Document Number:	P0896R2
Date:	2018-06-25
Audience:	Library Evolution Working Group, Library Working Group

Authors:	Eric Niebler Casey Carter Christopher Di Bella
Reply to:	Casey Carter casey@carter.net

The One Ranges Proposal
(was Merging the Ranges TS)

Three Proposals for `Views` under the Sky,
Seven for LEWG in their halls of stone,
Nine for the Ranges TS doomed to die,
One for LWG on its dark throne
in the Land of Geneva where the Standards lie.

One Proposal to `ranges::merge` them all, One Proposal to `ranges::find` them,
One Proposal to bring them all and in `namespace ranges` bind them,
In the Land of Geneva where the Standards lie.

With apologies to J.R.R. Tolkien.

Contents

1 Scope [intro.scope]

"Eventually, all things merge into one, and a river runs through it."

—*Norman Maclean*

1 This document proposes to merge the ISO/IEC TS 21425:2017, aka the Ranges TS, into the working draft. This document is intended to be taken in conjunction with P0898, a paper which proposes importing the definitions of the Ranges TS's Concepts library (Clause 7) into namespace std.

1.1 Revision History [intro.history]

1.1.1 Revision 2 [intro.history.r2]

- Merge P0789R3.
- Merge P1033R1.
- Reformulate non-member operators of common_iterator and counted_iterator as members or hidden friends.
- Merge P1037R0.
- Merge P0970R1: Drop dangling per LEWG request, and make calls that would have returned a dangling iterator ill-formed instead by redefining safe_iterator_t<R> to be ill-formed when iterators from R may dangle.
- Merge P0944R0.
- Drop tagged and related machinery. Algorithm foo that did return a tagged tuple or pair now instead returns a named type foo_result with public data members whose names are the same as the previous set of tag names. Exceptions:
  - The single-range transform overload returns unary_transform_result.
  - The dual-range transform overload returns binary_transform_result.
- LEWG was disturbed by the use of enable_if to define difference_type and value_type; use requires clauses instead.
- Per LEWG direction, rename header <range> to <ranges> to agree with the namespace name std::ranges.
- Remove inappropriate usage of value_type_t in the insert iterators: design intent of value_type_t is to be only an associated type trait for Readable, and the Container type parameter of the insert iterators is not Readable.
- Use remove_cvref_t where appropriate.
- Restore the design intent that neither Writable types nor non-Readable Iterator types are required to have an equality-preserving * operator.
- Require semantics for the Constructible requirements of IndirectlyMovableStorable and IndirectlyCopyableSto
- Declare constexpr the algorithms that are so declared in the working draft.
- constexpr some move_sentinel members that we apparently missed in P0579.

1.1.2 Revision 1 [intro.history.r1]

- Remove section [std2.numerics] which is incorporated into P0898.
- Do not propose ranges::exchange: it is not used in the Ranges TS.
- Rewrite nearly everything to merge into std::ranges[1] rather than into std2:
  - Occurrences of "std2." in stable names are either removed, or replaced with "range" when the name resulting from removal would conflict with an existing stable name.

1) std::two was another popular suggestion.

— Incorporate the `std2::swap` customization point from P0898R0 as `ranges::swap`. (This was necessarily dropped from P0898R1.) Perform the necessary surgery on the `Swappable` concept from P0898R1 to restore the intended design that uses the renamed customization point.

2 General Principles [intro]

2.1 Goals [intro.goals]

1 The primary goal of this proposal is to deliver high-quality, constrained generic Standard Library components at the same time that the language gets support for such components.

2.2 Rationale [intro.rationale]

1 The best, and arguably only practical way to achieve the goal stated above is by incorporating the Ranges TS into the working paper. The sooner we can agree on what we want "`Iterator`" and "`Range`" to mean going forward (for instance), and the sooner users are able to rely on them, the sooner we can start building and delivering functionality on top of those fundamental abstractions. (For example, see "P0789: Range Adaptors and Utilities" ([4]).)

2 The cost of not delivering such a set of Standard Library concepts and algorithms is that users will either do without or create a babel of mutually incompatible concepts and algorithms, often without the rigour followed by the Ranges TS. The experience of the authors and implementors of the Ranges TS is that getting concept definitions and algorithm constraints right is *hard*. The Standard Library should save its users from needless heartache.

2.3 Risks [intro.risks]

1 Shipping constrained components from the Ranges TS in the C++20 timeframe is not without risk. As of the time of writing (February 1, 2018), no major Standard Library vendor has shipped an implementation of the Ranges TS. Two of the three major compiler vendors have not even shipped an implementation of the concepts language feature. Arguably, we have not yet gotten the usage experience for which all Technical Specifications are intended.

2 On the other hand, the components of Ranges TS have been vetted very thoroughly by the range-v3 ([3]) project, on which the Ranges TS is based. There is no part of the Ranges TS – concepts included – that has not seen extensive use via range-v3. (The concepts in range-v3 are emulated with high fidelity through the use of generalized SFINAE for expressions.) As an Open Source project, usage statistics are hard to come by, but the following may be indicitive:

(2.1) — The range-v3 GitHub project has over 1,400 stars, over 120 watchers, and 145 forks.

(2.2) — It is cloned on average about 6,000 times a month.

(2.3) — A GitHub search, restricted to C++ files, for the string "`range/v3`" (a path component of all of range-v3's header files), turns up over 7,000 hits.

3 Lacking true concepts, range-v3 cannot emulate concept-based function overloading, or the sorts of constraints-checking short-circuit evaluation required by true concepts. For that reason, the authors of the Ranges TS have created a reference implementation: CMCSTL2 ([1]) using true concepts. To this reference implementation, the authors ported all of range-v3's tests. These exposed only a handful of concepts-specific bugs in the components of the Ranges TS (and a great many more bugs in compilers). Those improvements were back-ported to range-v3 where they have been thoroughly vetted over the past 2 years.

4 In short, concern about lack of implementation experience should not be a reason to withhold this important Standard Library advance from users.

2.4 Methodology [intro.methedology]

1 The contents of the Ranges TS, Clause 7 ("Concepts library") are proposed for namespace `std` by P0898, "Standard Library Concepts" ([2]). Additionally, P0898 proposes the `identity` function object (ISO/IEC TS 21425:2017 §[func.identity]) and the `common_reference` type trait (ISO/IEC TS 21425:2017 §[meta.trans.other]) for namespace `std`. The changes proposed by the Ranges TS to `common_type` are merged into the working paper (also by P0898). The "`invoke`" function and the "`swappable`" type traits (e.g., `is_swappable_with`) already exist in the text of the working paper, so they are omitted here.

2 The salient, high-level features of this proposal are as follows:

(2.1) — The remaining library components in the Ranges TS are proposed for namespace `::std::ranges`.

(2.2) — The text of the Ranges TS is rebased on the latest working draft.

(2.3) — Structurally, this paper proposes to specify each piece of `std::ranges` alongside the content of `std` from the same header. Since some Ranges TS components reuse names that previously had meaning in the C++ Standard, we sometimes rename old content to avoid name collisions.

(2.4) — The content of headers from the Ranges TS with the same base name as a standard header are merged into that standard header. For example, the content of `<experimental/ranges/iterator>` will be merged into `<iterator>`. The new header `<experimental/ranges/range>` will be added under the name `<ranges>`.

(2.5) — The Concepts Library clause, proposed by P0898, is located in that paper between the "Language Support Library" and the "Diagnostics library". In the organization proposed by this paper, that places it as subclause 20.3. This paper refers to it as such. FIXME

(2.6) — Where the text of the Ranges TS needs to be updated, the text is presented with change markings: ~~red strikethrough~~ for removed text and blue underline for added text. FIXME

(2.7) — The stable names of everything in the Ranges TS, clauses 6, 8-12 are changed by preprending "`range.`". References are updated accordingly.

2.5 Style of presentation [intro.style]

1 The remainder of this document is a technical specification in the form of editorial instructions directing that changes be made to the text of the C++ working draft. The formatting of the text suggests the origin of each portion of the wording.

Existing wording from the C++ working draft - included to provide context - is presented without decoration.

Entire clauses / subclauses / paragraphs incorporated from the Ranges TS are presented in a distinct cyan color.

In-line additions of wording from the Ranges TS to the C++ working draft are presented in cyan with underline.

~~In-line bits of wording that the Ranges TS strikes from the C++ working draft are presented in red with strike-through.~~

Wording to be added which is original to this document appears in gold with underline.

~~Wording which this document strikes is presented in magenta with strikethrough. (Hopefully context makes it clear whether the wording is currently in the C++ working draft, or wording that is not being added from the Ranges TS.)~~

Ideally, these formatting conventions make it clear which wording comes from which document in this three-way merge.

20 Library introduction [library]

[...]

20.1 General [library.general]

[...]

[Editor's note: Insert a new row in Table 15 for the ranges library:]

Table 15 — Library categories

Clause	Category
Clause [language.support]	Language support library
Clause 22	Concepts library
Clause [diagnostics]	Diagnostics library
Clause 24	General utilities library
Clause 25	Strings library
Clause [localization]	Localization library
Clause 27	Containers library
Clause 28	Iterators library
Clause 30	Algorithms library
Clause 29	Ranges library
Clause 31	Numerics library
Clause [input.output]	Input/output library
Clause [re]	Regular expressions library
Clause [atomics]	Atomic operations library
Clause [thread]	Thread support library

[...]

9 The containers (Clause 27), iterators (Clause 28), algorithms (Clause 30), and ranges (Clause 29) libraries provide a C++ program with access to a subset of the most widely used algorithms and data structures.

20.3 Definitions [definitions]

[...]

20.3.18 [defns.projection]

projection

⟨function object argument⟩ transformation which an algorithm applies before inspecting the values of elements

[*Example:*

```
std::pair<int, const char*> pairs[] = {{2, "foo"}, {1, "bar"}, {0, "baz"}};
std::ranges::sort(pairs, std::less<>{}, [](auto const& p) { return p.first; });
```

sorts the pairs in increasing order of their `first` members:

```
{{0, "baz"}, {1, "bar"}, {2, "foo"}}
```

— end example]

[...]

20.5 Library-wide requirements [requirements]

[...]

20.5.1.2 Headers [headers]

[Editor's note: Add header `<ranges>` to Table 16:]

[...]

Table 16 — C++ library headers

`<algorithm>`	`<forward_list>`	`<new>`	`<string>`
`<any>`	`<fstream>`	`<numeric>`	`<string_view>`
`<array>`	`<functional>`	`<optional>`	`<strstream>`
`<atomic>`	`<future>`	`<ostream>`	`<syncstream>`
`<bit>`	`<initializer_list>`	`<queue>`	`<system_error>`
`<bitset>`	`<iomanip>`	`<random>`	`<thread>`
`<charconv>`	`<ios>`	`<ranges>`	`<tuple>`
`<chrono>`	`<iosfwd>`	`<ratio>`	`<typeindex>`
`<codecvt>`	`<iostream>`	`<regex>`	`<typeinfo>`
`<compare>`	`<istream>`	`<scoped_allocator>`	`<type_traits>`
`<complex>`	`<iterator>`	`<set>`	`<unordered_map>`
`<concepts>`	`<limits>`	`<shared_mutex>`	`<unordered_set>`
`<condition_variable>`	`<list>`	`<span>`	`<utility>`
`<contract>`	`<locale>`	`<sstream>`	`<valarray>`
`<deque>`	`<map>`	`<stack>`	`<variant>`
`<exception>`	`<memory>`	`<stdexcept>`	`<vector>`
`<execution>`	`<memory_resource>`	`<streambuf>`	`<version>`
`<filesystem>`	`<mutex>`		

20.5.1.3 *Cpp17Allocator* requirements [allocator.requirements]

[...]

5 An allocator type `X` shall satisfy the *Cpp17CopyConstructible* requirements (Table [copyconstructible]). The `X::pointer`, `X::const_pointer`, `X::void_pointer`, and `X::const_void_pointer` types shall satisfy the *Cpp17NullablePointer* requirements (Table [nullablepointer]). No constructor, comparison function, copy operation, move operation, or swap operation on these pointer types shall exit via an exception. `X::pointer` and `X::const_pointer` shall also satisfy the requirements for a random access iterator (28.3.5.6) and ~~of a contiguous iterator (28.3.1).~~ the additional requirement that

```
addressof(*(a + (b - a))) == addressof(*a) + (b - a)
```

must hold for pointer values `a` and `b`.

22 Concepts library [concepts]

22.3 Language-related concepts [concepts.lang]

22.3.11 Concept `Swappable` [concept.swappable]

[Editor's note: Modify the definitions of the `Swappable` and `SwappableWith` concepts as follows (This restores the Ranges TS design for these concepts from which P0898 had to deviate due to the absence of the `ranges::swap` customization point):]

```
template<class T>
concept Swappable = is_swappable_v<T>; // see below
concept Swappable = requires(T& a, T& b) { ranges::swap(a, b); };
```

1 Let `a1` and `a2` denote distinct equal objects of type `T`, and let `b1` and `b2` similarly denote distinct equal objects of type `T`. `Swappable<T>` is satisfied only if:

(1.1) — After evaluating either `swap(a1, b1)` or `swap(b1, a1)` in the context described below, `a1` is equal to `b2` and `b1` is equal to `a2`.

2 The context in which `swap(a1, b1)` or `swap(b1, a1)` are evaluated shall ensure that a binary non-member function named "swap" is selected via overload resolution ([over.match]) on a candidate set that includes:

(2.1) — the two `swap` function templates defined in `<utility>` (24.2) and

(2.2) — the lookup set produced by argument-dependent lookup ([basic.lookup.argdep]).

```
template<class T, class U>
concept SwappableWith =
  is_swappable_with_v<T, T> && is_swappable_with_v<U, U> && // see below
  CommonReference<const remove_reference_t<T>&, const remove_reference_t<U>&> &&
  is_swappable_with_v<T, U> && is_swappable_with_v<U, T>; // see below
  requires(T&& t, U&& u) {
    ranges::swap(std::forward<T>(t), std::forward<T>(t));
    ranges::swap(std::forward<U>(u), std::forward<U>(u));
    ranges::swap(std::forward<T>(t), std::forward<U>(u));
    ranges::swap(std::forward<U>(u), std::forward<T>(t));
  };
```

3 Let `t1` and `t2` denote distinct equal objects of type `remove_cvref_t<T>`, and E_t be an expression that denotes `t1` such that `decltype((`E_t`))` is `T`. Let `u1` and `u2` similarly denote distinct equal objects of type `remove_cvref_t<U>`, and E_u be an expression that denotes `u1` such that `decltype((`E_u`))` is `U`. Let `C` be `common_reference_t<const remove_reference_t<T>&, const remove_reference_t<U>&>`. `SwappableWith<T, U>` is satisfied only if:

(3.1) — After evaluating either `swap(`E_t`, `E_u`)` or `swap(`E_u`, `E_t`)` in the context described above, `C(t1)` is equal to `C(u2)` and `C(u1)` is equal to `C(t2)`.

4 The context in which `swap(`E_t`, `E_u`)` or `swap(`E_u`, `E_t`)` are evaluated shall ensure that a binary non-member function named "swap" is selected via overload resolution ([over.match]) on a candidate set that includes:

(4.1) — the two `swap` function templates defined in `<utility>` (24.2) and

(4.2) — the lookup set produced by argument-dependent lookup ([basic.lookup.argdep]).

5 This subclause provides definitions for swappable types and expressions. In these definitions, let `t` denote an expression of type `T`, and let `u` denote an expression of type `U`.

6 An object `t` is *swappable with* an object `u` if and only if `SwappableWith<T, U>` is satisfied. `SwappableWith<T, U>` is satisfied only if given distinct objects `t2` equal to `t` and `u2` equal to `u`, after evaluating either `ranges::swap(t, u)` or `ranges::swap(u, t)`, `t2` is equal to `u` and `u2` is equal to `t`.

7 An rvalue or lvalue `t` is *swappable* if and only if `t` is swappable with any rvalue or lvalue, respectively, of type `T`.

[*Example:* User code can ensure that the evaluation of `swap` calls is performed in an appropriate context under the various conditions as follows:

```
#include <concepts>
#include <utility>

template<class T, classstd::SwappableWith<T> U>
void value_swap(T&& t, U&& u) {
  ranges::swap(std::forward<T>(t), std::forward<U>(u)); // OK: uses "swappable with" conditions
                                                        // for rvalues and lvalues
}

// Requires: lvalues of T shall be swappable.
template<classstd::Swappable T>
void lv_swap(T& t1, T& t2) {
  ranges::swap(t1, t2);                                 // OK: uses swappable conditions for
}                                                       // lvalues of type T

namespace N {
  struct A { int m; };
  struct Proxy { A* a; };
  Proxy proxy(A& a) { return Proxy{ &a }; }

  void swap(A& x, Proxy p) {
    ranges::swap(x.m, p.a->m);                 // OK: uses context equivalent to swappable
                                               // conditions for fundamental types
  }
  void swap(Proxy p, A& x) { swap(x, p); }   // satisfy symmetry constraint
}
```

```
int main() {
  int i = 1, j = 2;
  lv_swap(i, j);
  assert(i == 2 && j == 1);

  N::A a1 = { 5 }, a2 = { -5 };
  value_swap(a1, proxy(a2));
  assert(a1.m == -5 && a2.m == 5);
}
```

— *end example*]

24 General utilities library [utilities]

24.2 Utility components [utility]

24.2.1 Header <utility> synopsis [utility.syn]

[Editor's note: Add declarations to the `<utility>` synopsis (class template `tagged` and associated machinery from the Ranges TS are intentionally ommitted.):]

```
[...]
template<size_t I>
  struct in_place_index_t {
    explicit in_place_index_t() = default;
  };
template<size_t I> inline constexpr in_place_index_t<I> in_place_index{};

namespace ranges {
  // 24.2.3, ranges::swap
  inline namespace unspecified {
    inline constexpr unspecified swap = unspecified;
  }
}~~}}~~
}
```

[Editor's note: Insert the specification of `ranges::swap` after [utility.swap]:]

24.2.3 `ranges::swap` [range.swap]

1 The name `ranges::swap` denotes a customization point object ([customization.point.object]). The ~~effect of the~~ expression `ranges::swap(E1, E2)` for some subexpressions `E1` and `E2` is expression-equivalent to:

(1.1) — `(void)swap(E1, E2)`[2], if that expression is valid, with overload resolution performed in a context that includes the declarations

```
template<class T>
void swap(T&, T&) = delete;
template<class T, size_t N>
void swap(T(&)[N], T(&)[N]) = delete;
```

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

(1.2) — Otherwise, `(void)ranges::swap_ranges(E1, E2)` if `E1` and `E2` are lvalues of array types ([basic.compound]) with equal extent and `ranges::swap(*(E1), *(E2))` is a valid expression, except that `noexcept(ranges::swap(E1, E2))` is equal to `noexcept(ranges::swap(*(E1), *(E2)))`.

(1.3) — Otherwise, if `E1` and `E2` are lvalues of the same type `T` which meets the syntactic requirements of `MoveConstructible<T>` and `Assignable<T&, T>`, exchanges the referenced values. `ranges::swap(E1, E2)` is a constant expression if the constructor selected by overload resolution for `T{std::move(E1)}` is a constexpr constructor and the expression `E1 = std::move(E2)` can appear in a constexpr function. `noexcept(ranges::swap(E1, E2))` is equal to `is_nothrow_move_constructible_v<T>`~~`::value`~~ `&&`

2) The name `swap` is used here unqualified.

`is_nothrow_move_assignable_v<T>~~::value~~`. If either `MoveConstructible` or `Assignable` is not satisfied, the program is ill-formed with no diagnostic required.

(1.4) — Otherwise, `ranges::swap(E1, E2)` is ill-formed.

2 ~~*Remarks:*~~ [*Note:* Whenever `ranges::swap(E1, E2)` is a valid expression, it exchanges the values referenced by `E1` and `E2` and has type `void`. — *end note*]

[...]

24.10 Memory [memory]

[...]

24.10.2 Header <memory> synopsis [memory.syn]

[...]

```
namespace std {
  [...]

  // [default.allocator], the default allocator
  template<class T> class allocator;
  template<class T, class U>
    bool operator==(const allocator<T>&, const allocator<U>&) noexcept;
  template<class T, class U>
    bool operator!=(const allocator<T>&, const allocator<U>&) noexcept;

  // 24.10.11, specialized algorithms
  // 24.10.11.1, special memory concepts
  template<class I>
  concept no-throw-input-iterator = see below; // exposition only

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

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

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

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


  template<class T>
    constexpr T* addressof(T& r) noexcept;
  template<class T>
    const T* addressof(const T&&) = delete;
  template<class ForwardIterator>
    void uninitialized_default_construct(ForwardIterator first, ForwardIterator last);
  template<class ExecutionPolicy, class ForwardIterator>
    void uninitialized_default_construct(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                                         ForwardIterator first, ForwardIterator last);
  template<class ForwardIterator, class Size>
    ForwardIterator uninitialized_default_construct_n(ForwardIterator first, Size n);
  template<class ExecutionPolicy, class ForwardIterator, class Size>
    ForwardIterator uninitialized_default_construct_n(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                                                      ForwardIterator first, Size n);

  namespace ranges {
    template<no-throw-forward-iterator I, no-throw-sentinel<I> S>
        requires DefaultConstructible<iter_value_type_t<I>>
      I uninitialized_default_construct(I first, S last);
    template<no-throw-forward-range Rng>
        requires DefaultConstructible<iter_value_type_t<iterator_t<Rng>>>
      safe_iterator_t<Rng> uninitialized_default_construct(Rng&& rng);
```

```
  template<no-throw-forward-iterator I>
      requires DefaultConstructible<iter_value_type_t<I>>
    I uninitialized_default_construct_n(I first, iter_difference_type_t<I> n);
}

template<class ForwardIterator>
  void uninitialized_value_construct(ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator>
  void uninitialized_value_construct(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                                     ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Size>
  ForwardIterator uninitialized_value_construct_n(ForwardIterator first, Size n);
template<class ExecutionPolicy, class ForwardIterator, class Size>
  ForwardIterator uninitialized_value_construct_n(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                                                  ForwardIterator first, Size n);

namespace ranges {
  template<no-throw-forward-iterator I, no-throw-sentinel<I> S>
      requires DefaultConstructible<iter_value_type_t<I>>
    I uninitialized_value_construct(I first, S last);
  template<no-throw-forward-range Rng>
      requires DefaultConstructible<iter_value_type_t<iterator_t<Rng>>>
    safe_iterator_t<Rng> uninitialized_value_construct(Rng&& rng);

  template<no-throw-forward-iterator I>
      requires DefaultConstructible<iter_value_type_t<I>>
    I uninitialized_value_construct_n(I first, iter_difference_type_t<I> n);
}

template<class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_copy(InputIterator first, InputIterator last,
                                     ForwardIterator result);
template<class ExecutionPolicy, class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_copy(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                                     InputIterator first, InputIterator last,
                                     ForwardIterator result);
template<class InputIterator, class Size, class ForwardIterator>
  ForwardIterator uninitialized_copy_n(InputIterator first, Size n,
                                       ForwardIterator result);
template<class ExecutionPolicy, class InputIterator, class Size, class ForwardIterator>
  ForwardIterator uninitialized_copy_n(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                                       InputIterator first, Size n,
                                       ForwardIterator result);

namespace ranges {
  template<class I, class O>
  struct uninitialized_copy_result {
    I in;
    I out;
  };
  template<InputIterator I, Sentinel<I> S1, no-throw-forward-iterator O, no-throw-sentinel<O> S2>
      requires Constructible<iter_value_type_t<O>, iter_reference_t<I>>
    tagged_pair<tag::in(I), tag::out(O)>
    uninitialized_copy_result<I, O>
      uninitialized_copy(I ifirst, S1 ilast, O ofirst, S2 olast);
  template<InputRange IRng, no-throw-forward-range ORng>
      requires Constructible<iter_value_type_t<iterator_t<ORng>>, iter_reference_t<iterator_t<IRng>>>
    tagged_pair<tag::in(safe_iterator_t<IRng>), tag::out(safe_iterator_t<ORng>)>
    uninitialized_copy_result<iterator_t<IRng>, iterator_t<ORng>
      uninitialized_copy(IRng&& irng, ORng&& orng);

  template<class I, class O>
  using uninitialized_copy_n_result = uninitialized_copy_result<I, O>;
  template<InputIterator I, no-throw-forward-iterator O, no-throw-sentinel<O> S>
```

```
      requires Constructible<iter_value~~_type~~_t<O>, iter_reference_t<I>>
    ~~tagged_pair<tag::in(I), tag::out(O)>~~
    uninitialized_copy_n_result<I, O>
      uninitialized_copy_n(I ifirst, iter_difference~~_type~~_t<I> n, O ofirst, S olast);
}

template<class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_move(InputIterator first, InputIterator last,
                                     ForwardIterator result);
template<class ExecutionPolicy, class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_move(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                                     InputIterator first, InputIterator last,
                                     ForwardIterator result);
template<class InputIterator, class Size, class ForwardIterator>
  pair<InputIterator, ForwardIterator> uninitialized_move_n(InputIterator first, Size n,
                                                            ForwardIterator result);
template<class ExecutionPolicy, class InputIterator, class Size, class ForwardIterator>
  pair<InputIterator, ForwardIterator> uninitialized_move_n(ExecutionPolicy&& exec, // see [algo-
rithms.parallel.overloads]
                                                            InputIterator first, Size n,
                                                            ForwardIterator result);

namespace ranges {
  template<class I, class O>
  using uninitialized_move_result = uninitialized_copy_result<I, O>;
  template<InputIterator I, Sentinel<I> S1, no-throw-forward-iterator O, no-throw-sentinel<O> S2>
      requires Constructible<iter_value~~_type~~_t<O>, iter_rvalue_reference_t<I>>
    ~~tagged_pair<tag::in(I), tag::out(O)>~~
    uninitialized_move_result<I, O>
      uninitialized_move(I ifirst, S1 ilast, O ofirst, S2 olast);
  template<InputRange IRng, no-throw-forward-range ORng>
      requires Constructible<iter_value~~_type~~_t<iterator_t<ORng>>, iter_rvalue_reference_t<iterator_t<IRng>>>
    ~~tagged_pair<tag::in(safe_iterator_t<IRng>), tag::out(safe_iterator_t<ORng>)>~~
    uninitialized_move_result<safe_iterator_t<IRng>, safe_iterator_t<ORng>>
      uninitialized_move(IRng&& irng, ORng&& orng);

  template<class I, class O>
  using uninitialized_move_n_result = uninitialized_copy_result<I, O>;
  template<InputIterator I, no-throw-forward-iterator O, no-throw-sentinel<O> S>
      requires Constructible<iter_value~~_type~~_t<O>, iter_rvalue_reference_t<I>>
    ~~tagged_pair<tag::in(I), tag::out(O)>~~
    uninitialized_move_n_result<I, O>
      uninitialized_move_n(I ifirst, iter_difference~~_type~~_t<I> n, O ofirst, S olast);
}

template<class ForwardIterator, class T>
  void uninitialized_fill(ForwardIterator first, ForwardIterator last, const T& x);
template<class ExecutionPolicy, class ForwardIterator, class T>
  void uninitialized_fill(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                          ForwardIterator first, ForwardIterator last, const T& x);
template<class ForwardIterator, class Size, class T>
  ForwardIterator uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
template<class ExecutionPolicy, class ForwardIterator, class Size, class T>
  ForwardIterator uninitialized_fill_n(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                                       ForwardIterator first, Size n, const T& x);

namespace ranges {
  template<no-throw-forward-iterator I, no-throw-sentinel<I> S, class T>
      requires Constructible<iter_value~~_type~~_t<I>, const T&>
    I uninitialized_fill(I first, S last, const T& x);
  template<no-throw-forward-range Rng, class T>
      requires Constructible<iter_value~~_type~~_t<iterator_t<Rng>>, const T&>
    safe_iterator_t<Rng> uninitialized_fill(Rng&& rng, const T& x);
```

```
    template<no-throw-forward-iterator I, class T>
        requires Constructible<iter_value_type_t<I>, const T&>
      I uninitialized_fill_n(I first, iter_difference_type_t<I> n, const T& x);
  }

  template<class T>
    void destroy_at(T* location);
  template<class ForwardIterator>
    void destroy(ForwardIterator first, ForwardIterator last);
  template<class ExecutionPolicy, class ForwardIterator>
    void destroy(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                 ForwardIterator first, ForwardIterator last);
  template<class ForwardIterator, class Size>
    ForwardIterator destroy_n(ForwardIterator first, Size n);
  template<class ExecutionPolicy, class ForwardIterator, class Size>
    ForwardIterator destroy_n(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                              ForwardIterator first, Size n);

  namespace ranges {
    template<Destructible T>
      void destroy_at(T* location) noexcept;

    template<no-throw-input-iterator I, no-throw-sentinel<I> S>
        requires Destructible<iter_value_type_t<I>>
      I destroy(I first, S last) noexcept;
    template<no-throw-input-range Rng>
        requires Destructible<iter_value_type_t<iterator_t<Rng>>
      safe_iterator_t<Rng> destroy(Rng&& rng) noexcept;

    template<no-throw-input-iterator I>
        requires Destructible<iter_value_type_t<I>>
      I destroy_n(I first, iter_difference_type_t<I> n) noexcept;
  }

  [...]
}
```

[...]

24.10.11 Specialized algorithms [specialized.algorithms]

1 ~~Throughout this subclause,~~ For the algorithms in this subclause defined directly in namespace `std`, the names of template parameters are used to express type requirements.

(1.1) — If an algorithm's template parameter is named `InputIterator`, the template argument shall satisfy the *Cpp17InputIterator* requirements (28.3.5.2).

(1.2) — If an algorithm's template parameter is named `ForwardIterator`, the template argument shall satisfy the *Cpp17ForwardIterator* requirements (28.3.5.4), and is required to have the property that no exceptions are thrown from increment, assignment, comparison, or indirection through valid iterators.

Unless otherwise specified, if an exception is thrown in the following algorithms there are no effects.

[Editor's note: The following paragraphs incorporate various paragraphs from [algorithms.requirements]. We should consider simply relocating this entire subclause under [algorithms]:]

2 In the description of the algorithms operators `+` and `-` are used for some of the iterator categories for which they do not have to be defined. In these cases the semantics of `a+n` is the same as that of

```
X tmp = a;
advance(tmp, n);
return tmp;
```

and that of `b-a` is the same as of

```
return distance(a, b);
```

3 For the algorithms in this subclause defined in namespace `std::ranges`, the following additional requirements apply:

(3.1) — The function templates defined in the `std::ranges` namespace in this subclause are not found by argument-dependent name lookup ([basic.lookup.argdep]). When found by unqualified ([basic.lookup.unqual]) name lookup for the *postfix-expression* in a function call ([expr.call]), they inhibit argument-dependent name lookup.

(3.2) — Overloads of algorithms that take `Range` arguments (29.6.2) behave as if they are implemented by calling `ranges::begin` and `ranges::end` on the `Range`(s) and dispatching to the overload that takes separate iterator and sentinel arguments.

(3.3) — The number and order of template parameters for algorithm declarations is unspecified, except where explicitly stated otherwise.

4 [*Note:* Invocation of the algorithms specified in this subclause (24.10.11)) ~~shall only operate on ranges of complete objects ([intro.object]). Use of these functions~~ on ranges of potentially overlapping subobjects ([intro.object]) ~~is~~ results in undefined behavior. — *end note*]

24.10.11.1 Special memory concepts [special.mem.concepts]

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

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

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

3 [*Note:* The distinction between `InputIterator` and *`no-throw-input-iterator`* is purely semantic. — *end note*]

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

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

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

```
template<class Rng>
concept no-throw-input-range = // exposition only
  Range<Rng> &&
  no-throw-input-iterator<iterator_t<Rng>> &&
  no-throw-sentinel<sentinel_t<Rng>, iterator_t<Rng>>;
```

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

7 [*Note:* The distinction between `InputRange` and *`no-throw-input-range`* is purely semantic. — *end note*]

```
template<class I>
concept no-throw-forward-iterator = // exposition only
  no-throw-input-iterator<I> &&
  ForwardIterator<I> &&
  no-throw-sentinel<I, I>;
```

8 [*Note:* The distinction between `ForwardIterator` and *`no-throw-forward-iterator`* is purely semantic. — *end note*]

```
template<class Rng>
concept no-throw-forward-range = // exposition only
  no-throw-input-range<Rng> &&
  no-throw-forward-iterator<iterator_t<Rng>>;
```

9 [*Note:* The distinction between `ForwardRange` and *`no-throw-forward-range`* is purely semantic. — *end note*]

24.10.11.2 `addressof` [specialized.addressof]

[...]

24.10.11.3 `uninitialized_default_construct` [uninitialized.construct.default]

```
template<class ForwardIterator>
  void uninitialized_default_construct(ForwardIterator first, ForwardIterator last);
```

1 *Effects:* Equivalent to:

```
for (; first != last; ++first)
  ::new (static_cast<void*>(addressof(*first)))
    typename iterator_traits<ForwardIterator>::value_type;
```

```
namespace ranges {
  template<no-throw-forward-iterator I, no-throw-sentinel<I> S>
      requires DefaultConstructible<iter_value~~_type~~_t<I>>
    I uninitialized_default_construct(I first, S last);
  template<no-throw-forward-range Rng>
      requires DefaultConstructible<iter_value~~_type~~_t<iterator_t<Rng>>>
    safe_iterator_t<Rng> uninitialized_default_construct(Rng&& rng);
}
```

2 *Effects:* Equivalent to:

```
for (; first != last; ++first)
  ::new (const_cast<void*>(static_cast<const volatile void*>(addressof(*first))))
    remove_reference_t<iter_reference_t<I>>;
return first;
```

[Editor's note: `const_cast<void*>` is necessary to ensure that `::operator new<void*>` ('True Placement New') is called. The decision to cast `const`-ness away after calling `addressof` is an alternative to preventing users from being unable to pass ranges that are non-`const`.

When `addressof(*i)` returns a `const T*`, this will not convert to `void*`, and so no suitable overload of the True Placement New is found.

It is noted that `const`-qualified means 'do not modify', and that the `const_cast` ignores this (and is thus lying). However, these algorithms are also claiming that they iterate over objects of type `T`: nary a `T` is in the range. We present this as 'the objects in this range should be `const`', rather than 'the memory here is `const`'.]

```
template<class ForwardIterator, class Size>
  ForwardIterator uninitialized_default_construct_n(ForwardIterator first, Size n);
```

3 *Effects:* Equivalent to:

```
for (; n > 0; (void)++first, --n)
  ::new (static_cast<void*>(addressof(*first)))
    typename iterator_traits<ForwardIterator>::value_type;
return first;
```

```
namespace ranges {
  template<no-throw-forward-iterator I>
      requires DefaultConstructible<iter_value~~_type~~_t<I>>
    I uninitialized_default_construct_n(I first, iter_difference~~_type~~_t<I> n);
}
```

4 *Effects:* Equivalent to:

```
return uninitialized_default_construct(make_counted_iterator(first, n),
                                       default_sentinel{}).base();
```

24.10.11.4 `uninitialized_value_construct` [uninitialized.construct.value]

```
template<class ForwardIterator>
  void uninitialized_value_construct(ForwardIterator first, ForwardIterator last);
```

1 *Effects:* Equivalent to:

```
for (; first != last; ++first)
  ::new (static_cast<void*>(addressof(*first)))
    typename iterator_traits<ForwardIterator>::value_type();
```

```
namespace ranges {
  template<no-throw-forward-iterator I, no-throw-sentinel<I> S>
      requires DefaultConstructible<iter_value~~_type~~_t<I>>
    I uninitialized_value_construct(I first, S last);
  template<no-throw-forward-range Rng>
      requires DefaultConstructible<iter_value~~_type~~_t<iterator_t<Rng>>>
    safe_iterator_t<Rng> uninitialized_value_construct(Rng&& rng);
}
```

2 *Effects:* Equivalent to:

```
for (; first != last; ++first)
  ::new (const_cast<void*>(static_cast<const volatile void*>(addressof(*first))))
    remove_reference_t<iter_reference_t<I>>();
return first;
```

```
template<class ForwardIterator, class Size>
  ForwardIterator uninitialized_value_construct_n(ForwardIterator first, Size n);
```

3 *Effects:* Equivalent to:

```
for (; n > 0; (void)++first, --n)
  ::new (static_cast<void*>(addressof(*first)))
    typename iterator_traits<ForwardIterator>::value_type();
return first;
```

```
namespace ranges {
  template<no-throw-forward-iterator I>
      requires DefaultConstructible<iter_value~~_type~~_t<I>>
    I uninitialized_value_construct_n(I first, iter_difference~~_type~~_t<I> n);
}
```

4 *Effects:* Equivalent to:

```
return uninitialized_value_construct(make_counted_iterator(first, n),
                                     default_sentinel{}).base();
```

24.10.11.5 `uninitialized_copy` [uninitialized.copy]

```
template<class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_copy(InputIterator first, InputIterator last,
                                     ForwardIterator result);
```

1 *Requires:* `[result, result + (last - first))` shall not overlap with `[first, last)`.

2 *Effects:* As if by:

```
for (; first != last; ++result, (void) ++first)
  ::new (static_cast<void*>(addressof(*result)))
    typename iterator_traits<ForwardIterator>::value_type(*first);
```

3 *Returns:* `result`.

```
namespace ranges {
  template<InputIterator I, Sentinel<I> S1, no-throw-forward-iterator O, no-throw-sentinel<O> S2>
      requires Constructible<iter_value~~_type~~_t<O>, iter_reference_t<I>>
    ~~tagged_pair<tag::in(I), tag::out(O)>~~
    uninitialized_copy_result<I, O>
      uninitialized_copy(I ifirst, S1 ilast, O ofirst, S2 olast);
  template<InputRange IRng, no-throw-forward-range ORng>
      requires Constructible<iter_value~~_type~~_t<iterator_t<ORng>>, iter_reference_t<iterator_t<IRng>>>
    ~~tagged_pair<tag::in(safe_iterator_t<IRng>), tag::out(safe_iterator_t<ORng>)>~~
    uninitialized_copy_result<iterator_t<IRng>, iterator_t<ORng>
      uninitialized_copy(IRng&& irng, ORng&& orng);
}
```

4 *Requires:* `[ofirst, olast)` shall not overlap with `[ifirst, ilast)`.

5 *Effects:* Equivalent to:

```
    for (; ifirst != ilast && ofirst != olast; ++ofirst, (void)++ifirst) {
      ::new (const_cast<void*>(static_cast<const volatile void*>(addressof(*ofirst))))
        remove_reference_t<iter_reference_t<O>>(*ifirst);
    }
    return {ifirst, ofirst};
```

```
template<class InputIterator, class Size, class ForwardIterator>
  ForwardIterator uninitialized_copy_n(InputIterator first, Size n, ForwardIterator result);
```

6 *Requires:* `[result, result + n)` shall not overlap with `[first, first + n)`.

7 *Effects:* As if by:

```
    for ( ; n > 0; ++result, (void) ++first, --n) {
      ::new (static_cast<void*>(addressof(*result)))
        typename iterator_traits<ForwardIterator>::value_type(*first);
    }
```

8 *Returns:* `result`.

```
namespace ranges {
  template<InputIterator I, no-throw-forward-iterator O, no-throw-sentinel<O> S>
      requires Constructible<iter_value_type_t<O>, iter_reference_t<I>>
    tagged_pair<tag::in(I), tag::out(O)>
    uninitialized_copy_n_result<I, O>
      uninitialized_copy_n(I ifirst, iter_difference_type_t<I> n, O ofirst, S olast);
}
```

9 *Requires:* `[ofirst, olast)` shall not overlap with `[ifirst, ifirst + n` ~~`next(ifirst, n)`~~`)`.

10 *Effects:* Equivalent to:

```
    auto t = uninitialized_copy(make_counted_iterator(ifirst, n),
                                default_sentinel{}, ofirst, olast).base();
    return {t.in().base(), t.out()};
```

24.10.11.6 `uninitialized_move` [uninitialized.move]

```
template<class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_move(InputIterator first, InputIterator last,
                                     ForwardIterator result);
```

1 *Requires:* `[result, result + (last - first))` shall not overlap with `[first, last)`.

2 *Effects:* Equivalent to:

```
    for (; first != last; (void)++result, ++first)
      ::new (static_cast<void*>(addressof(*result)))
        typename iterator_traits<ForwardIterator>::value_type(std::move(*first));
    return result;
```

3 *Remarks:* If an exception is thrown, some objects in the range `[first, last)` are left in a valid but unspecified state.

```
namespace ranges {
  template<InputIterator I, Sentinel<I> S1, no-throw-forward-iterator O, no-throw-sentinel<O> S2>
      requires Constructible<iter_value_type_t<O>, iter_rvalue_reference_t<I>>
    tagged_pair<tag::in(I), tag::out(O)>
    uninitialized_move_result<I, O>
      uninitialized_move(I ifirst, S1 ilast, O ofirst, S2 olast);
  template<InputRange IRng, no-throw-forward-range ORng>
      requires Constructible<iter_value_type_t<iterator_t<ORng>>, iter_rvalue_reference_t<iterator_t<IRng>>>
    tagged_pair<tag::in(safe_iterator_t<IRng>), tag::out(safe_iterator_t<ORng>)>
    uninitialized_move_result<safe_iterator_t<IRng>, safe_iterator_t<ORng>>
      uninitialized_move(IRng&& irng, ORng&& orng);
}
```

4 *Requires:* `[ofirst, olast)` shall not overlap with `[ifirst, ilast)`.

5 *Effects:* Equivalent to:

```
    for (; ifirst != ilast && ofirst != olast; ++ofirst, (void)++ifirst) {
      ::new (const_cast<void*>(static_cast<const volatile void*>(addressof(*ofirst))))
        remove_reference_t<iter_reference_t<O>>(ranges::iter_move(ifirst));
    }
    return {ifirst, ofirst};
```

6 [*Note:* If an exception is thrown, some objects in the range `[first, last)` are left in a valid, but unspecified state. — *end note*]

```
template<class InputIterator, class Size, class ForwardIterator>
  pair<InputIterator, ForwardIterator>
    uninitialized_move_n(InputIterator first, Size n, ForwardIterator result);
```

7 *Requires:* `[result, result + n)` shall not overlap with `[first, first + n)`.

8 *Effects:* Equivalent to:

```
    for (; n > 0; ++result, (void) ++first, --n)
      ::new (static_cast<void*>(addressof(*result)))
        typename iterator_traits<ForwardIterator>::value_type(std::move(*first));
    return {first,result};
```

9 *Remarks:* If an exception is thrown, some objects in the range `[first, first + n ~~std::next(first,n)~~)` are left in a valid but unspecified state.

```
namespace ranges {
  template<InputIterator I, no-throw-forward-iterator O, no-throw-sentinel<O> S>
      requires Constructible<iter_value~~_type~~_t<O>, iter_rvalue_reference_t<I>>
    ~~tagged_pair<tag::in(I), tag::out(O)>~~
    uninitialized_move_n_result<I, O>
      uninitialized_move_n(I ifirst, iter_difference~~_type~~_t<I> n, O ofirst, S olast);
}
```

10 *Requires:* `[ofirst, olast)` shall not overlap with `[ifirst, ifirst + n ~~next(ifirst, n)~~)`.

11 *Effects:* Equivalent to:

```
    auto t = uninitialized_move(make_counted_iterator(ifirst, n),
                                default_sentinel{}, ofirst, olast)~~.base()~~;
    return {t.in().base(), t.out()};
```

12 [*Note:* If an exception is thrown, some objects in the range `[first, first + n ~~next(first, n)~~)` are left in a valid but unspecified state. — *end note*]

24.10.11.7 `uninitialized_fill` [uninitialized.fill]

```
template<class ForwardIterator, class T>
  void uninitialized_fill(ForwardIterator first, ForwardIterator last, const T& x);
```

1 *Effects:* As if by:

```
    for (; first != last; ++first)
      ::new (static_cast<void*>(addressof(*first)))
        typename iterator_traits<ForwardIterator>::value_type(x);
```

```
namespace ranges {
  template<no-throw-forward-iterator I, no-throw-sentinel<I> S, class T>
      requires Constructible<iter_value~~_type~~_t<I>, const T&>
    I uninitialized_fill(I first, S last, const T& x);
  template<no-throw-forward-range Rng, class T>
      requires Constructible<iter_value~~_type~~_t<iterator_t<Rng>>, const T&>
    safe_iterator_t<Rng> uninitialized_fill(Rng&& rng, const T& x);
}
```

Effects: Equivalent to:

```
    for (; first != last; ++first) {
      ::new (const_cast<void*>(static_cast<const volatile void*>(addressof(*first))))
        remove_reference_t<iter_reference<I>>(x);
    }
```

return first;

```
template<class ForwardIterator, class Size, class T>
  ForwardIterator uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
```

2 *Effects:* As if by:

```
for (; n--; ++first)
  ::new (static_cast<void*>(addressof(*first)))
    typename iterator_traits<ForwardIterator>::value_type(x);
return first;
```

```
namespace ranges {
  template<no-throw-forward-iterator I, class T>
      requires Constructible<iter_value~~_type~~_t<I>, const T&>
    I uninitialized_fill_n(I first, iter_difference~~_type~~_t<I> n, const T& x);
}
```

3 *Effects:* Equivalent to:

```
return uninitialized_fill(make_counted_iterator(first, n), default_sentinel{}, x).base();
```

24.10.11.8 destroy [specialized.destroy]

```
template<class T>
  void destroy_at(T* location);
```

```
namespace ranges {
  template<Destructible T>
    void destroy_at(T* location) noexcept;
}
```

1 *Effects:* Equivalent to:

```
location->~T();
```

```
template<class ForwardIterator>
  void destroy(ForwardIterator first, ForwardIterator last);
```

2 *Effects:* Equivalent to:

```
for (; first!=last; ++first)
  destroy_at(addressof(*first));
```

```
namespace ranges {
  template<no-throw-input-iterator I, no-throw-sentinel<I> S>
      requires Destructible<iter_value~~_type~~_t<I>>
    I destroy(I first, S last) noexcept;
  template<no-throw-input-range Rng>
      requires Destructible<iter_value~~_type~~_t<iterator_t<Rng>>
    safe_iterator_t<Rng> destroy(Rng&& rng) noexcept;
}
```

3 *Effects:* Equivalent to:

```
for (; first != last; ++first)
  destroy_at(addressof(*first));
return first;
```

[Editor's note: The International Standard requires `destroy` be a `ForwardIterator` to ensure that the iterator's `reference` type is a reference type. This requirement is relaxed for `ranges::destroy`, since *`no-throw-input-iterator`* requires that `iter_reference_t<I>` is an lvalue reference type.

The choice to weaken the iterator requirement from the International Standard is because the algorithm is a single-pass algorithm; thus, semantically, works on input ranges.]

```
template<class ForwardIterator, class Size>
  ForwardIterator destroy_n(ForwardIterator first, Size n);
```

4 *Effects:* Equivalent to:

```
for (; n > 0; (void)++first, --n)
  destroy_at(addressof(*first));
return first;
```

```
namespace ranges {
  template<no-throw-input-iterator I>
      requires Destructible<iter_value_type_t<I>>
    I destroy_n(I first, iter_difference_type_t<I> n) noexcept;
}
```

5 *Effects:* Equivalent to:

```
return destroy(make_counted_iterator(first, n), default_sentinel{}).base();
```

[...]

24.14 Function Objects [function.objects]

24.14.1 Header <functional> synopsis [functional.syn]

[Editor's note: Add declarations to `<functional>`:]

```
[...]

template<class T>
  inline constexpr bool is_bind_expression_v = is_bind_expression<T>::value;
template<class T>
  inline constexpr int is_placeholder_v = is_placeholder<T>::value;

namespace ranges {
  // 24.14.8, comparisons
  template<class T = void>
    requires see below
  struct equal_to;

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

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

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

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

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

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

[Editor's note: Add new subclause [range.comparisons] between [comparisons] and [logical.operations]:]

24.14.8 Comparisons (ranges) [range.comparisons]

1 ~~The library provides basic function object classes for all of the comparison operators in the language ([expr.rel], [expr.eq]).~~

2 In this section, *BUILTIN_PTR_CMP*(T, *op*, U) for types T and U and where *op* is an equality ([expr.eq]) or relational operator ([expr.rel]) is a boolean constant expression. *BUILTIN_PTR_CMP*(T, *op*, U) is `true` if and only if *op* in the expression `declval<T>()` *op* `declval<U>()` resolves to a built-in operator comparing pointers.

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

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

4 `operator()` has effects equivalent to: `return ranges::equal_to<>(x, y);`

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

5 `operator()` has effects equivalent to: `return !ranges::equal_to<>(x, y);`

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

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

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

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

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

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

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

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

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

  using is_transparent = unspecified;
};
```

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

11 *Effects:*

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

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

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

  using is_transparent = unspecified;
};
```

12 `operator()` has effects equivalent to:

```
return !ranges::equal_to<>{}(std::forward<T>(t), std::forward<U>(u));
```

```
template<> struct greater<void> {
  template<class T, class U>
    requires StrictTotallyOrderedWith<T, U> || BUILTIN_PTR_CMP(U, <, T)
  constexpr bool operator()(T&& t, U&& u) const;

  using is_transparent = unspecified;
};
```

13 `operator()` has effects equivalent to:

```
return ranges::less<>{}(std::forward<U>(u), std::forward<T>(t));
```

```
template<> struct less<void> {
  template<class T, class U>
    requires StrictTotallyOrderedWith<T, U> || BUILTIN_PTR_CMP(T, <, U)
  constexpr bool operator()(T&& t, U&& u) const;

  using is_transparent = unspecified;
};
```

14 *Requires:* If the expression `std::forward<T>(t) < std::forward<U>(u)` results in a call to a built-in operator `<` comparing pointers of type `P`, the conversion sequences from both `T` and `U` to `P` shall be equality-preserving ([concepts.general.equality]). For any expressions `ET` and `EU` such that `decltype((ET))` is `T` and `decltype((EU))` is `U`, exactly one of `ranges::less<>{}(ET, EU)`, `ranges::less<>{}(EU, ET)` or `ranges::equal_to<>{}(ET, EU)` shall be `true`.

15 *Effects:*

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

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

```
template<> struct greater_equal<void> {
  template<class T, class U>
    requires StrictTotallyOrderedWith<T, U> || BUILTIN_PTR_CMP(T, <, U)
  constexpr bool operator()(T&& t, U&& u) const;

  using is_transparent = unspecified;
};
```

16 `operator()` has effects equivalent to:

```
return !ranges::less<>{}(std::forward<T>(t), std::forward<U>(u));
```

```
template<> struct less_equal<void> {
  template<class T, class U>
    requires StrictTotallyOrderedWith<T, U> || BUILTIN_PTR_CMP(U, <, T)
  constexpr bool operator()(T&& t, U&& u) const;
```

```
  using is_transparent = unspecified;
};
```

17 `operator()` has effects equivalent to:

```
return !ranges::less<>{}(std::forward<U>(u), std::forward<T>(t));
```

25 Strings library [strings]

25.4 String view classes [string.view]

25.4.2 Class template `basic_string_view` [string.view.template]

```
template<class charT, class traits = char_traits<charT>>
class basic_string_view {
public:
  [...]

  // 25.4.2.2, iterator support
  constexpr const_iterator begin() const noexcept;
  constexpr const_iterator end() const noexcept;
  constexpr const_iterator cbegin() const noexcept;
  constexpr const_iterator cend() const noexcept;
  constexpr const_reverse_iterator rbegin() const noexcept;
  constexpr const_reverse_iterator rend() const noexcept;
  constexpr const_reverse_iterator crbegin() const noexcept;
  constexpr const_reverse_iterator crend() const noexcept;

  friend constexpr const_iterator begin(basic_string_view sv) noexcept { return sv.begin(); }
  friend constexpr const_iterator end(basic_string_view sv) noexcept { return sv.end(); }

  // [string.view.capacity], capacity
  constexpr size_type size() const noexcept;
  constexpr size_type length() const noexcept;
  constexpr size_type max_size() const noexcept;
  [[nodiscard]] constexpr bool empty() const noexcept;

  [...]
};
```

[Editor's note: In the paragraph that specifies `const_iterator`, cross-reference "contiguous iterator" to [iterator.concept.contiguous] instead of [iterator.requirements.general]. (Is this too subtle a means to require implementations to specialize `ranges::iterator_category`?)]

25.4.2.2 Iterator support [string.view.iterators]

`using const_iterator = implementation-defined;`

1 A type that meets the requirements of a constant random access iterator (28.3.5.6) and of a contiguous iterator (28.3.4.13) whose `value_type` is the template parameter `charT`.

2 [...]

27 Containers library [containers]

27.2 Container requirements [container.requirements]

27.2.1 General container requirements [container.requirements.general]

[Editor's note: In the paragraph that defines "contiguous container", cross-reference "contiguous iterator" to [iterator.concept.contiguous] instead of [iterator.requirements.general]. (Is this too subtle a means to require implementations to specialize `ranges::iterator_category` for the iterators of contiguous containers?)]

13 A *contiguous container* is a container that supports random access iterators (28.3.5.6) and whose member types `iterator` and `const_iterator` are contiguous iterators (28.3.4.13).

27.7.3 Class template span [views.span]

27.7.3.1 Overview [span.overview]

[Editor's note: In the paragraph that defines `span`'s iterator, cross-reference "contiguous iterator" to [iterator.concept.contiguous]. (Is this too subtle a means to require implementations to specialize `ranges::iterator_category` for `span`'s iterators?)]

4 The iterator type for `span` is a random access iterator and a contiguous iterator (28.3.4.13).

5 All member functions of `span` have constant time complexity.

```
namespace std {
  template<class ElementType, ptrdiff_t Extent = dynamic_extent>
  class span {
  public:
    [...]

    // [span.iterators], iterator support
    constexpr iterator begin() const noexcept;
    constexpr iterator end() const noexcept;
    constexpr const_iterator cbegin() const noexcept;
    constexpr const_iterator cend() const noexcept;
    constexpr reverse_iterator rbegin() const noexcept;
    constexpr reverse_iterator rend() const noexcept;
    constexpr const_reverse_iterator crbegin() const noexcept;
    constexpr const_reverse_iterator crend() const noexcept;

    friend constexpr iterator begin(span&& s) noexcept { return s.begin(); }
    friend constexpr iterator end(span&& s) noexcept { return s.end(); }
    friend constexpr iterator begin(const span&& s) noexcept { return s.begin(); }
    friend constexpr iterator end(const span&& s) noexcept { return s.end(); }

    [...]
  };
}
```

28 Iterators library [iterators]

28.1 General [iterators.general]

1 This Clause describes components that C++ programs may use to perform iterations over containers (Clause 27), streams ([iostream.format]), ~~and~~ stream buffers ([stream.buffers]), and other ranges (Clause 29).

2 The following subclauses describe iterator requirements, and components for iterator primitives, predefined iterators, and stream iterators, as summarized in Table 87.

Table 87 — Iterators library summary

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

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

28.2 Header <iterator> synopsis [iterator.synopsis]

```
#include <concepts>
```

```
namespace std {
  template<class T> concept bool dereferenceable // exposition only
    = requires(T& t) { { *t } -> auto&&; }; // not required to be equality-preserving

  // 28.3.2, associated types
  // 28.3.2.1, difference_typeincrementable traits
  template<class> struct difference_typeincrementable_traits;
  template<class T>
    using iter_difference_type_t = see below;
      = typename difference_type<T>::type;

  // 28.3.2.2, value_typereadable traits
  template<class> struct value_typereadable_traits;
  template<class T>
    using value_type_titer_value_t = see below;
      = typename value_type<T>::type;

  // [iterator.assoc.types.iterator_category], iterator_category:
  template<class> struct iterator_category;
  template<class T> using iterator_category_t
    = typename iterator_category<T>::type;
  // 28.3.2.3, Iterator traits [Editor's note: Moved here from below.]
  template<class Iterator> struct iterator_traits;
  template<class T> struct iterator_traits<T*>;

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

  namespace ranges {
    // 28.3.3, customization points
    inline namespace unspecified {
      // 28.3.3.1, iter_move
      inline constexpr unspecified iter_move = unspecified;

      // 28.3.3.2, iter_swap
      inline constexpr unspecified iter_swap = unspecified;
    }
  }

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

  // 28.3.4, iterator requirementsconcepts
  // 28.3.4.1, Readable
  template<class In>
  concept bool Readable = see below;

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

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

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

```
// 28.3.4.4, Incrementable
template<class I>
concept ~~bool~~ Incrementable = see below;

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

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

// 28.3.4.7, SizedSentinel
template<class S, class I>
inline constexpr bool disable_sized_sentinel = false;

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

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

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

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

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

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

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

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

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

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

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

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

template<class, class...>
struct indirect_result~~_of~~ { };
```

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

template<class F, class... Is>
using indirect_result~~_of~~_t
  = typename indirect_result~~_of~~<F, Is...>::type;

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

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

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

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

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

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

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

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

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

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

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

[Editor's note: ranges::iterator_traits from the Ranges TS is intentionally omitted.]

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

[Editor's note: The iterator tags from the Ranges TS are intentionally omitted.]

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

// 28.4.3, range iterator operations
namespace ranges {
  // 28.4.3, Range iterator operations
  namespace {
    constexpr unspecified advance = unspecified;
    constexpr unspecified distance = unspecified;
    constexpr unspecified next = unspecified;
    constexpr unspecified prev = unspecified;
  }
  // 28.4.3.1, 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);

  // 28.4.3.2, 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);

  // 28.4.3.3, 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);

  // 28.4.3.4, 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);
}

// 28.5, predefined iterators and sentinels
```

[Editor's note: reverse_iterator, move_iterator, and the insert iterators from the Ranges TS are intentionally omitted.]

```
// 28.5.1, 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);

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

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

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

// 28.5.3, 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;

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

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

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

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

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

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

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

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

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

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

// 28.5.5, default sentinels
class default_sentinel;

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

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

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

template<class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator==(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
  constexpr bool operator==(
    const counted_iterator<auto>& x, default_sentinel);
  constexpr bool operator==(
    default_sentinel, const counted_iterator<auto>& x);
template<class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator!=(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
  constexpr bool operator!=(
    const counted_iterator<auto>& x, default_sentinel y);
  constexpr bool operator!=(
    default_sentinel x, const counted_iterator<auto>& y);
template<class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator<(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template<class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator<=(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template<class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator>(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template<class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator>=(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template<class I1, class I2>
    requires Common<I1, I2>
  constexpr difference_type_t<I2> operator-(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template<class I>
  constexpr difference_type_t<I> operator-(
    const counted_iterator<I>& x, default_sentinel y);
template<class I>
  constexpr difference_type_t<I> operator-(
    default_sentinel x, const counted_iterator<I>& y);
template<RandomAccessIterator I>
  constexpr counted_iterator<I>
    operator+(difference_type_t<I> n, const counted_iterator<I>& x);
```

```
~~template<Iterator I>~~
~~  constexpr counted_iterator<I> make_counted_iterator(I i, difference_type_t<I> n);~~

// 28.5.7, unreachable sentinels
class unreachable;

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

// 28.6, stream iterators
[Editor's note:  The stream iterators from the Ranges TS are intentionally omitted.]
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;

// [iterator.range], 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));
```

```
  // [iterator.container], 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;
}
```

28.3 Iterator requirements [iterator.requirements]

28.3.1 In general [iterator.requirements.general]

[Editor's note: Merge the changes from the Ranges TS [iterator.requirements.general] as follows:]

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

2 Since iterators are an abstraction of pointers, their semantics is a generalization of most of the semantics of pointers in C++. This ensures that every function template that takes iterators works as well with regular pointers. This document defines ~~five~~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 88.

Table 88 — Relations among iterator categories

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

3 The ~~five~~six categories of iterators correspond to the iterator concepts `InputIterator` (28.3.4.8), `OutputIterator` (28.3.4.9), `ForwardIterator` (28.3.4.10), `BidirectionalIterator` (28.3.4.11) `RandomAccessIterator` (28.3.4.12), and `ContiguousIterator` (28.3.4.13), respectively. The generic term *iterator* refers to any type that satisfies the `Iterator` concept (28.3.4.5).

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-names* specified in 28.3.2.3 shall be provided for the iterator type. [*Note:* Either the iterator type must provide the *typedef-names* directly (in which case `iterator_traits` picks them up automatically), or an `iterator_traits` specialization must provide them. — *end note*]

7 Iterators that further satisfy the requirement that, for integral values `n` and dereferenceable iterator values `a` and `(a + n)`, `*(a + n)` is equivalent to `*(addressof(*a) + n)`, are called *contiguous iterators*. [*Note:* For example, the type "pointer to `int`" is a contiguous iterator, but `reverse_iterator<int *>` is not. For a valid iterator range [`a`,`b`) with dereferenceable `a`, the corresponding range denoted by pointers is [`addressof(*a)`,`addressof(*a) + (b - a)`); `b` might not be dereferenceable. — *end note*]

8 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.

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

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

11 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.[3]

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

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

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

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

16 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 ~~have a~~ specify complexity ~~column~~.

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

18 An *invalid* iterator is an iterator that may be singular.[4]

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

(19.1) — `swap`,

(19.2) — a pseudo-destructor call ([expr.pseudo]), and

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

3) 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.

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

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

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

[Editor's note: Relocate [iterator.assoc.types] from the Ranges TS here, and modify as follows:]

28.3.2 Associated types [iterator.assoc.types]

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

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

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

[Editor's note: Change the stable name of Ranges [iterator.assoc.types.difference_type] to [incrementable.traits] and modify as follows:]

28.3.2.1 Incrementable traits [incrementable.traits]

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

```
iter_difference_t<WI>
```

be defined as the incrementable type's difference type.

2 `iter_difference~~_type~~_t` is implemented as if:

```
namespace std::ranges {
  template<class> struct ~~difference_type~~incrementable_traits { };

  ~~template<class T>~~
  ~~struct difference_type<T*>~~
    ~~: enable_if<is_object<T>::value, ptrdiff_t> { };~~

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

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

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

  template<class T>
    requires !requires { typename T::difference_type; } &&
      requires(const T& a, const T& b) { { a - b } -> Integral; }
```

```
struct difference_typeincrementable_traits<T>
  : make_signed<decltype(declval<T>() - declval<T>())> {
  using difference_type = make_signed_t<decltype(declval<T>() - declval<T>())>;
};

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

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

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

[Editor's note: Change the stable name of Ranges TS [iterator.assoc.types.value_type] to [readable.traits] and modify as follows:]

28.3.2.2 Readable traits [readable.traits]

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

2 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 (28.3.4.1), the type

```
iter_value_t<R>
```

be defined as the readable type's value type.

3 `iter_value_t` is implemented as if:

```
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 value_typereadable_traits { };

template<class T>
struct value_typereadable_traits<T*>
  : enable_if<is_object<T>::value, remove_cv_t<T>>
    cond-value-type<T> { };

template<class I>
  requires is_array_v<I>::value
struct value_typereadable_traits<I>
  : value_typereadable_traits<decay_t<I>> { };

template<class I>
struct value_typereadable_traits<const I>
  : value_typereadable_traits<decayremove_const_t<I>> { };

template<class T>
  requires requires { typename T::value_type; }
struct value_typereadable_traits<T>
  : enable_if<is_object<typename T::value_type>::value, typename T::value_type>
    cond-value-type<typename T::value_type> { };

template<class T>
  requires requires { typename T::element_type; }
struct value_typereadable_traits<T>
  : enable_if<
      is_object<typename T::element_type>::value,
      remove_cv_t<typename T::element_type>>
```

`cond-value-type<typename T::element_type> { };`

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

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

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

6 Class template ~~`value_type`~~`readable_traits` may be specialized on user-defined types.

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

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

[Editor's note: Subclauses [iterator.assoc.types.iterator_category], [iterator.traits], [iterator.stdtraits], and [std.iterator.tags] from the Ranges TS are intentionally ommitted.]

[Editor's note: Relocate the working draft's [iterator.traits] from [iterator.primitives] to here and change it as follows:]

28.3.2.3 Iterator traits [iterator.traits]

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

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

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

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

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

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

may be defined as `void`.

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

```
template<class I>
concept _Cpp17Iterator =
  Copyable<I> && requires (I i) {
    { *i } -> auto &&;
    { ++i } -> Same<I>&;
    { *i++ } -> auto &&;
  };
```

```
template<class I>
concept _Cpp17InputIterator =
  _Cpp17Iterator<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 &>;
    typename common_reference_t<decltype(*i++) &&,
                                typename readable_traits<I>::value_type &>;
    requires SignedIntegral<typename incrementable_traits<I>::difference_type>;
  };

template<class I>
concept _Cpp17ForwardIterator =
  _Cpp17InputIterator<I> && Constructible<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 _Cpp17BidirectionalIterator =
  _Cpp17ForwardIterator<I> && requires (I i) {
    { --i } -> Same<I>&;
    { i-- } -> const I&;
    requires Same<iter_reference_t<I>, decltype(*i--)>;
  };

template<class I>
concept _Cpp17RandomAccessIterator =
  _Cpp17BidirectionalIterator<I> && StrictTotallyOrdered<I> &&
  requires (I i, typename incrementable_traits<I>::difference_type n) {
    { i += n } -> Same<I>&;
    { i -= n } -> Same<I>&;
    requires Same<I, decltype(i + n)>;
    requires Same<I, decltype(n + i)>;
    requires Same<I, decltype(i - n)>;
    requires Same<decltype(n), decltype(i - i)>;
    { i[n] } -> iter_reference_t<I>;
  };
```

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

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

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

(2.2) — Otherwise, if `I` satisfies the exposition-only concept *`_Cpp17InputIterator`*, `iterator_traits<I>` shall have the following as publicly accessible members:

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

If `I::pointer` is well-formed and names a type, `pointer` names that type. Otherwise, if `decltype( declval<I&>().operator->())` is well-formed, then `pointer` names that type. ~~Otherwise, if `iter_reference_t<I>`~~

~~is an lvalue reference type, `pointer` is `add_pointer_t<iter_reference_t<I>>`.~~ Otherwise, `pointer` is `void`.

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

If `I::iterator_category` is well-formed and names a type, `iterator_category` names that type. Otherwise, if `I` satisfies *`_Cpp17RandomAccessIterator`*, `iterator_category` is `random_access_iterator_tag`. Otherwise, if `I` satisfies *`_Cpp17BidirectionalIterator`*, `iterator_category` is `bidirectional_iterator_tag`. Otherwise, if `I` satisfies *`_Cpp17ForwardIterator`*, `iterator_category` is `forward_iterator_tag`. Otherwise, `iterator_category` is `input_iterator_tag`.

(2.3) — Otherwise, if `I` satisfies the exposition-only concept *`_Cpp17Iterator`*, `iterator_traits<I>` shall have the following as publicly accessible members:

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

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

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

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

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

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

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

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

— *end example*]

[Editor's note: Relocate [iterator.custpoints] here from the Ranges TS and modify as follows:]

28.3.3 Customization points [iterator.custpoints]

28.3.3.1 `iter_move` [iterator.custpoints.iter_move]

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

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

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

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

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

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

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

28.3.3.2 `iter_swap` [iterator.custpoints.iter_swap]

1 The name `iter_swap` denotes a *customization point object* ([customization.point.object]). The expression `ranges::iter_swap(E1, E2)` for some subexpressions `E1` and `E2` is expression-equivalent to the following:

(1.1) — `(void)iter_swap(E1, E2)`, if that expression is well-formed when evaluated in a context that does not include `ranges::iter_swap` but does include the lookup set produced by argument-dependent lookup ([basic.lookup.argdep]) and the following declaration:

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

(1.2) — Otherwise, if the types of `E1` and `E2` both ~~satisfy~~model `Readable`, and if the reference type of `E1` is swappable with (22.3.11) the reference type of `E2`, then `ranges::swap(*E1, *E2)`

(1.3) — Otherwise, if the types `T1` and `T2` of `E1` and `E2` ~~satisfy~~model `IndirectlyMovableStorable<T1, T2> && IndirectlyMovableStorable<T2, T1>`, `(void)(*E1 = iter_exchange_move(E2, E1))`, except that `E1` is evaluated only once.

(1.4) — Otherwise, `ranges::iter_swap(E1, E2)` is ill-formed.

2 If `ranges::iter_swap(E1, E2)` does not swap the values denoted by the expressions `E1` and `E2`, the program is ill-formed with no diagnostic required.

3 `iter_exchange_move` is an exposition-only function specified as:

`template<class X, class Y>`
`  constexpr iter_value`~~`_type`~~`_t<remove_reference_t<X>> iter_exchange_move(X&& x, Y&& y)`
`    noexcept(`*see below*`);`

4 *Effects:* Equivalent to:

`iter_value`~~`_type`~~`_t<remove_reference_t<X>> old_value(iter_move(x));`
`*x = iter_move(y);`
`return old_value;`

5 *Remarks:* The expression in the `noexcept` is equivalent to:

```
NE(remove_reference_t<X>, remove_reference_t<Y>) &&
NE(remove_reference_t<Y>, remove_reference_t<X>)
```

Where `NE(T1, T2)` is the expression:

`is_nothrow_constructible_v<iter_value`~~`_type`~~`_t<T1>, iter_rvalue_reference_t<T1>>`~~`::value`~~` &&`
`is_nothrow_assignable_v<iter_value`~~`_type`~~`_t<T1>&, iter_rvalue_reference_t<T1>>`~~`::value`~~` &&`
`is_nothrow_assignable_v<iter_reference_t<T1>, iter_rvalue_reference_t<T2>>`~~`::value`~~` &&`
`is_nothrow_assignable_v<iter_reference_t<T1>, iter_value`~~`_type`~~`_t<T2>>`~~`::value`~~`> &&`
`is_nothrow_move_constructible_v<iter_value`~~`_type`~~`_t<T1>>`~~`::value`~~` &&`
`noexcept(ranges::iter_move(declval<T1&>()))`

[Editor's note: The Ranges TS concept definitions `Readable` through `RandomAccessIterator` and P0944's `ContiguousIterator` concept all get moved into a new section [iterator.concepts]:]

28.3.4 Iterator concepts [iterator.concepts]

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

1 The `Readable` concept is satisfied by types that are readable by applying `operator*` including pointers, smart pointers, and iterators.

```
template<class In>
concept ~~bool~~ Readable =
  requires {
    typename iter_value~~_type~~_t<In>;
    typename iter_reference_t<In>;
    typename iter_rvalue_reference_t<In>;
  } &&
  CommonReference<iter_reference_t<In>&&, iter_value~~_type~~_t<In>&> &&
  CommonReference<iter_reference_t<In>&&, iter_rvalue_reference_t<In>&&> &&
  CommonReference<iter_rvalue_reference_t<In>&&, const iter_value~~_type~~_t<In>&>;
```

2 Given a value `i` of type `I`, `Readable<I>` is satisfied only if the expression `*i` (which is indirectly required to be valid via the exposition-only *dereferenceable* concept (29.3)) is equality-preserving.

28.3.4.2 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 ~~bool~~ 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`. `Writable<Out, T>` is satisfied only if

(2.1) — If `Readable<Out> && Same<iter_value~~_type~~_t<Out>, decay_t<T>>` is satisfied, 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 ([basic.lval]), the resulting state of the object it denotes is valid but unspecified ([lib.types.movedfrom]).

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*]

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

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 ~~bool~~ WeaklyIncrementable =
  Semiregular<I> &&
  requires(I i) {
    typename iter_difference~~_type~~_t<I>;
    requires SignedIntegral<iter_difference~~_type~~_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*. `WeaklyIncrementable<I>` is satisfied 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*]

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

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 requirement supersedes the annotations on the increment expressions in the definition of `WeaklyIncrementable`. — *end note*]

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

2 Let `a` and `b` be incrementable objects of type `I`. `Incrementable<I>` is satisfied 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 satisfy `Incrementable`. — *end note*]

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

1 The `Iterator` concept forms the basis of the iterator concept taxonomy; every iterator satisfies the `Iterator` requirements. This concept specifies operations for dereferencing and incrementing an iterator. Most algorithms will require additional operations to compare iterators with sentinels (28.3.4.6), to read (28.3.4.8) or write (28.3.4.9) values, or to provide a richer set of iterator movements (28.3.4.10, 28.3.4.11, 28.3.4.12).)

```
template<class I>
concept bool Iterator =
  requires(I i) {
    { *i } -> auto&&; // Requires: i is dereferenceable
  } &&
  WeaklyIncrementable<I>;
```

2 [*Note:* The requirement that the result of dereferencing the iterator is deducible from `auto&&` means that it cannot be `void`. — *end note*]

28.3.4.6 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 bool Sentinel =
  Semiregular<S> &&
  Iterator<I> &&
  weakly-equality-comparable-with<S, I>; // See [concept.equalitycomparable]
```

2 Let `s` and `i` be values of type `S` and `I` such that `[i, s)` denotes a range. Types `S` and `I` satisfy `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 `==` can change over time. Given an iterator `i` and sentinel `s` such that `[i, s)` denotes a range and `i != s`, `[i, s)` is not required to continue to denote a range after incrementing any iterator equal to `i`. Consequently, `i == s` is no longer required to be well-defined.

28.3.4.7 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 bool 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<difference_type_t<I>>&&;
    s - i; requires Same<decltype(s - i), iter_difference_t<I>>;
    { i - s } -> Same<difference_type_t<I>>&&;
    i - s; requires Same<decltype(i - s), 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`. `SizedSentinel<S, I>` is satisfied only if:

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

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

3 [*Note:* `disable_sized_sentinel` provides a mechanism to enable use of sentinels and iterators with the library that meet the syntactic requirements but do not in fact satisfy `SizedSentinel`. A program that instantiates a library template that requires `SizedSentinel` with an iterator type `I` and sentinel type `S` that meet the syntactic requirements of `SizedSentinel<S, I>` but do not satisfy `SizedSentinel` is ill-formed with no diagnostic required ~~unless `disable_sized_sentinel<S, I>` evaluates to `true`~~ ([structure.requirements]). — *end note*]

4 [*Note:* The `SizedSentinel` concept is satisfied by pairs of `RandomAccessIterators` (28.3.4.12) and by counted iterators and their sentinels (28.5.6.1). — *end note*]

28.3.4.8 Concept InputIterator [iterator.concept.input]

1 The `InputIterator` concept is a refinement of `Iterator` (28.3.4.5). It defines requirements for a type whose referenced values can be read (from the requirement for `Readable` (28.3.4.1)) and which can be both pre- and post-incremented. [*Note:* ~~Unlike in ISO/IEC 14882, input iterators are not required to satisfy `EqualityComparable` ([concept.equalitycomparable]).~~ Unlike the input iterator requirements in 28.3.5.2, the `InputIterator` concept does not require equality comparison. — *end note*]

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

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

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

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

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

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

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

28.3.4.9 Concept OutputIterator [iterator.concept.output]

1 The `OutputIterator` concept is a refinement of `Iterator` (28.3.4.5). It defines requirements for a type that can be used to write values (from the requirement for `Writable` (28.3.4.2)) and which can be both pre- and post-incremented. However, output iterators are not required to satisfy `EqualityComparable`.

```
template<class I, class T>
concept ~~bool~~ 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`. `OutputIterator<I, T>` is satisfied 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*]

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

1 The `ForwardIterator` concept refines `InputIterator` (28.3.4.8), adding equality comparison and the multi-pass guarantee, specified below.

```
template<class I>
concept ~~bool~~ ForwardIterator =
  InputIterator<I> &&
  DerivedFrom<~~iterator_category_t<I>~~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 `([](X x){++x;}(a), *a)` is equivalent to the expression `*a`.

5 [*Note:* The requirement that `a == b` implies `++a == ++b` (which is not true for weaker iterators) 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*]

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

1 The `BidirectionalIterator` concept refines `ForwardIterator` (28.3.4.10), and adds the ability to move an iterator backward as well as forward.

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

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

3 Let `a` and `b` be decrementable objects of type `I`. `BidirectionalIterator<I>` is satisfied only if:

(3.1) — ~~`&`~~`addressof(--a) == `~~`&`~~`addressof(a)`.

(3.2) — If `bool(a == b)`, then `bool(a-- == b)`.

(3.3) — If `bool(a == b)`, then after evaluating both `a--` and `--b`, `bool(a == b)` still holds.

(3.4) — If a is incrementable and bool(a == b), then bool(--(++a) == b).

(3.5) — If bool(a == b), then bool(++(--a) == b).

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

1 The RandomAccessIterator concept refines BidirectionalIterator (28.3.4.11) and adds support for constant-time advancement with +=, +, -=, and -, and the computation of distance in constant time with -. Random access iterators also support array notation via subscripting.

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

2 Let a and b be valid iterators of type I such that b is reachable from a. Let n be the smallest value of type iter_difference~~_type~~_t<I> such that after n applications of ++a, then bool(a == b). RandomAccessIterator<I> is satisfied 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 integers x and y, if a + (x + y) is valid, then a + (x + y) is equal to (a + x) + y.

(2.5) — a + 0 is equal to a.

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

28.3.4.13 Concept ContiguousIterator [iterator.concept.contiguous]

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

```
template<class I>
concept ContiguousIterator =
  RandomAccessIterator<I> &&
  DerivedFrom<~~iterator_category_t<I>~~ITER_CONCEPT(I), contiguous_iterator_tag> &&
  is_lvalue_reference_v<iter_reference_t<I>> &&
  Same<iter_value~~_type~~_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. ContiguousIterator<I> is satisfied only if

```
addressof(*(a + (b - a))) == addressof(*a) + (b - a)
```

is `true`.

[Editor's note: Add a new subclause [iterator.cpp17] with the "Cpp17" iterator category requirement subclauses:]

28.3.5 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>` (28.3.2.3). — *end note*]

28.3.5.1 *Cpp17Iterator* [iterator.iterators]

28.3.5.2 Input iterators [input.iterators]

[...]

28.3.5.3 Output iterators [output.iterators]

[...]

28.3.5.4 Forward iterators [forward.iterators]

[...]

28.3.5.5 Bidirectional iterators [bidirectional.iterators]

[...]

28.3.5.6 Random access iterators [random.access.iterators]

[...]

[Editor's note: Relocate Ranges TS [indirectcallable] here and modify as follows:]

28.3.6 Indirect callable requirements [indirectcallable]

28.3.6.1 General [indirectcallable.general]

1 There are several concepts that group requirements of algorithms that take callable objects ([func.require]) as arguments.

28.3.6.2 Indirect callables [indirectcallable.indirectinvocable]

1 The indirect callable concepts are used to constrain those algorithms that accept callable objects ([func.def]) as arguments.

```
namespace std {
  template<class F, class I>
  concept bool IndirectUnaryInvocable =
    Readable<I> &&
    CopyConstructible<F> &&
    Invocable<F&, iter_value_type_t<I>&> &&
    Invocable<F&, iter_reference_t<I>> &&
    Invocable<F&, iter_common_reference_t<I>> &&
    CommonReference<
      result_of_t<F&(value_type_t<I>&)>invoke_result_t<F&, iter_value_t<I>&>,
      result_of_t<F&(reference_t<I>&&)>invoke_result_t<F&, iter_reference_t<I>>>;

  template<class F, class I>
  concept bool IndirectRegularUnaryInvocable =
    Readable<I> &&
    CopyConstructible<F> &&
    RegularInvocable<F&, iter_value_type_t<I>&> &&
    RegularInvocable<F&, iter_reference_t<I>> &&
    RegularInvocable<F&, iter_common_reference_t<I>> &&
    CommonReference<
      result_of_t<F&(value_type_t<I>&)>invoke_result_t<F&, iter_value_t<I>&>,
      result_of_t<F&(reference_t<I>&&)>invoke_result_t<F&, iter_reference_t<I>>>;
```

```
  template<class F, class I>
  concept ~~bool~~ IndirectUnaryPredicate =
    Readable<I> &&
    CopyConstructible<F> &&
    Predicate<F&, iter_value~~_type~~_t<I>&> &&
    Predicate<F&, iter_reference_t<I>> &&
    Predicate<F&, iter_common_reference_t<I>>;

  template<class F, class I1, class I2 = I1>
  concept ~~bool~~ IndirectRelation =
    Readable<I1> && Readable<I2> &&
    CopyConstructible<F> &&
    Relation<F&, iter_value~~_type~~_t<I1>&, iter_value~~_type~~_t<I2>&> &&
    Relation<F&, iter_value~~_type~~_t<I1>&, iter_reference_t<I2>> &&
    Relation<F&, iter_reference_t<I1>, iter_value~~_type~~_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 ~~bool~~ IndirectStrictWeakOrder =
    Readable<I1> && Readable<I2> &&
    CopyConstructible<F> &&
    StrictWeakOrder<F&, iter_value~~_type~~_t<I1>&, iter_value~~_type~~_t<I2>&> &&
    StrictWeakOrder<F&, iter_value~~_type~~_t<I1>&, iter_reference_t<I2>> &&
    StrictWeakOrder<F&, iter_reference_t<I1>, iter_value~~_type~~_t<I2>&> &&
    StrictWeakOrder<F&, iter_reference_t<I1>, iter_reference_t<I2>> &&
    StrictWeakOrder<F&, iter_common_reference_t<I1>, iter_common_reference_t<I2>>;

  ~~template<class> struct indirect_result_of { };~~

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

[Editor's note: Relocate [projected] here from the Ranges TS and change as follows:]

28.3.6.3 Class template `projected` [projected]

1 The `projected` class template is intended for use when specifying the constraints of algorithms that accept callable objects and projections (20.3.18). It bundles a `Readable` type `I` and a function `Proj` into a new `Readable` type whose `reference` type is the result of applying `Proj` to the `iter_reference_t` of `I`.

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

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

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

[Editor's note: Relocate Ranges TS [commonalgoreq] here and modify as follows:]

28.3.7 Common algorithm requirements [commonalgoreq]

28.3.7.1 General [commonalgoreq.general]

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

2 [*Note:* The `ranges::equal_to<>` and `ranges::less<>` (24.14.8) function object types used in the concepts below impose ~~additional~~ constraints on their arguments ~~beyond~~ in addition to those that appear explicitly in the concepts' bodies. `ranges::equal_to<>` requires its arguments satisfy `EqualityComparableWith` ([concept.equalitycomparable]), and `ranges::less<>` requires its arguments satisfy `StrictTotallyOrderedWith` ([concept.stricttotallyordered]). — *end note*]

28.3.7.2 Concept `IndirectlyMovable` [commonalgoreq.indirectlymovable]

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

```
template<class In, class Out>
concept ~~bool~~ 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 ~~bool~~ IndirectlyMovableStorable =
  IndirectlyMovable<In, Out> &&
  Writable<Out, iter_value~~_type~~_t<In>> &&
  Movable<iter_value~~_type~~_t<In>> &&
  Constructible<iter_value~~_type~~_t<In>, iter_rvalue_reference_t<In>> &&
  Assignable<iter_value~~_type~~_t<In>&, iter_rvalue_reference_t<In>>;
```

3 Let `i` be a dereferenceable value of type `In`. `IndirectlyMovableStorable<In, Out>` is satisfied 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 ([lib.types.movedfrom]).

28.3.7.3 Concept `IndirectlyCopyable` [commonalgoreq.indirectlycopyable]

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

```
template<class In, class Out>
concept ~~bool~~ 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 ~~bool~~ IndirectlyCopyableStorable =
  IndirectlyCopyable<In, Out> &&
  Writable<Out, const iter_value~~_type~~_t<In>&> &&
  Copyable<iter_value~~_type~~_t<In>> &&
  Constructible<iter_value~~_type~~_t<In>, iter_reference_t<In>> &&
  Assignable<iter_value~~_type~~_t<In>&, iter_reference_t<In>>;
```

3 Let `i` be a dereferenceable value of type `In`. `IndirectlyCopyableStorable<In, Out>` is satisfied 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 ([lib.types.movedfrom]).

28.3.7.4 Concept `IndirectlySwappable` [commonalgoreq.indirectlyswappable]

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

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

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

28.3.7.5 Concept `IndirectlyComparable` [commonalgoreq.indirectlycomparable]

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

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

28.3.7.6 Concept `Permutable` [commonalgoreq.permutable]

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

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

28.3.7.7 Concept `Mergeable` [commonalgoreq.mergeable]

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

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

28.3.7.8 Concept `Sortable` [commonalgoreq.sortable]

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

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

28.4 Iterator primitives [iterator.primitives]

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

28.4.1 Standard iterator tags [std.iterator.tags]

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

```
namespace std {
  struct input_iterator_tag { };
  struct output_iterator_tag { };
  struct forward_iterator_tag: public input_iterator_tag { };
  struct bidirectional_iterator_tag: public forward_iterator_tag { };
  struct random_access_iterator_tag: public bidirectional_iterator_tag { };
  struct contiguous_iterator_tag:  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: [...] — *end example*]

[...] [Editor's note: Remainder as in the working draft.]

28.4.2 Iterator operations [iterator.operations]

[Editor's note: As in the working draft with no modifications.]

[Editor's note: Relocate [iterator.operations] here from the Ranges TS and modify as follows:]

28.4.3 Range iterator operations [range.iterator.operations]

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

2 The function templates defined in this subclause are not found by argument-dependent name lookup ([basic.lookup.argdep]). When found by unqualified ([basic.lookup.unqual]) name lookup for the *postfix-expression* in a function call ([expr.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 ([temp.func.order]) than `std::ranges::distance` since the former requires its first two parameters to have the same type. — *end example*]

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

1 ~~The name `advance` denotes a customization point object ([customization.point.object]). It has the following function call operators:~~

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

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

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

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

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

5 *Effects:*

(5.1) — If `Assignable<I&, S>` is satisfied, equivalent to `i = std::move(bound)`.

(5.2) — Otherwise, if `SizedSentinel<S, I>` is satisfied, equivalent to `advance(i, bound - i)`.

(5.3) — Otherwise, increments `i` until `i == bound`.

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

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

7 *Effects:*

(7.1) — If `SizedSentinel<S, I>` is satisfied:

(7.1.1) — If |`n`| >= |`bound - i`|, equivalent to `advance(i, bound)`.

(7.1.2) — Otherwise, equivalent to `advance(i, n)`.

(7.2) — Otherwise, increments (or decrements for negative `n`) iterator `i` either `n` times or until `i == bound`, whichever comes first.

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

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

1 ~~The name `distance` denotes a customization point object. It has the following function call operators:~~

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

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

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

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

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

```
return ranges::size(r); // 29.5.1
```

Otherwise, equivalent to:

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

5 ~~*Remarks:* Instantiations of this member function template may be ill-formed if the declarations in `<experimental/ranges/range>` are not in scope at the point of instantiation ([temp.point]).~~

`template<SizedRange R>`
`constexpr difference_type_t<iterator_t<R>> operator()(R&& r) const;`

6 *Effects:* Equivalent to: `return ranges::size(r);` (29.5.1)

7 *Remarks:* Instantiations of this member function template may be ill-formed if the declarations in `<experimental/ranges/range>` are not in scope at the point of instantiation ([temp.point]).

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

1 ~~The name `next` denotes a customization point object. It has the following function call operators:~~

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

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

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

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

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

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

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

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

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

1 ~~The name `prev` denotes a customization point object. It has the following function call operators:~~

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

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

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

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

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

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

28.5 Iterator adaptors [predef.iterators]

28.5.1 Reverse iterators [reverse.iterators]

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

28.5.1.1 Class template `reverse_iterator` [reverse.iterator]

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

```
namespace std {
  template<class Iterator>
  class reverse_iterator {
  public:
    using iterator_type     = Iterator;
    ~~using iterator_category = typename iterator_traits<Iterator>::iterator_category;~~
    ~~using value_type = typename iterator_traits<Iterator>::value_type;~~
    ~~using difference_type = typename iterator_traits<Iterator>::difference_type;~~
    ~~using pointer = typename iterator_traits<Iterator>::pointer;~~
    ~~using reference = typename iterator_traits<Iterator>::reference;~~
    using iterator_category = see below;
    using iterator_concept = see below;
    using value_type = iter_value_t<Iterator>;
    using difference_type = iter_difference_t<Iterator>;
    using pointer = Iterator;
    using reference = iter_reference_t<Iterator>;

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

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

```
    [...]

    constexpr reverse_iterator& operator-=(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 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_category` denotes `random_access_iterator_tag` if `iterator_traits<Iterator>::iterator_category` is `contiguous_iterator_tag`, and `iterator_traits<Iterator>::iterator_category` otherwise.

2 The member *typedef-name* `iterator_concept` denotes `random_access_iterator_tag` if `Iterator` models `RandomAccessIterator`, and `bidirectional_iterator_tag` otherwise.

[Editor's note: Change [reverse.iterator.elem] as follows (note that this change resolves LWG 1052):]

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

[...]

`constexpr pointer operator->() const;`

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

[...]

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

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

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

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

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

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

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

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

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

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

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

[Editor's note: Insert a new initial paragraph:]

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

[...]

28.5.2 Insert iterators [insert.iterators]

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

[Editor's note: Change `back_insert_iterator` so that it satisfies the `Iterator` concept.)]

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

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

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

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

[...]

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

[Editor's note: Change `front_insert_iterator` so that it satisfies the `Iterator` concept.]

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

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

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

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

[...]

28.5.2.3 Class template `insert_iterator` [insert.iterator]

[Editor's note: Change `insert_iterator` so it satisfies the `Iterator` concept:]

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

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

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

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

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

[...]

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

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

[...]

28.5.2.3.2 `inserter` [inserter]

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

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

[Editor's note: Retitle [move.iterators] from "Move iterators" to "Move iterators and sentinels" and modify as follows:]

28.5.3 Move iterators and sentinels [move.iterators]

[...]

28.5.3.1 Class template move_iterator [move.iterator]

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

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

    [...]

    constexpr move_iterator& operator++();
    constexpr ~~move_iterator~~decltype(auto) operator++(int);
    constexpr move_iterator& operator--();

    [...]

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

    [Editor's note: These operators are relocated from move_sentinel.]
    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 Iterator>
    constexpr move_iterator<Iterator> operator+(
```

~~`typename move_iterator<Iterator>::difference_type`~~`iter_difference_t<move_iterator<Iterator>>` `n,`

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

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

[...]

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

[...]

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

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

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

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

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

[Editor's note: My preference is to remove `operator->` since for `move_iterator`, the expressions `(*i).m` and `i->m` are not, and cannot be, equivalent. I am leaving the operator as-is in an excess of caution; perhaps we should consider deprecation for C++20?]

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

1 *Returns:* `current`.

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

[...]

`constexpr` ~~`move_iterator`~~`decltype(auto)` `operator++(int);`

3 *Effects:* ~~As if by~~If `Iterator` models `ForwardIterator`, equivalent to:

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

Otherwise, equivalent to `++current`.

[...]

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

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

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

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

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

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

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

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

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

[...]

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

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

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

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

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

3 *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)));
```

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

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

5 *Returns:* `x + n`.

[...]

[Editor's note: Relocate Ranges TS [move.sentinel] here and modify as follows:]

28.5.3.4 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` satisfy `Sentinel<S, I>`, `Sentinel<move_sentinel<S>, move_iterator<I>>` is satisfied as well.

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

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

— *end example*]

```
namespace std { ~~namespace experimental { namespace ranges { inline namespace v1 {~~
  template<Semiregular S>
  class move_sentinel {
  public:
    constexpr move_sentinel();
    explicit constexpr move_sentinel(S s);
    template<ConvertibleTo<S> S2>
      constexpr move_sentinel(const move_sentinel<~~ConvertibleTo<S>~~S2>& s);
    template<ConvertibleTo<S> S2>
      constexpr move_sentinel& operator=(const move_sentinel<~~ConvertibleTo<S>~~S2>& s);

    constexpr S base() const;

  private:
    S last; // exposition only
  };

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

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

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

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

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

```
constexpr move_sentinel();
```

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

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

2 *Effects:* Constructs a `move_sentinel`, initializing `last` with `std::move(s)`.

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

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

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

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

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

2 *Returns:* `*this`.

[Editor's note: Subclauses [move.sent.op.comp] and [move.sent.nonmember] from the Ranges TS intentionally ommitted.]

[Editor's note: Relocate [iterators.common] from the Ranges TS here and modify as follows:]

28.5.4 Common iterators [iterators.common]

[Editor's note: TODO: respecify this in terms of `std::variant`.]

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>;
// call fun on a range of 10 ints
fun(CI(make_counted_iterator(s.begin(), 10)),
    CI(default_sentinel()));
```

— *end example*]

28.5.4.1 Class template `common_iterator` [common.iterator]

```
namespace std { ~~namespace experimental { namespace ranges { inline namespace v1 {~~
  template<Iterator I, Sentinel<I> S>
    requires !Same<I, S>
  class common_iterator {
  public:
    using difference_type = iter_difference~~_type~~_t<I>;

    constexpr common_iterator();
    constexpr common_iterator(I i);
    constexpr common_iterator(S s);
    ~~constexpr common_iterator(const common_iterator<ConvertibleTo<I>, ConvertibleTo<S>>& u);~~
    template<ConvertibleTo<I> I2, ConvertibleTo<S> S2>
      constexpr common_iterator(const common_iterator<I2, S2>& that);

    ~~common_iterator& operator=(const common_iterator<ConvertibleTo<I>, ConvertibleTo<S>>& u);~~
    template<ConvertibleTo<I> I2, ConvertibleTo<S> S2>
      common_iterator& operator=(const common_iterator<I2, S2>& that);

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

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

```
    [Editor's note:  Make non-member operators hidden friends.]
    template<~~class I1~~, class I2, ~~Sentinel<I2> S1,~~ Sentinel<I1> S2>
      requires Sentinel<S1, I2>
    friend bool operator==(
      const common_iterator~~<I1, S1>~~& x, const common_iterator<I2, S2>& y);
    template<~~class I1,~~ class I2, ~~Sentinel<I2> S1,~~ Sentinel<I1> S2>
      requires Sentinel<S1, I2> && EqualityComparableWith<I1, I2>
    friend bool operator==(
      const common_iterator~~<I1, S1>~~& x, const common_iterator<I2, S2>& y);
    template<~~class I1,~~ class I2, ~~Sentinel<I2> S1,~~ Sentinel<I1> S2>
      requires Sentinel<S1, I2>
    friend bool operator!=(
      const common_iterator~~<I1, S1>~~& x, const common_iterator<I2, S2>& y);

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

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

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

  template<Readable I, class S>
  struct ~~value_type~~readable_traits<common_iterator<I, S>> {
    using value_type = iter_value~~_type~~_t<I>;
  };

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

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

1 For both specializations of `iterator_traits` for `common_iterator<I, S>`, the nested *typedef-name* `pointer` is defined as follows:

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

(1.2) — Otherwise, `pointer` denotes `void`.

28.5.4.2 `common_iterator` operations [common.iterator.ops]

28.5.4.2.1 `common_iterator` constructors and conversions [common.iterator.op.const]

```
constexpr common_iterator();
```

1 *Effects:* Constructs a `common_iterator`, value-initializing `is_sentinel`, `iter`, and `sentinel`. 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 `I`.

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

2 *Effects:* Constructs a `common_iterator`, initializing `is_sentinel` with `false`, `iter` with `i`, and value-initializing `sentinel`.

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

3 *Effects:* Constructs a `common_iterator`, initializing `is_sentinel` with `true`, value-initializing `iter`, and initializing `sentinel` with `s`.

~~`constexpr common_iterator(const common_iterator<ConvertibleTo<I>, ConvertibleTo<S>>& u);`~~

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

4 *Effects:* Constructs a `common_iterator`, initializing `is_sentinel` with ~~`u`~~`that.is_sentinel`, `iter` with ~~`u`~~`that.iter`, and `sentinel` with ~~`u`~~`that.sentinel`.

~~`common_iterator& operator=(const common_iterator<ConvertibleTo<I>, ConvertibleTo<S>>& u);`~~

```
template<ConvertibleTo<I> I2, ConvertibleTo<S> S2>
  common_iterator& operator=(const common_iterator<I2, S2>& that);
```

5 *Effects:* Assigns ~~`u`~~`that.is_sentinel` to `is_sentinel`, ~~`u`~~`that.iter` to `iter`, and ~~`u`~~`that.sentinel` to `sentinel`.

6 *Returns:* `*this`

28.5.4.2.2 `common_iterator::operator*` [common.iterator.op.star]

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

1 *Requires:* `!is_sentinel`

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

28.5.4.2.3 `common_iterator::operator->` [common.iterator.op.ref]

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

1 *Requires:* `!is_sentinel`

2 *Effects:* Equivalent to:

(2.1) — If `I` is a pointer type or if the expression `i.operator->()` is well-formed, `return iter;`

(2.2) — Otherwise, if the expression `*iter` is a glvalue:

```
auto&& tmp = *iter;
return addressof(tmp);
```

(2.3) — Otherwise, `return proxy(*iter);` where `proxy` is the exposition-only class:

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

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>>~~::value~~ ||
   Constructible<iter_value~~_type~~_t<I>, iter_reference_t<I>>)
```

28.5.4.2.4 `common_iterator::operator++` [common.iterator.op.incr]

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

1 *Requires:* `!is_sentinel`

2 *Effects:* Equivalent to `++iter`.

3 *Returns:* `*this`.

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

4 *Requires:* `!is_sentinel`.

5 *Effects:* Equivalent to: `return iter++;`

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

6 *Requires:* `!is_sentinel`

7 *Effects:* Equivalent to:

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

28.5.4.2.5 `common_iterator` comparisons [common.iterator.op.comp]

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

1 *Effects:* Equivalent to:

```
return x.is_sentinel
  ?  (y.is_sentinel || y.iter == x.sentinel)
  : (!y.is_sentinel || x.iter == y.sentinel);
```

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

2 *Effects:* Equivalent to:

```
return x.is_sentinel
  ? (y.is_sentinel || y.iter == x.sentinel)
  : (y.is_sentinel
    ? x.iter == y.sentinel
    : x.iter == y.iter);
```

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

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

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

4 *Effects:* Equivalent to:

```
    return x.is_sentinel
      ? (y.is_sentinel ? 0 : x.sentinel - y.iter)
      : (y.is_sentinel ? x.iter - y.sentinel : x.iter - y.iter);
```

28.5.4.2.6 `iter_move` [common.iterator.op.iter_move]

```
friend iter_rvalue_reference_t<I> iter_move(const common_iterator& i)
  noexcept(~~see below~~noexcept(ranges::iter_move(i.iter)))
    requires InputIterator<I>;
```

1 *Requires:* `!i.is_sentinel`.

2 *Effects:* Equivalent to: `return ranges::iter_move(i.iter);`

3 ~~*Remarks:* The expression in `noexcept` is equivalent to:~~

~~`noexcept(ranges::iter_move(i.iter))`~~

28.5.4.2.7 `iter_swap` [common.iterator.op.iter_swap]

```
template<IndirectlySwappable<I> I2>
  friend void iter_swap(const common_iterator& x, const common_iterator<I2>& y)
    noexcept(~~see below~~noexcept(ranges::iter_swap(x.iter, y.iter)));
```

1 *Requires:* `!x.is_sentinel && !y.is_sentinel`.

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

3 ~~*Remarks:* The expression in `noexcept` is equivalent to:~~

~~`noexcept(ranges::iter_swap(x.iter, y.iter))`~~

[Editor's note: Relocate Ranges TS [default.sentinels] here and modify as follows:]

28.5.5 Default sentinels [default.sentinels]

28.5.5.1 Class `default_sentinel` [default.sent]

```
namespace std { ~~namespace experimental { namespace ranges { inline namespace v1 {~~
  class default_sentinel { };
}~~}}}~~
```

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

[Editor's note: Merge Ranges TS [iterators.counted] and modify as follows:]

28.5.6 Counted iterators [iterators.counted]

28.5.6.1 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 its distance from its starting position. It can be used together with class `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*.

[Editor's note: The following example incorporates the PR for stl2#554:]

2 [*Example:*

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

— *end example*]

3 Two values `i1` and `i2` of (possibly differing) 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 { ~~namespace experimental { namespace ranges { inline namespace v1 {~~
  template<Iterator I>
  class counted_iterator {
  public:
    using iterator_type = I;
    using difference_type = iter_difference~~_type~~_t<I>;

    constexpr counted_iterator();
    constexpr counted_iterator(I x, iter_difference~~_type~~_t<I> n);
    template<ConvertibleTo<I> I2>
      constexpr counted_iterator(const counted_iterator<~~ConvertibleTo<I>~~I2>& that);
    template<ConvertibleTo<I> I2>
      constexpr counted_iterator& operator=(const counted_iterator<~~ConvertibleTo<I>~~I2>& that);

    [Editor's note: Non-member operators have been inlined, as members or hidden friends.]

    constexpr I base() const;
    constexpr iter_difference~~_type~~_t<I> count() const;
    constexpr decltype(auto) operator*();
    constexpr decltype(auto) operator*() const
      requires dereferenceable<const I>;

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

    constexpr counted_iterator  operator+ (difference_type n) const
      requires RandomAccessIterator<I>;
    ~~template<RandomAccessIterator I>~~
      friend constexpr counted_iterator~~<I>~~ operator+(
        iter_difference~~_type~~_t<I> n, const counted_iterator~~<I>~~& ~~x~~self)
          requires RandomAccessIterator<I>;
    constexpr counted_iterator& operator+=(difference_type n)
      requires RandomAccessIterator<I>;

    constexpr counted_iterator  operator- (difference_type n) const
      requires RandomAccessIterator<I>;
    template<~~class I1, class~~Common<I> I2>
        ~~requires Common<I1, I2>~~
      constexpr iter_difference~~_type~~_t<I2> operator-(
        ~~const counted_iterator<I1>& x,~~ const counted_iterator<I2>& ~~y~~that) const;
    ~~template<class I>~~
      constexpr iter_difference~~_type~~_t<I> operator-(
        ~~const counted_iterator<I>& x,~~ default_sentinel ~~y~~) const;
    ~~template<class I>~~
      friend constexpr iter_difference~~_type~~_t<I> operator-(
        default_sentinel ~~x~~, const counted_iterator~~<I>~~& ~~y~~self);
    constexpr counted_iterator& operator-=(difference_type n)
      requires RandomAccessIterator<I>;

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

    template<~~class I1, class~~Common<I> I2>
        ~~requires Common<I1, I2>~~
      constexpr bool operator==(
        ~~const counted_iterator<I1>& x,~~ const counted_iterator<I2>& ~~y~~that) const;
    constexpr bool operator==(
      ~~const counted_iterator<auto>& x,~~ default_sentinel) const;
```

```
    friend constexpr bool operator==(
      default_sentinel, const counted_iterator~~<auto>~~& ~~x~~self);

    template<~~class I1, class~~Common<I> I2>
        ~~requires Common<I1, I2>~~
      constexpr bool operator!=(
        ~~const counted_iterator<I1>& x,~~ const counted_iterator<I2>& ~~y~~that) const;
    constexpr bool operator!=(
      ~~const counted_iterator<auto>& x,~~ default_sentinel ~~y~~that) const;
    friend constexpr bool operator!=(
      default_sentinel ~~x~~that, const counted_iterator~~<auto>~~& ~~y~~self);

    template<~~class I1, class~~Common<I> I2>
        ~~requires Common<I1, I2>~~
      constexpr bool operator<(
        ~~const counted_iterator<I1>& x,~~ const counted_iterator<I2>& ~~y~~that) const;
    template<~~class I1, class~~Common<I> I2>
        ~~requires Common<I1, I2>~~
      constexpr bool operator>(
        ~~const counted_iterator<I1>& x,~~ const counted_iterator<I2>& ~~y~~that) const;
    template<~~class I1, class~~Common<I> I2>
        ~~requires Common<I1, I2>~~
      constexpr bool operator<=(
        ~~const counted_iterator<I1>& x,~~ const counted_iterator<I2>& ~~y~~that) const;
    template<~~class I1, class~~Common<I> I2>
        ~~requires Common<I1, I2>~~
      constexpr bool operator>=(
        ~~const counted_iterator<I1>& x,~~ const counted_iterator<I2>& ~~y~~that) const;

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

  private:
    I current; // exposition only
    iter_difference~~_type~~_t<I> cnt; // exposition only
  };

  template<Readable I>
  struct ~~value_type~~readable_traits<counted_iterator<I>> {
    using value_type = iter_value~~_type~~_t<I>;
  };

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

  ~~template<Iterator I>~~
    ~~constexpr counted_iterator<I> make_counted_iterator(I i, difference_type_t<I> n);~~
}~~}}}~~
```

28.5.6.2 counted_iterator operations [counted.iter.ops]

28.5.6.2.1 counted_iterator constructors and conversions [counted.iter.op.const]

```
constexpr counted_iterator();
```

1 *Effects:* Constructs a `counted_iterator`, value-initializing `current` and `cnt`. 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 `I`.

`constexpr counted_iterator(I i, iter_difference`~~`_type`~~`_t<I> n);`

2 *Requires:* `n >= 0`

3 *Effects:* Constructs a `counted_iterator`, initializing `current` with `i` and `cnt` with `n`.

`template<ConvertibleTo<I> I2>`
`  constexpr counted_iterator(const counted_iterator<`~~`ConvertibleTo<I>`~~`I2>& that);`

4 *Effects:* Constructs a `counted_iterator`, initializing `current` with `that.current` and `cnt` with `that.cnt`.

`template<ConvertibleTo<I> I2>`
`  constexpr counted_iterator& operator=(const counted_iterator<`~~`ConvertibleTo<I>`~~`I2>& that);`

5 *Effects:* Assigns `that.current` to `current` and `that.cnt` to `cnt`.

28.5.6.2.2 `counted_iterator` conversion [counted.iter.op.conv]

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

1 *Returns:* `current`.

28.5.6.2.3 `counted_iterator` count [counted.iter.op.cnt]

`constexpr iter_difference`~~`_type`~~`_t<I> count() const;`

1 *Returns:* `cnt`.

28.5.6.2.4 `counted_iterator::operator*` [counted.iter.op.star]

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

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

28.5.6.2.5 `counted_iterator::operator++` [counted.iter.op.incr]

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

1 *Requires:* `cnt > 0`

2 *Effects:* Equivalent to:

```
++current;
--cnt;
```

3 *Returns:* `*this`.

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

4 *Requires:* `cnt > 0`.

5 *Effects:* Equivalent to:

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

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

6 *Requires:* `cnt > 0`

7 *Effects:* Equivalent to:

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

28.5.6.2.6 `counted_iterator::operator--` [counted.iter.op.decr]

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

1 *Effects:* Equivalent to:

```
--current;
++cnt;
```

2 *Returns:* `*this`.

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

3 *Effects:* Equivalent to:

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

28.5.6.2.7 `counted_iterator::operator+` [counted.iter.op.+]

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

1 *Requires:* `n <= cnt`

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

~~`template<RandomAccessIterator I>`~~
`friend constexpr counted_iterator`~~`<I>`~~` operator+(`
`iter_difference`~~`_type`~~`_t<I> n, const counted_iterator`~~`<I>`~~`& `~~`x`~~`self)`
`requires RandomAccessIterator<I>;`

3 *Requires:* `n <= `~~`x`~~`self.cnt`.

4 *Effects:* Equivalent to: `return `~~`x`~~`self + n;`

28.5.6.2.8 `counted_iterator::operator+=` [counted.iter.op.+=]

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

1 *Requires:* `n <= cnt`

2 *Effects:*

```
current += n;
cnt -= n;
```

3 *Returns:* `*this`.

28.5.6.2.9 `counted_iterator::operator-` [counted.iter.op.-]

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

1 *Requires:* `-n <= cnt`

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

`template<`~~`class I1, class`~~`Common<I> I2>`
~~`requires Common<I1, I2>`~~
`constexpr iter_difference`~~`_type`~~`_t<I2> operator-(`
~~`const counted_iterator<I1>& x,`~~` const counted_iterator<I2>& `~~`y`~~`that) const;`

3 *Requires:* ~~`x`~~`*this` and ~~`y`~~`that` shall refer to elements of the same sequence (28.5.6.1).

4 *Effects:* Equivalent to: `return `~~`y`~~`that.cnt - `~~`x`~~`*this.cnt;`

~~`template<class I>`~~
`constexpr iter_difference`~~`_type`~~`_t<I> operator-(`
~~`const counted_iterator<I>& x,`~~` default_sentinel `~~`y`~~`) const;`

5 *Effects:* Equivalent to: `return -`~~`x.`~~`cnt;`

~~`template<class I>`~~
`friend constexpr iter_difference`~~`_type`~~`_t<I> operator-(`
`default_sentinel` ~~`x`~~`, const counted_iterator`~~`<I>`~~`&` ~~`y`~~`self);`

6 *Effects:* Equivalent to: `return` ~~`y`~~`self.cnt;`

28.5.6.2.10 `counted_iterator::operator-=` [counted.iter.op.-=]

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

1 *Requires:* `-n <= cnt`

2 *Effects:*

```
current -= n;
cnt += n;
```

3 *Returns:* `*this`.

28.5.6.2.11 `counted_iterator::operator[]` [counted.iter.op.index]

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

1 *Requires:* `n <= cnt`

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

28.5.6.2.12 `counted_iterator` comparisons [counted.iter.op.comp]

`template<`~~`class I1, class`~~`Common<I> I2>`
~~`requires Common<I1, I2>`~~
`constexpr bool operator==(`
~~`const counted_iterator<I1>& x,`~~ `const counted_iterator<I2>&` ~~`y`~~`that) const;`

1 *Requires:* ~~`x`~~`*this` and ~~`y`~~`that` shall refer to elements of the same sequence (28.5.6.1).

2 *Effects:* Equivalent to: `return` ~~`x.`~~`cnt ==` ~~`y`~~`that.cnt;`

`constexpr bool operator==(`
~~`const counted_iterator<auto>& x,`~~ `default_sentinel) const;`

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

`friend constexpr bool operator==(`
`default_sentinel, const counted_iterator`~~`<auto>`~~`&` ~~`x`~~`self));`

4 *Effects:* Equivalent to: `return self == default_sentinel{};`

`template<`~~`class I1, class`~~`Common<I> I2>`
~~`requires Common<I1, I2>`~~
`constexpr bool operator!=(`
~~`const counted_iterator<I1>& x,`~~ `const counted_iterator<I2>&` ~~`y`~~`that) const;`
`constexpr bool operator!=(`
~~`const counted_iterator<auto>& x,`~~ `default_sentinel` ~~`y`~~`that) const;`

5 *Requires:* For the first overload, ~~`x`~~`*this` and ~~`y`~~`that` shall refer to elements of the same sequence (28.5.6.1).

6 *Effects:* Equivalent to: `return !(`~~`x == y`~~`*this == that);`

`friend constexpr bool operator!=(`
`default_sentinel` ~~`x`~~`that, const counted_iterator`~~`<auto>`~~`&` ~~`y`~~`self);`

7 *Effects:* Equivalent to: `return !(that == self);`

`template<`~~`class I1, class`~~`Common<I> I2>`
~~`requires Common<I1, I2>`~~
`constexpr bool operator<(`
~~`const counted_iterator<I1>& x,`~~ `const counted_iterator<I2>&` ~~`y`~~`that) const;`

8 *Requires:* ~~`x`~~`*this` and ~~`y`~~`that` shall refer to elements of the same sequence (28.5.6.1).

9 *Effects:* Equivalent to: `return` ~~`y`~~`that.cnt <` ~~`x`~~`*this.cnt;`

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

`template<`~~`class I1, class`~~`Common<I> I2>`
    ~~`requires Common<I1, I2>`~~
  `constexpr bool operator>(`
    ~~`const counted_iterator<I1>& x,`~~ `const counted_iterator<I2>&` ~~`y`~~`that) const;`

11 *Requires:* ~~`x`~~`*this` and ~~`y`~~`that` shall refer to elements of the same sequence (28.5.6.1).

12 *Effects:* Equivalent to: `return` ~~`y`~~`that <` ~~`x`~~`*this;`

`template<`~~`class I1, class`~~`Common<I> I2>`
    ~~`requires Common<I1, I2>`~~
  `constexpr bool operator<=(`
    ~~`const counted_iterator<I1>& x,`~~ `const counted_iterator<I2>&` ~~`y`~~`that) const;`

13 *Requires:* ~~`x`~~`*this` and ~~`y`~~`that` shall refer to elements of the same sequence (28.5.6.1).

14 *Effects:* Equivalent to: `return !(`~~`y`~~`that <` ~~`x`~~`*this);`

`template<`~~`class I1, class`~~`Common<I> I2>`
    ~~`requires Common<I1, I2>`~~
  `constexpr bool operator>=(`
    ~~`const counted_iterator<I1>& x,`~~ `const counted_iterator<I2>&` ~~`y`~~`that) const;`

15 *Requires:* ~~`x`~~`*this` and ~~`y`~~`that` shall refer to elements of the same sequence (28.5.6.1).

16 *Effects:* Equivalent to: `return !(`~~`x`~~`*this <` ~~`y`~~`that);`

28.5.6.2.13 counted_iterator customizations [counted.iter.nonmember]

`friend constexpr iter_rvalue_reference_t<I> iter_move(const counted_iterator& \oldtxt{i}\newtxt{self})`
  `noexcept(`~~*see below*~~`noexcept(ranges::iter_move(self.current)))`
    `requires InputIterator<I>;`

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

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

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

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

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

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

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

~~`template<Iterator I>`~~
  ~~`constexpr counted_iterator<I> make_counted_iterator(I i, difference_type_t<I> n);`~~

5 ~~*Requires:* n >= 0.~~

6 ~~*Returns:* counted_iterator<I>(i, n).~~

[Editor's note: Merge Ranges TS [unreachable.sentinels] and modify as follows (this wording integrates the PR for stl2#507):]

28.5.7 Unreachable sentinel [unreachable.sentinels]

28.5.7.1 Class unreachable [unreachable.sentinel]

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

2 [*Example:*

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

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

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  class unreachable {
  public:
    template<WeaklyIncrementable I>
      friend constexpr bool operator==(unreachable, const I&) noexcept;
    template<WeaklyIncrementable I>
      friend constexpr bool operator==(const I&, unreachable) noexcept;
    template<WeaklyIncrementable I>
      friend constexpr bool operator!=(unreachable, const I&) noexcept;
    template<WeaklyIncrementable I>
      friend constexpr bool operator!=(const I&, unreachable) noexcept;
  };

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

28.5.7.2 unreachable operations [unreachable.sentinel.ops]

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

1 ~~*Returns:* false.~~

Effects: Equivalent to: `return false;`

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

2 ~~*Returns:* true.~~

Effects: Equivalent to: `return true;`

28.6 Stream iterators [stream.iterators]

[...]

28.6.1 Class template istream_iterator [istream.iterator]

[...]

```
namespace std {
  template<class T, class charT = char, class traits = char_traits<charT>,
           class Distance = ptrdiff_t>
  class istream_iterator {
  public:
    [...]

    constexpr istream_iterator();
    constexpr istream_iterator(default_sentinel);
    istream_iterator(istream_type& s);

    [...]

    istream_iterator  operator++(int);
```

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

  private:
    [...]
  };

  [...]
}
```

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

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

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

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

[...]

28.6.1.2 istream_iterator operations [istream.iterator.ops]

[...]

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

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

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

28.6.2 Class template ostream_iterator [ostream.iterator]

[...]

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

    constexpr ostream_iterator() noexcept = default;
    ostream_iterator(ostream_type& s);

    [...]
```

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

[...]

28.6.3 Class template `istreambuf_iterator` [istreambuf.iterator]

[...]

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

    constexpr istreambuf_iterator() noexcept;
    constexpr istreambuf_iterator(default_sentinel) noexcept;
    istreambuf_iterator(const istreambuf_iterator&) noexcept = default;

    [...]

    bool equal(const istreambuf_iterator& b) const;

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

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

  [...]
}
```

[...]

28.6.3.2 `istreambuf_iterator` constructors [istreambuf.iterator.cons]

[...]

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

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

[...]

28.6.3.3 `istreambuf_iterator` operations [istreambuf.iterator.ops]

[...]

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

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

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

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

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

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

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

[...]

28.6.4 Class template ostreambuf_iterator [ostreambuf.iterator]

[...]

```
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   = voidptrdiff_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;

    [...]

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

[Editor's note: Add a new clause between [iterators] and [algorithms] with the following content:]

29 Ranges library [range]

29.1 General [range.general]

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

2 The following subclauses describe range and view requirements, and components for range primitives as summarized in Table 89.

Table 89 — Ranges library summary

	Subclause	Header(s)
29.4	Range access	<~~experimental/ranges/~~ranges>
29.5	Range primitives	
29.6	Requirements	
29.7	Range utilities	
29.8	Range adaptors	

29.2 decay_copy [range.decaycopy]

[Editor's note: TODO: Replace the definition of [thread.decaycopy] with this definition.]

1 Several places in this clause use the expression *DECAY_COPY*(x), which is expression-equivalent to:

```
decay_t<decltype((x))>(x)
```

29.3 Header <ranges> synopsis [range.synopsis]

```
#include <initializer_list>
#include <experimental/ranges/iterator>
```

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

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

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

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

    template<class>
    inline constexpr bool disable_sized_range = false;

    template<class T>
    struct enable_view { };

    struct view_base { };

    // 29.7.1, dangling wrapper
    template<class T> class dangling;

    template<Range R>
    using safe_iterator_t = see belowdecltype(ranges::begin(declval<R>()));

    // 29.6, range requirements

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

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

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

    // 29.6.5, BoundedRangeCommonRange
    template<class T>
    concept bool BoundedRangeCommonRange = see below;

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

```
// 29.6.7, OutputRange
template<class R, class T>
concept ~~bool~~ OutputRange = see below;

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

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

// 29.6.10, RandomAccessRange
template<class T>
concept ~~bool~~ RandomAccessRange = see below;

// 29.6.11, ContiguousRange
template<class T>
concept ContiguousRange = see below;

// 29.6.12
template<class T>
concept ViewableRange = see below;

// 29.7.2
template<class D>
  requires is_class_v<D>
class view_interface;

// 29.7.3.1
enum class subrange_kind : bool { unsized, sized };

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

// 29.8.4
namespace view { inline constexpr unspecified all = unspecified; }

template<ViewableRange R>
using all_view = decltype(view::all(declval<R>()));

// 29.8.5
template<InputRange R, IndirectUnaryPredicate<iterator_t<R>> Pred>
  requires View<R>
class filter_view;

namespace view { inline constexpr unspecified filter = unspecified; }

// 29.8.7
template<InputRange R, CopyConstructible F>
  requires View<R> && is_object_v<F &&> Invocable<F&, iter_reference_t<iterator_t<R>>>
class transform_view;

namespace view { inline constexpr unspecified transform = unspecified; }

// 29.8.9
template<WeaklyIncrementable I, Semiregular Bound = unreachable>
  requires weakly-equality-comparable-with<I, Bound>
class iota_view;

namespace view { inline constexpr unspecified iota = unspecified; }
```

```
  // 29.8.13
  template<InputRange R>
    requires View<R> && InputRange<iter_reference_t<iterator_t<R>>> &&
      (is_reference_v<iter_reference_t<iterator_t<R>>> ||
        View<iter_value~~_type~~_t<iterator_t<R>>>)
  class join_view;

  namespace view { inline constexpr unspecified join = unspecified; }

  // 29.8.15
  template<class T>
    requires is_object_v<T>
  class empty_view;

  namespace view {
    template<class T>
    inline constexpr empty_view<T> empty {};
  }

  // 29.8.16
  template<CopyConstructible T>
    requires is_object_v<T>
  class single_view;

  namespace view { inline constexpr unspecified single = unspecified; }

  // exposition only
  template<class R>
  concept tiny-range = see below;

  // 29.8.18
  template<InputRange Rng, ForwardRange Pattern>
    requires View<Rng> && View<Pattern> &&
        IndirectlyComparable<iterator_t<Rng>, iterator_t<Pattern>> &&
        (ForwardRange<Rng> || tiny-range<Pattern>)
  class split_view;

  namespace view { inline constexpr unspecified split = unspecified; }

  // 29.8.20
  namespace view { inline constexpr unspecified counted = unspecified; }

  // 29.8.21
  template<View Rng>
    requires !CommonRange<Rng>
  class common_view;

  namespace view { inline constexpr unspecified common = unspecified; }

  // 29.8.23
  template<View Rng>
    requires BidirectionalRange<Rng>
  class reverse_view;

  namespace view { inline constexpr unspecified reverse = unspecified; }
}

namespace view = ranges::view;

template<class I, class S, ranges::subrange_kind K>
  struct tuple_size<ranges::subrange<I, S, K>>
    : integral_constant<size_t, 2> {};
```

```
  template<class I, class S, ranges::subrange_kind K>
    struct tuple_element<0, ranges::subrange<I, S, K>> {
      using type = I;
    };
  template<class I, class S, ranges::subrange_kind K>
    struct tuple_element<1, ranges::subrange<I, S, K>> {
      using type = S;
    };
}}}
```

29.4 Range access [range.access]

[Editor's note: This wording integrates the PR for stl2#547.]

1 In addition to being available via inclusion of the <~~experimental/ranges/~~ranges> header, the customization point objects in 29.4 are available when <~~experimental/ranges/~~iterator> is included.

29.4.1 `begin` [range.access.begin]

1 The name `begin` denotes a customization point object ([customization.point.object]). The expression `ranges::begin(E)` for some subexpression `E` is expression-equivalent to:

(1.1) — `ranges::begin(static_cast<const T&>(E))` if `E` is an rvalue of type `T`. This usage is deprecated. [*Note:* This deprecated usage exists so that `ranges::begin(E)` behaves similarly to `std::begin(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]

(1.2) — ~~Otherwise,~~ `(E) + 0` if `E` ~~has~~is an lvalue of array type ([basic.compound]).

(1.3) — Otherwise, if `E` is an lvalue, *`DECAY_COPY`*`((E).begin())` if it is a valid expression and its type `I` models `Iterator`.~~meets the syntactic requirements of `Iterator<I>`. If `Iterator` is not satisfied, the program is ill-formed with no diagnostic required.~~

(1.4) — Otherwise, *`DECAY_COPY`*`(begin(E))` if it is a valid expression and its type `I` ~~meets the syntactic requirements of `Iterator<I>`~~ models `Iterator` with overload resolution performed in a context that includes the declarations:

```
template<class T> void begin(autoT&&) = delete;
template<class T> void begin(initializer_list<T>&&) = delete;
```

and does not include a declaration of `ranges::begin`. ~~If `Iterator` is not satisfied, the program is ill-formed with no diagnostic required.~~

(1.5) — Otherwise, `ranges::begin(E)` is ill-formed.

2 [*Note:* Whenever `ranges::begin(E)` is a valid expression, its type ~~satisfies~~ models `Iterator`. — *end note*]

29.4.2 `end` [range.access.end]

1 The name `end` denotes a customization point object ([customization.point.object]). The expression `ranges::end(E)` for some subexpression `E` is expression-equivalent to:

(1.1) — `ranges::end(static_cast<const T&>(E))` if `E` is an rvalue of type `T`. This usage is deprecated. [*Note:* This deprecated usage exists so that `ranges::end(E)` behaves similarly to `std::end(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]

(1.2) — ~~Otherwise,~~ `(E) + extent_v<T>`~~`::value`~~ if `E` ~~has~~ is an lvalue of array type ([basic.compound]) `T`.

(1.3) — Otherwise, if `E` is an lvalue, *`DECAY_COPY`*`((E).end())` if it is a valid expression and its type `S` models ~~meets the syntactic requirements of~~ `Sentinel<`~~`S,`~~`decltype(ranges::begin(E))>`. ~~If `Sentinel` is not satisfied, the program is ill-formed with no diagnostic required.~~

(1.4) — Otherwise, *`DECAY_COPY`*`(end(E))` if it is a valid expression and its type `S` ~~meets the syntactic requirements of~~ models `Sentinel<`~~`S,`~~`decltype(ranges::begin(E))>` with overload resolution performed in a context that includes the declarations:

```
template<class T> void end(autoT&&) = delete;
template<class T> void end(initializer_list<T>&&) = delete;
```

and does not include a declaration of `ranges::end`. ~~If `Sentinel` is not satisfied, the program is ill-formed with no diagnostic required.~~

(1.5) — Otherwise, `ranges::end(E)` is ill-formed.

2 [*Note:* Whenever `ranges::end(E)` is a valid expression, the types of `ranges::end(E)` and `ranges::begin(E)` satisfy `Sentinel`. — *end note*]

29.4.3 `cbegin` [range.access.cbegin]

1 The name `cbegin` denotes a customization point object ([customization.point.object]). The expression `ranges::cbegin(E)` for some subexpression `E` of type `T` is expression-equivalent to:

(1.1) — `ranges::begin(static_cast<const T&>(E))` if `E` is an lvalue.

(1.2) — Otherwise, `ranges::begin(static_cast<const T&&>(E))`.

2 Use of `ranges::cbegin(E)` with rvalue `E` is deprecated. [*Note:* This deprecated usage exists so that `ranges::cbegin(E)` behaves similarly to `std::cbegin(E)` ~~as defined in ISO/IEC 14882~~ when `E` is an rvalue. — *end note*]

3 [*Note:* Whenever `ranges::cbegin(E)` is a valid expression, its type ~~satisfies~~ models `Iterator`. — *end note*]

29.4.4 `cend` [range.access.cend]

1 The name `cend` denotes a customization point object ([customization.point.object]). The expression `ranges::cend(E)` for some subexpression `E` of type `T` is expression-equivalent to:

(1.1) — `ranges::end(static_cast<const T&>(E))` if `E` is an lvalue.

(1.2) — Otherwise, `ranges::end(static_cast<const T&&>(E))`.

2 Use of `ranges::cend(E)` with rvalue `E` is deprecated. [*Note:* This deprecated usage exists so that `ranges::cend(E)` behaves similarly to `std::cend(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]

3 [*Note:* Whenever `ranges::cend(E)` is a valid expression, the types of `ranges::cend(E)` and `ranges::cbegin(E)` satisfy `Sentinel`. — *end note*]

29.4.5 `rbegin` [range.access.rbegin]

1 The name `rbegin` denotes a customization point object ([customization.point.object]). The expression `ranges::rbegin(E)` for some subexpression `E` is expression-equivalent to:

(1.1) — `ranges::rbegin(static_cast<const T&>(E))` if `E` is an rvalue of type `T`. This usage is deprecated. [*Note:* This deprecated usage exists so that `ranges::rbegin(E)` behaves similarly to `std::rbegin(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]

(1.2) — ~~Otherwise~~ If `E` is an lvalue, *`DECAY_COPY`*`((E).rbegin())` if it is a valid expression and its type `I` models `Iterator` ~~`meets the syntactic requirements of Iterator<I>. If Iterator is not satisfied, the program is ill-formed with no diagnostic required`~~.

(1.3) — Otherwise, *`DECAY_COPY`*`(rbegin(E))` if it is a valid expression and its type `I` models `Iterator` with overload resolution performed in a context that includes the declaration:

```
template<class T> void rbegin(T&&) = delete;
```

and does not include a declaration of `ranges::rbegin`.

(1.4) — Otherwise, `make_reverse_iterator(ranges::end(E))` if both `ranges::begin(E)` and `ranges::end(E)` are valid expressions of the same type `I` which models ~~meets the syntactic requirements of~~ `BidirectionalIterator`~~`<I>`~~ (28.3.4.11).

(1.5) — Otherwise, `ranges::rbegin(E)` is ill-formed.

2 [*Note:* Whenever `ranges::rbegin(E)` is a valid expression, its type ~~satisfies~~ models `Iterator`. — *end note*]

29.4.6 `rend` [range.access.rend]

1 The name `rend` denotes a customization point object ([customization.point.object]). The expression `ranges::rend(E)` for some subexpression `E` is expression-equivalent to:

(1.1) — `ranges::rend(static_cast<const T&>(E))` if `E` is an rvalue of type `T`. This usage is deprecated. [*Note:* This deprecated usage exists so that `ranges::rend(E)` behaves similarly to `std::rend(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]

(1.2) — ~~Otherwise~~ If `E` is an lvalue, *`DECAY_COPY`*`((E).rend())` if it is a valid expression and its type `S` ~~meets the syntactic requirements of~~ models `Sentinel<`~~`S, `~~`decltype(ranges::rbegin(E))>`. ~~If `Sentinel` is not satisfied, the program is ill-formed with no diagnostic required.~~

(1.3) — Otherwise, *`DECAY_COPY`*`(rend(E))` if it is a valid expression and its type `S` models `Sentinel<decltype(ranges::rbegin(E))>` with overload resolution performed in a context that includes the declaration:

```
template<class T> void rend(T&&) = delete;
```

and does not include a declaration of `ranges::rend`.

(1.4) — Otherwise, `make_reverse_iterator(ranges::begin(E))` if both `ranges::begin(E)` and `ranges::end(E)` are valid expressions of the same type `I` which ~~meets the syntactic requirements of~~ models `BidirectionalIterator`~~`<I>`~~ (28.3.4.11).

(1.5) — Otherwise, `ranges::rend(E)` is ill-formed.

2 [*Note:* Whenever `ranges::rend(E)` is a valid expression, the types of `ranges::rend(E)` and `ranges::rbegin(E)` satisfy `Sentinel`. — *end note*]

29.4.7 `crbegin` [range.access.crbegin]

1 The name `crbegin` denotes a customization point object ([customization.point.object]). The expression `ranges::crbegin(E)` for some subexpression `E` of type `T` is expression-equivalent to:

(1.1) — `ranges::rbegin(static_cast<const T&>(E))` if `E` is an lvalue.

(1.2) — Otherwise, `ranges::rbegin(static_cast<const T&&>(E))`.

2 Use of `ranges::crbegin(E)` with rvalue `E` is deprecated. [*Note:* This deprecated usage exists so that `ranges::crbegin(E)` behaves similarly to `std::crbegin(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]

3 [*Note:* Whenever `ranges::crbegin(E)` is a valid expression, its type ~~satisfies~~ models `Iterator`. — *end note*]

29.4.8 `crend` [range.access.crend]

1 The name `crend` denotes a customization point object ([customization.point.object]). The expression `ranges::crend(E)` for some subexpression `E` of type `T` is expression-equivalent to:

(1.1) — `ranges::rend(static_cast<const T&>(E))` if `E` is an lvalue.

(1.2) — Otherwise, `ranges::rend(static_cast<const T&&>(E))`.

2 Use of `ranges::crend(E)` with rvalue `E` is deprecated. [*Note:* This deprecated usage exists so that `ranges::crend(E)` behaves similarly to `std::crend(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]

3 [*Note:* Whenever `ranges::crend(E)` is a valid expression, the types of `ranges::crend(E)` and `ranges::crbegin(E)` satisfy `Sentinel`. — *end note*]

29.5 Range primitives [range.primitives]

1 In addition to being available via inclusion of the `<`~~`experimental/ranges/`~~`ranges>` header, the customization point objects in 29.5 are available when `<`~~`experimental/ranges/`~~`iterator>` is included.

29.5.1 `size` [range.primitives.size]

1 The name `size` denotes a customization point object ([customization.point.object]). The expression `ranges::size(E)` for some subexpression `E` with type `T` is expression-equivalent to:

(1.1) — *`DECAY_COPY`*`(extent_v<T>`~~`::value`~~`)` if `T` is an array type ([basic.compound]).

(1.2) — Otherwise, *`DECAY_COPY`*`(`~~`static_cast<const T&>(E)`~~`.size())` if it is a valid expression and its type `I` ~~satisfies~~ models `Integral`~~`<I>`~~ and `disable_sized_range<remove_cvref_t<T>>` (29.6.3) is `false`.

(1.3) — Otherwise, *`DECAY_COPY`*(`size(`~~`static_cast<const T&>(E)`~~`))` if it is a valid expression and its type `I` ~~satisfies~~ models `Integral`~~`<I>`~~ with overload resolution performed in a context that includes the declaration:

`template<class T> void size(`~~`const auto`~~`T&&) = delete;`

and does not include a declaration of `ranges::size`, and `disable_sized_range<remove_cvref_t<T>>` is `false`.

(1.4) — Otherwise, *`DECAY_COPY`*(`ranges::`~~`c`~~`end(E) - ranges::`~~`c`~~`begin(E))`, except that `E` is only evaluated once, if it is a valid expression and the types `I` and `S` of `ranges::`~~`c`~~`begin(E)` and `ranges::`~~`c`~~`end(E)` ~~meet the syntactic requirements of~~ model `SizedSentinel<S, I>` (28.3.4.7) and `ForwardIterator<I>`. ~~If `SizedSentinel` and `ForwardIterator` are not satisfied, the program is ill-formed with no diagnostic required.~~

(1.5) — Otherwise, `ranges::size(E)` is ill-formed.

2 [*Note:* Whenever `ranges::size(E)` is a valid expression, its type ~~satisfies~~ models `Integral`. — *end note*]

29.5.2 `empty` [range.primitives.empty]

1 The name `empty` denotes a customization point object ([customization.point.object]). The expression `ranges::empty(E)` for some subexpression `E` is expression-equivalent to:

(1.1) — `bool((E).empty())` if it is a valid expression.

(1.2) — Otherwise, `ranges::size(E) == 0` if it is a valid expression.

(1.3) — Otherwise, `bool(ranges::begin(E) == ranges::end(E))`, except that `E` is only evaluated once, if it is a valid expression and the type of `ranges::begin(E)` ~~satisfies~~ models `ForwardIterator`.

(1.4) — Otherwise, `ranges::empty(E)` is ill-formed.

2 [*Note:* Whenever `ranges::empty(E)` is a valid expression, it has type `bool`. — *end note*]

29.5.3 `data` [range.primitives.data]

1 The name `data` denotes a customization point object ([customization.point.object]). The expression `ranges::data(E)` for some subexpression `E` is expression-equivalent to:

(1.1) — `ranges::data(static_cast<const T&>(E))` if `E` is an rvalue of type `T`. This usage is deprecated. [*Note:* This deprecated usage exists so that `ranges::data(E)` behaves similarly to `std::data(E)` as defined in the C++Working Paper when `E` is an rvalue. — *end note*]

(1.2) — ~~Otherwise~~ If `E` is an lvalue, *`DECAY_COPY`*(`(E).data()`) if it is a valid expression of pointer to object type.

(1.3) — Otherwise, `ranges::begin(E)` if it is a valid expression of pointer to object type.

(1.4) — Otherwise, if `ranges::begin(E)` is a valid expression whose type models `ContiguousIterator`,

```
ranges::begin(E) == ranges::end(E) ? nullptr : addressof(*ranges::begin(E))
```

except that `E` is evaluated only once.

(1.5) — Otherwise, `ranges::data(E)` is ill-formed.

2 [*Note:* Whenever `ranges::data(E)` is a valid expression, it has pointer to object type. — *end note*]

29.5.4 `cdata` [range.primitives.cdata]

1 The name `cdata` denotes a customization point object ([customization.point.object]). The expression `ranges::cdata(E)` for some subexpression `E` of type `T` is expression-equivalent to:

(1.1) — `ranges::data(static_cast<const T&>(E))` if `E` is an lvalue.

(1.2) — Otherwise, `ranges::data(static_cast<const T&&>(E))`.

2 Use of `ranges::cdata(E)` with rvalue `E` is deprecated. [*Note:* This deprecated usage exists so that `ranges::cdata(E)` has behavior consistent with `ranges::data(E)` when `E` is an rvalue. — *end note*]

3 [*Note:* Whenever `ranges::cdata(E)` is a valid expression, it has pointer to object type. — *end note*]

29.6 Range requirements [range.requirements]

29.6.1 General [range.requirements.general]

1 Ranges are an abstraction of containers that allow a C++program to operate on elements of data structures uniformly. In their simplest form, a range object is one on which one can call **begin** and **end** to get an iterator (28.3.4.5) and a sentinel (28.3.4.6). To be able to construct template algorithms and range adaptors that work correctly and efficiently on different types of sequences, the library formalizes not just the interfaces but also the semantics and complexity assumptions of ranges.

2 This document defines three fundamental categories of ranges based on the syntax and semantics supported by each: *range*, *sized range* and *view*, as shown in Table 90.

Table 90 — Relations among range categories

Sized Range ↘

Range

View ↗

3 The `Range` concept requires only that `begin` and `end` return an iterator and a sentinel. The `SizedRange` concept refines `Range` with the requirement that the number of elements in the range can be determined in constant time using the `size` function. The `View` concept specifies requirements on a `Range` type with constant-time copy and assign operations.

4 In addition to the three fundamental range categories, this document defines a number of convenience refinements of `Range` that group together requirements that appear often in the concepts and algorithms. ~~*Bounded ranges*~~*Common ranges* are ranges for which `begin` and `end` return objects of the same type. *Random access ranges* are ranges for which `begin` returns a type that ~~satisfies~~ models `RandomAccessIterator` (28.3.4.12). The range categories *bidirectional ranges*, *forward ranges*, *input ranges*, and *output ranges* are defined similarly.

29.6.2 Ranges [range.range]

1 The `Range` concept defines the requirements of a type that allows iteration over its elements by providing a `begin` iterator and an `end` sentinel. [*Note:* Most algorithms requiring this concept simply forward to an `Iterator`-based algorithm by calling `begin` and `end`. — *end note*]

```
template<class T>
concept ~~bool Range~~ range-impl = // exposition only
  requires(T&& t) {
    ranges::begin(std::forward<T>(t)); // not necessarily equality-preserving (see below)
    ranges::end(std::forward<T>(t));
  };

template<class T>
concept Range = range-impl<T&>;

template<class T>
concept forwarding-range = // exposition only
  Range<T> && range-impl<T>;
```

2 Given an ~~lvalue `t` of type `remove_reference_t<T>`, `Range<T>` is satisfied~~ expression `E` such that `decltype((E))` is `T`, `T` models *`range-impl`* only if

(2.1) — [`ranges::begin(`~~`t`~~`E)`, `ranges::end(`~~`t`~~`E)`) denotes a range.

(2.2) — Both `ranges::begin(`~~`t`~~`E)` and `ranges::end(`~~`t`~~`E)` are amortized constant time and non-modifying. [*Note:* `ranges::begin(`~~`t`~~`E)` and `ranges::end(`~~`t`~~`E)` do not require implicit expression variations ([concepts.general.equality]). — *end note*]

(2.3) — If ~~`iterator_t<T>`~~ the type of `ranges::begin(E)` ~~satisfies~~ models `ForwardIterator`, `ranges::begin(`~~`t`~~`E)` is equality-preserving.

3 Given an expression E such that `decltype((E))` is `T`, `T` models *`forwarding-range`* only if

(3.1) — The expressions `ranges::begin(E)` and `ranges::begin(static_cast<T&>(E))` are expression-equivalent.

(3.2) — The expressions `ranges::end(E)` and `ranges::end(static_cast<T&>(E))` are expression-equivalent.

4 [*Note:* Equality preservation of both `ranges::begin` and `ranges::end` enables passing a `Range` whose iterator type ~~satisfies~~ models `ForwardIterator` to multiple algorithms and making multiple passes over the range by repeated calls to `ranges::begin` and `ranges::end`. Since `ranges::begin` is not required to be equality-preserving when the return type does not satisfy `ForwardIterator`, repeated calls might not return equal values or might not be well-defined; `ranges::begin` should be called at most once for such a range. — *end note*]

29.6.3 Sized ranges [range.sized]

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

```
template<class T>
concept ~~bool~~ SizedRange =
  Range<T> &&
  !disable_sized_range<remove_cv~~_t<remove_~~ref~~erence~~_t<T>~~>~~> &&
  requires(T& t) {
    { ranges::size(t) } -> ConvertibleTo<iter_difference~~_type~~_t<iterator_t<T>>>;
  };
```

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

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

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

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

29.6.4 Views [range.view]

1 The `View` concept specifies the requirements of a `Range` type that has constant time copy, move and assignment operators; that is, the cost of these operations is not proportional to the number of elements in the `View`.

2 [*Example:* Examples of `View`s are:

(2.1) — A `Range` type that wraps a pair of iterators.

(2.2) — A `Range` type that holds its elements by `shared_ptr` and shares ownership with all its copies.

(2.3) — A `Range` type that generates its elements on demand.

A container (Clause 27) is not a `View` since copying the container copies the elements, which cannot be done in constant time. — *end example*]

```
template<class T>
inline constexpr bool view-predicate // exposition only
  = see below;

template<class T>
concept ~~bool~~ View =
  Range<T> &&
  Semiregular<T> &&
  view-predicate<T>;
```

3 Since the difference between `Range` and `View` is largely semantic, the two are differentiated with the help of the `enable_view` trait. Users may specialize `enable_view` to derive from `true_type` or `false_type`.

4 For a type `T`, the value of *`view-predicate`*`<T>` shall be:

(4.1) — If the *qualified-id* `enable_view<T>::type` ~~has a member type `type`~~ is valid and denotes a type ([temp.deduct]), `enable_view<T>::type::value`;

(4.2) — Otherwise, if `DerivedFrom<T, view_base>` is `true` ~~derived from `view_base`~~, `true`;

(4.3) — Otherwise, if `T` is an ~~instantiation~~ specialization of class template `initializer_list` ([support.initlist]), `set` ([set]), `multiset` ([multiset]), `unordered_set` ([unord.set]), or `unordered_multiset` ([unord.multiset]), `false`;

(4.4) — Otherwise, if both `T` and `const T` satisfy `Range` and `iter_reference_t<iterator_t<T>>` is not the same type as `iter_reference_t<iterator_t<const T>>`, `false`; [*Note:* Deep `const`-ness implies element ownership, whereas shallow `const`-ness implies reference semantics. — *end note*]

(4.5) — Otherwise, `true`.

29.6.5 Common ranges [range.common]

[Editor's note: We've renamed "`BoundedRange`" to "`CommonRange`". The authors believe this is a better name than "`ClassicRange`", which LEWG weakly preferred. The reason is that the iterator and sentinel of a Common range have the same type in *common*. A non-Common range can be turned into a Common range with the help of `view::common`.]

1 The ~~`BoundedRange`~~`CommonRange` concept specifies requirements of a `Range` type for which `begin` and `end` return objects of the same type. [*Note:* The standard containers ([containers]) satisfy ~~`BoundedRange`~~`CommonRange`. — *end note*]

```
template<class T>
concept ~~bool~~ ~~BoundedRange~~CommonRange =
  Range<T> && Same<iterator_t<T>, sentinel_t<T>>;
```

29.6.6 Input ranges [range.input]

1 The `InputRange` concept specifies requirements of a `Range` type for which `begin` returns a type that ~~satisfies~~ models `InputIterator` (28.3.4.8).

```
template<class T>
concept ~~bool~~ InputRange =
  Range<T> && InputIterator<iterator_t<T>>;
```

29.6.7 Output ranges [range.output]

1 The `OutputRange` concept specifies requirements of a `Range` type for which `begin` returns a type that ~~satisfies~~ models `OutputIterator` (28.3.4.9).

```
template<class R, class T>
concept ~~bool~~ OutputRange =
  Range<R> && OutputIterator<iterator_t<R>, T>;
```

29.6.8 Forward ranges [range.forward]

1 The `ForwardRange` concept specifies requirements of an `InputRange` type for which `begin` returns a type that ~~satisfies~~ models `ForwardIterator` (28.3.4.10).

```
template<class T>
concept ~~bool~~ ForwardRange =
  InputRange<T> && ForwardIterator<iterator_t<T>>;
```

29.6.9 Bidirectional ranges [range.bidirectional]

1 The `BidirectionalRange` concept specifies requirements of a `ForwardRange` type for which `begin` returns a type that ~~satisfies~~ models `BidirectionalIterator` (28.3.4.11).

```
template<class T>
concept ~~bool~~ BidirectionalRange =
  ForwardRange<T> && BidirectionalIterator<iterator_t<T>>;
```

29.6.10 Random access ranges [range.random.access]

1 The `RandomAccessRange` concept specifies requirements of a `BidirectionalRange` type for which `begin` returns a type that ~~satisfies~~ models `RandomAccessIterator` (28.3.4.12).

```
template<class T>
concept ~~bool~~ RandomAccessRange =
  BidirectionalRange<T> && RandomAccessIterator<iterator_t<T>>;
```

29.6.11 Contiguous ranges [range.contiguous]

1 The `ContiguousRange` concept specifies requirements of a `RandomAccessRange` type for which `begin` returns a type that ~~satisfies~~ models `ContiguousIterator` (28.3.4.13).

```
template<class T>
concept ContiguousRange =
  RandomAccessRange<T> && ContiguousIterator<iterator_t<T>> &&
  requires(T& t) {
    ranges::data(t);
    requires Same<decltype(ranges::data(t)), add_pointer_t<iter_reference_t<iterator_t<T>>>>;
  };
```

29.6.12 Viewable ranges [range.viewable]

1 The `ViewableRange` concept specifies the requirements of a `Range` type that can be converted to a `View` safely.

```
template<class T>
concept ViewableRange =
  Range<T> && (~~is_lvalue_reference_v<T>~~forwarding-range<T> || View<decay_t<T>>); ~~// see below~~
```

2 ~~There need be no subsumption relationship between `ViewableRange<T>` and `is_lvalue_reference_v<T>`.~~

29.7 Range utilities [range.utility]

1 The components in this section are general utilities for representing and manipulating ranges.

29.7.1 Class template dangling [range.dangling]

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

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template<CopyConstructible T>
  class dangling {
  public:
    constexpr dangling() requires DefaultConstructible<T>;
    constexpr dangling(T t);
    constexpr T get_unsafe() const;
  private:
    T value; // exposition only
  };

  template<Range R>
  using safe_iterator_t =
    conditional_t<is_lvalue_reference<R>::value,
      iterator_t<R>,
      dangling<iterator_t<R>>>;
}}}}
```

29.7.1.1 dangling operations [range.dangling.ops]

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

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

```
constexpr dangling(T t);
```

2 *Effects:* Constructs a `dangling`, initializing `value` with `t`.

```
constexpr T get_unsafe() const;
```

3 *Returns:* `value`.

29.7.2 View interface [range.view_interface]

1 The class template `view_interface` is a helper for defining `View`-like types that offer a container-like interface. It is parameterized with the type that inherits from it.

```
namespace std::ranges { namespace ranges {
  template<Range R>
  struct range-common-iterator-impl { // exposition only
    using type = common_iterator<iterator_t<R>, sentinel_t<R>>;
  };
  template<CommonRange R>
  struct range-common-iterator-impl<R> \{ // exposition only
    using type = iterator_t<R>;
  };
  template<Range R>
    using range-common-iterator = // exposition only
      typename range-common-iterator-impl<R>::type;

  template<class D>
    requires is_class_v<D>
  class view_interface : public view_base {
  private:
    constexpr D& derived() noexcept { // exposition only
      return static_cast<D&>(*this);
    }
    constexpr const D& derived() const noexcept { // exposition only
      return static_cast<const D&>(*this);
    }
  public:
    constexpr bool empty() const requires ForwardRange<const D>;
    constexpr explicit operator bool() const
      requires requires { ranges::empty(derived()); };

    constexpr auto data()
      requires ContiguousIterator<iterator_t<D>>;
    constexpr auto data() const
      requires Range<const D> && ContiguousIterator<iterator_t<const D>>;
      requires RandomAccessRange<const D> && is_pointer_v<iterator_t<const D>>;

    constexpr auto size() const requires ForwardRange<const D> &&
      SizedSentinel<sentinel_t<const D>, iterator_t<const D>>;

    constexpr decltype(auto) front() requires ForwardRange<D>;
    constexpr decltype(auto) front() const requires ForwardRange<const D>;

    constexpr decltype(auto) back()
      requires BidirectionalRange<D> && CommonRange<D>;
    constexpr decltype(auto) back() const
      requires BidirectionalRange<const D> && CommonRange<const D>;

    template<RandomAccessRange R = D>
      constexpr decltype(auto) operator[](iter_difference_type_t<iterator_t<R>> n);
    template<RandomAccessRange R = const D>
      constexpr decltype(auto) operator[](iter_difference_type_t<iterator_t<R>> n) const;

    template<RandomAccessRange R = D>
        requires SizedRange<R>
      constexpr decltype(auto) at(ifference_type_t<iterator_t<R>> n);
    template<RandomAccessRange R = const D>
        requires SizedRange<R>
      constexpr decltype(auto) at(ifference_type_t<iterator_t<R>> n) const;
```

```
      template<ForwardRange C>
          requires !View<C> &&
            ConvertibleTo<iter_reference_t<iterator_t<const D>>,
              iter_value_type_t<iterator_t<C>>> &&
            Constructible<C, range-common-iterator<const D>,
              range-common-iterator<const D>>
        operator C() const;
    };
  }}
```

2 The template parameter for `view_interface` may be an incomplete type.

29.7.2.1 `view_interface` accessors [range.view__interface.accessors]

```
constexpr bool empty() const requires ForwardRange<const D>;
```

1 *Effects:* Equivalent to: `return ranges::begin(derived()) == ranges::end(derived());`

```
constexpr explicit operator bool() const
  requires requires { ranges::empty(derived()); };
```

2 *Effects:* Equivalent to: `return !ranges::empty(derived());`

```
constexpr auto data()
  requires ContiguousIterator<iterator_t<D>>;
constexpr auto data() const
  requires Range<const D> && ContiguousIterator<iterator_t<const D>>;
  requires RandomAccessRange<const D> && is_pointer_v<iterator_t<const D>>;
```

3 *Effects:* Equivalent to: `return ranges::begin(derived());`

```
constexpr auto size() const requires ForwardRange<const D> &&
SizedSentinel<sentinel_t<const D>, iterator_t<const D>>;
```

4 *Effects:* Equivalent to: `return ranges::end(derived()) - ranges::begin(derived());`

```
constexpr decltype(auto) front() requires ForwardRange<D>;
constexpr decltype(auto) front() const requires ForwardRange<const D>;
```

5 *Requires:* `!empty()`.

6 *Effects:* Equivalent to: `return *ranges::begin(derived());`

```
constexpr decltype(auto) back()
  requires BidirectionalRange<D> && CommonRange<D>;
constexpr decltype(auto) back() const
  requires BidirectionalRange<const D> && CommonRange<const D>;
```

7 *Requires:* `!empty()`.

8 *Effects:* Equivalent to: `return *ranges::prev(ranges::end(derived()));`

```
template<RandomAccessRange R = D>
constexpr decltype(auto) operator[](iter_difference_type_t<iterator_t<R>> n);
template<RandomAccessRange R = const D>
constexpr decltype(auto) operator[](iter_difference_type_t<iterator_t<R>> n) const;
```

9 ~~*Requires:* ranges::begin(derived()) + n is well-formed.~~

10 *Effects:* Equivalent to: `return ranges::begin(derived())[n];`

```
template<RandomAccessRange R = D>
  requires SizedRange<R>
constexpr decltype(auto) at(difference_type_t<iterator_t<R>> n);
template<RandomAccessRange R = const D>
  requires SizedRange<R>
constexpr decltype(auto) at(difference_type_t<iterator_t<R>> n) const;
```

11 ~~*Effects:* Equivalent to: `return derived()[n];`.~~

12 ~~*Throws:* out_of_range if `n < 0 || n >= ranges::size(derived())`.~~

```
template<ForwardRange C>
  requires !View<C> &&
    ConvertibleTo<iter_reference_t<iterator_t<const D>>,
      iter_value_type_t<iterator_t<C>>> &&
    Constructible<C, range-common-iterator<const D>,
      range-common-iterator<const D>>
operator C() const;
```

13 *Effects:* Equivalent to:

```
using I = range-common-iterator<R>;
return C(I{ranges::begin(derived())}, I{ranges::end(derived())});
```

29.7.3 Sub-ranges [range.subranges]

1 The `subrange` class template bundles together an iterator and a sentinel into a single object that ~~satisfies~~ models the `View` concept. Additionally, it ~~satisfies~~ models the `SizedRange` concept when the final template parameter is `subrange_kind::sized`.

29.7.3.1 `subrange` [range.subrange]

```
namespace std { namespace ranges {
    template<class T>
    concept pair-like = // exposition only
      requires(T t) {
        { tuple_size<T>::value } -> Integral;
        requires tuple_size<T>::value == 2;
        typename tuple_element_t<0, T>;
        typename tuple_element_t<1, T>;
        { get<0>(t) } -> const tuple_element_t<0, T>&;
        { get<1>(t) } -> const tuple_element_t<1, T>&;
      };

    template<class T, class U, class V>
    concept pair-like-convertible-to = // exposition only
      !Range<T> && pair-like<decay_t<T>> &&
      requires(T&& t) {
        { get<0>(std::forward<T>(t)) } -> ConvertibleTo<U>;
        { get<1>(std::forward<T>(t)) } -> ConvertibleTo<V>;
      };

    template<class T, class U, class V>
    concept pair-like-convertible-from = // exposition only
      !Range<T> && Same<T, decay_t<T>> && pair-like<T> &&
      Constructible<T, U, V>;

    template<class T>
    concept iterator-sentinel-pair = // exposition only
      !Range<T> && Same<T, decay_t<T>> && pair-like<T> &&
      Sentinel<tuple_element_t<1, T>, tuple_element_t<0, T>>;

    template<class T, class U>
    concept not-same-as = // exposition only
      !Same<remove_cvref_t<T>, remove_cvref_t<U>>;

    template<Iterator I, Sentinel<I> S = I, subrange_kind K =
        see belowSizedSentinel<S, I> ? subrange_kind::sized : subrange_kind::unsized>
      requires K == subrange_kind::sized || !SizedSentinel<S, I>
    class subrange : public view_interface<subrange<I, S, K>> {
    private:
      static constexpr bool StoreSize =
        K == subrange_kind::sized && !SizedSentinel<S, I>; // exposition only
      I begin_ {}; // exposition only
      S end_ {}; // exposition only
      iter_difference_type_t<I> size_ = 0; // exposition only; only present when StoreSize is true
```

```
public:
  using iterator = I;
  using sentinel = S;

  subrange() = default;

  constexpr subrange(I i, S s) requires !StoreSize;

  constexpr subrange(I i, S s, iter_difference~~_type~~_t<I> n)
    requires K == subrange_kind::sized;

  ~~template<ConvertibleTo<I> X, ConvertibleTo<S> Y, subrange_kind Z>~~
  ~~constexpr subrange(subrange<X, Y, Z> r)~~
    ~~requires !StoreSize || Z == subrange_kind::sized;~~

  ~~template<ConvertibleTo<I> X, ConvertibleTo<S> Y, subrange_kind Z>~~
  ~~constexpr subrange(subrange<X, Y, Z> r, difference_type_t<I> n)~~
    ~~requires K == subrange_kind::sized;~~

  template<not-same-as<subrange> R>
    requires forwarding-range<R> &&
      ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>
  constexpr subrange(R&& r) requires !StoreSize || SizedRange<R>;

  template<forwarding-range R>
    requires ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>
  constexpr subrange(R&& r, iter_difference_t<I> n)
    requires K == subrange_kind::sized;

  template<not-same-as<subrange> PairLike>
    requires pair-like-convertible-to<PairLike, I, S>
  constexpr subrange(PairLike&& r) requires !StoreSize;

  template<pair-like-convertible-to<I, S> PairLike>
  constexpr subrange(PairLike&& r, iter_difference~~_type~~_t<I> n)
    requires K == subrange_kind::sized;

  ~~template<not-name-as<subrange> R>~~
    ~~requires Range<R> && ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>~~
  ~~constexpr subrange(R& r) requires !StoreSize || SizedRange<R>;~~

  template<not-same-as<subrange> PairLike>
    requires pair-like-convertible-from<PairLike, const I&, const S&>
  constexpr operator PairLike() const;

  constexpr I begin() const;
  constexpr S end() const;

  constexpr bool empty() const;
  constexpr iter_difference~~_type~~_t<I> size() const
    requires K == subrange_kind::sized;

  [[nodiscard]] constexpr subrange next(iter_difference~~_type~~_t<I> n = 1) const;
  [[nodiscard]] constexpr subrange prev(iter_difference~~_type~~_t<I> n = 1) const
    requires BidirectionalIterator<I>;
  constexpr subrange& advance(iter_difference~~_type~~_t<I> n);

  friend constexpr I begin(subrange&& r) { return r.begin(); }
  friend constexpr S end(subrange&& r) { return r.end(); }
};

template<Iterator I, Sentinel<I> S>
subrange(I, S, iter_difference~~_type~~_t<I>) ->
  subrange<I, S, subrange_kind::sized>;
```

```
    template<iterator-sentinel-pair P>
    subrange(P) -> subrange<tuple_element_t<0, P>, tuple_element_t<1, P>>;

    template<iterator-sentinel-pair P>
    subrange(P, iter_difference_type_t<tuple_element_t<0, P>>) ->
      subrange<tuple_element_t<0, P>, tuple_element_t<1, P>, subrange_kind::sized>;

    template<Iterator I, Sentinel<I> S, subrange_kind K>
    subrange(subrange<I, S, K>, difference_type_t<I>) ->
      subrange<I, S, subrange_kind::sized>;

    template<Range R>
    subrange(R&) -> subrange<iterator_t<R>, sentinel_t<R>>;

    template<SizedRange R>
    subrange(R&) -> subrange<iterator_t<R>, sentinel_t<R>, subrange_kind::sized>;

    template<forwarding-range R>
    subrange(R&&) -> subrange<iterator_t<R>, sentinel_t<R>>;

    template<forwarding-range R>
      requires SizedRange<R>
    subrange(R&&) -> subrange<iterator_t<R>, sentinel_t<R>, subrange_kind::sized>;

    template<forwarding-range R>
    subrange(R&&, iter_difference_t<iterator_t<R>>) ->
      subrange<iterator_t<R>, sentinel_t<R>, subrange_kind::sized>;

    template<size_t N, class I, class S, subrange_kind K>
      requires N < 2
    constexpr auto get(const subrange<I, S, K>& r);

    template<forwarding-range R>
      using safe_subrange_t = subrange<iterator_t<R>>;
  }

  using ranges::get;
}
```

1 The default value for `subrange`'s third (non-type) template parameter is:

(1.1) — If `SizedSentinel<S, I>` is satisfied, `subrange_kind::sized`.

(1.2) — Otherwise, `subrange_kind::unsized`.

29.7.3.1.1 subrange constructors [range.subrange.ctor]

```
constexpr subrange(I i, S s) requires !StoreSize;
```

1 *Effects:* Initializes `begin_` with `i` and `end_` with `s`.

```
constexpr subrange(I i, S s, iter_difference_type_t<I> n)
  requires K == subrange_kind::sized;
```

2 *Requires:* `n == ranges::distance(i, s)`.

3 *Effects:* Initializes `begin_` with `i`, `end_` with `s`. If `StoreSize` is `true`, initializes `size_` with `n`.

```
template<ConvertibleTo<I> X, ConvertibleTo<S> Y, subrange_kind Z>
constexpr subrange(subrange<X, Y, Z> r)
  requires !StoreSize || Z == subrange_kind::sized;
```

4 *Effects:* Equivalent to:

(4.1) — If `StoreSize` is `true`, `subrange{r.begin(), r.end(), r.size()}`.

(4.2) — Otherwise, `subrange{r.begin(), r.end()}`.

```
template<ConvertibleTo<I> X, ConvertibleTo<S> Y, subrange_kind Z>
constexpr subrange(subrange<X, Y, Z> r, difference_type_t<I> n)
  requires K == subrange_kind::sized;
```

5 *Effects:* Equivalent to `subrange{r.begin(), r.end(), n}`.

```
template<not-same-as<subrange> R>
  requires forwarding-range<R> &&
    ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>
constexpr subrange(R&& r) requires !StoreSize || SizedRange<R>;
```

6 *Effects:* Equivalent to:

(6.1) — If `StoreSize` is `true`, `subrange{ranges::begin(r), ranges::end(r), ranges::size(r)}`.

(6.2) — Otherwise, `subrange{ranges::begin(r), ranges::end(r)}`.

```
template<forwarding-range R>
  requires ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>
constexpr subrange(R&& r, difference_type_t<I> n)
  requires K == subrange_kind::sized;
```

7 *Effects:* Equivalent to `subrange{ranges::begin(r), ranges::end(r), n}`.

```
template<not-same-as<subrange> PairLike>
  requires pair-like-convertible-to<PairLike, I, S>
constexpr subrange(PairLike&& r) requires !StoreSize;
```

8 *Effects:* Equivalent to:

```
subrange{get<0>(std::forward<PairLike>(r)), get<1>(std::forward<PairLike>(r))}
```

```
template<pair-like-convertible-to<I, S> PairLike>
constexpr subrange(PairLike&& r, iter_difference_type_t<I> n)
  requires K == subrange_kind::sized;
```

9 *Effects:* Equivalent to:

```
subrange{get<0>(std::forward<PairLike>(r)), get<1>(std::forward<PairLike>(r)), n}
```

```
template<not-name-as<subrange> R>
  requires Range<R> && ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>
constexpr subrange(R& r) requires !StoreSize || SizedRange<R>;
```

10 *Effects:* Equivalent to:

(10.1) — If `StoreSize` is `true`, `subrange{ranges::begin(r), ranges::end(r), distance(r)}`.

(10.2) — Otherwise, `subrange{ranges::begin(r), ranges::end(r)}`.

29.7.3.1.2 subrange operators [range.subrange.ops]

```
template<not-same-as<subrange> PairLike>
  requires pair-like-convertible-from<PairLike, const I&, const S&>
constexpr operator PairLike() const;
```

1 *Effects:* Equivalent to: `return PairLike(begin_, end_);`

29.7.3.1.3 subrange accessors [range.subrange.accessors]

```
constexpr I begin() const;
```

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

```
constexpr S end() const;
```

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

```
constexpr bool empty() const;
```

3 *Effects:* Equivalent to: `return begin_ == end_;`

```
constexpr iter_difference_t<I> size() const
  requires K == subrange_kind::sized;
```

4 *Effects:* Equivalent to: `if constexpr(StoreSize) return size_; else return end_ - begin_;`

(4.1) — If `StoreSize` is `true`, `return size_;`.

(4.2) — Otherwise, `return end_ - begin_;`.

```
[[nodiscard]] constexpr subrange next(iter_difference_t<I> n = 1) const;
```

5 *Effects:* Equivalent to:

```
auto tmp = *this;
tmp.advance(n);
return tmp;
```

6 [*Note:* If `ForwardIterator<I>` is not satisfied, `next` may invalidate `*this`. — *end note*]

```
[[nodiscard]] constexpr subrange prev(iter_difference_t<I> n = 1) const
  requires BidirectionalIterator<I>;
```

7 *Effects:* Equivalent to:

```
auto tmp = *this;
tmp.advance(-n);
return tmp;
```

```
constexpr subrange& advance(iter_difference_t<I> n);
```

8 *Effects:* Equivalent to:

(8.1) — If `StoreSize` is `true`,

```
size_ -= n - ranges::advance(begin_, n, end_);
return *this;
```

(8.2) — Otherwise,

```
ranges::advance(begin_, n, end_);
return *this;
```

29.7.3.1.4 subrange non-member functions [range.subrange.nonmember]

```
template<size_t N, class I, class S, subrange_kind K>
  requires N < 2
constexpr auto get(const subrange<I, S, K>& r);
```

1 *Effects:* Equivalent to:

```
if constexpr (N == 0)
  return r.begin();
else
  return r.end();
```

29.8 Range adaptors [range.adaptors]

1 This section defines *range adaptors*, which are utilities that transform a `Range` into a `View` with custom behaviors. These adaptors can be chained to create pipelines of range transformations that evaluate lazily as the resulting view is iterated.

2 Range adaptors are declared in namespace `std::ranges::view`.

3 The bitwise or operator is overloaded for the purpose of creating adaptor chain pipelines. The adaptors also support function call syntax with equivalent semantics.

4 [*Example:*

```
vector<int> ints{0,1,2,3,4,5};
auto even = [](int i){ return 0 == i % 2; };
auto square = [](int i) { return i * i; };
for (int i : ints | view::filter(even) | view::transform(square)) {
  cout << i << ' '; // prints: 0 4 16
}
assert(ranges::equal(ints | view::filter(even), view::filter(ints, even)));
```

— *end example*]

29.8.1 Range adaptor objects [range.adaptor.object]

1 A *range adaptor closure object* is a unary function object that accepts a `ViewableRange` as an argument and returns a `View`. For a range adaptor closure object `C` and an expression `R` such that `decltype((R))` ~~satisfies~~ models `ViewableRange`, the following expressions are equivalent and return a `View`:

```
C(R)
R | C
```

Given an additional range adaptor closure object `D`, the expression `C | D` is well-formed and produces another range adaptor closure object such that the following two expressions are equivalent:

```
R | C | D
R | (C | D)
```

2 A *range adaptor object* is a customization point object ([customization.point.object]) that accepts a `ViewableRange` as its first argument and returns a `View`.

3 If the adaptor accepts only one argument, then it is a range adaptor closure object.

4 If the adaptor accepts more than one argument, then the following expressions are equivalent:

```
adaptor(rng, args...)
adaptor(args...)(rng)
rng | adaptor(args...)
```

In this case, *`adaptor`*`(args...)` is a range adaptor closure object.

29.8.2 Semiregular wrapper [range.adaptor.semiregular_wrapper]

1 Many of the types in this section are specified in terms of an exposition-only helper called *`semiregular`*`<T>`. This type behaves exactly like `optional<T>` with the following exceptions:

(1.1) — *`semiregular`*`<T>` constrains its argument with `CopyConstructible<T> && is_object_v<T>`.

(1.2) — If `T` models `DefaultConstructible`, the default constructor of *`semiregular`*`<T>` is equivalent to:

```
constexpr semiregular()
  noexcept(is_nothrow_default_constructible_v<T>::value) :  \placeholder{semiregular}in_place
```

(1.3) — If ~~the syntactic requirements of~~ `Assignable<T&, const T&>` is ~~are~~ not satisfied, the copy assignment operator is equivalent to:

```
constexpr semiregular& operator=(const semiregular& that)
  noexcept(is_nothrow_copy_constructible_v<T>::value)
{
  if (that) emplace(*that);
  else reset();
  return *this;
}
```

(1.4) — If ~~the syntactic requirements of~~ `Assignable<T&, T>` is ~~are~~ not satisfied, the move assignment operator is equivalent to:

```
constexpr semiregular& operator=(semiregular&& that)
  noexcept(is_nothrow_move_constructible_v<T>::value)
{
  if (that) emplace(std::move(*that));
  else reset();
  return *this;
}
```

29.8.3 Helper concepts [range.adaptor.helpers]

1 Many of the types in this ~~section~~ subclause are specified in terms of ~~an~~ the following exposition-only concepts: ~~Boolean variable template called *`simple-view`*`<T>`, defined as follows:~~

```
template<class R>
concept __simple-view simple-view =
  View<R> && View<const R> &&
  Same<iterator_t<R>, iterator_t<const R>> &&
  Same<sentinel_t<R>, sentinel_t<const R>>;
```

```
template<class R>
  constexpr bool simple-view = false;

template<__simple-view R>
  constexpr bool simple-view<R> = true;

template<InputIterator I>
concept has-arrow = is_pointer_v<I> || requires(I i) { i.operator->(); };
```

29.8.4 `view::all` [range.adaptors.all]

1 ~~The purpose of~~ `view::all` ~~is to~~ returns a `View` that includes all elements of ~~the~~ its `Range` argument ~~passed in~~.

2 The name `view::all` denotes a range adaptor object (29.8.1). The expression `view::all(E)` for some subexpression `E` is expression-equivalent to:

(2.1) — *`DECAY_COPY`*`(E)` if the decayed type of `E` ~~satisfies the concept~~ models `View`.

(2.2) — Otherwise, *`ref-view`*`{E}` if that expression is well-formed, where *`ref-view`* is the exposition-only `View` specified below.

(2.3) — Otherwise, `subrange{E}` if that expression is well-formed. ~~E is an lvalue and has a type that satisfies concept Range.~~

(2.4) — Otherwise, `view::all(E)` is ill-formed.

[*Note:* Whenever `view::all(E)` is a valid expression, it is a prvalue whose type ~~satisfies~~ models `View`. — *end note*]

29.8.4.1 *ref-view* [range.view.ref]

```
namespace std::ranges {
  template<Range Rng>
    requires std::is_object_v<Rng> && !View<Rng>
  class ref_view : public view_interface<ref_view<Rng>> {
  private:
    Rng* rng_ = nullptr; // exposition only
  public:
    constexpr ref_view() noexcept = default;
    constexpr ref_view(Rng& rng) noexcept;

    constexpr Rng& base() const;

    constexpr iterator_t<Rng> begin() const
      noexcept(noexcept(ranges::begin(*rng_)));
    constexpr sentinel_t<Rng> end() const
      noexcept(noexcept(ranges::end(*rng_)));

    constexpr bool empty() const
      noexcept(noexcept(ranges::empty(*rng_)))
      requires { ranges::empty(*rng_); };

    constexpr auto size() const
      noexcept(noexcept(ranges::size(*rng_)))
      requires SizedRange<Rng>;

    constexpr auto data() const
      noexcept(noexcept(ranges::data(*rng_)))
      requires ContiguousRange<Rng>;

    friend constexpr iterator_t<Rng> begin(ref_view&& r)
      noexcept(noexcept(r.begin()));
    friend constexpr sentinel_t<Rng> end(ref_view&& r)
      noexcept(noexcept(r.end()));
  };
}
```

29.8.4.1.1 *ref-view* operations [range.view.ref.ops]

```
constexpr ref_view(Rng& rng) noexcept;
```

1 *Effects:* Initializes `rng_` with `addressof(rng)`.

```
constexpr Rng& base() const;
```

2 *Returns:* `*rng_`.

3 *Throws:* Nothing.

```
constexpr iterator_t<Rng> begin() const
  noexcept(noexcept(ranges::begin(*rng_)));
friend constexpr iterator_t<Rng> begin(ref_view&& r)
  noexcept(noexcept(r.begin()));
```

4 *Effects:* Equivalent to: `return ranges::begin(*rng_);` or `return r.begin();`, respectively.

```
constexpr sentinel_t<Rng> end() const
  noexcept(noexcept(ranges::end(*rng_)));
friend constexpr sentinel_t<Rng> end(ref_view&& r)
  noexcept(noexcept(r.end()));
```

5 *Effects:* Equivalent to: `return ranges::end(*rng_);` or `return r.end();`, respectively.

```
constexpr bool empty() const
  noexcept(noexcept(ranges::empty(*rng_)));
```

6 *Effects:* Equivalent to: `return ranges::empty(*rng_);`

```
constexpr auto size() const
  noexcept(noexcept(ranges::size(*rng_)))
  requires SizedRange<Rng>;
```

7 *Effects:* Equivalent to: `return ranges::size(*rng_);`

```
constexpr auto data() const
  noexcept(noexcept(ranges::data(*rng_)))
  requires ContiguousRange<Rng>;
```

8 *Effects:* Equivalent to: `return ranges::data(*rng_);`

29.8.5 Class template `filter_view` [range.adaptors.filter_view]

1 ~~The purpose of~~ `filter_view` ~~is to~~ presents a ~~view~~`View` of an underlying sequence without the elements that fail to satisfy a predicate.

2 [*Example:*

```
vector<int> is{ 0, 1, 2, 3, 4, 5, 6 };
filter_view evens{is, [](int i) { return 0 == i % 2; }};
for (int i : evens)
  cout << i << ' '; // prints: 0 2 4 6
```

— end example]

```
namespace std::ranges { namespace ranges {
  template<InputRange R, IndirectUnaryPredicate<iterator_t<R>> Pred>
    requires View<R> && is_object_v<Pred>
  class filter_view : public view_interface<filter_view<R, Pred>> {
  private:
    R base_ {}; // exposition only
    semiregular<Pred> pred_; // exposition only

    class iterator; // exposition only
    class sentinel; // exposition only

  public:
    filter_view() = default;
    constexpr filter_view(R base, Pred pred);
```

```
    template<InputRange O>
      requires ViewableRange<O> && Constructible<R, all_view<O>>
    constexpr filter_view(O&& o, Pred pred);

    constexpr R base() const;

    constexpr iterator begin();
    constexpr sentinel end();
    constexpr iterator end() requires CommonRange<R>;
  };

  template<~~InputRange~~class R, ~~CopyConstructible~~class Pred>
    ~~requires IndirectUnaryPredicate<Pred, iterator_t<R>> && ViewableRange<R>~~
  filter_view(R&&, Pred) -> filter_view<all_view<R>, Pred>;
}~~}~~
```

29.8.5.1 filter_view operations [range.adaptors.filter_view.ops]

29.8.5.1.1 filter_view constructors [range.adaptors.filter_view.ctor]

```
constexpr filter_view(R base, Pred pred);
```

1 *Effects:* Initializes `base_` with `std::move(base)` and initializes `pred_` with `std::move(pred)`.

```
template<InputRange O>
  requires ViewableRange<O> && Constructible<R, all_view<O>>
constexpr filter_view(O&& o, Pred pred);
```

2 *Effects:* Initializes `base_` with `view::all(std::forward<O>(o))` and initializes `pred_` with `std::move(pred)`.

29.8.5.1.2 filter_view conversion [range.adaptors.filter_view.conv]

```
constexpr R base() const;
```

1 *Returns:* `base_`.

29.8.5.1.3 filter_view range begin [range.adaptors.filter_view.begin]

```
constexpr iterator begin();
```

1 *Effects:* Equivalent to:

```
return {*this, ranges::find_if(base_, ref(*pred_))};
```

2 *Remarks:* In order to provide the amortized constant time complexity required by the `Range` concept, this function caches the result within the `filter_view` for use on subsequent calls.

29.8.5.1.4 filter_view range end [range.adaptors.filter_view.end]

```
constexpr sentinel end();
```

1 *Returns:* `sentinel{*this}`.

```
constexpr iterator end() requires CommonRange<R>;
```

2 *Returns:* `iterator{*this, ranges::end(base_)}`.

29.8.5.2 Class template filter_view::iterator [range.adaptors.filter_view.iterator]

```
namespace std::ranges { ~~namespace ranges {~~
  template<class R, class Pred>
  class filter_view<R, Pred>::iterator {
  private:
    iterator_t<R> current_ {}; // exposition only
    filter_view* parent_ = nullptr; // exposition only
  public:
    using iterator_category = see below;
    using iterator_concept = see below;
    using value_type        = iter_value~~_type~~_t<iterator_t<R>>;
    using difference_type   = iter_difference~~_type~~_t<iterator_t<R>>;
```

```
      iterator() = default;
      constexpr iterator(filter_view& parent, iterator_t<R> current);

      constexpr iterator_t<R> base() const;
      constexpr iter_reference_t<iterator_t<R>> operator*() const;
      constexpr iterator_t<R> operator->() const
        requires has-arrow<iterator_t<R>>;

      constexpr iterator& operator++();
      constexpr void operator++(int);
      constexpr iterator operator++(int) requires ForwardRange<R>;

      constexpr iterator& operator--() requires BidirectionalRange<R>;
      constexpr iterator operator--(int) requires BidirectionalRange<R>;

      friend constexpr bool operator==(const iterator& x, const iterator& y)
        requires EqualityComparable<iterator_t<R>>;
      friend constexpr bool operator!=(const iterator& x, const iterator& y)
        requires EqualityComparable<iterator_t<R>>;

      friend constexpr iter_rvalue_reference_t<iterator_t<R>> iter_move(const iterator& i)
        noexcept(see belownoexcept(ranges::iter_move(i.current_)));
      friend constexpr void iter_swap(const iterator& x, const iterator& y)
        noexcept(see belownoexcept(ranges::iter_swap(x.current_, y.current_)))
        requires IndirectlySwappable<iterator_t<R>>;
    };
  }}
```

1 ~~The type `filter_view<R>::`~~`iterator::iterator_category` is defined as follows:

(1.1) — Let `C` denote the type `iterator_traits<iterator_t<R>>::iterator_category`.

(1.2) — If ~~`R` satisfies `BidirectionalRange<R>`~~ `DerivedFrom<C, bidirectional_iterator_tag>` is satisfied, then `iterator_category` denotes ~~is an alias for `ranges::`~~`bidirectional_iterator_tag`.

(1.3) — Otherwise, if ~~`R` satisfies `ForwardRange<R>`~~ `DerivedFrom<C, forward_iterator_tag>` is satisfied, then `iterator_category` denotes ~~is an alias for `ranges::`~~`forward_iterator_tag`.

(1.4) — Otherwise, `iterator_category` denotes ~~is an alias for `ranges::`~~`input_iterator_tag`.

2 `iterator::iterator_concept` is defined as follows:

(2.1) — If `R` models `BidirectionalRange`, then `iterator_concept` denotes `bidirectional_iterator_tag`.

(2.2) — Otherwise, if `R` models `ForwardRange`, then `iterator_concept` denotes `forward_iterator_tag`.

(2.3) — Otherwise, `iterator_concept` denotes `input_iterator_tag`.

29.8.5.2.1 `filter_view::iterator` operations [range.adaptors.filter__view.iterator.ops]

29.8.5.2.1.1 `filter_view::iterator` constructors [range.adaptors.filter__view.iterator.ctor]

```
constexpr iterator(filter_view& parent, iterator_t<R> current);
```

1 *Effects:* Initializes `current_` with `current` and `parent_` with ~~`&`~~`addressof(parent)`.

29.8.5.2.1.2 `filter_view::iterator` conversion [range.adaptors.filter__view.iterator.conv]

```
constexpr iterator_t<R> base() const;
```

1 *Returns:* `current_`.

29.8.5.2.1.3 `filter_view::iterator::operator*` [range.adaptors.filter__view.iterator.star]

```
constexpr iter_reference_t<iterator_t<R>> operator*() const;
```

1 *Returns:* `*current_`.

29.8.5.2.1.4 `filter_view::iterator::operator->` [range.adaptors.filter_view.iterator.arrow]

```
constexpr iterator_t<R> operator->() const
  requires has-arrow<iterator_t<R>>;
```

1 *Returns:* `current_`.

29.8.5.2.1.5 `filter_view::iterator::operator++` [range.adaptors.filter_view.iterator.inc]

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

1 *Effects:* Equivalent to:

```
current_ = ranges::find_if(++current_, ranges::end(parent_->base_), ref(*parent_->pred_));
return *this;
```

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

2 *Effects:* Equivalent to ~~`(void)`~~`++*this`.

```
constexpr iterator operator++(int) requires ForwardRange<R>;
```

3 *Effects:* Equivalent to:

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

29.8.5.2.1.6 `filter_view::iterator::operator--` [range.adaptors.filter_view.iterator.dec]

```
constexpr iterator& operator--() requires BidirectionalRange<R>;
```

1 *Effects:* Equivalent to:

```
do
  --current_;
while (invoke(*parent_->pred_, *current_));
return *this;
```

```
constexpr iterator operator--(int) requires BidirectionalRange<R>;
```

2 *Effects:* Equivalent to:

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

29.8.5.2.1.7 `filter_view::iterator` comparisons [range.adaptors.filter_view.iterator.comp]

```
friend constexpr bool operator==(const iterator& x, const iterator& y)
  requires EqualityComparable<iterator_t<R>>;
```

1 *Returns:* `x.current_ == y.current_`.

```
friend constexpr bool operator!=(const iterator& x, const iterator& y)
  requires EqualityComparable<iterator_t<R>>;
```

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

29.8.5.2.2 `filter_view::iterator` non-member functions [range.adaptors.filter_view.iterator.nonmember]

```
friend constexpr iter_rvalue_reference_t<iterator_t<R>> iter_move(const iterator& i)
  noexcept(~~see below~~noexcept(ranges::iter_move(i.current_)));
```

1 *Returns:* `ranges::iter_move(i.current_)`.

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

~~`noexcept(ranges::iter_move(i.current_))`~~

```
friend constexpr void iter_swap(const iterator& x, const iterator& y)
  noexcept(~~see below~~noexcept(ranges::iter_swap(x.current_, y.current_)))
  requires IndirectlySwappable<iterator_t<R>>;
```

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

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

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

29.8.5.3 Class template `filter_view::sentinel` [range.adaptors.filter_view.sentinel]

```
namespace std::ranges { namespace ranges {
  template<class R, class Pred>
  class filter_view<R, Pred>::sentinel {
  private:
    sentinel_t<R> end_ {}; // exposition only
  public:
    sentinel() = default;
    explicit constexpr sentinel(filter_view& parent);

    constexpr sentinel_t<R> base() const;

    friend constexpr bool operator==(const iterator& x, const sentinel& y);
    friend constexpr bool operator==(const sentinel& x, const iterator& y);
    friend constexpr bool operator!=(const iterator& x, const sentinel& y);
    friend constexpr bool operator!=(const sentinel& x, const iterator& y);
  };
}}
```

29.8.5.3.1 `filter_view::sentinel` constructors [range.adaptors.filter_view.sentinel.ctor]

```
explicit constexpr sentinel(filter_view& parent);
```

1 *Effects:* Initializes `end_` with `ranges::end(parent)`.

29.8.5.3.2 `filter_view::sentinel` conversion [range.adaptors.filter_view.sentinel.conv]

```
constexpr sentinel_t<R> base() const;
```

1 *Returns:* `end_`.

29.8.5.3.3 `filter_view::sentinel` comparison [range.adaptors.filter_view.sentinel.comp]

```
friend constexpr bool operator==(const iterator& x, const sentinel& y);
```

1 *Returns:* `x.current_ == y.end_`.

```
friend constexpr bool operator==(const sentinel& x, const iterator& y);
```

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

```
friend constexpr bool operator!=(const iterator& x, const sentinel& y);
```

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

```
friend constexpr bool operator!=(const sentinel& x, const iterator& y);
```

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

29.8.6 `view::filter` [range.adaptors.filter]

1 The name `view::filter` denotes a range adaptor object (29.8.1). ~~Let `E` and `P` be expressions such that types `T` and `U` are `decltype((E))` and `decltype((P))` respectively. Then~~ The expression `view::filter(E, P)` for subexpressions `E` and `P` is expression-equivalent to `filter_view{E, P}`. ~~if `InputRange<T> && IndirectUnaryPredicate<decay_t<U>, iterator_t<T>>` is satisfied. Otherwise, `view::filter(E, P)` is ill-formed.~~

29.8.7 Class template `transform_view` [range.adaptors.transform_view]

1 ~~The purpose of~~ `transform_view` ~~is to~~ presents a ~~`view`~~`View` of an underlying sequence after applying a transformation function to each element.

2 [*Example:*

```
vector<int> is{ 0, 1, 2, 3, 4 };
transform_view squares{is, [](int i) { return i * i; }};
```

```
  for (int i : squares)
    cout << i << ' '; // prints: 0 1 4 9 16
```

— *end example*]

```
namespace std::ranges { namespace ranges {
  template<InputRange R, CopyConstructible F>
    requires View<R> && is_object_v<F> && Invocable<F&, iter_reference_t<iterator_t<R>>>
  class transform_view : public view_interface<transform_view<R, F>> {
  private:
    R base_ {}; // exposition only
    semiregular<F> fun_; // exposition only
    template<bool Const>
      struct iterator; // exposition only
    template<bool Const>
      struct sentinel; // exposition only
  public:
    transform_view() = default;
    constexpr transform_view(R base, F fun);
    template<InputRange O>
      requires ViewableRange<O> && Constructible<R, all_view<O>>
    constexpr transform_view(O&& o, F fun);

    constexpr R base() const;

    constexpr auto begin();
    constexpr auto begin() const requires Range<const R> &&
      Invocable<const F&, iter_reference_t<iterator_t<const R>>>;

    constexpr auto end();
    constexpr auto end() const requires Range<const R> &&
      Invocable<const F&, iter_reference_t<iterator_t<const R>>>;
    constexpr auto end() requires CommonRange<R>;
    constexpr auto end() const requires CommonRange<const R> &&
      Invocable<const F&, iter_reference_t<iterator_t<const R>>>;

    constexpr auto size() requires SizedRange<R>;
    constexpr auto size() const requires SizedRange<const R>;
  };

  template<class R, class F>
  transform_view(R&& r, F fun) -> transform_view<all_view<R>, F>;
}}
```

29.8.7.1 `transform_view` operations [range.adaptors.transform_view.ops]

29.8.7.1.1 `transform_view` constructors [range.adaptors.transform_view.ctor]

```
constexpr transform_view(R base, F fun);
```

1 *Effects:* Initializes `base_` with `std::move(base)` and initializes `fun_` with `std::move(fun)`.

```
template<InputRange O>
  requires ViewableRange<O> && Constructible<R, all_view<O>>
constexpr transform_view(O&& o, F fun);
```

2 *Effects:* Initializes `base_` with `view::all(std::forward<O>(o))` and initializes `fun_` with `std::move(fun)`.

29.8.7.1.2 `transform_view` conversion [range.adaptors.transform_view.conv]

```
constexpr R base() const;
```

1 *Returns:* `base_`.

29.8.7.1.3 `transform_view` range begin [range.adaptors.transform_view.begin]

```
constexpr auto begin();
```

```
constexpr auto begin() const requires Range<const R> &&
  Invocable<const F&, iter_reference_t<iterator_t<const R>>>;
```

1 *Effects:* Equivalent to:

```
return iterator<false>{*this, ranges::begin(base_)};
```

and

```
return iterator<true>{*this, ranges::begin(base_)};
```

for the first and second overload, respectively.

29.8.7.1.4 transform_view range end [range.adaptors.transform_view.end]

```
constexpr auto end();
constexpr auto end() const requires Range<const R> &&
  Invocable<const F&, iter_reference_t<iterator_t<const R>>>;
```

1 *Effects:* Equivalent to:

```
return sentinel<false>{ranges::end(base_)};
```

and

```
return sentinel<true>{ranges::end(base_)};
```

for the first and second overload, respectively.

```
constexpr auto end() requires CommonRange<R>;
constexpr auto end() const requires CommonRange<const R> &&
  Invocable<const F&, iter_reference_t<iterator_t<const R>>>;
```

2 *Effects:* Equivalent to:

```
return iterator<false>{*this, ranges::end(base_)};
```

and

```
return iterator<true>{*this, ranges::end(base_)};
```

for the first and second overload, respectively.

29.8.7.1.5 transform_view range size [range.adaptors.transform_view.size]

```
constexpr auto size() requires SizedRange<R>;
constexpr auto size() const requires SizedRange<const R>;
```

1 *Returns:* `ranges::size(base_)`.

29.8.7.2 Class template transform_view::iterator [range.adaptors.transform_view.iterator]

1 ~~transform_view<R, F>::iterator is an exposition-only type.~~

```
namespace std::ranges { namespace ranges {
  template<class R, class F>
  template<bool Const>
  class transform_view<R, F>::iterator { // exposition only
  private:
    placeholder{has-arrow} = conditional_t<Const, const transform_view, transform_view>;
    using Base   = conditional_t<Const, const R, R>;
    iterator_t<Base> current_ {};
    Parent* parent_ = nullptr;
  public:
    using iterator_category =
      typename iterator_traits<iterator_t<Base>>::iterator_category;
      iterator_category_t<iterator_t<Base>>;
    using iterator_concept = see below;
    using value_type        =
      remove_cvref_t<invoke_result_t<F&, iter_reference_t<iterator_t<Base>>>>;
      remove_const_t<remove_reference_t<invoke_result_t<F&, reference_t<iterator_t<Base>>>>>;
    using difference_type   = iter_difference_tdifference_type_t<iterator_t<Base>>;
```

```
    iterator() = default;
    constexpr iterator(Parent& parent, iterator_t<Base> current);
    constexpr iterator(iterator<!Const> i)
      requires Const && ConvertibleTo<iterator_t<R>, iterator_t<Base>>;

    constexpr iterator_t<Base> base() const;
    constexpr decltype(auto) operator*() const;

    constexpr iterator& operator++();
    constexpr void operator++(int);
    constexpr iterator operator++(int) requires ForwardRange<Base>;

    constexpr iterator& operator--() requires BidirectionalRange<Base>;
    constexpr iterator operator--(int) requires BidirectionalRange<Base>;

    constexpr iterator& operator+=(difference_type n)
      requires RandomAccessRange<Base>;
    constexpr iterator& operator-=(difference_type n)
      requires RandomAccessRange<Base>;
    constexpr decltype(auto) operator[](difference_type n) const
      requires RandomAccessRange<Base>;

    friend constexpr bool operator==(const iterator& x, const iterator& y)
      requires EqualityComparable<iterator_t<Base>>;
    friend constexpr bool operator!=(const iterator& x, const iterator& y)
      requires EqualityComparable<iterator_t<Base>>;

    friend constexpr bool operator<(const iterator& x, const iterator& y)
      requires RandomAccessRange<Base>;
    friend constexpr bool operator>(const iterator& x, const iterator& y)
      requires RandomAccessRange<Base>;
    friend constexpr bool operator<=(const iterator& x, const iterator& y)
      requires RandomAccessRange<Base>;
    friend constexpr bool operator>=(const iterator& x, const iterator& y)
      requires RandomAccessRange<Base>;

    friend constexpr iterator operator+(iterator i, difference_type n)
      requires RandomAccessRange<Base>;
    friend constexpr iterator operator+(difference_type n, iterator i)
      requires RandomAccessRange<Base>;

    friend constexpr iterator operator-(iterator i, difference_type n)
      requires RandomAccessRange<Base>;
    friend constexpr difference_type operator-(const iterator& x, const iterator& y)
      requires RandomAccessRange<Base>;

    friend constexpr decltype(auto) iter_move(const iterator& i)
      noexcept(~~see below~~noexcept(invoke(*i.parent_->fun_, *i.current_)));
    friend constexpr void iter_swap(const iterator& x, const iterator& y)
      noexcept(~~see below~~noexcept(ranges::iter_swap(x.current_, y.current_)))
      requires IndirectlySwappable<iterator_t<Base>>;
  };
}~~}~~
```

2 `iterator::iterator_concept` is defined as follows:

(2.1) — If `R` models `RandomAccessRange`, then `iterator_concept` denotes `random_access_iterator_tag`.

(2.2) — Otherwise, if `R` models `BidirectionalRange`, then `iterator_concept` denotes `bidirectional_iterator_tag`.

(2.3) — Otherwise, if `R` models `ForwardRange`, then `iterator_concept` denotes `forward_iterator_tag`.

(2.4) — Otherwise, `iterator_concept` denotes `input_iterator_tag`.

29.8.7.2.1 transform_view::iterator operations [range.adaptors.transform_view.iterator.ops]

29.8.7.2.1.1 transform_view::iterator constructors [range.adaptors.transform_view.iterator.ctor]

```
constexpr iterator(Parent& parent, iterator_t<Base> current);
```

1 *Effects:* Initializes current_ with current and initializes parent_ with &addressof(parent).

```
constexpr iterator(iterator<!Const> i)
  requires Const && ConvertibleTo<iterator_t<R>, iterator_t<Base>>;
```

2 *Effects:* Initializes parent_ with i.parent_ and current_ with i.current_.

29.8.7.2.1.2 transform_view::iterator conversion [range.adaptors.transform_view.iterator.conv]

```
constexpr iterator_t<Base> base() const;
```

1 *Returns:* current_.

29.8.7.2.1.3 transform_view::iterator::operator* [range.adaptors.transform_view.iterator.star]

```
constexpr decltype(auto) operator*() const;
```

1 *Returns:* invoke(*parent_->fun_, *current_).

29.8.7.2.1.4 transform_view::iterator::operator++ [range.adaptors.transform_view.iterator.inc]

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

1 *Effects:* Equivalent to:

```
++current_;
return *this;
```

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

2 *Effects:* Equivalent to:

```
++current_;
```

```
constexpr iterator operator++(int) requires ForwardRange<Base>;
```

3 *Effects:* Equivalent to:

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

29.8.7.2.1.5 transform_view::iterator::operator-- [range.adaptors.transform_view.iterator.dec]

```
constexpr iterator& operator--() requires BidirectionalRange<Base>;
```

1 *Effects:* Equivalent to:

```
--current_;
return *this;
```

```
constexpr iterator operator--(int) requires BidirectionalRange<Base>;
```

2 *Effects:* Equivalent to:

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

29.8.7.2.1.6 transform_view::iterator advance [range.adaptors.transform_view.iterator.adv]

```
constexpr iterator& operator+=(difference_type n)
  requires RandomAccessRange<Base>;
```

1 *Effects:* Equivalent to:

```
current_ += n;
return *this;
```

```
constexpr iterator& operator-=(difference_type n)
  requires RandomAccessRange<Base>;
```

2 *Effects:* Equivalent to:

```
current_ -= n;
return *this;
```

29.8.7.2.1.7 transform_view::iterator index [range.adaptors.transform_view.iterator.idx]

```
constexpr decltype(auto) operator[](difference_type n) const
  requires RandomAccessRange<Base>;
```

1 *Effects:* Equivalent to:

```
return invoke(*parent_->fun_, current_[n]);
```

29.8.7.2.2 transform_view::iterator comparisons [range.adaptors.transform_view.iterator.comp]

```
friend constexpr bool operator==(const iterator& x, const iterator& y)
  requires EqualityComparable<iterator_t<Base>>;
```

1 *Returns:* `x.current_ == y.current_`.

```
friend constexpr bool operator!=(const iterator& x, const iterator& y)
  requires EqualityComparable<iterator_t<Base>>;
```

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

```
friend constexpr bool operator<(const iterator& x, const iterator& y)
  requires RandomAccessRange<Base>;
```

3 *Returns:* `x.current_ < y.current_`.

```
friend constexpr bool operator>(const iterator& x, const iterator& y)
  requires RandomAccessRange<Base>;
```

4 *Returns:* `y < x`.

```
friend constexpr bool operator<=(const iterator& x, const iterator& y)
  requires RandomAccessRange<Base>;
```

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

```
friend constexpr bool operator>=(const iterator& x, const iterator& y)
  requires RandomAccessRange<Base>;
```

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

29.8.7.2.3 transform_view::iterator non-member functions [range.adaptors.transform_view.iterator.nonmember]

```
friend constexpr iterator operator+(iterator i, difference_type n)
  requires RandomAccessRange<Base>;
friend constexpr iterator operator+(difference_type n, iterator i)
  requires RandomAccessRange<Base>;
```

1 *Returns:* `iterator{*i.parent_, i.current_ + n}`.

```
friend constexpr iterator operator-(iterator i, difference_type n)
  requires RandomAccessRange<Base>;
```

2 *Returns:* `iterator{*i.parent_, i.current_ - n}`.

```
friend constexpr difference_type operator-(const iterator& x, const iterator& y)
  requires RandomAccessRange<Base>;
```

3 *Returns:* `x.current_ - y.current_`.

```
friend constexpr decltype(auto) iter_move(const iterator& i)
  noexcept(see belownoexcept(invoke(*i.parent_->fun_, *i.current_)));
```

4 *Effects:* Equivalent to:

(4.1) — If the expression `*i` is an lvalue~~, then~~: `return std::move(*i);`

(4.2) — Otherwise: `return *i;`

5 ~~*Remarks:* The expression in the `noexcept` is equivalent to:~~

~~`noexcept(invoke(*i.parent_->fun_, *i.current_))`~~

```
friend constexpr void iter_swap(const iterator& x, const iterator& y)
  noexcept(see belownoexcept(ranges::iter_swap(x.current_, y.current_)))
  requires IndirectlySwappable<iterator_t<Base>>;
```

6 *Effects:* Equivalent to `ranges::iter_swap(x.current_, y.current_)`.

7 ~~*Remarks:* The expression in the `noexcept` is equivalent to:~~

~~`noexcept(ranges::iter_swap(x.current_, y.current_))`~~

29.8.7.3 Class template `transform_view::sentinel` [range.adaptors.transform_view.sentinel]

1 ~~`transform_view<R, F>::sentinel` is an exposition-only type.~~

```
namespace std::ranges { namespace ranges {
  template<class R, class F>
  template<bool Const>
  class transform_view<R, F>::sentinel { // exposition only
  private:
    placeholder{has-arrow} = conditional_t<Const, const transform_view, transform_view>;
    using Base = conditional_t<Const, const R, R>;
    sentinel_t<Base> end_ {};
  public:
    sentinel() = default;
    explicit constexpr sentinel(sentinel_t<Base> end);
    constexpr sentinel(sentinel<!Const> i)
      requires Const && ConvertibleTo<sentinel_t<R>, sentinel_t<Base>>;

    constexpr sentinel_t<Base> base() const;

    friend constexpr bool operator==(const iterator<Const>& x, const sentinel& y);
    friend constexpr bool operator==(const sentinel& x, const iterator<Const>& y);
    friend constexpr bool operator!=(const iterator<Const>& x, const sentinel& y);
    friend constexpr bool operator!=(const sentinel& x, const iterator<Const>& y);

    friend constexpr iter_difference_type_t<iterator_t<Base>>
      operator-(const iterator<Const>& x, const sentinel& y)
        requires SizedSentinel<sentinel_t<Base>, iterator_t<Base>>;
    friend constexpr iter_difference_type_t<iterator_t<Base>>
      operator-(const sentinel& y, const iterator<Const>& x)
        requires SizedSentinel<sentinel_t<Base>, iterator_t<Base>>;
  };
}}
```

29.8.7.4 `transform_view::sentinel` constructors [range.adaptors.transform_view.sentinel.ctor]

```
explicit constexpr sentinel(sentinel_t<Base> end);
```

1 *Effects:* Initializes `end_` with `end`.

```
constexpr sentinel(sentinel<!Const> i)
  requires Const && ConvertibleTo<sentinel_t<R>, sentinel_t<Base>>;
```

2 *Effects:* Initializes `end_` with `i.end_`.

29.8.7.5 `transform_view::sentinel` conversion [range.adaptors.transform_view.sentinel.conv]

```
constexpr sentinel_t<Base> base() const;
```

1 *Returns:* `end_`.

29.8.7.6 `transform_view::sentinel` comparison [range.adaptors.transform_view.sentinel.comp]

```
friend constexpr bool operator==(const iterator<Const>& x, const sentinel& y);
```

1 *Returns:* `x.current_ == y.end_`.

```
friend constexpr bool operator==(const sentinel& x, const iterator<Const>& y);
```

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

```
friend constexpr bool operator!=(const iterator<Const>& x, const sentinel& y);
```

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

```
friend constexpr bool operator!=(const sentinel& x, const iterator<Const>& y);
```

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

29.8.7.7 `transform_view::sentinel` non-member functions [range.adaptors.transform_view.sentinel.nonmember]

```
friend constexpr iter_difference_type_t<iterator_t<Base>>
operator-(const iterator<Const>& x, const sentinel& y)
  requires SizedSentinel<sentinel_t<Base>, iterator_t<Base>>;
```

1 *Returns:* `x.current_ - y.end_`.

```
friend constexpr iter_difference_type_t<iterator_t<Base>>
operator-(const sentinel& y, const iterator<Const>& x)
  requires SizedSentinel<sentinel_t<Base>, iterator_t<Base>>;
```

2 *Returns:* `x.end_ - y.current_`.

29.8.8 `view::transform` [range.adaptors.transform]

1 The name `view::transform` denotes a range adaptor object (29.8.1). ~~Let `E` and `F` be expressions such that types `T` and `U` are `decltype((E))` and `decltype((F))` respectively. Then~~ The expression `view::transform(E, F)` for subexpressions `E` and `F` is expression-equivalent to `transform_view{E, F}`. ~~if `InputRange<T> && CopyConstructible<decay_t<U>> && Invocable<decay_t<U>&, reference_t<iterator_t<T>>>`is satisfied. Otherwise, `view::transform(E, F)` is ill-formed.~~

29.8.9 Class template `iota_view` [range.adaptors.iota_view]

1 ~~The purpose of~~ `iota_view` ~~is to~~ generates a sequence of elements by repeatedly incrementing an initial value.

2 [*Example:*

```
iota_view indices{1, 10};
for (int i : iota_view{1, 10} indices)
  cout << i << ' '; // prints: 1 2 3 4 5 6 7 8 9
```

— *end example*]

```
namespace std::ranges { namespace ranges {
  template<class I>
  concept Decrementable = // exposition only
    see below;
  template<class I>
  concept Advanceable = // exposition only
    see below;

  template<WeaklyIncrementable I, classSemiregular Bound = unreachable>
    requires weakly-equality-comparable-with<I, Bound>
  class iota_view : public view_interface<iota_view<I, Bound>> {
  private:
    I value_ {}; // exposition only
```

```
    Bound bound_ {}; // exposition only
    struct iterator; // exposition only
    struct sentinel; // exposition only
  public:
    iota_view() = default;
    constexpr explicit iota_view(I value);
    constexpr iota_view(I value, Bound bound); // see below

    constexpr iterator begin() const;
    constexpr sentinel end() const;
    constexpr iterator end() const requires Same<I, Bound>;

    constexpr auto size() const requires see below;
  };

  template<WeaklyIncrementableclass I>
  explicit iota_view(I) -> iota_view<I>;

  template<WeaklyIncrementableclass I, Semiregularclass Bound>
    requires
      (!Integral<I> || !Integral<Bound> || is_signed_v<I> == is_signed_v<Bound>)
  iota_view(I, Bound) -> iota_view<I, Bound>;
}}
```

3 The exposition-only *Decrementable* concept is equivalent to:

```
template<class I>
concept Decrementable =
  Incrementable<I> && requires(I i) {
    { --i } -> Same<I>&;
    i--; requires Same<I, decltype(i--)>;
  };
```

4 When an object is in the domain of both pre- and post-decrement, the object is said to be *Decrementable*.

5 Let `a` and `b` be incrementable and decrementable objects of type `I`. I models *Decrementable*~~<I> is satisfied~~ only if

(5.1) — `addressof(--a) == addressof(a)`.

(5.2) — If `bool(a == b)` then `bool(a-- == b)`.

(5.3) — If `bool(a == b)` then `bool((a--, a) == --b)`.

(5.4) — If `bool(a == b)` then `bool(--(++a) == b)` and `bool(++(--a) == b)`.

6 The exposition-only *Advanceable* concept is equivalent to:

```
template<class I>
concept Advanceable =
  Decrementable<I> && StrictTotallyOrdered<I> &&
  requires { typename iter_difference_type_t<I>; } &&
  requires(I i, const I j, const iter_difference_type_t<I> n) {
    { i += n } -> Same<I>&;
    { i -= n } -> Same<I>&;
    j + n; requires Same<I, decltype(j + n)>;
    n + j; requires Same<I, decltype(n + j)>;
    j - n; requires Same<I, decltype(j - n)>;
    j - j; requires Same<iter_difference_type_t<I>, decltype(j - j)>;
  };
```

Let `a` and `b` be objects of type `I` such that `b` is reachable from `a`. Let M be the smallest number of applications of `++a` necessary to make `bool(a == b)` be `true`. Let `n`, `zero`, and `one` be objects of type `iter_difference`~~`_type`~~`_t<I>` initialized with M, `0`, and `1`, respectively. Then if M is representable by `iter_difference`~~`_type`~~`_t<I>`, I models *Advanceable*~~<I> is satisfied~~ only if:

(6.1) — `(a += n)` is equal to `b`.

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

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

(6.4) — For any two positive integers `x` and `y`, if `a + (x + y)` is valid, then `a + (x + y)` is equal to `(a + x) + y`.

(6.5) — `a + zero` is equal to `a`.

(6.6) — If `(a + (n - one))` is valid, then `a + n` is equal to `++(a + (n - one))`.

(6.7) — `(b += -n)` is equal to `a`.

(6.8) — `(b -= n)` is equal to `a`.

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

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

(6.11) — `b - a` is equal to `n`.

(6.12) — `a - b` is equal to `-n`.

(6.13) — `a <= b`.

29.8.9.1 iota_view operations [range.adaptors.iota_view.ops]

29.8.9.1.1 iota_view constructors [range.adaptors.iota_view.ctor]

```
constexpr explicit iota_view(I value);
```

1 *Requires:* `Bound{}` is reachable from `value`.

2 *Effects:* Initializes `value_` with `value`.

```
constexpr iota_view(I value, Bound bound);
```

3 *Requires:* `bound` is reachable from `value`.

4 *Effects:* Initializes `value_` with `value` and `bound_` with `bound`.

5 *Remarks:* This constructor does not contribute a function template to the overload set used when resolving a placeholder for a deduced class type ([over.match.class.deduct]).

29.8.9.1.2 iota_view range begin [range.adaptors.iota_view.begin]

```
constexpr iterator begin() const;
```

1 *Returns:* `iterator{value_}`.

29.8.9.1.3 iota_view range end [range.adaptors.iota_view.end]

```
constexpr sentinel end() const;
```

1 *Returns:* `sentinel{bound_}`.

```
constexpr iterator end() const requires Same<I, Bound>;
```

2 *Returns:* `iterator{bound_}`.

29.8.9.1.4 iota_view range size [range.adaptors.iota_view.size]

`constexpr auto size() const requires` ~~`see below`~~
`(Same<I, Bound> && Advanceable<I>) ||`
`(Integral<I> && Integral<Bound>) ||`
`SizedSentinel<Bound, I>;`

1 ~~*Returns:*~~ *Effects:* Equivalent to: `return bound_ - value_;`

2 ~~*Remarks:* The expression in the `requires` clause is equivalent to:~~

~~`(Same<I, Bound> && Advanceable<I>) ||`~~
~~`(Integral<I> && Integral<Bound>) ||`~~
~~`SizedSentinel<Bound, I>`~~

29.8.9.2 Class iota_view::iterator [range.adaptors.iota_view.iterator]

```
namespace std::ranges { ~~namespace ranges {~~
  template<class I, class Bound>
  struct iota_view<I, Bound>::iterator {
  private:
    I value_ {}; // exposition only
  public:
    using iterator_category = see below;
    using value_type = I;
    using difference_type = iter_difference~~_type~~_t<I>;

    iterator() = default;
    explicit constexpr iterator(I value);

    constexpr I operator*() const noexcept(is_nothrow_copy_constructible_v<I>);

    constexpr iterator& operator++();
    constexpr void operator++(int);
    constexpr iterator operator++(int) requires Incrementable<I>;

    constexpr iterator& operator--() requires Decrementable<I>;
    constexpr iterator operator--(int) requires Decrementable<I>;

    constexpr iterator& operator+=(difference_type n)
      requires Advanceable<I>;
    constexpr iterator& operator-=(difference_type n)
      requires Advanceable<I>;
    constexpr I operator[](difference_type n) const
      requires Advanceable<I>;

    friend constexpr bool operator==(const iterator& x, const iterator& y)
      requires EqualityComparable<I>;
    friend constexpr bool operator!=(const iterator& x, const iterator& y)
      requires EqualityComparable<I>;

    friend constexpr bool operator<(const iterator& x, const iterator& y)
      requires StrictTotallyOrdered<I>;
    friend constexpr bool operator>(const iterator& x, const iterator& y)
      requires StrictTotallyOrdered<I>;
    friend constexpr bool operator<=(const iterator& x, const iterator& y)
      requires StrictTotallyOrdered<I>;
    friend constexpr bool operator>=(const iterator& x, const iterator& y)
      requires StrictTotallyOrdered<I>;

    friend constexpr iterator operator+(iterator i, difference_type n)
      requires Advanceable<I>;
    friend constexpr iterator operator+(difference_type n, iterator i)
      requires Advanceable<I>;

    friend constexpr iterator operator-(iterator i, difference_type n)
      requires Advanceable<I>;
    friend constexpr difference_type operator-(const iterator& x, const iterator& y)
      requires Advanceable<I>;
  };
}~~}~~
```

1 ~~iota_view<I, Bound>::~~iterator::iterator_category is defined as follows:

(1.1) — If I ~~satisfies~~ models *Advanceable*, then iterator_category is ~~ranges::~~random_access_iterator_tag.

(1.2) — Otherwise, if I ~~satisfies~~ models *Decrementable*, then iterator_category is ~~ranges::~~bidirectional_iterator_tag.

(1.3) — Otherwise, if I ~~satisfies~~ models `Incrementable`, then `iterator_category` is ~~`ranges::`~~`forward_iterator_tag`.

(1.4) — Otherwise, `iterator_category` is ~~`ranges::`~~`input_iterator_tag`.

2 [*Note:* Overloads for `iter_move` and `iter_swap` are omitted intentionally. — *end note*]

29.8.9.2.1 `iota_view::iterator` operations [range.adaptors.iota_view.iterator.ops]

29.8.9.2.1.1 `iota_view::iterator` constructors [range.adaptors.iota_view.iterator.ctor]

```
explicit constexpr iterator(I value);
```

1 *Effects:* Initializes `value_` with `value`.

29.8.9.2.1.2 `iota_view::iterator::operator*` [range.adaptors.iota_view.iterator.star]

```
constexpr I operator*() const noexcept(is_nothrow_copy_constructible_v<I>);
```

1 *Returns:* `value_`.

2 [*Note:* The `noexcept` clause is needed by the default `iter_move` implementation. — *end note*]

29.8.9.2.1.3 `iota_view::iterator::operator++` [range.adaptors.iota_view.iterator.inc]

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

1 *Effects:* Equivalent to:

```
++value_;
return *this;
```

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

2 *Effects:* Equivalent to `++*this`.

```
constexpr iterator operator++(int) requires Incrementable<I>;
```

3 *Effects:* Equivalent to:

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

29.8.9.2.1.4 `iota_view::iterator::operator--` [range.adaptors.iota_view.iterator.dec]

```
constexpr iterator& operator--() requires Decrementable<I>;
```

1 *Effects:* Equivalent to:

```
--value_;
return *this;
```

```
constexpr iterator operator--(int) requires Decrementable<I>;
```

2 *Effects:* Equivalent to:

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

29.8.9.2.1.5 `iota_view::iterator` advance [range.adaptors.iota_view.iterator.adv]

```
constexpr iterator& operator+=(difference_type n)
  requires Advanceable<I>;
```

1 *Effects:* Equivalent to:

```
value_ += n;
return *this;
```

```
constexpr iterator& operator-=(difference_type n)
  requires Advanceable<I>;
```

2 *Effects:* Equivalent to:

```
value_ -= n;
return *this;
```

29.8.9.2.1.6 iota_view::iterator index [range.adaptors.iota_view.iterator.idx]

```
constexpr I operator[](difference_type n) const
  requires Advanceable<I>;
```

1 *Returns:* `value_ + n`.

29.8.9.2.2 iota_view::iterator comparisons [range.adaptors.iota_view.iterator.cmp]

```
friend constexpr bool operator==(const iterator& x, const iterator& y)
  requires EqualityComparable<I>;
```

1 *Returns:* `x.value_ == y.value_`.

```
friend constexpr bool operator!=(const iterator& x, const iterator& y)
  requires EqualityComparable<I>;
```

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

```
friend constexpr bool operator<(const iterator& x, const iterator& y)
  requires StrictTotallyOrdered<I>;
```

3 *Returns:* `x.value_ < y.value_`.

```
friend constexpr bool operator>(const iterator& x, const iterator& y)
  requires StrictTotallyOrdered<I>;
```

4 *Returns:* `y < x`.

```
friend constexpr bool operator<=(const iterator& x, const iterator& y)
  requires StrictTotallyOrdered<I>;
```

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

```
friend constexpr bool operator>=(const iterator& x, const iterator& y)
  requires StrictTotallyOrdered<I>;
```

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

29.8.9.2.3 iota_view::iterator non-member functions [range.adaptors.iota_view.iterator.nonmember]

```
friend constexpr iterator operator+(iterator i, difference_type n)
  requires Advanceable<I>;
```

1 *Returns:* `iterator{*i + n}`.

```
friend constexpr iterator operator+(difference_type n, iterator i)
  requires Advanceable<I>;
```

2 *Returns:* `i + n`.

```
friend constexpr iterator operator-(iterator i, difference_type n)
  requires Advanceable<I>;
```

3 *Returns:* `i + -n`.

```
friend constexpr difference_type operator-(const iterator& x, const iterator& y)
  requires Advanceable<I>;
```

4 *Returns:* `*x - *y`.

29.8.9.3 Class iota_view::sentinel [range.adaptors.iota_view.sentinel]

```
namespace std::ranges { ~~namespace ranges {~~
  template<class I, class Bound>
  struct iota_view<I, Bound>::sentinel {
  private:
    Bound bound_ {}; // exposition only
  public:
    sentinel() = default;
    constexpr explicit sentinel(Bound bound);
```

```
    friend constexpr bool operator==(const iterator& x, const sentinel& y);
    friend constexpr bool operator==(const sentinel& x, const iterator& y);
    friend constexpr bool operator!=(const iterator& x, const sentinel& y);
    friend constexpr bool operator!=(const sentinel& x, const iterator& y);
  };
}}}}
```

29.8.9.3.1 `iota_view::sentinel` constructors [range.adaptors.iota_view.sentinel.ctor]

`constexpr explicit sentinel(Bound bound);`

1 *Effects:* Initializes `bound_` with `bound`.

29.8.9.3.2 `iota_view::sentinel` comparisons [range.adaptors.iota_view.sentinel.cmp]

`friend constexpr bool operator==(const iterator& x, const sentinel& y);`

1 *Returns:* `x.value_ == y.bound_`.

`friend constexpr bool operator==(const sentinel& x, const iterator& y);`

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

`friend constexpr bool operator!=(const iterator& x, const sentinel& y);`

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

`friend constexpr bool operator!=(const sentinel& x, const iterator& y);`

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

29.8.10 `view::iota` [range.adaptors.iota]

1 The name `view::iota` denotes a customization point object ([customization.point.object]). ~~Let `E` and `F` be expressions such that their cv-unqualified types are `I` and `J` respectively. Then~~ The expressions `view::iota(E)` and `view::iota(E, F)` for some subexpressions `E` and `F` are ~~is~~ expression-equivalent to `iota_view{E}` and `iota_view{E, F}`, respectively. ~~if `WeaklyIncrementable<I>` is satisfied. Otherwise, `view::iota(E)` is ill-formed.~~

2 The expression `view::iota(E, F)` is expression-equivalent to:

(2.1) — `iota_view{E, F}` if the following set of constraints is satisfied:

(2.1.1) — `WeaklyIncrementable<I> && Semiregular<J> && __WeaklyEqualityComparableWith<I, J> && (!Integral<I> || !Integral<Bound> || std::is_signed_v<I> == std::is_signed_v<Bound>)`

(2.2) — Otherwise, `view::iota(E, F)` is ill-formed.

29.8.11 Class template `take_view` [range.adaptors.take_view]

1 ~~The purpose of~~ `take_view` ~~is to~~ produces a ~~range~~`View` of the first N elements from another ~~range~~`View`.

2 [*Example:*

```
vector<int> is{0,1,2,3,4,5,6,7,8,9};
take_view few{is, 5};
for (int i : few)
  cout << i << ' '; // prints: 0 1 2 3 4
```

— *end example*]

```
namespace std::ranges { namespace ranges {
  template<InputRange R>
    requires View<R>
  class take_view : public view_interface<take_view<R>> {
  private:
    R base_ {}; // exposition only
    iter_difference_type_t<iterator_t<R>> count_ {}; // exposition only
    template<bool Const>
      struct sentinel; // exposition only
  public:
    take_view() = default;
```

```
    constexpr take_view(R base, iter_difference_t<iterator_t<R>> count);
    template<InputRange O>
      requires ViewableRange<O> && Constructible<R, all_view<O>>
    constexpr take_view(O&& o, iter_difference_t<iterator_t<R>> count);

    constexpr R base() const;

    constexpr auto begin();
    constexpr auto begin() const requires Range<const R>;
    constexpr auto begin() requires RandomAccessRange<R> && SizedRange<R>;
    constexpr auto begin() const
      requires RandomAccessRange<const R> && SizedRange<const R>;

    constexpr auto end();
    constexpr auto end() const requires Range<const R>;
    constexpr auto end() requires RandomAccessRange<R> && SizedRange<R>;
    constexpr auto end() const
      requires RandomAccessRange<const R> && SizedRange<const R>;

    constexpr auto size() requires SizedRange<R>;
    constexpr auto size() const requires SizedRange<const R>;
  };

  template<class R>
  take_view(R&& base, iter_difference_t<iterator_t<R>> n)
    -> take_view<all_view<R>>;
}
```

29.8.11.1 take_view operations [range.adaptors.take__view.ops]

29.8.11.1.1 take_view constructors [range.adaptors.take__view.ctor]

```
constexpr take_view(R base, iter_difference_t<iterator_t<R>> count);
```

1 *Effects:* Initializes `base_` with `std::move(base)` and initializes `count_` with `count`.

```
template<InputRange O>
  requires ViewableRange<O> && Constructible<R, all_view<O>>
constexpr take_view(O&& o, iter_difference_t<iterator_t<R>> count);
```

2 *Effects:* Initializes `base_` with `view::all(std::forward<O>(o))` and initializes `count_` with `count`.

29.8.11.1.2 take_view conversion [range.adaptors.take__view.conv]

```
constexpr R base() const;
```

1 *Returns:* `base_`.

29.8.11.1.3 take_view range begin [range.adaptors.take__view.begin]

```
constexpr auto begin();
constexpr auto begin() const requires Range<const R>;
```

1 *Effects:* Equivalent to: `return` ~~`make_`~~`counted_iterator`~~`(`~~`{ranges::begin(base_), count_}`~~`)`~~`;`

```
constexpr auto begin() requires RandomAccessRange<R> && SizedRange<R>;
constexpr auto begin() const
  requires RandomAccessRange<const R> && SizedRange<const R>;
```

2 *Effects:* Equivalent to: `return ranges::begin(base_);`

29.8.11.1.4 take_view range end [range.adaptors.take__view.end]

```
constexpr auto end();
constexpr auto end() const requires Range<const R>;
```

[Editor's note: LWG: Why not constrain the non-`const` overload with `!`*`simple-view<R>`* and return something like `constexpr bool i_am_const = is_const_v<remove_reference_t<decltype(*this)>>; sentinel<i_am_const>{ranges::end(base_)}`? (Similarly for the other occurrences elsewhere.)]

1 *Effects:* Equivalent to: return sentinel<~~simple-view<R>~~*C*>{ranges::end(base_)}; ~~and sentinel<true>{ranges::end(base_)}~~ where *C* is *simple-view*<R> or true for the first and second overload, respectively.

```
constexpr auto end() requires RandomAccessRange<R> && SizedRange<R>;
constexpr auto end() const
  requires RandomAccessRange<const R> && SizedRange<const R>;
```

2 *Effects:* Equivalent to: return ranges::begin(base_) + size();

29.8.11.1.5 take_view range size [range.adaptors.take_view.size]

```
constexpr auto size() requires SizedRange<R>;
constexpr auto size() const requires SizedRange<const R>;
```

1 *Effects:* Equivalent to: ~~ranges::size(base_) < count_ ? ranges::size(base_) : count_, except with only one call to ranges::size(base_).~~

```
auto n = ranges::size(base_);
return min(n, static_cast<decltype(n)>(count_));
```

29.8.11.2 Class template take_view::sentinel [range.adaptors.take_view.sentinel]

1 ~~take_view<R>::sentinel is an exposition-only type.~~

```
namespace std::ranges { namespace ranges {
  template<class R>
  template<bool Const>
  class take_view<R>::sentinel { // exposition only
  private:
    placeholder{has-arrow} = conditional_t<Const, const take_view, take_view>;
    using Base = conditional_t<Const, const R, R>;
    sentinel_t<Base> end_ {};
    using CI = counted_iterator<iterator_t<Base>>;
  public:
    sentinel() = default;
    constexpr explicit sentinel(sentinel_t<Base> end);
    constexpr sentinel(sentinel<!Const> s)
      requires Const && ConvertibleTo<sentinel_t<R>, sentinel_t<Base>>;

    constexpr sentinel_t<Base> base() const;

    friend constexpr bool operator==(const sentinel& x, const CI& y)
      requires EqualityComparable<iterator_t<Base>>;
    friend constexpr bool operator==(const CI& x, const sentinel& y)
      requires EqualityComparable<iterator_t<Base>>;
    friend constexpr bool operator!=(const sentinel& x, const CI& y)
      requires EqualityComparable<iterator_t<Base>>;
    friend constexpr bool operator!=(const CI& x, const sentinel& y)
      requires EqualityComparable<iterator_t<Base>>;
  };
}}
```

29.8.11.2.1 take_view::sentinel operations [range.adaptors.take_view.sentinel.ops]

29.8.11.2.1.1 take_view::sentinel constructors [range.adaptors.take_view.sentinel.ctor]

```
constexpr explicit sentinel(sentinel_t<Base> end);
```

1 *Effects:* Initializes end_ with end.

```
constexpr sentinel(sentinel<!Const> s)
  requires Const && ConvertibleTo<sentinel_t<R>, sentinel_t<Base>>;
```

2 *Effects:* Initializes end_ with s.end_.

29.8.11.2.1.2 take_view::sentinel conversion [range.adaptors.take_view.sentinel.conv]

```
constexpr sentinel_t<Base> base() const;
```

1 *Returns:* end_.

29.8.11.2.2 `take_view::sentinel` comparisons [range.adaptors.take_view.sentinel.comp]

```
friend constexpr bool operator==(const sentinel& x, const CI& y)
  requires EqualityComparable<iterator_t<Base>>;
```

1 *Returns:* `y.count() == 0 || y.base() == x.end_`.

```
friend constexpr bool operator==(const CI& x, const sentinel& y)
  requires EqualityComparable<iterator_t<Base>>;
```

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

```
friend constexpr bool operator!=(const sentinel& x, const CI& y)
  requires EqualityComparable<iterator_t<Base>>;
```

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

```
friend constexpr bool operator!=(const CI& x, const sentinel& y)
  requires EqualityComparable<iterator_t<Base>>;
```

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

29.8.12 `view::take` [range.adaptors.take]

1 The name `view::take` denotes a range adaptor object (29.8.1). ~~Let `E` and `F` be expressions such that type `T` is `decltype((E))`. Then~~ The expression `view::take(E, F)` for some subexpressions `E` and `F` is expression-equivalent to `take_view{E, F}`. ~~if `InputRange<T>` is satisfied and if `F` is implicitly convertible to `difference_type_t<iterator_t<T>>` . Otherwise, `view::take(E, F)` is ill-formed.~~

29.8.13 Class template `join_view` [range.adaptors.join_view]

1 ~~The purpose of~~ `join_view` ~~is to~~ flattens a ~~range~~`View` of ranges into a ~~range~~`View`.

2 [*Example:*

```
vector<string> ss{"hello", " ", "world", "!"};
join_view greeting{ss};
for (char ch : greeting)
  cout << ch; // prints: hello world!
```

— *end example*]

```
namespace std::ranges { namespace ranges {
  template<InputRange R>
    requires View<R> && InputRange<iter_reference_t<iterator_t<R>>> &&
      (is_reference_v<iter_reference_t<iterator_t<R>>> ||
      View<iter_value_type_t<iterator_t<R>>>)
  class join_view : public view_interface<join_view<R>> {
  private:
    using InnerRng = iter_reference_t<iterator_t<R>>; // exposition only
    template<bool Const>
      struct iterator; // exposition only
    template<bool Const>
      struct sentinel; // exposition only

    R base_ {}; // exposition only
    all_view<InnerRng> inner_ {}; // exposition only, only present when !is_reference_v<InnerRng>
  public:
    join_view() = default;
    constexpr explicit join_view(R base);

    [Editor's note: LWG: Should this constructor be conditionally explicit?]
    template<InputRange O>
        requires ViewableRange<O> && Constructible<R, all_view<O>>
      constexpr explicit join_view(O&& o);

    constexpr auto begin();

    constexpr auto begin() const requires InputRange<const R> &&
      is_reference_v<iter_reference_t<iterator_t<const R>>>;
```

```
    constexpr auto end();

    constexpr auto end() const requires InputRange<const R> &&
      is_reference_v<iter_reference_t<iterator_t<const R>>>;

    constexpr auto end() requires ForwardRange<R> &&
      is_reference_v<InnerRng> && ForwardRange<InnerRng> &&
      CommonRange<R> && CommonRange<InnerRng>;

    constexpr auto end() const requires ForwardRange<const R> &&
      is_reference_v<iter_reference_t<iterator_t<const R>>> &&
      ForwardRange<iter_reference_t<iterator_t<const R>>> &&
      CommonRange<const R> && CommonRange<iter_reference_t<iterator_t<const R>>>;
  };

  template<~~InputRange~~class R>
    ~~requires InputRange<iter_reference_t<iterator_t<R>>> &&~~
      ~~(is_reference_v<reference_t<iterator_t<R>>> ||~~
      ~~View<value_type_t<iterator_t<R>>>)~~
    explicit join_view(R&&) -> join_view<all_view<R>>;
}~~}~~
```

29.8.13.1 `join_view` operations [range.adaptors.join_view.ops]

29.8.13.1.1 `join_view` constructors [range.adaptors.join_view.ctor]

```
explicit constexpr join_view(R base);
```

1 *Effects:* Initializes `base_` with `std::move(base)`.

```
template<InputRange O>
  requires ViewableRange<O> && Constructible<R, all_view<O>>
constexpr explicit join_view(O&& o);
```

2 *Effects:* Initializes `base_` with `view::all(std::forward<O>(o))`.

29.8.13.1.2 `join_view` range begin [range.adaptors.join_view.begin]

```
constexpr auto begin();
constexpr auto begin() const requires InputRange<const R> &&
  is_reference_v<iter_reference_t<iterator_t<const R>>>;
```

1 *Effects:* Equivalent to: `return iterator<`~~`simple-view<R>`~~*C*`>{*this, ranges::begin(base_)};` ~~and `return iterator<true>{*this, ranges::begin(base_)};`~~ where *C* is *`simple-view`*`<R>` or `true` for the first and second overloads, respectively.

29.8.13.1.3 `join_view` range end [range.adaptors.join_view.end]

```
constexpr auto end();
constexpr auto end() const requires InputRange<const R> &&
  is_reference_v<iter_reference_t<iterator_t<const R>>>;
```

1 *Effects:* Equivalent to: `return sentinel<`~~`simple-view<R>`~~*C*`>{*this};` ~~and `return sentinel<true>{*this};`~~ where *C* is *`simple-view`*`<R>` or `true` for the first and second overload, respectively.

```
constexpr auto end() requires ForwardRange<R> &&
  is_reference_v<InnerRng> && ForwardRange<InnerRng> &&
  CommonRange<R> && CommonRange<InnerRng>;
constexpr auto end() const requires ForwardRange<const R> &&
  is_reference_v<iter_reference_t<iterator_t<const R>>> &&
  ForwardRange<iter_reference_t<iterator_t<const R>>> &&
  CommonRange<const R> && CommonRange<iter_reference_t<iterator_t<const R>>>;
```

2 *Effects:* Equivalent to: `return iterator<`~~`simple-view<R>`~~*C*`>{*this, ranges::end(base_)};` ~~and `return iterator<true>{*this, ranges::end(base_)};`~~ where *C* is *`simple-view`*`<R>` or `true` for the first and second overloads, respectively.

29.8.13.2 Class template join_view::iterator [range.adaptors.join_view.iterator]

1 ~~join_view::iterator is an exposition-only type.~~

```
namespace std::ranges { ~~namespace ranges {~~
template<class R>
  template<bool Const>
  struct join_view<R>::iterator { // exposition only
  private:
    using Parent = conditional_t<Const, const join_view, join_view>;
    using Base   = conditional_t<Const, const R, R>;

    static constexpr bool ref_is_glvalue = // exposition only
      is_reference_v<iter_reference_t<iterator_t<Base>>>;

    iterator_t<Base> outer_ {}; // exposition only
    iterator_t<iter_reference_t<iterator_t<Base>>> inner_ {}; // exposition only
    Parent* parent_ {}; // exposition only

    constexpr void satisfy(); // exposition only
  public:
    using iterator_category = see below;
    using iterator_concept = see below;
    using value_type = iter_value~~_type~~_t<iterator_t<iter_reference_t<iterator_t<Base>>>>;
    using difference_type = see below;

    iterator() = default;
    constexpr iterator(Parent& parent, iterator_t<R> outer);
    constexpr iterator(iterator<!Const> i) requires Const &&
      ConvertibleTo<iterator_t<R>, iterator_t<Base>> &&
      ConvertibleTo<iterator_t<InnerRng>,
        iterator_t<iter_reference_t<iterator_t<Base>>>>;

    constexpr decltype(auto) operator*() const;
    constexpr iterator_t<Base> operator->() const
      requires has-arrow<iterator_t<Base>>;

    constexpr iterator& operator++();
    constexpr void operator++(int);
    constexpr iterator operator++(int)
      requires ~~is_reference_v<reference_t<iterator_t<Base>>> &&~~
        ref_is_glvalue && ForwardRange<Base> &&
        ForwardRange<iter_reference_t<iterator_t<Base>>>;

    constexpr iterator& operator--();
      requires ~~is_reference_v<reference_t<iterator_t<Base>>> &&~~
        ref_is_glvalue && BidirectionalRange<Base> &&
        BidirectionalRange<iter_reference_t<iterator_t<Base>>>;

    constexpr iterator operator--(int)
      requires ~~is_reference_v<reference_t<iterator_t<Base>>> &&~~
        ref_is_glvalue && BidirectionalRange<Base> &&
        BidirectionalRange<iter_reference_t<iterator_t<Base>>>;

    friend constexpr bool operator==(const iterator& x, const iterator& y)
      requires ~~is_reference_v<reference_t<iterator_t<Base>>> &&~~
        ref_is_glvalue && EqualityComparable<iterator_t<Base>> &&
        EqualityComparable<iterator_t<iter_reference_t<iterator_t<Base>>>>;

    friend constexpr bool operator!=(const iterator& x, const iterator& y)
      requires ~~is_reference_v<reference_t<iterator_t<Base>>> &&~~
        ref_is_glvalue && EqualityComparable<iterator_t<Base>> &&
        EqualityComparable<iterator_t<iter_reference_t<iterator_t<Base>>>>;
```

```
    friend constexpr decltype(auto) iter_move(const iterator& i)
      noexcept(~~see below~~noexcept(ranges::iter_move(i.inner_)));

    friend constexpr void iter_swap(const iterator& x, const iterator& y)
      noexcept(~~see below~~noexcept(ranges::iter_swap(x.inner_, y.inner_)));
  };
}~~}~~
```

2 ~~`join_view<R>::`~~`iterator::iterator_category` is defined as follows:

(2.1) — Let *OUTERC* denote `iterator_traits<iterator_t<Base>>::iterator_category`, and let *INNERC* denote `iterator_traits<iterator_t<iter_reference_t<iterator_t<Base>>>>::iterator_category`.

(2.2) — If `ref_is_glvalue` is `true`,

(2.2.1) — If `DerivedFrom<OUTERC, bidirectional_iterator_tag>` and `DerivedFrom<INNERC, bidirectional_iterator_tag>` are each satisfied, `iterator_category` denotes `bidirectional_iterator_tag`.

(2.2.2) — Otherwise, if `DerivedFrom<OUTERC, forward_iterator_tag>` and `DerivedFrom<INNERC, forward_iterator_tag>` are each satisfied, `iterator_category` denotes `forward_iterator_tag`.

(2.3) — If `Base` satisfies `BidirectionalRange`, and if `is_reference_v<reference_t<iterator_t<Base>>>` is `true`, and if `reference_t<iterator_t<Base>>` satisfies `BidirectionalRange`, then `iterator_category` is `ranges::bidirectional_iterator_tag`.

(2.4) — Otherwise, if `Base` satisfies `ForwardRange`, and if `is_reference_v<reference_t<iterator_t<Base>>>` is `true`, and if `reference_t<iterator_t<Base>>` satisfies `ForwardRange`, then `iterator_category` is `ranges::forward_iterator_tag`.

(2.5) — Otherwise, `iterator_category` ~~is~~ denotes ~~`ranges::`~~`input_iterator_tag`.

3 `iterator::iterator_concept` is defined as follows:

(3.1) — If `ref_is_glvalue` is `true`,

(3.1.1) — If `Base` and `iter_reference_t<iterator_t<Base>>` each model `BidirectionalRange`, then `iterator_concept` denotes `bidirectional_iterator_tag`.

(3.1.2) — Otherwise, if `Base` and `iter_reference_t<iterator_t<Base>>` each model `ForwardRange`, then `iterator_concept` denotes `forward_iterator_tag`.

(3.2) — Otherwise, `iterator_concept` denotes `input_iterator_tag`.

4 ~~`join_view<R>::`~~`iterator::difference_type` ~~is an alias for~~ denotes the type:

```
common_type_t<
  iter_difference~~_type~~_t<iterator_t<Base>>,
  iter_difference~~_type~~_t<iterator_t<iter_reference_t<iterator_t<Base>>>>>
```

29.8.13.2.1 join_view::iterator operations [range.adaptors.join__view.iterator.ops]

29.8.13.2.1.1 satisfy [range.adaptors.join__view.iterator.satisfy]

1 `join_view` iterators use the `satisfy` function to skip over empty inner ranges.

`constexpr void satisfy(); // exposition only`

2 *Effects:* ~~The join_view<R>::iterator::satisfy function is~~ Equivalent to:

```
auto update_inner = [this](reference_t<iterator_t<Base>> x) -> decltype(auto) {
  if constexpr (ref_is_glvalue) // x is a reference
    return (x); // (x) is an lvalue
  else
    return (parent_->inner_ = view::all(x));
};

for (; outer_ != ranges::end(parent_->base_); ++outer_) {
  auto&& inner = update_inner(*outer_)~~inner-range-update~~;
  inner_ = ranges::begin(inner);
  if (inner_ != ranges::end(inner))
    return;
}
if constexpr (ref_is_glvalue~~is_reference_v<reference_t<iterator_t<Base>>>~~)
  inner_ = iterator_t<iter_reference_t<iterator_t<Base>>>{};
```

where *`inner-range-update`* is equivalent to:

(2.1) — If `is_reference_v<reference_t<iterator_t<Base>>>` is `true`, `*outer_`.

(2.2) — Otherwise,

```
[this](auto&& x) -> decltype(auto) {
  return (parent_->inner_ = view::all(x));
}(*outer_)
```

29.8.13.2.1.2 `join_view::iterator` constructors [range.adaptors.join_view.iterator.ctor]

```
constexpr iterator(Parent& parent, iterator_t<R> outer)
```

1 *Effects:* Initializes `outer_` with `outer` and initializes `parent_` with ~~`&`~~`addressof(parent)`; then calls `satisfy()`.

```
constexpr iterator(iterator<!Const> i) requires Const &&
  ConvertibleTo<iterator_t<R>, iterator_t<Base>> &&
  ConvertibleTo<iterator_t<InnerRng>,
      iterator_t<iter_reference_t<iterator_t<Base>>>>;
```

2 *Effects:* Initializes `outer_` with `i.outer_`, initializes `inner_` with `i.inner_`, and initializes `parent_` with `i.parent_`.

29.8.13.2.1.3 `join_view::iterator::operator*` [range.adaptors.join_view.iterator.star]

```
constexpr decltype(auto) operator*() const;
```

1 *Returns:* `*inner_`.

29.8.13.2.1.4 `join_view::iterator::operator->` [range.adaptors.join_view.iterator.arrow]

```
constexpr iterator_t<Base> operator->() const
  requires has-arrow<iterator_t<Base>>;
```

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

29.8.13.2.1.5 `join_view::iterator::operator++` [range.adaptors.join_view.iterator.inc]

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

1 *Effects:* Equivalent to:

```
if (++inner_ == ranges::end(inner-range)) {
  ++outer_;
  satisfy();
}
return *this;
```

where *`inner-range`* is equivalent to:

(1.1) — If ~~`is_reference_v<reference_t<iterator_t<Base>>>`~~`ref_is_glvalue` is `true`, `*outer_`.

(1.2) — Otherwise, `parent_->inner_`.

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

2 *Effects:* Equivalent to: ~~`(void)`~~`++*this`.

`constexpr iterator operator++(int)`
`  requires ` ~~`is_reference_v<reference_t<iterator_t<Base>>> &&`~~
`    ref_is_glvalue && ForwardRange<Base> &&`
`    ForwardRange<iter_reference_t<iterator_t<Base>>>;`

3 *Effects:* Equivalent to:

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

29.8.13.2.1.6 `join_view::iterator::operator--` [range.adaptors.join_view.iterator.dec]

```
constexpr iterator& operator--();
  requires ~~is_reference_v<reference_t<iterator_t<Base>>> &&~~
    ref_is_glvalue && BidirectionalRange<Base> &&
    BidirectionalRange<iter_reference_t<iterator_t<Base>>>;
```

1 *Effects:* Equivalent to:

```
if (outer_ == ranges::end(parent_->base_))
  inner_ = ranges::end(*--outer_);
while (inner_ == ranges::begin(*outer_))
  inner_ = ranges::end(*--outer_);
--inner_;
return *this;
```

```
constexpr iterator operator--(int)
  requires ~~is_reference_v<reference_t<iterator_t<Base>>> &&~~
    ref_is_glvalue && BidirectionalRange<Base> &&
    BidirectionalRange<iter_reference_t<iterator_t<Base>>>;
```

2 *Effects:* Equivalent to:

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

29.8.13.2.2 `join_view::iterator` comparisons [range.adaptors.join_view.iterator.comp]

```
friend constexpr bool operator==(const iterator& x, const iterator& y)
  requires ~~is_reference_v<reference_t<iterator_t<Base>>> &&~~
    ref_is_glvalue && EqualityComparable<iterator_t<Base>> &&
    EqualityComparable<iterator_t<iter_reference_t<iterator_t<Base>>>>;
```

1 *Returns:* `x.outer_ == y.outer_ && x.inner_ == y.inner_`.

```
friend constexpr bool operator!=(const iterator& x, const iterator& y)
  requires ~~is_reference_v<reference_t<iterator_t<Base>>> &&~~
    ref_is_glvalue && EqualityComparable<iterator_t<Base>> &&
    EqualityComparable<iterator_t<iter_reference_t<iterator_t<Base>>>>;
```

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

29.8.13.2.3 `join_view::iterator` non-member functions [range.adaptors.join_view.iterator.nonmember]

```
friend constexpr decltype(auto) iter_move(const iterator& i)
  noexcept(~~see below~~noexcept(ranges::iter_move(i.inner_)));
```

1 *Returns:* `ranges::iter_move(i.inner_)`.

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

```
~~noexcept(ranges::iter_move(i.inner_))~~
```

```
friend constexpr void iter_swap(const iterator& x, const iterator& y)
  noexcept(~~see below~~noexcept(ranges::iter_swap(x.inner_, y.inner_)));
```

3 *Returns:* `ranges::iter_swap(x.inner_, y.inner_)`.

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

```
~~noexcept(ranges::iter_swap(x.inner_, y.inner_))~~
```

29.8.13.3 Class template `join_view::sentinel` [range.adaptors.join_view.sentinel]

1 ~~`join_view::sentinel` is an exposition-only type.~~

```
namespace std::ranges { ~~namespace ranges {~~
  template<class R>
  template<bool Const>
  struct join_view<R>::sentinel { // exposition only
  private:
    using Parent = conditional_t<Const, const join_view, join_view>;
```

```
      using Base    = conditional_t<Const, const R, R>;
      sentinel_t<Base> end_ {};
    public:
      sentinel() = default;

      constexpr explicit sentinel(Parent& parent);
      constexpr sentinel(sentinel<!Const> s) requires Const &&
          ConvertibleTo<sentinel_t<R>, sentinel_t<Base>>;

      friend constexpr bool operator==(const iterator<Const>& x, const sentinel& y);
      friend constexpr bool operator==(const sentinel& x, const iterator<Const>& y);
      friend constexpr bool operator!=(const iterator<Const>& x, const sentinel& y);
      friend constexpr bool operator!=(const sentinel& x, const iterator<Const>& y);
    };
  }}
```

29.8.13.3.1 join_view::sentinel operations [range.adaptors.join_view.sentinel.ops]

```
constexpr explicit sentinel(Parent& parent);
```

1 *Effects:* Initializes `end_` with `ranges::end(parent.base_)`.

```
constexpr sentinel(sentinel<!Const> s) requires Const &&
  ConvertibleTo<sentinel_t<R>, sentinel_t<Base>>;
```

2 *Effects:* Initializes `end_` with `s.end_`.

```
friend constexpr bool operator==(const iterator<Const>& x, const sentinel& y);
```

3 *Returns:* `x.outer_ == y.end_`.

```
friend constexpr bool operator==(const sentinel& x, const iterator<Const>& y);
```

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

```
friend constexpr bool operator!=(const iterator<Const>& x, const sentinel& y);
```

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

```
friend constexpr bool operator!=(const sentinel& x, const iterator<Const>& y);
```

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

29.8.14 view::join [range.adaptors.join]

1 The name `view::join` denotes a range adaptor object (29.8.1). ~~Let `E` be an expression such that type `T` is `decltype((E))`. Then~~ The expression `view::join(E)` for some subexpression `E` is expression-equivalent to `join_view{E}`. if the following is satisfied:

```
InputRange<T> &&
InputRange<reference_t<iterator_t<T>>> &&
(is_reference_v<reference_t<iterator_t<T>>> ||
 View<value_type_t<iterator_t<T>>)
```

Otherwise, `view::join(E)` is ill-formed.

29.8.15 Class template empty_view [range.adaptors.empty_view]

1 ~~The purpose of~~ `empty_view` ~~is to~~ produces ~~an empty range~~ a `View` of no elements of a particular type.

2 [*Example:*

```
empty_view<int> e;
static_assert(ranges::empty(e));
static_assert(0 == e.size());
```

— *end example*]

```
namespace std::ranges { namespace ranges {
  template<class T>
    requires is_object_v<T>
  class empty_view : public view_interface<empty_view<T>> {
  public:
```

```
    constexpr static T* begin() noexcept; { return nullptr; }
    constexpr static T* end() noexcept; { return nullptr; }
    constexpr static ptrdiff_t size() noexcept; { return 0; }
    constexpr static T* data() noexcept; { return nullptr; }

    constexpr static bool empty() noexcept { return true; }

    friend constexpr T* begin(const empty_view&&) noexcept { return nullptr; }
    friend constexpr T* end(const empty_view&&) noexcept { return nullptr; }
  };
}}
```

29.8.16 Class template `single_view` [range.adaptors.single_view]

1 ~~The purpose of~~ `single_view` ~~is to~~ produces a ~~range~~`View` that contains exactly one element of a specified value.

2 [*Example:*

```
single_view s{4};
for (int i : s)
  cout << i; // prints 4
```

— *end example*]

```
namespace std::ranges { namespace ranges {
  template<CopyConstructible T>
    requires is_object_v<T>
  class single_view : public view_interface<single_view<T>> {
  private:
    semiregular<T> value_; // exposition only
  public:
    single_view() = default;
    constexpr explicit single_view(const T& t);
    constexpr explicit single_view(T&& t);
    template<class... Args>
      requires Constructible<T, Args...>
    constexpr single_view(in_place_t, Args&&... args);

    constexpr const T* begin() const noexcept;
    constexpr const T* end() const noexcept;
    constexpr static ptrdiff_t size() noexcept;
    constexpr const T* data() const noexcept;
  };

  template<class T>
    requires CopyConstructible<decay_t<T>>
  explicit single_view(T&&) -> single_view<decay_t<T>>;
}}
```

29.8.16.1 `single_view` operations [range.adaptors.single_view.ops]

```
constexpr explicit single_view(const T& t);
```

1 *Effects:* Initializes `value_` with `t`.

```
constexpr explicit single_view(T&& t);
```

2 *Effects:* Initializes `value_` with `std::move(t)`.

```
template<class... Args>
constexpr single_view(in_place_t, Args&&... args);
```

3 *Effects:* Initializes `value_` as if by `value_{in_place, std::forward<Args>(args)...}`.

```
constexpr const T* begin() const noexcept;
```

4 ~~*Requires:* bool(value_)~~

5 ~~*Returns:*~~ *Effects:* Equivalent to: `return value_.operator->();`~~.~~

`constexpr const T* end() const noexcept;`

6 ~~*Requires:* bool(value_)~~

7 ~~*Returns:*~~ *Effects:* Equivalent to: `return value_.operator->() + 1;`~~.~~

`constexpr static ptrdiff_t size() noexcept;`

8 ~~*Requires:* bool(value_)~~

9 *Returns:* `1`.

`constexpr const T* data() const noexcept;`

10 ~~*Requires:* bool(value_)~~

11 ~~*Returns:*~~ *Effects:* Equivalent to: `return begin();`~~.~~

29.8.17 `view::single` [range.adaptors.single]

1 The name `view::single` denotes a customization point object ([customization.point.object]). ~~Let `E` be an expression such that its cv-unqualified type is `I`. Then~~ The expression `view::single(E)` for some subexpression `E` is expression-equivalent to `single_view{E}`. ~~if `CopyConstructible<I>` is satisfied. Otherwise, `view::single(E)` is ill-formed.~~

29.8.18 Class template `split_view` [range.adaptors.split_view]

1 The `split_view` takes a ~~range~~ `View` and a delimiter, and splits the ~~range~~ `View` into subranges on the delimiter. The delimiter can be a single element or a ~~range~~ `View` of elements.

2 [*Example:*

```
string str{"the quick brown fox"};
split_view sentence{str, ' '};
for (auto word : sentence) {
  for (char ch : word)
    cout << ch;
  cout << " *";
}
// The above prints: the *quick *brown *fox *
```

— *end example*]

```
namespace std::ranges { namespace ranges {
  template<class R>
  concept tiny-range = // exposition only
    SizedRange<R> && requires {
      requires remove_reference_t<R>::size() <= 1;
    };

  template<InputRange Rng, ForwardRange Pattern>
    requires View<Rng> && View<Pattern> &&
      IndirectlyComparable<iterator_t<Rng>, iterator_t<Pattern>> &&
      (ForwardRange<Rng> || tiny-range<Pattern>)
  class split_view {
  private:
    Rng base_ {}; // exposition only
    Pattern pattern_ {}; // exposition only
    iterator_t<Rng> current_ {}; // exposition only, only present if !ForwardRange<Rng>
    template<bool Const> struct outer_iterator; // exposition only
    template<bool Const> struct outer_sentinel; // exposition only
    template<bool Const> struct inner_iterator; // exposition only
    template<bool Const> struct inner_sentinel; // exposition only
  public:
    split_view() = default;
    constexpr split_view(Rng base, Pattern pattern);
```

```
    template<InputRange O, ForwardRange P>
      requires ViewableRange<O> && ViewableRange<P> &&
        Constructible<Rng, all_view<O>> &&
        Constructible<Pattern, all_view<P>>
    constexpr split_view(O&& o, P&& p);

    template<InputRange O>
      requires ViewableRange<O> &&
        Constructible<Rng, all_view<O>> &&
        Constructible<Pattern, single_view<iter_value~~_type~~_t<iterator_t<O>>>>
    constexpr split_view(O&& o, iter_value~~_type~~_t<iterator_t<O>> e);

    constexpr auto begin();
    constexpr auto begin() requires ForwardRange<Rng>;
    constexpr auto begin() const requires ForwardRange<const Rng>;

    constexpr auto end()
    constexpr auto end() const requires ForwardRange<const Rng>;

    constexpr auto end()
      requires ForwardRange<Rng> && CommonRange<Rng>;
    constexpr auto end() const
      requires ForwardRange<const Rng> && CommonRange<const Rng>;
  };

  template<~~InputRange~~class O, ~~ForwardRange~~class P>
    ~~requires ViewableRange<O> && ViewableRange<P> &&~~
      ~~IndirectlyComparable<iterator_t<O>, iterator_t<P>> &&~~
      ~~(ForwardRange<O> || tiny-range<P>)~~
  split_view(O&&, P&&) -> split_view<all_view<O>, all_view<P>>;

  template<InputRange O>
    ~~requires ViewableRange<O> &&~~
      ~~IndirectlyComparable<iterator_t<Rng>, const value_type_t<iterator_t<Rng>>*> &&~~
      ~~CopyConstructible<value_type_t<iterator_t<O>>>~~
  split_view(O&&, iter_value~~_type~~_t<iterator_t<O>>)
    -> split_view<all_view<O>, single_view<iter_value~~_type~~_t<iterator_t<O>>>>;
}~~}~~
```

29.8.18.1 split_view operations [range.adaptors.split__view.ops]

29.8.18.1.1 split_view constructors [range.adaptors.split__view.ctor]

```
constexpr split_view(Rng base, Pattern pattern);
```

1 *Effects:* Initializes `base_` with `std::move(base)` and initializes `pattern_` with `std::move(pattern)`.

```
template<InputRange O, ForwardRange P>
  requires ViewableRange<O> && ViewableRange<P> &&
    Constructible<Rng, all_view<O>> &&
    Constructible<Pattern, all_view<P>>
constexpr split_view(O&& o, P&& p);
```

2 *Effects:* Delegates to `split_view{view::all(std::forward<O>(o)), view::all(std::forward<P>(p))}`.

```
template<InputRange O>
  requires ViewableRange<O> &&
    Constructible<Rng, all_view<O>> &&
    Constructible<Pattern, single_view<iter_value~~_type~~_t<iterator_t<O>>>>
constexpr split_view(O&& o, iter_value~~_type~~_t<iterator_t<O>> e);
```

3 *Effects:* Delegates to `split_view{view::all(std::forward<O>(o)), single_view{std::move(e)}}`.

29.8.18.1.2 split_view range begin [range.adaptors.split__view.begin]

```
constexpr auto begin();
```

1 *Effects:* Equivalent to:

```
    current_ = ranges::begin(base_);
    return iterator{*this};
```

```
constexpr auto begin() requires ForwardRange<Rng>;
constexpr auto begin() const requires ForwardRange<Rng>;
```

2 *Effects:* Equivalent to: `return outer_iterator<`~~`simple-view<R>`~~`C>{*this, ranges::begin(base_)};` ~~and `return outer_iterator<true>{*this, ranges::begin(base_)};`~~ where *C* is `simple-view<R>` or `true` for the first and second overloads, respectively.

29.8.18.1.3 split_view range end [range.adaptors.split_view.end]

```
constexpr auto end()
constexpr auto end() const requires ForwardRange<Rng>;
```

1 *Effects:* Equivalent to: `return outer_sentinel<`~~`simple-view<R>`~~`C>{*this};` ~~and `return outer_sentinel<true>{*this};`~~ where *C* is `simple-view<R>` or `true` for the first and second overloads, respectively.

```
constexpr auto end()
  requires ForwardRange<Rng> && CommonRange<Rng>;
constexpr auto end() const
  requires ForwardRange<Rng> && CommonRange<Rng>;
```

2 *Effects:* Equivalent to: `return outer_iterator<`~~`simple-view<R>`~~`C>{*this, ranges::end(base_)};` ~~and `return outer_iterator<true>{*this, ranges::end(base_)};`~~ where *C* is `simple-view<R>` or `true` for the first and second overload, respectively.

29.8.18.2 Class template split_view::outer_iterator [range.adaptors.split_view.outer_iterator]

1 [*Note:* `split_view::outer_iterator` is an exposition-only type. — *end note*]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template<class Rng, class Pattern>
  template<bool Const>
  struct split_view<Rng, Pattern>::outer_iterator {
  private:
    using Parent = conditional_t<Const, const split_view, split_view>;
    using Base   = conditional_t<Const, const Rng, Rng>;
    iterator_t<Base> current_ {}; // Only present if ForwardRange<Rng> is satisfied
    Parent* parent_ = nullptr;
  public:
    using iterator_category = see below;
    using difference_type = iter_difference_type_t<iterator_t<Base>>;
    struct value_type;

    outer_iterator() = default;
    constexpr explicit outer_iterator(Parent& parent);
    constexpr outer_iterator(Parent& parent, iterator_t<Base> current)
      requires ForwardRange<Base>;
    constexpr outer_iterator(outer_iterator<!Const> i) requires Const &&
      ConvertibleTo<iterator_t<Rng>, iterator_t<Base>>;

    constexpr value_type operator*() const;

    constexpr outer_iterator& operator++();
    constexpr void operator++(int);
    constexpr outer_iterator operator++(int) requires ForwardRange<Base>;

    friend constexpr bool operator==(const outer_iterator& x, const outer_iterator& y)
      requires ForwardRange<Base>;
    friend constexpr bool operator!=(const outer_iterator& x, const outer_iterator& y)
      requires ForwardRange<Base>;
```

  };
}}~~}}~~

2 `split_view<Rng, Pattern>::outer_iterator::iterator_category` is defines as follows:

(2.1) — If `outer_iterator::Base` ~~satisfies~~ models `ForwardRange`, then `iterator_category` is `ranges::forward_iterator_tag`.

(2.2) — Otherwise, `iterator_category` is `ranges::input_iterator_tag`.

29.8.18.3 `split_view::outer_iterator` operations [range.adaptors.split__view.outer__iterator.ops]

29.8.18.3.1 `split_view::outer_iterator` constructors [range.adaptors.split__view.outer__iterator.ctor]

```
constexpr explicit outer_iterator(Parent& parent);
```

1 *Effects:* Initializes `parent_` with ~~`&`~~`addressof(parent)`.

```
constexpr outer_iterator(Parent& parent, iterator_t<Base> current)
  requires ForwardRange<Base>;
```

2 *Effects:* Initializes `parent_` with ~~`&`~~`addressof(parent)` and `current_` with `current`.

```
constexpr outer_iterator(outer_iterator<!Const> i) requires Const &&
ConvertibleTo<iterator_t<Rng>, iterator_t<Base>>;
```

3 *Effects:* Initializes `parent_` with `i.parent_` and `current_` with `i.current_`.

29.8.18.3.2 `split_view::outer_iterator::operator*` [range.adaptors.split__view.outer__iterator.star]

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

1 *Returns:* `value_type{*this}`.

29.8.18.3.3 `split_view::outer_iterator::operator++` [range.adaptors.split__view.outer__iterator.inc]

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

1 *Effects:* Equivalent to:

```
auto const end = ranges::end(parent_->base_);
if (current == end) return *this;
auto const [pbegin, pend] = subrange{parent_->pattern_};
do {
  auto [b,p] = mismatch(current, end, pbegin, pend);
  if (p != pend) continue; // The pattern didn't match
  current = bump(b, pbegin, pend, end); // skip the pattern
  break;
} while (++current != end);
return *this;
```

Where *`current`* is equivalent to:

(1.1) — If `Rng` ~~satisfies~~ models `ForwardRange`, `current_`.

(1.2) — Otherwise, `parent_->current_`.

and *`bump`*`(b, x, y, e)` is equivalent to:

(1.3) — If `Rng` ~~satisfies~~ models `ForwardRange`, `ranges::next(b, (int)(x == y), e)`.

(1.4) — Otherwise, `b`.

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

2 *Effects:* Equivalent to ~~`(void)`~~`++*this`.

```
constexpr outer_iterator operator++(int) requires ForwardRange<Base>;
```

3 *Effects:* Equivalent to:

```
auto tmp = *this;
```

```
    ++*this;
    return tmp;
```

29.8.18.3.4 split_view::outer_iterator non-member functions [range.adaptors.split_view.outer_iterator.nonmember]

```
friend constexpr bool operator==(const outer_iterator& x, const outer_iterator& y)
  requires ForwardRange<Base>;
```

1 *Effects:* Equivalent to:

```
return x.current_ == y.current_;
```

```
friend constexpr bool operator!=(const outer_iterator& x, const outer_iterator& y)
  requires ForwardRange<Base>;
```

2 *Effects:* Equivalent to:

```
return !(x == y);
```

29.8.18.4 Class template split_view::outer_sentinel [range.adaptors.split_view.outer_sentinel]

1 [*Note:* `split_view::outer_sentinel` is an exposition-only type. — *end note*]

```
namespace std { ~~namespace experimental {~~ namespace ranges { ~~inline namespace v1 {~~
  template<class Rng, class Pattern>
  template<bool Const>
  struct split_view<Rng, Pattern>::outer_sentinel {
  private:
    using Parent = conditional_t<Const, const split_view, split_view>;
    using Base   = conditional_t<Const, const Rng, Rng>;
    sentinel_t<Base> end_;
  public:
    outer_sentinel() = default;
    constexpr explicit outer_sentinel(Parent& parent);

    friend constexpr bool operator==(const outer_iterator<Const>& x, const outer_sentinel& y);
    friend constexpr bool operator==(const outer_sentinel& x, const outer_iterator<Const>& y);
    friend constexpr bool operator!=(const outer_iterator<Const>& x, const outer_sentinel& y);
    friend constexpr bool operator!=(const outer_sentinel& x, const outer_iterator<Const>& y);
  };
}}~~}}~~
```

29.8.18.4.1 split_view::outer_sentinel constructors [range.adaptors.split_view.outer_sentinel.ctor]

```
constexpr explicit outer_sentinel(Parent& parent);
```

1 *Effects:* Initializes `end_` with `ranges::end(parent.base_)`.

29.8.18.4.2 split_view::outer_sentinel non-member functions [range.adaptors.split_view.outer_sentinel.nonmember]

```
friend constexpr bool operator==(const outer_iterator<Const>& x, const outer_sentinel& y);
```

1 *Effects:* Equivalent to:

```
return current(x) == y.end_;
```

Where *`current`*`(x)` is equivalent to:

(1.1) — If `Rng` ~~satisfies~~ models `ForwardRange`, `x.current_`.

(1.2) — Otherwise, `x.parent_->current_`.

```
friend constexpr bool operator==(const outer_sentinel& x, const outer_iterator<Const>& y);
```

2 *Effects:* Equivalent to:

```
return y == x;
```

```
friend constexpr bool operator!=(const outer_iterator<Const>& x, const outer_sentinel& y);
```

3 *Effects:* Equivalent to:

```
return !(x == y);
```

```
friend constexpr bool operator!=(const outer_sentinel& x, const outer_iterator<Const>& y);
```

4 *Effects:* Equivalent to:

```
return !(y == x);
```

29.8.18.5 Class split_view::outer_iterator::value_type [range.adaptors.split__view.outer__iterator.value__type]

1 [*Note:* split_view::outer_iterator::value_type is an exposition-only type. — *end note*]

```
namespace std { ~~namespace experimental {~~ namespace ranges { ~~inline namespace v1 {~~
  template<class Rng, class Pattern>
  template<bool Const>
  struct split_view<Rng, Pattern>::outer_iterator<Const>::value_type {
  private:
    outer_iterator i_ {};
  public:
    value_type() = default;
    constexpr explicit value_type(outer_iterator i);

    constexpr auto begin() const;
    constexpr auto end() const;
  };
}}~~}}~~
```

29.8.18.5.1 split_view::outer_iterator::value_type constructors [range.adaptors.split__view.outer__iterator.value__type.ctor]

```
constexpr explicit value_type(outer_iterator i);
```

1 *Effects:* Initializes i_ with i.

29.8.18.5.2 split_view::outer_iterator::value_type range begin [range.adaptors.split__view.outer__iterator.value__type.begin]

```
constexpr auto begin() const;
```

1 *Effects:* Equivalent to:

```
return inner_iterator<Const>{i_};
```

29.8.18.5.3 split_view::outer_iterator::value_type range end [range.adaptors.split__view.outer__iterator.value__type.end]

```
constexpr auto end() const;
```

1 *Effects:* Equivalent to:

```
return inner_sentinel<Const>{};
```

29.8.18.6 Class template split_view::inner_iterator [range.adaptors.split__view.inner__iterator]

1 [*Note:* split_view::inner_iterator is an exposition-only type. — *end note*]

2 In the definition of split_view<Rng, Pattern>::inner_iterator below, *current*(i) is equivalent to:

(2.1) — If Rng ~~satisfies~~ models ForwardRange, i.current_.

(2.2) — Otherwise, i.parent_->current_.

```
namespace std { ~~namespace experimental {~~ namespace ranges { ~~inline namespace v1 {~~
  template<class Rng, class Pattern>
  template<bool Const>
  struct split_view<Rng, Pattern>::inner_iterator {
  private:
    using Base = conditional_t<Const, const Rng, Rng>;
```

```
      outer_iterator<Const> i_ {};
      bool zero_ = false;
    public:
      using iterator_category = iterator_category_t<outer_iterator<Const>>;
      using difference_type = iter_difference_type_t<iterator_t<Base>>;
      using value_type = iter_value_type_t<iterator_t<Base>>;

      inner_iterator() = default;
      constexpr explicit inner_iterator(outer_iterator<Const> i);

      constexpr decltype(auto) operator*() const;

      constexpr inner_iterator& operator++();
      constexpr void operator++(int);
      constexpr inner_iterator operator++(int) requires ForwardRange<Base>;

      friend constexpr bool operator==(const inner_iterator& x, const inner_iterator& y)
        requires ForwardRange<Base>;
      friend constexpr bool operator!=(const inner_iterator& x, const inner_iterator& y)
        requires ForwardRange<Base>;

      friend constexpr decltype(auto) iter_move(const inner_iterator& i)
        noexcept(see below);
      friend constexpr void iter_swap(const inner_iterator& x, const inner_iterator& y)
        noexcept(see below) requires IndirectlySwappable<iterator_t<Base>>;
    };
  }}}}
```

29.8.18.6.1 split_view::inner_iterator constructors [range.adaptors.split__view.inner__iterator.ctor]

```
constexpr explicit inner_iterator(outer_iterator<Const> i);
```

1 *Effects:* Initializes `i_` with `i`.

29.8.18.6.2 split_view::inner_iterator::operator* [range.adaptors.split__view.inner__iterator.star]

```
constexpr decltype(auto) operator*() const;
```

1 *Returns:* `*current(i_)`.

29.8.18.6.3 split_view::inner_iterator::operator++ [range.adaptors.split__view.inner__iterator.inc]

```
constexpr decltype(auto) operator++() const;
```

1 *Effects:* Equivalent to:

```
  ++current(i_);
  zero_ = true;
  return *this;
```

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

2 *Effects:* Equivalent to ~~`(void)`~~`++*this`.

```
constexpr inner_iterator operator++(int) requires ForwardRange<Base>;
```

3 *Effects:* Equivalent to:

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

29.8.18.6.4 split_view::inner_iterator comparisons [range.adaptors.split_view.inner_iterator.comp]

```
friend constexpr bool operator==(const inner_iterator& x, const inner_iterator& y)
  requires ForwardRange<Base>;
```

1 *Effects:* Equivalent to:

```
return x.i_.current_ == y.i_.current_;
```

```
friend constexpr bool operator!=(const inner_iterator& x, const inner_iterator& y)
  requires ForwardRange<Base>;
```

2 *Effects:* Equivalent to:

```
return !(x == y);
```

29.8.18.6.5 split_view::inner_iterator non-member functions [range.adaptors.split_view.inner_iterator.nonmember]

```
friend constexpr decltype(auto) iter_move(const inner_iterator& i)
  noexcept(see belownoexcept(ranges::iter_move(current(i.i_))));
```

1 *Returns:* `ranges::iter_move(current(i.i_))`.

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

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

```
friend constexpr void iter_swap(const inner_iterator& x, const inner_iterator& y)
  noexcept(see belownoexcept(ranges::iter_swap(current(x.i_), current(y.i_))))
  requires IndirectlySwappable<iterator_t<Base>>;
```

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

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

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

29.8.18.7 Class template split_view::inner_sentinel [range.adaptors.split_view.inner_sentinel]

1 [*Note:* `split_view::inner_sentinel` is an exposition-only type. — *end note*]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template<class Rng, class Pattern>
  template<bool Const>
  struct split_view<Rng, Pattern>::inner_sentinel {
    friend constexpr bool operator==(const inner_iterator<Const>& x, inner_sentinel);
    friend constexpr bool operator==(inner_sentinel x, const inner_iterator<Const>& y);
    friend constexpr bool operator!=(const inner_iterator<Const>& x, inner_sentinel y);
    friend constexpr bool operator!=(inner_sentinel x, const inner_iterator<Const>& y);
  };
}}}}
```

29.8.18.7.1 split_view::inner_sentinel comparisons [range.adaptors.split_view.inner_sentinel.comp]

```
friend constexpr bool operator==(const inner_iterator<Const>& x, inner_sentinel)
```

1 *Effects:* Equivalent to:

```
auto cur = x.i_.current();
auto end = ranges::end(x.i_.parent_->base_);
if (cur == end) return true;
auto [pcur, pend] = subrange{x.i_.parent_->pattern_};
if (pcur == pend) return x.zero_;
do {
  if (*cur != *pcur) return false;
  if (++pcur == pend) return true;
} while (++cur != end);
return false;
```

```
friend constexpr bool operator==(inner_sentinel x, const inner_iterator<Const>& y);
```

2 *Effects:* Equivalent to:

```
return y == x;
```

```
friend constexpr bool operator!=(const inner_iterator<Const>& x, inner_sentinel y);
```

3 *Effects:* Equivalent to:

```
return !(x == y);
```

```
friend constexpr bool operator!=(inner_sentinel x, const inner_iterator<Const>& y);
```

4 *Effects:* Equivalent to:

```
return !(y == x);
```

29.8.19 `view::split` [range.adaptors.split]

1 The name `view::split` denotes a range adaptor object (29.8.1). Let `E` and `F` be expressions such that their types are `T` and `U` respectively. Then the expression `view::split(E, F)` is expression-equivalent to:

(1.1) — `split_view{E, F}` if either of the following sets of requirements is satisfied:

(1.1.1) — `InputRange<T> && ForwardRange<U> && ViewableRange<T> && ViewableRange<U> && IndirectlyComparable<iterator_t<T>, iterator_t<U>> && (ForwardRange<T> || tiny-range<U>)`

(1.1.2) — `InputRange<T> && ViewableRange<T> && IndirectlyComparable<iterator_t<T>, const iter_value`~~`_type`~~`_t<iterator_t<T>>*> && CopyConstructible<iter_value`~~`_type`~~`_t<iterator_t<T>>> && ConvertibleTo<U, iter_value`~~`_type`~~`_t<iterator_t<T>>>`

(1.2) — Otherwise, `view::split(E, F)` is ill-formed.

29.8.20 `view::counted` [range.adaptors.counted]

1 The name `view::counted` denotes a customization point object ([customization.point.object]). Let `E` and `F` be expressions such that their decayed types are `T` and `U` respectively. Then the expression `view::counted(E, F)` is expression-equivalent to:

(1.1) — `subrange{E, E + F}` if `T` is a pointer to an object type, and if `U` is implicitly convertible to `ptrdiff_t`.

(1.2) — Otherwise, `subrange{counted_iterator(E, static_cast<iter_difference`~~`_type`~~`_t<T>>(F)), default_sentinel{}}` if `Iterator<T> && ConvertibleTo<U, iter_difference`~~`_type`~~`_t<T>>` is satisfied.

(1.3) — Otherwise, `view::counted(E, F)` is ill-formed.

29.8.21 Class template `common_view` [range.adaptors.common_view]

1 The `common_view` takes a range which has different types for its iterator and sentinel and turns it into an equivalent range where the iterator and sentinel have the same type.

2 [*Note:* `common_view` is useful for calling legacy algorithms that expect a range's iterator and sentinel types to be the same. — *end note*]

3 [*Example:*

```
// Legacy algorithm:
template<class ForwardIterator>
size_t count(ForwardIterator first, ForwardIterator last);

template<ForwardRange R>
void my_algo(R&& r) {
  auto&& common = common_view{r};
  auto cnt = count(common.begin(), common.end());
  // ...
}
```

— *end example*]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template<View Rng>
    requires !CommonRange<Rng>
  class common_view : public view_interface<common_view<Rng>> {
  private:
    Rng base_ {}; // exposition only
  public:
    common_view() = default;

    explicit constexpr common_view(Rng rng);

    template<ViewableRange O>
      requires !CommonRange<O> && Constructible<Rng, all_view<O>>
    explicit constexpr common_view(O&& o);

    constexpr Rng base() const;

    constexpr auto begin();
    constexpr auto begin() const requires Range<const Rng>;

    constexpr auto begin()
      requires RandomAccessRange<Rng> && SizedRange<Rng>;
    constexpr auto begin() const
      requires RandomAccessRange<const Rng> && SizedRange<const Rng>;

    constexpr auto end();
    constexpr auto end() const requires Range<const Rng>;

    constexpr auto end()
      requires RandomAccessRange<Rng> && SizedRange<Rng>;
    constexpr auto end() const
      requires RandomAccessRange<const Rng> && SizedRange<Rng>;

    constexpr auto size() const requires SizedRange<const Rng>;
  };

  template<ViewableRange O>
    requires !CommonRange<O>
  common_view(O&&) -> common_view<all_view<O>>;
}}}}
```

29.8.21.1 `common_view` operations [range.adaptors.common_view.ops]

29.8.21.1.1 `common_view` constructors [range.adaptors.common_view.ctor]

```
explicit constexpr common_view(Rng base);
```

1 *Effects:* Initializes `base_` with `std::move(base)`.

```
template<ViewableRange O>
  requires !CommonRange<O> && Constructible<Rng, all_view<O>>
explicit constexpr common_view(O&& o);
```

2 *Effects:* Initializes `base_` with `view::all(std::forward<O>(o))`.

29.8.21.1.2 `common_view` conversion [range.adaptors.common_view.conv]

```
constexpr Rng base() const;
```

1 *Returns:* `base_`.

29.8.21.1.3 `common_view` begin [range.adaptors.common_view.begin]

```
constexpr auto begin();
constexpr auto begin() const requires Range<const Rng>;
```

1 *Effects:* Equivalent to:

```
return common_iterator<iterator_t<Rng>, sentinel_t<Rng>>(ranges::begin(base_));
```

and

```
return common_iterator<iterator_t<const Rng>, sentinel_t<const Rng>>(ranges::begin(base_));
```

for the first and second overloads, respectively.

```
constexpr auto begin()
  requires RandomAccessRange<Rng> && SizedRange<Rng>;
constexpr auto begin() const
  requires RandomAccessRange<const Rng> && SizedRange<const Rng>;
```

2 *Effects:* Equivalent to:

```
return ranges::begin(base_);
```

29.8.21.1.4 `common_view` end [range.adaptors.common_view.end]

```
constexpr auto end();
constexpr auto end() const requires Range<const Rng>;
```

1 *Effects:* Equivalent to:

```
return common_iterator<iterator_t<Rng>, sentinel_t<Rng>>(ranges::end(base_));
```

and

```
return common_iterator<iterator_t<const Rng>, sentinel_t<const Rng>>(ranges::end(base_));
```

for the first and second overloads, respectively.

```
constexpr auto end()
  requires RandomAccessRange<Rng> && SizedRange<Rng>;
constexpr auto end() const
  requires RandomAccessRange<const Rng> && SizedRange<const Rng>;
```

2 *Effects:* Equivalent to:

```
return ranges::begin(base_) + ranges::size(base_);
```

29.8.21.1.5 `common_view` size [range.adaptors.common_view.size]

```
constexpr auto size() const requires SizedRange<const Rng>;
```

1 *Effects:* Equivalent to: `return ranges::size(base_);`

29.8.22 `view::common` [range.adaptors.common]

1 The name `view::common` denotes a range adaptor object (29.8.1). Let `E` be an expression such that `U` is `decltype((E))`. Then the expression `view::common(E)` is expression-equivalent to:

(1.1) — If `ViewableRange<U> && CommonRange<U>` is satisfied, `view::all(E)`.

(1.2) — Otherwise, if `ViewableRange<U>` is satisfied, `common_view{E}`.

(1.3) — Otherwise, `view::common(E)` is ill-formed.

29.8.23 Class template `reverse_view` [range.adaptors.reverse_view]

1 The `reverse_view` takes a bidirectional range and produces another range that iterates the same elements in reverse order.

2 [*Example:*

```
vector<int> is {0,1,2,3,4};
reverse_view rv {is};
for (int i : rv)
  cout << i << ' '; // prints: 4 3 2 1 0
```

— *end example*]

```
namespace std { ~~namespace experimental {~~ namespace ranges { ~~inline namespace v1 {~~
  template<View Rng>
    requires BidirectionalRange<Rng>
  class reverse_view : public view_interface<reverse_view<Rng>> {
  private:
    Rng base_ {}; // exposition only
```

```
  public:
    reverse_view() = default;

    explicit constexpr reverse_view(Rng rng);

    template<ViewableRange O>
      requires BidirectionalRange<O> && Constructible<Rng, all_view<O>>
    explicit constexpr reverse_view(O&& o);

    constexpr Rng base() const;

    constexpr auto begin();
    constexpr auto begin() requires CommonRange<Rng>;
    constexpr auto begin() const requires CommonRange<const Rng>;

    constexpr auto end();
    constexpr auto end() const requires CommonRange<const Rng>;

    constexpr auto size() const requires SizedRange<const Rng>;
  };

  template<ViewableRange O>
    requires BidirectionalRange<O>
  reverse_view(O&&) -> reverse_view<all_view<O>>;
}}}}
```

29.8.23.1 `reverse_view` operations [range.adaptors.reverse_view.ops]

29.8.23.1.1 `reverse_view` constructors [range.adaptors.reverse_view.ctor]

```
explicit constexpr reverse_view(Rng base);
```

1 *Effects:* Initializes `base_` with `std::move(base)`.

```
template<ViewableRange O>
  requires BidirectionalRange<O> && Constructible<Rng, all_view<O>>
explicit constexpr reverse_view(O&& o);
```

2 *Effects:* Initializes `base_` with `view::all(std::forward<O>(o))`.

29.8.23.1.2 `reverse_view` conversion [range.adaptors.reverse_view.conv]

```
constexpr Rng base() const;
```

1 *Returns:* `base_`.

29.8.23.1.3 `reverse_view` begin [range.adaptors.reverse_view.begin]

```
constexpr auto begin();
```

1 *Effects:* Equivalent to:

```
return reverse_iterator{ranges::next(ranges::begin(base_), ranges::end(base_))};
```

2 *Remarks:* In order to provide the amortized constant time complexity required by the `Range` concept, this function caches the result within the `reverse_view` for use on subsequent calls.

```
constexpr auto begin() requires CommonRange<Rng>;
constexpr auto begin() const requires CommonRange<const Rng>;
```

3 *Effects:* Equivalent to:

```
return reverse_iterator{ranges::end(base_)};
```

29.8.23.1.4 `reverse_view` end [range.adaptors.reverse_view.end]

```
constexpr auto end() requires CommonRange<Rng>;
constexpr auto end() const requires CommonRange<const Rng>;
```

1 *Effects:* Equivalent to: `return reverse_iteratorranges::begin(base_);`

29.8.23.1.5 `reverse_view` size [range.adaptors.reverse_view.size]

```
constexpr auto size() const requires SizedRange<const Rng>;
```

1 *Effects:* Equivalent to:

```
return ranges::size(base_);
```

29.8.24 `view::reverse` [range.adaptors.reverse]

1 The name `view::reverse` denotes a range adaptor object (29.8.1). Let `E` be an expression such that `U` is `decltype((E))`. Then the expression `view::reverse(E)` is expression-equivalent to:

(1.1) — If `ViewableRange<U> && BidirectionalRange<U>` is satisfied, `reverse_view{E}`.

(1.2) — Otherwise, `view::reverse(E)` is ill-formed.

[Editor's note: Incorporate [algorithms] from the Ranges TS into [algorithms] as follows:]

30 Algorithms library [algorithms]

30.1 General [algorithms.general]

1 This Clause describes components that C++ programs may use to perform algorithmic operations on containers (Clause 27) and other sequences.

2 The following subclauses describe components for non-modifying sequence operations, mutating sequence operations, sorting and related operations, and algorithms from the ISO C library, as summarized in Table 91.

Table 91 — Algorithms library summary

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

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

```
#include <initializer_list>

namespace std {
  // 30.5, non-modifying sequence operations
  // 30.5.1, all of
  template<class InputIterator, class Predicate>
    constexpr bool all_of(InputIterator first, InputIterator last, Predicate pred);
  template<class ExecutionPolicy, class ForwardIterator, class Predicate>
    bool all_of(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                ForwardIterator first, ForwardIterator last, Predicate pred);

  namespace ranges {
    template<InputIterator I, Sentinel<I> S, class Proj = identity,
        IndirectUnaryPredicate<projected<I, Proj>> Pred>
      constexpr bool all_of(I first, S last, Pred pred, Proj proj = Proj{});
    template<InputRange Rng, class Proj = identity,
        IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
      constexpr bool all_of(Rng&& rng, Pred pred, Proj proj = Proj{});
  }

  // 30.5.2, any of
  template<class InputIterator, class Predicate>
    constexpr bool any_of(InputIterator first, InputIterator last, Predicate pred);
  template<class ExecutionPolicy, class ForwardIterator, class Predicate>
    bool any_of(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                ForwardIterator first, ForwardIterator last, Predicate pred);
```

```
namespace ranges {
  template<InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr bool any_of(I first, S last, Pred pred, Proj proj = Proj{});
  template<InputRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    constexpr bool any_of(Rng&& rng, Pred pred, Proj proj = Proj{});
}

// 30.5.3, none of
template<class InputIterator, class Predicate>
  constexpr bool none_of(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  bool none_of(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
               ForwardIterator first, ForwardIterator last, Predicate pred);

namespace ranges {
  template<InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr bool none_of(I first, S last, Pred pred, Proj proj = Proj{});
  template<InputRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    constexpr bool none_of(Rng&& rng, Pred pred, Proj proj = Proj{});
}

// 30.5.4, for each
template<class InputIterator, class Function>
  constexpr Function for_each(InputIterator first, InputIterator last, Function f);
template<class ExecutionPolicy, class ForwardIterator, class Function>
  void for_each(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                ForwardIterator first, ForwardIterator last, Function f);

namespace ranges {
  template<class I, class F>
  struct for_each_result {
    I in;
    F fun;
  };

  template<InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryInvocable<projected<I, Proj>> Fun>
    tagged_pair<tag::in(I), tag::fun(Fun)>
    constexpr for_each_result<I, Fun>
      for_each(I first, S last, Fun f, Proj proj = Proj{});
  template<InputRange Rng, class Proj = identity,
      IndirectUnaryInvocable<projected<iterator_t<Rng>, Proj>> Fun>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::fun(Fun)>
    constexpr for_each_result<safe_iterator_t<Rng>, Fun>
      for_each(Rng&& rng, Fun f, Proj proj = Proj{});
}

template<class InputIterator, class Size, class Function>
  constexpr InputIterator for_each_n(InputIterator first, Size n, Function f);
template<class ExecutionPolicy, class ForwardIterator, class Size, class Function>
  ForwardIterator for_each_n(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                             ForwardIterator first, Size n, Function f);

// 30.5.5, find
template<class InputIterator, class T>
  constexpr InputIterator find(InputIterator first, InputIterator last,
                               const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  ForwardIterator find(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                       ForwardIterator first, ForwardIterator last,
                       const T& value);
```

```
template<class InputIterator, class Predicate>
  constexpr InputIterator find_if(InputIterator first, InputIterator last,
                                  Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator find_if(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                          ForwardIterator first, ForwardIterator last,
                          Predicate pred);
template<class InputIterator, class Predicate>
  constexpr InputIterator find_if_not(InputIterator first, InputIterator last,
                                      Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator find_if_not(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                              ForwardIterator first, ForwardIterator last,
                              Predicate pred);

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

// 30.5.6, find end
template<class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
  constexpr ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    find_end(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1,
         class ForwardIterator2, class BinaryPredicate>
  ForwardIterator1
    find_end(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);

namespace ranges {
  template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2,
      class Pred = ranges::equal_to<>, class Proj1 = identity, class Proj2 = identity>
```

```
      IndirectRelation<I2, projected<I1, Proj>> Pred = equal_to<>>
    requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
    constexpr I1 subrange<I1>
      find_end(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<ForwardRange Rng1, ForwardRange Rng2, class Proj = identity,
      IndirectRelation<iterator_t<Rng2>,
        projected<iterator_t<Rng>, Proj>> Pred = equal_to<>>
      class Pred = ranges::equal_to<>, class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
    constexpr safe_iterator_t<Rng1> safe_subrange_t<Rng1>
      find_end(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

// 30.5.7, find first
template<class InputIterator, class ForwardIterator>
  constexpr InputIterator
    find_first_of(InputIterator first1, InputIterator last1,
                  ForwardIterator first2, ForwardIterator last2);
template<class InputIterator, class ForwardIterator, class BinaryPredicate>
  constexpr InputIterator
    find_first_of(InputIterator first1, InputIterator last1,
                  ForwardIterator first2, ForwardIterator last2,
                  BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    find_first_of(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                  ForwardIterator1 first1, ForwardIterator1 last1,
                  ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1,
         class ForwardIterator2, class BinaryPredicate>
  ForwardIterator1
    find_first_of(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                  ForwardIterator1 first1, ForwardIterator1 last1,
                  ForwardIterator2 first2, ForwardIterator2 last2,
                  BinaryPredicate pred);

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

// 30.5.8, adjacent find
template<class ForwardIterator>
  constexpr ForwardIterator
    adjacent_find(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class BinaryPredicate>
  constexpr ForwardIterator
    adjacent_find(ForwardIterator first, ForwardIterator last,
                  BinaryPredicate pred);
```

```
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator
    adjacent_find(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                  ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class BinaryPredicate>
  ForwardIterator
    adjacent_find(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                  ForwardIterator first, ForwardIterator last,
                  BinaryPredicate pred);

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

// 30.5.9, count
template<class InputIterator, class T>
  constexpr typename iterator_traits<InputIterator>::difference_type
    count(InputIterator first, InputIterator last, const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  typename iterator_traits<ForwardIterator>::difference_type
    count(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
          ForwardIterator first, ForwardIterator last, const T& value);
template<class InputIterator, class Predicate>
  constexpr typename iterator_traits<InputIterator>::difference_type
    count_if(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  typename iterator_traits<ForwardIterator>::difference_type
    count_if(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
             ForwardIterator first, ForwardIterator last, Predicate pred);

namespace ranges {
  template<InputIterator I, Sentinel<I> S, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T*>
    constexpr iter_difference~~_type~~_t<I>
      count(I first, S last, const T& value, Proj proj = Proj{});
  template<InputRange Rng, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
    constexpr iter_difference~~_type~~_t<iterator_t<Rng>>
      count(Rng&& rng, const T& value, Proj proj = Proj{});
  template<InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr iter_difference~~_type~~_t<I>
      count_if(I first, S last, Pred pred, Proj proj = Proj{});
  template<InputRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    constexpr iter_difference~~_type~~_t<iterator_t<Rng>>
      count_if(Rng&& rng, Pred pred, Proj proj = Proj{});
}

// 30.5.10, mismatch
template<class InputIterator1, class InputIterator2>
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2);
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
```

```
            InputIterator2 first2, BinaryPredicate pred);
template<class InputIterator1, class InputIterator2>
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2);
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2,
             BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);

namespace ranges {
  template<class I1, class I2>
  struct mismatch_result {
    I1 in1;
    I2 in2;
  };

  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      class Proj1 = identity, class Proj2 = identity,
      IndirectRelation<projected<I1, Proj1>, projected<I2, Proj2>> Pred = ranges::equal_to<>>
    tagged_pair<tag::in1(I1), tag::in2(I2)>
    constexpr mismatch_result<I1, I2>
      mismatch(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<InputRange Rng1, InputRange Rng2,
      class Proj1 = identity, class Proj2 = identity,
      IndirectRelation<projected<iterator_t<Rng1>, Proj1>,
        projected<iterator_t<Rng2>, Proj2>> Pred = ranges::equal_to<>>
    tagged_pair<tag::in1(safe_iterator_t<Rng1>),
                tag::in2(safe_iterator_t<Rng2>)>
    constexpr mismatch_result<safe_iterator_t<Rng1>, safe_iterator_t<Rng2>>
      mismatch(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

// 30.5.11, equal
template<class InputIterator1, class InputIterator2>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2);
```

```
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, BinaryPredicate pred);
template<class InputIterator1, class InputIterator2>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, InputIterator2 last2);
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, InputIterator2 last2,
                       BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  bool equal(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  bool equal(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  bool equal(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  bool equal(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);

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

// 30.5.12, is permutation
template<class ForwardIterator1, class ForwardIterator2>
  constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                                ForwardIterator2 first2);
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
  constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                                ForwardIterator2 first2, BinaryPredicate pred);
template<class ForwardIterator1, class ForwardIterator2>
  constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                                ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
  constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                                ForwardIterator2 first2, ForwardIterator2 last2,
                                BinaryPredicate pred);

namespace ranges {
  template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
      Sentinel<I2> S2, class Pred = ranges::equal_to<>, class Proj1 = identity,
      class Proj2 = identity>
    requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
    constexpr bool is_permutation(I1 first1, S1 last1, I2 first2, S2 last2,
```

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

// 30.5.13, search
template<class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator1
    search(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
  constexpr ForwardIterator1
    search(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2,
           BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    search(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
           ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  ForwardIterator1
    search(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
           ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2,
           BinaryPredicate pred);

namespace ranges {
  template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
      Sentinel<I2> S2, class Pred = ranges::equal_to<>,
      class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
    constexpr subrange<I1> I1
      search(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<ForwardRange Rng1, ForwardRange Rng2, class Pred = ranges::equal_to<>,
      class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
    constexpr safe_subrange_t<Rng1> safe_iterator_t<Rng1>
      search(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

template<class ForwardIterator, class Size, class T>
  constexpr ForwardIterator
    search_n(ForwardIterator first, ForwardIterator last,
             Size count, const T& value);
template<class ForwardIterator, class Size, class T, class BinaryPredicate>
  constexpr ForwardIterator
    search_n(ForwardIterator first, ForwardIterator last,
             Size count, const T& value,
             BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Size, class T>
  ForwardIterator
    search_n(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
             ForwardIterator first, ForwardIterator last,
             Size count, const T& value);
```

```
template<class ExecutionPolicy, class ForwardIterator, class Size, class T,
         class BinaryPredicate>
  ForwardIterator
    search_n(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
             ForwardIterator first, ForwardIterator last,
             Size count, const T& value,
             BinaryPredicate pred);

namespace ranges {
  template<ForwardIterator I, Sentinel<I> S, class T,
      class Pred = ranges::equal_to<>, class Proj = identity>
    requires IndirectlyComparable<I, const T*, Pred, Proj>
    constexpr subrange<I>
      search_n(I first, S last, iter_difference_type_t<I> count,
               const T& value, Pred pred = Pred{}, Proj proj = Proj{});
  template<ForwardRange Rng, class T, class Pred = ranges::equal_to<>,
      class Proj = identity>
    requires IndirectlyComparable<iterator_t<Rng>, const T*, Pred, Proj>
    constexpr safe_iteratorsubrange_t<Rng>
      search_n(Rng&& rng, iter_difference_type_t<iterator_t<Rng>> count,
               const T& value, Pred pred = Pred{}, Proj proj = Proj{});
}

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

// 30.6, mutating sequence operations
// 30.6.1, copy
template<class InputIterator, class OutputIterator>
  constexpr OutputIterator copy(InputIterator first, InputIterator last,
                                OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 copy(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                        ForwardIterator1 first, ForwardIterator1 last,
                        ForwardIterator2 result);

namespace ranges {
  template<class I, class O>
  struct copy_result {
    I in;
    O out;
  };

  template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
    requires IndirectlyCopyable<I, O>
    tagged_pair<tag::in(I), tag::out(O)>
    constexpr copy_result<I, O>
      copy(I first, S last, O result);
  template<InputRange Rng, WeaklyIncrementable O>
    requires IndirectlyCopyable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    constexpr copy_result<safe_iterator_t<Rng>, O>
      copy(Rng&& rng, O result);
}

template<class InputIterator, class Size, class OutputIterator>
  constexpr OutputIterator copy_n(InputIterator first, Size n,
                                  OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class Size,
         class ForwardIterator2>
  ForwardIterator2 copy_n(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                          ForwardIterator1 first, Size n,
                          ForwardIterator2 result);
```

```
namespace ranges {
  template<class I, class O>
  using copy_n_result = copy_result<I, O>;

  template<InputIterator I, WeaklyIncrementable O>
    requires IndirectlyCopyable<I, O>
    tagged_pair<tag::in(I), tag::out(O)>
    constexpr copy_n_result<I, O>
      copy_n(I first, iter_difference_type_t<I> n, O result);
}

template<class InputIterator, class OutputIterator, class Predicate>
  constexpr OutputIterator copy_if(InputIterator first, InputIterator last,
                                   OutputIterator result, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate>
  ForwardIterator2 copy_if(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                           ForwardIterator1 first, ForwardIterator1 last,
                           ForwardIterator2 result, Predicate pred);

namespace ranges {
  template<class I, class O>
  using copy_if_result = copy_result<I, O>;

  template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires IndirectlyCopyable<I, O>
    tagged_pair<tag::in(I), tag::out(O)>
    constexpr copy_if_result<I, O>
      copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});
  template<InputRange Rng, WeaklyIncrementable O, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    requires IndirectlyCopyable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    constexpr copy_if_result<safe_iterator_t<Rng>, O>
      copy_if(Rng&& rng, O result, Pred pred, Proj proj = Proj{});
}

template<class BidirectionalIterator1, class BidirectionalIterator2>
  constexpr BidirectionalIterator2
    copy_backward(BidirectionalIterator1 first, BidirectionalIterator1 last,
                  BidirectionalIterator2 result);

namespace ranges {
  template<class I1, class I2>
  using copy_backward_result = copy_result<I, I2>;

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

// 30.6.2, move
template<class InputIterator, class OutputIterator>
  constexpr OutputIterator move(InputIterator first, InputIterator last,
                                OutputIterator result);
```

```
namespace ranges {
  template<class I1, class O>
  using move_result = copy_result<I, O>;

  template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
    requires IndirectlyMovable<I, O>
    tagged_pair<tag::in(I), tag::out(O)>
    constexpr move_result<I, O>
      move(I first, S last, O result);
  template<InputRange Rng, WeaklyIncrementable O>
    requires IndirectlyMovable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    constexpr move_result<safe_iterator_t<Rng>, O>
      move(Rng&& rng, O result);
}

template<class ExecutionPolicy, class ForwardIterator1,
         class ForwardIterator2>
  ForwardIterator2 move(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                        ForwardIterator1 first, ForwardIterator1 last,
                        ForwardIterator2 result);
template<class BidirectionalIterator1, class BidirectionalIterator2>
  constexpr BidirectionalIterator2
    move_backward(BidirectionalIterator1 first, BidirectionalIterator1 last,
                  BidirectionalIterator2 result);

namespace ranges {
  template<class I1, class I2>
  using move_backward_result = copy_result<I1, I2>;

  template<BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
    requires IndirectlyMovable<I1, I2>
    tagged_pair<tag::in(I1), tag::out(I2)>
    constexpr move_backward_result<I1, I2>
      move_backward(I1 first, S1 last, I2 result);
  template<BidirectionalRange Rng, BidirectionalIterator I>
    requires IndirectlyMovable<iterator_t<Rng>, I>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(I)>
    constexpr move_backward_result<safe_iterator_t<Rng>, I>
      move_backward(Rng&& rng, I result);
}

// 30.6.3, swap
template<class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
                               ForwardIterator2 first2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 swap_ranges(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                               ForwardIterator1 first1, ForwardIterator1 last1,
                               ForwardIterator2 first2);

namespace ranges {
  template<class I1, class I2>
  using swap_ranges_result = mismatch_result<I1, I2>;

  template<ForwardInputIterator I1, Sentinel<I1> S1, ForwardInputIterator I2, Sentinel<I2> S2>
    requires IndirectlySwappable<I1, I2>
    tagged_pair<tag::in1(I1), tag::in2(I2)>
    constexpr swap_ranges_result<I1, I2>
      swap_ranges(I1 first1, S1 last1, I2 first2, S2 last2);
  template<ForwardInputRange Rng1, ForwardInputRange Rng2>
    requires IndirectlySwappable<iterator_t<Rng1>, iterator_t<Rng2>>
    tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::in2(safe_iterator_t<Rng2>)>
    constexpr swap_ranges_result<safe_iterator_t<Rng1>, safe_iterator_t<Rng2>>
      swap_ranges(Rng1&& rng1, Rng2&& rng2);
```

```
}

template<class ForwardIterator1, class ForwardIterator2>
  void iter_swap(ForwardIterator1 a, ForwardIterator2 b);

// 30.6.4, transform
template<class InputIterator, class OutputIterator, class UnaryOperation>
  constexpr OutputIterator
    transform(InputIterator first, InputIterator last,
              OutputIterator result, UnaryOperation op);
template<class InputIterator1, class InputIterator2, class OutputIterator,
         class BinaryOperation>
  constexpr OutputIterator
    transform(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, OutputIterator result,
              BinaryOperation binary_op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class UnaryOperation>
  ForwardIterator2
    transform(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
              ForwardIterator1 first, ForwardIterator1 last,
              ForwardIterator2 result, UnaryOperation op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class BinaryOperation>
  ForwardIterator
    transform(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
              ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator result,
              BinaryOperation binary_op);

namespace ranges {
  template<class I, class O>
  using unary_transform_result = copy_result<I, O>;

  template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
      CopyConstructible F, class Proj = identity>
    requires Writable<O, indirect_result~~_of~~_t<F&~~(~~, projected<I, Proj>~~)~~>>
    ~~tagged_pair<tag::in(I), tag::out(O)>~~
    constexpr unary_transform_result<I, O>
      transform(I first, S last, O result, F op, Proj proj = Proj{});
  template<InputRange Rng, WeaklyIncrementable O, CopyConstructible F,
      class Proj = identity>
    requires Writable<O, indirect_result~~_of~~_t<F&~~(~~,
      projected<iterator_t<R>, Proj>~~)~~>>
    ~~tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>~~
    constexpr unary_transform_result<safe_iterator_t<Rng>, O>
      transform(Rng&& rng, O result, F op, Proj proj = Proj{});

  template<class I1, class I2, class O>
  struct binary_transform_result {
    I1 in1;
    I2 in2;
    O out;
  };

  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      WeaklyIncrementable O, CopyConstructible F, class Proj1 = identity,
      class Proj2 = identity>
    requires Writable<O, indirect_result~~_of~~_t<F&~~(~~, projected<I1, Proj1>,
      projected<I2, Proj2>~~)~~>>
    ~~tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>~~
    constexpr binary_transform_result<I1, I2, O>
      transform(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

```
  template<InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
      CopyConstructible F, class Proj1 = identity, class Proj2 = identity>
    requires Writable<O, indirect_result~~_of~~_t<F&~~(~~,
      projected<iterator_t<Rng1>, Proj1>, projected<iterator_t<Rng2>, Proj2>~~)~~>>
    ~~tagged_tuple<tag::in1(safe_iterator_t<Rng1>),~~
                 ~~tag::in2(safe_iterator_t<Rng2>),~~
                 ~~tag::out(O)>~~
    constexpr binary_transform_result<safe_iterator_t<Rng1>, safe_iterator_t<Rng2>, O>
      transform(Rng1&& rng1, Rng2&& rng2, O result,
                F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

// 30.6.5, replace
template<class ForwardIterator, class T>
  constexpr void replace(ForwardIterator first, ForwardIterator last,
                         const T& old_value, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  void replace(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
               ForwardIterator first, ForwardIterator last,
               const T& old_value, const T& new_value);
template<class ForwardIterator, class Predicate, class T>
  constexpr void replace_if(ForwardIterator first, ForwardIterator last,
                            Predicate pred, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator, class Predicate, class T>
  void replace_if(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                  ForwardIterator first, ForwardIterator last,
                  Predicate pred, const T& new_value);

namespace ranges {
  template<InputIterator I, Sentinel<I> S, class T1, class T2, class Proj = identity>
    requires Writable<I, const T2&> &&
      IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T1*>
    constexpr I
      replace(I first, S last, const T1& old_value, const T2& new_value, Proj proj = Proj{});
  template<InputRange Rng, class T1, class T2, class Proj = identity>
    requires Writable<iterator_t<Rng>, const T2&> &&
      IndirectRelation<ranges::equal_to<>, projected<iterator_t<Rng>, Proj>, const T1*>
    constexpr safe_iterator_t<Rng>
      replace(Rng&& rng, const T1& old_value, const T2& new_value, Proj proj = Proj{});
  template<InputIterator I, Sentinel<I> S, class T, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires Writable<I, const T&>
    constexpr I replace_if(I first, S last, Pred pred, const T& new_value, Proj proj = Proj{});
  template<InputRange Rng, class T, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    requires Writable<iterator_t<Rng>, const T&>
    constexpr safe_iterator_t<Rng>
      replace_if(Rng&& rng, Pred pred, const T& new_value, Proj proj = Proj{});
}

template<class InputIterator, class OutputIterator, class T>
  constexpr OutputIterator replace_copy(InputIterator first, InputIterator last,
                                        OutputIterator result,
                                        const T& old_value, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T>
  ForwardIterator2 replace_copy(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                                ForwardIterator1 first, ForwardIterator1 last,
                                ForwardIterator2 result,
                                const T& old_value, const T& new_value);
template<class InputIterator, class OutputIterator, class Predicate, class T>
  constexpr OutputIterator replace_copy_if(InputIterator first, InputIterator last,
                                           OutputIterator result,
                                           Predicate pred, const T& new_value);
```

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate, class T>
  ForwardIterator2 replace_copy_if(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                                   ForwardIterator1 first, ForwardIterator1 last,
                                   ForwardIterator2 result,
                                   Predicate pred, const T& new_value);

namespace ranges {
  template<class I, class O>
  using replace_copy_result = copy_result<I, O>;

  template<InputIterator I, Sentinel<I> S, class T1, class T2, OutputIterator<const T2&> O,
      class Proj = identity>
    requires IndirectlyCopyable<I, O> &&
      IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T1*>
    tagged_pair<tag::in(I), tag::out(O)>
    constexpr replace_copy_result<I, O>
      replace_copy(I first, S last, O result, const T1& old_value, const T2& new_value,
                   Proj proj = Proj{});
  template<InputRange Rng, class T1, class T2, OutputIterator<const T2&> O,
      class Proj = identity>
    requires IndirectlyCopyable<iterator_t<Rng>, O> &&
      IndirectRelation<ranges::equal_to<>, projected<iterator_t<Rng>, Proj>, const T1*>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    constexpr replace_copy_result<safe_iterator_t<Rng>, O>
      replace_copy(Rng&& rng, O result, const T1& old_value, const T2& new_value,
                   Proj proj = Proj{});

  template<class I, class O>
  using replace_copy_if_result = copy_result<I, O>;

  template<InputIterator I, Sentinel<I> S, class T, OutputIterator<const T&> O,
      class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires IndirectlyCopyable<I, O>
    tagged_pair<tag::in(I), tag::out(O)>
    constexpr replace_copy_if_result<I, O>
      replace_copy_if(I first, S last, O result, Pred pred, const T& new_value,
                      Proj proj = Proj{});
  template<InputRange Rng, class T, OutputIterator<const T&> O, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    requires IndirectlyCopyable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    constexpr replace_copy_if_result<safe_iterator_t<Rng>, O>
      replace_copy_if(Rng&& rng, O result, Pred pred, const T& new_value,
                      Proj proj = Proj{});
}

// 30.6.6, fill
template<class ForwardIterator, class T>
  constexpr void fill(ForwardIterator first, ForwardIterator last, const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  void fill(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
            ForwardIterator first, ForwardIterator last, const T& value);
template<class OutputIterator, class Size, class T>
  constexpr OutputIterator fill_n(OutputIterator first, Size n, const T& value);
template<class ExecutionPolicy, class ForwardIterator,
         class Size, class T>
  ForwardIterator fill_n(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                         ForwardIterator first, Size n, const T& value);

namespace ranges {
  template<class T, OutputIterator<const T&> O, Sentinel<O> S>
    constexpr O fill(O first, S last, const T& value);
  template<class T, OutputRange<const T&> Rng>
    constexpr safe_iterator_t<Rng> fill(Rng&& rng, const T& value);
```

```
  template<class T, OutputIterator<const T&> O>
    constexpr O fill_n(O first, iter_difference~~_type~~_t<O> n, const T& value);
}

// 30.6.7, generate
template<class ForwardIterator, class Generator>
  constexpr void generate(ForwardIterator first, ForwardIterator last,
                          Generator gen);
template<class ExecutionPolicy, class ForwardIterator, class Generator>
  void generate(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                ForwardIterator first, ForwardIterator last,
                Generator gen);
template<class OutputIterator, class Size, class Generator>
  constexpr OutputIterator generate_n(OutputIterator first, Size n, Generator gen);
template<class ExecutionPolicy, class ForwardIterator, class Size, class Generator>
  ForwardIterator generate_n(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                             ForwardIterator first, Size n, Generator gen);

namespace ranges {
  template<Iterator O, Sentinel<O> S, CopyConstructible F>
      requires Invocable<F&> && Writable<O, ~~result_of_t<F&()>~~invoke_result_t<F&>>
    constexpr O generate(O first, S last, F gen);
  template<class Rng, CopyConstructible F>
      requires Invocable<F&> && OutputRange<Rng, ~~result_of_t<F&()>~~invoke_result_t<F&>>
    constexpr safe_iterator_t<Rng> generate(Rng&& rng, F gen);
  template<Iterator O, CopyConstructible F>
      requires Invocable<F&> && Writable<O, ~~result_of_t<F&()>~~invoke_result_t<F&>>
    constexpr O generate_n(O first, iter_difference~~_type~~_t<O> n, F gen);
}

// 30.6.8, remove
template<class ForwardIterator, class T>
  constexpr ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                                   const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  ForwardIterator remove(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                         ForwardIterator first, ForwardIterator last,
                         const T& value);
template<class ForwardIterator, class Predicate>
  constexpr ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                                      Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator remove_if(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                            ForwardIterator first, ForwardIterator last,
                            Predicate pred);

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

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

template<class InputIterator, class OutputIterator, class T>
  constexpr OutputIterator
    remove_copy(InputIterator first, InputIterator last,
                OutputIterator result, const T& value);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class T>
  ForwardIterator2
    remove_copy(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result, const T& value);
template<class InputIterator, class OutputIterator, class Predicate>
  constexpr OutputIterator
    remove_copy_if(InputIterator first, InputIterator last,
                   OutputIterator result, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate>
  ForwardIterator2
    remove_copy_if(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                   ForwardIterator1 first, ForwardIterator1 last,
                   ForwardIterator2 result, Predicate pred);

namespace ranges {
  template<class I, class O>
  using remove_copy_result = copy_result<I, O>;

  template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class T,
      class Proj = identity>
    requires IndirectlyCopyable<I, O> &&
      IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T*>
    tagged_pair<tag::in(I), tag::out(O)>
    constexpr remove_copy_result<I, O>
      remove_copy(I first, S last, O result, const T& value, Proj proj = Proj{});
  template<InputRange Rng, WeaklyIncrementable O, class T, class Proj = identity>
    requires IndirectlyCopyable<iterator_t<Rng>, O> &&
      IndirectRelation<ranges::equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    constexpr remove_copy_result<safe_iterator_t<Rng>, O>
      remove_copy(Rng&& rng, O result, const T& value, Proj proj = Proj{});

  template<class I, class O>
  using remove_copy_if_result = copy_result<I, O>;

  template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
      class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires IndirectlyCopyable<I, O>
    tagged_pair<tag::in(I), tag::out(O)>
    constexpr remove_copy_if_result<I, O>
      remove_copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});
  template<InputRange Rng, WeaklyIncrementable O, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    requires IndirectlyCopyable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    constexpr remove_copy_if_result<safe_iterator_t<Rng>, O>
      remove_copy_if(Rng&& rng, O result, Pred pred, Proj proj = Proj{});
}

// 30.6.9, unique
template<class ForwardIterator>
  constexpr ForwardIterator unique(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class BinaryPredicate>
  constexpr ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                                   BinaryPredicate pred);
```

```
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator unique(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                         ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class BinaryPredicate>
  ForwardIterator unique(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                         ForwardIterator first, ForwardIterator last,
                         BinaryPredicate pred);

namespace ranges {
  template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectRelation<projected<I, Proj>> R = ranges::equal_to<>>
    requires Permutable<I>
    constexpr I unique(I first, S last, R comp = R{}, Proj proj = Proj{});
  template<ForwardRange Rng, class Proj = identity,
      IndirectRelation<projected<iterator_t<Rng>, Proj>> R = ranges::equal_to<>>
    requires Permutable<iterator_t<Rng>>
    constexpr safe_iterator_t<Rng>
      unique(Rng&& rng, R comp = R{}, Proj proj = Proj{});
}

template<class InputIterator, class OutputIterator>
  constexpr OutputIterator
    unique_copy(InputIterator first, InputIterator last,
                OutputIterator result);
template<class InputIterator, class OutputIterator, class BinaryPredicate>
  constexpr OutputIterator
    unique_copy(InputIterator first, InputIterator last,
                OutputIterator result, BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2
    unique_copy(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  ForwardIterator2
    unique_copy(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result, BinaryPredicate pred);

namespace ranges {
  template<class I, class O>
  using unique_copy_result = copy_result<I, O>;

  template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
      class Proj = identity, IndirectRelation<projected<I, Proj>> R = ranges::equal_to<>>
    requires IndirectlyCopyable<I, O> &&
      (ForwardIterator<I> ||
      (InputIterator<O> && Same<iter_value~~_type~~_t<I>, iter_value~~_type~~_t<O>>) ||
      IndirectlyCopyableStorable<I, O>)
    ~~tagged_pair<tag::in(I), tag::out(O)>~~
    constexpr unique_copy_result<I, O>
      unique_copy(I first, S last, O result, R comp = R{}, Proj proj = Proj{});
  template<InputRange Rng, WeaklyIncrementable O, class Proj = identity,
      IndirectRelation<projected<iterator_t<Rng>, Proj>> R = ranges::equal_to<>>
    requires IndirectlyCopyable<iterator_t<Rng>, O> &&
      (ForwardIterator<iterator_t<Rng>> ||
      (InputIterator<O> && Same<iter_value~~_type~~_t<iterator_t<Rng>>, iter_value~~_type~~_t<O>>) ||
      IndirectlyCopyableStorable<iterator_t<Rng>, O>)
    ~~tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>~~
    constexpr unique_copy_result<safe_iterator_t<Rng>, O>
      unique_copy(Rng&& rng, O result, R comp = R{}, Proj proj = Proj{});
}
```

```
// 30.6.10, reverse
template<class BidirectionalIterator>
  void reverse(BidirectionalIterator first, BidirectionalIterator last);
template<class ExecutionPolicy, class BidirectionalIterator>
  void reverse(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
               BidirectionalIterator first, BidirectionalIterator last);

namespace ranges {
  template<BidirectionalIterator I, Sentinel<I> S>
    requires Permutable<I>
    constexpr I reverse(I first, S last);
  template<BidirectionalRange Rng>
    requires Permutable<iterator_t<Rng>>
    constexpr safe_iterator_t<Rng> reverse(Rng&& rng);
}

template<class BidirectionalIterator, class OutputIterator>
  constexpr OutputIterator
    reverse_copy(BidirectionalIterator first, BidirectionalIterator last,
                 OutputIterator result);
template<class ExecutionPolicy, class BidirectionalIterator, class ForwardIterator>
  ForwardIterator
    reverse_copy(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                 BidirectionalIterator first, BidirectionalIterator last,
                 ForwardIterator result);

namespace ranges {
  template<class I, class O>
  using reverse_copy_result = copy_result<I, O>;

  template<BidirectionalIterator I, Sentinel<I> S, WeaklyIncrementable O>
    requires IndirectlyCopyable<I, O>
    tagged_pair<tag::in(I), tag::out(O)>
    constexpr reverse_copy_result<I, O>
      reverse_copy(I first, S last, O result);
  template<BidirectionalRange Rng, WeaklyIncrementable O>
    requires IndirectlyCopyable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    constexpr reverse_copy_result<safe_iterator_t<Rng>, O>
      reverse_copy(Rng&& rng, O result);
}

// 30.6.11, rotate
template<class ForwardIterator>
  ForwardIterator rotate(ForwardIterator first,
                         ForwardIterator middle,
                         ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator rotate(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                         ForwardIterator first,
                         ForwardIterator middle,
                         ForwardIterator last);

namespace ranges {
  template<ForwardIterator I, Sentinel<I> S>
    requires Permutable<I>
    tagged_pair<tag::begin(I), tag::end(I)>
    constexpr subrange<I>
      rotate(I first, I middle, S last);
  template<ForwardRange Rng>
    requires Permutable<iterator_t<Rng>>
    tagged_pair<tag::begin(safe_iterator_t<Rng>),
                tag::end(safe_iterator_t<Rng>)>
    constexpr safe_subrange_t<Rng>
      rotate(Rng&& rng, iterator_t<Rng> middle);
}
```

```
template<class ForwardIterator, class OutputIterator>
  constexpr OutputIterator
    rotate_copy(ForwardIterator first, ForwardIterator middle,
                ForwardIterator last, OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2
    rotate_copy(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                ForwardIterator1 first, ForwardIterator1 middle,
                ForwardIterator1 last, ForwardIterator2 result);

namespace ranges {
  template<class I, class O>
  using rotate_copy_result = copy_result<I, O>;

  template<ForwardIterator I, Sentinel<I> S, WeaklyIncrementable O>
    requires IndirectlyCopyable<I, O>
    ~~tagged_pair<tag::in(I), tag::out(O)>~~
    constexpr rotate_copy_result<I, O>
      rotate_copy(I first, I middle, S last, O result);
  template<ForwardRange Rng, WeaklyIncrementable O>
    requires IndirectlyCopyable<iterator_t<Rng>, O>
    ~~tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>~~
    constexpr rotate_copy_result<safe_iterator_t<Rng>, O>
      rotate_copy(Rng&& rng, iterator_t<Rng> middle, O result);
}

// 30.6.12, sample
[...]

// 30.6.13, shuffle
template<class RandomAccessIterator, class UniformRandomBitGenerator>
  void shuffle(RandomAccessIterator first,
               RandomAccessIterator last,
               UniformRandomBitGenerator&& g);

namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Gen>
    requires Permutable<I> &&
      UniformRandom~~Number~~BitGenerator<remove_reference_t<Gen>> &&
      ConvertibleTo<~~result_of_t<Gen&()>~~invoke_result_t<Gen&>, iter_difference~~_type~~_t<I>>
    I shuffle(I first, S last, Gen&& g);
  template<RandomAccessRange Rng, class Gen>
    requires Permutable<I> &&
      UniformRandom~~Number~~BitGenerator<remove_reference_t<Gen>> &&
      ConvertibleTo<~~result_of_t<Gen&()>~~invoke_result_t<Gen&>, iter_difference~~_type~~_t<I>>
    safe_iterator_t<Rng>
      shuffle(Rng&& rng, Gen&& g);
}

// 30.7.4, partitions
template<class InputIterator, class Predicate>
  constexpr bool is_partitioned(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  bool is_partitioned(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                      ForwardIterator first, ForwardIterator last, Predicate pred);

namespace ranges {
  template<InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr bool is_partitioned(I first, S last, Pred pred, Proj proj = Proj{});
  template<InputRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    constexpr bool is_partitioned(Rng&& rng, Pred pred, Proj proj = Proj{});
}
```

```
template<class ForwardIterator, class Predicate>
  ForwardIterator partition(ForwardIterator first,
                            ForwardIterator last,
                            Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator partition(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                            ForwardIterator first,
                            ForwardIterator last,
                            Predicate pred);

namespace ranges {
  template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires Permutable<I>
    constexpr I
      partition(I first, S last, Pred pred, Proj proj = Proj{});
  template<ForwardRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    requires Permutable<iterator_t<Rng>>
    constexpr safe_iterator_t<Rng>
      partition(Rng&& rng, Pred pred, Proj proj = Proj{});
}

template<class BidirectionalIterator, class Predicate>
  BidirectionalIterator stable_partition(BidirectionalIterator first,
                                         BidirectionalIterator last,
                                         Predicate pred);
template<class ExecutionPolicy, class BidirectionalIterator, class Predicate>
  BidirectionalIterator stable_partition(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                                         BidirectionalIterator first,
                                         BidirectionalIterator last,
                                         Predicate pred);

namespace ranges {
  template<BidirectionalIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires Permutable<I>
    I stable_partition(I first, S last, Pred pred, Proj proj = Proj{});
  template<BidirectionalRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    requires Permutable<iterator_t<Rng>>
    safe_iterator_t<Rng> stable_partition(Rng&& rng, Pred pred, Proj proj = Proj{});
}

template<class InputIterator, class OutputIterator1,
         class OutputIterator2, class Predicate>
  constexpr pair<OutputIterator1, OutputIterator2>
    partition_copy(InputIterator first, InputIterator last,
                   OutputIterator1 out_true, OutputIterator2 out_false,
                   Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class ForwardIterator1,
         class ForwardIterator2, class Predicate>
  pair<ForwardIterator1, ForwardIterator2>
    partition_copy(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                   ForwardIterator first, ForwardIterator last,
                   ForwardIterator1 out_true, ForwardIterator2 out_false,
                   Predicate pred);

namespace ranges {
  template<class I, class O1, class O2>
  struct partition_copy_result {
    I in;
    O1 out1;
    O2 out2;
  };
```

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

template<class ForwardIterator, class Predicate>
  constexpr ForwardIterator
    partition_point(ForwardIterator first, ForwardIterator last,
                    Predicate pred);

namespace ranges {
  template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr I partition_point(I first, S last, Pred pred, Proj proj = Proj{});
  template<ForwardRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    constexpr safe_iterator_t<Rng>
      partition_point(Rng&& rng, Pred pred, Proj proj = Proj{});
}

// 30.7, sorting and related operations
// 30.7.1, sorting
template<class RandomAccessIterator>
  void sort(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void sort(RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator>
  void sort(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
            RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  void sort(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
            RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);

namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    constexpr I
      sort(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    constexpr safe_iterator_t<Rng>
      sort(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class RandomAccessIterator>
  void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);
```

```
template<class ExecutionPolicy, class RandomAccessIterator>
  void stable_sort(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                   RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  void stable_sort(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                   RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);

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

template<class RandomAccessIterator>
  void partial_sort(RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void partial_sort(RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last, Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator>
  void partial_sort(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                    RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  void partial_sort(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                    RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last, Compare comp);

namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    constexpr I
      partial_sort(I first, I middle, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    constexpr safe_iterator_t<Rng>
      partial_sort(Rng&& rng, iterator_t<Rng> middle, Comp comp = Comp{},
                   Proj proj = Proj{});
}

template<class InputIterator, class RandomAccessIterator>
  RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last);
template<class InputIterator, class RandomAccessIterator, class Compare>
  RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last,
                      Compare comp);
template<class ExecutionPolicy, class ForwardIterator, class RandomAccessIterator>
  RandomAccessIterator
    partial_sort_copy(ExecutionPolicy&& exec,   // see [algorithms.parallel.overloads]
```

```
                     ForwardIterator first, ForwardIterator last,
                     RandomAccessIterator result_first,
                     RandomAccessIterator result_last);
template<class ExecutionPolicy, class ForwardIterator, class RandomAccessIterator,
         class Compare>
  RandomAccessIterator
    partial_sort_copy(ExecutionPolicy&& exec,   // see [algorithms.parallel.overloads]
                      ForwardIterator first, ForwardIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last,
                      Compare comp);

namespace ranges {
  template<InputIterator I1, Sentinel<I1> S1, RandomAccessIterator I2, Sentinel<I2> S2,
      class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyCopyable<I1, I2> && Sortable<I2, Comp, Proj2> &&
        IndirectStrictWeakOrder<Comp, projected<I1, Proj1>, projected<I2, Proj2>>
    constexpr I2
      partial_sort_copy(I1 first, S1 last, I2 result_first, S2 result_last,
                        Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<InputRange Rng1, RandomAccessRange Rng2, class Comp = ranges::less<>,
      class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyCopyable<iterator_t<Rng1>, iterator_t<Rng2>> &&
        Sortable<iterator_t<Rng2>, Comp, Proj2> &&
        IndirectStrictWeakOrder<Comp, projected<iterator_t<Rng1>, Proj1>,
          projected<iterator_t<Rng2>, Proj2>>
    constexpr safe_iterator_t<Rng2>
      partial_sort_copy(Rng1&& rng, Rng2&& result_rng, Comp comp = Comp{},
                        Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

template<class ForwardIterator>
  constexpr bool is_sorted(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  constexpr bool is_sorted(ForwardIterator first, ForwardIterator last,
                           Compare comp);
template<class ExecutionPolicy, class ForwardIterator>
  bool is_sorted(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                 ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  bool is_sorted(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                 ForwardIterator first, ForwardIterator last,
                 Compare comp);

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

template<class ForwardIterator>
  constexpr ForwardIterator
    is_sorted_until(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  constexpr ForwardIterator
    is_sorted_until(ForwardIterator first, ForwardIterator last,
                    Compare comp);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator
    is_sorted_until(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                    ForwardIterator first, ForwardIterator last);
```

```
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  ForwardIterator
    is_sorted_until(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                    ForwardIterator first, ForwardIterator last,
                    Compare comp);

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

// 30.7.2, Nth element
template<class RandomAccessIterator>
  void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last, Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator>
  void nth_element(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                   RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  void nth_element(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                   RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last, Compare comp);

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

// 30.7.3, binary search
template<class ForwardIterator, class T>
  constexpr ForwardIterator
    lower_bound(ForwardIterator first, ForwardIterator last,
                const T& value);
template<class ForwardIterator, class T, class Compare>
  constexpr ForwardIterator
    lower_bound(ForwardIterator first, ForwardIterator last,
                const T& value, Compare comp);

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

```
template<class ForwardIterator, class T>
  constexpr ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last,
                const T& value);
template<class ForwardIterator, class T, class Compare>
  constexpr ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last,
                const T& value, Compare comp);

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

template<class ForwardIterator, class T>
  constexpr pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first, ForwardIterator last,
                const T& value);
template<class ForwardIterator, class T, class Compare>
  constexpr pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first, ForwardIterator last,
                const T& value, Compare comp);

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

template<class ForwardIterator, class T>
  constexpr bool
    binary_search(ForwardIterator first, ForwardIterator last,
                  const T& value);
template<class ForwardIterator, class T, class Compare>
  constexpr bool
    binary_search(ForwardIterator first, ForwardIterator last,
                  const T& value, Compare comp);

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

// 30.7.5, merge
template<class InputIterator1, class InputIterator2, class OutputIterator>
```

```
  constexpr OutputIterator
    merge(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator,
         class Compare>
  constexpr OutputIterator
    merge(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  ForwardIterator
    merge(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
          ForwardIterator1 first1, ForwardIterator1 last1,
          ForwardIterator2 first2, ForwardIterator2 last2,
          ForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  ForwardIterator
    merge(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
          ForwardIterator1 first1, ForwardIterator1 last1,
          ForwardIterator2 first2, ForwardIterator2 last2,
          ForwardIterator result, Compare comp);

namespace ranges {
  template<class I1, class I2, class O>
  using merge_result = binary_transform_result<I1, I2, O>;

  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      WeaklyIncrementable O, class Comp = ranges::less<>, class Proj1 = identity,
      class Proj2 = identity>
    requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
    tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    constexpr merge_result<I1, I2, O>
      merge(I1 first1, S1 last1, I2 first2, S2 last2, O result,
            Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<InputRange Rng1, InputRange Rng2, WeaklyIncrementable O, class Comp = ranges::less<>,
      class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
    tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
                 tag::in2(safe_iterator_t<Rng2>),
                 tag::out(O)>
    constexpr merge_result<safe_iterator_t<Rng1>, safe_iterator_t<Rng2>, O>
      merge(Rng1&& rng1, Rng2&& rng2, O result,
            Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

template<class BidirectionalIterator>
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last);
template<class BidirectionalIterator, class Compare>
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last, Compare comp);
template<class ExecutionPolicy, class BidirectionalIterator>
  void inplace_merge(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                     BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last);
template<class ExecutionPolicy, class BidirectionalIterator, class Compare>
  void inplace_merge(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                     BidirectionalIterator first,
```

```
                         BidirectionalIterator middle,
                         BidirectionalIterator last, Compare comp);

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

// 30.7.6, set operations
template<class InputIterator1, class InputIterator2>
  constexpr bool includes(InputIterator1 first1, InputIterator1 last1,
                          InputIterator2 first2, InputIterator2 last2);
template<class InputIterator1, class InputIterator2, class Compare>
  constexpr bool includes(InputIterator1 first1, InputIterator1 last1,
                          InputIterator2 first2, InputIterator2 last2,
                          Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  bool includes(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Compare>
  bool includes(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2, ForwardIterator2 last2,
                Compare comp);

namespace ranges {
  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      class Proj1 = identity, class Proj2 = identity,
      IndirectStrictWeakOrder<projected<I1, Proj1>, projected<I2, Proj2>> Comp = ranges::less<>>
    constexpr bool includes(I1 first1, S1 last1, I2 first2, S2 last2, Comp comp = Comp{},
                            Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<InputRange Rng1, InputRange Rng2, class Proj1 = identity,
      class Proj2 = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng1>, Proj1>,
        projected<iterator_t<Rng2>, Proj2>> Comp = ranges::less<>>
    constexpr bool includes(Rng1&& rng1, Rng2&& rng2, Comp comp = Comp{},
                            Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

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

```
              ForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  ForwardIterator
    set_union(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
              ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator2 last2,
              ForwardIterator result, Compare comp);

namespace ranges {
  template<class I1, class I2, class O>
  using set_union_result = binary_transform_result<I1, I2, O>;

  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      WeaklyIncrementable O, class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
    tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    constexpr set_union_result<I1, I2, O>
      set_union(I1 first1, S1 last1, I2 first2, S2 last2, O result, Comp comp = Comp{},
                Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
      class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
    tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
                 tag::in2(safe_iterator_t<Rng2>),
                 tag::out(O)>
    constexpr set_union_result<safe_iterator_t<Rng1>, safe_iterator_t<Rng2>, O>
      set_union(Rng1&& rng1, Rng2&& rng2, O result, Comp comp = Comp{},
                Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

template<class InputIterator1, class InputIterator2, class OutputIterator>
  constexpr OutputIterator
    set_intersection(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
  constexpr OutputIterator
    set_intersection(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  ForwardIterator
    set_intersection(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                     ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2, ForwardIterator2 last2,
                     ForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  ForwardIterator
    set_intersection(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                     ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2, ForwardIterator2 last2,
                     ForwardIterator result, Compare comp);

namespace ranges {
  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      WeaklyIncrementable O, class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
    constexpr O set_intersection(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                                 Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

```
  template<InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
      class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
    constexpr O set_intersection(Rng1&& rng1, Rng2&& rng2, O result,
                                 Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

template<class InputIterator1, class InputIterator2, class OutputIterator>
  constexpr OutputIterator
    set_difference(InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, InputIterator2 last2,
                   OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
  constexpr OutputIterator
    set_difference(InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, InputIterator2 last2,
                   OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  ForwardIterator
    set_difference(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                   ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2, ForwardIterator2 last2,
                   ForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  ForwardIterator
    set_difference(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                   ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2, ForwardIterator2 last2,
                   ForwardIterator result, Compare comp);

namespace ranges {
  template<class I, class O>
  using set_difference_result = copy_result<I, O>;

  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      WeaklyIncrementable O, class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
    tagged_pair<tag::in1(I1), tag::out(O)>
    constexpr set_difference_result<I1, O>
      set_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                     Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
      class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
    tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::out(O)>
    constexpr set_difference_result<safe_iterator_t<Rng1>, O>
      set_difference(Rng1&& rng1, Rng2&& rng2, O result,
                     Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

template<class InputIterator1, class InputIterator2, class OutputIterator>
  constexpr OutputIterator
    set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
  constexpr OutputIterator
    set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result, Compare comp);
```

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  ForwardIterator
    set_symmetric_difference(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                             ForwardIterator1 first1, ForwardIterator1 last1,
                             ForwardIterator2 first2, ForwardIterator2 last2,
                             ForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  ForwardIterator
    set_symmetric_difference(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                             ForwardIterator1 first1, ForwardIterator1 last1,
                             ForwardIterator2 first2, ForwardIterator2 last2,
                             ForwardIterator result, Compare comp);

namespace ranges {
  template<class I1, class I2, class O>
  using set_symmetric_difference_result = binary_transform_result<I1, I2, O>;

  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      WeaklyIncrementable O, class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
    tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    constexpr set_symmetric_difference_result<I1, I2, O>
      set_symmetric_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                               Comp comp = Comp{}, Proj1 proj1 = Proj1{},
                               Proj2 proj2 = Proj2{});
  template<InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
      class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
    tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
                 tag::in2(safe_iterator_t<Rng2>),
                 tag::out(O)>
    constexpr set_symmetric_difference_result<safe_iterator_t<Rng1>, safe_iterator_t<Rng2>, O>
      set_symmetric_difference(Rng1&& rng1, Rng2&& rng2, O result, Comp comp = Comp{},
                               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

// 30.7.7, heap operations
template<class RandomAccessIterator>
  void push_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void push_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);

namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    constexpr I
      push_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    constexpr safe_iterator_t<Rng>
      push_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class RandomAccessIterator>
  void pop_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
                Compare comp);
```

```
namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    constexpr I
      pop_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    constexpr safe_iterator_t<Rng>
      pop_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class RandomAccessIterator>
  void make_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void make_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);

namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    constexpr I
      make_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    constexpr safe_iterator_t<Rng>
      make_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class RandomAccessIterator>
  void sort_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);

namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    constexpr I
      sort_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    constexpr safe_iterator_t<Rng>
      sort_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class RandomAccessIterator>
  constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last,
                         Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator>
  bool is_heap(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
               RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  bool is_heap(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
               RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);

namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
      IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less<>>
```

```
    constexpr bool is_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<RandomAccessRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    constexpr bool is_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class RandomAccessIterator>
  constexpr RandomAccessIterator
    is_heap_until(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  constexpr RandomAccessIterator
    is_heap_until(RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator>
  RandomAccessIterator
    is_heap_until(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                  RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  RandomAccessIterator
    is_heap_until(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                  RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp);

namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
      IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less<>>
    constexpr I is_heap_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<RandomAccessRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    constexpr safe_iterator_t<Rng>
      is_heap_until(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

// 30.7.8, minimum and maximum
template<class T> constexpr const T& min(const T& a, const T& b);
template<class T, class Compare>
  constexpr const T& min(const T& a, const T& b, Compare comp);
template<class T>
  constexpr T min(initializer_list<T> t);
template<class T, class Compare>
  constexpr T min(initializer_list<T> t, Compare comp);

namespace ranges {
  template<class T, class Proj = identity,
      IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr const T& min(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
  template<Copyable T, class Proj = identity,
      IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr T min(initializer_list<T> t, Comp comp = Comp{}, Proj proj = Proj{});
  template<InputRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    requires Copyable<iter_value_type_t<iterator_t<Rng>>>
    constexpr iter_value_type_t<iterator_t<Rng>>
      min(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class T> constexpr const T& max(const T& a, const T& b);
template<class T, class Compare>
  constexpr const T& max(const T& a, const T& b, Compare comp);
template<class T>
  constexpr T max(initializer_list<T> t);
template<class T, class Compare>
  constexpr T max(initializer_list<T> t, Compare comp);
```

```
namespace ranges {
  template<class T, class Proj = identity,
      IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr const T& max(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
  template<Copyable T, class Proj = identity,
      IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr T max(initializer_list<T> t, Comp comp = Comp{}, Proj proj = Proj{});
  template<InputRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    requires Copyable<iter_value_type_t<iterator_t<Rng>>>
    constexpr iter_value_type_t<iterator_t<Rng>>
      max(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class T> constexpr pair<const T&, const T&> minmax(const T& a, const T& b);
template<class T, class Compare>
  constexpr pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
template<class T>
  constexpr pair<T, T> minmax(initializer_list<T> t);
template<class T, class Compare>
  constexpr pair<T, T> minmax(initializer_list<T> t, Compare comp);

namespace ranges {
  template<class T>
  struct minmax_result {
    T min;
    T max;
  };

  template<class T, class Proj = identity,
      IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr tagged_pair<tag::min(const T&), tag::max(const T&)>
    constexpr minmax_result<const T&>
      minmax(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
  template<Copyable T, class Proj = identity,
      IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr tagged_pair<tag::min(T), tag::max(T)>
    constexpr minmax_result<T>
      minmax(initializer_list<T> t, Comp comp = Comp{}, Proj proj = Proj{});
  template<InputRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    requires Copyable<iter_value_type_t<iterator_t<Rng>>>
    tagged_pair<tag::min(value_type_t<iterator_t<Rng>>),
                tag::max(value_type_t<iterator_t<Rng>>)>
    constexpr minmax_result<iter_value_t<iterator_t<Rng>>>
      minmax(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class ForwardIterator>
  constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
                                        Compare comp);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator min_element(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                              ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  ForwardIterator min_element(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                              ForwardIterator first, ForwardIterator last,
                              Compare comp);

namespace ranges {
  template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less<>>
```

```
    constexpr I min_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<ForwardRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    constexpr safe_iterator_t<Rng>
      min_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class ForwardIterator>
  constexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  constexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
                                        Compare comp);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator max_element(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                              ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  ForwardIterator max_element(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                              ForwardIterator first, ForwardIterator last,
                              Compare comp);

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

template<class ForwardIterator>
  constexpr pair<ForwardIterator, ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  constexpr pair<ForwardIterator, ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last, Compare comp);
template<class ExecutionPolicy, class ForwardIterator>
  pair<ForwardIterator, ForwardIterator>
    minmax_element(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                   ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  pair<ForwardIterator, ForwardIterator>
    minmax_element(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                   ForwardIterator first, ForwardIterator last, Compare comp);

namespace ranges {
  template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less<>>
    tagged_pair<tag::min(I), tag::max(I)>
    constexpr minmax_result<I>
      minmax_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<ForwardRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    tagged_pair<tag::min(safe_iterator_t<Rng>),
                tag::max(safe_iterator_t<Rng>)>
    constexpr minmax_result<safe_iterator_t<Rng>>
      minmax_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

// 30.7.9, bounded value
[...]

// 30.7.10, lexicographical comparison
template<class InputIterator1, class InputIterator2>
  constexpr bool
    lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
```

```
                            InputIterator2 first2, InputIterator2 last2);
template<class InputIterator1, class InputIterator2, class Compare>
  constexpr bool
    lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2,
                            Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  bool
    lexicographical_compare(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                            ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Compare>
  bool
    lexicographical_compare(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                            ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2,
                            Compare comp);

namespace ranges {
  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      class Proj1 = identity, class Proj2 = identity,
      IndirectStrictWeakOrder<projected<I1, Proj1>, projected<I2, Proj2>> Comp = ranges::less<>>
    constexpr bool
      lexicographical_compare(I1 first1, S1 last1, I2 first2, S2 last2,
                              Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<InputRange Rng1, InputRange Rng2, class Proj1 = identity,
      class Proj2 = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng1>, Proj1>,
        projected<iterator_t<Rng2>, Proj2>> Comp = ranges::less<>>
    constexpr bool
      lexicographical_compare(Rng1&& rng1, Rng2&& rng2, Comp comp = Comp{},
                              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

// 30.7.11, three-way comparison algorithms
[...]

// 30.7.12, permutations
template<class BidirectionalIterator>
  bool next_permutation(BidirectionalIterator first,
                        BidirectionalIterator last);
template<class BidirectionalIterator, class Compare>
  bool next_permutation(BidirectionalIterator first,
                        BidirectionalIterator last, Compare comp);

namespace ranges {
  template<BidirectionalIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    constexpr bool
      next_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<BidirectionalRange Rng, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    constexpr bool
      next_permutation(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class BidirectionalIterator>
  bool prev_permutation(BidirectionalIterator first,
                        BidirectionalIterator last);
template<class BidirectionalIterator, class Compare>
  bool prev_permutation(BidirectionalIterator first,
                        BidirectionalIterator last, Compare comp);
```

```
namespace ranges {
  template<BidirectionalIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    constexpr bool
      prev_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<BidirectionalRange Rng, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    constexpr bool
      prev_permutation(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
}
```

30.3 Algorithms requirements [algorithms.requirements]

1 All of the algorithms are separated from the particular implementations of data structures and are parameterized by iterator types. Because of this, they can work with program-defined data structures, as long as these data structures have iterator types satisfying the assumptions on the algorithms.

2 The function templates defined in the `std::ranges` namespace in this subclause are not found by argument-dependent name lookup ([basic.lookup.argdep]). When found by unqualified ([basic.lookup.unqual]) name lookup for the *postfix-expression* in a function call ([expr.call]), they inhibit argument-dependent name lookup.

[*Example:*

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

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

3 For purposes of determining the existence of data races, [...]

4 Throughout this Clause, where the template parameters are not constrained, the names of template parameters are used to express type requirements. [...]

5 If an algorithm's *Effects:* element specifies that a value pointed to by any iterator passed as an argument is modified, then that algorithm has an additional type requirement: The type of that argument shall satisfy the requirements of a mutable iterator (28.3). [*Note:* This requirement does not affect arguments that are named `OutputIterator`, `OutputIterator1`, or `OutputIterator2`, because output iterators must always be mutable, nor does it affect arguments that are constrained, for which mutability requirements are expressed explicitly. — *end note*]

6 Both in-place and copying versions are provided for certain algorithms. [...]

7 When not otherwise constrained, the `Predicate` parameter is used whenever an algorithm expects a function object[function.objects] that, when applied to the result of dereferencing the corresponding iterator, [...]

8 When not otherwise constrained, the `BinaryPredicate` parameter is used whenever an algorithm expects a function object that [...]

[...]

11 In the description of the algorithms operators `+` and `-` are used for some of the iterator categories [...]

12 In the description of algorithm return values, sentinel values are sometimes returned where an iterator is expected. In these cases, the semantics are as if the sentinel is converted into an iterator as follows:

```
I tmp = first;
while(tmp != last)
  ++tmp;
return tmp;
```

13 Overloads of algorithms that take `Range` arguments (29.6.2) behave as if they are implemented by calling `ranges::begin` and `ranges::end` on the `Range`(s) and dispatching to the overload that takes separate iterator and sentinel arguments.

[Editor's note: Reviewers: note that the following now applies to *all* algorithms, not only those in namespace `ranges`:]

14 The number and order of template parameters for algorithm declarations is unspecified, except where explicitly stated otherwise.

30.5 Non-modifying sequence operations [alg.nonmodifying]

30.5.1 All of [alg.all_of]

```
template<class InputIterator, class Predicate>
  constexpr bool all_of(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  bool all_of(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
              Predicate pred);

namespace ranges {
  template<InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr bool all_of(I first, S last, Pred pred, Proj proj = Proj{});
  template<InputRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    constexpr bool all_of(Rng&& rng, Pred pred, Proj proj = Proj{});
}
```

1 Let E be `pred(*i)` and `invoke(pred, invoke(proj, *i))` for the overloads in namespace `std` and `std::ranges`, respectively.

2 ~~*Returns:* true if `[first, last)` is empty or if `pred(*i)` is `true` for every iterator `i` in the range `[first, last)`, and `false` otherwise.~~

Returns: false if E is `false` for some iterator `i` in the range `[first, last)`, and `true` otherwise.

3 *Complexity:* At most `last - first` applications of the predicate and the projection.

30.5.2 Any of [alg.any_of]

```
template<class InputIterator, class Predicate>
  constexpr bool any_of(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  bool any_of(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
              Predicate pred);

namespace ranges {
  template<InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr bool any_of(I first, S last, Pred pred, Proj proj = Proj{});
  template<InputRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    constexpr bool any_of(Rng&& rng, Pred pred, Proj proj = Proj{});
}
```

1 Let E be `pred(*i)` and `invoke(pred, invoke(proj, *i))` for the overloads in namespace `std` and `std::ranges`, respectively.

2 ~~*Returns:* false if `[first, last)` is empty or if there is no iterator `i` in the range `[first, last)` such that `pred(*i)` is `true`, and `true` otherwise.~~

Returns: true if E is `true` for some iterator `i` in the range `[first, last)`, and `false` otherwise.

3 *Complexity:* At most `last - first` applications of the predicate and the projection.

30.5.3 None of [alg.none_of]

```
template<class InputIterator, class Predicate>
  constexpr bool none_of(InputIterator first, InputIterator last, Predicate pred);
```

```
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  bool none_of(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
               Predicate pred);

namespace ranges {
  template<InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr bool none_of(I first, S last, Pred pred, Proj proj = Proj{});
  template<InputRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    constexpr bool none_of(Rng&& rng, Pred pred, Proj proj = Proj{});
}
```

1 Let P be `pred(*i)` and `invoke(pred, invoke(proj, *i))` for the overloads in namespace `std` and `std::ranges`, respectively.

2 ~~*Returns:* `true` if `[first, last)` is empty or if `pred(*i)` is `false` for every iterator `i` in the range `[first, last)`, and `false` otherwise.~~

Returns: `false` if P is `true` for some iterator `i` in the range `[first, last)`, and `true` otherwise.

3 *Complexity:* At most `last - first` applications of the predicate and the projection.

30.5.4 For each [alg.foreach]

[...]

10 [*Note:* Does not return a copy of its `Function` parameter, since parallelization may not permit efficient state accumulation. — *end note*]

```
namespace ranges {
  template<InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryInvocable<projected<I, Proj>> Fun>
    tagged_pair<tag::in(I), tag::fun(Fun)>
    constexpr for_each_result<I, Fun>
      for_each(I first, S last, Fun f, Proj proj = Proj{});
  template<InputRange Rng, class Proj = identity,
      IndirectUnaryInvocable<projected<iterator_t<Rng>, Proj>> Fun>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::fun(Fun)>
    constexpr for_each_result<safe_iterator_t<Rng>, Fun>
      for_each(Rng&& rng, Fun f, Proj proj = Proj{});
}
```

11 *Effects:* Calls `invoke(f, invoke(proj, *i))` for every iterator `i` in the range `[first, last)`, starting from `first` and proceeding to `last - 1`. [*Note:* If the result of `invoke(proj, *i)` is a mutable reference, `f` may apply nonconstant functions. — *end note*]

12 *Returns:* `{last, std::move(f)}`.

13 *Complexity:* Applies `f` and `proj` exactly `last - first` times.

14 *Remarks:* If `f` returns a result, the result is ignored.

[...]

30.5.5 Find [alg.find]

```
template<class InputIterator, class T>
  constexpr InputIterator find(InputIterator first, InputIterator last,
                               const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  ForwardIterator find(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
                       const T& value);

template<class InputIterator, class Predicate>
  constexpr InputIterator find_if(InputIterator first, InputIterator last,
                                  Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator find_if(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
                          Predicate pred);
```

```
template<class InputIterator, class Predicate>
  constexpr InputIterator find_if_not(InputIterator first, InputIterator last,
                                      Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator find_if_not(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
                              Predicate pred);

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

1 Let E be:

(1.1) — `*i == value` for `find`,

(1.2) — `pred(*i) != false` for `find_if`,

(1.3) — `pred(*i) == false` for `find_if_not`,

(1.4) — `invoke(proj, *i) == value` for `ranges::find`,

(1.5) — `invoke(pred, invoke(proj, *i)) != false` for `ranges::find_if`,

(1.6) — `invoke(pred, invoke(proj, *i)) == false` for `ranges::find_if_not`.

2 *Returns:* The first iterator `i` in the range `[first, last)` for which E is `true`. ~~the following corresponding conditions hold: `*i == value`, `pred(*i) != false`, `pred(*i) == false`.~~ Returns `last` if no such iterator is found.

3 *Complexity:* At most `last - first` applications of the corresponding predicate and projection.

30.5.6 Find end [alg.find.end]

```
template<class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    find_end(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  constexpr ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);
```

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  ForwardIterator1
    find_end(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);

namespace ranges {
  template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2,
      class Pred = ranges::equal_to<>, class Proj1 = identity, class Proj2 = identity>
      IndirectRelation<I2, projected<I1, Proj>> Pred = equal_to<>>
    requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
    constexpr I1 subrange<I1>
      find_end(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<ForwardRange Rng1, ForwardRange Rng2, class Proj = identity,
      IndirectRelation<iterator_t<Rng2>,
        projected<iterator_t<Rng>, Proj>> Pred = equal_to<>>
      class Pred = ranges::equal_to<>, class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
    constexpr safe_iterator_t<Rng1> safe_subrange_t<Rng1>
      find_end(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

[Editor's note: This wording incorporates the PR for stl2#180 and stl2#526).]

1 Let:

(1.1) — `pred` be `equal_to<>{}` for the overloads with no parameter `pred`.

(1.2) — P be:

(1.2.1) — `pred(*(i + n), *(first2 + n))` for the overloads in namespace `std`,

(1.2.2) — `invoke(pred, invoke(proj1, *(i + n)), invoke(proj2, *(first2 + n)))` for the overloads in namespace `ranges`

(1.3) — `i` be the last iterator in the range `[first1, last1 - (last2 - first2))` such that for every non-negative integer `n < (last2 - first2)`, P is `true`; or `last1` if no such iterator exists.

2 *Effects:* Finds a subsequence of equal values in a sequence.

3 ~~*Returns:* The last iterator `i` in the range `[first1, last1 - (last2 - first2))` such that for every non-negative integer `n < (last2 - first2)`, the following corresponding conditions hold: `*(i + n) == *(first2 + n), pred(*(i + n), *(first2 + n)) != false`. Returns `last1` if `[first2, last2)` is empty or if no such iterator is found.~~

4 *Returns:*

(4.1) — `i` for the overloads in namespace `std`, and

(4.2) — `{i, i + (i == last1 ? 0 : last2 - first2)}` for the overloads in namespace `ranges`.

5 *Complexity:* At most `(last2 - first2) * (last1 - first1 - (last2 - first2) + 1)` applications of the corresponding predicate and projections.

30.5.7 Find first [alg.find.first.of]

```
template<class InputIterator, class ForwardIterator>
  constexpr InputIterator
    find_first_of(InputIterator first1, InputIterator last1,
                  ForwardIterator first2, ForwardIterator last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    find_first_of(ExecutionPolicy&& exec,
                  ForwardIterator1 first1, ForwardIterator1 last1,
                  ForwardIterator2 first2, ForwardIterator2 last2);
```

```
template<class InputIterator, class ForwardIterator,
         class BinaryPredicate>
  constexpr InputIterator
    find_first_of(InputIterator first1, InputIterator last1,
                  ForwardIterator first2, ForwardIterator last2,
                  BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  ForwardIterator1
    find_first_of(ExecutionPolicy&& exec,
                  ForwardIterator1 first1, ForwardIterator1 last1,
                  ForwardIterator2 first2, ForwardIterator2 last2,
                  BinaryPredicate pred);

namespace ranges {
  template<InputIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2,
      class Proj1 = identity, class Proj2 = identity,
      IndirectRelation<projected<I1, Proj1>, projected<I2, Proj2>> Pred = ranges::equal_to<>>
    constexpr I1 find_first_of(I1 first1, S1 last1, I2 first2, S2 last2,
                               Pred pred = Pred{},
                               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<InputRange Rng1, ForwardRange Rng2, class Proj1 = identity,
      class Proj2 = identity,
      IndirectRelation<projected<iterator_t<Rng1>, Proj1>,
        projected<iterator_t<Rng2>, Proj2>> Pred = ranges::equal_to<>>
    constexpr safe_iterator_t<Rng1>
      find_first_of(Rng1&& rng1, Rng2&& rng2,
                    Pred pred = Pred{},
                    Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

1 Let E be:

(1.1) — `*i == *j` for the overloads with no parameter `pred`,

(1.2) — `pred(*i, *j) != false` for the overloads with a parameter `pred` and no parameter `proj1`,

(1.3) — `invoke(pred, invoke(proj1, *i), invoke(proj2, *j)) != false` for the overloads with both parameters `pred` and `proj1`.

2 *Effects:* Finds an element that matches one of a set of values.

3 *Returns:* The first iterator `i` in the range `[first1, last1)` such that for some iterator `j` in the range `[first2, last2)` E holds. ~~the following conditions hold: `*i == *j, pred(*i, *j) != false`.~~ Returns `last1` if `[first2, last2)` is empty or if no such iterator is found.

4 *Complexity:* At most `(last1 - first1) * (last2 - first2)` applications of the corresponding predicate and each projection.

30.5.8 Adjacent find [alg.adjacent.find]

```
template<class ForwardIterator>
  constexpr ForwardIterator
    adjacent_find(ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator
    adjacent_find(ExecutionPolicy&& exec,
                  ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
  constexpr ForwardIterator
    adjacent_find(ForwardIterator first, ForwardIterator last,
                  BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator, class BinaryPredicate>
  ForwardIterator
    adjacent_find(ExecutionPolicy&& exec,
                  ForwardIterator first, ForwardIterator last,
```

```
                    BinaryPredicate pred);

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

1 Let E be:

(1.1) — `*i == *(i + 1)` for the overloads with no parameter `pred`,

(1.2) — `pred(*i, *(i + 1)) != false` for the overloads with a parameter `pred` and no parameter `proj`,

(1.3) — `invoke(pred, invoke(proj, *i), invoke(proj, *(i + 1))) != false` for the overloads with both parameters `pred` and `proj`.

2 *Returns:* The first iterator `i` such that both `i` and `i + 1` are in the range `[first, last)` for which E holds. ~~the following corresponding conditions hold: `*i == *(i + 1)`, `pred(*i, *(i + 1)) != false`.~~ Returns `last` if no such iterator is found.

3 *Complexity:* For the overloads with no `ExecutionPolicy`, exactly `min((i - first) + 1, (last - first) - 1)` applications of the corresponding predicate, where `i` is `adjacent_find`'s return value, and no more than twice as many applications of the projection. For the overloads with an `ExecutionPolicy`, $\mathcal{O}$(`last - first`) applications of the corresponding predicate.

30.5.9 Count [alg.count]

```
template<class InputIterator, class T>
  constexpr typename iterator_traits<InputIterator>::difference_type
    count(InputIterator first, InputIterator last, const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  typename iterator_traits<ForwardIterator>::difference_type
    count(ExecutionPolicy&& exec,
          ForwardIterator first, ForwardIterator last, const T& value);

template<class InputIterator, class Predicate>
  constexpr typename iterator_traits<InputIterator>::difference_type
    count_if(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  typename iterator_traits<ForwardIterator>::difference_type
    count_if(ExecutionPolicy&& exec,
             ForwardIterator first, ForwardIterator last, Predicate pred);

namespace ranges {
  template<InputIterator I, Sentinel<I> S, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T*>
    constexpr iter_difference_type_t<I>
      count(I first, S last, const T& value, Proj proj = Proj{});
  template<InputRange Rng, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
    constexpr iter_difference_type_t<iterator_t<Rng>>
      count(Rng&& rng, const T& value, Proj proj = Proj{});
  template<InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr iter_difference_type_t<I>
      count_if(I first, S last, Pred pred, Proj proj = Proj{});
  template<InputRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    constexpr iter_difference_type_t<iterator_t<Rng>>
      count_if(Rng&& rng, Pred pred, Proj proj = Proj{});
}
```

1 Let *E* be:

(1.1) — `*i == value` for the overloads with no parameter `pred` or `proj`,

(1.2) — `pred(*i) != false` for the overloads with a parameter `pred` but no parameter `proj`,

(1.3) — `invoke(proj, *i) == value` for the overloads with a parameter `proj` but no parameter `pred`,

(1.4) — `invoke(pred, invoke(proj, *i)) != false` for the overloads both parameters `proj` and `pred`.

2 *Effects:* Returns the number of iterators `i` in the range `[first, last)` for which *E* holds. ~~the following corresponding conditions hold: `*i == value, pred(*i) != false`.~~

3 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

30.5.10 Mismatch [mismatch]

```
template<class InputIterator1, class InputIterator2>
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2);

template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, BinaryPredicate pred);

template<class InputIterator1, class InputIterator2>
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);

template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2,
             BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);

namespace ranges {
  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      class Proj1 = identity, class Proj2 = identity,
      IndirectRelation<projected<I1, Proj1>, projected<I2, Proj2>> Pred = ranges::equal_to<>>
```

```
    tagged_pair<tag::in1(I1), tag::in2(I2)>
    constexpr mismatch_result<I1, I2>
      mismatch(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<InputRange Rng1, InputRange Rng2,
      class Proj1 = identity, class Proj2 = identity,
      IndirectRelation<projected<iterator_t<Rng1>, Proj1>,
        projected<iterator_t<Rng2>, Proj2>> Pred = ranges::equal_to<>>
    tagged_pair<tag::in1(safe_iterator_t<Rng1>),
                tag::in2(safe_iterator_t<Rng2>)>
    constexpr mismatch_result<safe_iterator_t<Rng1>, safe_iterator_t<Rng2>>
      mismatch(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

1 Let `last2` be `first2 + (last1 - first1)` for the overloads with no parameter `last2` or `rng2`.

~~*Remarks:* If `last2` was not given in the argument list, it denotes `first2 + (last1 - first1)` below.~~

2 Let E be:

(2.1) — `!(*(first1 + n) == *(first2 + n))` for the overloads with no parameter `pred`,

(2.2) — `pred(*(first1 + n), *(first2 + n)) == false` for the overloads with a parameter `pred` and no parameter `proj1`,

(2.3) — `!invoke(pred, invoke(proj1, *(first1 + n)), invoke(proj2, *(first2 + n)))` for the overloads with both parameters `pred` and `proj1`.

3 *Returns:* ~~A pair of iterators `first1 + n` and `first2 + n`~~ `{ first1 + n, first2 + n }`, where `n` is the smallest integer such that E holds, ~~respectively,~~

(3.1) — ~~`!(*(first1 + n) == *(first2 + n))` or~~

(3.2) — ~~`pred(*(first1 + n), *(first2 + n)) == false`,~~

or `min(last1 - first1, last2 - first2)` if no such integer exists.

4 *Complexity:* At most `min(last1 - first1, last2 - first2)` applications of the corresponding predicate and each projection.

30.5.11 Equal [alg.equal]

```
template<class InputIterator1, class InputIterator2>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  bool equal(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2);

template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  bool equal(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, BinaryPredicate pred);

template<class InputIterator1, class InputIterator2>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, InputIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  bool equal(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);
```

```
template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, InputIterator2 last2,
                       BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  bool equal(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);

namespace ranges {
  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      class Pred = ranges::equal_to<>, class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
    constexpr bool equal(I1 first1, S1 last1, I2 first2, S2 last2,
                         Pred pred = Pred{},
                         Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<InputRange Rng1, InputRange Rng2, class Pred = ranges::equal_to<>,
      class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
    constexpr bool equal(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
                         Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

1 ~~*Remarks:* If `last2` was not given in the argument list, it denotes `first2 + (last1 - first1)` below.~~

2 Let:

(2.1) — `last2` be `first2 + (last1 - first1)` for the overloads with no parameter `last2` or `rng2`,

(2.2) — `pred` be `equal_to<>{}` for the overloads with no parameter `pred`,

(2.3) — E be:

(2.3.1) — `pred(*i, *(first2 + (i - first1)))` for the overloads with no parameter `proj1`,

(2.3.2) — `invoke(pred, invoke(proj1, *i), invoke(proj2, *(first2 + (i - first1))))` for the overloads with parameter `proj1`.

3 *Returns:* If `last1 - first1 != last2 - first2`, return `false`. Otherwise return `true` if E holds for every iterator `i` in the range `[first1, last1)` ~~the following corresponding conditions hold: `*i == *(first2 + (i - first1))`, `pred(*i, *(first2 + (i - first1))) != false`. Otherwise, returns `false`.~~

4 *Complexity:* If the types of `first1`, `last1`, `first2`, and `last2`:

(4.1) — meet the *Cpp17RandomAccessIterator* requirements (28.3.5.6) for the overloads in namespace `std`, or

(4.2) — pairwise model `SizedSentinel` (28.3.4.7) for the overloads in namespace `ranges`,

and `last1 - first1 != last2 - first2`, then no applications of the corresponding predicate and each projection; otherwise,

(4.3) — For the overloads with no `ExecutionPolicy`, at most $\min($`last1 - first1`$,$`last2 - first2`$)$ applications of the corresponding predicate and each projection.

(4.3.1) — ~~if `InputIterator1` and `InputIterator2` meet the *Cpp17RandomAccessIterator* requirements (28.3.5.6) and `last1 - first1 != last2 - first2`, then no applications of the corresponding predicate; otherwise,~~

(4.4) — For the overloads with an `ExecutionPolicy`, $\mathcal{O}(\min($`last1 - first1`$,$`last2 - first2`$))$ applications of the corresponding predicate.

(4.4.1) — ~~if `ForwardIterator1` and `ForwardIterator2` meet the *Cpp17RandomAccessIterator* requirements and `last1 - first1 != last2 - first2`, then no applications of the corresponding predicate; otherwise,~~

30.5.12 Is permutation [alg.is_permutation]

[...]

```
namespace ranges {
  template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
      Sentinel<I2> S2, class Pred = ranges::equal_to<>, class Proj1 = identity,
      class Proj2 = identity>
    requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
    constexpr bool is_permutation(I1 first1, S1 last1, I2 first2, S2 last2,
                                  Pred pred = Pred{},
                                  Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<ForwardRange Rng1, ForwardRange Rng2, class Pred = ranges::equal_to<>,
      class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
    constexpr bool is_permutation(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
                                  Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

[Editor's note: FIXME: This formulation in terms of range-and-a-half `ranges::equal` is broken, since we are not proposing a range-and-a-half `ranges::equal`.]

5 *Returns:* If `last1 - first1 != last2 - first2`, return `false`. Otherwise return `true` if there exists a permutation of the elements in the range `[first2, first2 + (last1 - first1))`, beginning with `I2 begin`, such that `ranges::equal(first1, last1, begin, pred, proj1, proj2)` returns `true`; otherwise, returns `false`.

6 *Complexity:* No applications of the corresponding predicate and projections if:

(6.1) — `S1` and `I1` model `SizedSentinel`~~`<S1, I1>` is satisfied~~,

(6.2) — `S2` and `I2` model `SizedSentinel`~~`<S2, I2>` is satisfied~~,

(6.3) — `last1 - first1 != last2 - first2`.

Otherwise, exactly `last1 - first1` applications of the corresponding predicate and projections if `ranges::equal(first1, last1, first2, last2, pred, proj1, proj2)` would return `true`; otherwise, at worst $\mathcal{O}(N^2)$, where N has the value `last1 - first1`.

30.5.13 Search [alg.search]

[...]

3 *Complexity:* At most `(last1 - first1) * (last2 - first2)` applications of the corresponding predicate.

```
namespace ranges {
  template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
      Sentinel<I2> S2, class Pred = ranges::equal_to<>,
      class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
    constexpr subrange<I1> I1
      search(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<ForwardRange Rng1, ForwardRange Rng2, class Pred = ranges::equal_to<>,
      class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
    constexpr safe_subrange_t<Rng1> safe_iterator_t<Rng1>
      search(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

[Editor's note: This wording incorporates the PR for stl2#180 and stl2#526).]

4 *Effects:* Finds a subsequence of equal values in a sequence.

5 *Returns:*

(5.1) — `{first1, first1}` if `[first2, last2)` is empty, (5.2) — Otherwise, returns `{i, i + (last2 - first2)}` where `i` is the first iterator ~~`i`~~ in the range `[first1, last1 - (last2 - first2))` such that for every non-negative integer `n` less than `last2 - first2` the following condition holds:

`invoke(pred, invoke(proj1, *(i + n)), invoke(proj2, *(first2 + n)))`~~` != false`~~.

(5.3) — Returns ~~`first1` if~~ `[`~~`first2`~~`, `~~`last2`~~`)` ~~is empty, otherwise returns `last1`~~ `{last1, last1}` if no such iterator is found.

6 *Complexity:* At most `(last1 - first1) * (last2 - first2)` applications of the corresponding predicate and projections.

```
template<class ForwardIterator, class Size, class T>
  constexpr ForwardIterator
    search_n(ForwardIterator first, ForwardIterator last,
             Size count, const T& value);
```

[...]

10 *Complexity:* At most `last - first` applications of the corresponding predicate.

```
namespace ranges {
  template<ForwardIterator I, Sentinel<I> S, class T,
      class Pred = ranges::equal_to<>, class Proj = identity>
    requires IndirectlyComparable<I, const T*, Pred, Proj>
    constexpr subrange<I>
      search_n(I first, S last, iter_difference~~_type~~_t<I> count,
               const T& value, Pred pred = Pred{}, Proj proj = Proj{});
  template<ForwardRange Rng, class T, class Pred = ranges::equal_to<>,
      class Proj = identity>
    requires IndirectlyComparable<iterator_t<Rng>, const T*, Pred, Proj>
    constexpr safe_~~iterator~~subrange_t<Rng>
      search_n(Rng&& rng, iter_difference~~_type~~_t<iterator_t<Rng>> count,
               const T& value, Pred pred = Pred{}, Proj proj = Proj{});
}
```

[Editor's note: This wording incorporates the PR for stl2#180 and stl2#526).]

11 *Effects:* Finds a subsequence of equal values in a sequence.

12 *Returns:* `{i, i + count}` where `i` is the first iterator ~~`i`~~ in the range `[first, last - count)` such that for every non-negative integer `n` less than `count`, the following condition holds: `invoke(pred, invoke(proj, *(i + n)), value)`~~` != false`~~. Returns `{last, last}` if no such iterator is found.

13 *Complexity:* At most `last - first` applications of the corresponding predicate and projection.

[...]

30.6 Mutating sequence operations [alg.modifying.operations]

30.6.1 Copy [alg.copy]

```
template<class InputIterator, class OutputIterator>
  constexpr OutputIterator copy(InputIterator first, InputIterator last,
                                OutputIterator result);
```

```
namespace ranges {
  template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
    requires IndirectlyCopyable<I, O>
    ~~tagged_pair<tag::in(I), tag::out(O)>~~
    constexpr copy_result<I, O>
      copy(I first, S last, O result);
  template<InputRange Rng, WeaklyIncrementable O>
    requires IndirectlyCopyable<iterator_t<Rng>, O>
    ~~tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>~~
    constexpr copy_result<safe_iterator_t<Rng>, O>
      copy(Rng&& rng, O result);
}
```

1 *Requires:* `result` shall not be in the range `[first, last)`.

2 *Effects:* Copies elements in the range `[first, last)` into the range `[result, result + (last - first))` starting from `first` and proceeding to `last`. For each non-negative integer `n < (last - first)`, performs `*(result + n) = *(first + n)`.

3 *Returns:*

(3.1) — `result + (last - first)` for the overload in namespace `std`, or

(3.2) — `{last, result - (last - first)}` for the overload in namespace `ranges`.

4 *Complexity:* Exactly `last - first` assignments.

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 copy(ExecutionPolicy&& policy,
                        ForwardIterator1 first, ForwardIterator1 last,
                        ForwardIterator2 result);
```

5 *Requires:* The ranges `[first, last)` and `[result, result + (last - first))` shall not overlap.

6 *Effects:* Copies elements in the range `[first, last)` into the range `[result, result + (last - first))`. For each non-negative integer `n < (last - first)`, performs `*(result + n) = *(first + n)`.

7 *Returns:* `result + (last - first)`.

8 *Complexity:* Exactly `last - first` assignments.

```
template<class InputIterator, class Size, class OutputIterator>
  constexpr OutputIterator copy_n(InputIterator first, Size n,
                                  OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class Size, class ForwardIterator2>
  ForwardIterator2 copy_n(ExecutionPolicy&& exec,
                          ForwardIterator1 first, Size n,
                          ForwardIterator2 result);

namespace ranges {
  template<InputIterator I, WeaklyIncrementable O>
    requires IndirectlyCopyable<I, O>
    ~~tagged_pair<tag::in(I), tag::out(O)>~~
    constexpr copy_n_result<I, O>
      copy_n(I first, iter_difference~~_type~~_t<I> n, O result);
}
```

[Editor's note: This wording incorporates the PR for stl2#498).]

9 Let M be max(`n`, 0).

10 *Effects:* For each non-negative integer $i <$ ~~n~~M, performs `*(result + i) = *(first + i)`.

11 *Returns:*

(11.1) — `result +` ~~`n`~~`M` for the overload in namespace `std`, or

(11.2) — `{last +` ~~`n`~~`M, result +` ~~`n`~~`M}` for the overload in namespace `ranges`.

12 *Complexity:* Exactly ~~`n`~~`M` assignments.

```
template<class InputIterator, class OutputIterator, class Predicate>
  constexpr OutputIterator copy_if(InputIterator first, InputIterator last,
                                   OutputIterator result, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate>
  ForwardIterator2 copy_if(ExecutionPolicy&& exec,
                           ForwardIterator1 first, ForwardIterator1 last,
                           ForwardIterator2 result, Predicate pred);

namespace ranges {
  template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires IndirectlyCopyable<I, O>
    ~~tagged_pair<tag::in(I), tag::out(O)>~~
    constexpr copy_if_result<I, O>
```

```
      copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});
  template<InputRange Rng, WeaklyIncrementable O, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    requires IndirectlyCopyable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    constexpr copy_if_result<safe_iterator_t<Rng>, O>
      copy_if(Rng&& rng, O result, Pred pred, Proj proj = Proj{});
}
```

13 Let E be:

(13.1) — `pred(*i)` for the overloads in namespace `std`, or

(13.2) — `invoke(pred, invoke(proj, *i))` for the overloads in namespace `ranges`.

and N be the number of iterators `i` in the range `[first, last)` for which the condition E ~~`invoke(pred, invoke(proj, *i))`~~ holds.

14 *Requires:* The ranges `[first, last)` and `[result, result + (last - first))` shall not overlap. [*Note:* For the overload with an `ExecutionPolicy`, there may be a performance cost if `iterator_traits<ForwardIterator1>::value_type` is not `MoveConstructible` ([tab:moveconstructible]). — *end note*]

15 *Effects:* Copies all of the elements referred to by the iterator `i` in the range `[first, last)` for which E ~~`pred(*i)`~~ is `true`.

16 *Returns:* ~~The end of the resulting range.~~

(16.1) — `result + ` N for the overload in namespace `std`, or

(16.2) — `{last, result + ` N`}` for the overloads in namespace `ranges`.

17 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

18 *Remarks:* Stable ([algorithm.stable]).

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
  constexpr BidirectionalIterator2
    copy_backward(BidirectionalIterator1 first,
                  BidirectionalIterator1 last,
                  BidirectionalIterator2 result);

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

19 *Requires:* `result` shall not be in the range `(first, last]`.

20 *Effects:* Copies elements in the range `[first, last)` into the range `[result - (last-first), result)` starting from `last - 1` and proceeding to `first`.[5] For each positive integer `n <= (last - first)`, performs `*(result - n) = *(last - n)`.

21 *Returns:*

(21.1) — `result - (last - first)` for the overload in namespace `std`, or

(21.2) — `{last, result - (last - first)}` for the overloads in namespace `ranges`.

22 *Complexity:* Exactly `last - first` assignments.

5) `copy_backward` should be used instead of copy when `last` is in the range `[result - (last - first), result)`.

30.6.2 Move [alg.move]

```
template<class InputIterator, class OutputIterator>
  constexpr OutputIterator move(InputIterator first, InputIterator last,
                                OutputIterator result);
```

1 *Requires:* `result` shall not be in the range `[first, last)`.

2 *Effects:* Moves elements in the range `[first, last)` into the range `[result, result + (last - first))` starting from first and proceeding to last. For each non-negative integer `n < (last-first)`, performs `*(result + n) = std::move(*(first + n))`.

3 *Returns:* `result + (last - first)`.

4 *Complexity:* Exactly `last - first` move assignments.

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 move(ExecutionPolicy&& policy,
                        ForwardIterator1 first, ForwardIterator1 last,
                        ForwardIterator2 result);
```

5 *Requires:* The ranges `[first, last)` and `[result, result + (last - first))` shall not overlap.

6 *Effects:* Moves elements in the range `[first, last)` into the range `[result, result + (last - first))`. For each non-negative integer `n < (last - first)`, performs `*(result + n) = std::move(*(first + n))`.

7 *Returns:* `result + (last - first)`.

8 *Complexity:* Exactly `last - first` assignments.

```
namespace ranges {
  template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
    requires IndirectlyMovable<I, O>
    ~~tagged_pair<tag::in(I), tag::out(O)>~~
    constexpr move_result<I, O>
      move(I first, S last, O result);
  template<InputRange Rng, WeaklyIncrementable O>
    requires IndirectlyMovable<iterator_t<Rng>, O>
    ~~tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>~~
    constexpr move_result<safe_iterator_t<Rng>, O>
      move(Rng&& rng, O result);
}
```

9 *Requires:* `result` shall not be in the range `[first, last)`.

10 *Effects:* Moves elements in the range `[first, last)` into the range `[result, result + (last - first))` starting from first and proceeding to last. For each non-negative integer `n < (last-first)`, performs `*(result + n) = ranges::iter_move(first + n)`.

11 *Returns:* `{last, result + (last - first)}`.

12 *Complexity:* Exactly `last - first` move assignments.

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
  constexpr BidirectionalIterator2
    move_backward(BidirectionalIterator1 first, BidirectionalIterator1 last,
                  BidirectionalIterator2 result);
```

13 *Requires:* `result` shall not be in the range `(first, last]`.

14 *Effects:* Moves elements in the range `[first, last)` into the range `[result - (last-first), result)` starting from `last - 1` and proceeding to first.[6] For each positive integer `n <= (last - first)`, performs `*(result - n) = std::move(*(last - n))`.

15 *Returns:* `result - (last - first)`.

16 *Complexity:* Exactly `last - first` assignments.

6) `move_backward` should be used instead of move when last is in the range `[result - (last - first), result)`.

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

17 *Requires:* `result` shall not be in the range `(first, last]`.

18 *Effects:* Moves elements in the range `[first, last)` into the range `[result - (last-first), result)` starting from `last - 1` and proceeding to first.[7] For each positive integer `n <= (last - first)`, performs `*(result - n) = ranges::iter_move(last - n)`.

19 *Returns:* `{last, result - (last - first)}`.

20 *Complexity:* Exactly `last - first` assignments.

30.6.3 Swap [alg.swap]

```
template<class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2
    swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2
    swap_ranges(ExecutionPolicy&& exec,
                ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2);
```

1 *Requires:* The two ranges `[first1, last1)` and `[first2, first2 + (last1 - first1))` shall not overlap. `*(first1 + n)` shall be swappable with[swappable.requirements] `*(first2 + n)`.

2 *Effects:* For each non-negative integer `n < (last1 - first1)` performs: `swap(*(first1 + n), *(first2 + n))`.

3 *Returns:* `first2 + (last1 - first1)`.

4 *Complexity:* Exactly `last1 - first1` swaps.

```
namespace ranges {
  template<ForwardInputIterator I1, Sentinel<I1> S1, ForwardInputIterator I2, Sentinel<I2> S2>
    requires IndirectlySwappable<I1, I2>
    tagged_pair<tag::in1(I1), tag::in2(I2)>
    constexpr swap_ranges_result<I1, I2>
      swap_ranges(I1 first1, S1 last1, I2 first2, S2 last2);
  template<ForwardInputRange Rng1, ForwardInputRange Rng2>
    requires IndirectlySwappable<iterator_t<Rng1>, iterator_t<Rng2>>
    tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::in2(safe_iterator_t<Rng2>)>
    constexpr swap_ranges_result<safe_iterator_t<Rng1>, safe_iterator_t<Rng2>>
      swap_ranges(Rng1&& rng1, Rng2&& rng2);
}
```

[Editor's note: This wording integrates the PR for stl2#415.]

5 *Requires:* The two ranges `[first1, last1)` and `[first2, last2)` shall not overlap. ~~`*(first1 + n)` shall be swappable with (22.3.11) `*(first2 + n)`.~~

6 *Effects:* For each non-negative integer `n < min(last1 - first1, last2 - first2)` performs: `ranges::iter_swap(first1 + n, first2 + n)`.

7 *Returns:* `{first1 + n, first2 + n}`, where `n` is `min(last1 - first1, last2 - first2)`.

8 *Complexity:* Exactly `min(last1 - first1, last2 - first2)` swaps.

7) `move_backward` should be used instead of move when last is in the range `[result - (last - first), result)`.

```
template<class ForwardIterator1, class ForwardIterator2>
  void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

[...]

30.6.4 Transform [alg.transform]

```
template<class InputIterator, class OutputIterator,
         class UnaryOperation>
  constexpr OutputIterator
    transform(InputIterator first, InputIterator last,
              OutputIterator result, UnaryOperation op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class UnaryOperation>
  ForwardIterator2
    transform(ExecutionPolicy&& exec,
              ForwardIterator1 first, ForwardIterator1 last,
              ForwardIterator2 result, UnaryOperation op);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class BinaryOperation>
  constexpr OutputIterator
    transform(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, OutputIterator result,
              BinaryOperation binary_op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class BinaryOperation>
  ForwardIterator
    transform(ExecutionPolicy&& exec,
              ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator result,
              BinaryOperation binary_op);
```

1 *Requires:* `op` and `binary_op` shall not invalidate iterators or subranges, or modify elements in the ranges

(1.1) — `[first1, last1]`,

(1.2) — `[first2, first2 + (last1 - first1)]`, and

(1.3) — `[result, result + (last1 - first1)]`.[8]

2 *Effects:* Assigns through every iterator `i` in the range `[result, result + (last1 - first1))` a new corresponding value equal to `op(*(first1 + (i - result)))` or `binary_op(*(first1 + (i - result)), *(first2 + (i - result)))`.

3 *Returns:* `result + (last1 - first1)`.

4 *Complexity:* Exactly `last1 - first1` applications of `op` or `binary_op`. This requirement also applies to the overload with an `ExecutionPolicy` .

5 *Remarks:* `result` may be equal to `first` in case of unary transform, or to `first1` or `first2` in case of binary transform.

```
namespace ranges {
  template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
      CopyConstructible F, class Proj = identity>
    requires Writable<O, indirect_result_of_t<F&(, projected<I, Proj>)>>
    tagged_pair<tag::in(I), tag::out(O)>
    constexpr unary_transform_result<I, O>
      transform(I first, S last, O result, F op, Proj proj = Proj{});
  template<InputRange Rng, WeaklyIncrementable O, CopyConstructible F,
      class Proj = identity>
    requires Writable<O, indirect_result_of_t<F&(,
      projected<iterator_t<R>, Proj>)>>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    constexpr unary_transform_result<safe_iterator_t<Rng>, O>
```

8) The use of fully closed ranges is intentional.

```
      transform(Rng&& rng, O result, F op, Proj proj = Proj{});
  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      WeaklyIncrementable O, CopyConstructible F, class Proj1 = identity,
      class Proj2 = identity>
    requires Writable<O, indirect_result_of_t<F&(, projected<I1, Proj1>,
      projected<I2, Proj2>)>>
    tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    constexpr binary_transform_result<I1, I2, O>
      transform(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
      CopyConstructible F, class Proj1 = identity, class Proj2 = identity>
    requires Writable<O, indirect_result_of_t<F&(,
      projected<iterator_t<Rng1>, Proj1>, projected<iterator_t<Rng2>, Proj2>)>>
    tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
                 tag::in2(safe_iterator_t<Rng2>),
                 tag::out(O)>
    constexpr binary_transform_result<safe_iterator_t<Rng1>, safe_iterator_t<Rng2>, O>
      transform(Rng1&& rng1, Rng2&& rng2, O result,
                F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

6 Let N be `(last1 - first1)` for unary transforms, or `min(last1 - first1, last2 - first2) for binary transforms`.

7 *Effects:* Assigns through every iterator `i` in the range `[result, result +` N`)` a new corresponding value equal to `invoke(op, invoke(proj, *(first1 + (i - result))))` or `invoke(binary_op, invoke(proj1, *(first1 + (i - result))), invoke(proj2, *(first2 + (i - result))))`.

8 *Requires:* `op` and `binary_op` shall not invalidate iterators or subranges, or modify elements in the ranges `[first1, first1 +` N`]`, `[first2, first2 +` N`]`, and `[result, result +` N`]`.[9]

9 *Returns:* `{first1 +` N`, result +` N`}` or ~~`make_tagged_tuple<tag::in1, tag::in2, tag::out>(`~~`{` `first1 +` N`, first2 +` N`, result +` N~~`)`~~`}`.

10 *Complexity:* Exactly N applications of `op` or `binary_op` and the corresponding projection(s).

11 *Remarks:* `result` may be equal to `first1` in case of unary transform, or to `first1` or `first2` in case of binary transform.

30.6.5 Replace [alg.replace]

```
template<class ForwardIterator, class T>
  constexpr void replace(ForwardIterator first, ForwardIterator last,
                         const T& old_value, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  void replace(ExecutionPolicy&& exec,
               ForwardIterator first, ForwardIterator last,
               const T& old_value, const T& new_value);

template<class ForwardIterator, class Predicate, class T>
  constexpr void replace_if(ForwardIterator first, ForwardIterator last,
                            Predicate pred, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator, class Predicate, class T>
  void replace_if(ExecutionPolicy&& exec,
                  ForwardIterator first, ForwardIterator last,
                  Predicate pred, const T& new_value);
```

1 *Requires:* The expression `*first = new_value` shall be valid.

2 *Effects:* Substitutes elements referred by the iterator `i` in the range `[first, last)` with `new_value`, when the following corresponding conditions hold: `*i == old_value`, `pred(*i) != false`.

3 *Complexity:* Exactly `last - first` applications of the corresponding predicate.

9) The use of fully closed ranges is intentional.

```
namespace ranges {
  template<InputIterator I, Sentinel<I> S, class T1, class T2, class Proj = identity>
    requires Writable<I, const T2&> &&
      IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T1*>
    constexpr I
      replace(I first, S last, const T1& old_value, const T2& new_value, Proj proj = Proj{});
  template<InputRange Rng, class T1, class T2, class Proj = identity>
    requires Writable<iterator_t<Rng>, const T2&> &&
      IndirectRelation<ranges::equal_to<>, projected<iterator_t<Rng>, Proj>, const T1*>
    constexpr safe_iterator_t<Rng>
      replace(Rng&& rng, const T1& old_value, const T2& new_value, Proj proj = Proj{});
  template<InputIterator I, Sentinel<I> S, class T, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires Writable<I, const T&>
    constexpr I replace_if(I first, S last, Pred pred, const T& new_value, Proj proj = Proj{});
  template<InputRange Rng, class T, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    requires Writable<iterator_t<Rng>, const T&>
    constexpr safe_iterator_t<Rng>
      replace_if(Rng&& rng, Pred pred, const T& new_value, Proj proj = Proj{});
}
```

4 *Effects:* Assigns `new_value` through each iterator `i` in the range `[first, last)` when the following corresponding conditions hold: `invoke(proj, *i) == old_value`, `invoke(pred, invoke(proj, *i)) != false`.

5 *Returns:* `last`.

6 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

```
template<class InputIterator, class OutputIterator, class T>
  constexpr OutputIterator
    replace_copy(InputIterator first, InputIterator last,
                 OutputIterator result,
                 const T& old_value, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T>
  ForwardIterator2
    replace_copy(ExecutionPolicy&& exec,
                 ForwardIterator1 first, ForwardIterator1 last,
                 ForwardIterator2 result,
                 const T& old_value, const T& new_value);

template<class InputIterator, class OutputIterator, class Predicate, class T>
  constexpr OutputIterator
    replace_copy_if(InputIterator first, InputIterator last,
                    OutputIterator result,
                    Predicate pred, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
       class Predicate, class T>
  ForwardIterator2
    replace_copy_if(ExecutionPolicy&& exec,
                    ForwardIterator1 first, ForwardIterator1 last,
                    ForwardIterator2 result,
                    Predicate pred, const T& new_value);
```

7 *Requires:* The results of the expressions `*first` and `new_value` shall be writable (28.3.1) to the `result` output iterator. The ranges `[first, last)` and `[result, result + (last - first))` shall not overlap.

8 *Effects:* Assigns to every iterator `i` in the range `[result, result + (last - first))` either `new_value` or `*(first + (i - result))` depending on whether the following corresponding conditions hold:

```
*(first + (i - result)) == old_value
pred(*(first + (i - result))) != false
```

9 *Returns:* `result + (last - first)`.

10 *Complexity:* Exactly `last - first` applications of the corresponding predicate.

```
namespace ranges {
  template<InputIterator I, Sentinel<I> S, class T1, class T2, OutputIterator<const T2&> O,
      class Proj = identity>
    requires IndirectlyCopyable<I, O> &&
      IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T1*>
    tagged_pair<tag::in(I), tag::out(O)>
    constexpr replace_copy_result<I, O>
      replace_copy(I first, S last, O result, const T1& old_value, const T2& new_value,
                   Proj proj = Proj{});
  template<InputRange Rng, class T1, class T2, OutputIterator<const T2&> O,
      class Proj = identity>
    requires IndirectlyCopyable<iterator_t<Rng>, O> &&
      IndirectRelation<ranges::equal_to<>, projected<iterator_t<Rng>, Proj>, const T1*>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    constexpr replace_copy_result<safe_iterator_t<Rng>, O>
      replace_copy(Rng&& rng, O result, const T1& old_value, const T2& new_value,
                   Proj proj = Proj{});

  template<InputIterator I, Sentinel<I> S, class T, OutputIterator<const T&> O,
      class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires IndirectlyCopyable<I, O>
    tagged_pair<tag::in(I), tag::out(O)>
    constexpr replace_copy_if_result<I, O>
      replace_copy_if(I first, S last, O result, Pred pred, const T& new_value,
                      Proj proj = Proj{});
  template<InputRange Rng, class T, OutputIterator<const T&> O, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    requires IndirectlyCopyable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    constexpr replace_copy_if_result<safe_iterator_t<Rng>, O>
      replace_copy_if(Rng&& rng, O result, Pred pred, const T& new_value,
                      Proj proj = Proj{});
}
```

11 *Requires:* The ranges `[first, last)` and `[result, result + (last - first))` shall not overlap.

12 *Effects:* Assigns to every iterator `i` in the range `[result, result + (last - first))` either `new_value` or `*(first + (i - result))` depending on whether the following corresponding conditions hold:

```
invoke(proj, *(first + (i - result))) == old_value
invoke(pred, invoke(proj, *(first + (i - result)))) != false
```

13 *Returns:* `{last, result + (last - first)}`.

14 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

30.6.6 Fill [alg.fill]

```
template<class ForwardIterator, class T>
  constexpr void fill(ForwardIterator first, ForwardIterator last, const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  void fill(ExecutionPolicy&& exec,
            ForwardIterator first, ForwardIterator last, const T& value);

template<class OutputIterator, class Size, class T>
  constexpr OutputIterator fill_n(OutputIterator first, Size n, const T& value);
template<class ExecutionPolicy, class ForwardIterator, class Size, class T>
  ForwardIterator fill_n(ExecutionPolicy&& exec,
                         ForwardIterator first, Size n, const T& value);
```

1 *Requires:* The expression `value` shall be writable (28.3.1) to the output iterator. The type `Size` shall be convertible to an integral type (cxxrefconv.integral, cxxrefclass.conv).

2 *Effects:* The `fill` algorithms assign `value` through all the iterators in the range `[first, last)`. The `fill_n` algorithms assign `value` through all the iterators in the range `[first, first + n)` if `n` is positive, otherwise they do nothing.

3 *Returns:* `fill_n` returns `first + n` for non-negative values of `n` and `first` for negative values.

4 *Complexity:* Exactly `last - first`, `n`, or 0 assignments, respectively.

```
namespace ranges {
  template<class T, OutputIterator<const T&> O, Sentinel<O> S>
    constexpr O fill(O first, S last, const T& value);
  template<class T, OutputRange<const T&> Rng>
    constexpr safe_iterator_t<Rng> fill(Rng&& rng, const T& value);
  template<class T, OutputIterator<const T&> O>
    constexpr O fill_n(O first, iter_difference_type_t<O> n, const T& value);
}
```

5 *Effects:* `fill` assigns `value` through all the iterators in the range `[first, last)`. `fill_n` assigns `value` through all the iterators in the counted range `[first, n)` if `n` is positive, otherwise it does nothing.

6 *Returns:* `last`, where `last` is `first + max(n, 0)` for `fill_n`.

7 *Complexity:* Exactly `last - first` assignments.

30.6.7 Generate [alg.generate]

```
template<class ForwardIterator, class Generator>
  constexpr void generate(ForwardIterator first, ForwardIterator last,
                          Generator gen);
template<class ExecutionPolicy, class ForwardIterator, class Generator>
  void generate(ExecutionPolicy&& exec,
                ForwardIterator first, ForwardIterator last,
                Generator gen);

template<class OutputIterator, class Size, class Generator>
  constexpr OutputIterator generate_n(OutputIterator first, Size n, Generator gen);
template<class ExecutionPolicy, class ForwardIterator, class Size, class Generator>
  ForwardIterator generate_n(ExecutionPolicy&& exec,
                             ForwardIterator first, Size n, Generator gen);
```

1 *Requires:* `gen` takes no arguments, `Size` shall be convertible to an integral type (cxxrefconv.integral, cxxrefclass.conv).

2 *Effects:* The `generate` algorithms invoke the function object `gen` and assign the return value of `gen` through all the iterators in the range `[first, last)`. The `generate_n` algorithms invoke the function object `gen` and assign the return value of `gen` through all the iterators in the range `[first, first + n)` if `n` is positive, otherwise they do nothing.

3 *Returns:* `generate_n` returns `first + n` for non-negative values of `n` and `first` for negative values.

4 *Complexity:* Exactly `last - first`, `n`, or 0 invocations of `gen` and assignments, respectively.

```
namespace ranges {
  template<Iterator O, Sentinel<O> S, CopyConstructible F>
      requires Invocable<F&> && Writable<O, result_of_t<F&()>invoke_result_t<F&>>
    constexpr O generate(O first, S last, F gen);
  template<class Rng, CopyConstructible F>
      requires Invocable<F&> && OutputRange<Rng, result_of_t<F&()>invoke_result_t<F&>>
    constexpr safe_iterator_t<Rng> generate(Rng&& rng, F gen);
  template<Iterator O, CopyConstructible F>
      requires Invocable<F&> && Writable<O, result_of_t<F&()>invoke_result_t<F&>>
    constexpr O generate_n(O first, iter_difference_type_t<O> n, F gen);
}
```

5 *Effects:* The generate algorithms invoke the function object `gen` and assign the return value of `gen` through all the iterators in the range `[first, last)`. The `generate_n` algorithm invokes the function object `gen` and assigns the return value of `gen` through all the iterators in the counted range `[first, n)` if `n` is positive, otherwise it does nothing.

6 *Returns:* `last`, where `last` is `first + max(n, 0)` for `generate_n`.

7 *Complexity:* Exactly `last - first` evaluations of `invoke(gen)` and assignments.

30.6.8 Remove [alg.remove]

```
template<class ForwardIterator, class T>
  constexpr ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                                   const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  ForwardIterator remove(ExecutionPolicy&& exec,
                         ForwardIterator first, ForwardIterator last,
                         const T& value);

template<class ForwardIterator, class Predicate>
  constexpr ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                                      Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator remove_if(ExecutionPolicy&& exec,
                            ForwardIterator first, ForwardIterator last,
                            Predicate pred);
```

1 *Requires:* The type of `*first` shall satisfy the *Cpp17MoveAssignable* requirements ([tab:moveassignable]).

2 *Effects:* Eliminates all the elements referred to by iterator `i` in the range `[first, last)` for which the following corresponding conditions hold: `*i == value, pred(*i) != false`.

3 *Returns:* The end of the resulting range.

4 *Remarks:* Stable[algorithm.stable].

5 *Complexity:* Exactly `last - first` applications of the corresponding predicate.

6 [*Note:* Each element in the range `[ret, last)`, where `ret` is the returned value, has a valid but unspecified state, because the algorithms can eliminate elements by moving from elements that were originally in that range. — *end note*]

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

7 *Effects:* Eliminates all the elements referred to by iterator `i` in the range `[first, last)` for which the following corresponding conditions hold: `invoke(proj, *i) == value`, `invoke(pred, invoke(proj, *i)) != false`.

8 *Returns:* The end of the resulting range.

9 *Remarks:* Stable ([algorithm.stable]).

10 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

11 [*Note:* Each element in the range `[ret, last)`, where `ret` is the returned value, has a valid but unspecified state, because the algorithms can eliminate elements by moving from elements that were originally in that range. — *end note*]

```
template<class InputIterator, class OutputIterator, class T>
  constexpr OutputIterator
    remove_copy(InputIterator first, InputIterator last,
                OutputIterator result, const T& value);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class T>
  ForwardIterator2
    remove_copy(ExecutionPolicy&& exec,
                ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result, const T& value);

template<class InputIterator, class OutputIterator, class Predicate>
  constexpr OutputIterator
    remove_copy_if(InputIterator first, InputIterator last,
                   OutputIterator result, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate>
  ForwardIterator2
    remove_copy_if(ExecutionPolicy&& exec,
                   ForwardIterator1 first, ForwardIterator1 last,
                   ForwardIterator2 result, Predicate pred);
```

12 *Requires:* The ranges `[first, last)` and `[result, result + (last - first))` shall not overlap. The expression `*result = *first` shall be valid. [*Note:* For the overloads with an `ExecutionPolicy`, there may be a performance cost if `iterator_traits<ForwardIterator1>::value_type` is not `MoveConstructible` ([tab:moveconstructible]). — *end note*]

13 *Effects:* Copies all the elements referred to by the iterator `i` in the range `[first, last)` for which the following corresponding conditions do not hold: `*i == value, pred(*i) != false`.

14 *Returns:* The end of the resulting range.

15 *Complexity:* Exactly `last - first` applications of the corresponding predicate.

16 *Remarks:* Stable[algorithm.stable].

```
namespace ranges {
  template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class T,
      class Proj = identity>
    requires IndirectlyCopyable<I, O> &&
      IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T*>
    tagged_pair<tag::in(I), tag::out(O)>
    constexpr remove_copy_result<I, O>
      remove_copy(I first, S last, O result, const T& value, Proj proj = Proj{});
  template<InputRange Rng, WeaklyIncrementable O, class T, class Proj = identity>
    requires IndirectlyCopyable<iterator_t<Rng>, O> &&
      IndirectRelation<ranges::equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    constexpr remove_copy_result<safe_iterator_t<Rng>, O>
      remove_copy(Rng&& rng, O result, const T& value, Proj proj = Proj{});
  template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
      class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires IndirectlyCopyable<I, O>
    tagged_pair<tag::in(I), tag::out(O)>
    constexpr remove_copy_if_result<I, O>
      remove_copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});
  template<InputRange Rng, WeaklyIncrementable O, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    requires IndirectlyCopyable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    constexpr remove_copy_if_result<safe_iterator_t<Rng>, O>
      remove_copy_if(Rng&& rng, O result, Pred pred, Proj proj = Proj{});
```

```
}
```

17 *Requires:* The ranges `[first, last)` and `[result, result + (last - first))` shall not overlap.

18 *Effects:* Copies all the elements referred to by the iterator `i` in the range `[first, last)` for which the following corresponding conditions do not hold: `invoke(proj, *i) == value`, `invoke(pred, invoke(proj, *i)) != false`.

19 *Returns:* ~~A pair consisting of `last` and the end of the resulting range~~ `{last, result + (last - first)}`.

20 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

21 *Remarks:* Stable ([algorithm.stable]).

30.6.9 Unique [alg.unique]

```
template<class ForwardIterator>
  constexpr ForwardIterator unique(ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator unique(ExecutionPolicy&& exec,
                         ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
  constexpr ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                                   BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator, class BinaryPredicate>
  ForwardIterator unique(ExecutionPolicy&& exec,
                         ForwardIterator first, ForwardIterator last,
                         BinaryPredicate pred);
```

1 *Requires:* The comparison function shall be an equivalence relation. The type of `*first` shall satisfy the *Cpp17MoveAssignable* requirements ([tab:moveassignable]).

2 *Effects:* For a nonempty range, eliminates all but the first element from every consecutive group of equivalent elements referred to by the iterator `i` in the range `[first + 1, last)` for which the following conditions hold: `*(i - 1) == *i` or `pred(*(i - 1), *i) != false`.

3 *Returns:* The end of the resulting range.

4 *Complexity:* For nonempty ranges, exactly `(last - first) - 1` applications of the corresponding predicate.

```
namespace ranges {
  template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectRelation<projected<I, Proj>> R = ranges::equal_to<>>
    requires Permutable<I>
    constexpr I unique(I first, S last, R comp = R{}, Proj proj = Proj{});
  template<ForwardRange Rng, class Proj = identity,
      IndirectRelation<projected<iterator_t<Rng>, Proj>> R = ranges::equal_to<>>
    requires Permutable<iterator_t<Rng>>
    constexpr safe_iterator_t<Rng>
      unique(Rng&& rng, R comp = R{}, Proj proj = Proj{});
}
```

5 *Effects:* For a nonempty range, eliminates all but the first element from every consecutive group of equivalent elements referred to by the iterator `i` in the range `[first + 1, last)` for which the following conditions hold: `invoke(proj, *(i - 1)) == invoke(proj, *i)` or `invoke(pred, invoke(proj, *(i - 1)), invoke(proj, *i)) != false`.

6 *Returns:* The end of the resulting range.

7 *Complexity:* For nonempty ranges, exactly `(last - first) - 1` applications of the corresponding predicate and no more than twice as many applications of the projection.

```
template<class InputIterator, class OutputIterator>
  constexpr OutputIterator
    unique_copy(InputIterator first, InputIterator last,
                OutputIterator result);
```

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2
    unique_copy(ExecutionPolicy&& exec,
                ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result);

template<class InputIterator, class OutputIterator,
         class BinaryPredicate>
  constexpr OutputIterator
    unique_copy(InputIterator first, InputIterator last,
                OutputIterator result, BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  ForwardIterator2
    unique_copy(ExecutionPolicy&& exec,
                ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result, BinaryPredicate pred);
```

8 *Requires:*

(8.1) — The comparison function shall be an equivalence relation.

(8.2) — The ranges `[first, last)` and `[result, result+(last-first))` shall not overlap.

(8.3) — The expression `*result = *first` shall be valid.

(8.4) — For the overloads with no `ExecutionPolicy`, let `T` be the value type of `InputIterator`. If `InputIterator` meets the forward iterator requirements, then there are no additional requirements for `T`. Otherwise, if `OutputIterator` meets the forward iterator requirements and its value type is the same as `T`, then `T` shall be *Cpp17CopyAssignable* ([tab:copyassignable]). Otherwise, `T` shall be both *Cpp17CopyConstructible* ([tab:copyconstructible]) and *Cpp17CopyAssignable*. [*Note:* For the overloads with an `ExecutionPolicy`, there may be a performance cost if the value type of `ForwardIterator1` is not both *Cpp17CopyConstructible* and *Cpp17CopyAssignable*. — *end note*]

9 *Effects:* Copies only the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first, last)` for which the following corresponding conditions hold: `*i == *(i - 1)` or `pred(*i, *(i - 1)) != false`.

10 *Returns:* The end of the resulting range.

11 *Complexity:* For nonempty ranges, exactly `last - first - 1` applications of the corresponding predicate.

```
namespace ranges {
  template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
      class Proj = identity, IndirectRelation<projected<I, Proj>> R = ranges::equal_to<>>
    requires IndirectlyCopyable<I, O> &&
      (ForwardIterator<I> ||
      (InputIterator<O> && Same<iter_value~~_type~~_t<I>, iter_value~~_type~~_t<O>>) ||
      IndirectlyCopyableStorable<I, O>)
    ~~tagged_pair<tag::in(I), tag::out(O)>~~
    constexpr unique_copy_result<I, O>
      unique_copy(I first, S last, O result, R comp = R{}, Proj proj = Proj{});
  template<InputRange Rng, WeaklyIncrementable O, class Proj = identity,
      IndirectRelation<projected<iterator_t<Rng>, Proj>> R = ranges::equal_to<>>
    requires IndirectlyCopyable<iterator_t<Rng>, O> &&
      (ForwardIterator<iterator_t<Rng>> ||
      (InputIterator<O> && Same<iter_value~~_type~~_t<iterator_t<Rng>>, iter_value~~_type~~_t<O>>) ||
      IndirectlyCopyableStorable<iterator_t<Rng>, O>)
    ~~tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>~~
    constexpr unique_copy_result<safe_iterator_t<Rng>, O>
      unique_copy(Rng&& rng, O result, R comp = R{}, Proj proj = Proj{});
}
```

12 *Requires:* The ranges `[first, last)` and `[result, result + (last - first))` shall not overlap.

13 *Effects:* Copies only the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first, last)` for which the following corresponding conditions hold:

```
invoke(proj, *i) == invoke(proj, *(i - 1))
```

or

```
invoke(pred, invoke(proj, *i), invoke(proj, *(i - 1))) != false.
```

14 *Returns:* ~~A pair consisting of `last` and the end of the resulting range~~ `{last, result + (last - first)}`.

15 *Complexity:* For nonempty ranges, exactly `last - first - 1` applications of the corresponding predicate and no more than twice as many applications of the projection.

30.6.10 Reverse [alg.reverse]

```
template<class BidirectionalIterator>
  void reverse(BidirectionalIterator first, BidirectionalIterator last);
template<class ExecutionPolicy, class BidirectionalIterator>
  void reverse(ExecutionPolicy&& exec,
               BidirectionalIterator first, BidirectionalIterator last);
```

1 *Requires:* `BidirectionalIterator` shall satisfy the *Cpp17ValueSwappable* requirements ([swappable.requirements]).

2 *Effects:* For each non-negative integer `i < (last - first) / 2`, applies `iter_swap` to all pairs of iterators `first + i, (last - i) - 1`.

3 *Complexity:* Exactly `(last - first)/2` swaps.

```
namespace ranges {
  template<BidirectionalIterator I, Sentinel<I> S>
    requires Permutable<I>
    constexpr I reverse(I first, S last);
  template<BidirectionalRange Rng>
    requires Permutable<iterator_t<Rng>>
    constexpr safe_iterator_t<Rng> reverse(Rng&& rng);
}
```

4 *Effects:* For each non-negative integer `i < (last - first)/2`, applies `iter_swap` to all pairs of iterators `first + i, (last - i) - 1`.

5 *Returns:* `last`.

6 *Complexity:* Exactly `(last - first)/2` swaps.

```
template<class BidirectionalIterator, class OutputIterator>
  constexpr OutputIterator
    reverse_copy(BidirectionalIterator first, BidirectionalIterator last,
                 OutputIterator result);
template<class ExecutionPolicy, class BidirectionalIterator, class ForwardIterator>
  ForwardIterator
    reverse_copy(ExecutionPolicy&& exec,
                 BidirectionalIterator first, BidirectionalIterator last,
                 ForwardIterator result);
```

7 *Requires:* The ranges `[first, last)` and `[result, result + (last - first))` shall not overlap.

8 *Effects:* Copies the range `[first, last)` to the range `[result, result + (last - first))` such that for every non-negative integer `i < (last - first)` the following assignment takes place: `*(result + (last - first) - 1 - i) = *(first + i)`.

9 *Returns:* `result + (last - first)`.

10 *Complexity:* Exactly `last - first` assignments.

```
namespace ranges {
  template<BidirectionalIterator I, Sentinel<I> S, WeaklyIncrementable O>
    requires IndirectlyCopyable<I, O>
    tagged_pair<tag::in(I), tag::out(O)>
    constexpr reverse_copy_result<I, O>
      reverse_copy(I first, S last, O result);
```

```
  template<BidirectionalRange Rng, WeaklyIncrementable O>
    requires IndirectlyCopyable<iterator_t<Rng>, O>
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    constexpr reverse_copy_result<safe_iterator_t<Rng>, O>
      reverse_copy(Rng&& rng, O result);
}
```

11 *Effects:* Copies the range `[first, last)` to the range `[result, result + (last - first))` such that for every non-negative integer `i < (last - first)` the following assignment takes place: `*(result + (last - first) - 1 - i) = *(first + i)`.

12 *Requires:* The ranges `[first, last)` and `[result, result + (last - first))` shall not overlap.

13 *Returns:* `{last, result + (last - first)}`.

14 *Complexity:* Exactly `last - first` assignments.

30.6.11 Rotate [alg.rotate]

```
template<class ForwardIterator>
  ForwardIterator
    rotate(ForwardIterator first, ForwardIterator middle, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator
    rotate(ExecutionPolicy&& exec,
           ForwardIterator first, ForwardIterator middle, ForwardIterator last);
```

[Editor's note: This wording incorporates the PR for stl2#526).]

1 *Requires:* `[first, middle)` and `[middle, last)` shall be valid ranges. `ForwardIterator` shall satisfy the *Cpp17ValueSwappable* requirements ([swappable.requirements]). The type of `*first` shall satisfy the *Cpp17MoveConstructible* ([tab:moveconstructible]) and *Cpp17MoveAssignable* ([tab:moveassignable]) requirements.

2 *Effects:* For each non-negative integer `i < (last - first)`, places the element from the position `first + i` into position `first + (i + (last - middle)) % (last - first)`.

3 *Returns:* `first + (last - middle)`.

4 *Remarks:* This is a left rotate.

5 *Complexity:* At most `last - first` swaps.

```
namespace ranges {
  template<ForwardIterator I, Sentinel<I> S>
    requires Permutable<I>
    tagged_pair<tag::begin(I), tag::end(I)>
    constexpr subrange<I>
      rotate(I first, I middle, S last);
  template<ForwardRange Rng>
    requires Permutable<iterator_t<Rng>>
    tagged_pair<tag::begin(safe_iterator_t<Rng>),
                tag::end(safe_iterator_t<Rng>)>
    constexpr safe_subrange_t<Rng>
      rotate(Rng&& rng, iterator_t<Rng> middle);
}
```

6 *Effects:* For each non-negative integer `i < (last - first)`, places the element from the position `first + i` into position `first + (i + (last - middle)) % (last - first)`.

7 *Returns:* `{first + (last - middle), last}`.

8 *Remarks:* This is a left rotate.

9 *Requires:* `[first, middle)` and `[middle, last)` shall be valid ranges.

10 *Complexity:* At most `last - first` swaps.

```
template<class ForwardIterator, class OutputIterator>
  constexpr OutputIterator
    rotate_copy(ForwardIterator first, ForwardIterator middle, ForwardIterator last,
                OutputIterator result);
```

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2
    rotate_copy(ExecutionPolicy&& exec,
                ForwardIterator1 first, ForwardIterator1 middle, ForwardIterator1 last,
                ForwardIterator2 result);
```

11 *Requires:* The ranges `[first, last)` and `[result, result + (last - first))` shall not overlap.

12 *Effects:* Copies the range `[first, last)` to the range `[result, result + (last - first))` such that for each non-negative integer `i < (last - first)` the following assignment takes place: `*(result + i) = *(first + (i + (middle - first)) % (last - first))`.

13 *Returns:* `result + (last - first)`.

14 *Complexity:* Exactly `last - first` assignments.

```
namespace ranges {
  template<ForwardIterator I, Sentinel<I> S, WeaklyIncrementable O>
    requires IndirectlyCopyable<I, O>
    ~~tagged_pair<tag::in(I), tag::out(O)>~~
    constexpr rotate_copy_result<I, O>
      rotate_copy(I first, I middle, S last, O result);
  template<ForwardRange Rng, WeaklyIncrementable O>
    requires IndirectlyCopyable<iterator_t<Rng>, O>
    ~~tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>~~
    constexpr rotate_copy_result<safe_iterator_t<Rng>, O>
      rotate_copy(Rng&& rng, iterator_t<Rng> middle, O result);
}
```

15 *Effects:* Copies the range `[first, last)` to the range `[result, result + (last - first))` such that for each non-negative integer `i < (last - first)` the following assignment takes place: `*(result + i) = *(first + (i + (middle - first)) % (last - first))`.

16 *Returns:* `{last, result + (last - first)}`.

17 *Requires:* The ranges `[first, last)` and `[result, result + (last - first))` shall not overlap.

18 *Complexity:* Exactly `last - first` assignments.

30.6.12 Sample [alg.random.sample]

[...]

30.6.13 Shuffle [alg.random.shuffle]

```
template<class RandomAccessIterator, class UniformRandomBitGenerator>
  void shuffle(RandomAccessIterator first,
               RandomAccessIterator last,
               UniformRandomBitGenerator&& g);
```

1 *Requires:* `RandomAccessIterator` shall satisfy the *Cpp17ValueSwappable* requirements ([swappable.requirements]). The type `remove_reference_t<UniformRandomBitGenerator>` shall satisfy the requirements of a uniform random bit generator[rand.req.urng] type whose return type is convertible to `iterator_traits<RandomAccessIterator>::difference_type`.

2 *Effects:* Permutes the elements in the range `[first, last)` such that each possible permutation of those elements has equal probability of appearance.

3 *Complexity:* Exactly `(last - first) - 1` swaps.

4 *Remarks:* To the extent that the implementation of this function makes use of random numbers, the object `g` shall serve as the implementation's source of randomness.

```
namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Gen>
    requires Permutable<I> &&
      UniformRandom~~Number~~BitGenerator<remove_reference_t<Gen>> &&
      ConvertibleTo<~~result_of_t<Gen&()>~~invoke_result_t<Gen&>, iter_difference~~_type~~_t<I>>
    I shuffle(I first, S last, Gen&& g);
```

```
  template<RandomAccessRange Rng, class Gen>
    requires Permutable<I> &&
      UniformRandom~~Number~~BitGenerator<remove_reference_t<Gen>> &&
      ConvertibleTo<~~result_of_t<Gen&()>~~invoke_result_t<Gen&>, iter_difference~~_type~~_t<I>>
    safe_iterator_t<Rng>
      shuffle(Rng&& rng, Gen&& g);
}
```

5 *Effects:* Permutes the elements in the range `[first, last)` such that each possible permutation of those elements has equal probability of appearance.

6 *Complexity:* Exactly `(last - first) - 1` swaps.

7 *Returns:* `last`

8 *Remarks:* To the extent that the implementation of this function makes use of random numbers, the object `g` shall serve as the implementation's source of randomness.

30.7 Sorting and related operations [alg.sorting]

1 All the operations in 30.7 directly in namespace `std` have two versions: one that takes a function object of type `Compare` and one that uses an `operator<`.

2 `Compare` is a function object type[function.objects]. The return value of the function call operation applied to an object of type `Compare`, when contextually converted to `bool`[conv], yields `true` if the first argument of the call is less than the second, and `false` otherwise. `Compare comp` is used throughout for algorithms assuming an ordering relation. It is assumed that `comp` will not apply any non-constant function through the dereferenced iterator.

3 For all algorithms that take `Compare`, there is a version that uses `operator<` instead. That is, `comp(*i, *j) != false` defaults to `*i < *j != false`. For algorithms other than those described in 30.7.3, `comp` shall induce a strict weak ordering on the values.

[Editor's note: This specification of strict weak ordering may be redundant with the specification of strict weak ordering in [concepts].]

4 The term *strict* refers to the requirement of an irreflexive relation (`!comp(x, x)` for all `x`), and the term *weak* to requirements that are not as strong as those for a total ordering, but stronger than those for a partial ordering. If we define `equiv(a, b)` as `!comp(a, b) && !comp(b, a)`, then the requirements are that `comp` and `equiv` both be transitive relations:

(4.1) — `comp(a, b) && comp(b, c)` implies `comp(a, c)`

(4.2) — `equiv(a, b) && equiv(b, c)` implies `equiv(a, c)`

[*Note:* Under these conditions, it can be shown that

(4.3) — `equiv` is an equivalence relation

(4.4) — `comp` induces a well-defined relation on the equivalence classes determined by `equiv`

(4.5) — The induced relation is a strict total ordering.

— *end note*]

5 A sequence is *sorted with respect to a comparator* `comp` if for every iterator `i` pointing to the sequence and every non-negative integer `n` such that `i + n` is a valid iterator pointing to an element of the sequence, `comp(*(i + n), *i) == false`.

6 A sequence is *sorted with respect to a comparator and projection* `comp` and `proj` if for every iterator `i` pointing to the sequence and every non-negative integer `n` such that `i + n` is a valid iterator pointing to an element of the sequence, `invoke(comp, invoke(proj, *(i + n)), invoke(proj, *i)) == false`.

7 A sequence `[start, finish)` is *partitioned with respect to an expression* `f(e)` if there exists an integer `n` such that for all `0 <= i < (finish - start)`, `f(*(start + i))` is `true` if and only if `i < n`.

8 In the descriptions of the functions that deal with ordering relationships we frequently use a notion of equivalence to describe concepts such as stability. The equivalence to which we refer is not necessarily an `operator==`, but an equivalence relation induced by the strict weak ordering. That is, two elements `a` and `b` are considered equivalent if and only if `!(a < b) && !(b < a)`.

30.7.1 Sorting [alg.sort]

30.7.1.1 sort [sort]

```
template<class RandomAccessIterator>
  void sort(RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator>
  void sort(ExecutionPolicy&& exec,
            RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void sort(RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  void sort(ExecutionPolicy&& exec,
            RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);
```

1 *Requires:* `RandomAccessIterator` shall satisfy the *Cpp17ValueSwappable* requirements ([swappable.requirements]). The type of `*first` shall satisfy the *Cpp17MoveConstructible* ([tab:moveconstructible]) and *Cpp17MoveAssignable* ([tab:moveassignable]) requirements.

2 *Effects:* Sorts the elements in the range `[first, last)`.

3 *Complexity:* $\mathcal{O}(N \log N)$ comparisons, where $N =$ `last - first`.

```
namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    constexpr I
      sort(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    constexpr safe_iterator_t<Rng>
      sort(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

4 *Effects:* Sorts the elements in the range `[first, last)`.

5 *Returns:* `last`.

6 *Complexity:* $\mathcal{O}(N \log(N))$ (where N `== last - first`) comparisons, and twice as many applications of the projection.

30.7.1.2 stable_sort [stable.sort]

```
template<class RandomAccessIterator>
  void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator>
  void stable_sort(ExecutionPolicy&& exec,
                   RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  void stable_sort(ExecutionPolicy&& exec,
                   RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);
```

1 *Requires:* `RandomAccessIterator` shall satisfy the *Cpp17ValueSwappable* requirements ([swappable.requirements]). The type of `*first` shall satisfy the *Cpp17MoveConstructible* ([tab:moveconstructible]) and *Cpp17MoveAssignable* ([tab:moveassignable]) requirements.

2 *Effects:* Sorts the elements in the range `[first, last)`.

3 *Complexity:* At most $N \log^2(N)$ comparisons, where $N =$ `last - first`, but only $N \log N$ comparisons if there is enough extra memory.

4 *Remarks:* Stable[algorithm.stable].

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

5 *Effects:* Sorts the elements in the range `[first, last)`.

6 *Returns:* `last`.

7 *Complexity:* Let N `== last - first`. If enough extra memory is available, $N \log(N)$ comparisons. Otherwise, at most $N \log^2(N)$ comparisons. In either case, twice as many applications of the projection as the number of comparisons.

8 *Remarks:* Stable ([algorithm.stable]).

30.7.1.3 `partial_sort` [partial.sort]

```
template<class RandomAccessIterator>
  void partial_sort(RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator>
  void partial_sort(ExecutionPolicy&& exec,
                    RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last);

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

1 *Requires:* `RandomAccessIterator` shall satisfy the *Cpp17ValueSwappable* requirements ([swappable.requirements]). The type of `*first` shall satisfy the *Cpp17MoveConstructible* ([tab:moveconstructible]) and *Cpp17MoveAssignable* ([tab:moveassignable]) requirements.

2 *Effects:* Places the first `middle - first` sorted elements from the range `[first, last)` into the range `[first, middle)`. The rest of the elements in the range `[middle, last)` are placed in an unspecified order.

3 *Complexity:* Approximately `(last - first) * log(middle - first)` comparisons.

```
namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    constexpr I
      partial_sort(I first, I middle, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    constexpr safe_iterator_t<Rng>
      partial_sort(Rng&& rng, iterator_t<Rng> middle, Comp comp = Comp{},
```

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

4 *Effects:* Places the first `middle - first` sorted elements from the range `[first, last)` into the range `[first, middle)`. The rest of the elements in the range `[middle, last)` are placed in an unspecified order.

5 *Returns:* `last`.

6 *Complexity:* It takes approximately `(last - first) * log(middle - first)` comparisons, and exactly twice as many applications of the projection.

30.7.1.4 `partial_sort_copy` [partial.sort.copy]

```
template<class InputIterator, class RandomAccessIterator>
  RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last);
template<class ExecutionPolicy, class ForwardIterator, class RandomAccessIterator>
  RandomAccessIterator
    partial_sort_copy(ExecutionPolicy&& exec,
                      ForwardIterator first, ForwardIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last);

template<class InputIterator, class RandomAccessIterator,
         class Compare>
  RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last,
                      Compare comp);
template<class ExecutionPolicy, class ForwardIterator, class RandomAccessIterator,
         class Compare>
  RandomAccessIterator
    partial_sort_copy(ExecutionPolicy&& exec,
                      ForwardIterator first, ForwardIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last,
                      Compare comp);
```

1 *Requires:* `RandomAccessIterator` shall satisfy the *Cpp17ValueSwappable* requirements ([swappable.requirements]). The type of `*result_first` shall satisfy the *Cpp17MoveConstructible* ([tab:moveconstructible]) and *Cpp17MoveAssignable* ([tab:moveassignable]) requirements.

2 *Effects:* Places the first `min(last - first, result_last - result_first)` sorted elements into the range `[result_first, result_first + min(last - first, result_last - result_first))`.

3 *Returns:* The smaller of: `result_last` or `result_first + (last - first)`.

4 *Complexity:* Approximately `(last - first) * log(min(last - first, result_last - result_first))` comparisons.

```
namespace ranges {
  template<InputIterator I1, Sentinel<I1> S1, RandomAccessIterator I2, Sentinel<I2> S2,
      class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyCopyable<I1, I2> && Sortable<I2, Comp, Proj2> &&
        IndirectStrictWeakOrder<Comp, projected<I1, Proj1>, projected<I2, Proj2>>
    constexpr I2
      partial_sort_copy(I1 first, S1 last, I2 result_first, S2 result_last,
                        Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<InputRange Rng1, RandomAccessRange Rng2, class Comp = ranges::less<>,
      class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyCopyable<iterator_t<Rng1>, iterator_t<Rng2>> &&
        Sortable<iterator_t<Rng2>, Comp, Proj2> &&
        IndirectStrictWeakOrder<Comp, projected<iterator_t<Rng1>, Proj1>,
```

```
        projected<iterator_t<Rng2>, Proj2>>
    constexpr safe_iterator_t<Rng2>
      partial_sort_copy(Rng1&& rng, Rng2&& result_rng, Comp comp = Comp{},
                        Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

5 *Effects:* Places the first `min(last - first, result_last - result_first)` sorted elements into the range `[result_first, result_first + min(last - first, result_last - result_first))`.

6 *Returns:* The smaller of: `result_last` or `result_first + (last - first)`.

7 *Complexity:* Approximately

```
(last - first) * log(min(last - first, result_last - result_first))
```

comparisons, and exactly twice as many applications of the projection.

30.7.1.5 `is_sorted` [is.sorted]

```
template<class ForwardIterator>
  constexpr bool is_sorted(ForwardIterator first, ForwardIterator last);
```

1 *Returns:* `is_sorted_until(first, last) == last`.

```
template<class ExecutionPolicy, class ForwardIterator>
  bool is_sorted(ExecutionPolicy&& exec,
                 ForwardIterator first, ForwardIterator last);
```

2 *Returns:* `is_sorted_until(std::forward<ExecutionPolicy>(exec), first, last) == last`.

```
template<class ForwardIterator, class Compare>
  constexpr bool is_sorted(ForwardIterator first, ForwardIterator last,
                           Compare comp);
```

3 *Returns:* `is_sorted_until(first, last, comp) == last`.

```
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  bool is_sorted(ExecutionPolicy&& exec,
                 ForwardIterator first, ForwardIterator last,
                 Compare comp);
```

4 *Returns:*

```
is_sorted_until(std::forward<ExecutionPolicy>(exec), first, last, comp) == last
```

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

5 *Returns:* `is_sorted_until(first, last, comp, proj) == last`

```
template<class ForwardIterator>
  constexpr ForwardIterator
    is_sorted_until(ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator
    is_sorted_until(ExecutionPolicy&& exec,
                    ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
  constexpr ForwardIterator
    is_sorted_until(ForwardIterator first, ForwardIterator last,
                    Compare comp);
```

```
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  ForwardIterator
    is_sorted_until(ExecutionPolicy&& exec,
                    ForwardIterator first, ForwardIterator last,
                    Compare comp);
```

6 *Returns:* If `(last - first) < 2`, returns `last`. Otherwise, returns the last iterator `i` in `[first, last]` for which the range `[first, i)` is sorted.

7 *Complexity:* Linear.

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

8 *Returns:* If `distance(first, last) < 2`, returns `last`. Otherwise, returns the last iterator `i` in `[first, last]` for which the range `[first, i)` is sorted.

9 *Complexity:* Linear.

30.7.2 Nth element [alg.nth.element]

```
template<class RandomAccessIterator>
  void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator>
  void nth_element(ExecutionPolicy&& exec,
                   RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last,  Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  void nth_element(ExecutionPolicy&& exec,
                   RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last, Compare comp);
```

1 *Requires:* `RandomAccessIterator` shall satisfy the *Cpp17ValueSwappable* requirements ([swappable.requirements]). The type of `*first` shall satisfy the *Cpp17MoveConstructible* ([tab:moveconstructible]) and *Cpp17MoveAssignable* ([tab:moveassignable]) requirements.

2 *Effects:* After `nth_element` the element in the position pointed to by `nth` is the element that would be in that position if the whole range were sorted, unless `nth == last`. Also for every iterator `i` in the range `[first, nth)` and every iterator `j` in the range `[nth, last)` it holds that: `!(*j < *i)` or `comp(*j, *i) == false`.

3 *Complexity:* For the overloads with no `ExecutionPolicy`, linear on average. For the overloads with an `ExecutionPolicy`, $\mathcal{O}(N)$ applications of the predicate, and $\mathcal{O}(N \log N)$ swaps, where $N =$ `last - first`.

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

```
}
```

4 After `nth_element` the element in the position pointed to by `nth` is the element that would be in that position if the whole range were sorted, unless `nth == last`. Also for every iterator `i` in the range `[first, nth)` and every iterator `j` in the range `[nth, last)` it holds that: `invoke(comp, invoke(proj, *j), invoke(proj, *i)) == false`.

5 *Returns:* `last`.

6 *Complexity:* Linear on average.

30.7.3 Binary search [alg.binary.search]

1 All of the algorithms in this subclause are versions of binary search and assume that the sequence being searched is partitioned with respect to an expression formed by binding the search key to an argument of the implied or explicit comparison function (and possibly projection). They work on non-random access iterators minimizing the number of comparisons, which will be logarithmic for all types of iterators. They are especially appropriate for random access iterators, because these algorithms do a logarithmic number of steps through the data structure. For non-random access iterators they execute a linear number of steps.

30.7.3.1 `lower_bound` [lower.bound]

```
template<class ForwardIterator, class T>
  constexpr ForwardIterator
    lower_bound(ForwardIterator first, ForwardIterator last,
                const T& value);

template<class ForwardIterator, class T, class Compare>
  constexpr ForwardIterator
    lower_bound(ForwardIterator first, ForwardIterator last,
                const T& value, Compare comp);
```

1 *Requires:* The elements `e` of `[first, last)` shall be partitioned with respect to the expression `e < value` or `comp(e, value)`.

2 *Returns:* The furthermost iterator `i` in the range `[first, last]` such that for every iterator `j` in the range `[first, i)` the following corresponding conditions hold: `*j < value` or `comp(*j, value) != false`.

3 *Complexity:* At most $\log_2($`last - first`$) + \mathcal{O}(1)$ comparisons.

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

4 *Requires:* The elements `e` of `[first, last)` shall be partitioned with respect to the expression `invoke(comp, invoke(proj, e), value)`.

5 *Returns:* The furthermost iterator `i` in the range `[first, last]` such that for every iterator `j` in the range `[first, i)` the following corresponding condition holds: `invoke(comp, invoke(proj, *j), value) != false`.

6 *Complexity:* At most $\log_2($`last - first`$) + \mathcal{O}(1)$ applications of the comparison function and projection.

30.7.3.2 `upper_bound` [upper.bound]

```
template<class ForwardIterator, class T>
  constexpr ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last,
                const T& value);
```

```
template<class ForwardIterator, class T, class Compare>
  constexpr ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last,
                const T& value, Compare comp);
```

1 *Requires:* The elements `e` of `[first, last)` shall be partitioned with respect to the expression `!(value < e)` or `!comp(value, e)`.

2 *Returns:* The furthermost iterator `i` in the range `[first, last]` such that for every iterator `j` in the range `[first, i)` the following corresponding conditions hold: `!(value < *j)` or `comp(value, *j) == false`.

3 *Complexity:* At most $\log_2($`last - first`$) + \mathcal{O}(1)$ comparisons.

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

4 *Requires:* The elements `e` of `[first, last)` shall be partitioned with respect to the expression `!invoke(comp, value, invoke(proj, e))`.

5 *Returns:* The furthermost iterator `i` in the range `[first, last]` such that for every iterator `j` in the range `[first, i)` the following corresponding condition holds: `invoke(comp, value, invoke(proj, *j)) == false`.

6 *Complexity:* At most $\log_2($`last - first`$)+\mathcal{O}(1)$ applications of the comparison function and projection.

30.7.3.3 `equal_range` [equal.range]

```
template<class ForwardIterator, class T>
  constexpr pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first,
                ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
  constexpr pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first,
                ForwardIterator last, const T& value,
                Compare comp);
```

[Editor's note: This wording incorporates the PR for stl2#526).]

1 *Requires:* The elements `e` of `[first, last)` shall be partitioned with respect to the expressions `e < value` and `!(value < e)` or `comp(e, value)` and `!comp(value, e)`. Also, for all elements `e` of `[first, last)`, `e < value` shall imply `!(value < e)` or `comp(e, value)` shall imply `!comp(value, e)`.

2 *Returns:*

```
make_pair(lower_bound(first, last, value),
          upper_bound(first, last, value))
```

or

```
make_pair(lower_bound(first, last, value, comp),
          upper_bound(first, last, value, comp))
```

3 *Complexity:* At most $2 * \log_2($`last - first`$) + \mathcal{O}(1)$ comparisons.

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

```
  template<ForwardRange Rng, class T, class Proj = identity,
      IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    tagged_pair<tag::begin(safe_iterator_t<Rng>),
                tag::end(safe_iterator_t<Rng>)>
    constexpr safe_subrange_t<Rng>
      equal_range(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
}
```

4 *Requires:* The elements `e` of `[first, last)` shall be partitioned with respect to the expressions `invoke(comp, invoke(proj, e), value)` and `!invoke(comp, value, invoke(proj, e))`. Also, for all elements `e` of `[first, last)`, `invoke(comp, invoke(proj, e), value)` shall imply `!invoke(comp, value, invoke(proj, e))`.

5 *Returns:*

```
{lower_bound(first, last, value, comp, proj),
 upper_bound(first, last, value, comp, proj)}
```

6 *Complexity:* At most $2 * \log_2($`last - first`$) + \mathcal{O}(1)$ applications of the comparison function and projection.

30.7.3.4 `binary_search` [binary.search]

```
template<class ForwardIterator, class T>
  constexpr bool
    binary_search(ForwardIterator first, ForwardIterator last,
                  const T& value);

template<class ForwardIterator, class T, class Compare>
  constexpr bool
    binary_search(ForwardIterator first, ForwardIterator last,
                  const T& value, Compare comp);
```

1 *Requires:* The elements `e` of `[first, last)` shall be partitioned with respect to the expressions `e < value` and `!(value < e)` or `comp(e, value)` and `!comp(value, e)`. Also, for all elements `e` of `[first, last)`, `e < value` shall imply `!(value < e)` or `comp(e, value)` shall imply `!comp(value, e)`.

2 *Returns:* `true` if there is an iterator `i` in the range `[first, last)` that satisfies the corresponding conditions: `!(*i < value) && !(value < *i)` or `comp(*i, value) == false && comp(value, *i) == false`.

3 *Complexity:* At most $\log_2($`last - first`$) + \mathcal{O}(1)$ comparisons.

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

4 *Requires:* The elements `e` of `[first, last)` are partitioned with respect to the expressions `invoke(comp, invoke(proj, e), value)` and `!invoke(comp, value, invoke(proj, e))`. Also, for all elements `e` of `[first, last)`, `invoke(comp, invoke(proj, e), value)` shall imply `!invoke(comp, value, invoke(proj, e))`.

5 *Returns:* `true` if there is an iterator `i` in the range `[first, last)` that satisfies the corresponding conditions: `invoke(comp, invoke(proj, *i), value) == false && invoke(comp, value, invoke(proj, *i)) == false`.

6 *Complexity:* At most $\log_2($`last - first`$) + \mathcal{O}(1)$ applications of the comparison function and projection.

30.7.4 Partitions [alg.partitions]

```
template<class InputIterator, class Predicate>
  constexpr bool is_partitioned(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  bool is_partitioned(ExecutionPolicy&& exec,
                      ForwardIterator first, ForwardIterator last, Predicate pred);
```

1 *Requires:* For the overload with no `ExecutionPolicy`, `InputIterator`'s value type shall be convertible to `Predicate`'s argument type. For the overload with an `ExecutionPolicy`, `ForwardIterator`'s value type shall be convertible to `Predicate`'s argument type.

2 *Returns:* `true` if `[first, last)` is empty or if the elements `e` of `[first, last)` are partitioned with respect to the expression `pred(e)`.

3 *Complexity:* Linear. At most `last - first` applications of `pred`.

```
namespace ranges {
  template<InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr bool is_partitioned(I first, S last, Pred pred, Proj proj = Proj{});
  template<InputRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    constexpr bool is_partitioned(Rng&& rng, Pred pred, Proj proj = Proj{});
}
```

4 *Returns:* `true` if `[first, last)` is empty or if `[first, last)` is partitioned by `pred` and `proj`, i.e. if all iterators `i` for which `invoke(pred, invoke(proj, *i)) != false` come before those that do not, for every `i` in `[first, last)`.

5 *Complexity:* Linear. At most `last - first` applications of `pred` and `proj`.

```
template<class ForwardIterator, class Predicate>
  ForwardIterator
    partition(ForwardIterator first, ForwardIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator
    partition(ExecutionPolicy&& exec,
              ForwardIterator first, ForwardIterator last, Predicate pred);
```

6 *Requires:* `ForwardIterator` shall satisfy the *Cpp17ValueSwappable* requirements ([swappable.requirements]).

7 *Effects:* Places all the elements in the range `[first, last)` that satisfy `pred` before all the elements that do not satisfy it.

8 *Returns:* An iterator `i` such that for every iterator `j` in the range `[first, i) pred(*j) != false`, and for every iterator `k` in the range `[i, last)`, `pred(*k) == false`.

9 *Complexity:* Let $N =$ `last - first`:

(9.1) — For the overload with no `ExecutionPolicy`, exactly N applications of the predicate. At most $N/2$ swaps if `ForwardIterator` meets the `BidirectionalIterator` requirements and at most N swaps otherwise.

(9.2) — For the overload with an `ExecutionPolicy`, $\mathcal{O}(N \log N)$ swaps and $\mathcal{O}(N)$ applications of the predicate.

```
namespace ranges {
  template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires Permutable<I>
    constexpr I
      partition(I first, S last, Pred pred, Proj proj = Proj{});
  template<ForwardRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    requires Permutable<iterator_t<Rng>>
    constexpr safe_iterator_t<Rng>
      partition(Rng&& rng, Pred pred, Proj proj = Proj{});
```

```
}
```

10 *Effects:* Permutes the elements in the range `[first, last)` such that there exists an iterator `i` such that for every iterator `j` in the range `[first, i)` `invoke(pred, invoke(proj, *j)) != false`, and for every iterator `k` in the range `[i, last)`, `invoke(pred, invoke(proj, *k)) == false`.

11 *Returns:* An iterator `i` such that for every iterator `j` in the range `[first, i)` `invoke(pred, invoke(proj, *j)) != false`, and for every iterator `k` in the range `[i, last)`, `invoke(pred, invoke(proj, *k)) == false`.

12 *Complexity:* If I meets the requirements for a BidirectionalIterator, at most `(last - first) / 2` swaps; otherwise at most `last - first` swaps. Exactly `last - first` applications of the predicate and projection.

```
template<class BidirectionalIterator, class Predicate>
  BidirectionalIterator
    stable_partition(BidirectionalIterator first, BidirectionalIterator last, Predicate pred);
template<class ExecutionPolicy, class BidirectionalIterator, class Predicate>
  BidirectionalIterator
    stable_partition(ExecutionPolicy&& exec,
                     BidirectionalIterator first, BidirectionalIterator last, Predicate pred);
```

13 *Requires:* `BidirectionalIterator` shall satisfy the *Cpp17ValueSwappable* requirements ([swappable.requirements]). The type of `*first` shall satisfy the *Cpp17MoveConstructible* ([tab:moveconstructible]) and *Cpp17MoveAssignable* ([tab:moveassignable]) requirements.

14 *Effects:* Places all the elements in the range `[first, last)` that satisfy `pred` before all the elements that do not satisfy it.

15 *Returns:* An iterator `i` such that for every iterator `j` in the range `[first, i)`, `pred(*j) != false`, and for every iterator `k` in the range `[i, last)`, `pred(*k) == false`. The relative order of the elements in both groups is preserved.

16 *Complexity:* Let $N =$ `last - first`:

(16.1) — For the overload with no `ExecutionPolicy`, at most $N \log N$ swaps, but only $\mathscr{O}(N)$ swaps if there is enough extra memory. Exactly N applications of the predicate.

(16.2) — For the overload with an `ExecutionPolicy`, $\mathscr{O}(N \log N)$ swaps and $\mathscr{O}(N)$ applications of the predicate.

```
namespace ranges {
  template<BidirectionalIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires Permutable<I>
    I stable_partition(I first, S last, Pred pred, Proj proj = Proj{});
  template<BidirectionalRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    requires Permutable<iterator_t<Rng>>
    safe_iterator_t<Rng> stable_partition(Rng&& rng, Pred pred, Proj proj = Proj{});
}
```

17 *Effects:* Permutes the elements in the range `[first, last)` such that there exists an iterator `i` such that for every iterator `j` in the range `[first, i)` `invoke(pred, invoke(proj, *j)) != false`, and for every iterator `k` in the range `[i, last)`, `invoke(pred, invoke(proj, *k)) == false`.

18 *Returns:* An iterator `i` such that for every iterator `j` in the range `[first, i)`, `invoke(pred, invoke(proj, *j)) != false`, and for every iterator `k` in the range `[i, last)`, `invoke(pred, invoke(proj, *k)) == false`. The relative order of the elements in both groups is preserved.

19 *Complexity:* At most `(last - first) * log(last - first)` swaps, but only linear number of swaps if there is enough extra memory. Exactly `last - first` applications of the predicate and projection.

```
template<class InputIterator, class OutputIterator1,
         class OutputIterator2, class Predicate>
  constexpr pair<OutputIterator1, OutputIterator2>
    partition_copy(InputIterator first, InputIterator last,
                   OutputIterator1 out_true, OutputIterator2 out_false, Predicate pred);
```

```
template<class ExecutionPolicy, class ForwardIterator, class ForwardIterator1,
         class ForwardIterator2, class Predicate>
  pair<ForwardIterator1, ForwardIterator2>
    partition_copy(ExecutionPolicy&& exec,
                   ForwardIterator first, ForwardIterator last,
                   ForwardIterator1 out_true, ForwardIterator2 out_false, Predicate pred);
```

20 *Requires:*

(20.1) — For the overload with no `ExecutionPolicy`, `InputIterator`'s value type shall be *Cpp17CopyAssignable* ([tab:copyassignable]), and shall be writable (28.3.1) to the `out_true` and `out_false` `OutputIterators`, and shall be convertible to `Predicate`'s argument type.

(20.2) — For the overload with an `ExecutionPolicy`, `ForwardIterator`'s value type shall be `CopyAssignable`, and shall be writable to the `out_true` and `out_false` `ForwardIterators`, and shall be convertible to `Predicate`'s argument type. [*Note:* There may be a performance cost if `ForwardIterator`'s value type is not *Cpp17CopyConstructible*. — *end note*]

(20.3) — For both overloads, the input range shall not overlap with either of the output ranges.

21 *Effects:* For each iterator `i` in `[first, last)`, copies `*i` to the output range beginning with `out_true` if `pred(*i)` is `true`, or to the output range beginning with `out_false` otherwise.

22 *Returns:* A pair `p` such that `p.first` is the end of the output range beginning at `out_true` and `p.second` is the end of the output range beginning at `out_false`.

23 *Complexity:* Exactly `last - first` applications of `pred`.

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

24 *Requires:* The input range shall not overlap with either of the output ranges.

25 *Effects:* For each iterator `i` in `[first, last)`, copies `*i` to the output range beginning with `out_true` if `invoke(pred, invoke(proj, *i))` is `true`, or to the output range beginning with `out_false` otherwise.

26 *Returns:* ~~A tuple `p` such that `get<0>(p)` is `last`, `get<1>(p)` is the end of the output range beginning at `out_true`, and `get<2>(p)` is the end of the output range beginning at `out_false`~~ `{ last, o1, o2 }`, where `o1` is the end of the output range beginning at `out_true` and `o2` is the end of the output range beginning at `out_false`.

27 *Complexity:* Exactly `last - first` applications of `pred` and `proj`.

```
template<class ForwardIterator, class Predicate>
  constexpr ForwardIterator
    partition_point(ForwardIterator first, ForwardIterator last, Predicate pred);
```

28 *Requires:* `ForwardIterator`'s value type shall be convertible to `Predicate`'s argument type. The elements `e` of `[first, last)` shall be partitioned with respect to the expression `pred(e)`.

29 *Returns:* An iterator `mid` such that `all_of(first, mid, pred)` and `none_of(mid, last, pred)` are both `true`.

30 *Complexity:* $\mathcal{O}(\log(\texttt{last - first}))$ applications of `pred`.

```
namespace ranges {
  template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr I partition_point(I first, S last, Pred pred, Proj proj = Proj{});
  template<ForwardRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    constexpr safe_iterator_t<Rng>
      partition_point(Rng&& rng, Pred pred, Proj proj = Proj{});
}
```

31 *Requires:* `[first, last)` shall be partitioned by `pred` and `proj`, i.e. there shall be an iterator `mid` such that `all_of(first, mid, pred, proj)` and `none_of(mid, last, pred, proj)` are both true.

32 *Returns:* An iterator `mid` such that `all_of(first, mid, pred, proj)` and `none_of(mid, last, pred, proj)` are both true.

33 *Complexity:* $\mathcal{O}(\log($`last - first`$))$ applications of `pred` and `proj`.

30.7.5 Merge [alg.merge]

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

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

1 *Requires:* The ranges `[first1, last1)` and `[first2, last2)` shall be sorted with respect to `operator<` or `comp`. The resulting range shall not overlap with either of the original ranges.

2 *Effects:* Copies all the elements of the two ranges `[first1, last1)` and `[first2, last2)` into the range `[result, result_last)`, where `result_last` is `result + (last1 - first1) + (last2 - first2)`, such that the resulting range satisfies `is_sorted(result, result_last)` or `is_sorted(result, result_last, comp)`, respectively.

3 *Returns:* `result + (last1 - first1) + (last2 - first2)`.

4 *Complexity:* Let $N =$ `(last1 - first1) + (last2 - first2)`:

(4.1) — For the overloads with no `ExecutionPolicy`, at most $N - 1$ comparisons.

(4.2) — For the overloads with an `ExecutionPolicy`, $\mathcal{O}(N)$ comparisons.

5 *Remarks:* Stable[algorithm.stable].

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

6 *Effects:* Copies all the elements of the two ranges `[first1, last1)` and `[first2, last2)` into the range `[result, result_last)`, where `result_last` is `result + (last1 - first1) + (last2 - first2)`. If an element `a` precedes `b` in an input range, `a` is copied into the output range before `b`. If `e1` is an element of `[first1, last1)` and `e2` of `[first2, last2)`, `e2` is copied into the output range before `e1` if and only if `bool(invoke(comp, invoke(proj2, e2), invoke(proj1, e1)))` is `true`.

7 *Requires:* The ranges `[first1, last1)` and `[first2, last2)` shall be sorted with respect to `comp`, `proj1`, and `proj2`. The resulting range shall not overlap with either of the original ranges.

8 *Returns:* ~~`make_tagged_tuple<tag::in1, tag::in2, tag::out>(`~~`{last1, last2, result_last}`~~`)`~~.

9 *Complexity:* At most `(last1 - first1) + (last2 - first2) - 1` applications of the comparison function and each projection.

10 *Remarks:* Stable ([algorithm.stable]).

```
template<class BidirectionalIterator>
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last);
template<class ExecutionPolicy, class BidirectionalIterator>
  void inplace_merge(ExecutionPolicy&& exec,
                     BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last, Compare comp);
template<class ExecutionPolicy, class BidirectionalIterator, class Compare>
  void inplace_merge(ExecutionPolicy&& exec,
                     BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last, Compare comp);
```

11 *Requires:* The ranges `[first, middle)` and `[middle, last)` shall be sorted with respect to `operator<` or `comp`. `BidirectionalIterator` shall satisfy the *Cpp17ValueSwappable* requirements ([swappable.requirements]). The type of `*first` shall satisfy the *Cpp17MoveConstructible* ([tab:moveconstructible]) and *Cpp17MoveAssignable* ([tab:moveassignable]) requirements.

12 *Effects:* Merges two sorted consecutive ranges `[first, middle)` and `[middle, last)`, putting the result of the merge into the range `[first, last)`. The resulting range will be in non-decreasing order; that is, for every iterator `i` in `[first, last)` other than `first`, the condition `*i < *(i - 1)` or, respectively, `comp(*i, *(i - 1))` will be `false`.

13 *Complexity:* Let $N =$ `last - first`:

(13.1) — For the overloads with no `ExecutionPolicy`, if enough additional memory is available, exactly $N - 1$ comparisons.

(13.2) — For the overloads with no `ExecutionPolicy` if no additional memory is available, $\mathcal{O}(N \log N)$ comparisons.

(13.3) — For the overloads with an `ExecutionPolicy`, $\mathcal{O}(N \log N)$ comparisons.

14 *Remarks:* Stable[algorithm.stable].

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

15 *Effects:* Merges two sorted consecutive ranges `[first, middle)` and `[middle, last)`, putting the result of the merge into the range `[first, last)`. The resulting range will be in non-decreasing order; that is, for every iterator `i` in `[first, last)` other than `first`, the condition `invoke(comp, invoke(proj, *i), invoke(proj, *(i - 1)))` will be false.

16 *Requires:* The ranges `[first, middle)` and `[middle, last)` shall be sorted with respect to `comp` and `proj`.

17 *Returns:* `last`

18 *Complexity:* When enough additional memory is available, `(last - first) - 1` applications of the comparison function and projection. If no additional memory is available, an algorithm with complexity $N \log(N)$ (where `N` is equal to `last - first`) may be used.

19 *Remarks:* Stable ([algorithm.stable]).

30.7.6 Set operations on sorted structures [alg.set.operations]

1 This subclause defines all the basic set operations on sorted structures. They also work with `multiset`s ([multiset]) containing multiple copies of equivalent elements. The semantics of the set operations are generalized to `multiset`s in a standard way by defining `set_union()` to contain the maximum number of occurrences of every element, `set_intersection()` to contain the minimum, and so on.

30.7.6.1 `includes` [includes]

```
template<class InputIterator1, class InputIterator2>
  constexpr bool includes(InputIterator1 first1, InputIterator1 last1,
                          InputIterator2 first2, InputIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  bool includes(ExecutionPolicy&& exec,
                ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2, ForwardIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
  constexpr bool includes(InputIterator1 first1, InputIterator1 last1,
                          InputIterator2 first2, InputIterator2 last2,
                          Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class Compare>
  bool includes(ExecutionPolicy&& exec,
                ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2, ForwardIterator2 last2,
                Compare comp);
```

1 *Returns:* `true` if `[first2, last2)` is empty or if every element in the range `[first2, last2)` is contained in the range `[first1, last1)`. Returns `false` otherwise.

2 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

```
namespace ranges {
  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      class Proj1 = identity, class Proj2 = identity,
      IndirectStrictWeakOrder<projected<I1, Proj1>, projected<I2, Proj2>> Comp = ranges::less<>>
    constexpr bool includes(I1 first1, S1 last1, I2 first2, S2 last2, Comp comp = Comp{},
                            Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<InputRange Rng1, InputRange Rng2, class Proj1 = identity,
      class Proj2 = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng1>, Proj1>,
        projected<iterator_t<Rng2>, Proj2>> Comp = ranges::less<>>
    constexpr bool includes(Rng1&& rng1, Rng2&& rng2, Comp comp = Comp{},
                            Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

3 *Returns:* `true` if `[first2, last2)` is empty or if every element in the range `[first2, last2)` is contained in the range `[first1, last1)`. Returns `false` otherwise.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` applications of the comparison function and projections.

30.7.6.2 `set_union` [set.union]

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

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

1 *Requires:* The resulting range shall not overlap with either of the original ranges.

2 *Effects:* Constructs a sorted union of the elements from the two ranges; that is, the set of elements that are present in one or both of the ranges.

3 *Returns:* The end of the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

5 *Remarks:* If `[first1, last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, then all m elements from the first range shall be copied to the output range, in order, and then $\max(n - m, 0)$ elements from the second range shall be copied to the output range, in order.

```
namespace ranges {
  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      WeaklyIncrementable O, class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
```

```
    tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    constexpr set_union_result<I1, I2, O>
      set_union(I1 first1, S1 last1, I2 first2, S2 last2, O result, Comp comp = Comp{},
                Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
      class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
    tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
                 tag::in2(safe_iterator_t<Rng2>),
                 tag::out(O)>
    constexpr set_union_result<safe_iterator_t<Rng1>, safe_iterator_t<Rng2>, O>
      set_union(Rng1&& rng1, Rng2&& rng2, O result, Comp comp = Comp{},
                Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

6 *Effects:* Constructs a sorted union of the elements from the two ranges; that is, the set of elements that are present in one or both of the ranges.

7 *Requires:* The resulting range shall not overlap with either of the original ranges.

8 *Returns:* ~~`make_tagged_tuple<tag::in1, tag::in2, tag::out>(`~~`{last1, last2, result + `n`}`~~`)`~~, where n is the number of elements in the constructed range.

9 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` applications of the comparison function and projections.

10 *Remarks:* If `[first1, last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, then all m elements from the first range shall be copied to the output range, in order, and then $\max(n - m, 0)$ elements from the second range shall be copied to the output range, in order.

30.7.6.3 `set_intersection` [set.intersection]

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
  constexpr OutputIterator
    set_intersection(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  ForwardIterator
    set_intersection(ExecutionPolicy&& exec,
                     ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2, ForwardIterator2 last2,
                     ForwardIterator result);

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

1 *Requires:* The resulting range shall not overlap with either of the original ranges.

2 *Effects:* Constructs a sorted intersection of the elements from the two ranges; that is, the set of elements that are present in both of the ranges.

3 *Returns:* The end of the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

5 *Remarks:* If `[first1, last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, the first $\min(m, n)$ elements shall be copied from the first range to the output range, in order.

```
namespace ranges {
  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      WeaklyIncrementable O, class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
    constexpr O set_intersection(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                                 Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
      class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
    constexpr O set_intersection(Rng1&& rng1, Rng2&& rng2, O result,
                                 Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

6 *Effects:* Constructs a sorted intersection of the elements from the two ranges; that is, the set of elements that are present in both of the ranges.

7 *Requires:* The resulting range shall not overlap with either of the original ranges.

8 *Returns:* The end of the constructed range.

9 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` applications of the comparison function and projections.

10 *Remarks:* If `[first1, last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, the first $\min(m, n)$ elements shall be copied from the first range to the output range, in order.

30.7.6.4 `set_difference` [set.difference]

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
  constexpr OutputIterator
    set_difference(InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, InputIterator2 last2,
                   OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  ForwardIterator
    set_difference(ExecutionPolicy&& exec,
                   ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2, ForwardIterator2 last2,
                   ForwardIterator result);

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

1 *Requires:* The resulting range shall not overlap with either of the original ranges.

2 *Effects:* Copies the elements of the range `[first1, last1)` which are not present in the range `[first2, last2)` to the range beginning at `result`. The elements in the constructed range are sorted.

3 *Returns:* The end of the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

5 *Remarks:* If `[first1, last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, the last $\max(m - n, 0)$ elements from `[first1, last1)` shall be copied to the output range.

```
namespace ranges {
  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      WeaklyIncrementable O, class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
    ~~tagged_pair<tag::in1(I1), tag::out(O)>~~
    constexpr set_difference_result<I1, O>
      set_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                     Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
      class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
    ~~tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::out(O)>~~
    constexpr set_difference_result<safe_iterator_t<Rng1>, O>
      set_difference(Rng1&& rng1, Rng2&& rng2, O result,
                     Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

6 *Effects:* Copies the elements of the range `[first1, last1)` which are not present in the range `[first2, last2)` to the range beginning at `result`. The elements in the constructed range are sorted.

7 *Requires:* The resulting range shall not overlap with either of the original ranges.

8 *Returns:* `{last1, result +` n`}`, where n is the number of elements in the constructed range.

9 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` applications of the comparison function and projections.

10 *Remarks:* If `[first1, last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, the last $\max(m - n, 0)$ elements from `[first1, last1)` shall be copied to the output range.

30.7.6.5 `set_symmetric_difference` [set.symmetric.difference]

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
  constexpr OutputIterator
    set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  ForwardIterator
    set_symmetric_difference(ExecutionPolicy&& exec,
                             ForwardIterator1 first1, ForwardIterator1 last1,
                             ForwardIterator2 first2, ForwardIterator2 last2,
                             ForwardIterator result);

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

```
                    ForwardIterator result, Compare comp);
```

1 *Requires:* The resulting range shall not overlap with either of the original ranges.

2 *Effects:* Copies the elements of the range `[first1, last1)` that are not present in the range `[first2, last2)`, and the elements of the range `[first2, last2)` that are not present in the range `[first1, last1)` to the range beginning at `result`. The elements in the constructed range are sorted.

3 *Returns:* The end of the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

5 *Remarks:* If `[first1, last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, then $|m - n|$ of those elements shall be copied to the output range: the last $m - n$ of these elements from `[first1, last1)` if $m > n$, and the last $n - m$ of these elements from `[first2, last2)` if $m < n$.

```
namespace ranges {
  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      WeaklyIncrementable O, class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
    tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    constexpr set_symmetric_difference_result<I1, I2, O>
      set_symmetric_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                               Comp comp = Comp{}, Proj1 proj1 = Proj1{},
                               Proj2 proj2 = Proj2{});
  template<InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
      class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
    tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
                 tag::in2(safe_iterator_t<Rng2>),
                 tag::out(O)>
    constexpr set_symmetric_difference_result<safe_iterator_t<Rng1>, safe_iterator_t<Rng2>, O>
      set_symmetric_difference(Rng1&& rng1, Rng2&& rng2, O result, Comp comp = Comp{},
                               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

6 *Effects:* Copies the elements of the range `[first1, last1)` that are not present in the range `[first2, last2)`, and the elements of the range `[first2, last2)` that are not present in the range `[first1, last1)` to the range beginning at `result`. The elements in the constructed range are sorted.

7 *Requires:* The resulting range shall not overlap with either of the original ranges.

8 *Returns:* ~~`make_tagged_tuple<tag::in1, tag::in2, tag::out>(`~~`{last1, last2, result + `n`}`~~`)`~~, where n is the number of elements in the constructed range.

9 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` applications of the comparison function and projections.

10 *Remarks:* If `[first1, last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, then $|m - n|$ of those elements shall be copied to the output range: the last $m - n$ of these elements from `[first1, last1)` if $m > n$, and the last $n - m$ of these elements from `[first2, last2)` if $m < n$.

30.7.7 Heap operations [alg.heap.operations]

[...]

30.7.7.1 push_heap [push.heap]

```
template<class RandomAccessIterator>
  void push_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void push_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

1 *Requires:* The range `[first, last - 1)` shall be a valid heap. The type of `*first` shall satisfy the *Cpp17MoveConstructible* requirements ([tab:moveconstructible]) and the *Cpp17MoveAssignable* requirements ([tab:moveassignable]).

2 *Effects:* Places the value in the location `last - 1` into the resulting heap `[first, last)`.

3 *Complexity:* At most $\log(\texttt{last - first})$ comparisons.

```
namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    constexpr I
      push_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    constexpr safe_iterator_t<Rng>
      push_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

4 *Effects:* Places the value in the location `last - 1` into the resulting heap `[first, last)`.

5 *Requires:* The range `[first, last - 1)` shall be a valid heap.

6 *Returns:* `last`

7 *Complexity:* At most `log(last - first)` applications of the comparison function and projection.

30.7.7.2 `pop_heap` [pop.heap]

```
template<class RandomAccessIterator>
  void pop_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
                Compare comp);
```

1 *Requires:* The range `[first, last)` shall be a valid non-empty heap. `RandomAccessIterator` shall satisfy the *Cpp17ValueSwappable* requirements ([swappable.requirements]). The type of `*first` shall satisfy the *Cpp17MoveConstructible* ([tab:moveconstructible]) and *Cpp17MoveAssignable* ([tab:moveassignable]) requirements.

2 *Effects:* Swaps the value in the location `first` with the value in the location `last - 1` and makes `[first, last - 1)` into a heap.

3 *Complexity:* At most $2\log(\texttt{last - first})$ comparisons.

```
namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    constexpr I
      pop_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    constexpr safe_iterator_t<Rng>
      pop_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

4 *Requires:* The range `[first, last)` shall be a valid non-empty heap.

5 *Effects:* Swaps the value in the location `first` with the value in the location `last - 1` and makes `[first, last - 1)` into a heap.

6 *Returns:* `last`

7 *Complexity:* At most `2 * log(last - first)` applications of the comparison function and projection.

30.7.7.3 `make_heap` [make.heap]

```
template<class RandomAccessIterator>
  void make_heap(RandomAccessIterator first, RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class Compare>
  void make_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

1 *Requires:* The type of `*first` shall ~~satisfy~~ meet the *Cpp17MoveConstructible* requirements ([tab:moveconstructible]) and the *Cpp17MoveAssignable* requirements ([tab:moveassignable]).

2 *Effects:* Constructs a heap out of the range `[first, last)`.

3 *Complexity:* At most 3(`last - first`) comparisons.

```
namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    constexpr I
      make_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    constexpr safe_iterator_t<Rng>
      make_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

4 *Effects:* Constructs a heap out of the range `[first, last)`.

5 *Returns:* `last`

6 *Complexity:* At most `3 * (last - first)` applications of the comparison function and projection.

30.7.7.4 `sort_heap` [sort.heap]

```
template<class RandomAccessIterator>
  void sort_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

1 *Requires:* The range `[first, last)` shall be a valid heap. `RandomAccessIterator` shall satisfy the *Cpp17ValueSwappable* requirements ([swappable.requirements]). The type of `*first` shall satisfy the *Cpp17MoveConstructible* ([tab:moveconstructible]) and *Cpp17MoveAssignable* ([tab:moveassignable]) requirements.

2 *Effects:* Sorts elements in the heap `[first, last)`.

3 *Complexity:* At most $2N \log N$ comparisons, where $N =$ `last - first`.

```
namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    constexpr I
      sort_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<RandomAccessRange Rng, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    constexpr safe_iterator_t<Rng>
      sort_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

4 *Effects:* Sorts elements in the heap `[first, last)`.

5 *Requires:* The range `[first, last)` shall be a valid heap.

6 *Returns:* `last`

7 *Complexity:* At most $N \log(N)$ comparisons (where `N == last - first`), and exactly twice as many applications of the projection.

30.7.7.5 is_heap [is.heap]

```
template<class RandomAccessIterator>
  constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last);
```

1 *Returns:* `is_heap_until(first, last) == last`.

```
template<class ExecutionPolicy, class RandomAccessIterator>
  bool is_heap(ExecutionPolicy&& exec,
               RandomAccessIterator first, RandomAccessIterator last);
```

2 *Returns:* `is_heap_until(std::forward<ExecutionPolicy>(exec), first, last) == last`.

```
template<class RandomAccessIterator, class Compare>
  constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last,
                         Compare comp);
```

3 *Returns:* `is_heap_until(first, last, comp) == last`.

```
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  bool is_heap(ExecutionPolicy&& exec,
               RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

4 *Returns:*

```
is_heap_until(std::forward<ExecutionPolicy>(exec), first, last, comp) == last
```

```
namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
      IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less<>>
    constexpr bool is_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<RandomAccessRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    constexpr bool is_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

5 *Returns:* `is_heap_until(first, last, comp, proj) == last`

```
template<class RandomAccessIterator>
  constexpr RandomAccessIterator
    is_heap_until(RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator>
  RandomAccessIterator
    is_heap_until(ExecutionPolicy&& exec,
                  RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  constexpr RandomAccessIterator
    is_heap_until(RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  RandomAccessIterator
    is_heap_until(ExecutionPolicy&& exec,
                  RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp);
```

6 *Returns:* If `(last - first) < 2`, returns `last`. Otherwise, returns the last iterator `i` in `[first, last]` for which the range `[first, i)` is a heap.

7 *Complexity:* Linear.

```
namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
      IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less<>>
    constexpr I is_heap_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<RandomAccessRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    constexpr safe_iterator_t<Rng>
      is_heap_until(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

```
}
```

8 *Returns:* If `distance(first, last) < 2`, returns `last`. Otherwise, returns the last iterator `i` in `[first, last]` for which the range `[first, i)` is a heap.

9 *Complexity:* Linear.

30.7.8 Minimum and maximum [alg.min.max]

```
template<class T> constexpr const T& min(const T& a, const T& b);
template<class T, class Compare>
  constexpr const T& min(const T& a, const T& b, Compare comp);
```

1 *Requires:* For the first form, type `T` shall be `LessThanComparable` ([tab:lessthancomparable]).

2 *Returns:* The smaller value.

3 *Remarks:* Returns the first argument when the arguments are equivalent.

4 *Complexity:* Exactly one comparison.

```
namespace ranges {
  template<class T, class Proj = identity,
      IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr const T& min(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
}
```

5 *Returns:* The smaller value.

6 *Remarks:* Returns the first argument when the arguments are equivalent.

```
template<class T>
  constexpr T min(initializer_list<T> t);
template<class T, class Compare>
  constexpr T min(initializer_list<T> t, Compare comp);
```

7 *Requires:* `T` shall be *Cpp17CopyConstructible* and `t.size() > 0`. For the first form, type `T` shall be `LessThanComparable`.

8 *Returns:* The smallest value in the initializer list.

9 *Remarks:* Returns a copy of the leftmost argument when several arguments are equivalent to the smallest.

10 *Complexity:* Exactly `t.size() - 1` comparisons.

```
namespace ranges {
  template<Copyable T, class Proj = identity,
      IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr T min(initializer_list<T> t, Comp comp = Comp{}, Proj proj = Proj{});
  template<InputRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    requires Copyable<iter_value~~_type~~_t<iterator_t<Rng>>>
    constexpr iter_value~~_type~~_t<iterator_t<Rng>>
      min(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

11 *Requires:* `distance(rng) > 0`.

12 *Returns:* The smallest value in the `initializer_list` or range.

13 *Remarks:* Returns a copy of the leftmost argument when several arguments are equivalent to the smallest.

```
template<class T> constexpr const T& max(const T& a, const T& b);
template<class T, class Compare>
  constexpr const T& max(const T& a, const T& b, Compare comp);
```

14 *Requires:* For the first form, type `T` shall be `LessThanComparable` ([tab:lessthancomparable]).

15 *Returns:* The larger value.

16 *Remarks:* Returns the first argument when the arguments are equivalent.

17 *Complexity:* Exactly one comparison.

```
namespace ranges {
  template<class T, class Proj = identity,
      IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr const T& max(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
}
```

18 *Returns:* The larger value.

19 *Remarks:* Returns the first argument when the arguments are equivalent.

```
template<class T>
  constexpr T max(initializer_list<T> t);
template<class T, class Compare>
  constexpr T max(initializer_list<T> t, Compare comp);
```

20 *Requires:* `T` shall be *Cpp17CopyConstructible* and `t.size() > 0`. For the first form, type `T` shall be `LessThanComparable`.

21 *Returns:* The largest value in the initializer list.

22 *Remarks:* Returns a copy of the leftmost argument when several arguments are equivalent to the largest.

23 *Complexity:* Exactly `t.size() - 1` comparisons.

```
namespace ranges {
  template<Copyable T, class Proj = identity,
      IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr T max(initializer_list<T> t, Comp comp = Comp{}, Proj proj = Proj{});
  template<InputRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    requires Copyable<iter_value~~_type~~_t<iterator_t<Rng>>>
    constexpr iter_value~~_type~~_t<iterator_t<Rng>>
      max(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

24 *Requires:* `distance(rng) > 0`.

25 *Returns:* The largest value in the `initializer_list` or range.

26 *Remarks:* Returns a copy of the leftmost argument when several arguments are equivalent to the largest.

```
template<class T> constexpr pair<const T&, const T&> minmax(const T& a, const T& b);
template<class T, class Compare>
  constexpr pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
```

27 *Requires:* For the first form, type `T` shall be `LessThanComparable` ([tab:lessthancomparable]).

28 *Returns:* `pair<const T&, const T&>(b, a)` if `b` is smaller than `a`, and `pair<const T&, const T&>(a, b)` otherwise.

29 *Remarks:* Returns `pair<const T&, const T&>(a, b)` when the arguments are equivalent.

30 *Complexity:* Exactly one comparison.

```
namespace ranges {
  template<class T, class Proj = identity,
      IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    ~~constexpr tagged_pair<tag::min(const T&), tag::max(const T&)>~~
    constexpr minmax_result<const T&>
      minmax(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
}
```

31 *Returns:* `{b, a}` if `b` is smaller than `a`, and `{a, b}` otherwise.

32 *Remarks:* Returns `{a, b}` when the arguments are equivalent.

33 *Complexity:* Exactly one comparison and exactly two applications of the projection.

```
template<class T>
  constexpr pair<T, T> minmax(initializer_list<T> t);
```

```
template<class T, class Compare>
  constexpr pair<T, T> minmax(initializer_list<T> t, Compare comp);
```

34 *Requires:* `T` shall be *Cpp17CopyConstructible* and `t.size() > 0`. For the first form, type `T` shall be `LessThanComparable`.

35 *Returns:* `pair<T, T>(x, y)`, where `x` has the smallest and `y` has the largest value in the initializer list.

36 *Remarks:* `x` is a copy of the leftmost argument when several arguments are equivalent to the smallest. `y` is a copy of the rightmost argument when several arguments are equivalent to the largest.

37 *Complexity:* At most $(3/2)$`t.size()` applications of the corresponding predicate.

```
namespace ranges {
  template<Copyable T, class Proj = identity,
      IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr tagged_pair<tag::min(T), tag::max(T)>
    constexpr minmax_result<T>
      minmax(initializer_list<T> t, Comp comp = Comp{}, Proj proj = Proj{});
  template<InputRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    requires Copyable<iter_value_type_t<iterator_t<Rng>>>
    tagged_pair<tag::min(value_type_t<iterator_t<Rng>>),
                tag::max(value_type_t<iterator_t<Rng>>)>
    constexpr minmax_result<iter_value_t<iterator_t<Rng>>>
      minmax(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

38 *Requires:* `distance(rng) > 0`.

39 *Returns:* `{x, y}`, where `x` has the smallest and `y` has the largest value in the `initializer_list` or range.

40 *Remarks:* `x` is a copy of the leftmost argument when several arguments are equivalent to the smallest. `y` is a copy of the rightmost argument when several arguments are equivalent to the largest.

41 *Complexity:* At most `(3/2) * distance(rng)` applications of the corresponding predicate, and at most twice as many applications of the projection.

```
template<class ForwardIterator>
  constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator min_element(ExecutionPolicy&& exec,
                              ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
  constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
                                        Compare comp);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  ForwardIterator min_element(ExecutionPolicy&& exec,
                              ForwardIterator first, ForwardIterator last,
                              Compare comp);
```

42 *Returns:* The first iterator `i` in the range `[first, last)` such that for every iterator `j` in the range `[first, last)` the following corresponding conditions hold: `!(*j < *i)` or `comp(*j, *i) == false`. Returns `last` if `first == last`.

43 *Complexity:* Exactly $\max($`last - first - 1`$, 0)$ applications of the corresponding comparisons.

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

```
}
```

44 *Returns:* The first iterator `i` in the range `[first, last)` such that for every iterator `j` in the range `[first, last)` the following corresponding condition holds:
`invoke(comp, invoke(proj, *j), invoke(proj, *i)) == false`. Returns `last` if `first == last`.

45 *Complexity:* Exactly `max((last - first) - 1, 0)` applications of the comparison function and exactly twice as many applications of the projection.

```
template<class ForwardIterator>
  constexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator max_element(ExecutionPolicy&& exec,
                              ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
  constexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
                                        Compare comp);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  ForwardIterator max_element(ExecutionPolicy&& exec,
                              ForwardIterator first, ForwardIterator last,
                              Compare comp);
```

46 *Returns:* The first iterator `i` in the range `[first, last)` such that for every iterator `j` in the range `[first, last)` the following corresponding conditions hold: `!(*i < *j)` or `comp(*i, *j) == false`. Returns `last` if `first == last`.

47 *Complexity:* Exactly $\max(\texttt{last - first - 1}, 0)$ applications of the corresponding comparisons.

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

48 *Returns:* The first iterator `i` in the range `[first, last)` such that for every iterator `j` in the range `[first, last)` the following corresponding condition holds:
`invoke(comp, invoke(proj, *i), invoke(proj, *j)) == false`. Returns `last` if `first == last`.

49 *Complexity:* Exactly `max((last - first) - 1, 0)` applications of the comparison function and exactly twice as many applications of the projection.

```
template<class ForwardIterator>
  constexpr pair<ForwardIterator, ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator>
  pair<ForwardIterator, ForwardIterator>
    minmax_element(ExecutionPolicy&& exec,
                   ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
  constexpr pair<ForwardIterator, ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last, Compare comp);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  pair<ForwardIterator, ForwardIterator>
    minmax_element(ExecutionPolicy&& exec,
                   ForwardIterator first, ForwardIterator last, Compare comp);
```

50 *Returns:* `make_pair(first, first)` if `[first, last)` is empty, otherwise `make_pair(m, M)`, where `m` is the first iterator in `[first, last)` such that no iterator in the range refers to a smaller element,

and where `M` is the last iterator[10] in `[first, last)` such that no iterator in the range refers to a larger element.

51 *Complexity:* At most $\max(\lfloor\frac{3}{2}(N-1)\rfloor, 0)$ applications of the corresponding predicate, where N is `last - first`.

```
namespace ranges {
  template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less<>>
    tagged_pair<tag::min(I), tag::max(I)>
    constexpr minmax_result<I>
      minmax_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<ForwardRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = ranges::less<>>
    tagged_pair<tag::min(safe_iterator_t<Rng>),
                tag::max(safe_iterator_t<Rng>)>
    constexpr minmax_result<safe_iterator_t<Rng>>
      minmax_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

52 *Returns:* `{first, first}` if `[first, last)` is empty, otherwise `{m, M}`, where `m` is the first iterator in `[first, last)` such that no iterator in the range refers to a smaller element, and where `M` is the last iterator in `[first, last)` such that no iterator in the range refers to a larger element.

53 *Complexity:* At most $max(\lfloor\frac{3}{2}(N-1)\rfloor, 0)$ applications of the comparison function and at most twice as many applications of the projection, where N is `distance(first, last)`.

30.7.9 Bounded value [alg.clamp]

[...]

30.7.10 Lexicographical comparison [alg.lex.comparison]

```
template<class InputIterator1, class InputIterator2>
  constexpr bool
    lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  bool
    lexicographical_compare(ExecutionPolicy&& exec,
                            ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
  constexpr bool
    lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2,
                            Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
       class Compare>
  bool
    lexicographical_compare(ExecutionPolicy&& exec,
                            ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2,
                            Compare comp);
```

1 *Returns:* `true` if the sequence of elements defined by the range `[first1, last1)` is lexicographically less than the sequence of elements defined by the range `[first2, last2)` and `false` otherwise.

2 *Complexity:* At most $2\min($`last1 - first1`, `last2 - first2`$)$ applications of the corresponding comparison.

3 *Remarks:* If two sequences have the same number of elements and their corresponding elements (if any) are equivalent, then neither sequence is lexicographically less than the other. If one sequence is a prefix of the other, then the shorter sequence is lexicographically less than the longer sequence. Otherwise,

10) This behavior intentionally differs from `max_element()`.

the lexicographical comparison of the sequences yields the same result as the comparison of the first corresponding pair of elements that are not equivalent.

4 [*Example:* The following sample implementation satisfies these requirements:

```
for ( ; first1 != last1 && first2 != last2 ; ++first1, (void) ++first2) {
  if (*first1 < *first2) return true;
  if (*first2 < *first1) return false;
}
return first1 == last1 && first2 != last2;
```

— *end example*]

5 [*Note:* An empty sequence is lexicographically less than any non-empty sequence, but not less than any empty sequence. — *end note*]

```
namespace ranges {
  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      class Proj1 = identity, class Proj2 = identity,
      IndirectStrictWeakOrder<projected<I1, Proj1>, projected<I2, Proj2>> Comp = ranges::less<>>
    constexpr bool
      lexicographical_compare(I1 first1, S1 last1, I2 first2, S2 last2,
                              Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<InputRange Rng1, InputRange Rng2, class Proj1 = identity,
      class Proj2 = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng1>, Proj1>,
        projected<iterator_t<Rng2>, Proj2>> Comp = ranges::less<>>
    constexpr bool
      lexicographical_compare(Rng1&& rng1, Rng2&& rng2, Comp comp = Comp{},
                              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

6 *Returns:* `true` if the sequence of elements defined by the range `[first1, last1)` is lexicographically less than the sequence of elements defined by the range `[first2, last2)` and `false` otherwise.

7 *Complexity:* At most `2*min((last1 - first1), (last2 - first2))` applications of the corresponding comparison and projections.

8 *Remarks:* If two sequences have the same number of elements and their corresponding elements are equivalent, then neither sequence is lexicographically less than the other. If one sequence is a prefix of the other, then the shorter sequence is lexicographically less than the longer sequence. Otherwise, the lexicographical comparison of the sequences yields the same result as the comparison of the first corresponding pair of elements that are not equivalent.

```
for ( ; first1 != last1 && first2 != last2 ; ++first1, (void) ++first2) {
  if (invoke(comp, invoke(proj1, *first1), invoke(proj2, *first2))) return true;
  if (invoke(comp, invoke(proj2, *first2), invoke(proj1, *first1))) return false;
}
return first1 == last1 && first2 != last2;
```

9 *Remarks:* An empty sequence is lexicographically less than any non-empty sequence, but not less than any empty sequence.

30.7.11 Three-way comparison algorithms [alg.3way]

[...]

30.7.12 Permutation generators [alg.permutation.generators]

```
template<class BidirectionalIterator>
  bool next_permutation(BidirectionalIterator first,
                        BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
  bool next_permutation(BidirectionalIterator first,
                        BidirectionalIterator last, Compare comp);
```

1 *Requires:* `BidirectionalIterator` shall satisfy the *Cpp17ValueSwappable* requirements ([swappable.requirements]).

2 *Effects:* Takes a sequence defined by the range `[first, last)` and transforms it into the next permutation. The next permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `operator<` or `comp`.

3 *Returns:* `true` if such a permutation exists. Otherwise, it transforms the sequence into the smallest permutation, that is, the ascendingly sorted one, and returns `false`.

4 *Complexity:* At most `(last - first) / 2` swaps.

```
namespace ranges {
  template<BidirectionalIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    constexpr bool
      next_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<BidirectionalRange Rng, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    constexpr bool
      next_permutation(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

5 *Effects:* Takes a sequence defined by the range `[first, last)` and transforms it into the next permutation. The next permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `comp` and `proj`. If such a permutation exists, it returns `true`. Otherwise, it transforms the sequence into the smallest permutation, that is, the ascendingly sorted one, and returns `false`.

6 *Complexity:* At most `(last - first)/2` swaps.

```
template<class BidirectionalIterator>
  bool prev_permutation(BidirectionalIterator first,
                        BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
  bool prev_permutation(BidirectionalIterator first,
                        BidirectionalIterator last, Compare comp);
```

7 *Requires:* `BidirectionalIterator` shall satisfy the *Cpp17ValueSwappable* requirements ([swappable.requirements]).

8 *Effects:* Takes a sequence defined by the range `[first, last)` and transforms it into the previous permutation. The previous permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `operator<` or `comp`.

9 *Returns:* `true` if such a permutation exists. Otherwise, it transforms the sequence into the largest permutation, that is, the descendingly sorted one, and returns `false`.

10 *Complexity:* At most `(last - first) / 2` swaps.

```
namespace ranges {
  template<BidirectionalIterator I, Sentinel<I> S, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    constexpr bool
      prev_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<BidirectionalRange Rng, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    constexpr bool
      prev_permutation(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}
```

11 *Effects:* Takes a sequence defined by the range `[first, last)` and transforms it into the previous permutation. The previous permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `comp` and `proj`.

12 *Returns:* `true` if such a permutation exists. Otherwise, it transforms the sequence into the largest permutation, that is, the descendingly sorted one, and returns `false`.

13 *Complexity:* At most `(last - first)/2` swaps.

30.8 C library algorithms [alg.c.library]

[...]

31 Numerics library [numerics]

31.7 Numeric arrays [numarray]

[Editor's note: In the paragraph that specifies `valarray`'s iterators, cross-reference "contiguous iterator" to [iterator.concept.contiguous] instead of [iterator.requirements.general]. (Is this too subtle a means to require implementations to specialize `ranges::iterator_category`?)]

31.7.10 `valarray` range access [valarray.range]

1 In the `begin` and `end` function templates that follow, ***`unspecified`***1 is a type that meets the requirements of a mutable random access iterator ([random.access.iterators]) and of a contiguous iterator (28.3.4.13) whose `value_type` is the template parameter `T` and whose `reference` type is `T&`. ***`unspecified`***2 is a type that meets the requirements of a constant random access iterator ([random.access.iterators]) and of a contiguous iterator (28.3.4.13) whose `value_type` is the template parameter `T` and whose `reference` type is `const T&`.

2 [...]

Annex A (informative) Acknowledgements [acknowledgements]

This work was made possible in part by a grant from the Standard C++ Foundation, and by the support of the authors' employers Facebook and Microsoft.

We'd like to especially thank R. Tim Song for all of his work documenting Ranges on cppreference, and for all of the wording and design issues he has found and reported in the process.

Bibliography

[1] Casey Carter. Cmcstl2. https://github.com/CaseyCarter/CMCSTL2. Accessed: 2018-1-31.

[2] Casey Carter and Eric Niebler. P0898: Standard library concepts, 05 2018. http://wg21.link/P0898.

[3] Eric Niebler. Range-v3. https://github.com/ericniebler/range-v3. Accessed: 2018-1-31.

[4] Eric Niebler. P0789r3: Range adaptors and utilities, 05 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0789r3.pdf.

Index

Index of library names

Index of implementation-defined behavior

The entries in this section are rough descriptions; exact specifications are at the indicated page in the general text.