Document Number: P1464R1
Date: 2019-02-20
Reply to: Marshall Clow
CppAlliance
mclow.lists@gmail.com

Mandating the Standard Library: Clause 22 - Iterators library

With the adoption of P0788R3, we have a new way of specifying requirements for the library clauses of the standard. This is one of a series of papers reformulating the requirements into the new format. This effort was strongly influenced by the informational paper P1369R0.

The changes in this series of papers fall into four broad categories.

— Change "participate in overload resolution" wording into "Constraints" elements

— Change "Requires" elements into either "Mandates" or "Expects", depending (mostly) on whether or not they can be checked at compile time.

— Drive-by fixes (hopefully very few)

This paper covers Clause 22 (Iterators).

Because the iterator requirements have been recently re-worked to use concepts (as part of the adoption of Ranges), large portions of this clause required few (or no) changes.

As a drive-by fix, I have removed a bunch of empty descriptions of the form: "Effects: Constructs an object of class `Foo`."

The entire clause is reproduced here, but the changes are confined to a few sections:

— iterator.iterators 22.3.5.1

— bidirectional.iterators 22.3.5.5

— random.access.iterators 22.3.5.6

— move.iter.cons 22.5.3.3

— istream.iterator.cons 22.6.1.1

— istream.iterator.ops 22.6.1.2

— ostream.iterator.cons.des 22.6.2.1

— ostreambuf.iter.cons 22.6.4.1

Changes from R0:

— Changed a couple of the "Expects" into "Mandates".

— Changed two of "shall not be"s into "is not"s in ostreambuf.iter.cons 22.6.4.1.

— Changed several "Constraints: The expression XXXX shall be valid and convertible to bool" to "XXXX is well-formed and convertible to bool".

Help for the editors: The changes here can be viewed as latex sources with the following commands

```
git clone git@github.com:mclow/mandate.git
cd mandate
git diff master..chapter22 iterators.tex
```

22 Iterators library [iterators]

22.1 General [iterators.general]

1 This Clause describes components that C++ programs may use to perform iterations over containers (**??**), streams (**??**), stream buffers (**??**), and other ranges (**??**).

2 The following subclauses describe iterator requirements, and components for iterator primitives, predefined iterators, and stream iterators, as summarized in Table 71.

Table 71 — Iterators library summary

Subclause		Header(s)
22.3	Iterator requirements	`<iterator>`
22.4	Iterator primitives	
22.5	Iterator adaptors	
22.6	Stream iterators	
22.7	Range access	
22.8	Container and view access	

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

```
#include <concepts>

namespace std {
  template<class T> using with-reference = T&;  // exposition only
  template<class T> concept can-reference       // exposition only
    = requires { typename with-reference<T>; };
  template<class T> concept dereferenceable     // exposition only
    = requires(T& t) {
      { *t } -> can-reference;  // not required to be equality-preserving
    };

  // 22.3.2, associated types
  // 22.3.2.1, incrementable traits
  template<class> struct incrementable_traits;
  template<class T>
    using iter_difference_t = see below;

  // 22.3.2.2, readable traits
  template<class> struct readable_traits;
  template<class T>
    using iter_value_t = see below;

  // 22.3.2.3, iterator traits
  template<class I> struct iterator_traits;
  template<class T> struct iterator_traits<T*>;

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

  namespace ranges {
    // 22.3.3, customization points
    inline namespace unspecified {
      // 22.3.3.1, iter_move
      inline constexpr unspecified iter_move = unspecified;
```

```
    // 22.3.3.2, iter_swap
    inline constexpr unspecified iter_swap = unspecified;
  }
}

template<dereferenceable T>
  requires requires(T& t) {
    { ranges::iter_move(t) } -> can-reference;
  }
using iter_rvalue_reference_t
  = decltype(ranges::iter_move(declval<T&>()));

// 22.3.4, iterator concepts
// 22.3.4.2, concept Readable
template<class In>
  concept Readable = see below;

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

// 22.3.4.3, concept Writable
template<class Out, class T>
  concept Writable = see below;

// 22.3.4.4, concept WeaklyIncrementable
template<class I>
  concept WeaklyIncrementable = see below;

// 22.3.4.5, concept Incrementable
template<class I>
  concept Incrementable = see below;

// 22.3.4.6, concept Iterator
template<class I>
  concept Iterator = see below;

// 22.3.4.7, concept Sentinel
template<class S, class I>
  concept Sentinel = see below;

// 22.3.4.8, concept SizedSentinel
template<class S, class I>
  inline constexpr bool disable_sized_sentinel = false;

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

// 22.3.4.9, concept InputIterator
template<class I>
  concept InputIterator = see below;

// 22.3.4.10, concept OutputIterator
template<class I, class T>
  concept OutputIterator = see below;

// 22.3.4.11, concept ForwardIterator
template<class I>
  concept ForwardIterator = see below;

// 22.3.4.12, concept BidirectionalIterator
template<class I>
  concept BidirectionalIterator = see below;
```

```
// 22.3.4.13, concept RandomAccessIterator
template<class I>
  concept RandomAccessIterator = see below;

// 22.3.4.14, concept ContiguousIterator
template<class I>
  concept ContiguousIterator = see below;

// 22.3.6, indirect callable requirements
// 22.3.6.2, 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 F, class... Is>
  requires (Readable<Is> && ...) && Invocable<F, iter_reference_t<Is>...>
    using indirect_result_t = invoke_result_t<F, iter_reference_t<Is>...>;

// 22.3.6.3, projected
template<Readable I, IndirectRegularUnaryInvocable<I> Proj>
  struct projected;

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

// 22.3.7, common algorithm requirements
// 22.3.7.2, concept IndirectlyMovable
template<class In, class Out>
  concept IndirectlyMovable = see below;

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

// 22.3.7.3, concept IndirectlyCopyable
template<class In, class Out>
  concept IndirectlyCopyable = see below;

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

// 22.3.7.4, concept IndirectlySwappable
template<class I1, class I2 = I1>
  concept IndirectlySwappable = see below;

// 22.3.7.5, concept IndirectlyComparable
template<class I1, class I2, class R, class P1 = identity, class P2 = identity>
  concept IndirectlyComparable = see below;

// 22.3.7.6, concept Permutable
template<class I>
  concept Permutable = see below;
```

```
  // 22.3.7.7, concept 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;

  // 22.4, primitives
  // 22.4.1, 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 { };

  // 22.4.2, 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);

  // 22.4.3, range iterator operations
  namespace ranges {
    // 22.4.3.1, ranges::advance
    template<Iterator I>
      constexpr void advance(I& i, iter_difference_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_t<I> advance(I& i, iter_difference_t<I> n, S bound);

    // 22.4.3.2, ranges::distance
    template<Iterator I, Sentinel<I> S>
      constexpr iter_difference_t<I> distance(I first, S last);
    template<Range R>
      constexpr iter_difference_t<iterator_t<R>> distance(R&& r);

    // 22.4.3.3, ranges::next
    template<Iterator I>
      constexpr I next(I x);
    template<Iterator I>
      constexpr I next(I x, iter_difference_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_t<I> n, S bound);

    // 22.4.3.4, ranges::prev
    template<BidirectionalIterator I>
      constexpr I prev(I x);
    template<BidirectionalIterator I>
      constexpr I prev(I x, iter_difference_t<I> n);
```

```
  template<BidirectionalIterator I>
    constexpr I prev(I x, iter_difference_t<I> n, I bound);
}

// 22.5, predefined iterators and sentinels
// 22.5.1, reverse iterators
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);

// 22.5.2, insert iterators
template<class Container> class back_insert_iterator;
template<class Container>
  constexpr back_insert_iterator<Container> back_inserter(Container& x);

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

template<class Container> class insert_iterator;
template<class Container>
  constexpr insert_iterator<Container>
    inserter(Container& x, iterator_t<Container> i);

// 22.5.3, move iterators and sentinels
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);

template<Semiregular S> class move_sentinel;

// 22.5.4, common iterators
template<Iterator I, Sentinel<I> S>
  requires (!Same<I, S>)
    class common_iterator;

template<class I, class S>
  struct incrementable_traits<common_iterator<I, S>>;

template<InputIterator I, class S>
  struct iterator_traits<common_iterator<I, S>>;

// 22.5.5, default sentinels
struct default_sentinel_t;
inline constexpr default_sentinel_t default_sentinel{};

// 22.5.6, counted iterators
template<Iterator I> class counted_iterator;

template<class I>
  struct incrementable_traits<counted_iterator<I>>;

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

// 22.5.7, unreachable sentinels
struct unreachable_sentinel_t;
inline constexpr unreachable_sentinel_t unreachable_sentinel{};

// 22.6, 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;

  // 22.7, 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));

  // 22.8, 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;
}
```

22.3 Iterator requirements [iterator.requirements]

22.3.1 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`. For every iterator type `X`, there is a corresponding signed integer type called the *difference type* of the iterator.

2 Since iterators are an abstraction of pointers, their semantics are 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 72.

Table 72 — Relations among iterator categories

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

3 The six categories of iterators correspond to the iterator concepts `InputIterator` (22.3.4.9), `OutputIterator` (22.3.4.10), `ForwardIterator` (22.3.4.11), `BidirectionalIterator` (22.3.4.12) `RandomAccessIterator` (22.3.4.13), and `ContiguousIterator` (22.3.4.14), respectively. The generic term *iterator* refers to any type that models the `Iterator` concept (22.3.4.6).

4 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.

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

6 In addition to the requirements in this subclause, the nested *typedef-name*s specified in 22.3.2.3 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*]

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 *Cpp17DefaultConstructible* 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 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.[231]

9 An iterator and a sentinel denoting a range are comparable. 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, if any, pointed to by the first iterator `j` such that `j == s`.

10 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 valid range.

11 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, if any, pointed to by the

231) 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.

result of `n` applications of `++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.

12 The result of the application of library functions to invalid ranges is undefined.

13 All the categories of iterators require only those functions that are realizable for a given category in constant time (amortized). Therefore, requirement tables and concept definitions for the iterators do not specify complexity.

14 Destruction of a non-forward iterator may invalidate pointers and references previously obtained from that iterator.

15 An *invalid* iterator is an iterator that may be singular.[232]

16 Iterators are called *constexpr iterators* if all operations provided to meet iterator category requirements are constexpr functions, except for

(16.1) — a pseudo-destructor call (**??**), and

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

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

22.3.2 Associated types [iterator.assoc.types]

22.3.2.1 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 models the `WeaklyIncrementable` concept (22.3.4.4), the type

```
iter_difference_t<WI>
```

be defined as the incrementable type's difference type.

```
namespace std {
  template<class> struct incrementable_traits { };

  template<class T>
    requires is_object_v<T>
  struct incrementable_traits<T*> {
    using difference_type = ptrdiff_t;
  };

  template<class I>
  struct incrementable_traits<const I>
    : incrementable_traits<I> { };

  template<class T>
    requires requires { typename T::difference_type; }
  struct 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 incrementable_traits<T> {
    using difference_type = make_signed_t<decltype(declval<T>() - declval<T>())>;
  };

  template<class T>
    using iter_difference_t = see below;
}
```

2 The type `iter_difference_t<I>` denotes

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

(2.1) — `incrementable_traits<I>::difference_type` if `iterator_traits<I>` names a specialization generated from the primary template, and

(2.2) — `iterator_traits<I>::difference_type` otherwise.

3 Users may specialize `incrementable_traits` on program-defined types.

22.3.2.2 Readable traits [readable.traits]

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 models the `Readable` concept (22.3.4.2), the type

```
iter_value_t<R>
```

be defined as the readable type's value type.

```
template<class> struct cond-value-type { };   // exposition only
template<class T>
  requires is_object_v<T>
struct cond-value-type {
  using value_type = remove_cv_t<T>;
};

template<class> struct readable_traits { };

template<class T>
struct readable_traits<T*>
  : cond-value-type<T> { };

template<class I>
  requires is_array_v<I>
struct readable_traits<I> {
  using value_type = remove_cv_t<remove_extent_t<I>>;
};

template<class I>
struct readable_traits<const I>
  : readable_traits<I> { };

template<class T>
  requires requires { typename T::value_type; }
struct readable_traits<T>
  : cond-value-type<typename T::value_type> { };

template<class T>
  requires requires { typename T::element_type; }
struct readable_traits<T>
  : cond-value-type<typename T::element_type> { };

template<class T> using iter_value_t = see below;
```

2 The type `iter_value_t<I>` denotes

(2.1) — `readable_traits<I>::value_type` if `iterator_traits<I>` names a specialization generated from the primary template, and

(2.2) — `iterator_traits<I>::value_type` otherwise.

3 Class template `readable_traits` may be specialized on program-defined types.

4 [*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*]

5 [*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*]

22.3.2.3 Iterator traits [iterator.traits]

1 To implement algorithms only in terms of iterators, it is sometimes necessary to determine the iterator category that corresponds to a particular iterator type. Accordingly, it is required that if `I` is the type of an iterator, the type

```
iterator_traits<I>::iterator_category
```

be defined as the iterator's iterator category. In addition, the types

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

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

```
iterator_traits<I>::value_type
iterator_traits<I>::difference_type
iterator_traits<I>::reference
```

may be defined as `void`.

2 The definitions in this subclause make use of the following exposition-only concepts:

```
template<class I>
concept cpp17-iterator =
  Copyable<I> && requires(I i) {
    {   *i } -> can-reference;
    {  ++i } -> Same<I&>;
    { *i++ } -> can-reference;
  };

template<class I>
concept cpp17-input-iterator =
  cpp17-iterator<I> && EqualityComparable<I> && requires(I i) {
    typename incrementable_traits<I>::difference_type;
    typename readable_traits<I>::value_type;
    typename common_reference_t<iter_reference_t<I> &&,
                                typename readable_traits<I>::value_type &>;
    *i++;
    typename common_reference_t<decltype(*i++) &&,
                                typename readable_traits<I>::value_type &>;
    requires SignedIntegral<typename incrementable_traits<I>::difference_type>;
  };

template<class I>
concept cpp17-forward-iterator =
  cpp17-input-iterator<I> && Constructible<I> &&
  is_lvalue_reference_v<iter_reference_t<I>> &&
  Same<remove_cvref_t<iter_reference_t<I>>, typename readable_traits<I>::value_type> &&
  requires(I i) {
    { i++ } -> const I&;
    requires Same<iter_reference_t<I>, decltype(*i++)>;
  };

template<class I>
concept cpp17-bidirectional-iterator =
  cpp17-forward-iterator<I> && requires(I i) {
    {  --i } -> Same<I&>;
    {  i-- } -> const I&;
    { *i-- } -> Same<iter_reference_t<I>>;
  };

template<class I>
concept cpp17-random-access-iterator =
  cpp17-bidirectional-iterator<I> && StrictTotallyOrdered<I> &&
```

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

[3] The members of a specialization `iterator_traits<I>` generated from the `iterator_traits` primary template are computed as follows:

(3.1) — If `I` has valid (**??**) member types `difference_type`, `value_type`, `reference`, and `iterator_category`, then `iterator_traits<I>` has the following publicly accessible members:

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

If the *qualified-id* `I::pointer` is valid and denotes a type, then `iterator_traits<I>::pointer` names that type; otherwise, it names `void`.

(3.2) — Otherwise, if `I` satisfies the exposition-only concept *cpp17-input-iterator*, `iterator_traits<I>` has the following publicly accessible members:

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

(3.2.1) — If the *qualified-id* `I::pointer` is valid and denotes a type, `pointer` names that type. Otherwise, if `decltype(declval<I&>().operator->())` is well-formed, then `pointer` names that type. Otherwise, `pointer` names `void`.

(3.2.2) — If the *qualified-id* `I::reference` is valid and denotes a type, `reference` names that type. Otherwise, `reference` names `iter_reference_t<I>`.

(3.2.3) — If the *qualified-id* `I::iterator_category` is valid and denotes a type, `iterator_category` names that type. Otherwise, `iterator_category` names:

(3.2.3.1) — `random_access_iterator_tag` if `I` satisfies *cpp17-random-access-iterator*, or otherwise

(3.2.3.2) — `bidirectional_iterator_tag` if `I` satisfies *cpp17-bidirectional-iterator*, or otherwise

(3.2.3.3) — `forward_iterator_tag` if `I` satisfies *cpp17-forward-iterator*, or otherwise

(3.2.3.4) — `input_iterator_tag`.

(3.3) — Otherwise, if `I` satisfies the exposition-only concept *cpp17-iterator*, then `iterator_traits<I>` has the following publicly accessible members:

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

If the *qualified-id* `incrementable_traits<I>::difference_type` is valid and denotes a type, then `difference_type` names that type; otherwise, it names `void`.

(3.4) — Otherwise, `iterator_traits<I>` has no members by any of the above names.

[4] Explicit or partial specializations of `iterator_traits` may have a member type `iterator_concept` that is used to indicate conformance to the iterator concepts (22.3.4).

[5] `iterator_traits` is specialized for pointers as

```
namespace std {
  template<class T>
```

```
    requires is_object_v<T>
  struct iterator_traits<T*> {
    using iterator_concept  = contiguous_iterator_tag;
    using iterator_category = random_access_iterator_tag;
    using value_type        = remove_cv_t<T>;
    using difference_type   = ptrdiff_t;
    using pointer           = T*;
    using reference         = T&;
  };
}
```

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

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

— *end example*]

22.3.3 Customization points [iterator.cust]

22.3.3.1 `iter_move` [iterator.cust.move]

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

(1.1) — `iter_move(E)`, if that expression is valid, with overload resolution performed in a context that does not include a declaration of `ranges::iter_move`.

(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, `*E`.

(1.3) — Otherwise, `ranges::iter_move(E)` is ill-formed. [*Note*: This case can result in substitution failure when `ranges::iter_move(E)` appears in the immediate context of a template instantiation. — *end note*]

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

22.3.3.2 `iter_swap` [iterator.cust.swap]

1 The name `iter_swap` denotes a customization point object (**??**) that exchanges the values (**??**) denoted by its arguments.

2 Let *`iter-exchange-move`* be the exposition-only function:

```
template<class X, class Y>
  constexpr iter_value_t<remove_reference_t<X>> iter-exchange-move(X&& x, Y&& y)
    noexcept(noexcept(iter_value_t<remove_reference_t<X>>(iter_move(x))) &&
      noexcept(*x = iter_move(y)));
```

3 *Effects:* Equivalent to:

```
iter_value_t<remove_reference_t<X>> old_value(iter_move(x));
*x = iter_move(y);
return old_value;
```

4 The expression `ranges::iter_swap(E1, E2)` for some subexpressions `E1` and `E2` is expression-equivalent to the following:

(4.1) — `(void)iter_swap(E1, E2)`, if that expression is valid, with overload resolution performed in a context that includes the declaration

```
template<class I1, class I2>
  void iter_swap(I1, I2) = delete;
```

and does not include a declaration of `ranges::iter_swap`. If the function selected by overload resolution does not exchange the values denoted by `E1` and `E2`, the program is ill-formed with no diagnostic required.

(4.2) — Otherwise, if the types of `E1` and `E2` each model `Readable`, and if the reference types of `E1` and `E2` model `SwappableWith` (??), then `ranges::swap(*E1, *E2)`.

(4.3) — Otherwise, if the types `T1` and `T2` of `E1` and `E2` model `IndirectlyMovableStorable<T1, T2>` and `IndirectlyMovableStorable<T2, T1>`, then `(void)(*E1 =` *`iter-exchange-move`*`(E2, E1))`, except that `E1` is evaluated only once.

(4.4) — Otherwise, `ranges::iter_swap(E1, E2)` is ill-formed. [*Note*: This case can result in substitution failure when `ranges::iter_swap(E1, E2)` appears in the immediate context of a template instantiation. — *end note*]

22.3.4 Iterator concepts [iterator.concepts]

22.3.4.1 General [iterator.concepts.general]

1 For a type `I`, let *`ITER_TRAITS`*`(I)` denote the type `I` if `iterator_traits<I>` names a specialization generated from the primary template. Otherwise, *`ITER_TRAITS`*`(I)` denotes `iterator_traits<I>`.

(1.1) — If the *qualified-id* *`ITER_TRAITS`*`(I)::iterator_concept` is valid and names a type, then *`ITER_CONCEPT`*`(I)` denotes that type.

(1.2) — Otherwise, if the *qualified-id* *`ITER_TRAITS`*`(I)::iterator_category` is valid and names a type, then *`ITER_CONCEPT`*`(I)` denotes that type.

(1.3) — Otherwise, if `iterator_traits<I>` names a specialization generated from the primary template, then *`ITER_CONCEPT`*`(I)` denotes `random_access_iterator_tag`.

(1.4) — Otherwise, *`ITER_CONCEPT`*`(I)` does not denote a type.

2 [*Note*: *`ITER_TRAITS`* enables independent syntactic determination of an iterator's category and concept. — *end note*] [*Example*:

```
struct I {
  using value_type = int;
  using difference_type = int;

  int operator*() const;
  I& operator++();
  I operator++(int);
  I& operator--();
  I operator--(int);

  bool operator==(I) const;
  bool operator!=(I) const;
};
```

`iterator_traits<I>::iterator_category` denotes `input_iterator_tag`, and *`ITER_CONCEPT`*`(I)` denotes `random_access_iterator_tag`. — *end example*]

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

1 Types that are readable by applying `operator*` model the `Readable` concept, including pointers, smart pointers, and iterators.

```
template<class In>
  concept Readable =
    requires {
      typename iter_value_t<In>;
      typename iter_reference_t<In>;
      typename iter_rvalue_reference_t<In>;
    } &&
```

```
    CommonReference<iter_reference_t<In>&&, iter_value_t<In>&> &&
    CommonReference<iter_reference_t<In>&&, iter_rvalue_reference_t<In>&&> &&
    CommonReference<iter_rvalue_reference_t<In>&&, const iter_value_t<In>&>;
```

2 Given a value `i` of type `I`, `I` models `Readable` only if the expression `*i` (which is indirectly required to be valid via the exposition-only *dereferenceable* concept (22.2)) is equality-preserving.

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

1 The `Writable` concept specifies the requirements for writing a value into an iterator's referenced object.

```
template<class Out, class T>
  concept Writable =
    requires(Out&& o, T&& t) {
      *o = std::forward<T>(t); // not required to be equality-preserving
      *std::forward<Out>(o) = std::forward<T>(t); // not required to be equality-preserving
      const_cast<const iter_reference_t<Out>&&>(*o) =
        std::forward<T>(t); // not required to be equality-preserving
      const_cast<const iter_reference_t<Out>&&>(*std::forward<Out>(o)) =
        std::forward<T>(t); // not required to be equality-preserving
    };
```

2 Let `E` be an an expression such that `decltype((E))` is `T`, and let `o` be a dereferenceable object of type `Out`. `Out` and `T` model `Writable<Out, T>` only if

(2.1) — If `Out` and `T` model `Readable<Out> && Same<iter_value_t<Out>, decay_t<T>>`, then `*o` after any above assignment is equal to the value of `E` before the assignment.

3 After evaluating any above assignment expression, `o` is not required to be dereferenceable.

4 If `E` is an xvalue (??), the resulting state of the object it denotes is valid but unspecified (??).

5 [*Note*: The only valid use of an `operator*` is on the left side of the assignment statement. Assignment through the same value of the writable type happens only once. — *end note*]

6 [*Note*: `Writable` has the awkward `const_cast` expressions to reject iterators with prvalue non-proxy reference types that permit rvalue assignment but do not also permit `const` rvalue assignment. Consequently, an iterator type `I` that returns `std::string` by value does not model `Writable<I, std::string>`. — *end note*]

22.3.4.4 Concept `WeaklyIncrementable` [iterator.concept.winc]

1 The `WeaklyIncrementable` concept specifies the requirements on types that can be incremented with the pre- and post-increment operators. The increment operations are not required to be equality-preserving, nor is the type required to be `EqualityComparable`.

```
template<class I>
  concept WeaklyIncrementable =
    Semiregular<I> &&
    requires(I i) {
      typename iter_difference_t<I>;
      requires SignedIntegral<iter_difference_t<I>>;
      { ++i } -> Same<I&>; // not required to be equality-preserving
      i++; // not required to be equality-preserving
    };
```

2 Let `i` be an object of type `I`. When `i` is in the domain of both pre- and post-increment, `i` is said to be *incrementable*. `I` models `WeaklyIncrementable<I>` only if

(2.1) — The expressions `++i` and `i++` have the same domain.

(2.2) — If `i` is incrementable, then both `++i` and `i++` advance `i` to the next element.

(2.3) — If `i` is incrementable, then `addressof(++i)` is equal to `addressof(i)`.

3 [*Note*: For `WeaklyIncrementable` types, `a` equals `b` does not imply that `++a` equals `++b`. (Equality does not guarantee the substitution property or referential transparency.) Algorithms on weakly incrementable types should never attempt to pass through the same incrementable value twice. They should be single-pass algorithms. These algorithms can be used with istreams as the source of the input data through the `istream_iterator` class template. — *end note*]

22.3.4.5 Concept `Incrementable` [iterator.concept.inc]

1 The `Incrementable` concept specifies requirements on types that can be incremented with the pre- and post-increment operators. The increment operations are required to be equality-preserving, and the type is required to be `EqualityComparable`. [*Note*: This supersedes the annotations on the increment expressions in the definition of `WeaklyIncrementable`. — *end note*]

```
template<class I>
  concept Incrementable =
    Regular<I> &&
    WeaklyIncrementable<I> &&
    requires(I i) {
      { i++ } -> Same<I>;
    };
```

2 Let `a` and `b` be incrementable objects of type `I`. `I` models `Incrementable` only if

(2.1) — If `bool(a == b)` then `bool(a++ == b)`.

(2.2) — If `bool(a == b)` then `bool(((void)a++, a) == ++b)`.

3 [*Note*: The requirement that `a` equals `b` implies `++a` equals `++b` (which is not true for weakly incrementable types) allows the use of multi-pass one-directional algorithms with types that model `Incrementable`. — *end note*]

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

1 The `Iterator` concept forms the basis of the iterator concept taxonomy; every iterator models `Iterator`. This concept specifies operations for dereferencing and incrementing an iterator. Most algorithms will require additional operations to compare iterators with sentinels (22.3.4.7), to read (22.3.4.9) or write (22.3.4.10) values, or to provide a richer set of iterator movements (22.3.4.11, 22.3.4.12, 22.3.4.13).)

```
template<class I>
  concept Iterator =
    requires(I i) {
      { *i } -> can-reference;
    } &&
    WeaklyIncrementable<I>;
```

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

1 The `Sentinel` concept specifies the relationship between an `Iterator` type and a `Semiregular` type whose values denote a range.

```
template<class S, class I>
  concept Sentinel =
    Semiregular<S> &&
    Iterator<I> &&
    weakly-equality-comparable-with<S, I>; // See ??
```

2 Let `s` and `i` be values of type `S` and `I` such that `[i, s)` denotes a range. Types `S` and `I` model `Sentinel<S, I>` only if

(2.1) — `i == s` is well-defined.

(2.2) — If `bool(i != s)` then `i` is dereferenceable and `[++i, s)` denotes a range.

3 The domain of `==` is not static. Given an iterator `i` and sentinel `s` such that `[i, s)` denotes a range and `i != s`, `i` and `s` are not required to continue to denote a range after incrementing any other iterator equal to `i`. Consequently, `i == s` is no longer required to be well-defined.

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

1 The `SizedSentinel` concept specifies requirements on an `Iterator` and a `Sentinel` that allow the use of the `-` operator to compute the distance between them in constant time.

```
template<class S, class I>
  concept SizedSentinel =
    Sentinel<S, I> &&
    !disable_sized_sentinel<remove_cv_t<S>, remove_cv_t<I>> &&
    requires(const I& i, const S& s) {
```

```
    { s - i } -> Same<iter_difference_t<I>>;
    { i - s } -> Same<iter_difference_t<I>>;
  };
```

2 Let `i` be an iterator of type `I`, and `s` a sentinel of type `S` such that `[i, s)` denotes a range. Let N be the smallest number of applications of `++i` necessary to make `bool(i == s)` be `true`. `S` and `I` model `SizedSentinel<S, I>` only if

(2.1) — If N is representable by `iter_difference_t<I>`, then `s - i` is well-defined and equals N.

(2.2) — If $-N$ is representable by `iter_difference_t<I>`, then `i - s` is well-defined and equals $-N$.

3 [*Note*: `disable_sized_sentinel` allows use of sentinels and iterators with the library that satisfy but do not in fact model `SizedSentinel`. — *end note*]

4 [*Example*: The `SizedSentinel` concept is modeled by pairs of `RandomAccessIterators` (22.3.4.13) and by counted iterators and their sentinels (22.5.6.1). — *end example*]

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

1 The `InputIterator` concept defines requirements for a type whose referenced values can be read (from the requirement for `Readable` (22.3.4.2)) and which can be both pre- and post-incremented. [*Note*: Unlike the *Cpp17InputIterator* requirements (22.3.5.2), the `InputIterator` concept does not need equality comparison since iterators are typically compared to sentinels. — *end note*]

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

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

1 The `OutputIterator` concept defines requirements for a type that can be used to write values (from the requirement for `Writable` (22.3.4.3)) and which can be both pre- and post-incremented. [*Note*: Output iterators are not required to model `EqualityComparable`. — *end note*]

```
template<class I, class T>
  concept OutputIterator =
    Iterator<I> &&
    Writable<I, T> &&
    requires(I i, T&& t) {
      *i++ = std::forward<T>(t); // not required to be equality-preserving
    };
```

2 Let `E` be an expression such that `decltype((E))` is `T`, and let `i` be a dereferenceable object of type `I`. `I` and `T` model `OutputIterator<I, T>` only if `*i++ = E;` has effects equivalent to:

```
*i = E;
++i;
```

3 [*Note*: Algorithms on output iterators should never attempt to pass through the same iterator twice. They should be single-pass algorithms. — *end note*]

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

1 The `ForwardIterator` concept adds equality comparison and the multi-pass guarantee, specified below.

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

2 The domain of `==` for forward iterators is that of iterators over the same underlying sequence. However, value-initialized iterators of the same type 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 Pointers and references obtained from a forward iterator into a range `[i, s)` shall remain valid while `[i, s)` continues to denote a range.

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

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

(4.2) — The expression `((void)[](X x){++x;}(a), *a)` is equivalent to the expression `*a`.

5 [*Note*: The requirement that `a == b` implies `++a == ++b` and the removal of the restrictions on the number of assignments through a mutable iterator (which applies to output iterators) allow the use of multi-pass one-directional algorithms with forward iterators. — *end note*]

22.3.4.12 Concept `BidirectionalIterator` [iterator.concept.bidir]

1 The `BidirectionalIterator` concept adds the ability to move an iterator backward as well as forward.

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

2 A bidirectional iterator `r` is decrementable if and only if there exists some `q` such that `++q == r`. Decrementable iterators `r` shall be in the domain of the expressions `--r` and `r--`.

3 Let `a` and `b` be equal objects of type `I`. `I` models `BidirectionalIterator` only if:

(3.1) — If `a` and `b` are decrementable, then all of the following are `true`:

(3.1.1) — `addressof(--a) == addressof(a)`

(3.1.2) — `bool(a-- == b)`

(3.1.3) — after evaluating both `a--` and `--b` `bool(a == b)` is still `true`

(3.1.4) — `bool(++(--a) == b)`

(3.2) — If `a` and `b` are incrementable, then `bool(--(++a) == b)`.

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

1 The `RandomAccessIterator` concept adds support for constant-time advancement with `+=`, `+`, `-=`, and `-`, as well as the computation of distance in constant time with `-`. Random access iterators also support array notation via subscripting.

```
template<class I>
  concept RandomAccessIterator =
    BidirectionalIterator<I> &&
    DerivedFrom<ITER_CONCEPT(I), random_access_iterator_tag> &&
    StrictTotallyOrdered<I> &&
    SizedSentinel<I, I> &&
    requires(I i, const I j, const iter_difference_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]  } -> Same<iter_reference_t<I>>;
    };
```

2 Let `a` and `b` be valid iterators of type `I` such that `b` is reachable from `a` after `n` applications of `++a`, let `D` be `iter_difference_t<I>`, and let `n` denote a value of type `D`. `I` models `RandomAccessIterator` only if

(2.1) — `(a += n)` is equal to `b`.

(2.2) — `addressof(a += n)` is equal to `addressof(a)`.

(2.3) — `(a + n)` is equal to `(a += n)`.

(2.4) — For any two positive values `x` and `y` of type `D`, if `(a + D(x + y))` is valid, then `(a + D(x + y))` is equal to `((a + x) + y)`.

(2.5) — `(a + D(0))` is equal to `a`.

(2.6) — If `(a + D(n - 1))` is valid, then `(a + n)` is equal to `++(a + D(n - 1))`.

(2.7) — `(b += -n)` is equal to `a`.

(2.8) — `(b -= n)` is equal to `a`.

(2.9) — `addressof(b -= n)` is equal to `addressof(b)`.

(2.10) — `(b - n)` is equal to `(b -= n)`.

(2.11) — If `b` is dereferenceable, then `a[n]` is valid and is equal to `*b`.

(2.12) — `bool(a <= b)` is `true`.

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

1 The `ContiguousIterator` concept provides a guarantee that the denoted elements are stored contiguously in memory.

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

2 Let `a` and `b` be dereferenceable iterators of type `I` such that `b` is reachable from `a`, and let `D` be `iter_difference_t<I>`. The type `I` models `ContiguousIterator` only if `addressof(*(a + D(b - a)))` is equal to `addressof(*a) + D(b - a)`.

22.3.5 C++17 iterator requirements [iterator.cpp17]

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>` (22.3.2.3). — *end note*]

22.3.5.1 *Cpp17Iterator* [iterator.iterators]

1 The *Cpp17Iterator* requirements form the basis of the iterator taxonomy; every iterator satisfies the *Cpp17Iterator* requirements. This set of requirements specifies operations for dereferencing and incrementing an iterator. Most algorithms will require additional operations to read (22.3.5.2) or write (22.3.5.3) values, or to provide a richer set of iterator movements (22.3.5.4, 22.3.5.5, 22.3.5.6).

2 A type `X` satisfies the *Cpp17Iterator* requirements if:

(2.1) — `X` satisfies the *Cpp17CopyConstructible*, *Cpp17CopyAssignable*, and *Cpp17Destructible* requirements (**??**) and lvalues of type `X` are swappable (**??**), and

(2.2) — the expressions in Table 73 are valid and have the indicated semantics.

Table 73 — *Cpp17Iterator* requirements

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

22.3.5.2 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 *Cpp17Iterator* (22.3.5.1) and *Cpp17EqualityComparable* (Table **??**) requirements and the expressions in Table 74 are valid and have the indicated semantics.

2 In Table 74, 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 74 — *Cpp17InputIterator* requirements (in addition to *Cpp17Iterator*)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`a != b`	contextually convertible to `bool`	`!(a == b)`	~~*Requires:*~~ *Expects:* `(a, b)` is in the domain of `==`.
`*a`	`reference`, convertible to `T`		~~*Requires:*~~ *Expects:* `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:*~~ *Expects:* `a` is dereferenceable.
`++r`	`X&`		~~*Requires:*~~ *Expects:* `r` is dereferenceable. *Ensures:* `r` is dereferenceable or `r` is past-the-end; any copies of the previous value of `r` are no longer required to be dereferenceable nor 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 *Cpp17CopyAssignable* type (Table **??**). These algorithms can be used with istreams as the source of the input data through the `istream_iterator` class template. — *end note*]

22.3.5.3 Output iterators [output.iterators]

[1] A class or pointer type `X` satisfies the requirements of an output iterator if `X` satisfies the *Cpp17Iterator* requirements (22.3.5.1) and the expressions in Table 75 are valid and have the indicated semantics.

Table 75 — *Cpp17OutputIterator* requirements (in addition to *Cpp17Iterator*)

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. *Ensures:* `r` is incrementable.

Table 75 — *Cpp17OutputIterator* requirements (in addition to *Cpp17Iterator*) (continued)

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

22.3.5.4 Forward iterators [forward.iterators]

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

(1.1) — `X` satisfies the *Cpp17InputIterator* requirements (22.3.5.2),

(1.2) — `X` satisfies the *Cpp17DefaultConstructible* requirements (**??**),

(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 76 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 76 — *Cpp17ForwardIterator* requirements (in addition to *Cpp17InputIterator*)

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.

22.3.5.5 Bidirectional iterators [bidirectional.iterators]

1 A class or pointer type `X` satisfies the requirements of a bidirectional iterator if, in addition to satisfying the *Cpp17ForwardIterator* requirements, the following expressions are valid as shown in Table 77.

Table 77 — *Cpp17BidirectionalIterator* requirements (in addition to *Cpp17ForwardIterator*)

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

22.3.5.6 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 *Cpp17BidirectionalIterator* requirements, the following expressions are valid as shown in Table 78.

Table 78 — *Cpp17RandomAccessIterator* requirements (in addition to *Cpp17BidirectionalIterator*)

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:*~~ *Expects:* 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:*~~ *Expects:* there exists a value `n` of type `difference_type` such that `a + n == b`. `b == a + (b - a)`.

Table 78 — *Cpp17RandomAccessIterator* requirements (in addition to *Cpp17BidirectionalIterator*) (continued)

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

22.3.6 Indirect callable requirements [**indirectcallable**]

22.3.6.1 General [**indirectcallable.general**]

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

22.3.6.2 Indirect callables [**indirectcallable.indirectinvocable**]

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

```
namespace std {
  template<class F, class I>
    concept IndirectUnaryInvocable =
      Readable<I> &&
      CopyConstructible<F> &&
      Invocable<F&, iter_value_t<I>&> &&
      Invocable<F&, iter_reference_t<I>> &&
      Invocable<F&, iter_common_reference_t<I>> &&
      CommonReference<
        invoke_result_t<F&, iter_value_t<I>&>,
        invoke_result_t<F&, iter_reference_t<I>>>;

  template<class F, class I>
    concept IndirectRegularUnaryInvocable =
      Readable<I> &&
      CopyConstructible<F> &&
      RegularInvocable<F&, iter_value_t<I>&> &&
      RegularInvocable<F&, iter_reference_t<I>> &&
      RegularInvocable<F&, iter_common_reference_t<I>> &&
      CommonReference<
        invoke_result_t<F&, iter_value_t<I>&>,
        invoke_result_t<F&, iter_reference_t<I>>>;

  template<class F, class I>
    concept IndirectUnaryPredicate =
      Readable<I> &&
      CopyConstructible<F> &&
      Predicate<F&, iter_value_t<I>&> &&
      Predicate<F&, iter_reference_t<I>> &&
      Predicate<F&, iter_common_reference_t<I>>;
```

```
  template<class F, class I1, class I2 = I1>
    concept IndirectRelation =
      Readable<I1> && Readable<I2> &&
      CopyConstructible<F> &&
      Relation<F&, iter_value_t<I1>&, iter_value_t<I2>&> &&
      Relation<F&, iter_value_t<I1>&, iter_reference_t<I2>> &&
      Relation<F&, iter_reference_t<I1>, iter_value_t<I2>&> &&
      Relation<F&, iter_reference_t<I1>, iter_reference_t<I2>> &&
      Relation<F&, iter_common_reference_t<I1>, iter_common_reference_t<I2>>;

  template<class F, class I1, class I2 = I1>
    concept IndirectStrictWeakOrder =
      Readable<I1> && Readable<I2> &&
      CopyConstructible<F> &&
      StrictWeakOrder<F&, iter_value_t<I1>&, iter_value_t<I2>&> &&
      StrictWeakOrder<F&, iter_value_t<I1>&, iter_reference_t<I2>> &&
      StrictWeakOrder<F&, iter_reference_t<I1>, iter_value_t<I2>&> &&
      StrictWeakOrder<F&, iter_reference_t<I1>, iter_reference_t<I2>> &&
      StrictWeakOrder<F&, iter_common_reference_t<I1>, iter_common_reference_t<I2>>;
}
```

22.3.6.3 Class template `projected` [projected]

1 Class template `projected` is used to constrain algorithms that accept callable objects and projections (**??**). It combines a `Readable` type `I` and a callable object type `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<indirect_result_t<Proj&, I>>;
    indirect_result_t<Proj&, I> operator*() const; // not defined
  };

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

22.3.7 Common algorithm requirements [alg.req]

22.3.7.1 General [alg.req.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::less<>` function object type used in the concepts below imposes constraints on the concepts' arguments in addition to those that appear in the concepts' bodies (**??**). — *end note*]

22.3.7.2 Concept `IndirectlyMovable` [alg.req.ind.move]

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, iter_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, iter_value_t<In>> &&
    Movable<iter_value_t<In>> &&
    Constructible<iter_value_t<In>, iter_rvalue_reference_t<In>> &&
    Assignable<iter_value_t<In>&, iter_rvalue_reference_t<In>>;
```

[3] Let `i` be a dereferenceable value of type `In`. `In` and `Out` model `IndirectlyMovableStorable<In, Out>` only if after the initialization of the object `obj` in

```
iter_value_t<In> obj(ranges::iter_move(i));
```

`obj` is equal to the value previously denoted by `*i`. If `iter_rvalue_reference_t<In>` is an rvalue reference type, the resulting state of the value denoted by `*i` is valid but unspecified (**??**).

22.3.7.3 Concept `IndirectlyCopyable` [alg.req.ind.copy]

[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, iter_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 iter_value_t<In>&> &&
    Copyable<iter_value_t<In>> &&
    Constructible<iter_value_t<In>, iter_reference_t<In>> &&
    Assignable<iter_value_t<In>&, iter_reference_t<In>>;
```

[3] Let `i` be a dereferenceable value of type `In`. `In` and `Out` model `IndirectlyCopyableStorable<In, Out>` only if after the initialization of the object `obj` in

```
iter_value_t<In> obj(*i);
```

`obj` is equal to the value previously denoted by `*i`. If `iter_reference_t<In>` is an rvalue reference type, the resulting state of the value denoted by `*i` is valid but unspecified (**??**).

22.3.7.4 Concept `IndirectlySwappable` [alg.req.ind.swap]

[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(i1, i1);
      ranges::iter_swap(i2, i2);
      ranges::iter_swap(i1, i2);
      ranges::iter_swap(i2, i1);
    };
```

22.3.7.5 Concept `IndirectlyComparable` [alg.req.ind.cmp]

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

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

22.3.7.6 Concept `Permutable` [alg.req.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>;
```

22.3.7.7 Concept `Mergeable` [alg.req.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>>;
```

22.3.7.8 Concept `Sortable` [alg.req.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>>;
```

22.4 Iterator primitives [iterator.primitives]

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

22.4.1 Standard iterator tags [std.iterator.tags]

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: `output_iterator_tag`, `input_iterator_tag`, `forward_iterator_tag`, `bidirectional_iterator_tag`, `random_access_iterator_tag`, and `contiguous_iterator_tag`. For every iterator of type `I`, `iterator_traits<I>::iterator_category` shall be defined to be a category tag that describes the iterator's behavior. Additionally, `iterator_traits<I>::iterator_concept` may be used to indicate conformance to the iterator concepts (22.3.4).

```
namespace std {
  struct output_iterator_tag { };
  struct input_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*]

22.4.2 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 *Expects:* `n` is negative only for bidirectional iterators.

3 *Effects:* Increments `i` by `n` if `n` is non-negative, and decrements `i` by `-n` otherwise.

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

4 *Expects:* `last` is reachable from `first`, or `InputIterator` meets the *Cpp17RandomAccessIterator* requirements and `first` is reachable from `last`.

5 *Effects:* If `InputIterator` meets the *Cpp17RandomAccessIterator* requirements, returns `(last - first)`; otherwise, returns the number of increments needed to get from `first` to `last`.

```
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;`

22.4.3 Range iterator operations [range.iter.ops]

1 The library includes the function templates `ranges::advance`, `ranges::distance`, `ranges::next`, and `ranges::prev` to manipulate iterators. These operations adapt to the set of operators provided by each iterator category to provide the most efficient implementation possible for a concrete iterator type. [*Example*: `ranges::advance` uses the `+` operator to move a `RandomAccessIterator` forward `n` steps in constant time. For an iterator type that does not model `RandomAccessIterator`, `ranges::advance` instead performs `n` individual increments with the `++` operator. — *end example*]

2 The function templates defined in this subclause are not found by argument-dependent name lookup (**??**). When found by unqualified (**??**) 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 (**??**) than `std::ranges::distance` since the former requires its first two parameters to have the same type. — *end example*]

3 The number and order of deducible template parameters for the function templates defined in this subclause is unspecified, except where explicitly stated otherwise.

22.4.3.1 `ranges::advance` [range.iter.op.advance]

```
template<Iterator I>
  constexpr void advance(I& i, iter_difference_t<I> n);
```

1 *Expects:* If `I` does not model `BidirectionalIterator`, `n` is not negative.

2 *Effects:*

(2.1) — If `I` models `RandomAccessIterator`, equivalent to `i += n`.

(2.2) — Otherwise, if `n` is non-negative, increments `i` by `n`.

(2.3) — Otherwise, decrements `i` by `-n`.

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

3 *Expects:* `[i, bound)` denotes a range.

4 *Effects:*

(4.1) — If `I` and `S` model `Assignable<I&, S>`, equivalent to `i = std::move(bound)`.

(4.2) — Otherwise, if `S` and `I` model `SizedSentinel<S, I>`, equivalent to `ranges::advance(i, bound - i)`.

(4.3) — Otherwise, while `bool(i != bound)` is `true`, increments `i`.

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

5 *Expects:* If `n > 0`, `[i, bound)` denotes a range. If `n == 0`, `[i, bound)` or `[bound, i)` denotes a range. If `n < 0`, `[bound, i)` denotes a range, `I` models `BidirectionalIterator`, and `I` and `S` model `Same<I, S>`.

6 *Effects:*

(6.1) — If `S` and `I` model `SizedSentinel<S, I>`:

(6.1.1) — If $|$`n`$| \geq |$`bound - i`$|$, equivalent to `ranges::advance(i, bound)`.

(6.1.2) — Otherwise, equivalent to `ranges::advance(i, n)`.

(6.2) — Otherwise,

(6.2.1) — if `n` is non-negative, while `bool(i != bound)` is `true`, increments `i` but at most `n` times.

(6.2.2) — Otherwise, while `bool(i != bound)` is `true`, decrements `i` but at most `-n` times.

7 *Returns:* `n - `M, where M is the difference between the ending and starting positions of `i`.

22.4.3.2 `ranges::distance` [range.iter.op.distance]

```
template<Iterator I, Sentinel<I> S>
```

```
constexpr iter_difference_t<I> distance(I first, S last);
```

1 *Expects:* `[first, last)` denotes a range, or `[last, first)` denotes a range and `S` and `I` model `Same<S, I> && SizedSentinel<S, I>`.

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

```
template<Range R>
  constexpr iter_difference_t<iterator_t<R>> distance(R&& r);
```

3 *Effects:* If `R` models `SizedRange`, equivalent to:

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

Otherwise, equivalent to:

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

22.4.3.3 `ranges::next` [range.iter.op.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_t<I> n);
```

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

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

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

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

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

22.4.3.4 `ranges::prev` [range.iter.op.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_t<I> n);
```

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

```
template<BidirectionalIterator I>
  constexpr I prev(I x, iter_difference_t<I> n, I bound);
```

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

22.5 Iterator adaptors [predef.iterators]

22.5.1 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.

22.5.1.1 Class template `reverse_iterator` [reverse.iterator]

```
namespace std {
  template<class Iterator>
  class reverse_iterator {
  public:
    using iterator_type     = Iterator;
    using iterator_concept  = see below;
    using iterator_category = see below;
    using value_type        = iter_value_t<Iterator>;
```

```
  using difference_type   = iter_difference_t<Iterator>;
  using pointer           = typename iterator_traits<Iterator>::pointer;
  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;
  constexpr reference operator*() const;
  constexpr pointer   operator->() const requires see below;

  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>)
  inline constexpr bool disable_sized_sentinel<reverse_iterator<Iterator1>,
                                               reverse_iterator<Iterator2>> = true;
}
```

1 The member *typedef-name* `iterator_concept` denotes

(1.1) — `random_access_iterator_tag` if `Iterator` models `RandomAccessIterator`, and

(1.2) — `bidirectional_iterator_tag` otherwise.

2 The member *typedef-name* `iterator_category` denotes

(2.1) — `random_access_iterator_tag` if the type `iterator_traits<Iterator>::iterator_category` models `DerivedFrom<random_access_iterator_tag>`, and

(2.2) — `iterator_traits<Iterator>::iterator_category` otherwise.

22.5.1.2 Requirements [reverse.iter.requirements]

1 The template parameter `Iterator` shall either meet the requirements of a *Cpp17BidirectionalIterator* (22.3.5.5) or model `BidirectionalIterator` (22.3.4.12).

2 Additionally, `Iterator` shall either meet the requirements of a *Cpp17RandomAccessIterator* (22.3.5.6) or model `RandomAccessIterator` (22.3.4.13) if the definitions of any of the members

(2.1) — `operator+`, `operator-`, `operator+=`, `operator-=` (22.5.1.6), or

(2.2) — `operator[]` (22.5.1.5),

or the non-member operators (22.5.1.7)

(2.3) — `operator<`, `operator>`, `operator<=`, `operator>=`, `operator-`, or `operator+` (22.5.1.8)

are instantiated (**??**).

22.5.1.3 Construction and assignment [reverse.iter.cons]

```
constexpr reverse_iterator();
```

1 *Effects:* Value-initializes `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 reverse_iterator(Iterator x);
```

2 *Effects:* Initializes `current` with `x`.

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

3 *Effects:* Initializes `current` with `u.current`.

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

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

5 *Returns:* `*this`.

22.5.1.4 Conversion [reverse.iter.conv]

```
constexpr Iterator base() const;          // explicit
```

1 *Returns:* `current`.

22.5.1.5 Element access [reverse.iter.elem]

```
constexpr reference operator*() const;
```

1 *Effects:* As if by:

```
Iterator tmp = current;
return *--tmp;
```

```
constexpr pointer operator->() const
  requires (is_pointer_v<Iterator> ||
            requires (const Iterator i) { i.operator->(); });
```

2 *Effects:*

(2.1) — If `Iterator` is a pointer type, equivalent to: `return prev(current);`

(2.2) — Otherwise, equivalent to: `return prev(current).operator->();`

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

3 *Returns:* `current[-n-1]`.

22.5.1.6 Navigation [reverse.iter.nav]

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

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

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

2 *Returns:* `reverse_iterator(current+n)`.

```
constexpr reverse_iterator& operator++();
```

3 *Effects:* As if by: `--current;`

4 *Returns:* `*this`.

```
constexpr reverse_iterator operator++(int);
```

5 *Effects:* As if by:

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

```
constexpr reverse_iterator& operator--();
```

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

7 *Returns:* `*this`.

```
constexpr reverse_iterator operator--(int);
```

8 *Effects:* As if by:

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

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

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

10 *Returns:* `*this`.

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

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

12 *Returns:* `*this`.

22.5.1.7 Comparisons [reverse.iter.cmp]

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

```
    const reverse_iterator<Iterator2>& y);
```

1 *Constraints:* ~~The expression~~ `x.current == y.current` ~~shall be valid~~is well-formed and convertible to `bool`.

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 *Constraints:* ~~The expression~~ `x.current != y.current` ~~shall be valid~~is well-formed and convertible to `bool`.

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 *Constraints:* ~~The expression~~ `x.current > y.current` ~~shall be valid~~is well-formed and convertible to `bool`.

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 *Constraints:* ~~The expression~~ `x.current < y.current` ~~shall be valid~~is well-formed and convertible to `bool`.

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

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

9 *Constraints:* ~~The expression~~ `x.current >= y.current` ~~shall be valid~~is well-formed and convertible to `bool`.

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

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

11 *Constraints:* ~~The expression~~ `x.current <= y.current` ~~shall be valid~~is well-formed and convertible to `bool`.

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

22.5.1.8 Non-member functions [reverse.iter.nonmember]

```
template<class Iterator1, class Iterator2>
    constexpr auto operator-(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y) -> decltype(y.base() - x.base());
```

1 *Returns:* `y.current - x.current`.

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

2 *Returns:* `reverse_iterator<Iterator> (x.current - n)`.

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

3 *Effects:* Equivalent to:

```
auto tmp = i.current;
return ranges::iter_move(--tmp);
```

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

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

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

5 *Effects:* Equivalent to:

```
auto xtmp = x.current;
auto ytmp = y.current;
ranges::iter_swap(--xtmp, --ytmp);
```

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

```
is_nothrow_copy_constructible_v<Iterator> &&
is_nothrow_copy_constructible_v<Iterator2> &&
noexcept(ranges::iter_swap(--declval<Iterator&>(), --declval<Iterator2&>()))
```

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

7 *Returns:* `reverse_iterator<Iterator>(i)`.

22.5.2 Insert iterators [insert.iterators]

1 To make it possible to deal with insertion in the same way as writing into an array, a special kind of iterator adaptors, called *insert iterators*, are provided in the library. With regular iterator classes,

```
while (first != last) *result++ = *first++;
```

causes a range `[first, last)` to be copied into a range starting with result. The same code with `result` being an insert iterator will insert corresponding elements into the container. This device allows all of the copying algorithms in the library to work in the *insert mode* instead of the *regular overwrite* mode.

2 An insert iterator is constructed from a container and possibly one of its iterators pointing to where insertion takes place if it is neither at the beginning nor at the end of the container. Insert iterators satisfy the requirements of output iterators. `operator*` returns the insert iterator itself. The assignment `operator=(const T& x)` is defined on insert iterators to allow writing into them, it inserts `x` right before where the insert iterator is pointing. In other words, an insert iterator is like a cursor pointing into the container where the insertion takes place. `back_insert_iterator` inserts elements at the end of a container, `front_insert_iterator` inserts elements at the beginning of a container, and `insert_iterator` inserts elements where the iterator points to in a container. `back_inserter`, `front_inserter`, and `inserter` are three functions making the insert iterators out of a container.

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

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

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

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

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

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

22.5.2.1.1 Operations [back.insert.iter.ops]

`constexpr explicit back_insert_iterator(Container& x);`

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

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

2 *Effects:* As if by: `container->push_back(value);`

3 *Returns:* `*this`.

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

4 *Effects:* As if by: `container->push_back(std::move(value));`

5 *Returns:* `*this`.

`constexpr back_insert_iterator& operator*();`

6 *Returns:* `*this`.

```
constexpr back_insert_iterator& operator++();
constexpr back_insert_iterator  operator++(int);
```

7 *Returns:* `*this`.

22.5.2.1.2 `back_inserter` [back.inserter]

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

1 *Returns:* `back_insert_iterator<Container>(x)`.

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

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

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

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

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

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

22.5.2.2.1 Operations [front.insert.iter.ops]

```
constexpr explicit front_insert_iterator(Container& x);
```

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

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

2 *Effects:* As if by: `container->push_front(value);`

3 *Returns:* `*this`.

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

4 *Effects:* As if by: `container->push_front(std::move(value));`

5 *Returns:* `*this`.

```
constexpr front_insert_iterator& operator*();
```

6 *Returns:* `*this`.

```
constexpr front_insert_iterator& operator++();
constexpr front_insert_iterator  operator++(int);
```

7 *Returns:* `*this`.

22.5.2.2.2 `front_inserter` [front.inserter]

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

1 *Returns:* `front_insert_iterator<Container>(x)`.

22.5.2.3 Class template `insert_iterator` [insert.iterator]

```
namespace std {
  template<class Container>
  class insert_iterator {
  protected:
    Container* container = nullptr;
    iterator_t<Container> iter = iterator_t<Container>();

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

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

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

```
  template<class Container>
    constexpr insert_iterator<Container>
      inserter(Container& x, iterator_t<Container> i);
}
```

22.5.2.3.1 Operations [insert.iter.ops]

```
constexpr insert_iterator(Container& x, iterator_t<Container> i);
```

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

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

2 *Effects:* As if by:

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

3 *Returns:* `*this`.

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

4 *Effects:* As if by:

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

5 *Returns:* `*this`.

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

6 *Returns:* `*this`.

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

7 *Returns:* `*this`.

22.5.2.3.2 `inserter` [inserter]

```
template<class Container>
  constexpr insert_iterator<Container>
    inserter(Container& x, iterator_t<Container> i);
```

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

22.5.3 Move iterators and sentinels [move.iterators]

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*]

22.5.3.1 Class template `move_iterator` [move.iterator]

```
namespace std {
  template<class Iterator>
  class move_iterator {
  public:
    using iterator_type     = Iterator;
    using iterator_concept  = input_iterator_tag;
    using iterator_category = see below;
    using value_type        = iter_value_t<Iterator>;
    using difference_type   = iter_difference_t<Iterator>;
```

```
  using pointer           = Iterator;
  using reference         = iter_rvalue_reference_t<Iterator>;

  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 auto 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 reference operator[](difference_type n) const;

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

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+(iter_difference_t<Iterator> n, const move_iterator<Iterator>& x);
  template<class Iterator>
    constexpr move_iterator<Iterator> make_move_iterator(Iterator i);
}
```

1 The member *typedef-name* `iterator_category` denotes

(1.1) — `random_access_iterator_tag` if the type `iterator_traits<Iterator>::iterator_category` models `DerivedFrom<random_access_iterator_tag>`, and

(1.2) — `iterator_traits<Iterator>::iterator_category` otherwise.

22.5.3.2 Requirements [move.iter.requirements]

1 The template parameter `Iterator` shall either meet the *Cpp17InputIterator* requirements (22.3.5.2) or model `InputIterator` (22.3.4.9). Additionally, if any of the bidirectional traversal functions are instantiated, the template parameter shall either meet the *Cpp17BidirectionalIterator* requirements (22.3.5.5) or model `BidirectionalIterator` (22.3.4.12). If any of the random access traversal functions are instantiated, the template parameter shall either meet the *Cpp17RandomAccessIterator* requirements (22.3.5.6) or model `RandomAccessIterator` (22.3.4.13).

22.5.3.3 Construction and assignment [move.iter.cons]

```
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 *Mandates:* U is convertible to `Iterator`.

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

5 ~~*Requires:* U shall be convertible to `Iterator`.~~

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

6 *Mandates:* U is convertible to `Iterator`.

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

8 ~~*Requires:* U shall be convertible to `Iterator`.~~

22.5.3.4 Conversion [move.iter.op.conv]

```
constexpr Iterator base() const;
```

1 *Returns:* `current`.

22.5.3.5 Element access [move.iter.elem]

```
constexpr reference operator*() const;
```

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

```
constexpr pointer operator->() const;
```

2 *Returns:* `current`.

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

3 *Effects:* Equivalent to: `ranges::iter_move(current + n);`

22.5.3.6 Navigation [move.iter.nav]

```
constexpr move_iterator& operator++();
```

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

2 *Returns:* `*this`.

```
constexpr auto operator++(int);
```

3 *Effects:* If `Iterator` models `ForwardIterator`, equivalent to:

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

Otherwise, equivalent to `++current`.

```
constexpr move_iterator& operator--();
```

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

5 *Returns:* `*this`.

```
constexpr move_iterator operator--(int);
```

6 *Effects:* As if by:

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

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

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

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

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

9 *Returns:* `*this`.

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

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

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

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

12 *Returns:* `*this`.

22.5.3.7 Comparisons [move.iter.op.comp]

```
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);
```

1 *Constraints:* ~~The expression~~ `x.base() == y.base()` ~~shall be valid~~is well-formed and convertible to `bool`.

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 *Constraints:* ~~The expression~~ `x.base() == y.base()` ~~shall be valid~~is well-formed and convertible to `bool`.

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

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

5 *Constraints:* ~~The expression~~ `x.base() < y.base()` ~~shall be valid~~is well-formed and convertible to `bool`.

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

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

7 *Constraints:* ~~The expression~~ `y.base() < x.base()` ~~shall be valid~~is well-formed and convertible to `bool`.

8 *Returns:* `y < x`.

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

9 *Constraints:* ~~The expression~~ `y.base() < x.base()` ~~shall be valid~~is well-formed and convertible to `bool`.

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

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

11 *Constraints:* ~~The expression~~ `x.base() < y.base()` ~~shall be valid~~is well-formed and convertible to `bool`.

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

22.5.3.8 Non-member functions [move.iter.nonmember]

```
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);
```

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

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

2 *Constraints:* ~~The expression~~ `x + n` ~~shall be valid~~is well-formed and ~~have~~has type `Iterator`.

3 *Returns:* `x + n`.

```
friend constexpr iter_rvalue_reference_t<Iterator>
  iter_move(const move_iterator& i)
    noexcept(noexcept(ranges::iter_move(i.current)));
```

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

```
template<IndirectlySwappable<Iterator> Iterator2>
  friend constexpr void
    iter_swap(const move_iterator& x, const move_iterator<Iterator2>& y)
      noexcept(noexcept(ranges::iter_swap(x.current, y.current)));
```

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

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

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

22.5.3.9 Class template `move_sentinel` [move.sentinel]

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` model `Sentinel<S, I>`, `move_sentinel<S>` and `move_iterator<I>` model `Sentinel<move_sentinel<S>, move_iterator<I>>` 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) {
  std::ranges::copy_if(move_iterator<I>{first}, move_sentinel<S>{last}, out, pred);
}
```

— *end example*]

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

    constexpr S base() const;
  private:
    S last;      // exposition only
  };
}
```

22.5.3.10 Operations [move.sent.ops]

```
constexpr move_sentinel();
```

1 *Effects:* Value-initializes `last`. If `is_trivially_default_constructible_v<S>` is `true`, then this constructor is a `constexpr` constructor.

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

2 *Effects:* Initializes `last` with `std::move(s)`.

```
template<class S2>
  requires ConvertibleTo<const S2&, S>
    constexpr move_sentinel(const move_sentinel<S2>& s);
```

3 *Effects:* Initializes `last` with `s.last`.

```
template<class S2>
  requires Assignable<S&, const S2&>
    constexpr move_sentinel& operator=(const move_sentinel<S2>& s);
```

4 *Effects:* Equivalent to: `last = s.last; return *this;`

22.5.4 Common iterators [iterators.common]

22.5.4.1 Class template `common_iterator` [common.iterator]

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.

2 [*Note*: The `common_iterator` type is useful for interfacing with legacy code that expects the begin and end of a range to have the same type. — *end note*]

3 [*Example*:

```
template<class ForwardIterator>
void fun(ForwardIterator begin, ForwardIterator end);

list<int> s;
// populate the list s
using CI = common_iterator<counted_iterator<list<int>::iterator>, default_sentinel_t>;
// call fun on a range of 10 ints
fun(CI(counted_iterator(s.begin(), 10)), CI(default_sentinel));
```

— *end example*]

```
namespace std {
  template<Iterator I, Sentinel<I> S>
    requires (!Same<I, S>)
  class common_iterator {
  public:
    constexpr common_iterator() = default;
    constexpr common_iterator(I i);
    constexpr common_iterator(S s);
    template<class I2, class S2>
      requires ConvertibleTo<const I2&, I> && ConvertibleTo<const S2&, S>
        constexpr common_iterator(const common_iterator<I2, S2>& x);

    template<class I2, class S2>
      requires ConvertibleTo<const I2&, I> && ConvertibleTo<const S2&, S> &&
               Assignable<I&, const I2&> && Assignable<S&, const S2&>
        common_iterator& operator=(const common_iterator<I2, S2>& x);

    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);
```

```
    template<class I2, Sentinel<I> S2>
      requires Sentinel<S, I2>
    friend bool operator==(
      const common_iterator& x, const common_iterator<I2, S2>& y);
    template<class I2, Sentinel<I> S2>
      requires Sentinel<S, I2> && EqualityComparableWith<I, I2>
    friend bool operator==(
      const common_iterator& x, const common_iterator<I2, S2>& y);
    template<class I2, Sentinel<I> S2>
      requires Sentinel<S, I2>
    friend bool operator!=(
      const common_iterator& x, const common_iterator<I2, S2>& y);

    template<SizedSentinel<I> I2, SizedSentinel<I> S2>
      requires SizedSentinel<S, I2>
    friend iter_difference_t<I2> operator-(
      const common_iterator& x, const common_iterator<I2, S2>& y);

    friend iter_rvalue_reference_t<I> iter_move(const common_iterator& i)
      noexcept(noexcept(ranges::iter_move(declval<const I&>())))
        requires InputIterator<I>;
    template<IndirectlySwappable<I> I2, class S2>
      friend void iter_swap(const common_iterator& x, const common_iterator<I2, S2>& y)
        noexcept(noexcept(ranges::iter_swap(declval<const I&>(), declval<const I2&>())));

  private:
    variant<I, S> v_;    // exposition only
  };

  template<class I, class S>
  struct incrementable_traits<common_iterator<I, S>> {
    using difference_type = iter_difference_t<I>;
  };

  template<InputIterator I, class S>
  struct iterator_traits<common_iterator<I, S>> {
    using iterator_concept = see below;
    using iterator_category = see below;
    using value_type = iter_value_t<I>;
    using difference_type = iter_difference_t<I>;
    using pointer = see below;
    using reference = iter_reference_t<I>;
  };
}
```

22.5.4.2 Associated types [common.iter.types]

1 The nested *typedef-name*s of the specialization of `iterator_traits` for `common_iterator<I, S>` are defined as follows.

(1.1) — `iterator_concept` denotes `forward_iterator_tag` if `I` models `ForwardIterator`; otherwise it denotes `input_iterator_tag`.

(1.2) — `iterator_category` denotes `forward_iterator_tag` if `iterator_traits<I>::iterator_category` models `DerivedFrom<forward_iterator_tag>`; otherwise it denotes `input_iterator_tag`.

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

22.5.4.3 Constructors and conversions [common.iter.const]

```
constexpr common_iterator(I i);
```

1 *Effects:* Initializes `v_` as if by `v_{in_place_type<I>, std::move(i)}`.

```
constexpr common_iterator(S s);
```

2 *Effects:* Initializes `v_` as if by `v_{in_place_type<S>, std::move(s)}`.

```
template<class I2, class S2>
  requires ConvertibleTo<const I2&, I> && ConvertibleTo<const S2&, S>
    constexpr common_iterator(const common_iterator<I2, S2>& x);
```

3 *Expects:* `x.v_.valueless_by_exception()` is `false`.

4 *Effects:* Initializes `v_` as if by `v_{in_place_index<`*i*`>, get<`*i*`>(x.v_)}`, where *i* is `x.v_.index()`.

```
template<class I2, class S2>
  requires ConvertibleTo<const I2&, I> && ConvertibleTo<const S2&, S> &&
           Assignable<I&, const I2&> && Assignable<S&, const S2&>
    common_iterator& operator=(const common_iterator<I2, S2>& x);
```

5 *Expects:* `x.v_.valueless_by_exception()` is `false`.

6 *Effects:* Equivalent to:

(6.1) — If `v_.index() == x.v_.index()`, then `get<`*i*`>(v_) = get<`*i*`>(x.v_)`.

(6.2) — Otherwise, `v_.emplace<`*i*`>(get<`*i*`>(x.v_))`.

where *i* is `x.v_.index()`.

7 *Returns:* `*this`

22.5.4.4 Accessors [common.iter.access]

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

1 *Expects:* `holds_alternative<I>(v_)`.

2 *Effects:* Equivalent to: `return *get<I>(v_);`

```
decltype(auto) operator->() const
  requires see below;
```

3 The expression in the requires clause is equivalent to:

```
Readable<const I> &&
(requires(const I& i) { i.operator->(); } ||
 is_reference_v<iter_reference_t<I>> ||
 Constructible<iter_value_t<I>, iter_reference_t<I>>)
```

4 *Expects:* `holds_alternative<I>(v_)`.

5 *Effects:*

(5.1) — If `I` is a pointer type or if the expression `get<I>(v_).operator->()` is well-formed, equivalent to: `return get<I>(v_);`

(5.2) — Otherwise, if `iter_reference_t<I>` is a reference type, equivalent to:

```
auto&& tmp = *get<I>(v_);
return addressof(tmp);
```

(5.3) — Otherwise, equivalent to: `return` *`proxy`*`(*get<I>(v_));` where *`proxy`* is the exposition-only class:

```
class proxy {
  iter_value_t<I> keep_;
  proxy(iter_reference_t<I>&& x)
    : keep_(std::move(x)) {}
public:
  const iter_value_t<I>* operator->() const {
    return addressof(keep_);
  }
};
```

22.5.4.5 Navigation [common.iter.nav]

```
common_iterator& operator++();
```

1 *Expects:* `holds_alternative<I>(v_)`.

2 *Effects:* Equivalent to `++get<I>(v_)`.

3 *Returns:* `*this`.

```
decltype(auto) operator++(int);
```

4 *Expects:* `holds_alternative<I>(v_)`.

5 *Effects:* If `I` models `ForwardIterator`, equivalent to:

```
common_iterator tmp = *this;
++*this;
return tmp;
```

Otherwise, equivalent to: `return get<I>(v_)++;`

22.5.4.6 Comparisons [common.iter.cmp]

```
template<class I2, Sentinel<I> S2>
  requires Sentinel<S, I2>
friend bool operator==(
  const common_iterator& x, const common_iterator<I2, S2>& y);
```

1 *Expects:* `x.v_.valueless_by_exception()` and `y.v_.valueless_by_exception()` are each `false`.

2 *Returns:* `true` if i `==` j, and otherwise `get<`i`>(x.v_) == get<`j`>(y.v_)`, where i is `x.v_.index()` and j is `y.v_.index()`.

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

3 *Expects:* `x.v_.valueless_by_exception()` and `y.v_.valueless_by_exception()` are each `false`.

4 *Returns:* `true` if i and j are each `1`, and otherwise `get<`i`>(x.v_) == get<`j`>(y.v_)`, where i is `x.v_.index()` and j is `y.v_.index()`.

```
template<class I2, Sentinel<I> S2>
  requires Sentinel<S, I2>
friend bool operator!=(
  const common_iterator& x, const common_iterator<I2, S2>& y);
```

5 *Effects:* Equivalent to: `return !(x == y);`

```
template<SizedSentinel<I> I2, SizedSentinel<I> S2>
  requires SizedSentinel<S, I2>
friend iter_difference_t<I2> operator-(
  const common_iterator& x, const common_iterator<I2, S2>& y);
```

6 *Expects:* `x.v_.valueless_by_exception()` and `y.v_.valueless_by_exception()` are each `false`.

7 *Returns:* `0` if i and j are each `1`, and otherwise `get<`i`>(x.v_) - get<`j`>(y.v_)`, where i is `x.v_.index()` and j is `y.v_.index()`.

22.5.4.7 Customization [common.iter.cust]

```
friend iter_rvalue_reference_t<I> iter_move(const common_iterator& i)
  noexcept(noexcept(ranges::iter_move(declval<const I&>())))
    requires InputIterator<I>;
```

1 *Expects:* `holds_alternative<I>(v_)`.

2 *Effects:* Equivalent to: `return ranges::iter_move(get<I>(i.v_));`

```
template<IndirectlySwappable<I> I2, class S2>
  friend void iter_swap(const common_iterator& x, const common_iterator<I2, S2>& y)
    noexcept(noexcept(ranges::iter_swap(declval<const I&>(), declval<const I2&>())));
```

3 *Expects:* `holds_alternative<I>(x.v_)` and `holds_alternative<I2>(y.v_)` are each `true`.

4 *Effects:* Equivalent to `ranges::iter_swap(get<I>(x.v_), get<I2>(y.v_))`.

22.5.5 Default sentinels [default.sentinels]

```
namespace std {
  struct default_sentinel_t { };
}
```

1 Class `default_sentinel_t` is an empty type used to denote the end of a range. It can be used together with iterator types that know the bound of their range (e.g., `counted_iterator` (22.5.6.1)).

22.5.6 Counted iterators [iterators.counted]

22.5.6.1 Class template `counted_iterator` [counted.iterator]

1 Class template `counted_iterator` is an iterator adaptor with the same behavior as the underlying iterator except that it keeps track of the distance to the end of its range. It can be used together with `default_sentinel` in calls to generic algorithms to operate on a range of N elements starting at a given position without needing to know the end position a priori.

2 [*Example*:

```
list<string> s;
// populate the list s with at least 10 strings
vector<string> v;
// copies 10 strings into v:
ranges::copy(counted_iterator(s.begin(), 10), default_sentinel, back_inserter(v));
```

— *end example*]

3 Two values `i1` and `i2` of types `counted_iterator<I1>` and `counted_iterator<I2>` refer to elements of the same sequence if and only if `next(i1.base(), i1.count())` and `next(i2.base(), i2.count())` refer to the same (possibly past-the-end) element.

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

    constexpr counted_iterator() = default;
    constexpr counted_iterator(I x, iter_difference_t<I> n);
    template<class I2>
      requires ConvertibleTo<const I2&, I>
        constexpr counted_iterator(const counted_iterator<I2>& x);

    template<class I2>
      requires Assignable<I&, const I2&>
        constexpr counted_iterator& operator=(const counted_iterator<I2>& x);

    constexpr I base() const;
    constexpr iter_difference_t<I> count() const noexcept;
    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+(iter_difference_t<I> n) const
    requires RandomAccessIterator<I>;
  friend constexpr counted_iterator operator+(
    iter_difference_t<I> n, const counted_iterator& x)
      requires RandomAccessIterator<I>;
  constexpr counted_iterator& operator+=(iter_difference_t<I> n)
    requires RandomAccessIterator<I>;

  constexpr counted_iterator operator-(iter_difference_t<I> n) const
    requires RandomAccessIterator<I>;
  template<Common<I> I2>
    friend constexpr iter_difference_t<I2> operator-(
      const counted_iterator& x, const counted_iterator<I2>& y);
  friend constexpr iter_difference_t<I> operator-(
    const counted_iterator& x, default_sentinel_t);
  friend constexpr iter_difference_t<I> operator-(
    default_sentinel_t, const counted_iterator& y);
  constexpr counted_iterator& operator-=(iter_difference_t<I> n)
    requires RandomAccessIterator<I>;

  constexpr decltype(auto) operator[](iter_difference_t<I> n) const
    requires RandomAccessIterator<I>;

  template<Common<I> I2>
    friend constexpr bool operator==(
      const counted_iterator& x, const counted_iterator<I2>& y);
  friend constexpr bool operator==(
    const counted_iterator& x, default_sentinel_t);
  friend constexpr bool operator==(
    default_sentinel_t, const counted_iterator& x);

  template<Common<I> I2>
    friend constexpr bool operator!=(
      const counted_iterator& x, const counted_iterator<I2>& y);
  friend constexpr bool operator!=(
    const counted_iterator& x, default_sentinel_t y);
  friend constexpr bool operator!=(
    default_sentinel_t x, const counted_iterator& y);

  template<Common<I> I2>
    friend constexpr bool operator<(
      const counted_iterator& x, const counted_iterator<I2>& y);
  template<Common<I> I2>
    friend constexpr bool operator>(
      const counted_iterator& x, const counted_iterator<I2>& y);
  template<Common<I> I2>
    friend constexpr bool operator<=(
      const counted_iterator& x, const counted_iterator<I2>& y);
  template<Common<I> I2>
    friend constexpr bool operator>=(
      const counted_iterator& x, const counted_iterator<I2>& y);

  friend constexpr iter_rvalue_reference_t<I> iter_move(const counted_iterator& i)
    noexcept(noexcept(ranges::iter_move(i.current)))
      requires InputIterator<I>;
  template<IndirectlySwappable<I> I2>
    friend constexpr void iter_swap(const counted_iterator& x, const counted_iterator<I2>& y)
      noexcept(noexcept(ranges::iter_swap(x.current, y.current)));

private:
  I current = I();                    // exposition only
```

```
    iter_difference_t<I> length = 0;     // exposition only
  };

  template<class I>
  struct incrementable_traits<counted_iterator<I>> {
    using difference_type = iter_difference_t<I>;
  };

  template<InputIterator I>
  struct iterator_traits<counted_iterator<I>> : iterator_traits<I> {
    using pointer = void;
  };
}
```

22.5.6.2 Constructors and conversions [counted.iter.const]

```
constexpr counted_iterator(I i, iter_difference_t<I> n);
```

1 *Expects:* `n >= 0`.

2 *Effects:* Initializes `current` with `i` and `length` with `n`.

```
template<class I2>
  requires ConvertibleTo<const I2&, I>
    constexpr counted_iterator(const counted_iterator<I2>& x);
```

3 *Effects:* Initializes `current` with `x.current` and `length` with `x.length`.

```
template<class I2>
  requires Assignable<I&, const I2&>
    constexpr counted_iterator& operator=(const counted_iterator<I2>& x);
```

4 *Effects:* Assigns `x.current` to `current` and `x.length` to `length`.

5 *Returns:* `*this`.

22.5.6.3 Accessors [counted.iter.access]

```
constexpr I base() const;
```

1 *Effects:* Equivalent to: `return current;`

```
constexpr iter_difference_t<I> count() const noexcept;
```

2 *Effects:* Equivalent to: `return length;`

22.5.6.4 Element access [counted.iter.elem]

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

1 *Effects:* Equivalent to: `return *current;`

```
constexpr decltype(auto) operator[](iter_difference_t<I> n) const
  requires RandomAccessIterator<I>;
```

2 *Expects:* `n < length`.

3 *Effects:* Equivalent to: `return current[n];`

22.5.6.5 Navigation [counted.iter.nav]

```
constexpr counted_iterator& operator++();
```

1 *Expects:* `length > 0`.

2 *Effects:* Equivalent to:

```
++current;
--length;
return *this;
```

```
decltype(auto) operator++(int);
```

3 *Expects:* `length > 0`.

4 *Effects:* Equivalent to:

```
--length;
try { return current++; }
catch(...) { ++length; throw; }
```

```
constexpr counted_iterator operator++(int)
  requires ForwardIterator<I>;
```

5 *Effects:* Equivalent to:

```
counted_iterator tmp = *this;
++*this;
return tmp;
```

```
constexpr counted_iterator& operator--();
  requires BidirectionalIterator<I>
```

6 *Effects:* Equivalent to:

```
--current;
++length;
return *this;
```

```
constexpr counted_iterator operator--(int)
  requires BidirectionalIterator<I>;
```

7 *Effects:* Equivalent to:

```
counted_iterator tmp = *this;
--*this;
return tmp;
```

```
constexpr counted_iterator operator+(iter_difference_t<I> n) const
  requires RandomAccessIterator<I>;
```

8 *Effects:* Equivalent to: `return counted_iterator(current + n, length - n);`

```
friend constexpr counted_iterator operator+(
  iter_difference_t<I> n, const counted_iterator& x)
    requires RandomAccessIterator<I>;
```

9 *Effects:* Equivalent to: `return x + n;`

```
constexpr counted_iterator& operator+=(iter_difference_t<I> n)
  requires RandomAccessIterator<I>;
```

10 *Expects:* `n <= length`.

11 *Effects:* Equivalent to:

```
current += n;
length -= n;
return *this;
```

```
constexpr counted_iterator operator-(iter_difference_t<I> n) const
  requires RandomAccessIterator<I>;
```

12 *Effects:* Equivalent to: `return counted_iterator(current - n, length + n);`

```
template<Common<I> I2>
  friend constexpr iter_difference_t<I2> operator-(
    const counted_iterator& x, const counted_iterator<I2>& y);
```

13 *Expects:* `x` and `y` refer to elements of the same sequence (22.5.6.1).

14 *Effects:* Equivalent to: `return y.length - x.length;`

```
friend constexpr iter_difference_t<I> operator-(
  const counted_iterator& x, default_sentinel_t);
```

15 *Effects:* Equivalent to: `return -x.length;`

```
friend constexpr iter_difference_t<I> operator-(
  default_sentinel_t, const counted_iterator& y);
```

16 *Effects:* Equivalent to: `return y.length;`

```
constexpr counted_iterator& operator-=(iter_difference_t<I> n)
  requires RandomAccessIterator<I>;
```

17 *Expects:* `-n <= length`.

18 *Effects:* Equivalent to:

```
current -= n;
length += n;
return *this;
```

22.5.6.6 Comparisons [counted.iter.cmp]

```
template<Common<I> I2>
  friend constexpr bool operator==(
    const counted_iterator& x, const counted_iterator<I2>& y);
```

1 *Expects:* `x` and `y` refer to elements of the same sequence (22.5.6.1).

2 *Effects:* Equivalent to: `return x.length == y.length;`

```
friend constexpr bool operator==(
  const counted_iterator& x, default_sentinel_t);
friend constexpr bool operator==(
  default_sentinel_t, const counted_iterator& x);
```

3 *Effects:* Equivalent to: `return x.length == 0;`

```
template<Common<I> I2>
  friend constexpr bool operator!=(
    const counted_iterator& x, const counted_iterator<I2>& y);
friend constexpr bool operator!=(
  const counted_iterator& x, default_sentinel_t y);
friend constexpr bool operator!=(
  default_sentinel_t x, const counted_iterator& y);
```

4 *Effects:* Equivalent to: `return !(x == y);`

```
template<Common<I> I2>
  friend constexpr bool operator<(
    const counted_iterator& x, const counted_iterator<I2>& y);
```

5 *Expects:* `x` and `y` refer to elements of the same sequence (22.5.6.1).

6 *Effects:* Equivalent to: `return y.length < x.length;`

7 [*Note*: The argument order in the *Effects* element is reversed because `length` counts down, not up. — *end note*]

```
template<Common<I> I2>
  friend constexpr bool operator>(
    const counted_iterator& x, const counted_iterator<I2>& y);
```

8 *Effects:* Equivalent to: `return y < x;`

```
template<Common<I> I2>
  friend constexpr bool operator<=(
    const counted_iterator& x, const counted_iterator<I2>& y);
```

9 *Effects:* Equivalent to: `return !(y < x);`

```
template<Common<I> I2>
  friend constexpr bool operator>=(
    const counted_iterator& x, const counted_iterator<I2>& y);
```

10 *Effects:* Equivalent to: `return !(x < y);`

22.5.6.7 Customizations [counted.iter.cust]

```
friend constexpr iter_rvalue_reference_t<I>
  iter_move(const counted_iterator& i)
    noexcept(noexcept(ranges::iter_move(i.current)))
    requires InputIterator<I>;
```

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

```
template<IndirectlySwappable<I> I2>
  friend constexpr void
    iter_swap(const counted_iterator& x, const counted_iterator<I2>& y)
      noexcept(noexcept(ranges::iter_swap(x.current, y.current)));
```

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

22.5.7 Unreachable sentinel [unreachable.sentinels]

22.5.7.1 Class `unreachable_sentinel_t` [unreachable.sentinel]

1 Class `unreachable_sentinel_t` can be used with any `WeaklyIncrementable` type to denote the "upper bound" of an open interval.

2 [*Example*:

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

Provided a newline character really exists in the buffer, the use of `unreachable_sentinel` 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 {
  struct unreachable_sentinel_t {
    template<WeaklyIncrementable I>
      friend constexpr bool operator==(unreachable_sentinel_t, const I&) noexcept;
    template<WeaklyIncrementable I>
      friend constexpr bool operator==(const I&, unreachable_sentinel_t) noexcept;
    template<WeaklyIncrementable I>
      friend constexpr bool operator!=(unreachable_sentinel_t, const I&) noexcept;
    template<WeaklyIncrementable I>
      friend constexpr bool operator!=(const I&, unreachable_sentinel_t) noexcept;
  };
}
```

22.5.7.2 Comparisons [unreachable.sentinel.cmp]

```
template<WeaklyIncrementable I>
  friend constexpr bool operator==(unreachable_sentinel_t, const I&) noexcept;
template<WeaklyIncrementable I>
  friend constexpr bool operator==(const I&, unreachable_sentinel_t) noexcept;
```

1 *Returns:* `false`.

```
template<WeaklyIncrementable I>
  friend constexpr bool operator!=(unreachable_sentinel_t, const I&) noexcept;
template<WeaklyIncrementable I>
  friend constexpr bool operator!=(const I&, unreachable_sentinel_t) noexcept;
```

2 *Returns:* `true`.

22.6 Stream iterators [stream.iterators]

1 To make it possible for algorithmic templates to work directly with input/output streams, appropriate iterator-like class templates are provided.

[*Example*:

```
partial_sum(istream_iterator<double, char>(cin),
  istream_iterator<double, char>(),
  ostream_iterator<double, char>(cout, "\n"));
```

reads a file containing floating-point numbers from `cin`, and prints the partial sums onto `cout`. — *end example*]

22.6.1 Class template `istream_iterator` [istream.iterator]

1 The class template `istream_iterator` is an input iterator (22.3.5.2) that reads (using `operator>>`) successive elements from the input stream for which it was constructed. After it is constructed, and every time `++` is used, the iterator reads and stores a value of `T`. If the iterator fails to read and store a value of `T` (`fail()` on the stream returns `true`), the iterator becomes equal to the *end-of-stream* iterator value. The constructor with no arguments `istream_iterator()` always constructs an end-of-stream input iterator object, which is the only legitimate iterator to be used for the end condition. The result of `operator*` on an end-of-stream iterator is not defined. For any other iterator value a `const T&` is returned. The result of `operator->` on an end-of-stream iterator is not defined. For any other iterator value a `const T*` is returned. The behavior of a program that applies `operator++()` to an end-of-stream iterator is undefined. It is impossible to store things into istream iterators. The type `T` shall satisfy the *Cpp17DefaultConstructible*, *Cpp17CopyConstructible*, and *Cpp17CopyAssignable* requirements.

2 Two end-of-stream iterators are always equal. An end-of-stream iterator is not equal to a non-end-of-stream iterator. Two non-end-of-stream iterators are equal when they are constructed from the same stream.

```
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_t);
    istream_iterator(istream_type& s);
    istream_iterator(const istream_iterator& x) = default;
    ~istream_iterator() = default;
    istream_iterator& operator=(const istream_iterator&) = default;

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

    friend bool operator==(const istream_iterator& i, default_sentinel_t);
    friend bool operator==(default_sentinel_t, const istream_iterator& i);
    friend bool operator!=(const istream_iterator& x, default_sentinel_t y);
    friend bool operator!=(default_sentinel_t x, const istream_iterator& 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);
}
```

22.6.1.1 Constructors and destructor [istream.iterator.cons]

```
constexpr istream_iterator();
constexpr istream_iterator(default_sentinel_t);
```

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

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

```
istream_iterator(istream_type& s);
```

3 *Effects:* Initializes `in_stream` with `addressof(s)`. `value` may be initialized during construction or the first time it is referenced.

4 *Ensures:* `in_stream == addressof(s)`.

```
istream_iterator(const istream_iterator& x) = default;
```

5 *Effects:* Constructs a copy of `x`. If `is_trivially_copy_constructible_v<T>` is `true`, then this constructor is a trivial copy constructor.

6 *Ensures:* `in_stream == x.in_stream`.

7 *Remarks:* If `is_trivially_copy_constructible_v<T>` is `true`, then this constructor is a trivial copy constructor.

```
~istream_iterator() = default;
```

8 *Effects:* The iterator is destroyed. If `is_trivially_destructible_v<T>` is `true`, then this destructor is trivial.

22.6.1.2 Operations [istream.iterator.ops]

```
const T& operator*() const;
```

1 *Returns:* `value`.

```
const T* operator->() const;
```

2 *Returns:* `addressof(operator*())`.

```
istream_iterator& operator++();
```

3 ~~*Requires:*~~ *Expects:* `in_stream != 0` is `true`.

4 *Effects:* As if by: `*in_stream >> value;`

5 *Returns:* `*this`.

```
istream_iterator operator++(int);
```

6 ~~*Requires:*~~ *Expects:* `in_stream != 0` is `true`.

7 *Effects:* As if by:

```
istream_iterator tmp = *this;
*in_stream >> value;
return (tmp);
```

```
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);
```

8 *Returns:* `x.in_stream == y.in_stream`.

```
friend bool operator==(default_sentinel_t, const istream_iterator& i);
friend bool operator==(const istream_iterator& i, default_sentinel_t);
```

9 *Returns:* `!i.in_stream`.

```
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_t x, const istream_iterator& y);
```

```
friend bool operator!=(const istream_iterator& x, default_sentinel_t y);
```

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

22.6.2 Class template `ostream_iterator` [ostream.iterator]

1 `ostream_iterator` writes (using `operator<<`) successive elements onto the output stream from which it was constructed. If it was constructed with `charT*` as a constructor argument, this string, called a *delimiter string*, is written to the stream after every `T` is written. It is not possible to get a value out of the output iterator. Its only use is as an output iterator in situations like

```
while (first != last)
  *result++ = *first++;
```

2 `ostream_iterator` is defined as:

```
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   = ptrdiff_t;
    using pointer           = void;
    using reference         = void;
    using char_type         = charT;
    using traits_type       = traits;
    using ostream_type      = basic_ostream<charT,traits>;

    constexpr ostreambuf_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 ostream_iterator&) = default;
    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
  };
}
```

22.6.2.1 Constructors and destructor [ostream.iterator.cons.des]

```
ostream_iterator(ostream_type& s);
```

1 *Effects:* Initializes `out_stream` with `addressof(s)` and `delim` with null.

```
ostream_iterator(ostream_type& s, const charT* delimiter);
```

2 *Effects:* Initializes `out_stream` with `addressof(s)` and `delim` with `delimiter`.

```
ostream_iterator(const ostream_iterator& x);
```

3 *Effects:* Constructs a copy of `x`.

```
~ostream_iterator();
```

4 *Effects:* The iterator is destroyed.

22.6.2.2 Operations [ostream.iterator.ops]

```
ostream_iterator& operator=(const T& value);
```

1 *Effects:* As if by:

```
      *out_stream << value;
      if (delim != 0)
        *out_stream << delim;
      return *this;
```

```
ostream_iterator& operator*();
```

2 *Returns:* `*this`.

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

3 *Returns:* `*this`.

22.6.3 Class template `istreambuf_iterator` [istreambuf.iterator]

1 The class template `istreambuf_iterator` defines an input iterator (22.3.5.2) that reads successive *characters* from the streambuf for which it was constructed. `operator*` provides access to the current input character, if any. Each time `operator++` is evaluated, the iterator advances to the next input character. If the end of stream is reached (`streambuf_type::sgetc()` returns `traits::eof()`), the iterator becomes equal to the *end-of-stream* iterator value. The default constructor `istreambuf_iterator()` and the constructor `istreambuf_iterator(0)` both construct an end-of-stream iterator object suitable for use as an end-of-range. All specializations of `istreambuf_iterator` shall have a trivial copy constructor, a `constexpr` default constructor, and a trivial destructor.

2 The result of `operator*()` on an end-of-stream iterator is undefined. For any other iterator value a `char_type` value is returned. It is impossible to assign a character via an input iterator.

```
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_t) 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;
    istreambuf_iterator& operator=(const istreambuf_iterator&) noexcept = default;
    charT operator*() const;
    istreambuf_iterator& operator++();
    proxy operator++(int);
    bool equal(const istreambuf_iterator& b) const;

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

22.6.3.1 Class `istreambuf_iterator::proxy` [istreambuf.iterator.proxy]

```
namespace std {
  template<class charT, class traits>
  class istreambuf_iterator<charT, traits>::proxy { // exposition only
    charT keep_;
    basic_streambuf<charT,traits>* sbuf_;
    proxy(charT c, basic_streambuf<charT,traits>* sbuf)
      : keep_(c), sbuf_(sbuf) { }
  public:
    charT operator*() { return keep_; }
  };
}
```

1 Class `istreambuf_iterator<charT,traits>::proxy` is for exposition only. An implementation is permitted to provide equivalent functionality without providing a class with this name. Class `istreambuf_iterator<charT, traits>::proxy` provides a temporary placeholder as the return value of the post-increment operator (`operator++`). It keeps the character pointed to by the previous value of the iterator for some possible future access to get the character.

22.6.3.2 Constructors [istreambuf.iterator.cons]

1 For each `istreambuf_iterator` constructor in this subclause, an end-of-stream iterator is constructed if and only if the exposition-only member `sbuf_` is initialized with a null pointer value.

```
constexpr istreambuf_iterator() noexcept;
constexpr istreambuf_iterator(default_sentinel_t) noexcept;
```

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

```
istreambuf_iterator(istream_type& s) noexcept;
```

3 *Effects:* Initializes `sbuf_` with `s.rdbuf()`.

```
istreambuf_iterator(streambuf_type* s) noexcept;
```

4 *Effects:* Initializes `sbuf_` with `s`.

```
istreambuf_iterator(const proxy& p) noexcept;
```

5 *Effects:* Initializes `sbuf_` with `p.sbuf_`.

22.6.3.3 Operations [istreambuf.iterator.ops]

```
charT operator*() const
```

1 *Returns:* The character obtained via the `streambuf` member `sbuf_->sgetc()`.

```
istreambuf_iterator& operator++();
```

2 *Effects:* As if by `sbuf_->sbumpc()`.

3 *Returns:* `*this`.

```
proxy operator++(int);
```

4 *Returns:* `proxy(sbuf_->sbumpc(), sbuf_)`.

```
bool equal(const istreambuf_iterator& b) const;
```

5 *Returns:* `true` if and only if both iterators are at end-of-stream, or neither is at end-of-stream, regardless of what `streambuf` object they use.

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

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

22.6.4 Class template `ostreambuf_iterator` [ostreambuf.iterator]

```
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   = 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
  };
}
```

1 The class template `ostreambuf_iterator` writes successive *characters* onto the output stream from which it was constructed. It is not possible to get a character value out of the output iterator.

22.6.4.1 Constructors [ostreambuf.iter.cons]

```
ostreambuf_iterator(ostream_type& s) noexcept;
```

1 ~~*Requires:*~~ *Expects:* `s.rdbuf()` ~~shall not be~~is not a null pointer.

2 *Effects:* Initializes `sbuf_` with `s.rdbuf()`.

```
ostreambuf_iterator(streambuf_type* s) noexcept;
```

3 ~~*Requires:*~~ *Expects:* `s` ~~shall not be~~is not a null pointer.

4 *Effects:* Initializes `sbuf_` with `s`.

22.6.4.2 Operations [ostreambuf.iter.ops]

```
ostreambuf_iterator& operator=(charT c);
```

1 *Effects:* If `failed()` yields `false`, calls `sbuf_->sputc(c)`; otherwise has no effect.

2 *Returns:* `*this`.

```
ostreambuf_iterator& operator*();
```

3 *Returns:* `*this`.

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

4 *Returns:* `*this`.

```
bool failed() const noexcept;
```

5 *Returns:* `true` if in any prior use of member `operator=`, the call to `sbuf_->sputc()` returned `traits::eof()`; or `false` otherwise.

22.7 Range access [iterator.range]

1 In addition to being available via inclusion of the `<iterator>` header, the function templates in 22.7 are available when any of the following headers are included: `<array>`, `<deque>`, `<forward_list>`, `<list>`, `<map>`, `<regex>`, `<set>`, `<span>`, `<string>`, `<string_view>`, `<unordered_map>`, `<unordered_set>`, and `<vector>`. Each of these templates is a designated customization point (**??**).

```
template<class C> constexpr auto begin(C& c) -> decltype(c.begin());
template<class C> constexpr auto begin(const C& c) -> decltype(c.begin());
```

2 *Returns:* `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());
```

3 *Returns:* `c.end()`.

```
template<class T, size_t N> constexpr T* begin(T (&array)[N]) noexcept;
```

4 *Returns:* `array`.

```
template<class T, size_t N> constexpr T* end(T (&array)[N]) noexcept;
```

5 *Returns:* `array + N`.

```
template<class C> constexpr auto cbegin(const C& c) noexcept(noexcept(std::begin(c)))
  -> decltype(std::begin(c));
```

6 *Returns:* `std::begin(c)`.

```
template<class C> constexpr auto cend(const C& c) noexcept(noexcept(std::end(c)))
  -> decltype(std::end(c));
```

7 *Returns:* `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());
```

8 *Returns:* `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());
```

9 *Returns:* `c.rend()`.

```
template<class T, size_t N> constexpr reverse_iterator<T*> rbegin(T (&array)[N]);
```

10 *Returns:* `reverse_iterator<T*>(array + N)`.

```
template<class T, size_t N> constexpr reverse_iterator<T*> rend(T (&array)[N]);
```

11 *Returns:* `reverse_iterator<T*>(array)`.

```
template<class E> constexpr reverse_iterator<const E*> rbegin(initializer_list<E> il);
```

12 *Returns:* `reverse_iterator<const E*>(il.end())`.

```
template<class E> constexpr reverse_iterator<const E*> rend(initializer_list<E> il);
```

13 *Returns:* `reverse_iterator<const E*>(il.begin())`.

```
template<class C> constexpr auto crbegin(const C& c) -> decltype(std::rbegin(c));
```

14 *Returns:* `std::rbegin(c)`.

```
template<class C> constexpr auto crend(const C& c) -> decltype(std::rend(c));
```

15 *Returns:* `std::rend(c)`.

22.8 Container and view access [iterator.container]

1 In addition to being available via inclusion of the `<iterator>` header, the function templates in 22.8 are available when any of the following headers are included: `<array>`, `<deque>`, `<forward_list>`, `<list>`, `<map>`, `<regex>`, `<set>`, `<span>`, `<string>`, `<string_view>`, `<unordered_map>`, `<unordered_set>`, and `<vector>`. Each of these templates is a designated customization point (**??**).

```
template<class C> constexpr auto size(const C& c) -> decltype(c.size());
```

2 *Returns:* `c.size()`.

```
template<class T, size_t N> constexpr size_t size(const T (&array)[N]) noexcept;
```

3 *Returns:* `N`.

```
template<class C> [[nodiscard]] constexpr auto empty(const C& c) -> decltype(c.empty());
```

4 *Returns:* `c.empty()`.

```
template<class T, size_t N> [[nodiscard]] constexpr bool empty(const T (&array)[N]) noexcept;
```

5 *Returns:* `false`.

```
template<class E> [[nodiscard]] constexpr bool empty(initializer_list<E> il) noexcept;
```

6 *Returns:* `il.size() == 0`.

```
template<class C> constexpr auto data(C& c) -> decltype(c.data());
template<class C> constexpr auto data(const C& c) -> decltype(c.data());
```

7 *Returns:* `c.data()`.

```
template<class T, size_t N> constexpr T* data(T (&array)[N]) noexcept;
```

8 *Returns:* `array`.

```
template<class E> constexpr const E* data(initializer_list<E> il) noexcept;
```

9 *Returns:* `il.begin()`.