_type
    distance(InputIterator first, InputIterator last);
```

4 *Effects:* If `InputIterator` meets the requirements of random access iterator, returns `(last - first)`; otherwise, returns the number of increments needed to get from `first` to `last`.

5 *Requires:* If `InputIterator` meets the requirements of random access iterator, `last` shall be reachable from `first` or `first` shall be reachable from `last`; otherwise, `last` shall be reachable from `first`.

```
template<class InputIterator>
  constexpr InputIterator next(InputIterator x,
    typename iterator_traits<InputIterator>::difference_type n = 1);
```

6 *Effects:* Equivalent to: `advance(x, n); return x;`

```
template<class BidirectionalIterator>
  constexpr BidirectionalIterator prev(BidirectionalIterator x,
    typename iterator_traits<BidirectionalIterator>::difference_type n = 1);
```

7 *Effects:* Equivalent to: `advance(x, -n); return x;`

28.4.3 Range iterator operations [ranges.iterator.operations]

[Editor's note: Copied [ranges.iterator.operations] in P0896R1 and modified as follows:]

1 Since only types that satisfy `RandomAccessIterator` provide the `+` operator, and types that satisfy `SizedSentinel` provide the `-` operator, the library provides function templates `advance`, `distance`, `next`, and `prev`. These function templates use `+` and `-` for random access iterators and ranges that satisfy `SizedSentinel` (and are, therefore, constant time for them); for output, input, forward and bidirectional iterators they use `++` to provide linear time implementations.

2 The function templates defined in this subclause are not found by argument-dependent name lookup (6.4.2). When found by unqualified (6.4.1) name lookup for the *postfix-expression* in a function call (), they inhibit argument-dependent name lookup.

[*Example:*

```
void foo() {
    using namespace std::ranges;
    std::vector<int> vec{1,2,3};
    distance(begin(vec), end(vec)); // #1
}
```

The function call expression at `#1` invokes `std::ranges::distance`, not `std::distance`, despite that (a) the iterator type returned from `begin(vec)` and `end(vec)` may be associated with namespace `std` and (b) `std::distance` is more specialized (17.6.6.2) than `std::ranges::distance` since the former requires its first two parameters to have the same type. — *end example*]

28.4.3.1 `ranges::advance` [ranges.iterator.operations.advance]

`template <Iterator I>`
`constexpr void advance(I& i, iter_difference`~~`_type`~~`_t<I> n);`

1 *Requires:* `n` shall be negative only for bidirectional iterators.

2 *Effects:* For random access iterators, equivalent to `i += n`. Otherwise, increments (or decrements for negative `n`) iterator `i` by `n`.

```
template <Iterator I, Sentinel<I> S>
  constexpr void advance(I& i, S bound);
```

3 *Requires:* If `Assignable<I&, S>` is not satisfied, `[i,bound)` shall denote a range.

4 *Effects:*

- (4.1) If `Assignable<I&, S>` is satisfied, equivalent to `i = std::move(bound)`.
- (4.2) Otherwise, if `SizedSentinel<S, I>` is satisfied, equivalent to `advance(i, bound - i)`.
- (4.3) Otherwise, increments `i` until `i == bound`.

`template <Iterator I, Sentinel<I> S>`
`constexpr iter_difference`~~`_type`~~`_t<I> advance(I& i, iter_difference`~~`_type`~~`_t<I> n, S bound);`

5 *Requires:* If `n > 0`, `[i,bound)` shall denote a range. If `n == 0`, `[i,bound)` or `[bound,i)` shall denote a range. If `n < 0`, `[bound,i)` shall denote a range and `(BidirectionalIterator<I> && Same<I, S>)` shall be satisfied.

6 *Effects:*

- (6.1) If `SizedSentinel<S, I>` is satisfied:
  - (6.1.1) If |`n`| >= |`bound - i`|, equivalent to `advance(i, bound)`.
  - (6.1.2) Otherwise, equivalent to `advance(i, n)`.
- (6.2) Otherwise, increments (or decrements for negative `n`) iterator `i` either `n` times or until `i == bound`, whichever comes first.

7 *Returns:* `n` - M, where M is the distance from the starting position of `i` to the ending position.

28.4.3.2 `ranges::distance` [ranges.iterator.operations.distance]

`template <Iterator I, Sentinel<I> S>`
`constexpr iter_difference`~~`_type`~~`_t<I> distance(I first, S last);`

1 *Requires:* `[first,last)` shall denote a range, or `(Same<S, I> && SizedSentinel<S, I>)` shall be satisfied and `[last,first)` shall denote a range.

2 *Effects:* If `SizedSentinel<S, I>` is satisfied, returns `(last - first)`; otherwise, returns the number of increments needed to get from `first` to `last`.

`template <Range R>`
`constexpr iter_difference`~~`_type`~~`_t<iterator_t<R>> distance(R&& r);`

3 *Effects:* If `SizedRange<R>` is satisfied, equivalent to:

```
return ranges::size(r); // 29.6.1
```

Otherwise, equivalent to:

```
return distance(ranges::begin(r), ranges::end(r)); // 29.4
```

28.4.3.3 `ranges::next` [ranges.iterator.operations.next]

```
template <Iterator I>
  constexpr I next(I x);
```

1 *Effects:* Equivalent to: `++x; return x;`

```
template <Iterator I>
  constexpr I next(I x, iter_difference~~_type~~_t<I> n);
```

2 *Effects:* Equivalent to: `advance(x, n); return x;`

```
template <Iterator I, Sentinel<I> S>
  constexpr I next(I x, S bound);
```

3 *Effects:* Equivalent to: `advance(x, bound); return x;`

```
template <Iterator I, Sentinel<I> S>
  constexpr I next(I x, iter_difference~~_type~~_t<I> n, S bound);
```

4 *Effects:* Equivalent to: `advance(x, n, bound); return x;`

28.4.3.4 `ranges::prev` [ranges.iterator.operations.prev]

```
template <BidirectionalIterator I>
  constexpr I prev(I x);
```

1 *Effects:* Equivalent to: `--x; return x;`

```
template <BidirectionalIterator I>
  constexpr I prev(I x, iter_difference~~_type~~_t<I> n);
```

2 *Effects:* Equivalent to: `advance(x, -n); return x;`

```
template <BidirectionalIterator I>
  constexpr I prev(I x, iter_difference~~_type~~_t<I> n, I bound);
```

3 *Effects:* Equivalent to: `advance(x, -n, bound); return x;`

28.5 Iterator adaptors [predef.iterators]

28.5.1 Reverse iterators [reverse.iterators]

1 Class template `reverse_iterator` is an iterator adaptor that iterates from the end of the sequence defined by its underlying iterator to the beginning of that sequence. ~~The fundamental relation between a reverse iterator and its corresponding iterator `i` is established by the identity: `&*(reverse_iterator(i)) == &*(i - 1)`.~~

28.5.1.1 Class template `reverse_iterator` [reverse.iterator]

[Editor's note: Change the synopsis of `reverse_iterator` as follows:]

```
namespace std {
  template<class Iterator>
  class reverse_iterator {
  public:
    using iterator_type     = Iterator;
```

```
  ~~using iterator_category = typename iterator_traits<Iterator>::iterator_category;~~
  ~~using value_type = typename iterator_traits<Iterator>::value_type;~~
  ~~using difference_type = typename iterator_traits<Iterator>::difference_type;~~
  ~~using pointer = typename iterator_traits<Iterator>::pointer;~~
  ~~using reference = typename iterator_traits<Iterator>::reference;~~
  using iterator_category = see below;
  using iterator_concept = see below;
  using value_type = iter_value_t<Iterator>;
  using difference_type = iter_difference_t<Iterator>;
  using pointer = Iterator;
  using reference = iter_reference_t<Iterator>;

  constexpr reverse_iterator();
  constexpr explicit reverse_iterator(Iterator x);
  template<class U> constexpr reverse_iterator(const reverse_iterator<U>& u);
  template<class U> constexpr reverse_iterator& operator=(const reverse_iterator<U>& u);

  constexpr Iterator base() const;      // explicit
  constexpr reference operator*() const;
  constexpr pointer   operator->() const;

  constexpr reverse_iterator& operator++();
  constexpr reverse_iterator  operator++(int);
  constexpr reverse_iterator& operator--();
  constexpr reverse_iterator  operator--(int);

  constexpr reverse_iterator  operator+ (difference_type n) const;
  constexpr reverse_iterator& operator+=(difference_type n);
  constexpr reverse_iterator  operator- (difference_type n) const;
  constexpr reverse_iterator& operator-=(difference_type n);
  constexpr unspecified  operator[](difference_type n) const;

  friend constexpr iter_rvalue_reference_t<Iterator> iter_move(const reverse_iterator& i)
    noexcept(see below);
  template <IndirectlySwappable<Iterator> Iterator2>
    friend constexpr void iter_swap(const reverse_iterator& x,
                                    const reverse_iterator<Iterator2>& y)
      noexcept(see below);

protected:
  Iterator current;
};

template<class Iterator1, class Iterator2>
  constexpr bool operator==(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
template<class Iterator1, class Iterator2>
  constexpr bool operator<(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
template<class Iterator1, class Iterator2>
  constexpr bool operator!=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

```
  template<class Iterator1, class Iterator2>
    constexpr bool operator>(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y);
  template<class Iterator1, class Iterator2>
    constexpr bool operator>=(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y);
  template<class Iterator1, class Iterator2>
    constexpr bool operator<=(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y);
  template<class Iterator1, class Iterator2>
    constexpr auto operator-(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y) -> decltype(y.base() - x.base());
  template<class Iterator>
    constexpr reverse_iterator<Iterator> operator+(
      typename reverse_iterator<Iterator>::difference_type n,
      const reverse_iterator<Iterator>& x);

  template<class Iterator>
    constexpr reverse_iterator<Iterator> make_reverse_iterator(Iterator i);

  template <class Iterator1, class Iterator2>
    requires !SizedSentinel<Iterator1, Iterator2>
  constexpr bool disable_sized_sentinel<reverse_iterator<Iterator1>,
                                        reverse_iterator<Iterator2>> = true;
}
```

1 The member type `iterator_category` is defined as follows:

- (1.1) — If `iterator_traits<Iterator>::iterator_category` is `contiguous_iterator_tag`, then `random_access_iterator_tag`.
- (1.2) — Otherwise, `iterator_traits<Iterator>::iterator_category`.

2 The member type `iterator_concept` is defined as follows:

- (2.1) — If `Iterator` satisfies `RandomAccessIterator`, then `random_access_iterator_tag`.
- (2.2) — Otherwise, `bidirectional_iterator_tag`.

28.5.1.5 `reverse_iterator` element access [reverse.iterator.elem]

[Editor's note: Change section "`reverse_iterator` element access" /p2 as follows:]

`constexpr pointer operator->() const;`

2 *Returns:* ~~`addressof(operator*())`~~`prev(current)`.

[Editor's note: After [reverse.iter.nav], add a new subsection for `reverse_iterator` friend functions.]

28.5.1.7 `reverse_iterator` friend functions [reverse.iter.friends]

```
friend constexpr iter_rvalue_reference_t<Iterator> iter_move(const reverse_iterator& i)
  noexcept(see below);
```

1 *Effects:* Equivalent to: `return ranges::iter_move(prev(i.current));`

2 *Remarks:* The expression in `noexcept` is equivalent to:

```
noexcept(ranges::iter_move(declval<Iterator&>())) && noexcept(--declval<Iterator&>()) &&
  is_nothrow_copy_constructible<Iterator>::value
```

```
template <IndirectlySwappable<Iterator> Iterator2>
  friend constexpr void iter_swap(const reverse_iterator& x, const reverse_iterator<Iterator2>& y)
    noexcept(see below);
```

3 *Effects:* Equivalent to `ranges::iter_swap(prev(x.current), prev(y.current))`.

4 *Remarks:* The expression in `noexcept` is equivalent to:

```
noexcept(ranges::iter_swap(declval<Iterator>(), declval<Iterator>())) &&
  noexcept(--declval<Iterator&>())
```

28.5.1.8 `reverse_iterator` comparisons [reverse.iter.cmp]

1 The functions in this subsection only participate in overload resolution if the expression in their *Returns:* clause is well-formed.

```
template<class Iterator1, class Iterator2>
  constexpr bool operator==(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

2 *Returns:* `x.current == y.current`.

```
template<class Iterator1, class Iterator2>
  constexpr bool operator<(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

3 *Returns:* `x.current > y.current`.

```
template<class Iterator1, class Iterator2>
  constexpr bool operator!=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

4 *Returns:* `x.current != y.current`.

```
template<class Iterator1, class Iterator2>
  constexpr bool operator>(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

5 *Returns:* `x.current < y.current`.

```
template<class Iterator1, class Iterator2>
  constexpr bool operator>=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

6 *Returns:* `x.current <= y.current`.

```
template<class Iterator1, class Iterator2>
  constexpr bool operator<=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

7 *Returns:* `x.current >= y.current`.

28.5.2 Insert iterators [insert.iterators]

28.5.2.1 Class template `back_insert_iterator` [back.insert.iterator]

[Editor's note: Change the class synopsis of `back_insert_iterator` as follows. The addition of the default constructor is so that `back_insert_iterator` satisfies the `Iterator` concept.]

```
namespace std {
  template<class Container>
  class back_insert_iterator {
  protected:
    Container* container = nullptr; // exposition only

  public:
    using iterator_category = output_iterator_tag;
    using value_type        = void;
    using difference_type   = ~~void~~ptrdiff_t;
    using pointer           = void;
    using reference         = void;
    using container_type    = Container;

    constexpr back_insert_iterator() noexcept = default;
    explicit back_insert_iterator(Container& x);
    back_insert_iterator& operator=(const typename Container::value_type& value);
    back_insert_iterator& operator=(typename Container::value_type&& value);

    back_insert_iterator& operator*();
    back_insert_iterator& operator++();
    back_insert_iterator  operator++(int);
  };

  template<class Container>
    back_insert_iterator<Container> back_inserter(Container& x);
}
```

28.5.2.2 Class template `front_insert_iterator` [front.insert.iterator]

[Editor's note: Change the class synopsis of `front_insert_iterator` as follows. The addition of the default constructor is so that `front_insert_iterator` satisfies the `Iterator` concept.]

```
namespace std {
  template<class Container>
  class front_insert_iterator {
  protected:
    Container* container = nullptr; // exposition only

  public:
    using iterator_category = output_iterator_tag;
    using value_type        = void;
```

```
    using difference_type   = ~~void~~ptrdiff_t;
    using pointer           = void;
    using reference         = void;
    using container_type    = Container;

    constexpr front_insert_iterator() noexcept = default;
    explicit front_insert_iterator(Container& x);
    front_insert_iterator& operator=(const typename Container::value_type& value);
    front_insert_iterator& operator=(typename Container::value_type&& value);

    front_insert_iterator& operator*();
    front_insert_iterator& operator++();
    front_insert_iterator  operator++(int);
  };

  template<class Container>
    front_insert_iterator<Container> front_inserter(Container& x);
}
```

28.5.2.3 Class template `insert_iterator` [insert.iterator]

[Editor's note: Change the class synopsis of `insert_iterator` as follows:]

```
namespace std {
  template<class Container>
  class insert_iterator {
  protected:
    Container* container = nullptr; // exposition only
    ~~typename Container::iterator~~iterator_t<Container> iter {}; // exposition only

  public:
    using iterator_category = output_iterator_tag;
    using value_type        = void;
    using difference_type   = ~~void~~ptrdiff_t;
    using pointer           = void;
    using reference         = void;
    using container_type    = Container;

    insert_iterator() = default;
    insert_iterator(Container& x, ~~typename Container::iterator~~iterator_t<Container> i);
    insert_iterator& operator=(const typename Container::value_type& value);
    insert_iterator& operator=(typename Container::value_type&& value);

    insert_iterator& operator*();
    insert_iterator& operator++();
    insert_iterator& operator++(int);
  };

  template<class Container>
    insert_iterator<Container> inserter(Container& x, ~~typename Container::iterator~~iterator_t<Container> i);
}
```

28.5.2.3.1 `insert_iterator` operations [insert.iter.ops]

`insert_iterator(Container& x, ~~typename Container::iterator~~iterator_t<Container> i);`

1 *Effects:* Initializes `container` with `addressof(x)` and `iter` with `i`.

```
insert_iterator& operator=(const typename Container::value_type& value);
```

2 *Effects:* As if by:

```
iter = container->insert(iter, value);
++iter;
```

3 *Returns:* `*this`.

```
insert_iterator& operator=(typename Container::value_type&& value);
```

4 *Effects:* As if by:

```
iter = container->insert(iter, std::move(value));
++iter;
```

5 *Returns:* `*this`.

```
insert_iterator& operator*();
```

6 *Returns:* `*this`.

```
insert_iterator& operator++();
insert_iterator& operator++(int);
```

7 *Returns:* `*this`.

28.5.2.3.2 `inserter` [inserter]

`template<class Container>`
`  insert_iterator<Container> inserter(Container& x, `~~`typename Container::iterator`~~`iterator_t<Container> i);`

1 *Returns:* `insert_iterator<Container>(x, i)`.

28.5.3 Move iterators and sentinels [move.iterators]

[Editor's note: Note that this section in the working draft has changed name from "Move iterators" to "Move iterators and sentinels".]

1 Class template `move_iterator` is an iterator adaptor with the same behavior as the underlying iterator except that its indirection operator implicitly converts the value returned by the underlying iterator's indirection operator to an rvalue. Some generic algorithms can be called with move iterators to replace copying with moving.

2 [*Example:*

```
list<string> s;
// populate the list s
vector<string> v1(s.begin(), s.end());          // copies strings into v1
vector<string> v2(make_move_iterator(s.begin()),
                  make_move_iterator(s.end())); // moves strings into v2
```

— *end example*]

28.5.3.1 Class template `move_iterator` [move.iterator]

```
namespace std {
  template<class Iterator>
  class move_iterator {
  public:
    using iterator_type     = Iterator;
    using iterator_category = typename iterator_traits<Iterator>::iterator_category;
```

```
  using value_type        = ~~typename iterator_traits<Iterator>::value_type~~iter_value_t<Iterator>;
  using difference_type   = ~~typename iterator_traits<Iterator>::difference_type~~iter_difference_t<Iterator>;
  using pointer           = Iterator;
  using reference         = ~~see below~~iter_rvalue_reference_t<Iterator>;
  using iterator_concept = input_iterator_tag;

  constexpr move_iterator();
  constexpr explicit move_iterator(Iterator i);
  template<class U> constexpr move_iterator(const move_iterator<U>& u);
  template<class U> constexpr move_iterator& operator=(const move_iterator<U>& u);

  constexpr iterator_type base() const;
  constexpr reference operator*() const;
  constexpr pointer operator->() const;

  constexpr move_iterator& operator++();
  constexpr move_iterator operator++(int);
  constexpr move_iterator& operator--();
  constexpr move_iterator operator--(int);

  constexpr move_iterator operator+(difference_type n) const;
  constexpr move_iterator& operator+=(difference_type n);
  constexpr move_iterator operator-(difference_type n) const;
  constexpr move_iterator& operator-=(difference_type n);
  constexpr ~~unspecified ~~reference operator[](difference_type n) const;

  [Editor’s note: These are relocated from the ranges:: namespace.]
  template <Sentinel<Iterator> S>
    friend constexpr bool operator==(
      const move_iterator& x, const move_sentinel<S>& y);
  template <Sentinel<Iterator> S>
    friend constexpr bool operator==(
      const move_sentinel<S>& x, const move_iterator& y);
  template <Sentinel<Iterator> S>
    friend constexpr bool operator!=(
      const move_iterator& x, const move_sentinel<S>& y);
  template <Sentinel<Iterator> S>
    friend constexpr bool operator!=(
      const move_sentinel<S>& x, const move_iterator& y);

  template <SizedSentinel<Iterator> S>
    friend constexpr iter_difference_t<Iterator> operator-(
      const move_sentinel<S>& x, const move_iterator& y);
  template <SizedSentinel<Iterator> S>
    friend constexpr iter_difference_t<Iterator> operator-(
      const move_iterator& x, const move_sentinel<S>& y);

  friend constexpr iter_rvalue_reference_t<Iterator> iter_move(const move_iterator& i)
    noexcept(see below);
  template <IndirectlySwappable<Iterator> Iterator2>
    friend constexpr void iter_swap(const move_iterator& x, const move_iterator<Iterator2>& y)
      noexcept(see below);
private:
  Iterator current;    // exposition only
};
```

```
    template<class Iterator1, class Iterator2>
      constexpr bool operator==(
        const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
    template<class Iterator1, class Iterator2>
      constexpr bool operator!=(
        const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
    template<class Iterator1, class Iterator2>
      constexpr bool operator<(
        const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
    template<class Iterator1, class Iterator2>
      constexpr bool operator<=(
        const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
    template<class Iterator1, class Iterator2>
      constexpr bool operator>(
        const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
    template<class Iterator1, class Iterator2>
      constexpr bool operator>=(
        const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

    template<class Iterator1, class Iterator2>
      constexpr auto operator-(
        const move_iterator<Iterator1>& x,
        const move_iterator<Iterator2>& y) -> decltype(x.base() - y.base());
    template<class Iterator>
      constexpr move_iterator<Iterator> operator+(
        ~~typename move_iterator<Iterator>::difference_type~~iter_difference_t<move_iterator<Iterator>> n,
        const move_iterator<Iterator>& x);
    template<class Iterator>
      constexpr move_iterator<Iterator> make_move_iterator(Iterator i);
  }
```

1 Let *R* denote `iterator_traits<Iterator>::reference`. If `is_reference_v<`*R*` >` is `true`, the template specialization `move_iterator<Iterator>` shall define the nested type named `reference` as a synonym for `remove_reference_t<`*R*` >&&`, otherwise as a synonym for *R*.

28.5.3.2 `move_iterator` requirements [move.iter.requirements]

1 The template parameter `Iterator` shall satisfy the requirements of an input iterator28.3.5.2. Additionally, if any of the bidirectional or random access traversal functions are instantiated, the template parameter shall satisfy the requirements for a bidirectional iterator28.3.5.5 or a random access iterator28.3.5.6, respectively.

28.5.3.3 `move_iterator` operations [move.iter.ops]

28.5.3.3.1 `move_iterator` constructors [move.iter.op.const]

```
constexpr move_iterator();
```

1 *Effects:* Constructs a `move_iterator`, value-initializing `current`. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are defined on a value-initialized iterator of type `Iterator`.

```
constexpr explicit move_iterator(Iterator i);
```

2 *Effects:* Constructs a `move_iterator`, initializing `current` with `i`.

```
template<class U> constexpr move_iterator(const move_iterator<U>& u);
```

3 *Effects:* Constructs a `move_iterator`, initializing `current` with `u.base()`.

4 *Requires:* `U` shall be convertible to `Iterator`.

28.5.3.3.2 `move_iterator::operator=` [move.iter.op=]

`template<class U> constexpr move_iterator& operator=(const move_iterator<U>& u);`

1 *Effects:* Assigns `u.base()` to `current`.

2 *Requires:* `U` shall be convertible to `Iterator`.

28.5.3.3.3 `move_iterator` conversion [move.iter.op.conv]

`constexpr Iterator base() const;`

1 *Returns:* `current`.

28.5.3.3.4 `move_iterator::operator*` [move.iter.op.star]

`constexpr reference operator*() const;`

1 *Returns:* ~~`static_cast<reference>(*current)`~~`ranges::iter_move(current)`.

28.5.3.3.5 `move_iterator::operator->` [move.iter.op.ref]

[Editor's note: My preference is to remove this operator since for `move_iterator`, the expressions `(*i).m` and `i->m` are not, and cannot be, equivalent. I am leaving the operator as-is in an excess of caution.]

`constexpr pointer operator->() const;`

1 *Returns:* `current`.

28.5.3.3.6 `move_iterator::operator++` [move.iter.op.incr]

`constexpr move_iterator& operator++();`

1 *Effects:* As if by `++current`.

2 *Returns:* `*this`.

`constexpr move_iterator operator++(int);`

3 *Effects:* As if by:

```
move_iterator tmp = *this;
++current;
return tmp;
```

28.5.3.3.7 `move_iterator::operator--` [move.iter.op.decr]

`constexpr move_iterator& operator--();`

1 *Effects:* As if by `--current`.

2 *Returns:* `*this`.

`constexpr move_iterator operator--(int);`

3 *Effects:* As if by:

```
move_iterator tmp = *this;
--current;
return tmp;
```

28.5.3.3.8 `move_iterator::operator+` [move.iter.op.+]

```
constexpr move_iterator operator+(difference_type n) const;
```

1 *Returns:* `move_iterator(current + n)`.

28.5.3.3.9 `move_iterator::operator+=` [move.iter.op.+=]

```
constexpr move_iterator& operator+=(difference_type n);
```

1 *Effects:* As if by: `current += n;`

2 *Returns:* `*this`.

28.5.3.3.10 `move_iterator::operator-` [move.iter.op.-]

```
constexpr move_iterator operator-(difference_type n) const;
```

1 *Returns:* `move_iterator(current - n)`.

28.5.3.3.11 `move_iterator::operator-=` [move.iter.op.-=]

```
constexpr move_iterator& operator-=(difference_type n);
```

1 *Effects:* As if by: `current -= n;`

2 *Returns:* `*this`.

28.5.3.3.12 `move_iterator::operator[]` [move.iter.op.index]

`constexpr` ~~*unspecified*~~ `reference` `operator[](difference_type n) const;`

1 *Returns:* ~~`std::move(current[n])`~~`ranges::iter_move(current + n)`.

[Editor's note: Add a new subsection for `move_iterator`'s friend functions:]

28.5.3.3.13 `move_iterator` friend functions [move.iter.op.friend]

```
friend constexpr iter_rvalue_reference_t<Iterator> iter_move(const move_iterator& i)
  noexcept(see below);
```

1 *Effects:* Equivalent to: `return ranges::iter_move(i.current);`

2 *Remarks:* The expression in `noexcept` is equivalent to:

```
noexcept(ranges::iter_move(i.current))
```

```
template <IndirectlySwappable<Iterator> Iterator2>
  friend constexpr void iter_swap(const move_iterator& x, const move_iterator<Iterator2>& y)
    noexcept(see below);
```

3 *Effects:* Equivalent to: `ranges::iter_swap(x.current, y.current)`.

4 *Remarks:* The expression in `noexcept` is equivalent to:

```
noexcept(ranges::iter_swap(x.current, y.current))
```

28.5.3.3.14 `move_iterator` comparisons [move.iter.op.comp]

1 The functions in this subsection only participate in overload resolution if the expression in their *Returns:* clause is well-formed.

```
template<class Iterator1, class Iterator2>
constexpr bool operator==(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <Sentinel<Iterator> S>
friend constexpr bool operator==(const move_iterator& x, const move_sentinel<S>& y);
template <Sentinel<Iterator> S>
friend constexpr bool operator==(const move_sentinel<S>& x, const move_iterator& y);
```

2 *Returns:* `x.base() == y.base()`.

```
template<class Iterator1, class Iterator2>
constexpr bool operator!=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <Sentinel<Iterator> S>
friend constexpr bool operator!=(const move_iterator& x, const move_sentinel<S>& y);
template <Sentinel<Iterator> S>
friend constexpr bool operator!=(const move_sentinel<S>& x, const move_iterator& y);
```

3 *Returns:* `!(x == y)`.

```
template<class Iterator1, class Iterator2>
constexpr bool operator<(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

4 *Returns:* `x.base() < y.base()`.

```
template<class Iterator1, class Iterator2>
constexpr bool operator<=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

5 *Returns:* `!(y < x)`.

```
template<class Iterator1, class Iterator2>
constexpr bool operator>(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

6 *Returns:* `y < x`.

```
template<class Iterator1, class Iterator2>
constexpr bool operator>=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

7 *Returns:* `!(x < y)`.

28.5.3.3.15 `move_iterator` non-member functions [move.iter.nonmember]

1 The functions in this subsection only participate in overload resolution if the expression in their *Returns:* clause is well-formed.

```
template<class Iterator1, class Iterator2>
  constexpr auto operator-(
    const move_iterator<Iterator1>& x,
    const move_iterator<Iterator2>& y) -> decltype(x.base() - y.base());
template <SizedSentinel<Iterator> S>
friend constexpr iter_difference_t<Iterator> operator-(
    const move_sentinel<S>& x, const move_iterator& y);
template <SizedSentinel<Iterator> S>
friend constexpr iter_difference_t<Iterator> operator-(
    const move_iterator& x, const move_sentinel<S>& y);
```

2 *Returns:* `x.base() - y.base()`.

```
template<class Iterator>
  constexpr move_iterator<Iterator> operator+(
    ~~typename move_iterator<Iterator>::difference_type~~iter_difference_t<move_iterator<Iterator>> n,
    const move_iterator<Iterator>& x);
```

3 *Returns:* `x + n`.

```
template<class Iterator>
constexpr move_iterator<Iterator> make_move_iterator(Iterator i);
```

4 *Returns:* `move_iterator<Iterator>(i)`.

28.5.3.4 Class template `move_sentinel` [move.sentinel]

[Editor's note: This section is relocated from REF TODO.]

1 Class template `move_sentinel` is a sentinel adaptor useful for denoting ranges together with `move_iterator`. When an input iterator type `I` and sentinel type `S` satisfy `Sentinel<S, I>`, `Sentinel<move_sentinel<S>, move_iterator<I>>` is satisfied as well.

2 [*Example:* A `move_if` algorithm is easily implemented with `copy_if` using `move_iterator` and `move_sentinel`:

```
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
          IndirectUnaryPredicate<I> Pred>
  requires IndirectlyMovable<I, O>
void move_if(I first, S last, O out, Pred pred)
{
  copy_if(move_iterator<I>{first}, move_sentinel<S>{last}, out, pred);
}
```

— *end example*]

```
namespace std { ~~namespace ranges {~~
  template <Semiregular S>
  class move_sentinel {
  public:
    constexpr move_sentinel();
    explicit constexpr move_sentinel(S s);
    template <ConvertibleTo<S> S2>
      move_sentinel(const move_sentinel<S2>& s);
    template <ConvertibleTo<S> S2>
      move_sentinel& operator=(const move_sentinel<S2>& s);

    S base() const;

  private:
    S last; // exposition only
  };

  ~~template <class I, Sentinel<I> S>~~
    ~~constexpr bool operator==(~~
      ~~const move_iterator<I>& i, const move_sentinel<S>& s);~~
  ~~template <class I, Sentinel<I> S>~~
    ~~constexpr bool operator==(~~
      ~~const move_sentinel<S>& s, const move_iterator<I>& i);~~
  ~~template <class I, Sentinel<I> S>~~
    ~~constexpr bool operator!=(~~
      ~~const move_iterator<I>& i, const move_sentinel<S>& s);~~
  ~~template <class I, Sentinel<I> S>~~
    ~~constexpr bool operator!=(~~
      ~~const move_sentinel<S>& s, const move_iterator<I>& i);~~
```

```
    ~~template <class I, SizedSentinel<I> S>~~
      ~~constexpr difference_type_t<I> operator-(~~
        ~~const move_sentinel<S>& s, const move_iterator<I>& i);~~
    ~~template <class I, SizedSentinel<I> S>~~
      ~~constexpr difference_type_t<I> operator-(~~
        ~~const move_iterator<I>& i, const move_sentinel<S>& s);~~

    ~~template <Semiregular S>~~
      ~~constexpr move_sentinel<S> make_move_sentinel(S s);~~
  }~~}~~
```

28.5.3.5 `move_sentinel` operations [move.sent.ops]

28.5.3.5.1 `move_sentinel` constructors [move.sent.op.const]

```
constexpr move_sentinel();
```

1 *Effects:* Constructs a `move_sentinel`, value-initializing `last`. If `is_trivially_default_constructible<S>::value` is `true`, then this constructor is a `constexpr` constructor.

```
explicit constexpr move_sentinel(S s);
```

2 *Effects:* Constructs a `move_sentinel`, initializing `last` with `s`.

```
template <ConvertibleTo<S> S2>
  move_sentinel(const move_sentinel<S2>& s);
```

3 *Effects:* Constructs a `move_sentinel`, initializing `last` with `s.last`.

28.5.3.5.2 `move_sentinel::operator=` [move.sent.op=]

```
template <ConvertibleTo<S> S2>
  move_sentinel& operator=(const move_sentinel<S2>& s);
```

1 *Effects:* Assigns `s.last` to `last`.

2 *Returns:* `*this`.

[Editor's note: Remove subsections REF TODO [move.sent.op.comp] and [move.sent.nonmember].]

28.5.4 Common iterators [iterators.common]

[Editor's note: Change sub-subsection [iterators.common]/p1 from the Ranges TS as follows:]

1 Class template `common_iterator` is an iterator/sentinel adaptor that is capable of representing a non-common range of elements (where the types of the iterator and sentinel differ) as a common range (where they are the same). It does this by holding either an iterator or a sentinel, and implementing the equality comparison operators appropriately.

[Editor's note: Change sub-subsection "[common.iterator]" from the Ranges TS as follows:]

28.5.4.1 Class template `common_iterator` [common.iterator]

```
  namespace std { ~~namespace ranges {~~
    template <Iterator I, Sentinel<I> S>
      requires !Same<I, S>
    class common_iterator {
    public:
      using difference_type = difference_type_t<I>;

      constexpr common_iterator();
```

```
  constexpr common_iterator(I i);
  constexpr common_iterator(S s);
  template <ConvertibleTo<I> I2, ConvertibleTo<S> S2>
    constexpr common_iterator(const common_iterator<I2, S2>& u);
  template <ConvertibleTo<I> I2, ConvertibleTo<S> S2>
    common_iterator& operator=(const common_iterator<I2, S2>& u);

  decltype(auto) operator*();
  decltype(auto) operator*() const
    requires dereferenceable <const I>;
  decltype(auto) operator->() const
    requires see below;

  common_iterator& operator++();
  decltype(auto) operator++(int);
  common_iterator operator++(int)
    requires ForwardIterator<I>;

  friend rvalue_reference_t<I> iter_move(const common_iterator& i)
    noexcept(see below)
      requires InputIterator<I>;
  template <IndirectlySwappable<I> I2, class S2>
    friend void iter_swap(const common_iterator& x, const common_iterator<I2, S2>& y)
      noexcept(see below);

private:
  bool is_sentinel; // exposition only
  I iter;           // exposition only
  S sentinel;       // exposition only
};

template <Readable I, class S>
struct ~~value_type~~readable_traits<common_iterator<I, S>> {
  ~~using type = value_type_t<I>;~~
  using value_type = iter_value_t<I>;
};

template <InputIterator I, class S>
struct iterator_~~category~~traits<common_iterator<I, S>> {
  using difference_type = iter_difference_t<I>;
  using value_type = iter_value_t<I>;
  using reference = iter_reference_t<I>;
  using pointer = see below;
  using ~~type~~iterator_category = input_iterator_tag;
  using iterator_concept = input_iterator_tag;
};

template <ForwardIterator I, class S>
struct iterator_~~category~~traits<common_iterator<I, S>> {
  using difference_type = iter_difference_t<I>;
  using value_type = iter_value_t<I>;
  using reference = iter_reference_t<I>;
  using pointer = see below;
  using ~~type~~iterator_category = forward_iterator_tag;
  using iterator_concept = forward_iterator_tag;
```

```
  };

  template <class I1, class I2, Sentinel<I2> S1, Sentinel<I1> S2>
  bool operator==(
    const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);
  template <class I1, class I2, Sentinel<I2> S1, Sentinel<I1> S2>
    requires EqualityComparableWith<I1, I2>
  bool operator==(
    const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);
  template <class I1, class I2, Sentinel<I2> S1, Sentinel<I1> S2>
  bool operator!=(
    const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);

  template <class I2, SizedSentinel<I2> I1, SizedSentinel<I2> S1, SizedSentinel<I1> S2>
  difference_type_t<I2> operator-(
    const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);
}~~}~~
```

1 For both specializations of `iterator_traits` for `common_iterator<I, S>`, the nested `pointer` type alias is defined as follows:

(1.1) — If the expression `a.operator->()` is well-formed, where `a` is an lvalue of type `const common_iterator<I, S>`, then `pointer` is an alias for the type of that expression.

(1.2) — Otherwise, `pointer` is an alias for `void`.

28.5.5 Default sentinels [default.sentinels]

28.5.5.1 Class `default_sentinel` [default.sent]

```
namespace std { ~~namespace ranges {~~
  class default_sentinel { };
}~~}~~
```

1 Class `default_sentinel` is an empty type used to denote the end of a range. It is intended to be used together with iterator types that know the bound of their range (e.g., `counted_iterator` (28.5.6.1)).

28.5.6 Counted iterators [iterators.counted]

28.5.6.1 Class template `counted_iterator` [counted.iterator]

[Editor's note: Change the class synopsis of `counted_iterator` as follows:]

```
namespace std { ~~namespace ranges {~~
  template <Iterator I>
  class counted_iterator {
  public:
    using iterator_type = I;
    using difference_type = difference_type_t<I>;

    constexpr counted_iterator();
    constexpr counted_iterator(I x, difference_type_t<I> n);
    template <ConvertibleTo<I> I2>
      constexpr counted_iterator(const counted_iterator<I2>& i);
    template <ConvertibleTo<S> S2>
      constexpr counted_iterator& operator=(const counted_iterator<S2>& i);

    constexpr I base() const;
    constexpr difference_type_t<I> count() const;
```

```
  constexpr decltype(auto) operator*();
  constexpr decltype(auto) operator*() const
   requires dereferenceable <const I>;

  constexpr counted_iterator& operator++();
  decltype(auto) operator++(int);
  constexpr counted_iterator operator++(int)
   requires ForwardIterator<I>;
  constexpr counted_iterator& operator--()
   requires BidirectionalIterator<I>;
  constexpr counted_iterator operator--(int)
   requires BidirectionalIterator<I>;

  constexpr counted_iterator  operator+ (difference_type n) const
   requires RandomAccessIterator<I>;
  constexpr counted_iterator& operator+=(difference_type n)
   requires RandomAccessIterator<I>;
  constexpr counted_iterator  operator- (difference_type n) const
   requires RandomAccessIterator<I>;
  constexpr counted_iterator& operator-=(difference_type n)
   requires RandomAccessIterator<I>;
  constexpr decltype(auto) operator[](difference_type n) const
   requires RandomAccessIterator<I>;

  friend constexpr rvalue_reference_t<I> iter_move(const counted_iterator& i)
   noexcept(see below)
    requires InputIterator<I>;
  template <IndirectlySwappable<I> I2>
   friend constexpr void iter_swap(const counted_iterator& x, const counted_iterator<I2>& y)
    noexcept(see below);

 private:
  I current; // exposition only
  difference_type_t<I> cnt; // exposition only
 };

 template <Readable I>
 struct ~~value_type~~readable_traits<counted_iterator<I>> {
  ~~using type = value_type_t<I>;~~
  using value_type = iter_value_t<I>;
 };

 template <InputIterator I>
 struct iterator_~~category~~traits<counted_iterator<I>>
  :  iterator_traits<I> {
  ~~using type = iterator_category_t<I>;~~
  using pointer = void;
 };

 template <class I1, class I2>
   requires Common<I1, I2>
  constexpr bool operator==(
   const counted_iterator<I1>& x, const counted_iterator<I2>& y);
 // ... as before
}~~}~~
```

28.5.7 Unreachable sentinel [unreachable.sentinels]

28.5.7.1 Class `unreachable` [unreachable.sentinel]

[Editor's note: This section integrates the fix for stl2#507.]

1 Class `unreachable` is a ~~sentinel~~placeholder type that can be used with any ~~`Iterator`~~`WeaklyIncrementable` type to denote ~~an infinite range~~the "upper bound" of an open interval. Comparing ~~an iterator~~anything for equality with an object of type `unreachable` always returns `false`.

[*Example:*

```
char* p;
// set p to point to a character buffer containing newlines
char* nl = find(p, unreachable(), '\n');
```

Provided a newline character really exists in the buffer, the use of `unreachable` above potentially makes the call to `find` more efficient since the loop test against the sentinel does not require a conditional branch. — *end example*]

```
namespace std { ~~namespace ranges {~~
  class unreachable { };

  template <~~Iterator~~WeaklyIncrementable I>
    constexpr bool operator==(const I&, unreachable) noexcept;
  template <~~Iterator~~WeaklyIncrementable I>
    constexpr bool operator==(unreachable, const I&) noexcept;
  template <~~Iterator~~WeaklyIncrementable I>
    constexpr bool operator!=(const I&, unreachable) noexcept;
  template <~~Iterator~~WeaklyIncrementable I>
    constexpr bool operator!=(unreachable, const I&) noexcept;
}~~}~~
```

28.5.7.2 `unreachable` operations [unreachable.sentinel.ops]

28.5.7.2.1 `operator==` [unreachable.sentinel.op==]

```
template <~~Iterator~~WeaklyIncrementable I>
  constexpr bool operator==(const I&, unreachable) noexcept;
template <~~Iterator~~WeaklyIncrementable I>
  constexpr bool operator==(unreachable, const I&) noexcept;
```

1 *Returns:* `false`.

28.5.7.2.2 `operator!=` [unreachable.sentinel.op!=]

```
template <~~Iterator~~WeaklyIncrementable I>
  constexpr bool operator!=(const I& x, unreachable y) noexcept;
template <~~Iterator~~WeaklyIncrementable I>
  constexpr bool operator!=(unreachable x, const I& y) noexcept;
```

1 *Returns:* `true`.

28.6 Stream iterators [stream.iterators]

28.6.1 Class template `istream_iterator` [istream.iterator]

[Editor's note: Change the class synopsis of `istream_iterator` from the working draft of the IS as follows:]

```
namespace std {
```

```
    template<class T, class charT = char, class traits = char_traits<charT>,
             class Distance = ptrdiff_t>
    class istream_iterator {
    public:
      using iterator_category = input_iterator_tag;
      using value_type        = T;
      using difference_type   = Distance;
      using pointer           = const T*;
      using reference         = const T&;
      using char_type         = charT;
      using traits_type       = traits;
      using istream_type      = basic_istream<charT,traits>;

      constexpr istream_iterator();
      constexpr istream_iterator(default_sentinel);
      istream_iterator(istream_type& s);
      istream_iterator(const istream_iterator& x) = default;
      ~istream_iterator() = default;

      const T& operator*() const;
      const T* operator->() const;
      istream_iterator& operator++();
      istream_iterator  operator++(int);

      [Editor's note:  Relocated from namespace ranges:]
      friend bool operator==(default_sentinel, const istream_iterator& i);
      friend bool operator==(const istream_iterator& i, default_sentinel);
      friend bool operator!=(default_sentinel x, const istream_iterator& y);
      friend bool operator!=(const istream_iterator& x, default_sentinel y);

    private:
      basic_istream<charT,traits>* in_stream; // exposition only
      T value;                                // exposition only
    };

    template<class T, class charT, class traits, class Distance>
      bool operator==(const istream_iterator<T,charT,traits,Distance>& x,
              const istream_iterator<T,charT,traits,Distance>& y);
    template<class T, class charT, class traits, class Distance>
      bool operator!=(const istream_iterator<T,charT,traits,Distance>& x,
              const istream_iterator<T,charT,traits,Distance>& y);
  }
```

28.6.1.1 istream_iterator constructors and destructor [istream.iterator.cons]

[Editor's note: Change [istream.iterator.cons] as follows:]

```
constexpr istream_iterator();
constexpr istream_iterator(default_sentinel);
```

1 *Effects:* Constructs the end-of-stream iterator. If `is_trivially_default_constructible_v<T>` is `true`, then ~~this constructor is a~~these constructors are constexpr constructors.

2 *Postconditions:* `in_stream == 0`.

28.6.1.2 istream_iterator operations [istream.iterator.ops]

[Editor's note: Change [istream.iterator.ops] as follows:]

```
template<class T, class charT, class traits, class Distance>
  bool operator==(const istream_iterator<T,charT,traits,Distance>& x,
                  const istream_iterator<T,charT,traits,Distance>& y);
friend bool operator==(default_sentinel, const istream_iterator& i);
friend bool operator==(const istream_iterator& i, default_sentinel);
```

8 *Returns:* `x.in_stream == y.in_stream` for the first overload, and `!i.in_stream` for the other two.

```
template<class T, class charT, class traits, class Distance>
  bool operator!=(const istream_iterator<T,charT,traits,Distance>& x,
                  const istream_iterator<T,charT,traits,Distance>& y);
friend bool operator!=(default_sentinel x, const istream_iterator& y);
friend bool operator!=(const istream_iterator& x, default_sentinel y);
```

9 *Returns:* `!(x == y)`

28.6.2 Class template `ostream_iterator` [ostream.iterator]

[Editor's note: Change the class synopsis of `ostream_iterator` as follows:]

```
namespace std {
  template<class T, class charT = char, class traits = char_traits<charT>>
  class ostream_iterator {
  public:
    using iterator_category = output_iterator_tag;
    using value_type        = void;
    using difference_type   = ~~void~~ptrdiff_t;
    using pointer           = void;
    using reference         = void;
    using char_type         = charT;
    using traits_type       = traits;
    using ostream_type      = basic_ostream<charT,traits>;

    constexpr ostream_iterator() noexcept = default;
    ostream_iterator(ostream_type& s);
    ostream_iterator(ostream_type& s, const charT* delimiter);
    ostream_iterator(const ostream_iterator& x);
    ~ostream_iterator();
    ostream_iterator& operator=(const T& value);

    ostream_iterator& operator*();
    ostream_iterator& operator++();
    ostream_iterator& operator++(int);

  private:
    basic_ostream<charT,traits>* out_stream = nullptr;   // exposition only
    const charT* delim = nullptr;                        // exposition only
  };
}
```

28.6.3 Class template `istreambuf_iterator` [istreambuf.iterator]

[Editor's note: Change the class synopsis of `istreambuf_iterator` as follows:]

```
namespace std {
  template<class charT, class traits = char_traits<charT>>
  class istreambuf_iterator {
  public:
```

```
      using iterator_category = input_iterator_tag;
      using value_type        = charT;
      using difference_type   = typename traits::off_type;
      using pointer           = unspecified ;
      using reference         = charT;
      using char_type         = charT;
      using traits_type       = traits;
      using int_type          = typename traits::int_type;
      using streambuf_type    = basic_streambuf<charT,traits>;
      using istream_type      = basic_istream<charT,traits>;

      class proxy;                          // exposition only

      constexpr istreambuf_iterator() noexcept;
      constexpr istreambuf_iterator(default_sentinel) noexcept;
      istreambuf_iterator(const istreambuf_iterator&) noexcept = default;
      ~istreambuf_iterator() = default;
      istreambuf_iterator(istream_type& s) noexcept;
      istreambuf_iterator(streambuf_type* s) noexcept;
      istreambuf_iterator(const proxy& p) noexcept;
      charT operator*() const;
      istreambuf_iterator& operator++();
      proxy operator++(int);
      bool equal(const istreambuf_iterator& b) const;

      friend bool operator==(default_sentinel s, const istreambuf_iterator& i);
      friend bool operator==(const istreambuf_iterator& i, default_sentinel s);
      friend bool operator!=(default_sentinel a, const istreambuf_iterator& b);
      friend bool operator!=(const istreambuf_iterator& a, default_sentinel b);

    private:
      streambuf_type* sbuf_;                // exposition only
    };

    template<class charT, class traits>
      bool operator==(const istreambuf_iterator<charT,traits>& a,
              const istreambuf_iterator<charT,traits>& b);
    template<class charT, class traits>
      bool operator!=(const istreambuf_iterator<charT,traits>& a,
              const istreambuf_iterator<charT,traits>& b);
  }
```

28.6.3.2 istreambuf_iterator constructors [istreambuf.iterator.cons]

[Editor's note: Change `istreambuf_iterator`'s constructors as follows:]

```
constexpr istreambuf_iterator() noexcept;
constexpr istreambuf_iterator(default_sentinel) noexcept;
```

2 *Effects:* Initializes `sbuf_` with `nullptr`.

28.6.3.3 istreambuf_iterator operations [istreambuf.iterator.ops]

[Editor's note: Change `istreambuf_iterator`'s comparison operators as follows:]

```
template<class charT, class traits>
  bool operator==(const istreambuf_iterator<charT,traits>& a,
```

```
                const istreambuf_iterator<charT,traits>& b);
```

6 *Returns:* `a.equal(b)`.

```
friend bool operator==(default_sentinel s, const istreambuf_iterator& i);
friend bool operator==(const istreambuf_iterator& i, default_sentinel s);
```

7 *Returns:* `i.equal(s)`.

```
template<class charT, class traits>
  bool operator!=(const istreambuf_iterator<charT,traits>& a,
                const istreambuf_iterator<charT,traits>& b);
friend bool operator!=(default_sentinel a, const istreambuf_iterator& b);
friend bool operator!=(const istreambuf_iterator& a, default_sentinel b);
```

8 *Returns:* ~~`!a.equal(b)`~~`!(a == b)`.

28.6.4 Class template `ostreambuf_iterator` [ostreambuf.iterator]

[Editor's note: Change the `ostreambuf_iterator` class synopsis as follows:]

```
  namespace std {
    template<class charT, class traits = char_traits<charT>>
    class ostreambuf_iterator {
    public:
      using iterator_category = output_iterator_tag;
      using value_type        = void;
      using difference_type   = ~~void~~ptrdiff_t;
      using pointer           = void;
      using reference         = void;
      using char_type         = charT;
      using traits_type       = traits;
      using streambuf_type    = basic_streambuf<charT,traits>;
      using ostream_type      = basic_ostream<charT,traits>;

      constexpr ostreambuf_iterator() noexcept = default;
      ostreambuf_iterator(ostream_type& s) noexcept;
      ostreambuf_iterator(streambuf_type* s) noexcept;
      ostreambuf_iterator& operator=(charT c);

      ostreambuf_iterator& operator*();
      ostreambuf_iterator& operator++();
      ostreambuf_iterator& operator++(int);
      bool failed() const noexcept;

    private:
      streambuf_type* sbuf_ = nullptr;                // exposition only
    };
  }
```

29 Ranges library [ranges]

29.1 General [ranges.general]

1 This clause describes components for dealing with ranges of elements.

2 The following subclauses describe range and view requirements, and components for range primitives as summarized in Table 10.

Table 10 — Ranges library summary

Subclause		Header(s)
~~Clause 28~~	~~Iterators~~	~~`<range>`~~
29.4	Range access	`<range>`
29.5	Range primitives	
29.6	Requirements	
~~Clause 30~~	~~Algorithms~~	

29.2 decay_copy [ranges.decaycopy]

[Editor's note: ... as in P0896R1.]

29.3 Header `<range>` synopsis [ranges.synopsis]

[Editor's note: Remove the `<range>` synopsis from P0896R1 and replace it with the following. Note that everything except the range access points and range primitives gets promoted to namespace `std`.]

```
#include <iterator>
#include <initializer_list>

namespace std {
  namespace ranges {
    inline namespace unspecified  {
      // 29.4, range access:
      inline constexpr unspecified  begin = unspecified ;
      inline constexpr unspecified  end = unspecified ;
      inline constexpr unspecified  cbegin = unspecified ;
      inline constexpr unspecified  cend = unspecified ;
      inline constexpr unspecified  rbegin = unspecified ;
      inline constexpr unspecified  rend = unspecified ;
      inline constexpr unspecified  crbegin = unspecified ;
      inline constexpr unspecified  crend = unspecified ;

      // 29.5, range primitives:
      inline constexpr unspecified  size = unspecified ;
      inline constexpr unspecified  empty = unspecified ;
      inline constexpr unspecified  data = unspecified ;
      inline constexpr unspecified  cdata = unspecified ;
    }
  }

  template <class T>
```

```
  using iterator_t = decltype(ranges::begin(declval<T&>()));

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

  template <class>
  constexpr bool disable_sized_range = false;

  template <class T>
  struct enable_view { };

  struct view_base { };

  // 29.6, range requirements:

  // 29.6.2, Range:
  template <class T>
  concept Range = see below;

  // 29.6.3, SizedRange:
  template <class T>
  concept SizedRange = see below;

  // 29.6.4, View:
  template <class T>
  concept View = see below;

  // 29.6.5, CommonRange:
  template <class T>
  concept CommonRange = see below;

  // 29.6.6, InputRange:
  template <class T>
  concept InputRange = see below;

  // 29.6.7, OutputRange:
  template <class R, class T>
  concept OutputRange = see below;

  // 29.6.8, ForwardRange:
  template <class T>
  concept ForwardRange = see below;

  // 29.6.9, BidirectionalRange:
  template <class T>
  concept BidirectionalRange = see below;

  // 29.6.10, RandomAccessRange:
  template <class T>
  concept RandomAccessRange = see below;
}
```

[Editor's note: Remove 29.4, [ranges.iterators], from P0896R1.]

29.4 Range access [ranges.access]

1 In addition to begin available via inclusion of the `<range>` header, the customization point objects in 29.4 are available when `<iterator>` is included.

[Editor's note: ...otherwise, as in P0896R1.]

29.5 Range primitives [ranges.primitives]

1 In addition to begin available via inclusion of the `<range>` header, the customization point objects in 29.5 are available when `<iterator>` is included.

[Editor's note: ...otherwise, as in P0896R1.]

29.6 Range requirements [ranges.requirements]

29.6.1 General [ranges.requirements.general]

[Editor's note: ...as in P0896R1.]

29.6.2 Ranges [ranges.range]

[Editor's note: ...as in P0896R1.]

29.6.3 Sized ranges [ranges.sized]

[Editor's note: Change the definition of the `SizedRange` concept as follows:]

1 The `SizedRange` concept specifies the requirements of a `Range` type that knows its size in constant time with the `size` function.

```
template <class T>
concept SizedRange =
  Range<T> &&
  !disable_sized_range<remove_cvref_t<~~remove_reference_t<~~R>~~>~~> &&
  requires(T& t) {
    { ranges::size(t) } -> ConvertibleTo<~~difference_type_t~~iter_difference_t<iterator_t<T>>>;
  };
```

2 Given an lvalue `t` of type `remove_reference_t<T>`, `SizedRange<T>` is satisfied only if:

(2.1) — `ranges::size(t)` is $\mathcal{O}(1)$, does not modify `t`, and is equal to `ranges::distance(t)`.

(2.2) — If `iterator_t<T>` satisfies `ForwardIterator`, `size(t)` is well-defined regardless of the evaluation of `begin(t)`. [*Note:* `size(t)` is otherwise not required be well-defined after evaluating `begin(t)`. For a `SizedRange` whose iterator type does not model `ForwardIterator`, for example, `size(t)` might only be well-defined if evaluated before the first call to `begin(t)`. — *end note*]

3 [*Note:* The `disable_sized_range` predicate provides a mechanism to enable use of range types with the library that meet the syntactic requirements but do not in fact satisfy `SizedRange`. A program that instantiates a library template that requires a `Range` with such a range type `R` is ill-formed with no diagnostic required unless `disable_sized_range<remove_cvref_t<`~~`remove_reference_t<`~~`R>`~~`>`~~`>` evaluates to `true` (). — *end note*]

29.6.4 Views [ranges.view]

[Editor's note: ...as in P0896R1.]

29.6.5 Common ranges [ranges.common]

[Editor's note: ...as in P0896R1.]

29.6.6 Input ranges [ranges.input]

[Editor's note: ...as in P0896R1.]

29.6.7 Output ranges [ranges.output]

[Editor's note: ...as in P0896R1.]

29.6.8 Forward ranges [ranges.forward]

[Editor's note: ...as in P0896R1.]

29.6.9 Bidirectional ranges [ranges.bidirectional]

[Editor's note: ...as in P0896R1.]

29.6.10 Random access ranges [ranges.random.access]

[Editor's note: ...as in P0896R1.]

29.6.11 Contiguous ranges [ranges.contiguous]

[Editor's note: ...as in P0944.]

29.7 Dangling wrapper [dangling.wrappers]

29.7.1 Class template `dangling` [dangling.wrap]

1 Class template `dangling` is a wrapper for an object that refers to another object whose lifetime may have ended. It is used by algorithms that accept rvalue ranges and return iterators.

```
namespace std { namespace ranges {
  template <CopyConstructible T>
  class dangling {
  public:
    constexpr dangling() requires DefaultConstructible<T>;
    constexpr dangling(T t);
    constexpr T get_unsafe() const;
  private:
    T value; // exposition only
  };

  template <Range R>
  using safe_iterator_t =
    conditional_t<is_lvalue_reference_v<R>,
      iterator_t<R>,
      dangling<iterator_t<R>>>;
}}
```

29.7.1.1 `dangling` operations [dangling.wrap.ops]

29.7.1.1.1 `dangling` constructors [dangling.wrap.op.const]

`constexpr dangling() requires DefaultConstructible<T>;`

1 *Effects:* Constructs a `dangling`, value-initializing `value`.

`constexpr dangling(T t);`

2 *Effects:* Constructs a `dangling`, initializing `value` with `std::move(t)`.

29.7.1.1.2 `dangling::get_unsafe` [dangling.wrap.op.get]

`constexpr T get_unsafe() const;`

1 *Returns:* `value`.

[Editor's note: Remove 29.9, [ranges.algorithms], from P0896R1.]

30 Algorithms library [algorithms]

30.1 General [algorithms.general]

1 This Clause describes components that C++ programs may use to perform algorithmic operations on containers26 and other sequences.

2 The following subclauses describe components for non-modifying sequence operations, mutating sequence operations, sorting and related operations, and algorithms from the ISO C library, as summarized in Table 11.

Table 11 — Algorithms library summary

Subclause		Header(s)
30.5	Non-modifying sequence operations	`<algorithm>`
30.6	Mutating sequence operations	
30.7	Sorting and related operations	
30.8	C library algorithms	`<cstdlib>`

30.2 Header `<algorithm>` synopsis [algorithm.syn]

```
#include <initializer_list>

namespace std {
  // 30.5, non-modifying sequence operations
  // 30.5.1, all of
  template<class InputIterator, class Predicate>
    constexpr bool all_of(InputIterator first, InputIterator last, Predicate pred);
  template<class ExecutionPolicy, class ForwardIterator, class Predicate>
    bool all_of(ExecutionPolicy&& exec, // see 30.4.5
                ForwardIterator first, ForwardIterator last, Predicate pred);
  namespace ranges {
    template <InputIterator I, Sentinel<I> S, class Proj = identity,
        IndirectUnaryPredicate<projected<I, Proj>> Pred>
      bool all_of(I first, S last, Pred pred, Proj proj = Proj{});
    template <InputRange Rng, class Proj = identity,
        IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
      bool all_of(Rng&& rng, Pred pred, Proj proj = Proj{});
  }

  // 30.5.2, any of
  template<class InputIterator, class Predicate>
    constexpr bool any_of(InputIterator first, InputIterator last, Predicate pred);
  template<class ExecutionPolicy, class ForwardIterator, class Predicate>
    bool any_of(ExecutionPolicy&& exec, // see 30.4.5
                ForwardIterator first, ForwardIterator last, Predicate pred);
  namespace ranges {
    template <InputIterator I, Sentinel<I> S, class Proj = identity,
        IndirectUnaryPredicate<projected<I, Proj>> Pred>
      bool any_of(I first, S last, Pred pred, Proj proj = Proj{});
    template <InputRange Rng, class Proj = identity,
        IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
```

```
    bool any_of(Rng&& rng, Pred pred, Proj proj = Proj{});
}

// 30.5.3, none of
template<class InputIterator, class Predicate>
  constexpr bool none_of(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  bool none_of(ExecutionPolicy&& exec, // see 30.4.5
               ForwardIterator first, ForwardIterator last, Predicate pred);
namespace ranges {
  template <InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    bool none_of(I first, S last, Pred pred, Proj proj = Proj{});
  template <InputRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    bool none_of(Rng&& rng, Pred pred, Proj proj = Proj{});
}

// 30.5.4, for each
template<class InputIterator, class Function>
  constexpr Function for_each(InputIterator first, InputIterator last, Function f);
template<class ExecutionPolicy, class ForwardIterator, class Function>
  void for_each(ExecutionPolicy&& exec, // see 30.4.5
                ForwardIterator first, ForwardIterator last, Function f);
namespace ranges {
  template <InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryInvocable<projected<I, Proj>> Fun>
    tagged_pair<tag::in(I), tag::fun(Fun)>
      for_each(I first, S last, Fun f, Proj proj = Proj{});
  template <InputRange Rng, class Proj = identity,
      IndirectUnaryInvocable<projected<iterator_t<Rng>, Proj>> Fun>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::fun(Fun)>
      for_each(Rng&& rng, Fun f, Proj proj = Proj{});
}
template<class InputIterator, class Size, class Function>
  constexpr InputIterator for_each_n(InputIterator first, Size n, Function f);
template<class ExecutionPolicy, class ForwardIterator, class Size, class Function>
  ForwardIterator for_each_n(ExecutionPolicy&& exec, // see 30.4.5
                             ForwardIterator first, Size n, Function f);

// 30.5.5, find
template<class InputIterator, class T>
  constexpr InputIterator find(InputIterator first, InputIterator last,
                               const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  ForwardIterator find(ExecutionPolicy&& exec, // see 30.4.5
                       ForwardIterator first, ForwardIterator last,
                       const T& value);
template<class InputIterator, class Predicate>
  constexpr InputIterator find_if(InputIterator first, InputIterator last,
                                  Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator find_if(ExecutionPolicy&& exec, // see 30.4.5
                          ForwardIterator first, ForwardIterator last,
                          Predicate pred);
```

```
template<class InputIterator, class Predicate>
  constexpr InputIterator find_if_not(InputIterator first, InputIterator last,
                                      Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator find_if_not(ExecutionPolicy&& exec, // see 30.4.5
                              ForwardIterator first, ForwardIterator last,
                              Predicate pred);
namespace ranges {
  template <InputIterator I, Sentinel<I> S, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T*>
    I find(I first, S last, const T& value, Proj proj = Proj{});
  template <InputRange Rng, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
    safe_iterator_t<Rng>
      find(Rng&& rng, const T& value, Proj proj = Proj{});
  template <InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    I find_if(I first, S last, Pred pred, Proj proj = Proj{});
  template <InputRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    safe_iterator_t<Rng>
      find_if(Rng&& rng, Pred pred, Proj proj = Proj{});
  template <InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    I find_if_not(I first, S last, Pred pred, Proj proj = Proj{});
  template <InputRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    safe_iterator_t<Rng>
      find_if_not(Rng&& rng, Pred pred, Proj proj = Proj{});
}

// 30.5.6, find end
template<class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
  constexpr ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    find_end(ExecutionPolicy&& exec, // see 30.4.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1,
         class ForwardIterator2, class BinaryPredicate>
  ForwardIterator1
    find_end(ExecutionPolicy&& exec, // see 30.4.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);
namespace ranges {
  template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
```

```
   Sentinel<I2> S2, class Proj = identity,
   IndirectRelation<I2, projected<I1, Proj>> Pred = ranges::equal_to<>>
  I1
   find_end(I1 first1, S1 last1, I2 first2, S2 last2,
       Pred pred = Pred{}, Proj proj = Proj{});
 template <ForwardRange Rng1, ForwardRange Rng2, class Proj = identity,
   IndirectRelation<iterator_t<Rng2>,
    projected<iterator_t<Rng>, Proj>> Pred = ranges::equal_to<>>
  safe_iterator_t<Rng1>
   find_end(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{}, Proj proj = Proj{});
}

// 30.5.7, find first
template<class InputIterator, class ForwardIterator>
 constexpr InputIterator
  find_first_of(InputIterator first1, InputIterator last1,
         ForwardIterator first2, ForwardIterator last2);
template<class InputIterator, class ForwardIterator, class BinaryPredicate>
 constexpr InputIterator
  find_first_of(InputIterator first1, InputIterator last1,
         ForwardIterator first2, ForwardIterator last2,
         BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
 ForwardIterator1
  find_first_of(ExecutionPolicy&& exec, // see 30.4.5
         ForwardIterator1 first1, ForwardIterator1 last1,
         ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1,
    class ForwardIterator2, class BinaryPredicate>
 ForwardIterator1
  find_first_of(ExecutionPolicy&& exec, // see 30.4.5
         ForwardIterator1 first1, ForwardIterator1 last1,
         ForwardIterator2 first2, ForwardIterator2 last2,
         BinaryPredicate pred);
namespace ranges {
 template <InputIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2,
   class Proj1 = identity, class Proj2 = identity,
   IndirectRelation<projected<I1, Proj1>, projected<I2, Proj2>> Pred = ranges::equal_to<>>
  I1
   find_first_of(I1 first1, S1 last1, I2 first2, S2 last2,
          Pred pred = Pred{},
          Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
 template <InputRange Rng1, ForwardRange Rng2, class Proj1 = identity,
   class Proj2 = identity,
   IndirectRelation<projected<iterator_t<Rng1>, Proj1>,
    projected<iterator_t<Rng2>, Proj2>> Pred = ranges::equal_to<>>
  safe_iterator_t<Rng1>
   find_first_of(Rng1&& rng1, Rng2&& rng2,
          Pred pred = Pred{},
          Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

// 30.5.8, adjacent find
template<class ForwardIterator>
 constexpr ForwardIterator
```

```
    adjacent_find(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class BinaryPredicate>
  constexpr ForwardIterator
    adjacent_find(ForwardIterator first, ForwardIterator last,
                  BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator
    adjacent_find(ExecutionPolicy&& exec, // see 30.4.5
                  ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class BinaryPredicate>
  ForwardIterator
    adjacent_find(ExecutionPolicy&& exec, // see 30.4.5
                  ForwardIterator first, ForwardIterator last,
                  BinaryPredicate pred);
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectRelation<projected<I, Proj>> Pred = ranges::equal_to<>>
    I adjacent_find(I first, S last, Pred pred = Pred{},
                    Proj proj = Proj{});
  template <ForwardRange Rng, class Proj = identity,
      IndirectRelation<projected<iterator_t<Rng>, Proj>> Pred = ranges::equal_to<>>
    safe_iterator_t<Rng>
      adjacent_find(Rng&& rng, Pred pred = Pred{}, Proj proj = Proj{});
}

// 30.5.9, count
template<class InputIterator, class T>
  constexpr typename iterator_traits<InputIterator>::difference_type
    count(InputIterator first, InputIterator last, const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  typename iterator_traits<ForwardIterator>::difference_type
    count(ExecutionPolicy&& exec, // see 30.4.5
          ForwardIterator first, ForwardIterator last, const T& value);
template<class InputIterator, class Predicate>
  constexpr typename iterator_traits<InputIterator>::difference_type
    count_if(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  typename iterator_traits<ForwardIterator>::difference_type
    count_if(ExecutionPolicy&& exec, // see 30.4.5
             ForwardIterator first, ForwardIterator last, Predicate pred);
namespace ranges {
  template <InputIterator I, Sentinel<I> S, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T*>
    iter_difference_t<I>
      count(I first, S last, const T& value, Proj proj = Proj{});
  template <InputRange Rng, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
    iter_difference_t<iterator_t<Rng>>
      count(Rng&& rng, const T& value, Proj proj = Proj{});
  template <InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    iter_difference_t<I>
      count_if(I first, S last, Pred pred, Proj proj = Proj{});
  template <InputRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
```

```
    iter_difference_t<iterator_t<Rng>>
      count_if(Rng&& rng, Pred pred, Proj proj = Proj{});
}

// 30.5.10, mismatch
template<class InputIterator1, class InputIterator2>
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2);
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, BinaryPredicate pred);
template<class InputIterator1, class InputIterator2>
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2);
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2,
             BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec, // see 30.4.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec, // see 30.4.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec, // see 30.4.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec, // see 30.4.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);
namespace ranges {
  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      class Proj1 = identity, class Proj2 = identity,
      IndirectRelation<projected<I1, Proj1>, projected<I2, Proj2>> Pred = ranges::equal_to<>>
    tagged_pair<tag::in1(I1), tag::in2(I2)>
      mismatch(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template <InputRange Rng1, InputRange Rng2,
      class Proj1 = identity, class Proj2 = identity,
      IndirectRelation<projected<iterator_t<Rng1>, Proj1>,
```

```
        projected<iterator_t<Rng2>, Proj2>> Pred = ranges::equal_to<>>
    tagged_pair<tag::in1(safe_iterator_t<Rng1>),
                tag::in2(safe_iterator_t<Rng2>)>
      mismatch(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

// 30.5.11, equal
template<class InputIterator1, class InputIterator2>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2);
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, BinaryPredicate pred);
template<class InputIterator1, class InputIterator2>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, InputIterator2 last2);
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, InputIterator2 last2,
                       BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  bool equal(ExecutionPolicy&& exec, // see 30.4.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  bool equal(ExecutionPolicy&& exec, // see 30.4.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  bool equal(ExecutionPolicy&& exec, // see 30.4.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  bool equal(ExecutionPolicy&& exec, // see 30.4.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);
namespace ranges {
  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      class Pred = ranges::equal_to<>, class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
    bool equal(I1 first1, S1 last1, I2 first2, S2 last2,
              Pred pred = Pred{},
              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template <InputRange Rng1, InputRange Rng2, class Pred = ranges::equal_to<>,
      class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
    bool equal(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

// 30.5.12, is permutation
```

```
template<class ForwardIterator1, class ForwardIterator2>
  constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                                ForwardIterator2 first2);
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
  constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                                ForwardIterator2 first2, BinaryPredicate pred);
template<class ForwardIterator1, class ForwardIterator2>
  constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                                ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
  constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                                ForwardIterator2 first2, ForwardIterator2 last2,
                                BinaryPredicate pred);
namespace ranges {
  template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
      Sentinel<I2> S2, class Pred = ranges::equal_to<>, class Proj1 = identity,
      class Proj2 = identity>
    requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
    bool is_permutation(I1 first1, S1 last1, I2 first2, S2 last2,
                        Pred pred = Pred{},
                        Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

  template <ForwardRange Rng1, ForwardRange Rng2, class Pred = ranges::equal_to<>,
      class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
    bool is_permutation(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
                        Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

// 30.5.13, search
template<class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator1
    search(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
  constexpr ForwardIterator1
    search(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2,
           BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    search(ExecutionPolicy&& exec, // see 30.4.5
           ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  ForwardIterator1
    search(ExecutionPolicy&& exec, // see 30.4.5
           ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2,
           BinaryPredicate pred);
namespace ranges {
  template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
      Sentinel<I2> S2, class Pred = ranges::equal_to<>,
      class Proj1 = identity, class Proj2 = identity>
```

```
    requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
    I1 search(I1 first1, S1 last1, I2 first2, S2 last2,
              Pred pred = Pred{},
              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

  template <ForwardRange Rng1, ForwardRange Rng2, class Pred = ranges::equal_to<>,
      class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
    safe_iterator_t<Rng1>
      search(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
            Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
template<class ForwardIterator, class Size, class T>
  constexpr ForwardIterator
    search_n(ForwardIterator first, ForwardIterator last,
             Size count, const T& value);
template<class ForwardIterator, class Size, class T, class BinaryPredicate>
  constexpr ForwardIterator
    search_n(ForwardIterator first, ForwardIterator last,
             Size count, const T& value,
             BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Size, class T>
  ForwardIterator
    search_n(ExecutionPolicy&& exec, // see 30.4.5
             ForwardIterator first, ForwardIterator last,
             Size count, const T& value);
template<class ExecutionPolicy, class ForwardIterator, class Size, class T,
         class BinaryPredicate>
  ForwardIterator
    search_n(ExecutionPolicy&& exec, // see 30.4.5
             ForwardIterator first, ForwardIterator last,
             Size count, const T& value,
             BinaryPredicate pred);
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class T,
      class Pred = ranges::equal_to<>, class Proj = identity>
    requires IndirectlyComparable<I, const T*, Pred, Proj>
    I search_n(I first, S last, iter_difference_t<I> count,
               const T& value, Pred pred = Pred{},
               Proj proj = Proj{});

  template <ForwardRange Rng, class T, class Pred = ranges::equal_to<>,
      class Proj = identity>
    requires IndirectlyComparable<iterator_t<Rng>, const T*, Pred, Proj>
    safe_iterator_t<Rng>
      search_n(Rng&& rng, iter_difference_t<iterator_t<Rng>> count,
               const T& value, Pred pred = Pred{}, Proj proj = Proj{});
}

template<class ForwardIterator, class Searcher>
  constexpr ForwardIterator
    search(ForwardIterator first, ForwardIterator last, const Searcher& searcher);

// 30.6, mutating sequence operations
// 30.6.1, copy
```

```
template<class InputIterator, class OutputIterator>
  constexpr OutputIterator copy(InputIterator first, InputIterator last,
                                OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 copy(ExecutionPolicy&& exec, // see 30.4.5
                        ForwardIterator1 first, ForwardIterator1 last,
                        ForwardIterator2 result);
namespace ranges {
  template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
    requires IndirectlyCopyable<I, O>
    tagged_pair<tag::in(I), tag::out(O)>
      copy(I first, S last, O result);
  template <InputRange Rng, WeaklyIncrementable O>
    requires IndirectlyCopyable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
      copy(Rng&& rng, O result);
}
template<class InputIterator, class Size, class OutputIterator>
  constexpr OutputIterator copy_n(InputIterator first, Size n,
                                  OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class Size,
         class ForwardIterator2>
  ForwardIterator2 copy_n(ExecutionPolicy&& exec, // see 30.4.5
                          ForwardIterator1 first, Size n,
                          ForwardIterator2 result);
namespace ranges {
  template <InputIterator I, WeaklyIncrementable O>
    requires IndirectlyCopyable<I, O>
    tagged_pair<tag::in(I), tag::out(O)>
      copy_n(I first, iter_difference_t<I> n, O result);
}
template<class InputIterator, class OutputIterator, class Predicate>
  constexpr OutputIterator copy_if(InputIterator first, InputIterator last,
                                   OutputIterator result, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate>
  ForwardIterator2 copy_if(ExecutionPolicy&& exec, // see 30.4.5
                           ForwardIterator1 first, ForwardIterator1 last,
                           ForwardIterator2 result, Predicate pred);
namespace ranges {
  template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires IndirectlyCopyable<I, O>
    tagged_pair<tag::in(I), tag::out(O)>
      copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});
  template <InputRange Rng, WeaklyIncrementable O, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    requires IndirectlyCopyable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
      copy_if(Rng&& rng, O result, Pred pred, Proj proj = Proj{});
}
template<class BidirectionalIterator1, class BidirectionalIterator2>
  constexpr BidirectionalIterator2
    copy_backward(BidirectionalIterator1 first, BidirectionalIterator1 last,
                  BidirectionalIterator2 result);
```

```
namespace ranges {
  template <BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
    requires IndirectlyCopyable<I1, I2>
    tagged_pair<tag::in(I1), tag::out(I2)>
      copy_backward(I1 first, S1 last, I2 result);
  template <BidirectionalRange Rng, BidirectionalIterator I>
    requires IndirectlyCopyable<iterator_t<Rng>, I>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(I)>
      copy_backward(Rng&& rng, I result);
}

// 30.6.2, move
template<class InputIterator, class OutputIterator>
  constexpr OutputIterator move(InputIterator first, InputIterator last,
                                OutputIterator result);
namespace ranges {
  template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
    requires IndirectlyMovable<I, O>
    tagged_pair<tag::in(I), tag::out(O)>
      move(I first, S last, O result);
  template <InputRange Rng, WeaklyIncrementable O>
    requires IndirectlyMovable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
      move(Rng&& rng, O result);
}
template<class ExecutionPolicy, class ForwardIterator1,
         class ForwardIterator2>
  ForwardIterator2 move(ExecutionPolicy&& exec, // see 30.4.5
                        ForwardIterator1 first, ForwardIterator1 last,
                        ForwardIterator2 result);
template<class BidirectionalIterator1, class BidirectionalIterator2>
  constexpr BidirectionalIterator2
    move_backward(BidirectionalIterator1 first, BidirectionalIterator1 last,
                  BidirectionalIterator2 result);
namespace ranges {
  template <BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
    requires IndirectlyMovable<I1, I2>
    tagged_pair<tag::in(I1), tag::out(I2)>
      move_backward(I1 first, S1 last, I2 result);
  template <BidirectionalRange Rng, BidirectionalIterator I>
    requires IndirectlyMovable<iterator_t<Rng>, I>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(I)>
      move_backward(Rng&& rng, I result);
}

// 30.6.3, swap
template<class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
                               ForwardIterator2 first2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 swap_ranges(ExecutionPolicy&& exec, // see 30.4.5
                               ForwardIterator1 first1, ForwardIterator1 last1,
                               ForwardIterator2 first2);
namespace ranges {
  template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2>
```

```
    requires IndirectlySwappable<I1, I2>
    tagged_pair<tag::in1(I1), tag::in2(I2)>
      swap_ranges(I1 first1, S1 last1, I2 first2, S2 last2);
  template <ForwardRange Rng1, ForwardRange Rng2>
    requires IndirectlySwappable<iterator_t<Rng1>, iterator_t<Rng2>>
    tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::in2(safe_iterator_t<Rng2>)>
      swap_ranges(Rng1&& rng1, Rng2&& rng2);
}
template<class ForwardIterator1, class ForwardIterator2>
  void iter_swap(ForwardIterator1 a, ForwardIterator2 b);

// 30.6.4, transform
template<class InputIterator, class OutputIterator, class UnaryOperation>
  constexpr OutputIterator
    transform(InputIterator first, InputIterator last,
              OutputIterator result, UnaryOperation op);
template<class InputIterator1, class InputIterator2, class OutputIterator,
         class BinaryOperation>
  constexpr OutputIterator
    transform(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, OutputIterator result,
              BinaryOperation binary_op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class UnaryOperation>
  ForwardIterator2
    transform(ExecutionPolicy&& exec, // see 30.4.5
              ForwardIterator1 first, ForwardIterator1 last,
              ForwardIterator2 result, UnaryOperation op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class BinaryOperation>
  ForwardIterator
    transform(ExecutionPolicy&& exec, // see 30.4.5
              ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator result,
              BinaryOperation binary_op);
namespace ranges {
  template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
      CopyConstructible F, class Proj = identity>
    requires Writable<O, indirect_result_t<F&, projected<I, Proj>>>
    tagged_pair<tag::in(I), tag::out(O)>
      transform(I first, S last, O result, F op, Proj proj = Proj{});
  template <InputRange Rng, WeaklyIncrementable O, CopyConstructible F,
      class Proj = identity>
    requires Writable<O, indirect_result_t<F&, projected<iterator_t<R>, Proj>>>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
      transform(Rng&& rng, O result, F op, Proj proj = Proj{});
  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      WeaklyIncrementable O, CopyConstructible F, class Proj1 = identity,
      class Proj2 = identity>
    requires Writable<O, indirect_result_t<F&, projected<I1, Proj1>,
      projected<I2, Proj2>>>
    tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
      transform(I1 first1, S1 last1, I2 first2, S2 last2, O result,
              F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
```

```
      CopyConstructible F, class Proj1 = identity, class Proj2 = identity>
    requires Writable<O, indirect_result_t<F&,
      projected<iterator_t<Rng1>, Proj1>, projected<iterator_t<Rng2>, Proj2>>>
    tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
                 tag::in2(safe_iterator_t<Rng2>),
                 tag::out(O)>
      transform(Rng1&& rng1, Rng2&& rng2, O result,
                F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

// 30.6.5, replace
template<class ForwardIterator, class T>
  constexpr void replace(ForwardIterator first, ForwardIterator last,
                         const T& old_value, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  void replace(ExecutionPolicy&& exec, // see 30.4.5
               ForwardIterator first, ForwardIterator last,
               const T& old_value, const T& new_value);
template<class ForwardIterator, class Predicate, class T>
  constexpr void replace_if(ForwardIterator first, ForwardIterator last,
                            Predicate pred, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator, class Predicate, class T>
  void replace_if(ExecutionPolicy&& exec, // see 30.4.5
                  ForwardIterator first, ForwardIterator last,
                  Predicate pred, const T& new_value);
namespace ranges {
  template <InputIterator I, Sentinel<I> S, class T1, class T2, class Proj = identity>
    requires Writable<I, const T2&> &&
      IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T1*>
    I replace(I first, S last, const T1& old_value, const T2& new_value, Proj proj = Proj{});
  template <InputRange Rng, class T1, class T2, class Proj = identity>
    requires Writable<iterator_t<Rng>, const T2&> &&
      IndirectRelation<ranges::equal_to<>, projected<iterator_t<Rng>, Proj>, const T1*>
    safe_iterator_t<Rng>
      replace(Rng&& rng, const T1& old_value, const T2& new_value, Proj proj = Proj{});
  template <InputIterator I, Sentinel<I> S, class T, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires Writable<I, const T&>
    I replace_if(I first, S last, Pred pred, const T& new_value, Proj proj = Proj{});
  template <InputRange Rng, class T, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    requires Writable<iterator_t<Rng>, const T&>
    safe_iterator_t<Rng>
      replace_if(Rng&& rng, Pred pred, const T& new_value, Proj proj = Proj{});
}
template<class InputIterator, class OutputIterator, class T>
  constexpr OutputIterator replace_copy(InputIterator first, InputIterator last,
                                        OutputIterator result,
                                        const T& old_value, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T>
  ForwardIterator2 replace_copy(ExecutionPolicy&& exec, // see 30.4.5
                                ForwardIterator1 first, ForwardIterator1 last,
                                ForwardIterator2 result,
                                const T& old_value, const T& new_value);
template<class InputIterator, class OutputIterator, class Predicate, class T>
```

```
  constexpr OutputIterator replace_copy_if(InputIterator first, InputIterator last,
                                           OutputIterator result,
                                           Predicate pred, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate, class T>
  ForwardIterator2 replace_copy_if(ExecutionPolicy&& exec, // see 30.4.5
                                   ForwardIterator1 first, ForwardIterator1 last,
                                   ForwardIterator2 result,
                                   Predicate pred, const T& new_value);
namespace ranges {
  template <InputIterator I, Sentinel<I> S, class T1, class T2, OutputIterator<const T2&> O,
      class Proj = identity>
    requires IndirectlyCopyable<I, O> &&
      IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T1*>
    tagged_pair<tag::in(I), tag::out(O)>
      replace_copy(I first, S last, O result, const T1& old_value, const T2& new_value,
                  Proj proj = Proj{});
  template <InputRange Rng, class T1, class T2, OutputIterator<const T2&> O,
      class Proj = identity>
    requires IndirectlyCopyable<iterator_t<Rng>, O> &&
      IndirectRelation<ranges::equal_to<>, projected<iterator_t<Rng>, Proj>, const T1*>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
      replace_copy(Rng&& rng, O result, const T1& old_value, const T2& new_value,
                  Proj proj = Proj{});
  template <InputIterator I, Sentinel<I> S, class T, OutputIterator<const T&> O,
      class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires IndirectlyCopyable<I, O>
    tagged_pair<tag::in(I), tag::out(O)>
      replace_copy_if(I first, S last, O result, Pred pred, const T& new_value,
                      Proj proj = Proj{});
  template <InputRange Rng, class T, OutputIterator<const T&> O, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    requires IndirectlyCopyable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
      replace_copy_if(Rng&& rng, O result, Pred pred, const T& new_value,
                      Proj proj = Proj{});
}

// 30.6.6, fill
template<class ForwardIterator, class T>
  constexpr void fill(ForwardIterator first, ForwardIterator last, const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  void fill(ExecutionPolicy&& exec, // see 30.4.5
            ForwardIterator first, ForwardIterator last, const T& value);
template<class OutputIterator, class Size, class T>
  constexpr OutputIterator fill_n(OutputIterator first, Size n, const T& value);
template<class ExecutionPolicy, class ForwardIterator,
         class Size, class T>
  ForwardIterator fill_n(ExecutionPolicy&& exec, // see 30.4.5
                         ForwardIterator first, Size n, const T& value);
namespace ranges {
  template <class T, OutputIterator<const T&> O, Sentinel<O> S>
    O fill(O first, S last, const T& value);
  template <class T, OutputRange<const T&> Rng>
    safe_iterator_t<Rng>
```

```
      fill(Rng&& rng, const T& value);
  template <class T, OutputIterator<const T&> O>
    O fill_n(O first, iter_difference_t<O> n, const T& value);
}

// 30.6.7, generate
template<class ForwardIterator, class Generator>
  constexpr void generate(ForwardIterator first, ForwardIterator last,
                          Generator gen);
template<class ExecutionPolicy, class ForwardIterator, class Generator>
  void generate(ExecutionPolicy&& exec, // see 30.4.5
                ForwardIterator first, ForwardIterator last,
                Generator gen);
template<class OutputIterator, class Size, class Generator>
  constexpr OutputIterator generate_n(OutputIterator first, Size n, Generator gen);
template<class ExecutionPolicy, class ForwardIterator, class Size, class Generator>
  ForwardIterator generate_n(ExecutionPolicy&& exec, // see 30.4.5
                             ForwardIterator first, Size n, Generator gen);
namespace ranges {
  template <Iterator O, Sentinel<O> S, CopyConstructible F>
      requires Invocable<F&> && Writable<O, invoke_result_t<F&>>
    O generate(O first, S last, F gen);
  template <class Rng, CopyConstructible F>
      requires Invocable<F&> && OutputRange<Rng, invoke_result_t<F&>>
    safe_iterator_t<Rng>
      generate(Rng&& rng, F gen);
  template <Iterator O, CopyConstructible F>
      requires Invocable<F&> && Writable<O, invoke_result_t<F&>>
    O generate_n(O first, iter_difference_t<O> n, F gen);
}

// 30.6.8, remove
template<class ForwardIterator, class T>
  constexpr ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                                   const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  ForwardIterator remove(ExecutionPolicy&& exec, // see 30.4.5
                         ForwardIterator first, ForwardIterator last,
                         const T& value);
template<class ForwardIterator, class Predicate>
  constexpr ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                                      Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator remove_if(ExecutionPolicy&& exec, // see 30.4.5
                            ForwardIterator first, ForwardIterator last,
                            Predicate pred);
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity>
    requires Permutable<I> &&
      IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T*>
    I remove(I first, S last, const T& value, Proj proj = Proj{});
  template <ForwardRange Rng, class T, class Proj = identity>
    requires Permutable<iterator_t<Rng>> &&
      IndirectRelation<ranges::equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
    safe_iterator_t<Rng>
```

```
      remove(Rng&& rng, const T& value, Proj proj = Proj{});
  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires Permutable<I>
    I remove_if(I first, S last, Pred pred, Proj proj = Proj{});
  template <ForwardRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    requires Permutable<iterator_t<Rng>>
    safe_iterator_t<Rng>
      remove_if(Rng&& rng, Pred pred, Proj proj = Proj{});
}
template<class InputIterator, class OutputIterator, class T>
  constexpr OutputIterator
    remove_copy(InputIterator first, InputIterator last,
                OutputIterator result, const T& value);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class T>
  ForwardIterator2
    remove_copy(ExecutionPolicy&& exec, // see 30.4.5
                ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result, const T& value);
template<class InputIterator, class OutputIterator, class Predicate>
  constexpr OutputIterator
    remove_copy_if(InputIterator first, InputIterator last,
                   OutputIterator result, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate>
  ForwardIterator2
    remove_copy_if(ExecutionPolicy&& exec, // see 30.4.5
                   ForwardIterator1 first, ForwardIterator1 last,
                   ForwardIterator2 result, Predicate pred);
namespace ranges {
  template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class T,
      class Proj = identity>
    requires IndirectlyCopyable<I, O> &&
      IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T*>
    tagged_pair<tag::in(I), tag::out(O)>
      remove_copy(I first, S last, O result, const T& value, Proj proj = Proj{});
  template <InputRange Rng, WeaklyIncrementable O, class T, class Proj = identity>
    requires IndirectlyCopyable<iterator_t<Rng>, O> &&
      IndirectRelation<ranges::equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
      remove_copy(Rng&& rng, O result, const T& value, Proj proj = Proj{});
  template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
      class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires IndirectlyCopyable<I, O>
    tagged_pair<tag::in(I), tag::out(O)>
      remove_copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});
  template <InputRange Rng, WeaklyIncrementable O, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    requires IndirectlyCopyable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
      remove_copy_if(Rng&& rng, O result, Pred pred, Proj proj = Proj{});
}
```

```
// 30.6.9, unique
template<class ForwardIterator>
  constexpr ForwardIterator unique(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class BinaryPredicate>
  constexpr ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                                   BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator unique(ExecutionPolicy&& exec, // see 30.4.5
                         ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class BinaryPredicate>
  ForwardIterator unique(ExecutionPolicy&& exec, // see 30.4.5
                         ForwardIterator first, ForwardIterator last,
                         BinaryPredicate pred);
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectRelation<projected<I, Proj>> R = ranges::equal_to<>>
    requires Permutable<I>
    I unique(I first, S last, R comp = R{}, Proj proj = Proj{});
  template <ForwardRange Rng, class Proj = identity,
      IndirectRelation<projected<iterator_t<Rng>, Proj>> R = ranges::equal_to<>>
    requires Permutable<iterator_t<Rng>>
    safe_iterator_t<Rng>
      unique(Rng&& rng, R comp = R{}, Proj proj = Proj{});
}
template<class InputIterator, class OutputIterator>
  constexpr OutputIterator
    unique_copy(InputIterator first, InputIterator last,
                OutputIterator result);
template<class InputIterator, class OutputIterator, class BinaryPredicate>
  constexpr OutputIterator
    unique_copy(InputIterator first, InputIterator last,
                OutputIterator result, BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2
    unique_copy(ExecutionPolicy&& exec, // see 30.4.5
                ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  ForwardIterator2
    unique_copy(ExecutionPolicy&& exec, // see 30.4.5
                ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result, BinaryPredicate pred);
namespace ranges {
  template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
      class Proj = identity, IndirectRelation<projected<I, Proj>> R = ranges::equal_to<>>
    requires IndirectlyCopyable<I, O> &&
      (ForwardIterator<I> ||
      (InputIterator<O> && Same<iter_value_t<I>, iter_value_t<O>>) ||
      IndirectlyCopyableStorable<I, O>)
    tagged_pair<tag::in(I), tag::out(O)>
      unique_copy(I first, S last, O result, R comp = R{}, Proj proj = Proj{});
  template <InputRange Rng, WeaklyIncrementable O, class Proj = identity,
      IndirectRelation<projected<iterator_t<Rng>, Proj>> R = ranges::equal_to<>>
    requires IndirectlyCopyable<iterator_t<Rng>, O> &&
```

```
      (ForwardIterator<iterator_t<Rng>> ||
      (InputIterator<O> && Same<iter_value_t<iterator_t<Rng>>, iter_value_t<O>>) ||
      IndirectlyCopyableStorable<iterator_t<Rng>, O>)
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
      unique_copy(Rng&& rng, O result, R comp = R{}, Proj proj = Proj{});
}

// 30.6.10, reverse
template<class BidirectionalIterator>
  void reverse(BidirectionalIterator first, BidirectionalIterator last);
template<class ExecutionPolicy, class BidirectionalIterator>
  void reverse(ExecutionPolicy&& exec, // see 30.4.5
               BidirectionalIterator first, BidirectionalIterator last);
namespace ranges {
  template <BidirectionalIterator I, Sentinel<I> S>
    requires Permutable<I>
    I reverse(I first, S last);
  template <BidirectionalRange Rng>
    requires Permutable<iterator_t<Rng>>
    safe_iterator_t<Rng>
      reverse(Rng&& rng);
}
template<class BidirectionalIterator, class OutputIterator>
  constexpr OutputIterator
    reverse_copy(BidirectionalIterator first, BidirectionalIterator last,
                 OutputIterator result);
template<class ExecutionPolicy, class BidirectionalIterator, class ForwardIterator>
  ForwardIterator
    reverse_copy(ExecutionPolicy&& exec, // see 30.4.5
                 BidirectionalIterator first, BidirectionalIterator last,
                 ForwardIterator result);
namespace ranges {
  template <BidirectionalIterator I, Sentinel<I> S, WeaklyIncrementable O>
    requires IndirectlyCopyable<I, O>
    tagged_pair<tag::in(I), tag::out(O)> reverse_copy(I first, S last, O result);
  template <BidirectionalRange Rng, WeaklyIncrementable O>
    requires IndirectlyCopyable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
      reverse_copy(Rng&& rng, O result);
}

// 30.6.11, rotate
template<class ForwardIterator>
  ForwardIterator rotate(ForwardIterator first,
                         ForwardIterator middle,
                         ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator rotate(ExecutionPolicy&& exec, // see 30.4.5
                         ForwardIterator first,
                         ForwardIterator middle,
                         ForwardIterator last);
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S>
    requires Permutable<I>
    tagged_pair<tag::begin(I), tag::end(I)>
```

```
      rotate(I first, I middle, S last);
  template <ForwardRange Rng>
    requires Permutable<iterator_t<Rng>>
    tagged_pair<tag::begin(safe_iterator_t<Rng>),
                tag::end(safe_iterator_t<Rng>)>
      rotate(Rng&& rng, iterator_t<Rng> middle);
}
template<class ForwardIterator, class OutputIterator>
  constexpr OutputIterator
    rotate_copy(ForwardIterator first, ForwardIterator middle,
                ForwardIterator last, OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2
    rotate_copy(ExecutionPolicy&& exec, // see 30.4.5
                ForwardIterator1 first, ForwardIterator1 middle,
                ForwardIterator1 last, ForwardIterator2 result);
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, WeaklyIncrementable O>
    requires IndirectlyCopyable<I, O>
    tagged_pair<tag::in(I), tag::out(O)>
      rotate_copy(I first, I middle, S last, O result);
  template <ForwardRange Rng, WeaklyIncrementable O>
    requires IndirectlyCopyable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
      rotate_copy(Rng&& rng, iterator_t<Rng> middle, O result);
}

// 30.6.12, sample
template<class PopulationIterator, class SampleIterator,
         class Distance, class UniformRandomBitGenerator>
  SampleIterator sample(PopulationIterator first, PopulationIterator last,
                        SampleIterator out, Distance n,
                        UniformRandomBitGenerator&& g);

// 30.6.13, shuffle
template<class RandomAccessIterator, class UniformRandomBitGenerator>
  void shuffle(RandomAccessIterator first,
               RandomAccessIterator last,
               UniformRandomBitGenerator&& g);
namespace ranges {
  template <RandomAccessIterator I, Sentinel<I> S, class Gen>
    requires Permutable<I> &&
      UniformRandomBitGenerator<remove_reference_t<Gen>> &&
      ConvertibleTo<invoke_result_t<Gen&>, iter_difference_t<I>>
    I shuffle(I first, S last, Gen&& g);
  template <RandomAccessRange Rng, class Gen>
    requires Permutable<I> &&
      UniformRandomBitGenerator<remove_reference_t<Gen>> &&
      ConvertibleTo<invoke_result_t<Gen&>, iter_difference_t<I>>
    safe_iterator_t<Rng>
      shuffle(Rng&& rng, Gen&& g);
}

// 30.7.4, partitions
template<class InputIterator, class Predicate>
```

```
  constexpr bool is_partitioned(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  bool is_partitioned(ExecutionPolicy&& exec, // see 30.4.5
                      ForwardIterator first, ForwardIterator last, Predicate pred);
namespace ranges {
  template <InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    bool is_partitioned(I first, S last, Pred pred, Proj proj = Proj{});

  template <InputRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    bool is_partitioned(Rng&& rng, Pred pred, Proj proj = Proj{});
}

template<class ForwardIterator, class Predicate>
  ForwardIterator partition(ForwardIterator first,
                            ForwardIterator last,
                            Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator partition(ExecutionPolicy&& exec, // see 30.4.5
                            ForwardIterator first,
                            ForwardIterator last,
                            Predicate pred);
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires Permutable<I>
    I partition(I first, S last, Pred pred, Proj proj = Proj{});

  template <ForwardRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    requires Permutable<iterator_t<Rng>>
    safe_iterator_t<Rng>
      partition(Rng&& rng, Pred pred, Proj proj = Proj{});
}
template<class BidirectionalIterator, class Predicate>
  BidirectionalIterator stable_partition(BidirectionalIterator first,
                                         BidirectionalIterator last,
                                         Predicate pred);
template<class ExecutionPolicy, class BidirectionalIterator, class Predicate>
  BidirectionalIterator stable_partition(ExecutionPolicy&& exec, // see 30.4.5
                                         BidirectionalIterator first,
                                         BidirectionalIterator last,
                                         Predicate pred);
namespace ranges {
  template <BidirectionalIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires Permutable<I>
    I stable_partition(I first, S last, Pred pred, Proj proj = Proj{});

  template <BidirectionalRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    requires Permutable<iterator_t<Rng>>
    safe_iterator_t<Rng>
      stable_partition(Rng&& rng, Pred pred, Proj proj = Proj{});
```

```
}
template<class InputIterator, class OutputIterator1,
         class OutputIterator2, class Predicate>
  constexpr pair<OutputIterator1, OutputIterator2>
    partition_copy(InputIterator first, InputIterator last,
                   OutputIterator1 out_true, OutputIterator2 out_false,
                   Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class ForwardIterator1,
         class ForwardIterator2, class Predicate>
  pair<ForwardIterator1, ForwardIterator2>
    partition_copy(ExecutionPolicy&& exec, // see 30.4.5
                   ForwardIterator first, ForwardIterator last,
                   ForwardIterator1 out_true, ForwardIterator2 out_false,
                   Predicate pred);
namespace ranges {
  template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O1, WeaklyIncrementable O2,
      class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires IndirectlyCopyable<I, O1> && IndirectlyCopyable<I, O2>
    tagged_tuple<tag::in(I), tag::out1(O1), tag::out2(O2)>
      partition_copy(I first, S last, O1 out_true, O2 out_false, Pred pred,
                     Proj proj = Proj{});

  template <InputRange Rng, WeaklyIncrementable O1, WeaklyIncrementable O2,
      class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    requires IndirectlyCopyable<iterator_t<Rng>, O1> &&
      IndirectlyCopyable<iterator_t<Rng>, O2>
    tagged_tuple<tag::in(safe_iterator_t<Rng>), tag::out1(O1), tag::out2(O2)>
      partition_copy(Rng&& rng, O1 out_true, O2 out_false, Pred pred, Proj proj = Proj{});
}
template<class ForwardIterator, class Predicate>
  constexpr ForwardIterator
    partition_point(ForwardIterator first, ForwardIterator last,
                    Predicate pred);
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    I partition_point(I first, S last, Pred pred, Proj proj = Proj{});
  template <ForwardRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    safe_iterator_t<Rng>
      partition_point(Rng&& rng, Pred pred, Proj proj = Proj{});
}

// 30.7, sorting and related operations
// 30.7.1, sorting
template<class RandomAccessIterator>
  void sort(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void sort(RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator>
  void sort(ExecutionPolicy&& exec, // see 30.4.5
            RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
```

```
  void sort(ExecutionPolicy&& exec, // see 30.4.5
            RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);
namespace ranges {
  template <RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    I sort(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

  template <RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    safe_iterator_t<Rng>
      sort(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class RandomAccessIterator>
  void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator>
  void stable_sort(ExecutionPolicy&& exec, // see 30.4.5
                   RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  void stable_sort(ExecutionPolicy&& exec, // see 30.4.5
                   RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);
namespace ranges {
  template <RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    I stable_sort(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    safe_iterator_t<Rng>
      stable_sort(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class RandomAccessIterator>
  void partial_sort(RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void partial_sort(RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last, Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator>
  void partial_sort(ExecutionPolicy&& exec, // see 30.4.5
                    RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  void partial_sort(ExecutionPolicy&& exec, // see 30.4.5
                    RandomAccessIterator first,
                    RandomAccessIterator middle,
```

```
                    RandomAccessIterator last, Compare comp);
namespace ranges {
  template <RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    I partial_sort(I first, I middle, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    safe_iterator_t<Rng>
      partial_sort(Rng&& rng, iterator_t<Rng> middle, Comp comp = Comp{},
                   Proj proj = Proj{});
}
template<class InputIterator, class RandomAccessIterator>
  RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last);
template<class InputIterator, class RandomAccessIterator, class Compare>
  RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last,
                      Compare comp);
template<class ExecutionPolicy, class ForwardIterator, class RandomAccessIterator>
  RandomAccessIterator
    partial_sort_copy(ExecutionPolicy&& exec,  // see 30.4.5
                      ForwardIterator first, ForwardIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last);
template<class ExecutionPolicy, class ForwardIterator, class RandomAccessIterator,
         class Compare>
  RandomAccessIterator
    partial_sort_copy(ExecutionPolicy&& exec,  // see 30.4.5
                      ForwardIterator first, ForwardIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last,
                      Compare comp);
namespace ranges {
  template <InputIterator I1, Sentinel<I1> S1, RandomAccessIterator I2, Sentinel<I2> S2,
      class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyCopyable<I1, I2> && Sortable<I2, Comp, Proj2> &&
        IndirectStrictWeakOrder<Comp, projected<I1, Proj1>, projected<I2, Proj2>>
    I2 partial_sort_copy(I1 first, S1 last, I2 result_first, S2 result_last,
                         Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template <InputRange Rng1, RandomAccessRange Rng2, class Comp = ranges::less<>,
      class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyCopyable<iterator_t<Rng1>, iterator_t<Rng2>> &&
        Sortable<iterator_t<Rng2>, Comp, Proj2> &&
        IndirectStrictWeakOrder<Comp, projected<iterator_t<Rng1>, Proj1>,
          projected<iterator_t<Rng2>, Proj2>>
    safe_iterator_t<Rng2>
      partial_sort_copy(Rng1&& rng, Rng2&& result_rng, Comp comp = Comp{},
                        Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
template<class ForwardIterator>
```

```
  constexpr bool is_sorted(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  constexpr bool is_sorted(ForwardIterator first, ForwardIterator last,
                           Compare comp);
template<class ExecutionPolicy, class ForwardIterator>
  bool is_sorted(ExecutionPolicy&& exec, // see 30.4.5
                 ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  bool is_sorted(ExecutionPolicy&& exec, // see 30.4.5
                 ForwardIterator first, ForwardIterator last,
                 Compare comp);
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less<>>
    bool is_sorted(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <ForwardRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    bool is_sorted(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
template<class ForwardIterator>
  constexpr ForwardIterator
    is_sorted_until(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  constexpr ForwardIterator
    is_sorted_until(ForwardIterator first, ForwardIterator last,
                    Compare comp);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator
    is_sorted_until(ExecutionPolicy&& exec, // see 30.4.5
                    ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  ForwardIterator
    is_sorted_until(ExecutionPolicy&& exec, // see 30.4.5
                    ForwardIterator first, ForwardIterator last,
                    Compare comp);
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less<>>
    I is_sorted_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <ForwardRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    safe_iterator_t<Rng>
      is_sorted_until(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

// 30.7.2, Nth element
template<class RandomAccessIterator>
  void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last, Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator>
  void nth_element(ExecutionPolicy&& exec, // see 30.4.5
                   RandomAccessIterator first, RandomAccessIterator nth,
```

```
                   RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  void nth_element(ExecutionPolicy&& exec, // see 30.4.5
                   RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last, Compare comp);
namespace ranges {
  template <RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    I nth_element(I first, I nth, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    safe_iterator_t<Rng>
      nth_element(Rng&& rng, iterator_t<Rng> nth, Comp comp = Comp{}, Proj proj = Proj{});
}

// 30.7.3, binary search
template<class ForwardIterator, class T>
  constexpr ForwardIterator
    lower_bound(ForwardIterator first, ForwardIterator last,
                const T& value);
template<class ForwardIterator, class T, class Compare>
  constexpr ForwardIterator
    lower_bound(ForwardIterator first, ForwardIterator last,
                const T& value, Compare comp);
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
      IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = ranges::less<>>
    I lower_bound(I first, S last, const T& value, Comp comp = Comp{},
                  Proj proj = Proj{});
  template <ForwardRange Rng, class T, class Proj = identity,
      IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    safe_iterator_t<Rng>
      lower_bound(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class ForwardIterator, class T>
  constexpr ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last,
                const T& value);
template<class ForwardIterator, class T, class Compare>
  constexpr ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last,
                const T& value, Compare comp);
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
      IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = ranges::less<>>
    I upper_bound(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
  template <ForwardRange Rng, class T, class Proj = identity,
      IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    safe_iterator_t<Rng>
      upper_bound(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
}
```

```
template<class ForwardIterator, class T>
  constexpr pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first, ForwardIterator last,
                const T& value);
template<class ForwardIterator, class T, class Compare>
  constexpr pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first, ForwardIterator last,
                const T& value, Compare comp);
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
      IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = ranges::less<>>
    tagged_pair<tag::begin(I), tag::end(I)>
      equal_range(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
  template <ForwardRange Rng, class T, class Proj = identity,
      IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    tagged_pair<tag::begin(safe_iterator_t<Rng>),
                tag::end(safe_iterator_t<Rng>)>
      equal_range(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class ForwardIterator, class T>
  constexpr bool
    binary_search(ForwardIterator first, ForwardIterator last,
                  const T& value);
template<class ForwardIterator, class T, class Compare>
  constexpr bool
    binary_search(ForwardIterator first, ForwardIterator last,
                  const T& value, Compare comp);
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
      IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = ranges::less<>>
    bool binary_search(I first, S last, const T& value, Comp comp = Comp{},
                       Proj proj = Proj{});
  template <ForwardRange Rng, class T, class Proj = identity,
      IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    bool binary_search(Rng&& rng, const T& value, Comp comp = Comp{},
                       Proj proj = Proj{});
}

// 30.7.5, merge
template<class InputIterator1, class InputIterator2, class OutputIterator>
  constexpr OutputIterator
    merge(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator,
         class Compare>
  constexpr OutputIterator
    merge(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  ForwardIterator
    merge(ExecutionPolicy&& exec, // see 30.4.5
```

```
          ForwardIterator1 first1, ForwardIterator1 last1,
          ForwardIterator2 first2, ForwardIterator2 last2,
          ForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  ForwardIterator
    merge(ExecutionPolicy&& exec, // see 30.4.5
          ForwardIterator1 first1, ForwardIterator1 last1,
          ForwardIterator2 first2, ForwardIterator2 last2,
          ForwardIterator result, Compare comp);
namespace ranges {
  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      WeaklyIncrementable O, class Comp = ranges::less<>, class Proj1 = identity,
      class Proj2 = identity>
    requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
    tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
      merge(I1 first1, S1 last1, I2 first2, S2 last2, O result,
            Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O, class Comp = ranges::less<>,
      class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
    tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
                 tag::in2(safe_iterator_t<Rng2>),
                 tag::out(O)>
      merge(Rng1&& rng1, Rng2&& rng2, O result,
            Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

template<class BidirectionalIterator>
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last);
template<class BidirectionalIterator, class Compare>
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last, Compare comp);
template<class ExecutionPolicy, class BidirectionalIterator>
  void inplace_merge(ExecutionPolicy&& exec, // see 30.4.5
                     BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last);
template<class ExecutionPolicy, class BidirectionalIterator, class Compare>
  void inplace_merge(ExecutionPolicy&& exec, // see 30.4.5
                     BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last, Compare comp);
namespace ranges {
  template <BidirectionalIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    I inplace_merge(I first, I middle, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <BidirectionalRange Rng, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    safe_iterator_t<Rng>
      inplace_merge(Rng&& rng, iterator_t<Rng> middle, Comp comp = Comp{},
```

```
                      Proj proj = Proj{});
}

// 30.7.6, set operations
template<class InputIterator1, class InputIterator2>
  constexpr bool includes(InputIterator1 first1, InputIterator1 last1,
                          InputIterator2 first2, InputIterator2 last2);
template<class InputIterator1, class InputIterator2, class Compare>
  constexpr bool includes(InputIterator1 first1, InputIterator1 last1,
                          InputIterator2 first2, InputIterator2 last2,
                          Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  bool includes(ExecutionPolicy&& exec, // see 30.4.5
                ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Compare>
  bool includes(ExecutionPolicy&& exec, // see 30.4.5
                ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2, ForwardIterator2 last2,
                Compare comp);
namespace ranges {
  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      class Proj1 = identity, class Proj2 = identity,
      IndirectStrictWeakOrder<projected<I1, Proj1>, projected<I2, Proj2>> Comp = ranges::less<>>
    bool includes(I1 first1, S1 last1, I2 first2, S2 last2, Comp comp = Comp{},
                  Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template <InputRange Rng1, InputRange Rng2, class Proj1 = identity,
      class Proj2 = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng1>, Proj1>,
        projected<iterator_t<Rng2>, Proj2>> Comp = ranges::less<>>
    bool includes(Rng1&& rng1, Rng2&& rng2, Comp comp = Comp{},
                  Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

template<class InputIterator1, class InputIterator2, class OutputIterator>
  constexpr OutputIterator
    set_union(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
  constexpr OutputIterator
              set_union(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  ForwardIterator
    set_union(ExecutionPolicy&& exec, // see 30.4.5
              ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator2 last2,
              ForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  ForwardIterator
```

```
    set_union(ExecutionPolicy&& exec, // see 30.4.5
              ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator2 last2,
              ForwardIterator result, Compare comp);
namespace ranges {
  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      WeaklyIncrementable O, class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
    tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
      set_union(I1 first1, S1 last1, I2 first2, S2 last2, O result, Comp comp = Comp{},
                Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
      class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
    tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
                 tag::in2(safe_iterator_t<Rng2>),
                 tag::out(O)>
      set_union(Rng1&& rng1, Rng2&& rng2, O result, Comp comp = Comp{},
                Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

template<class InputIterator1, class InputIterator2, class OutputIterator>
  constexpr OutputIterator
    set_intersection(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
  constexpr OutputIterator
    set_intersection(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  ForwardIterator
    set_intersection(ExecutionPolicy&& exec, // see 30.4.5
                     ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2, ForwardIterator2 last2,
                     ForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  ForwardIterator
    set_intersection(ExecutionPolicy&& exec, // see 30.4.5
                     ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2, ForwardIterator2 last2,
                     ForwardIterator result, Compare comp);
namespace ranges {
  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      WeaklyIncrementable O, class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
    O set_intersection(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                       Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
      class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
    O set_intersection(Rng1&& rng1, Rng2&& rng2, O result,
```

```
                        Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

template<class InputIterator1, class InputIterator2, class OutputIterator>
  constexpr OutputIterator
    set_difference(InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, InputIterator2 last2,
                   OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
  constexpr OutputIterator
    set_difference(InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, InputIterator2 last2,
                   OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  ForwardIterator
    set_difference(ExecutionPolicy&& exec, // see 30.4.5
                   ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2, ForwardIterator2 last2,
                   ForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  ForwardIterator
    set_difference(ExecutionPolicy&& exec, // see 30.4.5
                   ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2, ForwardIterator2 last2,
                   ForwardIterator result, Compare comp);
namespace ranges {
  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      WeaklyIncrementable O, class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
    tagged_pair<tag::in1(I1), tag::out(O)>
      set_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                     Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
      class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
    tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::out(O)>
      set_difference(Rng1&& rng1, Rng2&& rng2, O result,
                     Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

template<class InputIterator1, class InputIterator2, class OutputIterator>
  constexpr OutputIterator
    set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
  constexpr OutputIterator
    set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  ForwardIterator
```

```
    set_symmetric_difference(ExecutionPolicy&& exec, // see 30.4.5
                             ForwardIterator1 first1, ForwardIterator1 last1,
                             ForwardIterator2 first2, ForwardIterator2 last2,
                             ForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  ForwardIterator
    set_symmetric_difference(ExecutionPolicy&& exec, // see 30.4.5
                             ForwardIterator1 first1, ForwardIterator1 last1,
                             ForwardIterator2 first2, ForwardIterator2 last2,
                             ForwardIterator result, Compare comp);
namespace ranges {
  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      WeaklyIncrementable O, class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
    tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
      set_symmetric_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                               Comp comp = Comp{}, Proj1 proj1 = Proj1{},
                               Proj2 proj2 = Proj2{});
  template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
      class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
    tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
                 tag::in2(safe_iterator_t<Rng2>),
                 tag::out(O)>
      set_symmetric_difference(Rng1&& rng1, Rng2&& rng2, O result, Comp comp = Comp{},
                               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

// 30.7.7, heap operations
template<class RandomAccessIterator>
  void push_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void push_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
namespace ranges {
  template <RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    I push_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    safe_iterator_t<Rng>
      push_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class RandomAccessIterator>
  void pop_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
                Compare comp);
namespace ranges {
  template <RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
```

```
    requires Sortable<I, Comp, Proj>
    I pop_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    safe_iterator_t<Rng>
      pop_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class RandomAccessIterator>
  void make_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void make_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
namespace ranges {
  template <RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    I make_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    safe_iterator_t<Rng>
      make_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class RandomAccessIterator>
  void sort_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
namespace ranges {
  template <RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    I sort_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    safe_iterator_t<Rng>
      sort_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class RandomAccessIterator>
  constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last,
                         Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator>
  bool is_heap(ExecutionPolicy&& exec, // see 30.4.5
               RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  bool is_heap(ExecutionPolicy&& exec, // see 30.4.5
               RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
namespace ranges {
  template <RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
```

```
      IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less<>>
    bool is_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <RandomAccessRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    bool is_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
template<class RandomAccessIterator>
  constexpr RandomAccessIterator
    is_heap_until(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  constexpr RandomAccessIterator
    is_heap_until(RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator>
  RandomAccessIterator
    is_heap_until(ExecutionPolicy&& exec, // see 30.4.5
                  RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  RandomAccessIterator
    is_heap_until(ExecutionPolicy&& exec, // see 30.4.5
                  RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp);
namespace ranges {
  template <RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
      IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less<>>
    I is_heap_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <RandomAccessRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    safe_iterator_t<Rng>
      is_heap_until(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

// 30.7.8, minimum and maximum
template<class T> constexpr const T& min(const T& a, const T& b);
template<class T, class Compare>
  constexpr const T& min(const T& a, const T& b, Compare comp);
template<class T>
  constexpr T min(initializer_list<T> t);
template<class T, class Compare>
  constexpr T min(initializer_list<T> t, Compare comp);
namespace ranges {
  template <class T, class Proj = identity,
      IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr const T& min(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
  template <Copyable T, class Proj = identity,
      IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr T min(initializer_list<T> t, Comp comp = Comp{}, Proj proj = Proj{});
  template <InputRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    requires Copyable<iter_value_t<iterator_t<Rng>>>
    iter_value_t<iterator_t<Rng>>
      min(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

```
template<class T> constexpr const T& max(const T& a, const T& b);
template<class T, class Compare>
  constexpr const T& max(const T& a, const T& b, Compare comp);
template<class T>
  constexpr T max(initializer_list<T> t);
template<class T, class Compare>
  constexpr T max(initializer_list<T> t, Compare comp);
namespace ranges {
  template <class T, class Proj = identity,
      IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr const T& max(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
  template <Copyable T, class Proj = identity,
      IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr T max(initializer_list<T> t, Comp comp = Comp{}, Proj proj = Proj{});
  template <InputRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    requires Copyable<iter_value_t<iterator_t<Rng>>>
    iter_value_t<iterator_t<Rng>>
      max(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class T> constexpr pair<const T&, const T&> minmax(const T& a, const T& b);
template<class T, class Compare>
  constexpr pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
template<class T>
  constexpr pair<T, T> minmax(initializer_list<T> t);
template<class T, class Compare>
  constexpr pair<T, T> minmax(initializer_list<T> t, Compare comp);
namespace ranges {
  template <class T, class Proj = identity,
      IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr tagged_pair<tag::min(const T&), tag::max(const T&)>
      minmax(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
  template <Copyable T, class Proj = identity,
      IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr tagged_pair<tag::min(T), tag::max(T)>
      minmax(initializer_list<T> t, Comp comp = Comp{}, Proj proj = Proj{});
  template <InputRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    requires Copyable<iter_value_t<iterator_t<Rng>>>
    tagged_pair<tag::min(iter_value_t<iterator_t<Rng>>),
            tag::max(iter_value_t<iterator_t<Rng>>)>
      minmax(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class ForwardIterator>
  constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
                                        Compare comp);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator min_element(ExecutionPolicy&& exec, // see 30.4.5
                              ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
```

```
  ForwardIterator min_element(ExecutionPolicy&& exec, // see 30.4.5
                              ForwardIterator first, ForwardIterator last,
                              Compare comp);
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less<>>
    I min_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <ForwardRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    safe_iterator_t<Rng>
      min_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
template<class ForwardIterator>
  constexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  constexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
                                        Compare comp);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator max_element(ExecutionPolicy&& exec, // see 30.4.5
                              ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  ForwardIterator max_element(ExecutionPolicy&& exec, // see 30.4.5
                              ForwardIterator first, ForwardIterator last,
                              Compare comp);
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less<>>
    I max_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <ForwardRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    safe_iterator_t<Rng>
      max_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
template<class ForwardIterator>
  constexpr pair<ForwardIterator, ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  constexpr pair<ForwardIterator, ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last, Compare comp);
template<class ExecutionPolicy, class ForwardIterator>
  pair<ForwardIterator, ForwardIterator>
    minmax_element(ExecutionPolicy&& exec, // see 30.4.5
                   ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  pair<ForwardIterator, ForwardIterator>
    minmax_element(ExecutionPolicy&& exec, // see 30.4.5
                   ForwardIterator first, ForwardIterator last, Compare comp);
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less<>>
    tagged_pair<tag::min(I), tag::max(I)>
      minmax_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <ForwardRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
```

```
    tagged_pair<tag::min(safe_iterator_t<Rng>),
                tag::max(safe_iterator_t<Rng>)>
      minmax_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

// 30.7.9, bounded value
template<class T>
  constexpr const T& clamp(const T& v, const T& lo, const T& hi);
template<class T, class Compare>
  constexpr const T& clamp(const T& v, const T& lo, const T& hi, Compare comp);

// 30.7.10, lexicographical comparison
template<class InputIterator1, class InputIterator2>
  constexpr bool
    lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2);
template<class InputIterator1, class InputIterator2, class Compare>
  constexpr bool
    lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2,
                            Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  bool
    lexicographical_compare(ExecutionPolicy&& exec, // see 30.4.5
                            ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Compare>
  bool
    lexicographical_compare(ExecutionPolicy&& exec, // see 30.4.5
                            ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2,
                            Compare comp);
namespace ranges {
  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      class Proj1 = identity, class Proj2 = identity,
      IndirectStrictWeakOrder<projected<I1, Proj1>, projected<I2, Proj2>> Comp = ranges::less<>>
    bool
      lexicographical_compare(I1 first1, S1 last1, I2 first2, S2 last2,
                              Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template <InputRange Rng1, InputRange Rng2, class Proj1 = identity,
      class Proj2 = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng1>, Proj1>,
        projected<iterator_t<Rng2>, Proj2>> Comp = ranges::less<>>
    bool
      lexicographical_compare(Rng1&& rng1, Rng2&& rng2, Comp comp = Comp{},
                              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

// 30.7.11, three-way comparison algorithms
template<class T, class U>
  constexpr auto compare_3way(const T& a, const U& b);
template<class InputIterator1, class InputIterator2, class Cmp>
  constexpr auto
    lexicographical_compare_3way(InputIterator1 b1, InputIterator1 e1,
```

```
                              InputIterator2 b2, InputIterator2 e2,
                              Cmp comp)
      -> common_comparison_category_t<decltype(comp(*b1, *b2)), strong_ordering>;
  template<class InputIterator1, class InputIterator2>
    constexpr auto
      lexicographical_compare_3way(InputIterator1 b1, InputIterator1 e1,
                                   InputIterator2 b2, InputIterator2 e2);

  // 30.7.12, permutations
  template<class BidirectionalIterator>
    bool next_permutation(BidirectionalIterator first,
                          BidirectionalIterator last);
  template<class BidirectionalIterator, class Compare>
    bool next_permutation(BidirectionalIterator first,
                          BidirectionalIterator last, Compare comp);
  namespace ranges {
    template <BidirectionalIterator I, Sentinel<I> S, class Comp = ranges::less<>,
        class Proj = identity>
      requires Sortable<I, Comp, Proj>
      bool next_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
    template <BidirectionalRange Rng, class Comp = ranges::less<>,
        class Proj = identity>
      requires Sortable<iterator_t<Rng>, Comp, Proj>
      bool next_permutation(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
  }
  template<class BidirectionalIterator>
    bool prev_permutation(BidirectionalIterator first,
                          BidirectionalIterator last);
  template<class BidirectionalIterator, class Compare>
    bool prev_permutation(BidirectionalIterator first,
                          BidirectionalIterator last, Compare comp);
  namespace ranges {
    template <BidirectionalIterator I, Sentinel<I> S, class Comp = ranges::less<>,
        class Proj = identity>
      requires Sortable<I, Comp, Proj>
      bool prev_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
    template <BidirectionalRange Rng, class Comp = ranges::less<>,
        class Proj = identity>
      requires Sortable<iterator_t<Rng>, Comp, Proj>
      bool prev_permutation(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
  }
}
```

30.3 Algorithms requirements [algorithms.requirements]

1 All of the algorithms are separated from the particular implementations of data structures and are parameterized by iterator types. Because of this, they can work with program-defined data structures, as long as these data structures have iterator types satisfying the assumptions on the algorithms.

2 The function templates defined in the `std::ranges` namespace in this subclause are not found by argument-dependent name lookup (6.4.2). When found by unqualified (6.4.1) name lookup for the *postfix-expression* in a function call (), they inhibit argument-dependent name lookup.

[*Example:*

```
void foo() {
    using namespace std::ranges;
```

```
    std::vector<int> vec{1,2,3};
    find(begin(vec), end(vec), 2); // #1
}
```

The function call expression at `#1` invokes `std::ranges::find`, not `std::find`, despite that (a) the iterator type returned from `begin(vec)` and `end(vec)` may be associated with namespace `std` and (b) `std::find` is more specialized (17.6.6.2) than `std::ranges::find` since the former requires its first two parameters to have the same type. *— end example*]

3 For purposes of determining the existence of data races, algorithms shall not modify objects referenced through an iterator argument unless the specification requires such modification.

4 Throughout this Clause, where the template parameters are not constrained, the names of template parameters are used to express type requirements.

(4.1) — If an algorithm's template parameter is named `InputIterator`, `InputIterator1`, or `InputIterator2`, the template argument shall satisfy the requirements of ~~an~~a C++98 input iterator (28.3.5.2).

(4.2) — If an algorithm's template parameter is named `OutputIterator`, `OutputIterator1`, or `OutputIterator2`, the template argument shall satisfy the requirements of ~~an~~a C++98 output iterator (28.3.5.3).

(4.3) — If an algorithm's template parameter is named `ForwardIterator`, `ForwardIterator1`, or `ForwardIterator2`, the template argument shall satisfy the requirements of a C++98 forward iterator (28.3.5.4).

(4.4) — If an algorithm's template parameter is named `BidirectionalIterator`, `BidirectionalIterator1`, or `BidirectionalIterator2`, the template argument shall satisfy the requirements of a C++98 bidirectional iterator (28.3.5.5).

(4.5) — If an algorithm's template parameter is named `RandomAccessIterator`, `RandomAccessIterator1`, or `RandomAccessIterator2`, the template argument shall satisfy the requirements of a C++98 random-access iterator (28.3.5.6).

5 If an algorithm's *Effects:* element specifies that a value pointed to by any iterator passed as an argument is modified, then that algorithm has an additional type requirement: The type of that argument shall satisfy the requirements of a mutable iterator (28.3). [*Note:* This requirement does not affect arguments that are named `OutputIterator`, `OutputIterator1`, or `OutputIterator2`, because output iterators must always be mutable, neither does it affect arguments that are constrained, as the requirement for mutability is expressed explicitly. *— end note*]

6 Both in-place and copying versions are provided for certain algorithms.[3] When such a version is provided for *algorithm* it is called *algorithm*`_copy`. Algorithms that take predicates end with the suffix `_if` (which follows the suffix `_copy`).

7 ~~T~~When not otherwise constrained, the `Predicate` parameter is used whenever an algorithm expects a function object that, when applied to the result of dereferencing the corresponding iterator, returns a value testable as `true`. In other words, if an algorithm takes `Predicate pred` as its argument and `first` as its iterator argument, it should work correctly in the construct `pred(*first)` contextually converted to `bool`7. The function object `pred` shall not apply any non-constant function through the dereferenced iterator.

8 ~~T~~When not otherwise constrained, the `BinaryPredicate` parameter is used whenever an algorithm expects a function object that when applied to the result of dereferencing two corresponding iterators or to dereferencing an iterator and type `T` when `T` is part of the signature returns a value testable as `true`. In other

---

[3] The decision whether to include a copying version was usually based on complexity considerations. When the cost of doing the operation dominates the cost of copy, the copying version is not included. For example, `sort_copy` is not included because the cost of sorting is much more significant, and users might as well do `copy` followed by `sort`.

words, if an algorithm takes `BinaryPredicate binary_pred` as its argument and `first1` and `first2` as its iterator arguments, it should work correctly in the construct `binary_pred(*first1, *first2)` contextually converted to `bool`7. `BinaryPredicate` always takes the first iterator's `value_type` as its first argument, that is, in those cases when `T value` is part of the signature, it should work correctly in the construct `binary_pred(*first1, value)` contextually converted to `bool`7. `binary_pred` shall not apply any non-constant function through the dereferenced iterators.

9 [*Note:* Unless otherwise specified, algorithms that take function objects as arguments are permitted to copy those function objects freely. Programmers for whom object identity is important should consider using a wrapper class that points to a noncopied implementation object such as `reference_wrapper<T>`23.14.5, or some equivalent solution. — *end note*]

10 When the description of an algorithm gives an expression such as `*first == value` for a condition, the expression shall evaluate to either `true` or `false` in boolean contexts.

11 In the description of the algorithms operators `+` and `-` are used for some of the iterator categories for which they do not have to be defined. In these cases the semantics of `a+n` is the same as that of

```
X tmp = a;
advance(tmp, n);
return tmp;
```

and that of `b-a` is the same as of

```
return distance(a, b);
```

12 In the description of algorithm return values, sentinel values are sometimes returned where an iterator is expected. In these cases, the semantics are as if the sentinel is converted into an iterator as follows:

```
I tmp = first;
while(tmp != last)
  ++tmp;
return tmp;
```

13 Overloads of algorithms that take `Range` arguments (29.6.2) behave as if they are implemented by calling `range::begin` and `range::end` on the `Range` and dispatching to the overload that takes separate iterator and sentinel arguments.

14 The number and order of template parameters for algorithm declarations is unspecified, except where explicitly stated otherwise.

30.4 Parallel algorithms [algorithms.parallel]

1 This subclause describes components that C++ programs may use to perform operations on containers and other sequences in parallel.

30.4.1 Terms and definitions [algorithms.parallel.defns]

1 A *parallel algorithm* is a function template listed in this document with a template parameter named `ExecutionPolicy`.

2 Parallel algorithms access objects indirectly accessible via their arguments by invoking the following functions:

(2.1) — All operations of the categories of the iterators that the algorithm is instantiated with.

(2.2) — Operations on those sequence elements that are required by its specification.

(2.3) — User-provided function objects to be applied during the execution of the algorithm, if required by the specification.

(2.4) — Operations on those function objects required by the specification. [*Note:* See 30.1. — *end note*]

These functions are herein called *element access functions*. [*Example:* The `sort` function may invoke the following element access functions:

(2.5) — Operations of the random-access iterator of the actual template argument (as per 28.3.5.6), as implied by the name of the template parameter `RandomAccessIterator`.

(2.6) — The `swap` function on the elements of the sequence (as per the preconditions specified in 30.7.1.1).

(2.7) — The user-provided `Compare` function object.

— *end example*]

30.4.2 Requirements on user-provided function objects [algorithms.parallel.user]

1 Unless otherwise specified, function objects passed into parallel algorithms as objects of type `Predicate`, `BinaryPredicate`, `Compare`, `UnaryOperation`, `BinaryOperation`, `BinaryOperation1`, `BinaryOperation2`, and the operators used by the analogous overloads to these parallel algorithms that could be formed by the invocation with the specified default predicate or operation (where applicable) shall not directly or indirectly modify objects via their arguments, nor shall they rely on the identity of the provided objects.

30.4.3 Effect of execution policies on algorithm execution [algorithms.parallel.exec]

1 Parallel algorithms have template parameters named `ExecutionPolicy` which describe the manner in which the execution of these algorithms may be parallelized and the manner in which they apply the element access functions.

2 If an object is modified by an element access function, the algorithm will perform no other unsynchronized accesses to that object. The modifying element access functions are those which are specified as modifying the object. [*Note:* For example, `swap()`, `++`, `--`, `@=`, and assignments modify the object. For the assignment and `@=` operators, only the left argument is modified. — *end note*]

3 Unless otherwise stated, implementations may make arbitrary copies of elements (with type `T`) from sequences where `is_trivially_copy_constructible_v<T>` and `is_trivially_destructible_v<T>` are `true`. [*Note:* This implies that user-supplied function objects should not rely on object identity of arguments for such input sequences. Users for whom the object identity of the arguments to these function objects is important should consider using a wrapping iterator that returns a non-copied implementation object such as `reference_wrapper<T>`23.14.5 or some equivalent solution. — *end note*]

4 The invocations of element access functions in parallel algorithms invoked with an execution policy object of type `execution::sequenced_policy` all occur in the calling thread of execution. [*Note:* The invocations are not interleaved; see 6.8.1. — *end note*]

5 The invocations of element access functions in parallel algorithms invoked with an execution policy object of type `execution::parallel_policy` are permitted to execute in either the invoking thread of execution or in a thread of execution implicitly created by the library to support parallel algorithm execution. If the threads of execution created by `thread` provide concurrent forward progress guarantees, then a thread of execution implicitly created by the library will provide parallel forward progress guarantees; otherwise, the provided forward progress guarantee is implementation-defined. Any such invocations executing in the same thread of execution are indeterminately sequenced with respect to each other. [*Note:* It is the caller's responsibility to ensure that the invocation does not introduce data races or deadlocks. — *end note*] [*Example:*

```
int a[] = {0,1};
std::vector<int> v;
std::for_each(std::execution::par, std::begin(a), std::end(a), [&](int i) {
  v.push_back(i*2+1); // incorrect: data race
});
```

The program above has a data race because of the unsynchronized access to the container `v`. *— end example*] [*Example:*

```
std::atomic<int> x{0};
int a[] = {1,2};
std::for_each(std::execution::par, std::begin(a), std::end(a), [&](int) {
  x.fetch_add(1, std::memory_order::relaxed);
  // spin wait for another iteration to change the value of x
  while (x.load(std::memory_order::relaxed) == 1) { } // incorrect: assumes execution order
});
```

The above example depends on the order of execution of the iterations, and will not terminate if both iterations are executed sequentially on the same thread of execution. *— end example*] [*Example:*

```
int x = 0;
std::mutex m;
int a[] = {1,2};
std::for_each(std::execution::par, std::begin(a), std::end(a), [&](int) {
  std::lock_guard<mutex> guard(m);
  ++x;
});
```

The above example synchronizes access to object `x` ensuring that it is incremented correctly. *— end example*]

6 The invocations of element access functions in parallel algorithms invoked with an execution policy of type `execution::parallel_unsequenced_policy` are permitted to execute in an unordered fashion in unspecified threads of execution, and unsequenced with respect to one another within each thread of execution. These threads of execution are either the invoking thread of execution or threads of execution implicitly created by the library; the latter will provide weakly parallel forward progress guarantees. [*Note:* This means that multiple function object invocations may be interleaved on a single thread of execution, which overrides the usual guarantee from 6.8.1 that function executions do not interleave with one another. *— end note*] Since `execution::parallel_unsequenced_policy` allows the execution of element access functions to be interleaved on a single thread of execution, blocking synchronization, including the use of mutexes, risks deadlock. Thus, the synchronization with `execution::parallel_unsequenced_policy` is restricted as follows: A standard library function is *vectorization-unsafe* if it is specified to synchronize with another function invocation, or another function invocation is specified to synchronize with it, and if it is not a memory allocation or deallocation function. Vectorization-unsafe standard library functions may not be invoked by user code called from `execution::parallel_unsequenced_policy` algorithms. [*Note:* Implementations must ensure that internal synchronization inside standard library functions does not prevent forward progress when those functions are executed by threads of execution with weakly parallel forward progress guarantees. *— end note*] [*Example:*

```
int x = 0;
std::mutex m;
int a[] = {1,2};
std::for_each(std::execution::par_unseq, std::begin(a), std::end(a), [&](int) {
  std::lock_guard<mutex> guard(m); // incorrect: lock_guard constructor calls m.lock()
  ++x;
});
```

The above program may result in two consecutive calls to `m.lock()` on the same thread of execution (which may deadlock), because the applications of the function object are not guaranteed to run on different threads of execution. *— end example*] [*Note:* The semantics of the `execution::parallel_policy` or the `execution::parallel_unsequenced_policy` invocation allow the implementation to fall back to sequential execution if the system cannot parallelize an algorithm invocation due to lack of resources. *— end note*]

7 If an invocation of a parallel algorithm uses threads of execution implicitly created by the library, then the invoking thread of execution will either

(7.1) — temporarily block with forward progress guarantee delegation on the completion of these library-managed threads of execution, or

(7.2) — eventually execute an element access function;

the thread of execution will continue to do so until the algorithm is finished. [*Note:* In blocking with forward progress guarantee delegation in this context, a thread of execution created by the library is considered to have finished execution as soon as it has finished the execution of the particular element access function that the invoking thread of execution logically depends on. — *end note*]

8 The semantics of parallel algorithms invoked with an execution policy object of implementation-defined type are implementation-defined.

30.4.4 Parallel algorithm exceptions [algorithms.parallel.exceptions]

1 During the execution of a parallel algorithm, if temporary memory resources are required for parallelization and none are available, the algorithm throws a `bad_alloc` exception.

2 During the execution of a parallel algorithm, if the invocation of an element access function exits via an uncaught exception, the behavior is determined by the `ExecutionPolicy`.

30.4.5 `ExecutionPolicy` algorithm overloads [algorithms.parallel.overloads]

1 Parallel algorithms are algorithm overloads. Each parallel algorithm overload has an additional template type parameter named `ExecutionPolicy`, which is the first template parameter. Additionally, each parallel algorithm overload has an additional function parameter of type `ExecutionPolicy&&`, which is the first function parameter. [*Note:* Not all algorithms have parallel algorithm overloads. — *end note*]

2 Unless otherwise specified, the semantics of `ExecutionPolicy` algorithm overloads are identical to their overloads without.

3 Unless otherwise specified, the complexity requirements of `ExecutionPolicy` algorithm overloads are relaxed from the complexity requirements of the overloads without as follows: when the guarantee says "at most *expr*" or "exactly *expr*" and does not specify the number of assignments or swaps, and *expr* is not already expressed with $\mathcal{O}()$ notation, the complexity of the algorithm shall be $\mathcal{O}(expr)$.

4 Parallel algorithms shall not participate in overload resolution unless `is_execution_policy_v<decay_t<ExecutionPolicy>>` is `true`.

30.5 Non-modifying sequence operations [alg.nonmodifying]

30.5.1 All of [alg.all_of]

```
template<class InputIterator, class Predicate>
  constexpr bool all_of(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  bool all_of(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
              Predicate pred);
namespace ranges {
  template <InputIterator I, Sentinel<I> S, class Proj = identity,
            IndirectUnaryPredicate<projected<I, Proj>> Pred>
    bool all_of(I first, S last, Pred pred, Proj proj = Proj{});
  template <InputRange Rng, class Proj = identity,
            IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    bool all_of(Rng&& rng, Pred pred, Proj proj = Proj{});
}
```

1 *Returns:* `true` if `[first,last)` is empty or if ~~`pred(*i)`~~*E* is `true` for every iterator `i` in the range `[first,last)`, and `false` otherwise, where *E* is `pred(*i)` and `invoke(pred, invoke(proj, *i))` for the overloads in namespace `std` and `std::ranges`, respectively.

2 *Complexity:* At most `last - first` applications of the predicate and `last - first` applications of the projection.

30.5.2 Any of [alg.any_of]

```
template<class InputIterator, class Predicate>
  constexpr bool any_of(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  bool any_of(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
              Predicate pred);
namespace ranges {
  template <InputIterator I, Sentinel<I> S, class Proj = identity,
            IndirectUnaryPredicate<projected<I, Proj>> Pred>
    bool any_of(I first, S last, Pred pred, Proj proj = Proj{});
  template <InputRange Rng, class Proj = identity,
            IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    bool any_of(Rng&& rng, Pred pred, Proj proj = Proj{});
}
```

1 *Returns:* `false` if `[first,last)` is empty or if there is no iterator `i` in the range `[first,last)` such that ~~`pred(*i)`~~*E* is `true`, and `true` otherwise, where *E* is `pred(*i)` and `invoke(pred, invoke(proj, *i))` for the overloads in namespace `std` and `std::ranges`, respectively.

2 *Complexity:* At most `last - first` applications of the predicate and `last - first` applications of the projection.

30.5.3 None of [alg.none_of]

```
template<class InputIterator, class Predicate>
  constexpr bool none_of(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  bool none_of(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
               Predicate pred);
namespace ranges {
  template <InputIterator I, Sentinel<I> S, class Proj = identity,
            IndirectUnaryPredicate<projected<I, Proj>> Pred>
    bool none_of(I first, S last, Pred pred, Proj proj = Proj{});
  template <InputRange Rng, class Proj = identity,
            IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    bool none_of(Rng&& rng, Pred pred, Proj proj = Proj{});
}
```

1 *Returns:* `true` if `[first,last)` is empty or if ~~`pred(*i)`~~*E* is `false` for every iterator `i` in the range `[first,last)`, and `false` otherwise, where *E* is `pred(*i)` and `invoke(pred, invoke(proj, *i))` for the overloads in namespace `std` and `std::ranges`, respectively.

2 *Complexity:* At most `last - first` applications of the predicate and `last - first` applications of the projection.

30.5.4 For each [alg.foreach]

```
template<class InputIterator, class Function>
  constexpr Function for_each(InputIterator first, InputIterator last, Function f);
```

1 *Requires:* `Function` shall satisfy the *Cpp98MoveConstructible* requirements (23). [*Note:* `Function` need not meet the requirements of *Cpp98CopyConstructible* (24). — *end note*]

2 *Effects:* Applies `f` to the result of dereferencing every iterator in the range `[first,last)`, starting from `first` and proceeding to `last - 1`. [*Note:* If the type of `first` satisfies the requirements of a mutable iterator, `f` may apply non-constant functions through the dereferenced iterator. — *end note*]

3 *Returns:* `f`.

4 *Complexity:* Applies `f` exactly `last - first` times.

5 *Remarks:* If `f` returns a result, the result is ignored.

```
template<class ExecutionPolicy, class ForwardIterator, class Function>
  void for_each(ExecutionPolicy&& exec,
                ForwardIterator first, ForwardIterator last,
                Function f);
```

6 *Requires:* `Function` shall satisfy the *Cpp98CopyConstructible* requirements.

7 *Effects:* Applies `f` to the result of dereferencing every iterator in the range `[first,last)`. [*Note:* If the type of `first` satisfies the requirements of a mutable iterator, `f` may apply non-constant functions through the dereferenced iterator. — *end note*]

8 *Complexity:* Applies `f` exactly `last - first` times.

9 *Remarks:* If `f` returns a result, the result is ignored. Implementations do not have the freedom granted under 30.4.3 to make arbitrary copies of elements from the input sequence.

10 [*Note:* Does not return a copy of its `Function` parameter, since parallelization may not permit efficient state accumulation. — *end note*]

```
namespace ranges {
  template <InputIterator I, Sentinel<I> S, class Proj = identity,
            IndirectUnaryInvocable<projected<I, Proj>> Fun>
    tagged_pair<tag::in(I), tag::fun(Fun)>
      for_each(I first, S last, Fun f, Proj proj = Proj{});
  template <InputRange Rng, class Proj = identity,
            IndirectUnaryInvocable<projected<iterator_t<Rng>, Proj>> Fun>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::fun(Fun)>
      for_each(Rng&& rng, Fun f, Proj proj = Proj{});
}
```

11 *Effects:* Calls `invoke(f, invoke(proj, *i))` for every iterator `i` in the range `[first,last)`, starting from `first` and proceeding to `last - 1`. [*Note:* If the result of `invoke(proj, *i)` is a mutable reference, `f` may apply nonconstant functions. — *end note*]

12 *Returns:* `{last, std::move(f)}`.

13 *Complexity:* Applies `f` and `proj` exactly `last - first` times.

14 *Remarks:* If `f` returns a result, the result is ignored.

15 [*Note:* The requirements of this algorithm are more strict than those for the sequential version of `std::for_each`; this algorithm requires `Fun` to satisfy `CopyConstructible`. — *end note*]

```
template<class InputIterator, class Size, class Function>
  constexpr InputIterator for_each_n(InputIterator first, Size n, Function f);
```

16 *Requires:* `Function` shall satisfy the *Cpp98MoveConstructible* requirements [*Note:* `Function` need not meet the requirements of *Cpp98CopyConstructible*. — *end note*]

17 *Requires:* `n >= 0`.

18 *Effects:* Applies `f` to the result of dereferencing every iterator in the range `[first,first + n)` in order. [*Note:* If the type of `first` satisfies the requirements of a mutable iterator, `f` may apply non-constant functions through the dereferenced iterator. — *end note*]

19 *Returns:* `first + n`.

20 *Remarks:* If `f` returns a result, the result is ignored.

```
template<class ExecutionPolicy, class ForwardIterator, class Size, class Function>
  ForwardIterator for_each_n(ExecutionPolicy&& exec, ForwardIterator first, Size n,
                             Function f);
```

21 *Requires:* `Function` shall satisfy the *Cpp98CopyConstructible* requirements.

22 *Requires:* `n >= 0`.

23 *Effects:* Applies `f` to the result of dereferencing every iterator in the range `[first,first + n)`. [*Note:* If the type of `first` satisfies the requirements of a mutable iterator, `f` may apply non-constant functions through the dereferenced iterator. — *end note*]

24 *Returns:* `first + n`.

25 *Remarks:* If `f` returns a result, the result is ignored. Implementations do not have the freedom granted under 30.4.3 to make arbitrary copies of elements from the input sequence.

30.5.5 Find [alg.find]

```
template<class InputIterator, class T>
  constexpr InputIterator find(InputIterator first, InputIterator last,
                               const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  ForwardIterator find(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
                       const T& value);

template<class InputIterator, class Predicate>
  constexpr InputIterator find_if(InputIterator first, InputIterator last,
                                  Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator find_if(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
                          Predicate pred);

template<class InputIterator, class Predicate>
  constexpr InputIterator find_if_not(InputIterator first, InputIterator last,
                                      Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator find_if_not(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
                              Predicate pred);
```

1 *Returns:* The first iterator `i` in the range `[first,last)` for which the following corresponding conditions hold: `*i == value`, `pred(*i) != false`, `pred(*i) == false`. Returns `last` if no such iterator is found.

2 *Complexity:* At most `last - first` applications of the corresponding predicate.

```
namespace ranges {
  template <InputIterator I, Sentinel<I> S, class T, class Proj = identity>
      requires IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T*>
    I find(I first, S last, const T& value, Proj proj = Proj{});
  template <InputRange Rng, class T, class Proj = identity>
      requires IndirectRelation<ranges::equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
```

```
    safe_iterator_t<Rng> find(Rng&& rng, const T& value, Proj proj = Proj{});
  template <InputIterator I, Sentinel<I> S, class Proj = identity,
            IndirectUnaryPredicate<projected<I, Proj>> Pred>
    I find_if(I first, S last, Pred pred, Proj proj = Proj{});
  template <InputRange Rng, class Proj = identity,
            IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    safe_iterator_t<Rng> find_if(Rng&& rng, Pred pred, Proj proj = Proj{});
  template <InputIterator I, Sentinel<I> S, class Proj = identity,
            IndirectUnaryPredicate<projected<I, Proj>> Pred>
    I find_if_not(I first, S last, Pred pred, Proj proj = Proj{});
  template <InputRange Rng, class Proj = identity,
            IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    safe_iterator_t<Rng> find_if_not(Rng&& rng, Pred pred, Proj proj = Proj{});
}
```

3 *Returns:* The first iterator `i` in the range `[first,last)` for which the following corresponding conditions hold: `invoke(proj, *i) == value`, `invoke(pred, invoke(proj, *i)) != false`, `invoke(pred, invoke(proj, *i)) == false`. Returns `last` if no such iterator is found.

4 *Complexity:* At most `last - first` applications of the corresponding predicate and projection.

30.5.6 Find end [alg.find.end]

```
template<class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    find_end(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  constexpr ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  ForwardIterator1
    find_end(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);
```

1 *Effects:* Finds a subsequence of equal values in a sequence.

2 *Returns:* The last iterator `i` in the range `[first1,last1 - (last2 - first2))` such that for every non-negative integer `n < (last2 - first2)`, the following corresponding conditions hold: `*(i + n) == *(first2 + n)`, `pred(*(i + n), *(first2 + n)) != false`. Returns `last1` if `[first2,last2)` is empty or if no such iterator is found.

3 *Complexity:* At most `(last2 - first2) * (last1 - first1 - (last2 - first2) + 1)` applications of the corresponding predicate.

```
namespace ranges {
  template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
            Sentinel<I2> S2, class Proj = identity,
            IndirectRelation<I2, projected<I1, Proj>> Pred = ranges::equal_to<>>
    I1 find_end(I1 first1, S1 last1, I2 first2, S2 last2,
                Pred pred = Pred{}, Proj proj = Proj{});
  template <ForwardRange Rng1, ForwardRange Rng2, class Proj = identity,
            IndirectRelation<iterator_t<Rng2>,
                             projected<iterator_t<Rng>, Proj>> Pred = ranges::equal_to<>>
    safe_iterator_t<Rng1> find_end(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
                                   Proj proj = Proj{});
}
```

4 *Effects:* Finds a subsequence of equal values in a sequence.

5 *Returns:* The last iterator `i` in the range `[first1,last1 - (last2 - first2))` such that for every non-negative integer `n < (last2 - first2)`, the following condition holds: `invoke(pred, invoke(proj, *(i + n)), *(first2 + n)) != false`. Returns `last1` if `[first2,last2)` is empty or if no such iterator is found.

6 *Complexity:* At most `(last2 - first2) * (last1 - first1 - (last2 - first2) + 1)` applications of the corresponding predicate and projection.

30.5.7 Find first [alg.find.first.of]

```
template<class InputIterator, class ForwardIterator>
  constexpr InputIterator
    find_first_of(InputIterator first1, InputIterator last1,
                  ForwardIterator first2, ForwardIterator last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    find_first_of(ExecutionPolicy&& exec,
                  ForwardIterator1 first1, ForwardIterator1 last1,
                  ForwardIterator2 first2, ForwardIterator2 last2);

template<class InputIterator, class ForwardIterator,
         class BinaryPredicate>
  constexpr InputIterator
    find_first_of(InputIterator first1, InputIterator last1,
                  ForwardIterator first2, ForwardIterator last2,
                  BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  ForwardIterator1
    find_first_of(ExecutionPolicy&& exec,
                  ForwardIterator1 first1, ForwardIterator1 last1,
                  ForwardIterator2 first2, ForwardIterator2 last2,
                  BinaryPredicate pred);
```

1 *Effects:* Finds an element that matches one of a set of values.

2 *Returns:* The first iterator `i` in the range `[first1,last1)` such that for some iterator `j` in the range `[first2,last2)` the following conditions hold: `*i == *j, pred(*i, *j) != false`. Returns `last1` if `[first2,last2)` is empty or if no such iterator is found.

3 *Complexity:* At most `(last1-first1) * (last2-first2)` applications of the corresponding predicate.

```
namespace ranges {
  template <InputIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2,
          class Proj1 = identity, class Proj2 = identity,
          IndirectRelation<projected<I1, Proj1>, projected<I2, Proj2>> Pred = ranges::equal_to<>>
    I1 find_first_of(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
                     Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template <InputRange Rng1, ForwardRange Rng2, class Proj1 = identity, class Proj2 = identity,
          IndirectRelation<projected<iterator_t<Rng1>, Proj1>,
                           projected<iterator_t<Rng2>, Proj2>> Pred = ranges::equal_to<>>
    safe_iterator_t<Rng1> find_first_of(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
                                        Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

4 *Effects:* Finds an element that matches one of a set of values.

5 *Returns:* The first iterator `i` in the range `[first1,last1)` such that for some iterator `j` in the range `[first2,last2)` the following condition holds: `invoke(pred, invoke(proj1, *i), invoke(proj2, *j)) != false`. Returns `last1` if `[first2,last2)` is empty or if no such iterator is found.

6 *Complexity:* At most `(last1-first1) * (last2-first2)` applications of the corresponding predicate and the two projections.

30.5.8 Adjacent find [alg.adjacent.find]

```
template<class ForwardIterator>
  constexpr ForwardIterator
    adjacent_find(ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator
    adjacent_find(ExecutionPolicy&& exec,
                  ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
  constexpr ForwardIterator
    adjacent_find(ForwardIterator first, ForwardIterator last,
                  BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator, class BinaryPredicate>
  ForwardIterator
    adjacent_find(ExecutionPolicy&& exec,
                  ForwardIterator first, ForwardIterator last,
                  BinaryPredicate pred);
```

1 *Returns:* The first iterator `i` such that both `i` and `i + 1` are in the range `[first,last)` for which the following corresponding conditions hold: `*i == *(i + 1), pred(*i, *(i + 1)) != false`. Returns `last` if no such iterator is found.

2 *Complexity:* For the overloads with no `ExecutionPolicy`, exactly `min((i - first) + 1, (last - first) - 1)` applications of the corresponding predicate, where `i` is `adjacent_find`'s return value. For the overloads with an `ExecutionPolicy`, $\mathcal{O}($`last - first`$)$ applications of the corresponding predicate.

```
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
          IndirectRelation<projected<I, Proj>> Pred = ranges::equal_to<>>
    I adjacent_find(I first, S last, Pred pred = Pred{}, Proj proj = Proj{});
  template <ForwardRange Rng, class Proj = identity,
          IndirectRelation<projected<iterator_t<Rng>, Proj>> Pred = ranges::equal_to<>>
```

```
    safe_iterator_t<Rng> adjacent_find(Rng&& rng, Pred pred = Pred{}, Proj proj = Proj{});
}
```

3 *Returns:* The first iterator `i` such that both `i` and `i + 1` are in the range `[first,last)` for which the following corresponding condition holds: `invoke(pred, invoke(proj, *i), invoke(proj, *(i + 1))) != false`. Returns `last` if no such iterator is found.

4 *Complexity:* For a nonempty range, exactly `min((i - first) + 1, (last - first) - 1)` applications of the corresponding predicate, where `i` is `adjacent_find`'s return value, and no more than twice as many applications of the projection.

30.5.9 Count [alg.count]

```
template<class InputIterator, class T>
  constexpr typename iterator_traits<InputIterator>::difference_type
    count(InputIterator first, InputIterator last, const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  typename iterator_traits<ForwardIterator>::difference_type
    count(ExecutionPolicy&& exec,
          ForwardIterator first, ForwardIterator last, const T& value);

template<class InputIterator, class Predicate>
  constexpr typename iterator_traits<InputIterator>::difference_type
    count_if(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  typename iterator_traits<ForwardIterator>::difference_type
    count_if(ExecutionPolicy&& exec,
             ForwardIterator first, ForwardIterator last, Predicate pred);
```

1 *Effects:* Returns the number of iterators `i` in the range `[first,last)` for which the following corresponding conditions hold: `*i == value, pred(*i) != false`.

2 *Complexity:* Exactly `last - first` applications of the corresponding predicate.

```
namespace ranges {
  template <InputIterator I, Sentinel<I> S, class T, class Proj = identity>
      requires IndirectRelation<equal_to<>, projected<I, Proj>, const T*>
    iter_difference_t<I> count(I first, S last, const T& value, Proj proj = Proj{});
  template <InputRange Rng, class T, class Proj = identity>
      requires IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
    iter_difference_t<iterator_t<Rng>> count(Rng&& rng, const T& value, Proj proj = Proj{});

  template <InputIterator I, Sentinel<I> S, class Proj = identity,
            IndirectUnaryPredicate<projected<I, Proj>> Pred>
    iter_difference_t<I> count_if(I first, S last, Pred pred, Proj proj = Proj{});
  template <InputRange Rng, class Proj = identity,
            IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    iter_difference_t<iterator_t<Rng>> count_if(Rng&& rng, Pred pred, Proj proj = Proj{});
}
```

3 *Effects:* Returns the number of iterators `i` in the range `[first,last)` for which the following corresponding conditions hold: `invoke(proj, *i) == value, invoke(pred, invoke(proj, *i)) != false`.

4 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

30.5.10 Mismatch [mismatch]

```
template<class InputIterator1, class InputIterator2>
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2);

template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, BinaryPredicate pred);

template<class InputIterator1, class InputIterator2>
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);

template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2,
             BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);
```

1 *Remarks:* If `last2` was not given in the argument list, it denotes `first2 + (last1 - first1)` below.

2 *Returns:* A pair of iterators `first1 + n` and `first2 + n`, where `n` is the smallest integer such that, respectively,

- (2.1) `!(*(first1 + n) == *(first2 + n))` or
- (2.2) `pred(*(first1 + n), *(first2 + n)) == false`,

or `min(last1 - first1, last2 - first2)` if no such integer exists.

3 *Complexity:* At most `min(last1 - first1, last2 - first2)` applications of the corresponding predicate.

```
namespace ranges {
  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
            class Proj1 = identity, class Proj2 = identity,
            IndirectRelation<projected<I1, Proj1>, projected<I2, Proj2>> Pred = ranges::equal_to<>>
    tagged_pair<tag::in1(I1), tag::in2(I2)>
      mismatch(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template <InputRange Rng1, InputRange Rng2, class Proj1 = identity, class Proj2 = identity,
            IndirectRelation<projected<iterator_t<Rng1>, Proj1>,
                             projected<iterator_t<Rng2>, Proj2>> Pred = ranges::equal_to<>>
    tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::in2(safe_iterator_t<Rng2>)>
      mismatch(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{}, Proj1 proj1 = Proj1{},
               Proj2 proj2 = Proj2{});
}
```

4 *Returns:* A pair of iterators `i` and `j` such that `j == first2 + (i - first1)` and `i` is the first iterator in the range `[first1,last1)` for which the following corresponding conditions hold:

- (4.1) — `j` is in the range `[first2, last2)`.
- (4.2) — `*i != *(first2 + (i - first1))`
- (4.3) — `!invoke(pred, invoke(proj1, *i), invoke(proj2, *(first2 + (i - first1))))`

Returns the pair `first1 + min(last1 - first1, last2 - first2)` and `first2 + min(last1 - first1, last2 - first2)` if such an iterator `i` is not found.

5 *Complexity:* At most `last1 - first1` applications of the corresponding predicate and both projections.

30.5.11 Equal [alg.equal]

```
template<class InputIterator1, class InputIterator2>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  bool equal(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2);

template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  bool equal(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, BinaryPredicate pred);

template<class InputIterator1, class InputIterator2>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, InputIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  bool equal(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
```

```
            ForwardIterator2 first2, ForwardIterator2 last2);

template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, InputIterator2 last2,
                       BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  bool equal(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);
```

1 *Remarks:* If `last2` was not given in the argument list, it denotes `first2 + (last1 - first1)` below.

2 *Returns:* If `last1 - first1 != last2 - first2`, return `false`. Otherwise return `true` if for every iterator `i` in the range `[first1,last1)` the following corresponding conditions hold: `*i == *(first2 + (i - first1))`, `pred(*i, *(first2 + (i - first1))) != false`. Otherwise, returns `false`.

3 *Complexity:*

- (3.1) For the overloads with no `ExecutionPolicy`,
  - (3.1.1) if `InputIterator1` and `InputIterator2` meet the requirements of random access iterators (28.3.5.6) and `last1 - first1 != last2 - first2`, then no applications of the corresponding predicate; otherwise,
  - (3.1.2) at most $\min($`last1 - first1`$,$`last2 - first2`$)$ applications of the corresponding predicate.
- (3.2) For the overloads with an `ExecutionPolicy`,
  - (3.2.1) if `ForwardIterator1` and `ForwardIterator2` meet the requirements of random access iterators and `last1 - first1 != last2 - first2`, then no applications of the corresponding predicate; otherwise,
  - (3.2.2) $\mathcal{O}(\min($`last1 - first1`$,$`last2 - first2`$))$ applications of the corresponding predicate.

```
namespace ranges {
  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
            class Pred = ranges::equal_to<>, class Proj1 = identity, class Proj2 = identity>
      requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
    bool equal(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

  template <InputRange Rng1, InputRange Rng2, class Pred = ranges::equal_to<>,
            class Proj1 = identity, class Proj2 = identity>
      requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
    bool equal(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{}, Proj1 proj1 = Proj1{},
               Proj2 proj2 = Proj2{});
}
```

4 *Returns:* If `last1 - first1 != last2 - first2`, return `false`. Otherwise return `true` if for every iterator `i` in the range `[first1,last1)` the following condition holds: `invoke(pred, invoke(proj1, *i), invoke(proj2, *(first2 + (i - first1))))`. Otherwise, returns `false`.

5 *Complexity:* No applications of the corresponding predicate and projections if:

- (5.1) `SizedSentinel<S1, I1>` is satisfied, and
- (5.2) `SizedSentinel<S2, I2>` is satisfied, and

(5.3) — `last1 - first1 != last2 - first2`.

Otherwise, at most `min(last1 - first1, last2 - first2)` applications of the corresponding predicate and projections.

30.5.12 Is permutation [alg.is_permutation]

```
template<class ForwardIterator1, class ForwardIterator2>
  constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                                ForwardIterator2 first2);
template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                                ForwardIterator2 first2, BinaryPredicate pred);
template<class ForwardIterator1, class ForwardIterator2>
  constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                                ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                                ForwardIterator2 first2, ForwardIterator2 last2,
                                BinaryPredicate pred);
```

1 *Requires:* `ForwardIterator1` and `ForwardIterator2` shall have the same value type. The comparison function shall be an equivalence relation.

2 *Remarks:* If `last2` was not given in the argument list, it denotes `first2 + (last1 - first1)` below.

3 *Returns:* If `last1 - first1 != last2 - first2`, return `false`. Otherwise return `true` if there exists a permutation of the elements in the range `[first2,first2 + (last1 - first1))`, beginning with `ForwardIterator2 begin`, such that `equal(first1, last1, begin)` returns `true` or `equal(first1, last1, begin, pred)` returns `true`; otherwise, returns `false`.

4 *Complexity:* No applications of the corresponding predicate if `ForwardIterator1` and `ForwardIterator2` meet the requirements of random access iterators and `last1 - first1 != last2 - first2`. Otherwise, exactly `last1 - first1` applications of the corresponding predicate if `equal(first1, last1, first2, last2)` would return `true` if `pred` was not given in the argument list or `equal(first1, last1, first2, last2, pred)` would return `true` if pred was given in the argument list; otherwise, at worst $\mathcal{O}(N^2)$, where N has the value `last1 - first1`.

```
namespace ranges {
  template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
            Sentinel<I2> S2, class Pred = ranges::equal_to<>, class Proj1 = identity,
            class Proj2 = identity>
      requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
    bool is_permutation(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
                        Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template <ForwardRange Rng1, ForwardRange Rng2, class Pred = ranges::equal_to<>,
            class Proj1 = identity, class Proj2 = identity>
      requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
    bool is_permutation(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{}, Proj1 proj1 = Proj1{},
                        Proj2 proj2 = Proj2{});
}
```

5 *Returns:* If `last1 - first1 != last2 - first2`, return `false`. Otherwise return `true` if there exists a permutation of the elements in the range `[first2,first2 + (last1 - first1))`, beginning with `I2 begin`, such that `equal(first1, last1, begin, pred, proj1, proj2)` returns `true`; otherwise, returns `false`.

6 *Complexity:* No applications of the corresponding predicate and projections if:

- (6.1) `SizedSentinel<S1, I1>` is satisfied, and
- (6.2) `SizedSentinel<S2, I2>` is satisfied, and
- (6.3) `last1 - first1 != last2 - first2`.

Otherwise, exactly `last1 - first1` applications of the corresponding predicate and projections if `equal(first1, last1, first2, last2, pred, proj1, proj2)` would return `true`; otherwise, at worst $\mathcal{O}(N^2)$, where N has the value `last1 - first1`.

30.5.13 Search [alg.search]

```
template<class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator1
    search(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    search(ExecutionPolicy&& exec,
           ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  constexpr ForwardIterator1
    search(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2,
           BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  ForwardIterator1
    search(ExecutionPolicy&& exec,
           ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2,
           BinaryPredicate pred);
```

1 *Effects:* Finds a subsequence of equal values in a sequence.

2 *Returns:* The first iterator `i` in the range `[first1,last1 - (last2-first2))` such that for every non-negative integer `n` less than `last2 - first2` the following corresponding conditions hold: `*(i + n) == *(first2 + n), pred(*(i + n), *(first2 + n)) != false`. Returns `first1` if `[first2, last2)` is empty, otherwise returns `last1` if no such iterator is found.

3 *Complexity:* At most `(last1 - first1) * (last2 - first2)` applications of the corresponding predicate.

```
namespace ranges {
  template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2,
            class Pred = ranges::equal_to<>, class Proj1 = identity, class Proj2 = identity>
      requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
    I1 search(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template <ForwardRange Rng1, ForwardRange Rng2, class Pred = ranges::equal_to<>,
            class Proj1 = identity, class Proj2 = identity>
      requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
    safe_iterator_t<Rng1> search(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
```

```
                                   Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

4 *Effects:* Finds a subsequence of equal values in a sequence.

5 *Returns:* The first iterator `i` in the range `[first1,last1 - (last2-first2))` such that for every non-negative integer `n` less than `last2 - first2` the following condition holds:

`invoke(pred, invoke(proj1, *(i + n)), invoke(proj2, *(first2 + n))) != false.`

Returns `first1` if `[first2,last2)` is empty, otherwise returns `last1` if no such iterator is found.

6 *Complexity:* At most `(last1 - first1) * (last2 - first2)` applications of the corresponding predicate and projections.

```
template<class ForwardIterator, class Size, class T>
  constexpr ForwardIterator
    search_n(ForwardIterator first, ForwardIterator last,
             Size count, const T& value);
template<class ExecutionPolicy, class ForwardIterator, class Size, class T>
  ForwardIterator
    search_n(ExecutionPolicy&& exec,
             ForwardIterator first, ForwardIterator last,
             Size count, const T& value);

template<class ForwardIterator, class Size, class T,
         class BinaryPredicate>
  constexpr ForwardIterator
    search_n(ForwardIterator first, ForwardIterator last,
             Size count, const T& value,
             BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Size, class T,
         class BinaryPredicate>
  ForwardIterator
    search_n(ExecutionPolicy&& exec,
             ForwardIterator first, ForwardIterator last,
             Size count, const T& value,
             BinaryPredicate pred);
```

7 *Requires:* The type `Size` shall be convertible to integral type (cxxrefconv.integral, cxxrefclass.conv).

8 *Effects:* Finds a subsequence of equal values in a sequence.

9 *Returns:* The first iterator `i` in the range `[first,last-count)` such that for every non-negative integer `n` less than `count` the following corresponding conditions hold: `*(i + n) == value, pred(*(i + n),value) != false`. Returns `last` if no such iterator is found.

10 *Complexity:* At most `last - first` applications of the corresponding predicate.

```
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class T, class Pred = ranges::equal_to<>,
            class Proj = identity>
      requires IndirectlyComparable<I, const T*, Pred, Proj>
    I search_n(I first, S last, iter_difference_t<I> count, const T& value, Pred pred = Pred{},
               Proj proj = Proj{});
  template <ForwardRange Rng, class T, class Pred = ranges::equal_to<>, class Proj = identity>
      requires IndirectlyComparable<iterator_t<Rng>, const T*, Pred, Proj>
    safe_iterator_t<Rng> search_n(Rng&& rng, iter_difference_t<iterator_t<Rng>> count,
                                  const T& value, Pred pred = Pred{}, Proj proj = Proj{});
}
```

11 *Effects:* Finds a subsequence of equal values in a sequence.

12 *Returns:* The first iterator `i` in the range `[first,last-count)` such that for every non-negative integer `n` less than `count` the following condition holds: `invoke(pred, invoke(proj, *(i + n)), value) != false`. Returns `last` if no such iterator is found.

13 *Complexity:* At most `last - first` applications of the corresponding predicate and projection.

```
template<class ForwardIterator, class Searcher>
  constexpr ForwardIterator
    search(ForwardIterator first, ForwardIterator last, const Searcher& searcher);
```

14 *Effects:* Equivalent to: `return searcher(first, last).first;`

15 *Remarks:* `Searcher` need not meet the *Cpp98CopyConstructible* requirements.

30.6 Mutating sequence operations [alg.modifying.operations]

30.6.1 Copy [alg.copy]

```
template<class InputIterator, class OutputIterator>
  constexpr OutputIterator copy(InputIterator first, InputIterator last,
                                OutputIterator result);
```

1 *Requires:* `result` shall not be in the range `[first,last)`.

2 *Effects:* Copies elements in the range `[first,last)` into the range `[result,result + (last - first))` starting from `first` and proceeding to `last`. For each non-negative integer `n < (last - first)`, performs `*(result + n) = *(first + n)`.

3 *Returns:* `result + (last - first)`.

4 *Complexity:* Exactly `last - first` assignments.

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 copy(ExecutionPolicy&& policy,
                        ForwardIterator1 first, ForwardIterator1 last,
                        ForwardIterator2 result);
```

5 *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

6 *Effects:* Copies elements in the range `[first,last)` into the range `[result,result + (last - first))`. For each non-negative integer `n < (last - first)`, performs `*(result + n) = *(first + n)`.

7 *Returns:* `result + (last - first)`.

8 *Complexity:* Exactly `last - first` assignments.

```
namespace ranges {
  template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
      requires IndirectlyCopyable<I, O>
    tagged_pair<tag::in(I), tag::out(O)> copy(I first, S last, O result);
  template <InputRange Rng, WeaklyIncrementable O>
      requires IndirectlyCopyable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)> copy(Rng&& rng, O result);
}
```

9 *Effects:* Copies elements in the range `[first,last)` into the range `[result,result + (last - first))` starting from `first` and proceeding to `last`. For each non-negative integer `n < (last - first)`, performs `*(result + n) = *(first + n)`.

10 *Returns:* `{last, result + (last - first)}`.

11 *Requires:* `result` shall not be in the range `[first,last)`.

12 *Complexity:* Exactly `last - first` assignments.

```
template<class InputIterator, class Size, class OutputIterator>
  constexpr OutputIterator copy_n(InputIterator first, Size n,
                                  OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class Size, class ForwardIterator2>
  ForwardIterator2 copy_n(ExecutionPolicy&& exec,
                          ForwardIterator1 first, Size n,
                          ForwardIterator2 result);
```

13 *Effects:* For each non-negative integer $i < n$, performs `*(result + i) = *(first + i)`.

14 *Returns:* `result + n`.

15 *Complexity:* Exactly `n` assignments.

```
namespace ranges {
  template <InputIterator I, WeaklyIncrementable O>
      requires IndirectlyCopyable<I, O>
    tagged_pair<tag::in(I), tag::out(O)> copy_n(I first, iter_difference_t<I> n, O result);
}
```

16 *Effects:* For each non-negative integer $i < n$, performs `*(result + i) = *(first + i)`.

17 *Returns:* `{first + n, result + n}`.

18 *Complexity:* Exactly `n` assignments.

```
template<class InputIterator, class OutputIterator, class Predicate>
  constexpr OutputIterator copy_if(InputIterator first, InputIterator last,
                                   OutputIterator result, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate>
  ForwardIterator2 copy_if(ExecutionPolicy&& exec,
                           ForwardIterator1 first, ForwardIterator1 last,
                           ForwardIterator2 result, Predicate pred);
```

19 *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap. [*Note:* For the overload with an `ExecutionPolicy`, there may be a performance cost if `iterator_traits<ForwardIterator1>::value_type` is not `MoveConstructible` (23). — *end note*]

20 *Effects:* Copies all of the elements referred to by the iterator `i` in the range `[first,last)` for which `pred(*i)` is `true`.

21 *Returns:* The end of the resulting range.

22 *Complexity:* Exactly `last - first` applications of the corresponding predicate.

23 *Remarks:* Stable20.5.5.7.

```
namespace ranges {
  template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class Proj = identity,
            IndirectUnaryPredicate<projected<I, Proj>> Pred>
      requires IndirectlyCopyable<I, O>
    tagged_pair<tag::in(I), tag::out(O)>
      copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});
  template <InputRange Rng, WeaklyIncrementable O, class Proj = identity,
            IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
      requires IndirectlyCopyable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
```

```
      copy_if(Rng&& rng, O result, Pred pred, Proj proj = Proj{});
}
```

24 Let N be the number of iterators `i` in the range `[first,last)` for which the condition `invoke(pred, invoke(proj, *i))` holds.

25 *Requires:* The ranges `[first,last)` and `[result,result +` N`)` shall not overlap.

26 *Effects:* Copies all of the elements referred to by the iterator `i` in the range `[first,last)` for which `invoke(pred, invoke(proj, *i))` is `true`.

27 *Returns:* `{last, result +` N`}`.

28 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

29 *Remarks:* Stable (20.5.5.7).

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
  constexpr BidirectionalIterator2
    copy_backward(BidirectionalIterator1 first,
                  BidirectionalIterator1 last,
                  BidirectionalIterator2 result);
```

30 *Requires:* `result` shall not be in the range `(first,last]`.

31 *Effects:* Copies elements in the range `[first,last)` into the range `[result - (last-first),result )` starting from `last - 1` and proceeding to `first`.[4] For each positive integer `n <= (last - first)`, performs `*(result - n) = *(last - n)`.

32 *Returns:* `result - (last - first)`.

33 *Complexity:* Exactly `last - first` assignments.

```
namespace ranges {
  template <BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
      requires IndirectlyCopyable<I1, I2>
    tagged_pair<tag::in(I1), tag::out(I2)> copy_backward(I1 first, S1 last, I2 result);
  template <BidirectionalRange Rng, BidirectionalIterator I>
      requires IndirectlyCopyable<iterator_t<Rng>, I>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(I)>
      copy_backward(Rng&& rng, I result);
}
```

34 *Effects:* Copies elements in the range `[first,last)` into the range `[result - (last-first),result )` starting from `last - 1` and proceeding to `first`.[5] For each positive integer `n <= (last - first)`, performs `*(result - n) = *(last - n)`.

35 *Requires:* `result` shall not be in the range `(first,last]`.

36 *Returns:* `{last, result - (last - first)}`.

37 *Complexity:* Exactly `last - first` assignments.

30.6.2 Move [alg.move]

```
template<class InputIterator, class OutputIterator>
  constexpr OutputIterator move(InputIterator first, InputIterator last,
                                OutputIterator result);
```

---

[4] `copy_backward` should be used instead of copy when `last` is in the range `[result - (last - first),result)`.
[5] `copy_backward` should be used instead of copy when `last` is in the range `[result - (last - first),result)`.

1 *Requires:* `result` shall not be in the range `[first,last)`.

2 *Effects:* Moves elements in the range `[first,last)` into the range `[result,result + (last - first))` starting from first and proceeding to last. For each non-negative integer `n < (last-first)`, performs `*(result + n) = std::move(*(first + n))`.

3 *Returns:* `result + (last - first)`.

4 *Complexity:* Exactly `last - first` move assignments.

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 move(ExecutionPolicy&& policy,
                        ForwardIterator1 first, ForwardIterator1 last,
                        ForwardIterator2 result);
```

5 *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

6 *Effects:* Moves elements in the range `[first,last)` into the range `[result,result + (last - first))`. For each non-negative integer `n < (last - first)`, performs `*(result + n) = std::move(*(first + n))`.

7 *Returns:* `result + (last - first)`.

8 *Complexity:* Exactly `last - first` assignments.

```
namespace ranges {
  template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
      requires IndirectlyMovable<I, O>
    tagged_pair<tag::in(I), tag::out(O)> move(I first, S last, O result);
  template <InputRange Rng, WeaklyIncrementable O>
      requires IndirectlyMovable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)> move(Rng&& rng, O result);
}
```

9 *Effects:* Moves elements in the range `[first,last)` into the range `[result,result + (last - first))` starting from first and proceeding to last. For each non-negative integer `n < (last-first)`, performs `*(result + n) = ranges::iter_move(first + n)`.

10 *Returns:* `{last, result + (last - first)}`.

11 *Requires:* `result` shall not be in the range `[first,last)`.

12 *Complexity:* Exactly `last - first` move assignments.

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
  constexpr BidirectionalIterator2
    move_backward(BidirectionalIterator1 first, BidirectionalIterator1 last,
                  BidirectionalIterator2 result);
```

13 *Requires:* `result` shall not be in the range `(first,last]`.

14 *Effects:* Moves elements in the range `[first,last)` into the range `[result - (last-first),result )` starting from `last - 1` and proceeding to first.[6] For each positive integer `n <= (last - first)`, performs `*(result - n) = std::move(*(last - n))`.

15 *Returns:* `result - (last - first)`.

16 *Complexity:* Exactly `last - first` assignments.

---

[6] `move_backward` should be used instead of move when last is in the range `[result - (last - first),result)`.

```
namespace ranges {
  template <BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
      requires IndirectlyMovable<I1, I2>
    tagged_pair<tag::in(I1), tag::out(I2)> move_backward(I1 first, S1 last, I2 result);
  template <BidirectionalRange Rng, BidirectionalIterator I>
      requires IndirectlyMovable<iterator_t<Rng>, I>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(I)> move_backward(Rng&& rng, I result);
}
```

17 *Effects:* Moves elements in the range `[first,last)` into the range `[result - (last-first),result )` starting from `last - 1` and proceeding to first.[7] For each positive integer `n <= (last - first)`, performs `*(result - n) = ranges::iter_move(last - n)`.

18 *Requires:* `result` shall not be in the range `(first,last]`.

19 *Returns:* `{last, result - (last - first)}`.

20 *Complexity:* Exactly `last - first` assignments.

30.6.3 Swap [alg.swap]

```
template<class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2
    swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2
    swap_ranges(ExecutionPolicy&& exec,
                ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2);
```

1 *Requires:* The two ranges `[first1,last1)` and `[first2,first2 + (last1 - first1))` shall not overlap. `*(first1 + n)` shall be swappable with?.3.11 `*(first2 + n)`.

2 *Effects:* For each non-negative integer `n < (last1 - first1)` performs: `swap(*(first1 + n), *(first2 + n))`.

3 *Returns:* `first2 + (last1 - first1)`.

4 *Complexity:* Exactly `last1 - first1` swaps.

```
namespace ranges {
  template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2>
      requires IndirectlySwappable<I1, I2>
    tagged_pair<tag::in1(I1), tag::in2(I2)> swap_ranges(I1 first1, S1 last1, I2 first2, S2 last2);
  template <ForwardRange Rng1, ForwardRange Rng2>
      requires IndirectlySwappable<iterator_t<Rng1>, iterator_t<Rng2>>
    tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::in2(safe_iterator_t<Rng2>)>
      swap_ranges(Rng1&& rng1, Rng2&& rng2);
}
```

5 *Effects:* For each non-negative integer `n < min(last1 - first1, last2 - first2)` performs: `ranges::iter_swap(first1 + n, first2 + n)`.

6 *Requires:* The two ranges `[first1,last1)` and `[first2,last2)` shall not overlap. `*(first1 + n)` shall be swappable with (?.3.11) `*(first2 + n)`.

7 *Returns:* `{first1 + n, first2 + n}`, where `n` is `min(last1 - first1, last2 - first2)`.

8 *Complexity:* Exactly `min(last1 - first1, last2 - first2)` swaps.

---

[7] `move_backward` should be used instead of move when last is in the range `[result - (last - first),result)`.

```
template<class ForwardIterator1, class ForwardIterator2>
  void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

9 *Requires:* `a` and `b` shall be dereferenceable. `*a` shall be swappable with?.3.11 `*b`.

10 *Effects:* As if by `swap(*a, *b)`.

30.6.4 Transform [alg.transform]

```
template<class InputIterator, class OutputIterator,
         class UnaryOperation>
  constexpr OutputIterator
    transform(InputIterator first, InputIterator last,
              OutputIterator result, UnaryOperation op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class UnaryOperation>
  ForwardIterator2
    transform(ExecutionPolicy&& exec,
              ForwardIterator1 first, ForwardIterator1 last,
              ForwardIterator2 result, UnaryOperation op);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class BinaryOperation>
  constexpr OutputIterator
    transform(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, OutputIterator result,
              BinaryOperation binary_op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class BinaryOperation>
  ForwardIterator
    transform(ExecutionPolicy&& exec,
              ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator result,
              BinaryOperation binary_op);
```

1 *Requires:* `op` and `binary_op` shall not invalidate iterators or subranges, or modify elements in the ranges

(1.1) — `[first1,last1]`,

(1.2) — `[first2,first2 + (last1 - first1)]`, and

(1.3) — `[result,result + (last1 - first1)]`.[8]

2 *Effects:* Assigns through every iterator `i` in the range `[result,result + (last1 - first1))` a new corresponding value equal to `op(*(first1 + (i - result)))` or `binary_op(*(first1 + (i - result)), *(first2 + (i - result)))`.

3 *Returns:* `result + (last1 - first1)`.

4 *Complexity:* Exactly `last1 - first1` applications of `op` or `binary_op`. This requirement also applies to the overload with an `ExecutionPolicy` .

5 *Remarks:* `result` may be equal to `first` in case of unary transform, or to `first1` or `first2` in case of binary transform.

```
namespace ranges {
  template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
```

---

[8] The use of fully closed ranges is intentional.

```
          CopyConstructible F, class Proj = identity>
      requires Writable<O, indirect_result_t<F&, projected<I, Proj>>>
    tagged_pair<tag::in(I), tag::out(O)>
      transform(I first, S last, O result, F op, Proj proj = Proj{});
  template <InputRange Rng, WeaklyIncrementable O, CopyConstructible F,
            class Proj = identity>
    requires Writable<O, indirect_result_t<F&, projected<iterator_t<R>, Proj>>>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
      transform(Rng&& rng, O result, F op, Proj proj = Proj{});
  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
            WeaklyIncrementable O, CopyConstructible F, class Proj1 = identity,
            class Proj2 = identity>
      requires Writable<O, indirect_result_t<F&, projected<I1, Proj1>, projected<I2, Proj2>>>
    tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
      transform(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
            CopyConstructible F, class Proj1 = identity, class Proj2 = identity>
      requires Writable<O, indirect_result_t<F&, projected<iterator_t<Rng1>, Proj1>,
                                                  projected<iterator_t<Rng2>, Proj2>>>
    tagged_tuple<tag::in1(safe_iterator_t<Rng1>), tag::in2(safe_iterator_t<Rng2>), tag::out(O)>
      transform(Rng1&& rng1, Rng2&& rng2, O result,
                F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

6 Let N be `(last1 - first1)` for unary transforms, or `min(last1 - first1, last2 - first2) for binary transforms`.

7 *Effects:* Assigns through every iterator `i` in the range `[result,result +` N`)` a new corresponding value equal to `invoke(op, invoke(proj, *(first1 + (i - result))))` or `invoke(binary_op, invoke(proj1, *(first1 + (i - result))), invoke(proj2, *(first2 + (i - result))))`.

8 *Requires:* `op` and `binary_op` shall not invalidate iterators or subranges, or modify elements in the ranges `[first1,first1 +` N`]`, `[first2,first2 +` N`]`, and `[result,result +` N`]`.[9]

9 *Returns:* `{first1 +` N`, result +` N`}` or `make_tagged_tuple<tag::in1, tag::in2, tag::out>( first1 +` N`, first2 +` N`, result +` N`)`.

10 *Complexity:* Exactly N applications of `op` or `binary_op` and the corresponding projection(s).

11 *Remarks:* `result` may be equal to `first1` in case of unary transform, or to `first1` or `first2` in case of binary transform.

30.6.5 Replace [alg.replace]

```
template<class ForwardIterator, class T>
  constexpr void replace(ForwardIterator first, ForwardIterator last,
                         const T& old_value, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  void replace(ExecutionPolicy&& exec,
               ForwardIterator first, ForwardIterator last,
               const T& old_value, const T& new_value);

template<class ForwardIterator, class Predicate, class T>
  constexpr void replace_if(ForwardIterator first, ForwardIterator last,
                            Predicate pred, const T& new_value);
```

---

[9] The use of fully closed ranges is intentional.

```
template<class ExecutionPolicy, class ForwardIterator, class Predicate, class T>
  void replace_if(ExecutionPolicy&& exec,
                  ForwardIterator first, ForwardIterator last,
                  Predicate pred, const T& new_value);
```

1 *Requires:* The expression `*first = new_value` shall be valid.

2 *Effects:* Substitutes elements referred by the iterator `i` in the range `[first,last)` with `new_value`, when the following corresponding conditions hold: `*i == old_value`, `pred(*i) != false`.

3 *Complexity:* Exactly `last - first` applications of the corresponding predicate.

```
namespace ranges {
  template <InputIterator I, Sentinel<I> S, class T1, class T2, class Proj = identity>
      requires Writable<I, const T2&> &&
               IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T1*>
    I replace(I first, S last, const T1& old_value, const T2& new_value, Proj proj = Proj{});
  template <InputRange Rng, class T1, class T2, class Proj = identity>
      requires Writable<iterator_t<Rng>, const T2&> &&
               IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T1*>
    safe_iterator_t<Rng>
      replace(Rng&& rng, const T1& old_value, const T2& new_value, Proj proj = Proj{});

  template <InputIterator I, Sentinel<I> S, class T, class Proj = identity,
            IndirectUnaryPredicate<projected<I, Proj>> Pred>
      requires Writable<I, const T&>
    I replace_if(I first, S last, Pred pred, const T& new_value, Proj proj = Proj{});
  template <InputRange Rng, class T, class Proj = identity,
            IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
      requires Writable<iterator_t<Rng>, const T&>
    safe_iterator_t<Rng>
      replace_if(Rng&& rng, Pred pred, const T& new_value, Proj proj = Proj{});
}
```

4 *Effects:* Assigns `new_value` through each iterator `i` in the range `[first,last)` when the following corresponding conditions hold: `invoke(proj, *i) == old_value`, `invoke(pred, invoke(proj, *i)) != false`.

5 *Returns:* `last`.

6 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

```
template<class InputIterator, class OutputIterator, class T>
  constexpr OutputIterator
    replace_copy(InputIterator first, InputIterator last,
                 OutputIterator result,
                 const T& old_value, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T>
  ForwardIterator2
    replace_copy(ExecutionPolicy&& exec,
                 ForwardIterator1 first, ForwardIterator1 last,
                 ForwardIterator2 result,
                 const T& old_value, const T& new_value);

template<class InputIterator, class OutputIterator, class Predicate, class T>
  constexpr OutputIterator
    replace_copy_if(InputIterator first, InputIterator last,
                    OutputIterator result,
```

```
                    Predicate pred, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate, class T>
  ForwardIterator2
    replace_copy_if(ExecutionPolicy&& exec,
                    ForwardIterator1 first, ForwardIterator1 last,
                    ForwardIterator2 result,
                    Predicate pred, const T& new_value);
```

7 *Requires:* The results of the expressions `*first` and `new_value` shall be writable (28.3.1) to the `result` output iterator. The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

8 *Effects:* Assigns to every iterator `i` in the range `[result,result + (last - first))` either `new_value` or `*(first + (i - result))` depending on whether the following corresponding conditions hold:

```
*(first + (i - result)) == old_value
pred(*(first + (i - result))) != false
```

9 *Returns:* `result + (last - first)`.

10 *Complexity:* Exactly `last - first` applications of the corresponding predicate.

```
namespace ranges {
  template <InputIterator I, Sentinel<I> S, class T1, class T2, OutputIterator<const T2&> O,
            class Proj = identity>
      requires IndirectlyCopyable<I, O> &&
               IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T1*>
    tagged_pair<tag::in(I), tag::out(O)>
      replace_copy(I first, S last, O result, const T1& old_value, const T2& new_value,
                   Proj proj = Proj{});
  template <InputRange Rng, class T1, class T2, OutputIterator<const T2&> O,
            class Proj = identity>
      requires IndirectlyCopyable<iterator_t<Rng>, O> &&
               IndirectRelation<ranges::equal_to<>, projected<iterator_t<Rng>, Proj>,
                                 const T1*>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
      replace_copy(Rng&& rng, O result, const T1& old_value, const T2& new_value,
                   Proj proj = Proj{});

  template <InputIterator I, Sentinel<I> S, class T, OutputIterator<const T&> O,
            class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
      requires IndirectlyCopyable<I, O>
    tagged_pair<tag::in(I), tag::out(O)>
      replace_copy_if(I first, S last, O result, Pred pred, const T& new_value,
                      Proj proj = Proj{});
  template <InputRange Rng, class T, OutputIterator<const T&> O, class Proj = identity,
            IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
      requires IndirectlyCopyable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
      replace_copy_if(Rng&& rng, O result, Pred pred, const T& new_value,
                      Proj proj = Proj{});
}
```

11 *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

12 *Effects:* Assigns to every iterator `i` in the range `[result,result + (last - first))` either `new_value` or `*(first + (i - result))` depending on whether the following corresponding conditions hold:

```
invoke(proj, *(first + (i - result))) == old_value
invoke(pred, invoke(proj, *(first + (i - result)))) != false
```

13 *Returns:* `{last, result + (last - first)}`.

14 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

30.6.6 Fill [alg.fill]

```
template<class ForwardIterator, class T>
  constexpr void fill(ForwardIterator first, ForwardIterator last, const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  void fill(ExecutionPolicy&& exec,
            ForwardIterator first, ForwardIterator last, const T& value);

template<class OutputIterator, class Size, class T>
  constexpr OutputIterator fill_n(OutputIterator first, Size n, const T& value);
template<class ExecutionPolicy, class ForwardIterator, class Size, class T>
  ForwardIterator fill_n(ExecutionPolicy&& exec,
                         ForwardIterator first, Size n, const T& value);
```

1 *Requires:* The expression `value` shall be writable (28.3.1) to the output iterator. The type `Size` shall be convertible to an integral type (cxxrefconv.integral, cxxrefclass.conv).

2 *Effects:* The `fill` algorithms assign `value` through all the iterators in the range `[first,last)`. The `fill_n` algorithms assign `value` through all the iterators in the range `[first,first + n)` if `n` is positive, otherwise they do nothing.

3 *Returns:* `fill_n` returns `first + n` for non-negative values of `n` and `first` for negative values.

4 *Complexity:* Exactly `last - first`, `n`, or 0 assignments, respectively.

```
namespace ranges {
  template <class T, OutputIterator<const T&> O, Sentinel<O> S>
    O fill(O first, S last, const T& value);
  template <class T, OutputRange<const T&> Rng>
    safe_iterator_t<Rng> fill(Rng&& rng, const T& value);

  template <class T, OutputIterator<const T&> O>
    O fill_n(O first, iter_difference_t<O> n, const T& value);
}
```

5 *Effects:* `fill` assigns `value` through all the iterators in the range `[first,last)`. `fill_n` assigns `value` through all the iterators in the counted range `[first,n)` if `n` is positive, otherwise it does nothing.

6 *Returns:* `last`, where `last` is `first + max(n, 0)` for `fill_n`.

7 *Complexity:* Exactly `last - first` assignments.

30.6.7 Generate [alg.generate]

```
template<class ForwardIterator, class Generator>
  constexpr void generate(ForwardIterator first, ForwardIterator last,
                          Generator gen);
template<class ExecutionPolicy, class ForwardIterator, class Generator>
  void generate(ExecutionPolicy&& exec,
```

```
                ForwardIterator first, ForwardIterator last,
                Generator gen);

template<class OutputIterator, class Size, class Generator>
  constexpr OutputIterator generate_n(OutputIterator first, Size n, Generator gen);
template<class ExecutionPolicy, class ForwardIterator, class Size, class Generator>
  ForwardIterator generate_n(ExecutionPolicy&& exec,
                             ForwardIterator first, Size n, Generator gen);
```

1 *Requires:* `gen` takes no arguments, `Size` shall be convertible to an integral type (cxxrefconv.integral, cxxrefclass.conv).

2 *Effects:* The `generate` algorithms invoke the function object `gen` and assign the return value of `gen` through all the iterators in the range `[first,last)`. The `generate_n` algorithms invoke the function object `gen` and assign the return value of `gen` through all the iterators in the range `[first,first + n)` if `n` is positive, otherwise they do nothing.

3 *Returns:* `generate_n` returns `first + n` for non-negative values of `n` and `first` for negative values.

4 *Complexity:* Exactly `last - first`, `n`, or 0 invocations of `gen` and assignments, respectively.

```
namespace ranges {
  template <Iterator O, Sentinel<O> S, CopyConstructible F>
      requires Invocable<F&> && Writable<O, invoke_result_t<F&>>
    O generate(O first, S last, F gen);
  template <class Rng, CopyConstructible F>
      requires Invocable<F&> && OutputRange<Rng, invoke_result_t<F&>>
    safe_iterator_t<Rng> generate(Rng&& rng, F gen);

  template <Iterator O, CopyConstructible F>
      requires Invocable<F&> && Writable<O, invoke_result_t<F&>>
    O generate_n(O first, iter_difference_t<O> n, F gen);
}
```

5 *Effects:* The generate algorithms invoke the function object `gen` and assign the return value of `gen` through all the iterators in the range `[first,last)`. The `generate_n` algorithm invokes the function object `gen` and assigns the return value of `gen` through all the iterators in the counted range `[first,n)` if `n` is positive, otherwise it does nothing.

6 *Returns:* `last`, where `last` is `first + max(n, 0)` for `generate_n`.

7 *Complexity:* Exactly `last - first` evaluations of `invoke(gen)` and assignments.

30.6.8 Remove [alg.remove]

```
template<class ForwardIterator, class T>
  constexpr ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                                   const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  ForwardIterator remove(ExecutionPolicy&& exec,
                         ForwardIterator first, ForwardIterator last,
                         const T& value);

template<class ForwardIterator, class Predicate>
  constexpr ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                                      Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator remove_if(ExecutionPolicy&& exec,
                            ForwardIterator first, ForwardIterator last,
```

```
                              Predicate pred);
```

1 *Requires:* The type of `*first` shall satisfy the *Cpp98MoveAssignable* requirements (25).

2 *Effects:* Eliminates all the elements referred to by iterator `i` in the range `[first,last)` for which the following corresponding conditions hold: `*i == value`, `pred(*i) != false`.

3 *Returns:* The end of the resulting range.

4 *Remarks:* Stable20.5.5.7.

5 *Complexity:* Exactly `last - first` applications of the corresponding predicate.

6 [*Note:* Each element in the range `[ret,last)`, where `ret` is the returned value, has a valid but unspecified state, because the algorithms can eliminate elements by moving from elements that were originally in that range. — *end note*]

```
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity>
    requires Permutable<I> &&
             IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T*>
    I remove(I first, S last, const T& value, Proj proj = Proj{});
  template <ForwardRange Rng, class T, class Proj = identity>
      requires Permutable<iterator_t<Rng>> &&
               IndirectRelation<ranges::equal_to<>, projected<iterator_t<Rng>, Proj>,
                                const T*>
    safe_iterator_t<Rng> remove(Rng&& rng, const T& value, Proj proj = Proj{});

  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
            IndirectUnaryPredicate<projected<I, Proj>> Pred>
      requires Permutable<I>
    I remove_if(I first, S last, Pred pred, Proj proj = Proj{});
  template <ForwardRange Rng, class Proj = identity,
            IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
      requires Permutable<iterator_t<Rng>>
    safe_iterator_t<Rng> remove_if(Rng&& rng, Pred pred, Proj proj = Proj{});
}
```

7 *Effects:* Eliminates all the elements referred to by iterator `i` in the range `[first,last)` for which the following corresponding conditions hold: `invoke(proj, *i) == value`, `invoke(pred, invoke(proj, *i)) != false`.

8 *Returns:* The end of the resulting range.

9 *Remarks:* Stable (20.5.5.7).

10 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

11 *Note:* each element in the range `[ret,last)`, where `ret` is the returned value, has a valid but unspecified state, because the algorithms can eliminate elements by moving from elements that were originally in that range.

```
template<class InputIterator, class OutputIterator, class T>
  constexpr OutputIterator
    remove_copy(InputIterator first, InputIterator last,
                OutputIterator result, const T& value);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class T>
  ForwardIterator2
    remove_copy(ExecutionPolicy&& exec,
                ForwardIterator1 first, ForwardIterator1 last,
```

```
                ForwardIterator2 result, const T& value);

template<class InputIterator, class OutputIterator, class Predicate>
  constexpr OutputIterator
    remove_copy_if(InputIterator first, InputIterator last,
                   OutputIterator result, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate>
  ForwardIterator2
    remove_copy_if(ExecutionPolicy&& exec,
                   ForwardIterator1 first, ForwardIterator1 last,
                   ForwardIterator2 result, Predicate pred);
```

12 *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap. The expression `*result = *first` shall be valid. [*Note:* For the overloads with an `ExecutionPolicy`, there may be a performance cost if `iterator_traits<ForwardIterator1>::value_type` is not `MoveConstructible` (23). — *end note*]

13 *Effects:* Copies all the elements referred to by the iterator `i` in the range `[first,last)` for which the following corresponding conditions do not hold: `*i == value`, `pred(*i) != false`.

14 *Returns:* The end of the resulting range.

15 *Complexity:* Exactly `last - first` applications of the corresponding predicate.

16 *Remarks:* Stable20.5.5.7.

```
namespace ranges {
  template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class T,
            class Proj = identity>
      requires IndirectlyCopyable<I, O> &&
               IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T*>
    tagged_pair<tag::in(I), tag::out(O)>
      remove_copy(I first, S last, O result, const T& value, Proj proj = Proj{});
  template <InputRange Rng, WeaklyIncrementable O, class T, class Proj = identity>
      requires IndirectlyCopyable<iterator_t<Rng>, O> &&
               IndirectRelation<ranges::equal_to<>, projected<iterator_t<Rng>, Proj>,
                                const T*>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
      remove_copy(Rng&& rng, O result, const T& value, Proj proj = Proj{});

  template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
            class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
      requires IndirectlyCopyable<I, O>
    tagged_pair<tag::in(I), tag::out(O)>
      remove_copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});
  template <InputRange Rng, WeaklyIncrementable O, class Proj = identity,
            IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
      requires IndirectlyCopyable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
      remove_copy_if(Rng&& rng, O result, Pred pred, Proj proj = Proj{});
}
```

17 *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

18 *Effects:* Copies all the elements referred to by the iterator `i` in the range `[first,last)` for which the following corresponding conditions do not hold: `invoke(proj, *i) == value`, `invoke(pred, invoke(proj, *i)) != false`.

19 *Returns:* A pair consisting of `last` and the end of the resulting range.

20 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

21 *Remarks:* Stable (20.5.5.7).

30.6.9 Unique [alg.unique]

```
template<class ForwardIterator>
  constexpr ForwardIterator unique(ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator unique(ExecutionPolicy&& exec,
                         ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
  constexpr ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                                   BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator, class BinaryPredicate>
  ForwardIterator unique(ExecutionPolicy&& exec,
                         ForwardIterator first, ForwardIterator last,
                         BinaryPredicate pred);
```

1 *Requires:* The comparison function shall be an equivalence relation. The type of `*first` shall satisfy the *Cpp98MoveAssignable* requirements (25).

2 *Effects:* For a nonempty range, eliminates all but the first element from every consecutive group of equivalent elements referred to by the iterator `i` in the range `[first + 1,last)` for which the following conditions hold: `*(i - 1) == *i` or `pred(*(i - 1), *i) != false`.

3 *Returns:* The end of the resulting range.

4 *Complexity:* For nonempty ranges, exactly `(last - first) - 1` applications of the corresponding predicate.

```
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
            IndirectRelation<projected<I, Proj>> R = ranges::equal_to<>>
    requires Permutable<I>
    I unique(I first, S last, R comp = R{}, Proj proj = Proj{});
  template <ForwardRange Rng, class Proj = identity,
            IndirectRelation<projected<iterator_t<Rng>, Proj>> R = ranges::equal_to<>>
    requires Permutable<iterator_t<Rng>>
    safe_iterator_t<Rng> unique(Rng&& rng, R comp = R{}, Proj proj = Proj{});
}
```

5 *Effects:* For a nonempty range, eliminates all but the first element from every consecutive group of equivalent elements referred to by the iterator `i` in the range `[first + 1,last)` for which the following conditions hold: `invoke(proj, *(i - 1)) == invoke(proj, *i)` or `invoke(pred, invoke(proj, *(i - 1)), invoke(proj, *i)) != false`.

6 *Returns:* The end of the resulting range.

7 *Complexity:* For nonempty ranges, exactly `(last - first) - 1` applications of the corresponding predicate and no more than twice as many applications of the projection.

```
template<class InputIterator, class OutputIterator>
  constexpr OutputIterator
    unique_copy(InputIterator first, InputIterator last,
                OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
```

```
  ForwardIterator2
    unique_copy(ExecutionPolicy&& exec,
                ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result);

template<class InputIterator, class OutputIterator,
         class BinaryPredicate>
  constexpr OutputIterator
    unique_copy(InputIterator first, InputIterator last,
                OutputIterator result, BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  ForwardIterator2
    unique_copy(ExecutionPolicy&& exec,
                ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result, BinaryPredicate pred);
```

8 *Requires:*

- (8.1) The comparison function shall be an equivalence relation.
- (8.2) The ranges `[first,last)` and `[result,result+(last-first))` shall not overlap.
- (8.3) The expression `*result = *first` shall be valid.
- (8.4) For the overloads with no `ExecutionPolicy`, let `T` be the value type of `InputIterator`. If `InputIterator` meets the forward iterator requirements, then there are no additional requirements for `T`. Otherwise, if `OutputIterator` meets the forward iterator requirements and its value type is the same as `T`, then `T` shall be *Cpp98CopyAssignable* (). Otherwise, `T` shall be both *Cpp98CopyConstructible* (24) and *Cpp98CopyAssignable*. [*Note:* For the overloads with an `ExecutionPolicy`, there may be a performance cost if the value type of `ForwardIterator1` is not both *Cpp98CopyConstructible* and *Cpp98CopyAssignable*. — *end note*]

9 *Effects:* Copies only the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first,last)` for which the following corresponding conditions hold: `*i == *(i - 1)` or `pred(*i, *(i - 1)) != false`.

10 *Returns:* The end of the resulting range.

11 *Complexity:* For nonempty ranges, exactly `last - first - 1` applications of the corresponding predicate.

```
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
          class Proj = identity, IndirectRelation<projected<I, Proj>> R = ranges::equal_to<>>
    requires IndirectlyCopyable<I, O> &&
            (ForwardIterator<I> ||
             (InputIterator<O> && Same<iter_value_t<I>, iter_value_t<O>>) ||
             IndirectlyCopyableStorable<I, O>)
  tagged_pair<tag::in(I), tag::out(O)>
    unique_copy(I first, S last, O result, R comp = R{}, Proj proj = Proj{});
template <InputRange Rng, WeaklyIncrementable O, class Proj = identity,
          IndirectRelation<projected<iterator_t<Rng>, Proj>> R = ranges::equal_to<>>
    requires IndirectlyCopyable<iterator_t<Rng>, O> &&
             (ForwardIterator<iterator_t<Rng>> ||
              (InputIterator<O> && Same<iter_value_t<iterator_t<Rng>>, iter_value_t<O>>) ||
              IndirectlyCopyableStorable<iterator_t<Rng>, O>)
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    unique_copy(Rng&& rng, O result, R comp = R{}, Proj proj = Proj{});
```

12 *Requires:* The ranges `[first,last)` and `[result,result+(last-first))` shall not overlap.

13 *Effects:* Copies only the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first,last)` for which the following corresponding conditions hold:

```
invoke(proj, *i) == invoke(proj, *(i - 1))
```

or

```
invoke(pred, invoke(proj, *i), invoke(proj, *(i - 1))) != false.
```

14 *Returns:* A pair consisting of `last` and the end of the resulting range.

15 *Complexity:* For nonempty ranges, exactly `last - first - 1` applications of the corresponding predicate and no more than twice as many applications of the projection.

30.6.10 Reverse [alg.reverse]

```
template<class BidirectionalIterator>
  void reverse(BidirectionalIterator first, BidirectionalIterator last);
template<class ExecutionPolicy, class BidirectionalIterator>
  void reverse(ExecutionPolicy&& exec,
               BidirectionalIterator first, BidirectionalIterator last);
```

1 *Requires:* `BidirectionalIterator` shall satisfy the `ValueSwappable` requirements?.3.11.

2 *Effects:* For each non-negative integer `i < (last - first) / 2`, applies `iter_swap` to all pairs of iterators `first + i, (last - i) - 1`.

3 *Complexity:* Exactly `(last - first)/2` swaps.

```
namespace ranges {
  template <BidirectionalIterator I, Sentinel<I> S>
      requires Permutable<I>
    I reverse(I first, S last);
  template <BidirectionalRange Rng>
      requires Permutable<iterator_t<Rng>>
    safe_iterator_t<Rng> reverse(Rng&& rng);
}
```

4 *Effects:* For each non-negative integer `i < (last - first)/2`, applies `iter_swap` to all pairs of iterators `first + i, (last - i) - 1`.

5 *Returns:* `last`.

6 *Complexity:* Exactly `(last - first)/2` swaps.

```
template<class BidirectionalIterator, class OutputIterator>
  constexpr OutputIterator
    reverse_copy(BidirectionalIterator first, BidirectionalIterator last,
                 OutputIterator result);
template<class ExecutionPolicy, class BidirectionalIterator, class ForwardIterator>
  ForwardIterator
    reverse_copy(ExecutionPolicy&& exec,
                 BidirectionalIterator first, BidirectionalIterator last,
                 ForwardIterator result);
```

7 *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

8 *Effects:* Copies the range `[first,last)` to the range `[result,result + (last - first))` such that for every non-negative integer `i < (last - first)` the following assignment takes place: `*(result + (last - first) - 1 - i) = *(first + i)`.

9 *Returns:* `result + (last - first)`.

10 *Complexity:* Exactly `last - first` assignments.

```
namespace ranges {
  template <BidirectionalIterator I, Sentinel<I> S, WeaklyIncrementable O>
      requires IndirectlyCopyable<I, O>
    tagged_pair<tag::in(I), tag::out(O)> reverse_copy(I first, S last, O result);
  template <BidirectionalRange Rng, WeaklyIncrementable O>
      requires IndirectlyCopyable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)> reverse_copy(Rng&& rng, O result);
}
```

11 *Effects:* Copies the range `[first,last)` to the range `[result,result+(last-first))` such that for every non-negative integer `i < (last - first)` the following assignment takes place: `*(result + (last - first) - 1 - i) = *(first + i)`.

12 *Requires:* The ranges `[first,last)` and `[result,result+(last-first))` shall not overlap.

13 *Returns:* `{last, result + (last - first)}`.

14 *Complexity:* Exactly `last - first` assignments.

30.6.11 Rotate [alg.rotate]

```
template<class ForwardIterator>
  ForwardIterator
    rotate(ForwardIterator first, ForwardIterator middle, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator
    rotate(ExecutionPolicy&& exec,
           ForwardIterator first, ForwardIterator middle, ForwardIterator last);
```

1 *Requires:* `[first,middle)` and `[middle,last)` shall be valid ranges. `ForwardIterator` shall satisfy the `ValueSwappable` requirements?.3.11. The type of `*first` shall satisfy the *Cpp98MoveConstructible* (23) and *Cpp98MoveAssignable* (25) requirements.

2 *Effects:* For each non-negative integer `i < (last - first)`, places the element from the position `first + i` into position `first + (i + (last - middle)) % (last - first)`.

3 *Returns:* `first + (last - middle)`.

4 *Remarks:* This is a left rotate.

5 *Complexity:* At most `last - first` swaps.

```
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S>
      requires Permutable<I>
    tagged_pair<tag::begin(I), tag::end(I)> rotate(I first, I middle, S last);
  template <ForwardRange Rng>
      requires Permutable<iterator_t<Rng>>
    tagged_pair<tag::begin(safe_iterator_t<Rng>), tag::end(safe_iterator_t<Rng>)>
      rotate(Rng&& rng, iterator_t<Rng> middle);
}
```

6 *Effects:* For each non-negative integer `i < (last - first)`, places the element from the position `first + i` into position `first + (i + (last - middle)) % (last - first)`.

7 *Returns:* `{first + (last - middle), last}`.

8 *Remarks:* This is a left rotate.

9 *Requires:* `[first,middle)` and `[middle,last)` shall be valid ranges.

10 *Complexity:* At most `last - first` swaps.

```
template<class ForwardIterator, class OutputIterator>
  constexpr OutputIterator
    rotate_copy(ForwardIterator first, ForwardIterator middle, ForwardIterator last,
                OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2
    rotate_copy(ExecutionPolicy&& exec,
                ForwardIterator1 first, ForwardIterator1 middle, ForwardIterator1 last,
                ForwardIterator2 result);
```

11 *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

12 *Effects:* Copies the range `[first,last)` to the range `[result,result + (last - first))` such that for each non-negative integer `i < (last - first)` the following assignment takes place: `*(result + i) = *(first + (i + (middle - first)) % (last - first))`.

13 *Returns:* `result + (last - first)`.

14 *Complexity:* Exactly `last - first` assignments.

```
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, WeaklyIncrementable O>
      requires IndirectlyCopyable<I, O>
    tagged_pair<tag::in(I), tag::out(O)> rotate_copy(I first, I middle, S last, O result);
  template <ForwardRange Rng, WeaklyIncrementable O>
      requires IndirectlyCopyable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
      rotate_copy(Rng&& rng, iterator_t<Rng> middle, O result);
}
```

15 *Effects:* Copies the range `[first,last)` to the range `[result,result + (last - first))` such that for each non-negative integer `i < (last - first)` the following assignment takes place: `*(result + i) = *(first + (i + (middle - first)) % (last - first))`.

16 *Returns:* `{last, result + (last - first)}`.

17 *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

18 *Complexity:* Exactly `last - first` assignments.

30.6.12 Sample [alg.random.sample]

```
template<class PopulationIterator, class SampleIterator,
         class Distance, class UniformRandomBitGenerator>
  SampleIterator sample(PopulationIterator first, PopulationIterator last,
                        SampleIterator out, Distance n,
                        UniformRandomBitGenerator&& g);
```

1 *Requires:*

- (1.1) `PopulationIterator` shall satisfy the requirements of an input iterator (28.3.5.2).
- (1.2) `SampleIterator` shall satisfy the requirements of an output iterator (28.3.5.3).
- (1.3) `SampleIterator` shall satisfy the additional requirements of a random access iterator (28.3.5.6) unless `PopulationIterator` satisfies the additional requirements of a forward iterator (28.3.5.4).
- (1.4) `PopulationIterator`'s value type shall be writable (28.3.1) to `out`.

(1.5) — `Distance` shall be an integer type.

(1.6) — `remove_reference_t<UniformRandomBitGenerator>` shall satisfy the requirements of a uniform random bit generator type29.6.1.3 whose return type is convertible to `Distance`.

(1.7) — `out` shall not be in the range `[first,last)`.

2 *Effects:* Copies `min(last - first, n)` elements (the *sample*) from `[first,last)` (the *population*) to `out` such that each possible sample has equal probability of appearance. [*Note:* Algorithms that obtain such effects include *selection sampling* and *reservoir sampling*. — *end note*]

3 *Returns:* The end of the resulting sample range.

4 *Complexity:* $\mathcal{O}$(`last - first`).

5 *Remarks:*

(5.1) — Stable if and only if `PopulationIterator` satisfies the requirements of a forward iterator.

(5.2) — To the extent that the implementation of this function makes use of random numbers, the object `g` shall serve as the implementation's source of randomness.

30.6.13 Shuffle [alg.random.shuffle]

```
template<class RandomAccessIterator, class UniformRandomBitGenerator>
  void shuffle(RandomAccessIterator first,
               RandomAccessIterator last,
               UniformRandomBitGenerator&& g);
```

1 *Requires:* `RandomAccessIterator` shall satisfy the `ValueSwappable` requirements?.3.11. The type `remove_reference_t<UniformRandomBitGenerator>` shall satisfy the requirements of a uniform random bit generator29.6.1.3 type whose return type is convertible to `iterator_traits<RandomAccessIterator>::difference_type`.

2 *Effects:* Permutes the elements in the range `[first,last)` such that each possible permutation of those elements has equal probability of appearance.

3 *Complexity:* Exactly `(last - first) - 1` swaps.

4 *Remarks:* To the extent that the implementation of this function makes use of random numbers, the object `g` shall serve as the implementation's source of randomness.

```
template <RandomAccessIterator I, Sentinel<I> S, class Gen>
    requires Permutable<I> && UniformRandomBitGenerator<remove_reference_t<Gen>> &&
             ConvertibleTo<invoke_result_t<Gen&>, iter_difference_t<I>>
  I shuffle(I first, S last, Gen&& g);
template <RandomAccessRange Rng, class Gen>
    requires Permutable<I> && UniformRandomBitGenerator<remove_reference_t<Gen>> &&
             ConvertibleTo<invoke_result_t<Gen&>, iter_difference_t<I>>
  safe_iterator_t<Rng> shuffle(Rng&& rng, Gen&& g);
```

5 *Effects:* Permutes the elements in the range `[first,last)` such that each possible permutation of those elements has equal probability of appearance.

6 *Complexity:* Exactly `(last - first) - 1` swaps.

7 *Returns:* `last`

8 *Remarks:* To the extent that the implementation of this function makes use of random numbers, the object `g` shall serve as the implementation's source of randomness.

30.7 Sorting and related operations [alg.sorting]

1 All the operations in 30.7 directly in namespace `std` have two versions: one that takes a function object of type `Compare` and one that uses an `operator<`.

2 `Compare` is a function object type. The return value of the function call operation applied to an object of type `Compare`, when contextually converted to `bool7`, yields `true` if the first argument of the call is less than the second, and `false` otherwise. `Compare comp` is used throughout for algorithms assuming an ordering relation. It is assumed that `comp` will not apply any non-constant function through the dereferenced iterator.

3 For all algorithms that take `Compare`, there is a version that uses `operator<` instead. That is, `comp(*i, *j) != false` defaults to `*i < *j != false`. For algorithms other than those described in 30.7.3, `comp` shall induce a strict weak ordering on the values.

[Editor's note: This specification of strict weak ordering may be redundant with the specification of strict weak ordering in .]

4 The term *strict* refers to the requirement of an irreflexive relation (`!comp(x, x)` for all `x`), and the term *weak* to requirements that are not as strong as those for a total ordering, but stronger than those for a partial ordering. If we define `equiv(a, b)` as `!comp(a, b) && !comp(b, a)`, then the requirements are that `comp` and `equiv` both be transitive relations:

(4.1) — `comp(a, b) && comp(b, c)` implies `comp(a, c)`

(4.2) — `equiv(a, b) && equiv(b, c)` implies `equiv(a, c)`

[*Note:* Under these conditions, it can be shown that

(4.3) — `equiv` is an equivalence relation

(4.4) — `comp` induces a well-defined relation on the equivalence classes determined by `equiv`

(4.5) — The induced relation is a strict total ordering.

— *end note*]

5 A sequence is *sorted with respect to a comparator* `comp` if for every iterator `i` pointing to the sequence and every non-negative integer `n` such that `i + n` is a valid iterator pointing to an element of the sequence, `comp(*(i + n), *i) == false`.

6 A sequence `[start,finish)` is *partitioned with respect to an expression* `f(e)` if there exists an integer `n` such that for all `0 <= i < (finish - start)`, `f(*(start + i))` is `true` if and only if `i < n`.

7 In the descriptions of the functions that deal with ordering relationships we frequently use a notion of equivalence to describe concepts such as stability. The equivalence to which we refer is not necessarily an `operator==`, but an equivalence relation induced by the strict weak ordering. That is, two elements `a` and `b` are considered equivalent if and only if `!(a < b) && !(b < a)`.

30.7.1 Sorting [alg.sort]

30.7.1.1 `sort` [sort]

```
template<class RandomAccessIterator>
  void sort(RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator>
  void sort(ExecutionPolicy&& exec,
            RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void sort(RandomAccessIterator first, RandomAccessIterator last,
```

```
            Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  void sort(ExecutionPolicy&& exec,
            RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);
```

1 *Requires:* `RandomAccessIterator` shall satisfy the `ValueSwappable` requirements?.3.11. The type of `*first` shall satisfy the *Cpp98MoveConstructible* (23) and *Cpp98MoveAssignable* (25) requirements.

2 *Effects:* Sorts the elements in the range `[first,last)`.

3 *Complexity:* $\mathcal{O}(N \log N)$ comparisons, where $N =$ `last - first`.

```
namespace ranges {
  template <RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
            class Proj = identity>
      requires Sortable<I, Comp, Proj>
    I sort(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
      requires Sortable<iterator_t<Rng>, Comp, Proj>
    safe_iterator_t<Rng> sort(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

4 *Effects:* Sorts the elements in the range `[first,last)`.

5 *Returns:* `last`.

6 *Complexity:* $\mathcal{O}(N \log(N))$ (where N `== last - first`) comparisons, and twice as many applications of the projection.

30.7.1.2 `stable_sort` [stable.sort]

```
template<class RandomAccessIterator>
  void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator>
  void stable_sort(ExecutionPolicy&& exec,
                   RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  void stable_sort(ExecutionPolicy&& exec,
                   RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);
```

1 *Requires:* `RandomAccessIterator` shall satisfy the `ValueSwappable` requirements?.3.11. The type of `*first` shall satisfy the *Cpp98MoveConstructible* (23) and *Cpp98MoveAssignable* (25) requirements.

2 *Effects:* Sorts the elements in the range `[first,last)`.

3 *Complexity:* At most $N \log^2(N)$ comparisons, where $N =$ `last - first`, but only $N \log N$ comparisons if there is enough extra memory.

4 *Remarks:* Stable20.5.5.7.

```
namespace ranges {
  template <RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
            class Proj = identity>
      requires Sortable<I, Comp, Proj>
    I stable_sort(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
```

```
  template <RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
      requires Sortable<iterator_t<Rng>, Comp, Proj>
    safe_iterator_t<Rng> stable_sort(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

5 *Effects:* Sorts the elements in the range `[first,last)`.

6 *Returns:* `last`.

7 *Complexity:* Let N `== last - first`. If enough extra memory is available, $N \log(N)$ comparisons. Otherwise, at most $N \log^2(N)$ comparisons. In either case, twice as many applications of the projection as the number of comparisons.

8 *Remarks:* Stable (20.5.5.7).

30.7.1.3 `partial_sort` [partial.sort]

```
template<class RandomAccessIterator>
  void partial_sort(RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator>
  void partial_sort(ExecutionPolicy&& exec,
                    RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void partial_sort(RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last,
                    Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  void partial_sort(ExecutionPolicy&& exec,
                    RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last,
                    Compare comp);
```

1 *Requires:* `RandomAccessIterator` shall satisfy the `ValueSwappable` requirements?.3.11. The type of `*first` shall satisfy the *Cpp98MoveConstructible* (23) and *Cpp98MoveAssignable* (25) requirements.

2 *Effects:* Places the first `middle - first` sorted elements from the range `[first,last)` into the range `[first,middle)`. The rest of the elements in the range `[middle,last)` are placed in an unspecified order.

3 *Complexity:* Approximately `(last - first) * log(middle - first)` comparisons.

```
namespace ranges {
  template <RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
            class Proj = identity>
      requires Sortable<I, Comp, Proj>
    I partial_sort(I first, I middle, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
      requires Sortable<iterator_t<Rng>, Comp, Proj>
    safe_iterator_t<Rng> partial_sort(Rng&& rng, iterator_t<Rng> middle, Comp comp = Comp{},
                                      Proj proj = Proj{});
}
```

4 *Effects:* Places the first `middle - first` sorted elements from the range `[first,last)` into the range `[first,middle)`. The rest of the elements in the range `[middle,last)` are placed in an unspecified order.

5 *Returns:* `last`.

6 *Complexity:* It takes approximately `(last - first) * log(middle - first)` comparisons, and exactly twice as many applications of the projection.

30.7.1.4 `partial_sort_copy` [partial.sort.copy]

```
template<class InputIterator, class RandomAccessIterator>
  RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last);
template<class ExecutionPolicy, class ForwardIterator, class RandomAccessIterator>
  RandomAccessIterator
    partial_sort_copy(ExecutionPolicy&& exec,
                      ForwardIterator first, ForwardIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last);

template<class InputIterator, class RandomAccessIterator,
         class Compare>
  RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last,
                      Compare comp);
template<class ExecutionPolicy, class ForwardIterator, class RandomAccessIterator,
         class Compare>
  RandomAccessIterator
    partial_sort_copy(ExecutionPolicy&& exec,
                      ForwardIterator first, ForwardIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last,
                      Compare comp);
```

1 *Requires:* `RandomAccessIterator` shall satisfy the `ValueSwappable` requirements?.3.11. The type of `*result_first` shall satisfy the *Cpp98MoveConstructible* (23) and `MoveAssignable` (25) requirements.

2 *Effects:* Places the first `min(last - first, result_last - result_first)` sorted elements into the range `[result_first,result_first + min(last - first, result_last - result_first))`.

3 *Returns:* The smaller of: `result_last` or `result_first + (last - first)`.

4 *Complexity:* Approximately `(last - first) * log(min(last - first, result_last - result_first))` comparisons.

```
namespace ranges {
  template <InputIterator I1, Sentinel<I1> S1, RandomAccessIterator I2, Sentinel<I2> S2,
            class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
      requires IndirectlyCopyable<I1, I2> && Sortable<I2, Comp, Proj2> &&
               IndirectStrictWeakOrder<Comp, projected<I1, Proj1>, projected<I2, Proj2>>
    I2 partial_sort_copy(I1 first, S1 last, I2 result_first, S2 result_last,
                         Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

```
  template <InputRange Rng1, RandomAccessRange Rng2, class Comp = ranges::less<>,
            class Proj1 = identity, class Proj2 = identity>
      requires IndirectlyCopyable<iterator_t<Rng1>, iterator_t<Rng2>> &&
               Sortable<iterator_t<Rng2>, Comp, Proj2> &&
               IndirectStrictWeakOrder<Comp, projected<iterator_t<Rng1>, Proj1>,
                                             projected<iterator_t<Rng2>, Proj2>>
    safe_iterator_t<Rng2>
      partial_sort_copy(Rng1&& rng, Rng2&& result_rng, Comp comp = Comp{},
                        Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

5 *Effects:* Places the first `min(last - first, result_last - result_first)` sorted elements into the range `[result_first,result_first + min(last - first, result_last - result_first))`.

6 *Returns:* The smaller of: `result_last` or `result_first + (last - first)`.

7 *Complexity:* Approximately

```
(last - first) * log(min(last - first, result_last - result_first))
```

comparisons, and exactly twice as many applications of the projection.

30.7.1.5 `is_sorted` [is.sorted]

```
template<class ForwardIterator>
  constexpr bool is_sorted(ForwardIterator first, ForwardIterator last);
```

1 *Returns:* `is_sorted_until(first, last) == last`.

```
template<class ExecutionPolicy, class ForwardIterator>
  bool is_sorted(ExecutionPolicy&& exec,
                 ForwardIterator first, ForwardIterator last);
```

2 *Returns:* `is_sorted_until(std::forward<ExecutionPolicy>(exec), first, last) == last`.

```
template<class ForwardIterator, class Compare>
  constexpr bool is_sorted(ForwardIterator first, ForwardIterator last,
                           Compare comp);
```

3 *Returns:* `is_sorted_until(first, last, comp) == last`.

```
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  bool is_sorted(ExecutionPolicy&& exec,
                 ForwardIterator first, ForwardIterator last,
                 Compare comp);
```

4 *Returns:*

```
is_sorted_until(std::forward<ExecutionPolicy>(exec), first, last, comp) == last
```

```
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
            IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less<>>
    bool is_sorted(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <ForwardRange Rng, class Proj = identity,
            IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    bool is_sorted(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

5 *Returns:* `is_sorted_until(first, last, comp, proj) == last`

```
template<class ForwardIterator>
  constexpr ForwardIterator
    is_sorted_until(ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator
    is_sorted_until(ExecutionPolicy&& exec,
                    ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
  constexpr ForwardIterator
    is_sorted_until(ForwardIterator first, ForwardIterator last,
                    Compare comp);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  ForwardIterator
    is_sorted_until(ExecutionPolicy&& exec,
                    ForwardIterator first, ForwardIterator last,
                    Compare comp);
```

6 *Returns:* If `(last - first) < 2`, returns `last`. Otherwise, returns the last iterator `i` in `[first, last]` for which the range `[first,i)` is sorted.

7 *Complexity:* Linear.

```
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
            IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less<>>
    I is_sorted_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <ForwardRange Rng, class Proj = identity,
            IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    safe_iterator_t<Rng> is_sorted_until(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

8 *Returns:* If `distance(first, last) < 2`, returns `last`. Otherwise, returns the last iterator `i` in `[first,last]` for which the range `[first,i)` is sorted.

9 *Complexity:* Linear.

30.7.2 Nth element [alg.nth.element]

```
template<class RandomAccessIterator>
  void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator>
  void nth_element(ExecutionPolicy&& exec,
                   RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last,  Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  void nth_element(ExecutionPolicy&& exec,
                   RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last, Compare comp);
```

1 *Requires:* `RandomAccessIterator` shall satisfy the `ValueSwappable` requirements?.3.11. The type of `*first` shall satisfy the *Cpp98MoveConstructible* (23) and *Cpp98MoveAssignable* (25) requirements.

2 *Effects:* After `nth_element` the element in the position pointed to by `nth` is the element that would be in that position if the whole range were sorted, unless `nth == last`. Also for every iterator `i` in the range `[first,nth)` and every iterator `j` in the range `[nth,last)` it holds that: `!(*j < *i)` or `comp(*j, *i) == false`.

3 *Complexity:* For the overloads with no `ExecutionPolicy`, linear on average. For the overloads with an `ExecutionPolicy`, $\mathcal{O}(N)$ applications of the predicate, and $\mathcal{O}(N \log N)$ swaps, where $N =$ `last - first`.

```
namespace ranges {
  template <RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
            class Proj = identity>
      requires Sortable<I, Comp, Proj>
    I nth_element(I first, I nth, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
      requires Sortable<iterator_t<Rng>, Comp, Proj>
    safe_iterator_t<Rng> nth_element(Rng&& rng, iterator_t<Rng> nth, Comp comp = Comp{},
                                     Proj proj = Proj{});
}
```

4 After `nth_element` the element in the position pointed to by `nth` is the element that would be in that position if the whole range were sorted, unless `nth == last`. Also for every iterator `i` in the range `[first,nth)` and every iterator `j` in the range `[nth,last)` it holds that: `invoke(comp, invoke(proj, *j), invoke(proj, *i)) == false`.

5 *Returns:* `last`.

6 *Complexity:* Linear on average.

30.7.3 Binary search [alg.binary.search]

1 All of the algorithms in this subclause are versions of binary search and assume that the sequence being searched is partitioned with respect to an expression formed by binding the search key to an argument of the implied or explicit comparison function. They work on non-random access iterators minimizing the number of comparisons, which will be logarithmic for all types of iterators. They are especially appropriate for random access iterators, because these algorithms do a logarithmic number of steps through the data structure. For non-random access iterators they execute a linear number of steps.

30.7.3.1 `lower_bound` [lower.bound]

```
template<class ForwardIterator, class T>
  constexpr ForwardIterator
    lower_bound(ForwardIterator first, ForwardIterator last,
                const T& value);

template<class ForwardIterator, class T, class Compare>
  constexpr ForwardIterator
    lower_bound(ForwardIterator first, ForwardIterator last,
                const T& value, Compare comp);
```

1 *Requires:* The elements `e` of `[first,last)` shall be partitioned with respect to the expression `e < value` or `comp(e, value)`.

2 *Returns:* The furthermost iterator `i` in the range `[first,last]` such that for every iterator `j` in the range `[first,i)` the following corresponding conditions hold: `*j < value` or `comp(*j, value) != false`.

3 *Complexity:* At most $\log_2($`last - first`$) + \mathcal{O}(1)$ comparisons.

```
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
            IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = ranges::less<>>
    I lower_bound(I first, S last, const T& value, Comp comp = Comp{},
                  Proj proj = Proj{});
  template <ForwardRange Rng, class T, class Proj = identity,
            IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    safe_iterator_t<Rng> lower_bound(Rng&& rng, const T& value, Comp comp = Comp{},
                                     Proj proj = Proj{});
}
```

4 *Requires:* The elements `e` of `[first,last)` shall be partitioned with respect to the expression `invoke(comp, invoke(proj, e), value)`.

5 *Returns:* The furthermost iterator `i` in the range `[first,last]` such that for every iterator `j` in the range `[first,i)` the following corresponding condition holds: `invoke(comp, invoke(proj, *j), value) != false`.

6 *Complexity:* At most $\log_2($`last - first`$) + \mathcal{O}(1)$ applications of the comparison function and projection.

30.7.3.2 upper_bound [upper.bound]

```
template<class ForwardIterator, class T>
  constexpr ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last,
                const T& value);

template<class ForwardIterator, class T, class Compare>
  constexpr ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last,
                const T& value, Compare comp);
```

1 *Requires:* The elements `e` of `[first,last)` shall be partitioned with respect to the expression `!(value < e)` or `!comp(value, e)`.

2 *Returns:* The furthermost iterator `i` in the range `[first,last]` such that for every iterator `j` in the range `[first,i)` the following corresponding conditions hold: `!(value < *j)` or `comp(value, *j) == false`.

3 *Complexity:* At most $\log_2($`last - first`$) + \mathcal{O}(1)$ comparisons.

```
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
            IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = ranges::less<>>
    I upper_bound(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

  template <ForwardRange Rng, class T, class Proj = identity,
            IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    safe_iterator_t<Rng>
      upper_bound(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
}
```

4 *Requires:* The elements `e` of `[first,last)` shall be partitioned with respect to the expression `!invoke(comp, value, invoke(proj, e))`.

5 *Returns:* The furthermost iterator `i` in the range `[first,last]` such that for every iterator `j` in the range `[first,i)` the following corresponding condition holds: `invoke(comp, value, invoke(proj, *j)) == false`.

6 *Complexity:* At most $\log_2($`last - first`$) + \mathcal{O}(1)$ applications of the comparison function and projection.

30.7.3.3 `equal_range` [equal.range]

```
template<class ForwardIterator, class T>
  constexpr pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first,
                ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
  constexpr pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first,
                ForwardIterator last, const T& value,
                Compare comp);
```

1 *Requires:* The elements `e` of `[first,last)` shall be partitioned with respect to the expressions `e < value` and `!(value < e)` or `comp(e, value)` and `!comp(value, e)`. Also, for all elements `e` of `[first, last)`, `e < value` shall imply `!(value < e)` or `comp(e, value)` shall imply `!comp(value, e)`.

2 *Returns:*

```
make_pair(lower_bound(first, last, value),
          upper_bound(first, last, value))
```

or

```
make_pair(lower_bound(first, last, value, comp),
          upper_bound(first, last, value, comp))
```

3 *Complexity:* At most $2 * \log_2($`last - first`$) + \mathcal{O}(1)$ comparisons.

```
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
            IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = ranges::less<>>
    tagged_pair<tag::begin(I), tag::end(I)>
      equal_range(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
  template <ForwardRange Rng, class T, class Proj = identity,
            IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    tagged_pair<tag::begin(safe_iterator_t<Rng>), tag::end(safe_iterator_t<Rng>)>
      equal_range(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
}
```

4 *Requires:* The elements `e` of `[first,last)` shall be partitioned with respect to the expressions `invoke(comp, invoke(proj, e), value)` and `!invoke(comp, value, invoke(proj, e))`. Also, for all elements `e` of `[first, last)`, `invoke(comp, invoke(proj, e), value)` shall imply `!invoke(comp, value, invoke(proj, e))`.

5 *Returns:*

```
{lower_bound(first, last, value, comp, proj),
 upper_bound(first, last, value, comp, proj)}
```

6 *Complexity:* At most $2 * \log_2($`last - first`$) + \mathcal{O}(1)$ applications of the comparison function and projection.

30.7.3.4 `binary_search` [binary.search]

```
template<class ForwardIterator, class T>
  constexpr bool
    binary_search(ForwardIterator first, ForwardIterator last,
                  const T& value);

template<class ForwardIterator, class T, class Compare>
  constexpr bool
    binary_search(ForwardIterator first, ForwardIterator last,
                  const T& value, Compare comp);
```

1 *Requires:* The elements `e` of `[first,last)` shall be partitioned with respect to the expressions `e < value` and `!(value < e)` or `comp(e, value)` and `!comp(value, e)`. Also, for all elements `e` of `[first, last)`, `e < value` shall imply `!(value < e)` or `comp(e, value)` shall imply `!comp(value, e)`.

2 *Returns:* `true` if there is an iterator `i` in the range `[first,last)` that satisfies the corresponding conditions: `!(*i < value) && !(value < *i)` or `comp(*i, value) == false && comp(value, *i) == false`.

3 *Complexity:* At most $\log_2($`last - first`$) + \mathcal{O}(1)$ comparisons.

```
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
            IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = ranges::less<>>
    bool binary_search(I first, S last, const T& value, Comp comp = Comp{},
                       Proj proj = Proj{});
  template <ForwardRange Rng, class T, class Proj = identity,
            IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    bool binary_search(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
}
```

4 *Requires:* The elements `e` of `[first,last)` are partitioned with respect to the expressions `invoke(comp, invoke(proj, e), value)` and `!invoke(comp, value, invoke(proj, e))`. Also, for all elements `e` of `[first, last)`, `invoke(comp, invoke(proj, e), value)` shall imply `!invoke(comp, value, invoke(proj, e))`.

5 *Returns:* `true` if there is an iterator `i` in the range `[first,last)` that satisfies the corresponding conditions: `invoke(comp, invoke(proj, *i), value) == false && invoke(comp, value, invoke(proj, *i)) == false`.

6 *Complexity:* At most $\log_2($`last - first`$) + \mathcal{O}(1)$ applications of the comparison function and projection.

30.7.4 Partitions [alg.partitions]

```
template<class InputIterator, class Predicate>
  constexpr bool is_partitioned(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  bool is_partitioned(ExecutionPolicy&& exec,
                      ForwardIterator first, ForwardIterator last, Predicate pred);
```

1 *Requires:* For the overload with no `ExecutionPolicy`, `InputIterator`'s value type shall be convertible to `Predicate`'s argument type. For the overload with an `ExecutionPolicy`, `ForwardIterator`'s value type shall be convertible to `Predicate`'s argument type.

2 *Returns:* `true` if `[first,last)` is empty or if the elements `e` of `[first,last)` are partitioned with respect to the expression `pred(e)`.

3 *Complexity:* Linear. At most `last - first` applications of `pred`.

```
namespace ranges {
  template <InputIterator I, Sentinel<I> S, class Proj = identity,
            IndirectUnaryPredicate<projected<I, Proj>> Pred>
    bool is_partitioned(I first, S last, Pred pred, Proj proj = Proj{});
  template <InputRange Rng, class Proj = identity,
            IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    bool is_partitioned(Rng&& rng, Pred pred, Proj proj = Proj{});
}
```

4 *Returns:* `true` if `[first,last)` is empty or if `[first,last)` is partitioned by `pred` and `proj`, i.e. if all iterators `i` for which `invoke(pred, invoke(proj, *i)) != false` come before those that do not, for every `i` in `[first,last)`.

5 *Complexity:* Linear. At most `last - first` applications of `pred` and `proj`.

```
template<class ForwardIterator, class Predicate>
  ForwardIterator
    partition(ForwardIterator first, ForwardIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator
    partition(ExecutionPolicy&& exec,
              ForwardIterator first, ForwardIterator last, Predicate pred);
```

6 *Requires:* `ForwardIterator` shall satisfy the `ValueSwappable` requirements?.3.11.

7 *Effects:* Places all the elements in the range `[first,last)` that satisfy `pred` before all the elements that do not satisfy it.

8 *Returns:* An iterator `i` such that for every iterator `j` in the range `[first,i)` `pred(*j) != false`, and for every iterator `k` in the range `[i,last)`, `pred(*k) == false`.

9 *Complexity:* Let $N =$ `last - first`:

(9.1) — For the overload with no `ExecutionPolicy`, exactly N applications of the predicate. At most $N/2$ swaps if `ForwardIterator` meets the `BidirectionalIterator` requirements and at most N swaps otherwise.

(9.2) — For the overload with an `ExecutionPolicy`, $\mathcal{O}(N \log N)$ swaps and $\mathcal{O}(N)$ applications of the predicate.

```
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
            IndirectUnaryPredicate<projected<I, Proj>> Pred>
      requires Permutable<I>
    I partition(I first, S last, Pred pred, Proj proj = Proj{});
  template <ForwardRange Rng, class Proj = identity,
            IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
      requires Permutable<iterator_t<Rng>>
    safe_iterator_t<Rng> partition(Rng&& rng, Pred pred, Proj proj = Proj{});
}
```

10 *Effects:* Permutes the elements in the range `[first,last)` such that there exists an iterator `i` such that for every iterator `j` in the range `[first,i)` `invoke(pred, invoke(proj, *j)) != false`, and for every iterator `k` in the range `[i,last)`, `invoke(pred, invoke(proj, *k)) == false`.

11 *Returns:* An iterator `i` such that for every iterator `j` in the range `[first,i)` `invoke(pred, invoke(proj, *j)) != false`, and for every iterator `k` in the range `[i,last)`, `invoke(pred, invoke(proj, *k)) == false`.

12 *Complexity:* If I meets the requirements for a BidirectionalIterator, at most `(last - first) / 2` swaps; otherwise at most `last - first` swaps. Exactly `last - first` applications of the predicate and projection.

```
template<class BidirectionalIterator, class Predicate>
  BidirectionalIterator
    stable_partition(BidirectionalIterator first, BidirectionalIterator last, Predicate pred);
template<class ExecutionPolicy, class BidirectionalIterator, class Predicate>
  BidirectionalIterator
    stable_partition(ExecutionPolicy&& exec,
                     BidirectionalIterator first, BidirectionalIterator last, Predicate pred);
```

13 *Requires:* `BidirectionalIterator` shall satisfy the `ValueSwappable` requirements?.3.11. The type of `*first` shall satisfy the *Cpp98MoveConstructible* (23) and *Cpp98MoveAssignable* (25) requirements.

14 *Effects:* Places all the elements in the range `[first,last)` that satisfy `pred` before all the elements that do not satisfy it.

15 *Returns:* An iterator `i` such that for every iterator `j` in the range `[first,i)`, `pred(*j) != false`, and for every iterator `k` in the range `[i,last)`, `pred(*k) == false`. The relative order of the elements in both groups is preserved.

16 *Complexity:* Let N = `last - first`:

(16.1) — For the overload with no `ExecutionPolicy`, at most $N \log N$ swaps, but only $\mathcal{O}(N)$ swaps if there is enough extra memory. Exactly N applications of the predicate.

(16.2) — For the overload with an `ExecutionPolicy`, $\mathcal{O}(N \log N)$ swaps and $\mathcal{O}(N)$ applications of the predicate.

```
namespace ranges {
  template <BidirectionalIterator I, Sentinel<I> S, class Proj = identity,
            IndirectUnaryPredicate<projected<I, Proj>> Pred>
      requires Permutable<I>
    I stable_partition(I first, S last, Pred pred, Proj proj = Proj{});
  template <BidirectionalRange Rng, class Proj = identity,
            IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
      requires Permutable<iterator_t<Rng>>
    safe_iterator_t<Rng> stable_partition(Rng&& rng, Pred pred, Proj proj = Proj{});
}
```

17 *Effects:* Permutes the elements in the range `[first,last)` such that there exists an iterator `i` such that for every iterator `j` in the range `[first,i)` `invoke(pred, invoke(proj, *j)) != false`, and for every iterator `k` in the range `[i,last)`, `invoke(pred, invoke(proj, *k)) == false`.

18 *Returns:* An iterator `i` such that for every iterator `j` in the range `[first,i)`, `invoke(pred, invoke(proj, *j)) != false`, and for every iterator `k` in the range `[i,last)`, `invoke(pred, invoke(proj, *k)) == false`. The relative order of the elements in both groups is preserved.

19 *Complexity:* At most `(last - first) * log(last - first)` swaps, but only linear number of swaps if there is enough extra memory. Exactly `last - first` applications of the predicate and projection.

```
template<class InputIterator, class OutputIterator1,
         class OutputIterator2, class Predicate>
  constexpr pair<OutputIterator1, OutputIterator2>
    partition_copy(InputIterator first, InputIterator last,
                   OutputIterator1 out_true, OutputIterator2 out_false, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class ForwardIterator1,
         class ForwardIterator2, class Predicate>
```

```
pair<ForwardIterator1, ForwardIterator2>
  partition_copy(ExecutionPolicy&& exec,
                 ForwardIterator first, ForwardIterator last,
                 ForwardIterator1 out_true, ForwardIterator2 out_false, Predicate pred);
```

20 *Requires:*

- (20.1) For the overload with no `ExecutionPolicy`, `InputIterator`'s value type shall be *Cpp98CopyAssignable* (), and shall be writable (28.3.1) to the `out_true` and `out_false` `OutputIterator`s, and shall be convertible to `Predicate`'s argument type.
- (20.2) For the overload with an `ExecutionPolicy`, `ForwardIterator`'s value type shall be `CopyAssignable`, and shall be writable to the `out_true` and `out_false` `ForwardIterator`s, and shall be convertible to `Predicate`'s argument type. [*Note:* There may be a performance cost if `ForwardIterator`'s value type is not *Cpp98CopyConstructible*. — *end note*]
- (20.3) For both overloads, the input range shall not overlap with either of the output ranges.

21 *Effects:* For each iterator `i` in `[first,last)`, copies `*i` to the output range beginning with `out_true` if `pred(*i)` is `true`, or to the output range beginning with `out_false` otherwise.

22 *Returns:* A pair `p` such that `p.first` is the end of the output range beginning at `out_true` and `p.second` is the end of the output range beginning at `out_false`.

23 *Complexity:* Exactly `last - first` applications of `pred`.

```
namespace ranges {
  template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O1, WeaklyIncrementable O2,
            class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
      requires IndirectlyCopyable<I, O1> && IndirectlyCopyable<I, O2>
    tagged_tuple<tag::in(I), tag::out1(O1), tag::out2(O2)>
      partition_copy(I first, S last, O1 out_true, O2 out_false, Pred pred,
                     Proj proj = Proj{});
  template <InputRange Rng, WeaklyIncrementable O1, WeaklyIncrementable O2, class Proj = identity,
            IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
      requires IndirectlyCopyable<iterator_t<Rng>, O1> &&
               IndirectlyCopyable<iterator_t<Rng>, O2>
    tagged_tuple<tag::in(safe_iterator_t<Rng>), tag::out1(O1), tag::out2(O2)>
      partition_copy(Rng&& rng, O1 out_true, O2 out_false, Pred pred, Proj proj = Proj{});
}
```

24 *Requires:* The input range shall not overlap with either of the output ranges.

25 *Effects:* For each iterator `i` in `[first,last)`, copies `*i` to the output range beginning with `out_true` if `invoke(pred, invoke(proj, *i))` is `true`, or to the output range beginning with `out_false` otherwise.

26 *Returns:* A tuple `p` such that `get<0>(p)` is `last`, `get<1>(p)` is the end of the output range beginning at `out_true`, and `get<2>(p)` is the end of the output range beginning at `out_false`.

27 *Complexity:* Exactly `last - first` applications of `pred` and `proj`.

```
template<class ForwardIterator, class Predicate>
  constexpr ForwardIterator
    partition_point(ForwardIterator first, ForwardIterator last, Predicate pred);
```

28 *Requires:* `ForwardIterator`'s value type shall be convertible to `Predicate`'s argument type. The elements `e` of `[first,last)` shall be partitioned with respect to the expression `pred(e)`.

29 *Returns:* An iterator `mid` such that `all_of(first, mid, pred)` and `none_of(mid, last, pred)` are both `true`.

30 *Complexity:* $\mathcal{O}(\log(\texttt{last - first}))$ applications of `pred`.

```
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
            IndirectUnaryPredicate<projected<I, Proj>> Pred>
    I partition_point(I first, S last, Pred pred, Proj proj = Proj{});

  template <ForwardRange Rng, class Proj = identity,
            IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    safe_iterator_t<Rng> partition_point(Rng&& rng, Pred pred, Proj proj = Proj{});
}
```

31 *Requires:* `[first,last)` shall be partitioned by `pred` and `proj`, i.e. there shall be an iterator `mid` such that `all_of(first, mid, pred, proj)` and `none_of(mid, last, pred, proj)` are both true.

32 *Returns:* An iterator `mid` such that `all_of(first, mid, pred, proj)` and `none_of(mid, last, pred, proj)` are both true.

33 *Complexity:* $\mathcal{O}(\log(\texttt{last - first}))$ applications of `pred` and `proj`.

30.7.5 Merge [alg.merge]

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
  constexpr OutputIterator
    merge(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  ForwardIterator
    merge(ExecutionPolicy&& exec,
          ForwardIterator1 first1, ForwardIterator1 last1,
          ForwardIterator2 first2, ForwardIterator2 last2,
          ForwardIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  constexpr OutputIterator
    merge(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  ForwardIterator
    merge(ExecutionPolicy&& exec,
          ForwardIterator1 first1, ForwardIterator1 last1,
          ForwardIterator2 first2, ForwardIterator2 last2,
          ForwardIterator result, Compare comp);
```

1 *Requires:* The ranges `[first1,last1)` and `[first2,last2)` shall be sorted with respect to `operator<` or `comp`. The resulting range shall not overlap with either of the original ranges.

2 *Effects:* Copies all the elements of the two ranges `[first1,last1)` and `[first2,last2)` into the range `[result,result_last)`, where `result_last` is `result + (last1 - first1) + (last2 - first2)`, such that the resulting range satisfies `is_sorted(result, result_last)` or `is_sorted(result, result_last, comp)`, respectively.

3 *Returns:* `result + (last1 - first1) + (last2 - first2)`.

4 *Complexity:* Let $N =$ `(last1 - first1) + (last2 - first2)`:

(4.1) — For the overloads with no `ExecutionPolicy`, at most $N - 1$ comparisons.

(4.2) — For the overloads with an `ExecutionPolicy`, $\mathcal{O}(N)$ comparisons.

5 *Remarks:* Stable20.5.5.7.

```
namespace ranges {
  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
            WeaklyIncrementable O, class Comp = ranges::less<>, class Proj1 = identity,
            class Proj2 = identity>
      requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
    tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
      merge(I1 first1, S1 last1, I2 first2, S2 last2, O result,
            Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O, class Comp = ranges::less<>,
            class Proj1 = identity, class Proj2 = identity>
      requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
    tagged_tuple<tag::in1(safe_iterator_t<Rng1>), tag::in2(safe_iterator_t<Rng2>), tag::out(O)>
      merge(Rng1&& rng1, Rng2&& rng2, O result, Comp comp = Comp{}, Proj1 proj1 = Proj1{},
            Proj2 proj2 = Proj2{});
}
```

6 *Effects:* Copies all the elements of the two ranges `[first1,last1)` and `[first2,last2)` into the range `[result,result_last)`, where `result_last` is `result + (last1 - first1) + (last2 - first2)`. If an element `a` precedes `b` in an input range, `a` is copied into the output range before `b`. If `e1` is an element of `[first1,last1)` and `e2` of `[first2,last2)`, `e2` is copied into the output range before `e1` if and only if `bool(invoke(comp, invoke(proj2, e2), invoke(proj1, e1)))` is `true`.

7 *Requires:* The ranges `[first1,last1)` and `[first2,last2)` shall be sorted with respect to `comp`, `proj1`, and `proj2`. The resulting range shall not overlap with either of the original ranges.

8 *Returns:* `make_tagged_tuple<tag::in1, tag::in2, tag::out>(last1, last2, result_last)`.

9 *Complexity:* At most `(last1 - first1) + (last2 - first2) - 1` applications of the comparison function and each projection.

10 *Remarks:* Stable (20.5.5.7).

```
template<class BidirectionalIterator>
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last);
template<class ExecutionPolicy, class BidirectionalIterator>
  void inplace_merge(ExecutionPolicy&& exec,
                     BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last, Compare comp);
template<class ExecutionPolicy, class BidirectionalIterator, class Compare>
  void inplace_merge(ExecutionPolicy&& exec,
                     BidirectionalIterator first,
```

```
                    BidirectionalIterator middle,
                    BidirectionalIterator last, Compare comp);
```

11 *Requires:* The ranges `[first,middle)` and `[middle,last)` shall be sorted with respect to `operator<` or `comp`. `BidirectionalIterator` shall satisfy the `ValueSwappable` requirements?.3.11. The type of `*first` shall satisfy the *Cpp98MoveConstructible* (23) and *Cpp98MoveAssignable* (25) requirements.

12 *Effects:* Merges two sorted consecutive ranges `[first,middle)` and `[middle,last)`, putting the result of the merge into the range `[first,last)`. The resulting range will be in non-decreasing order; that is, for every iterator `i` in `[first,last)` other than `first`, the condition `*i < *(i - 1)` or, respectively, `comp(*i, *(i - 1))` will be `false`.

13 *Complexity:* Let $N =$ `last - first`:

- (13.1) For the overloads with no `ExecutionPolicy`, if enough additional memory is available, exactly $N - 1$ comparisons.
- (13.2) For the overloads with no `ExecutionPolicy` if no additional memory is available, $\mathcal{O}(N \log N)$ comparisons.
- (13.3) For the overloads with an `ExecutionPolicy`, $\mathcal{O}(N \log N)$ comparisons.

14 *Remarks:* Stable20.5.5.7.

```
namespace ranges {
  template <BidirectionalIterator I, Sentinel<I> S, class Comp = ranges::less<>,
            class Proj = identity>
    requires Sortable<I, Comp, Proj>
    I inplace_merge(I first, I middle, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <BidirectionalRange Rng, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    safe_iterator_t<Rng> inplace_merge(Rng&& rng, iterator_t<Rng> middle, Comp comp = Comp{},
                                       Proj proj = Proj{});
}
```

15 *Effects:* Merges two sorted consecutive ranges `[first,middle)` and `[middle,last)`, putting the result of the merge into the range `[first,last)`. The resulting range will be in non-decreasing order; that is, for every iterator `i` in `[first,last)` other than `first`, the condition `invoke(comp, invoke(proj, *i), invoke(proj, *(i - 1)))` will be false.

16 *Requires:* The ranges `[first,middle)` and `[middle,last)` shall be sorted with respect to `comp` and `proj`.

17 *Returns:* `last`

18 *Complexity:* When enough additional memory is available, `(last - first) - 1` applications of the comparison function and projection. If no additional memory is available, an algorithm with complexity $N \log(N)$ (where `N` is equal to `last - first`) may be used.

19 *Remarks:* Stable (20.5.5.7).

30.7.6 Set operations on sorted structures [alg.set.operations]

1 This subclause defines all the basic set operations on sorted structures. They also work with `multiset`s26.4.7 containing multiple copies of equivalent elements. The semantics of the set operations are generalized to `multiset`s in a standard way by defining `set_union()` to contain the maximum number of occurrences of every element, `set_intersection()` to contain the minimum, and so on.

30.7.6.1 `includes` [includes]

```
template<class InputIterator1, class InputIterator2>
  constexpr bool includes(InputIterator1 first1, InputIterator1 last1,
                          InputIterator2 first2, InputIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  bool includes(ExecutionPolicy&& exec,
                ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2, ForwardIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
  constexpr bool includes(InputIterator1 first1, InputIterator1 last1,
                          InputIterator2 first2, InputIterator2 last2,
                          Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class Compare>
  bool includes(ExecutionPolicy&& exec,
                ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2, ForwardIterator2 last2,
                Compare comp);
```

1 *Returns:* `true` if `[first2,last2)` is empty or if every element in the range `[first2,last2)` is contained in the range `[first1,last1)`. Returns `false` otherwise.

2 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

```
namespace ranges {
  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
          class Proj1 = identity, class Proj2 = identity,
      IndirectStrictWeakOrder<projected<I1, Proj1>, projected<I2, Proj2>> Comp = ranges::less<>>
    bool includes(I1 first1, S1 last1, I2 first2, S2 last2, Comp comp = Comp{},
                  Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

  template <InputRange Rng1, InputRange Rng2, class Proj1 = identity, class Proj2 = identity,
          IndirectStrictWeakOrder<projected<iterator_t<Rng1>, Proj1>,
                                  projected<iterator_t<Rng2>, Proj2>> Comp = ranges::less<>>
    bool includes(Rng1&& rng1, Rng2&& rng2, Comp comp = Comp{}, Proj1 proj1 = Proj1{},
                  Proj2 proj2 = Proj2{});
}
```

3 *Returns:* `true` if `[first2,last2)` is empty or if every element in the range `[first2,last2)` is contained in the range `[first1,last1)`. Returns `false` otherwise.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` applications of the comparison function and projections.

30.7.6.2 `set_union` [set.union]

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
  constexpr OutputIterator
    set_union(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  ForwardIterator
    set_union(ExecutionPolicy&& exec,
              ForwardIterator1 first1, ForwardIterator1 last1,
```

```
              ForwardIterator2 first2, ForwardIterator2 last2,
              ForwardIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  constexpr OutputIterator
    set_union(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  ForwardIterator
    set_union(ExecutionPolicy&& exec,
              ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator2 last2,
              ForwardIterator result, Compare comp);
```

1 *Requires:* The resulting range shall not overlap with either of the original ranges.

2 *Effects:* Constructs a sorted union of the elements from the two ranges; that is, the set of elements that are present in one or both of the ranges.

3 *Returns:* The end of the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

5 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, then all m elements from the first range shall be copied to the output range, in order, and then $\max(n - m, 0)$ elements from the second range shall be copied to the output range, in order.

```
namespace ranges {
  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
            WeaklyIncrementable O, class Comp = ranges::less<>, class Proj1 = identity,
            class Proj2 = identity>
      requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
    tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
      set_union(I1 first1, S1 last1, I2 first2, S2 last2, O result, Comp comp = Comp{},
                Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
            class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
      requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
    tagged_tuple<tag::in1(safe_iterator_t<Rng1>), tag::in2(safe_iterator_t<Rng2>), tag::out(O)>
      set_union(Rng1&& rng1, Rng2&& rng2, O result, Comp comp = Comp{}, Proj1 proj1 = Proj1{},
                Proj2 proj2 = Proj2{});
}
```

6 *Effects:* Constructs a sorted union of the elements from the two ranges; that is, the set of elements that are present in one or both of the ranges.

7 *Requires:* The resulting range shall not overlap with either of the original ranges.

8 *Returns:* `make_tagged_tuple<tag::in1, tag::in2, tag::out>(last1, last2, result +` n`)`, where n is the number of elements in the constructed range.

9 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` applications of the comparison function and projections.

10 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, then all m elements from the first range shall

be copied to the output range, in order, and then $\max(n - m, 0)$ elements from the second range shall be copied to the output range, in order.

30.7.6.3 `set_intersection` [set.intersection]

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
  constexpr OutputIterator
    set_intersection(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  ForwardIterator
    set_intersection(ExecutionPolicy&& exec,
                     ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2, ForwardIterator2 last2,
                     ForwardIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  constexpr OutputIterator
    set_intersection(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  ForwardIterator
    set_intersection(ExecutionPolicy&& exec,
                     ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2, ForwardIterator2 last2,
                     ForwardIterator result, Compare comp);
```

1 *Requires:* The resulting range shall not overlap with either of the original ranges.

2 *Effects:* Constructs a sorted intersection of the elements from the two ranges; that is, the set of elements that are present in both of the ranges.

3 *Returns:* The end of the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

5 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, the first $\min(m, n)$ elements shall be copied from the first range to the output range, in order.

```
namespace ranges {
  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
            WeaklyIncrementable O, class Comp = ranges::less<>, class Proj1 = identity,
            class Proj2 = identity>
      requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
    O set_intersection(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                       Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

  template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
            class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
      requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
    O set_intersection(Rng1&& rng1, Rng2&& rng2, O result,
```

```
                          Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

6 *Effects:* Constructs a sorted intersection of the elements from the two ranges; that is, the set of elements that are present in both of the ranges.

7 *Requires:* The resulting range shall not overlap with either of the original ranges.

8 *Returns:* The end of the constructed range.

9 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` applications of the comparison function and projections.

10 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, the first $\min(m, n)$ elements shall be copied from the first range to the output range, in order.

30.7.6.4 `set_difference` [set.difference]

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
  constexpr OutputIterator
    set_difference(InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, InputIterator2 last2,
                   OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  ForwardIterator
    set_difference(ExecutionPolicy&& exec,
                   ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2, ForwardIterator2 last2,
                   ForwardIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  constexpr OutputIterator
    set_difference(InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, InputIterator2 last2,
                   OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  ForwardIterator
    set_difference(ExecutionPolicy&& exec,
                   ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2, ForwardIterator2 last2,
                   ForwardIterator result, Compare comp);
```

1 *Requires:* The resulting range shall not overlap with either of the original ranges.

2 *Effects:* Copies the elements of the range `[first1,last1)` which are not present in the range `[first2, last2)` to the range beginning at `result`. The elements in the constructed range are sorted.

3 *Returns:* The end of the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

5 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, the last $\max(m-n, 0)$ elements from `[first1, last1)` shall be copied to the output range.

```
namespace ranges {
  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
            WeaklyIncrementable O, class Comp = ranges::less<>, class Proj1 = identity,
            class Proj2 = identity>
      requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
    tagged_pair<tag::in1(I1), tag::out(O)>
      set_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                     Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

  template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
            class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
      requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
    tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::out(O)>
      set_difference(Rng1&& rng1, Rng2&& rng2, O result,
                     Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

6 *Effects:* Copies the elements of the range `[first1,last1)` which are not present in the range `[first2, last2)` to the range beginning at `result`. The elements in the constructed range are sorted.

7 *Requires:* The resulting range shall not overlap with either of the original ranges.

8 *Returns:* `{last1, result +` n`}`, where n is the number of elements in the constructed range.

9 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` applications of the comparison function and projections.

10 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, the last $\max(m-n, 0)$ elements from `[first1, last1)` shall be copied to the output range.

30.7.6.5 `set_symmetric_difference` [set.symmetric.difference]

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
  constexpr OutputIterator
    set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  ForwardIterator
    set_symmetric_difference(ExecutionPolicy&& exec,
                             ForwardIterator1 first1, ForwardIterator1 last1,
                             ForwardIterator2 first2, ForwardIterator2 last2,
                             ForwardIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  constexpr OutputIterator
    set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  ForwardIterator
    set_symmetric_difference(ExecutionPolicy&& exec,
                             ForwardIterator1 first1, ForwardIterator1 last1,
```

```
                                   ForwardIterator2 first2, ForwardIterator2 last2,
                                   ForwardIterator result, Compare comp);
```

1 *Requires:* The resulting range shall not overlap with either of the original ranges.

2 *Effects:* Copies the elements of the range `[first1,last1)` that are not present in the range `[first2, last2)`, and the elements of the range `[first2,last2)` that are not present in the range `[first1, last1)` to the range beginning at `result`. The elements in the constructed range are sorted.

3 *Returns:* The end of the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

5 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, then $|m-n|$ of those elements shall be copied to the output range: the last $m-n$ of these elements from `[first1,last1)` if $m > n$, and the last $n-m$ of these elements from `[first2,last2)` if $m < n$.

```
namespace ranges {
  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
            WeaklyIncrementable O, class Comp = ranges::less<>, class Proj1 = identity,
            class Proj2 = identity>
      requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
    tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
      set_symmetric_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                               Comp comp = Comp{}, Proj1 proj1 = Proj1{},
                               Proj2 proj2 = Proj2{});
  template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
            class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
      requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
    tagged_tuple<tag::in1(safe_iterator_t<Rng1>), tag::in2(safe_iterator_t<Rng2>), tag::out(O)>
      set_symmetric_difference(Rng1&& rng1, Rng2&& rng2, O result, Comp comp = Comp{},
                               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

6 *Effects:* Copies the elements of the range `[first1,last1)` that are not present in the range `[first2, last2)`, and the elements of the range `[first2,last2)` that are not present in the range `[first1, last1)` to the range beginning at `result`. The elements in the constructed range are sorted.

7 *Requires:* The resulting range shall not overlap with either of the original ranges.

8 *Returns:* `make_tagged_tuple<tag::in1, tag::in2, tag::out>(last1, last2, result +` n`)`, where n is the number of elements in the constructed range.

9 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` applications of the comparison function and projections.

10 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, then $|m-n|$ of those elements shall be copied to the output range: the last $m-n$ of these elements from `[first1,last1)` if $m > n$, and the last $n-m$ of these elements from `[first2,last2)` if $m < n$.

30.7.7 Heap operations [alg.heap.operations]

1 A *heap* is a particular organization of elements in a range between two random access iterators `[a,b)` such that:

- (1.1) — With N `= b - a`, for all i, $0 < i < N$, `comp(a[`$\left\lfloor \frac{i-1}{2} \right\rfloor$`], a[`$i$`])` is `false`.
- (1.2) — `*a` may be removed by `pop_heap()`, or a new element added by `push_heap()`, in $\mathcal{O}(\log N)$ time.

2 These properties make heaps useful as priority queues.

3 `make_heap()` converts a range into a heap and `sort_heap()` turns a heap into a sorted sequence.

30.7.7.1 `push_heap` [push.heap]

```
template<class RandomAccessIterator>
  void push_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void push_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

1 *Requires:* The range `[first,last - 1)` shall be a valid heap. The type of `*first` shall satisfy the *Cpp98MoveConstructible* requirements (23) and the *Cpp98MoveAssignable* requirements (25).

2 *Effects:* Places the value in the location `last - 1` into the resulting heap `[first,last)`.

3 *Complexity:* At most log(`last - first`) comparisons.

```
namespace ranges {
  template <RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
            class Proj = identity>
      requires Sortable<I, Comp, Proj>
    I push_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

  template <RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
      requires Sortable<iterator_t<Rng>, Comp, Proj>
    safe_iterator_t<Rng> push_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

4 *Effects:* Places the value in the location `last - 1` into the resulting heap `[first,last)`.

5 *Requires:* The range `[first,last - 1)` shall be a valid heap.

6 *Returns:* `last`

7 *Complexity:* At most `log(last - first)` applications of the comparison function and projection.

30.7.7.2 `pop_heap` [pop.heap]

```
template<class RandomAccessIterator>
  void pop_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
                Compare comp);
```

1 *Requires:* The range `[first,last)` shall be a valid non-empty heap. `RandomAccessIterator` shall satisfy the `ValueSwappable` requirements?.3.11. The type of `*first` shall satisfy the *Cpp98MoveConstructible* (23) and *Cpp98MoveAssignable* (25) requirements.

2 *Effects:* Swaps the value in the location `first` with the value in the location `last - 1` and makes `[first,last - 1)` into a heap.

3 *Complexity:* At most 2 log(`last - first`) comparisons.

```
namespace ranges {
  template <RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
            class Proj = identity>
      requires Sortable<I, Comp, Proj>
    I pop_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
```

```
  template <RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
      requires Sortable<iterator_t<Rng>, Comp, Proj>
    safe_iterator_t<Rng> pop_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

4 *Requires:* The range `[first,last)` shall be a valid non-empty heap.

5 *Effects:* Swaps the value in the location `first` with the value in the location `last - 1` and makes `[first,last - 1)` into a heap.

6 *Returns:* `last`

7 *Complexity:* At most `2 * log(last - first)` applications of the comparison function and projection.

30.7.7.3 `make_heap` [make.heap]

```
template<class RandomAccessIterator>
  void make_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void make_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

1 *Requires:* The type of `*first` shall satisfy the *Cpp98MoveConstructible* requirements (23) and the *Cpp98MoveAssignable* requirements (25).

2 *Effects:* Constructs a heap out of the range `[first,last)`.

3 *Complexity:* At most 3(`last - first`) comparisons.

```
namespace ranges {
  template <RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
          class Proj = identity>
      requires Sortable<I, Comp, Proj>
    I make_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
      requires Sortable<iterator_t<Rng>, Comp, Proj>
    safe_iterator_t<Rng> make_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

4 *Effects:* Constructs a heap out of the range `[first,last)`.

5 *Returns:* `last`

6 *Complexity:* At most `3 * (last - first)` applications of the comparison function and projection.

30.7.7.4 `sort_heap` [sort.heap]

```
template<class RandomAccessIterator>
  void sort_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

1 *Requires:* The range `[first,last)` shall be a valid heap. `RandomAccessIterator` shall satisfy the `ValueSwappable` requirements?.3.11. The type of `*first` shall satisfy the *Cpp98MoveConstructible* (23) and *Cpp98MoveAssignable* (25) requirements.

2 *Effects:* Sorts elements in the heap `[first,last)`.

3 *Complexity:* At most $2N \log N$ comparisons, where $N =$ `last - first`.

```
namespace ranges {
  template <RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
            class Proj = identity>
      requires Sortable<I, Comp, Proj>
    I sort_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

  template <RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
      requires Sortable<iterator_t<Rng>, Comp, Proj>
    safe_iterator_t<Rng> sort_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

4 *Effects:* Sorts elements in the heap `[first,last)`.

5 *Requires:* The range `[first,last)` shall be a valid heap.

6 *Returns:* `last`

7 *Complexity:* At most $N \log(N)$ comparisons (where `N == last - first`), and exactly twice as many applications of the projection.

30.7.7.5 `is_heap` [is.heap]

```
template<class RandomAccessIterator>
  constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last);
```

1 *Returns:* `is_heap_until(first, last) == last`.

```
template<class ExecutionPolicy, class RandomAccessIterator>
  bool is_heap(ExecutionPolicy&& exec,
               RandomAccessIterator first, RandomAccessIterator last);
```

2 *Returns:* `is_heap_until(std::forward<ExecutionPolicy>(exec), first, last) == last`.

```
template<class RandomAccessIterator, class Compare>
  constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last,
                         Compare comp);
```

3 *Returns:* `is_heap_until(first, last, comp) == last`.

```
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  bool is_heap(ExecutionPolicy&& exec,
               RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

4 *Returns:*

```
is_heap_until(std::forward<ExecutionPolicy>(exec), first, last, comp) == last
```

```
namespace ranges {
  template <RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
            IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less<>>
    bool is_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <RandomAccessRange Rng, class Proj = identity,
            IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    bool is_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

5 *Returns:* `is_heap_until(first, last, comp, proj) == last`

```
template<class RandomAccessIterator>
  constexpr RandomAccessIterator
    is_heap_until(RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator>
  RandomAccessIterator
    is_heap_until(ExecutionPolicy&& exec,
                  RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  constexpr RandomAccessIterator
    is_heap_until(RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  RandomAccessIterator
    is_heap_until(ExecutionPolicy&& exec,
                  RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp);
```

6 *Returns:* If `(last - first) < 2`, returns `last`. Otherwise, returns the last iterator `i` in `[first, last]` for which the range `[first,i)` is a heap.

7 *Complexity:* Linear.

```
namespace ranges {
  template <RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
            IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less<>>
    I is_heap_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <RandomAccessRange Rng, class Proj = identity,
            IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    safe_iterator_t<Rng> is_heap_until(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

8 *Returns:* If `distance(first, last) < 2`, returns `last`. Otherwise, returns the last iterator `i` in `[first,last]` for which the range `[first,i)` is a heap.

9 *Complexity:* Linear.

30.7.8 Minimum and maximum [alg.min.max]

```
template<class T> constexpr const T& min(const T& a, const T& b);
template<class T, class Compare>
  constexpr const T& min(const T& a, const T& b, Compare comp);
```

1 *Requires:* For the first form, type `T` shall be `LessThanComparable` (21).

2 *Returns:* The smaller value.

3 *Remarks:* Returns the first argument when the arguments are equivalent.

4 *Complexity:* Exactly one comparison.

```
namespace ranges {
  template <class T, class Proj = identity,
            IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr const T& min(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
}
```

5 *Returns:* The smaller value.

6 *Remarks:* Returns the first argument when the arguments are equivalent.

```
template<class T>
  constexpr T min(initializer_list<T> t);
template<class T, class Compare>
  constexpr T min(initializer_list<T> t, Compare comp);
```

7 *Requires:* `T` shall be *Cpp98CopyConstructible* and `t.size() > 0`. For the first form, type `T` shall be `LessThanComparable`.

8 *Returns:* The smallest value in the initializer list.

9 *Remarks:* Returns a copy of the leftmost argument when several arguments are equivalent to the smallest.

10 *Complexity:* Exactly `t.size() - 1` comparisons.

```
namespace ranges {
  template <Copyable T, class Proj = identity,
            IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr T min(initializer_list<T> rng, Comp comp = Comp{}, Proj proj = Proj{});
  template <InputRange Rng, class Proj = identity,
            IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
     requires Copyable<iter_value_t<iterator_t<Rng>>>
    iter_value_t<iterator_t<Rng>> min(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

11 *Requires:* `distance(rng) > 0`.

12 *Returns:* The smallest value in the `initializer_list` or range.

13 *Remarks:* Returns a copy of the leftmost argument when several arguments are equivalent to the smallest.

```
template<class T> constexpr const T& max(const T& a, const T& b);
template<class T, class Compare>
  constexpr const T& max(const T& a, const T& b, Compare comp);
```

14 *Requires:* For the first form, type `T` shall be `LessThanComparable` (21).

15 *Returns:* The larger value.

16 *Remarks:* Returns the first argument when the arguments are equivalent.

17 *Complexity:* Exactly one comparison.

```
namespace ranges {
  template <class T, class Proj = identity,
            IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr const T& max(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
}
```

18 *Returns:* The larger value.

19 *Remarks:* Returns the first argument when the arguments are equivalent.

```
template<class T>
  constexpr T max(initializer_list<T> t);
template<class T, class Compare>
  constexpr T max(initializer_list<T> t, Compare comp);
```

20 *Requires:* `T` shall be *Cpp98CopyConstructible* and `t.size() > 0`. For the first form, type `T` shall be `LessThanComparable`.

21 *Returns:* The largest value in the initializer list.

22 *Remarks:* Returns a copy of the leftmost argument when several arguments are equivalent to the largest.

23 *Complexity:* Exactly `t.size() - 1` comparisons.

```
namespace ranges {
  template <Copyable T, class Proj = identity,
            IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr T max(initializer_list<T> rng, Comp comp = Comp{}, Proj proj = Proj{});
  template <InputRange Rng, class Proj = identity,
            IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
      requires Copyable<iter_value_t<iterator_t<Rng>>>
    iter_value_t<iterator_t<Rng>> max(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

24 *Requires:* `distance(rng) > 0`.

25 *Returns:* The largest value in the `initializer_list` or range.

26 *Remarks:* Returns a copy of the leftmost argument when several arguments are equivalent to the largest.

```
template<class T> constexpr pair<const T&, const T&> minmax(const T& a, const T& b);
template<class T, class Compare>
  constexpr pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
```

27 *Requires:* For the first form, type `T` shall be `LessThanComparable` (21).

28 *Returns:* `pair<const T&, const T&>(b, a)` if `b` is smaller than `a`, and `pair<const T&, const T&>(a, b)` otherwise.

29 *Remarks:* Returns `pair<const T&, const T&>(a, b)` when the arguments are equivalent.

30 *Complexity:* Exactly one comparison.

```
namespace ranges {
  template <class T, class Proj = identity,
            IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr tagged_pair<tag::min(const T&), tag::max(const T&)>
      minmax(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
}
```

31 *Returns:* `{b, a}` if `b` is smaller than `a`, and `{a, b}` otherwise.

32 *Remarks:* Returns `{a, b}` when the arguments are equivalent.

33 *Complexity:* Exactly one comparison and exactly two applications of the projection.

```
template<class T>
  constexpr pair<T, T> minmax(initializer_list<T> t);
template<class T, class Compare>
  constexpr pair<T, T> minmax(initializer_list<T> t, Compare comp);
```

34 *Requires:* `T` shall be *Cpp98CopyConstructible* and `t.size() > 0`. For the first form, type `T` shall be `LessThanComparable`.

35 *Returns:* `pair<T, T>(x, y)`, where `x` has the smallest and `y` has the largest value in the initializer list.

36 *Remarks:* `x` is a copy of the leftmost argument when several arguments are equivalent to the smallest. `y` is a copy of the rightmost argument when several arguments are equivalent to the largest.

37 *Complexity:* At most (3/2)`t.size()` applications of the corresponding predicate.

```
namespace ranges {
  template <Copyable T, class Proj = identity,
            IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr tagged_pair<tag::min(T), tag::max(T)>
      minmax(initializer_list<T> rng, Comp comp = Comp{}, Proj proj = Proj{});

  template <InputRange Rng, class Proj = identity,
            IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj> Comp = ranges::less<>>
      requires Copyable<iter_value_t<iterator_t<Rng>>>
    tagged_pair<tag::min(iter_value_t<iterator_t<Rng>>), tag::max(iter_value_t<iterator_t<Rng>>)>
      minmax(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

38 *Requires:* `distance(rng) > 0`.

39 *Returns:* `{x, y}`, where `x` has the smallest and `y` has the largest value in the `initializer_list` or range.

40 *Remarks:* `x` is a copy of the leftmost argument when several arguments are equivalent to the smallest. `y` is a copy of the rightmost argument when several arguments are equivalent to the largest.

41 *Complexity:* At most `(3/2) * distance(rng)` applications of the corresponding predicate, and at most twice as many applications of the projection.

```
template<class ForwardIterator>
  constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator min_element(ExecutionPolicy&& exec,
                              ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
  constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
                                        Compare comp);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  ForwardIterator min_element(ExecutionPolicy&& exec,
                              ForwardIterator first, ForwardIterator last,
                              Compare comp);
```

42 *Returns:* The first iterator `i` in the range `[first,last)` such that for every iterator `j` in the range `[first,last)` the following corresponding conditions hold: `!(*j < *i)` or `comp(*j, *i) == false`. Returns `last` if `first == last`.

43 *Complexity:* Exactly max(`last - first - 1`, 0) applications of the corresponding comparisons.

```
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
          IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less<>>
  I min_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
template <ForwardRange Rng, class Proj = identity,
          IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
  safe_iterator_t<Rng> min_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

44 *Returns:* The first iterator `i` in the range `[first,last)` such that for every iterator `j` in the range `[first,last)` the following corresponding condition holds: `invoke(comp, invoke(proj, *j), invoke(proj, *i)) == false`. Returns `last` if `first == last`.

45 *Complexity:* Exactly `max((last - first) - 1, 0)` applications of the comparison function and exactly twice as many applications of the projection.

```
template<class ForwardIterator>
  constexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator max_element(ExecutionPolicy&& exec,
                              ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
  constexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
                                        Compare comp);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  ForwardIterator max_element(ExecutionPolicy&& exec,
                              ForwardIterator first, ForwardIterator last,
                              Compare comp);
```

46 *Returns:* The first iterator `i` in the range `[first,last)` such that for every iterator `j` in the range `[first,last)` the following corresponding conditions hold: `!(*i < *j)` or `comp(*i, *j) == false`. Returns `last` if `first == last`.

47 *Complexity:* Exactly max(`last - first - 1`, 0) applications of the corresponding comparisons.

```
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
            IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less<>>
    I max_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <ForwardRange Rng, class Proj = identity,
            IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    safe_iterator_t<Rng> max_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

48 *Returns:* The first iterator `i` in the range `[first,last)` such that for every iterator `j` in the range `[first,last)` the following corresponding condition holds:
`invoke(comp, invoke(proj, *i), invoke(proj, *j)) == false`. Returns `last` if `first == last`.

49 *Complexity:* Exactly `max((last - first) - 1, 0)` applications of the comparison function and exactly twice as many applications of the projection.

```
template<class ForwardIterator>
  constexpr pair<ForwardIterator, ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator>
  pair<ForwardIterator, ForwardIterator>
    minmax_element(ExecutionPolicy&& exec,
                   ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
  constexpr pair<ForwardIterator, ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last, Compare comp);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  pair<ForwardIterator, ForwardIterator>
    minmax_element(ExecutionPolicy&& exec,
                   ForwardIterator first, ForwardIterator last, Compare comp);
```

50 *Returns:* `make_pair(first, first)` if `[first,last)` is empty, otherwise `make_pair(m, M)`, where `m` is the first iterator in `[first,last)` such that no iterator in the range refers to a smaller element, and where `M` is the last iterator[10] in `[first,last)` such that no iterator in the range refers to a larger element.

---

[10] This behavior intentionally differs from `max_element()`.

51 *Complexity:* At most $\max(\lfloor\frac{3}{2}(N-1)\rfloor, 0)$ applications of the corresponding predicate, where N is `last - first`.

```
namespace ranges {
  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
            IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less<>>
    tagged_pair<tag::min(I), tag::max(I)>
      minmax_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <ForwardRange Rng, class Proj = identity,
            IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    tagged_pair<tag::min(safe_iterator_t<Rng>), tag::max(safe_iterator_t<Rng>)>
      minmax_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

52 *Returns:* `{first, first}` if `[first,last)` is empty, otherwise `{m, M}`, where `m` is the first iterator in `[first,last)` such that no iterator in the range refers to a smaller element, and where `M` is the last iterator in `[first,last)` such that no iterator in the range refers to a larger element.

53 *Complexity:* At most $max(\lfloor\frac{3}{2}(N-1)\rfloor, 0)$ applications of the comparison function and at most twice as many applications of the projection, where N is `distance(first, last)`.

30.7.9 Bounded value [alg.clamp]

```
template<class T>
  constexpr const T& clamp(const T& v, const T& lo, const T& hi);
template<class T, class Compare>
  constexpr const T& clamp(const T& v, const T& lo, const T& hi, Compare comp);
```

1 *Requires:* The value of `lo` shall be no greater than `hi`. For the first form, type `T` shall be `LessThanComparable` (21).

2 *Returns:* `lo` if `v` is less than `lo`, `hi` if `hi` is less than `v`, otherwise `v`.

3 [*Note:* If NaN is avoided, `T` can be a floating-point type. — *end note*]

4 *Complexity:* At most two comparisons.

30.7.10 Lexicographical comparison [alg.lex.comparison]

```
template<class InputIterator1, class InputIterator2>
  constexpr bool
    lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  bool
    lexicographical_compare(ExecutionPolicy&& exec,
                            ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
  constexpr bool
    lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2,
                            Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Compare>
  bool
    lexicographical_compare(ExecutionPolicy&& exec,
                            ForwardIterator1 first1, ForwardIterator1 last1,
```

```
                           ForwardIterator2 first2, ForwardIterator2 last2,
                           Compare comp);
```

1 *Returns:* `true` if the sequence of elements defined by the range `[first1,last1)` is lexicographically less than the sequence of elements defined by the range `[first2,last2)` and `false` otherwise.

2 *Complexity:* At most $2\min($`last1 - first1`, `last2 - first2`$)$ applications of the corresponding comparison.

3 *Remarks:* If two sequences have the same number of elements and their corresponding elements (if any) are equivalent, then neither sequence is lexicographically less than the other. If one sequence is a prefix of the other, then the shorter sequence is lexicographically less than the longer sequence. Otherwise, the lexicographical comparison of the sequences yields the same result as the comparison of the first corresponding pair of elements that are not equivalent.

4 [*Example:* The following sample implementation satisfies these requirements:

```
for ( ; first1 != last1 && first2 != last2 ; ++first1, (void) ++first2) {
  if (*first1 < *first2) return true;
  if (*first2 < *first1) return false;
}
return first1 == last1 && first2 != last2;
```

—*end example*]

5 [*Note:* An empty sequence is lexicographically less than any non-empty sequence, but not less than any empty sequence. —*end note*]

```
namespace ranges {
  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
            class Proj1 = identity, class Proj2 = identity,
            IndirectStrictWeakOrder<projected<I1, Proj1>, projected<I2, Proj2>> Comp = ranges::less<>>
    bool lexicographical_compare(I1 first1, S1 last1, I2 first2, S2 last2,
                                 Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template <InputRange Rng1, InputRange Rng2, class Proj1 = identity, class Proj2 = identity,
            IndirectStrictWeakOrder<projected<iterator_t<Rng1>, Proj1>,
                                    projected<iterator_t<Rng2>, Proj2>> Comp = ranges::less<>>
    bool lexicographical_compare(Rng1&& rng1, Rng2&& rng2, Comp comp = Comp{},
                                 Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

6 *Returns:* `true` if the sequence of elements defined by the range `[first1,last1)` is lexicographically less than the sequence of elements defined by the range `[first2,last2)` and `false` otherwise.

7 *Complexity:* At most `2*min((last1 - first1), (last2 - first2))` applications of the corresponding comparison and projections.

8 *Remarks:* If two sequences have the same number of elements and their corresponding elements are equivalent, then neither sequence is lexicographically less than the other. If one sequence is a prefix of the other, then the shorter sequence is lexicographically less than the longer sequence. Otherwise, the lexicographical comparison of the sequences yields the same result as the comparison of the first corresponding pair of elements that are not equivalent.

```
for ( ; first1 != last1 && first2 != last2 ; ++first1, (void) ++first2) {
  if (invoke(comp, invoke(proj1, *first1), invoke(proj2, *first2))) return true;
  if (invoke(comp, invoke(proj2, *first2), invoke(proj1, *first1))) return false;
}
return first1 == last1 && first2 != last2;
```

9 *Remarks:* An empty sequence is lexicographically less than any non-empty sequence, but not less than any empty sequence.

30.7.11 Three-way comparison algorithms [alg.3way]

```
template<class T, class U> constexpr auto compare_3way(const T& a, const U& b);
```

1 *Effects:* Compares two values and produces a result of the strongest applicable comparison category type:

- (1.1) — Returns `a <=> b` if that expression is well-formed.
- (1.2) — Otherwise, if the expressions `a == b` and `a < b` are each well-formed and convertible to `bool`, returns `strong_ordering::equal` when `a == b` is `true`, otherwise returns `strong_ordering::less` when `a < b` is `true`, and otherwise returns `strong_ordering::greater`.
- (1.3) — Otherwise, if the expression `a == b` is well-formed and convertible to `bool`, returns `strong_equality::equal` when `a == b` is `true`, and otherwise returns `strong_equality::nonequal`.
- (1.4) — Otherwise, the function is defined as deleted.

```
template<class InputIterator1, class InputIterator2, class Cmp>
  constexpr auto
    lexicographical_compare_3way(InputIterator1 b1, InputIterator1 e1,
                                 InputIterator2 b2, InputIterator2 e2,
                                 Cmp comp)
      -> common_comparison_category_t<decltype(comp(*b1, *b2)), strong_ordering>;
```

2 *Requires:* `Cmp` shall be a function object type whose return type is a comparison category type.

3 *Effects:* Lexicographically compares two ranges and produces a result of the strongest applicable comparison category type. Equivalent to:

```
for ( ; b1 != e1 && b2 != e2; void(++b1), void(++b2) )
  if (auto cmp = comp(*b1,*b2); cmp != 0)
    return cmp;
return b1 != e1 ? strong_ordering::greater :
       b2 != e2 ? strong_ordering::less :
                  strong_ordering::equal;
```

```
template<class InputIterator1, class InputIterator2>
  constexpr auto
    lexicographical_compare_3way(InputIterator1 b1, InputIterator1 e1,
                                 InputIterator2 b2, InputIterator2 e2);
```

4 *Effects:* Equivalent to:

```
return lexicographical_compare_3way(b1, e1, b2, e2,
                                    [](const auto& t, const auto& u) {
                                      return compare_3way(t, u);
                                    });
```

30.7.12 Permutation generators [alg.permutation.generators]

```
template<class BidirectionalIterator>
  bool next_permutation(BidirectionalIterator first,
                        BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
  bool next_permutation(BidirectionalIterator first,
                        BidirectionalIterator last, Compare comp);
```

1 *Requires:* `BidirectionalIterator` shall satisfy the `ValueSwappable` requirements?.3.11.

2 *Effects:* Takes a sequence defined by the range `[first,last)` and transforms it into the next permutation. The next permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `operator<` or `comp`.

3 *Returns:* `true` if such a permutation exists. Otherwise, it transforms the sequence into the smallest permutation, that is, the ascendingly sorted one, and returns `false`.

4 *Complexity:* At most `(last - first) / 2` swaps.

```
namespace ranges {
  template <BidirectionalIterator I, Sentinel<I> S, class Comp = ranges::less<>,
            class Proj = identity>
      requires Sortable<I, Comp, Proj>
    bool next_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

  template <BidirectionalRange Rng, class Comp = ranges::less<>, class Proj = identity>
      requires Sortable<iterator_t<Rng>, Comp, Proj>
    bool next_permutation(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

5 *Effects:* Takes a sequence defined by the range `[first,last)` and transforms it into the next permutation. The next permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `comp` and `proj`. If such a permutation exists, it returns `true`. Otherwise, it transforms the sequence into the smallest permutation, that is, the ascendingly sorted one, and returns `false`.

6 *Complexity:* At most `(last - first)/2` swaps.

```
template<class BidirectionalIterator>
  bool prev_permutation(BidirectionalIterator first,
                        BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
  bool prev_permutation(BidirectionalIterator first,
                        BidirectionalIterator last, Compare comp);
```

7 *Requires:* `BidirectionalIterator` shall satisfy the `ValueSwappable` requirements?.3.11.

8 *Effects:* Takes a sequence defined by the range `[first,last)` and transforms it into the previous permutation. The previous permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `operator<` or `comp`.

9 *Returns:* `true` if such a permutation exists. Otherwise, it transforms the sequence into the largest permutation, that is, the descendingly sorted one, and returns `false`.

10 *Complexity:* At most `(last - first) / 2` swaps.

```
namespace ranges {
  template <BidirectionalIterator I, Sentinel<I> S, class Comp = ranges::less<>,
            class Proj = identity>
      requires Sortable<I, Comp, Proj>
    bool prev_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template <BidirectionalRange Rng, class Comp = ranges::less<>, class Proj = identity>
      requires Sortable<iterator_t<Rng>, Comp, Proj>
    bool prev_permutation(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

11 *Effects:* Takes a sequence defined by the range `[first,last)` and transforms it into the previous permutation. The previous permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `comp` and `proj`.

12 *Returns:* `true` if such a permutation exists. Otherwise, it transforms the sequence into the largest permutation, that is, the descendingly sorted one, and returns `false`.

13 *Complexity:* At most `(last - first)/2` swaps.

30.8 C library algorithms [alg.c.library]

1 [*Note:* The header `<cstdlib>` declares the functions described in this subclause. — *end note*]

```
void* bsearch(const void* key, const void* base, size_t nmemb, size_t size,
              c-compare-pred * compar);
void* bsearch(const void* key, const void* base, size_t nmemb, size_t size,
              compare-pred * compar);
void qsort(void* base, size_t nmemb, size_t size, c-compare-pred * compar);
void qsort(void* base, size_t nmemb, size_t size, compare-pred * compar);
```

2 *Effects:* These functions have the semantics specified in the C standard library.

3 *Remarks:* The behavior is undefined unless the objects in the array pointed to by `base` are of trivial type.

4 *Throws:* Any exception thrown by `compar()`20.5.5.12.

SEE ALSO: ISO C 7.22.5.

Annex A (informative) Acknowledgements [acknowledgements]

This work was made possible in part by a grant from the Standard C++ Foundation, and by the support of my employer, Facebook.

Bibliography

[1] Casey Carter. P0944: Standard library concepts, 02 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0944r0.html.

[2] Eric Niebler and Casey Carter. P0896: Merging the ranges ts, 05 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0896r1.pdf.

[3] John Spicer and J. Stephen Adamczyk. Solving the sfinae problem for expressions, 5 2008. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2634.html.

Index

Index of library names

Index of implementation-defined behavior

The entries in this section are rough descriptions; exact specifications are at the indicated page in the general text.