Document Number: P0896R3
Date: 2018-10-07
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 **1**
1.1 Revision History . . . 1

2 General Principles **2**
2.1 Goals . . . 2
2.2 Rationale . . . 2
2.3 Risks . . . 2
2.4 Methodology . . . 3
2.5 Style of presentation . . . 3

15 Library introduction **5**
15.1 General . . . 5
15.3 Definitions . . . 5
15.5 Library-wide requirements . . . 5

17 Concepts library **6**
17.3 Header `<concepts>` synopsis . . . 6
17.4 Language-related concepts . . . 7

19 General utilities library **9**
19.10 Memory . . . 9
19.14 Function Objects . . . 19

20 Strings library **22**
20.4 String view classes . . . 22

21 Containers library **23**
21.2 Container requirements . . . 23
21.7 Views . . . 23

22 Iterators library **23**
22.1 General . . . 23
22.2 Header `<iterator>` synopsis . . . 24
22.3 Iterator requirements . . . 29
22.4 Iterator primitives . . . 45
22.5 Iterator adaptors . . . 47
22.6 Stream iterators . . . 65

23 Ranges library **68**
23.1 General . . . 68
23.2 decay_copy . . . 68
23.3 Header `<ranges>` synopsis . . . 68
23.4 Range access . . . 72
23.5 Range primitives . . . 73
23.6 Range requirements . . . 75
23.7 Range utilities . . . 78
23.8 Range adaptors . . . 84

24 Algorithms library **121**
24.1 General . . . 121
24.2 Header `<algorithm>` synopsis . . . 121
24.3 Algorithms requirements . . . 155
24.5 Non-modifying sequence operations . . . 156
24.6 Mutating sequence operations . . . 167

24.7 Sorting and related operations 182

25 Numerics library **209**
25.8 Numeric arrays 209

A Acknowledgements **210**

Bibliography **210**

Index **211**

Index of library names **212**

Index of implementation-defined behavior **217**

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 3 [intro.history.r3]

— (Throughout) Use R (r) instead of Rng (rng) for Range template (function) parameter names.

— (Throughout) In addition to changing the cross references, replace "is a contiguous iterator" in container requirements with "models ContiguousIterator"

defns.projection Use string_view instead of char* in example.

ocator.requirements Rephrase final sentence

concepts.swappable Define semantics of the swap concepts completely in the ranges::swap customization point, which here from [utility]

iterator.synopsis Replace bogus -> auto&& deduction constraints with a legit requirement that the expression has a referenceable type

ustpoints.iter_swap Vastly simplify conditional noexcept for the exposition-only *iter-exchange-move*.

range.subrange Merge subrange deduction guides with identical parameter-deeclaration-clauses that otherwise conflict per [temp.deduct.guide]/3.

algorithm.syn Simplify algorithm declarations with the form:

```
template <ForwardIterator I, [...]>
  requires Permutable<I> && [...]
[...]
```

to the equivalent (since Permutable subsumes ForwardIterator):

```
template <Permutable I, [...]>
  requires [...]
[...]
```

which favors smaller declarations over consistency in form between iterator-sentinel and range overloads of algorithms that require Permutable.

1.1.2 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.

1.1.3 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.

— 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.

1) `std::two` was another popular suggestion.

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.

15 Library introduction [library]

[...]

15.1 General [library.general]

[...]

[Editor's note: Insert a new row in Table 17 for the ranges library:]

Table 17 — Library categories

Clause	Category
Clause [language.support]	Language support library
Clause 17	Concepts library
Clause [diagnostics]	Diagnostics library
Clause 19	General utilities library
Clause 20	Strings library
Clause [localization]	Localization library
Clause 21	Containers library
Clause 22	Iterators library
Clause 23	Ranges library
Clause 24	Algorithms library
Clause 25	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 21), iterators (Clause 22), ranges (Clause 23), and algorithms (Clause 24) libraries provide a C++ program with access to a subset of the most widely used algorithms and data structures.

15.3 Definitions [definitions]

[...]

15.3.18 [defns.projection]

projection

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

[*Example:*

```
std::pair<int, const char* std::string_view> pairs[] = {{2, "foo"}, {1, "bar"}, {0, "baz"}};
std::ranges::sort(pairs, std::ranges::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*]

[...]

15.5 Library-wide requirements [requirements]

[...]

15.5.1.2 Headers [headers]

[...]

Table 18 — C++ library headers

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

15.5.1.3 *Cpp17Allocator* requirements [allocator.requirements]

[...]

5 An allocator type `X` shall ~~satisfy~~meet the *Cpp17CopyConstructible* requirements (Table [copyconstructible]). The `X::pointer`, `X::const_pointer`, `X::void_pointer`, and `X::const_void_pointer` types shall ~~satisfy~~meet 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~~meet the requirements for a ~~random access iterator~~ *Cpp17RandomAccessIterator* (22.3.5.6) and ~~of a contiguous iterator (22.3.1).~~ the additional requirement that, when `a` and `(a + n)` are valid dereferenceable pointer values for some integral value `n`,

```
addressof(*(a + n)) == addressof(*a) + n
```

is `true`.

17 Concepts library [concepts]

17.3 Header <concepts> synopsis [concepts.syn]

```
namespace std {
  [...]

  // [concept.assignable], concept Assignable
  template<class LHS, class RHS>
    concept Assignable = see below;

  // 17.4.11, concept Swappable
  namespace ranges {
    inline namespace unspecified {
      inline constexpr unspecified swap = unspecified;
    }
  }

  template<class T>
    concept Swappable = see below;
  template<class T, class U>
    concept SwappableWith = see below;
```

```
    [...]
  }
```

17.4 Language-related concepts [concepts.lang]

17.4.11 Concept `Swappable` [concept.swappable]

1 Let `t1` and `t2` be equality-preserving expressions that denote distinct equal objects of type `T`, and let `u1` and `u2` similarly denote distinct equal objects of type `U`. An operation *exchanges the values* denoted by `t1` and `u1` if and only if the operation modifies neither `t2` nor `u2` and:

(1.1) — If `T` and `U` are the same type, the result of the operation is that `t1` equals `u2` and `u1` equals `t2`.

(1.2) — If `T` and `U` are different types that model `CommonReference<const T&, const U&>`, the result of the operation is that `C(t1)` equals `C(u2)` and `C(u1)` equals `C(t2)` where `C` is `common_reference_t<const T&, const U&>`.

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

(2.1) — `(void)swap(E1, E2)`[2], if `E1` or `E2` has class type ([basic.compound]) and 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~~ denoted by `E1` and `E2`, the program is ill-formed with no diagnostic required.

(2.2) — 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~~)~~))`.

(2.3) — Otherwise, if `E1` and `E2` are lvalues of the same type `T` that ~~meets the syntactic requirements of~~ models `MoveConstructible<T>` and `Assignable<T&, T>`, exchanges the ~~referenced~~ denoted 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.~~

(2.3.1) — both `E1 = std::move(E2)` and `E2 = std::move(E1)` are constant expressions, and

(2.3.2) — the initializers in the declarations `T t1(std::move(E1));` and `T t2(std::move(E2));` are constant initializers ([basic.start.static]).

`noexcept(ranges::swap(E1, E2))` is equal to `is_nothrow_move_constructible_v<T> && is_nothrow_move_assignable_v<T>`. ~~If either `MoveConstructible` or `Assignable` is not satisfied, the program is ill-formed with no diagnostic required.~~

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

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

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

4 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 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`.

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

(5.1) — the two `swap` function templates defined in `<utility>` ([utility]) and

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

(5.2) — 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> &&
    CommonReference<const remove_reference_t<T>&, const remove_reference_t<U>&> &&
    is_swappable_with_v<T, U> && is_swappable_with_v<U, T>;
    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));
    };
```

6 Let:

(6.1) — `t1` and `t2` denote distinct equal objects of type `remove_cvref_t<T>`,

(6.2) — E_t be an expression that denotes `t1` such that `decltype((`E_t`))` is `T`,

(6.3) — `u1` and `u2` similarly denote distinct equal objects of type `remove_cvref_t<U>`,

(6.4) — E_u be an expression that denotes `u1` such that `decltype((`E_u`))` is `U`, and

(6.5) — `C` be

```
common_reference_t<const remove_reference_t<T>&, const remove_reference_t<U>&>
```

`SwappableWith<T, U>` is satisfied only if 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)`.

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

(7.1) — the two `swap` function templates defined in `<utility>` ([utility]) and

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

8 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`.

9 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`.

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

11 [*Note:* The semantics of the `Swappable` and `SwappableWith` concepts are fully defined by the `ranges::swap` customization point. — *end note*]

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

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

namespace ranges = std::ranges;

template<class T, std::SwappableWith<T> U>
void value_swap(T&& t, U&& u) {
  using std::swap;
  ranges::swap(std::forward<T>(t), std::forward<U>(u)); // OK: uses ‘‘swappable with’’ conditions
                                                         // for rvalues and lvalues
}

template<std::Swappable T>
void lv_swap(T& t1, T& t2) {
  using std::swap;
```

```
    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) {
      stdranges::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 constraintrequirement
  }

  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*]

19 General utilities library [utilities]

19.10 Memory [memory]

19.10.2 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;

  // 19.10.11, specialized algorithms
  // 19.10.11.1, 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 R>
    concept no-throw-input-range = see below; // exposition only

  template<class R>
    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_t<I>>
      I uninitialized_default_construct(I first, S last);
  template<no-throw-forward-range R>
    requires DefaultConstructible<iter_value_t<iterator_t<R>>>
      safe_iterator_t<R> uninitialized_default_construct(R&& r);

  template<no-throw-forward-iterator I>
    requires DefaultConstructible<iter_value_t<I>>
      I uninitialized_default_construct_n(I first, iter_difference_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_t<I>>
      I uninitialized_value_construct(I first, S last);
  template<no-throw-forward-range R>
    requires DefaultConstructible<iter_value_t<iterator_t<R>>>
      safe_iterator_t<R> uninitialized_value_construct(R&& r);

  template<no-throw-forward-iterator I>
    requires DefaultConstructible<iter_value_t<I>>
      I uninitialized_value_construct_n(I first, iter_difference_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;
    O out;
  };
  using uninitialized_copy_result = copy_result<I, O>;
  template<InputIterator I, Sentinel<I> S1, no-throw-forward-iterator O, no-throw-sentinel<O> S2>
    requires Constructible<iter_value_t<O>, iter_reference_t<I>>
      uninitialized_copy_result<I, O>
        uninitialized_copy(I ifirst, S1 ilast, O ofirst, S2 olast);
  template<InputRange IR, no-throw-forward-range OR>
    requires Constructible<iter_value_t<iterator_t<OR>>, iter_reference_t<iterator_t<IR>>>
      uninitialized_copy_result<safe_iterator_t<IR>, safe_iterator_t<OR>>
        uninitialized_copy(IR&& ir, OR&& or);

  template<class I, class O>
    using uninitialized_copy_n_result = uninitialized_copy_result<I, O>;
  template<InputIterator I, no-throw-forward-iterator O, no-throw-sentinel<O> S>
    requires Constructible<iter_value_t<O>, iter_reference_t<I>>
      uninitialized_copy_n_result<I, O>
        uninitialized_copy_n(I ifirst, iter_difference_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_t<O>, iter_rvalue_reference_t<I>>
      uninitialized_move_result<I, O>
        uninitialized_move(I ifirst, S1 ilast, O ofirst, S2 olast);
  template<InputRange IR, no-throw-forward-range OR>
    requires Constructible<iter_value_t<iterator_t<OR>>, iter_rvalue_reference_t<iterator_t<IR>>>
      uninitialized_move_result<safe_iterator_t<IR>, safe_iterator_t<OR>>
        uninitialized_move(IR&& ir, OR&& or);

  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_t<O>, iter_rvalue_reference_t<I>>
      uninitialized_move_n_result<I, O>
        uninitialized_move_n(I ifirst, iter_difference_t<I> n, O ofirst, S olast);
}
```

```
  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_t<I>, const T&>
        I uninitialized_fill(I first, S last, const T& x);
    template<no-throw-forward-range R, class T>
      requires Constructible<iter_value_t<iterator_t<R>>, const T&>
        safe_iterator_t<R> uninitialized_fill(R&& r, const T& x);

    template<no-throw-forward-iterator I, class T>
      requires Constructible<iter_value_t<I>, const T&>
        I uninitialized_fill_n(I first, iter_difference_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_t<I>>
        I destroy(I first, S last) noexcept;
    template<no-throw-input-range R>
      requires Destructible<iter_value_t<iterator_t<R>>
        safe_iterator_t<R> destroy(R&& r) noexcept;

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

  [...]
}
```

[...]

19.10.11 Specialized algorithms [specialized.algorithms]

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

(1.1) — If an algorithm's template parameter is named `InputIterator`, the template argument shall ~~satisfy~~meet the *Cpp17InputIterator* requirements (22.3.5.2).

(1.2) — If an algorithm's template parameter is named `ForwardIterator`, the template argument shall ~~satisfy~~meet the *Cpp17ForwardIterator* requirements (22.3.5.4), 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.~~

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

3 In the description of the algorithms operator~~s + and~~ - ~~are~~is used for some of the iterator categories for which ~~they do not have to~~ it need not 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~~ value of the expression `b - a` is the number of increments of `a` needed to make `bool(a == b)` be `true`.

~~the same as of~~

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

4 ~~For the algorithms in this subclause defined in namespace `std::ranges`,~~ The following additional requirements apply for those algorithms defined in namespace `std::ranges`:

(4.1) — 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.

(4.2) — Overloads of algorithms that take `Range` arguments (23.6.2) 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.

(4.3) — The number and order of template parameters for algorithm declarations is unspecified, except where explicitly stated otherwise. [*Note:* Consequently, these algorithms may not be called with explicitly template arguments. — *end note*]

5 [*Note:* When invoked on ranges of potentially overlapping subobjects ([intro.object]), ~~Invocation of~~ the algorithms specified in this subclause 19.10.11 ~~on ranges of potentially overlapping subobjects ([intro.object])~~ result~~s~~ in undefined behavior. — *end note*]

6 Some algorithms defined in this clause make use of the exposition-only function *`voidify`*:

```
template<class T>
  void* voidify(T& ptr) noexcept {
    return const_cast<void*>(static_cast<const volatile void*>(addressof(ptr)));
  }
```

19.10.11.1 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 construction, move construction, copy assignment, move assignment, or indirection through valid iterators.

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

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

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

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

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

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

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

19.10.11.2 addressof [specialized.addressof]

[...]

19.10.11.3 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_t<I>>
    I uninitialized_default_construct(I first, S last);
  template<no-throw-forward-range R>
      requires DefaultConstructible<iter_value_t<iterator_t<R>>>
    safe_iterator_t<R> uninitialized_default_construct(R&& r);
}
```

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>>;
  ::new (voidify(*first)) remove_reference_t<iter_reference_t<I>>;
return first;
```

```
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_t<I>>
    I uninitialized_default_construct_n(I first, iter_difference_t<I> n);
}
```

4 *Effects:* Equivalent to:

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

19.10.11.4 `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_t<I>>
    I uninitialized_value_construct(I first, S last);
  template<no-throw-forward-range R>
      requires DefaultConstructible<iter_value_t<iterator_t<R>>>
    safe_iterator_t<R> uninitialized_value_construct(R&& r);
}
```

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>>();
  ::new (voidify(*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_t<I>>
    I uninitialized_value_construct_n(I first, iter_difference_t<I> n);
}
```

4 *Effects:* Equivalent to:

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

19.10.11.5 `uninitialized_copy` [uninitialized.copy]

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

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

2 *Effects:* ~~As if by~~Equivalent to:

```
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_t<O>, iter_reference_t<I>>
    uninitialized_copy_result<I, O>
      uninitialized_copy(I ifirst, S1 ilast, O ofirst, S2 olast);
```

```
  template<InputRange IR, no-throw-forward-range OR>
      requires Constructible<iter_value_t<iterator_t<OR>>, iter_reference_t<iterator_t<IR>>>
    uninitialized_copy_result<safe_iterator_t<IR>, safe_iterator_t<OR>>
      uninitialized_copy(IR&& ir, OR&& or);
}
```

4 *Expects:* `[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);
  ::new (voidify(*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 *Expects:* `[result, result + n)` shall not overlap with `[first, first + n)`.

7 *Effects:* ~~As if by~~Equivalent to:

```
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_t<O>, iter_reference_t<I>>
    uninitialized_copy_n_result<I, O>
      uninitialized_copy_n(I ifirst, iter_difference_t<I> n, O ofirst, S olast);
}
```

9 *Expects:* `[ofirst, olast)` shall not overlap with `[ifirst, ifirst + n)`.

10 *Effects:* Equivalent to:

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

19.10.11.6 `uninitialized_move` [uninitialized.move]

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

1 *Expects:* `[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_t<O>, iter_rvalue_reference_t<I>>
    uninitialized_move_result<I, O>
      uninitialized_move(I ifirst, S1 ilast, O ofirst, S2 olast);
```

```
  template<InputRange IR, no-throw-forward-range OR>
      requires Constructible<iter_value_t<iterator_t<OR>>, iter_rvalue_reference_t<iterator_t<IR>>>
    uninitialized_move_result<safe_iterator_t<IR>, safe_iterator_t<OR>>
      uninitialized_move(IR&& ir, OR&& or);
}
```

4 *Expects:* `[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));
  ::new (voidify(*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 *Expects:* `[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, std::next(first,n)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_t<O>, iter_rvalue_reference_t<I>>
    uninitialized_move_n_result<I, O>
      uninitialized_move_n(I ifirst, iter_difference_t<I> n, O ofirst, S olast);
}
```

10 *Expects:* `[ofirst, olast)` shall not overlap with `[ifirst, ifirst + n)`.

11 *Effects:* Equivalent to:

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

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

19.10.11.7 `uninitialized_fill` [uninitialized.fill]

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

1 *Effects:* As if byEquivalent to:

```
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_t<I>, const T&>
    I uninitialized_fill(I first, S last, const T& x);
```

```
  template<no-throw-forward-range R, class T>
      requires Constructible<iter_value_t<iterator_t<R>>, const T&>
    safe_iterator_t<R> uninitialized_fill(R&& r, 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_t<I>>(x);
  ::new (voidify(*first)) remove_reference_t<iter_reference_t<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 byEquivalent to:

```
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_t<I>, const T&>
    I uninitialized_fill_n(I first, iter_difference_t<I> n, const T& x);
}
```

3 *Effects:* Equivalent to:

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

19.10.11.8 destroy [specialized.destroy]

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

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

1 *Effects:*

(1.1) — If T is an array type, equivalent to destroy(begin(*location), end(*location)).

(1.2) — Otherwise, 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_t<I>>
    I destroy(I first, S last) noexcept;
  template<no-throw-input-range R>
      requires Destructible<iter_value_t<iterator_t<R>>>
    safe_iterator_t<R> destroy(R&& r) noexcept;
}
```

3 *Effects:* Equivalent to:

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

```
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_t<I>>
    I destroy_n(I first, iter_difference_t<I> n) noexcept;
}
```

5 *Effects:* Equivalent to:

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

[...]

19.14 Function Objects [function.objects]

19.14.1 Header <functional> synopsis [functional.syn]

```
[...]

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 {
  // 19.14.8, 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>;
}
}
```

[...]

19.14.7 Comparisons [comparisons]

[...]

19.14.8 Comparisons (ranges) [range.comparisons]

1 In this subclause, *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.

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

```
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;
};
```

3 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;
};
```

4 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;
};
```

5 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;
};
```

6 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;
};
```

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 less_equal {
  constexpr bool operator()(const T& x, const T& y) const;
};
```

8 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;
};
```

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

10 *Effects:*

(10.1) — 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`.

(10.2) — 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;
};
```

11 `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;
};
```

12 `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;
};
```

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

14 *Effects:*

(14.1) — 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`.

(14.2) — 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;
};
```

15 `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;
};
```

16 `operator()` has effects equivalent to:

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

19.14.9 Logical operations [logical.operations]

[...]

20 Strings library [strings]

[...]

20.4 String view classes [string.view]

[...]

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

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

    // 20.4.2.2, 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;

    [...]
  };
```

[...]

20.4.2.2 Iterator support [string.view.iterators]

`using const_iterator = implementation-defined;`

1 A type that meets the requirements of a constant random access iterator (22.3.5.6) and ~~of a contiguous iterator (22.3.4.14)~~ models `ContiguousIterator` whose `value_type` is the template parameter `charT`.

2 [...]

[...]

21 Containers library [containers]

[...]

21.2 Container requirements [container.requirements]

21.2.1 General container requirements [container.requirements.general]

[...]

13 A *contiguous container* is a container that supports random access iterators (22.3.5.6) and whose member types `iterator` and `const_iterator` ~~are contiguous iterators (22.3.1)~~ each model `ContiguousIterator` (22.3.4.14).

[...]

21.7 Views [views]

[...]

21.7.3 Class template span [views.span]

21.7.3.1 Overview [span.overview]

[...]

2 The iterator types `span::iterator` and `span::const_iterator` are random access iterators (22.3.5.6)~~, contiguous iterators (22.3.1),~~ and constexpr iterators (22.3.1), and each model `ContiguousIterator`. All requirements on container iterators (21.2) apply to `span::iterator` and `span::const_iterator` as well.

3 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(); }

    [...]
  };
}
```

[...]

22 Iterators library [iterators]

22.1 General [iterators.general]

1 This Clause describes components that C++ programs may use to perform iterations over containers (Clause 21), streams ([iostream.format]), ~~and~~ stream buffers ([stream.buffers]), and other ranges (Clause 23).

2 The following subclauses describe iterator requirements, and components for iterator primitives, predefined iterators, and stream iterators, as summarized in Table 73.

Table 73 — Iterators library summary

	Subclause	Header(s)
22.3	~~R~~Iterator requirements	<iterator>
22.3.6	Indirect callable requirements	
22.3.7	Common algorithm requirements	
22.4	Iterator primitives	~~<iterator>~~
22.5	Predefined iterators	
22.6	Stream iterators	

[Editor's note: Move [iterator.synopsis] here immediately after [iterators.general] and modify as follows:]

22.2 Header <iterator> synopsis [iterator.synopsis]

```
#include <concepts>

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

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

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

  // 22.4, primitives22.3.2.3, Iterator traits
  template<class IteratorI> struct iterator_traits;
  template<class T> struct iterator_traits<T*>;

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

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

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

  template<dereferenceable T>
    requires requires (T& t) { { ranges::iter_move(t) } -> auto &&; }
```

```
    using iter_rvalue_reference_t
      = decltype(ranges::iter_move(declval<T&>()));

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
// 22.3.6, indirect callable requirements
// 22.3.6.2, indirect callables
template<class F, class I>
  concept IndirectUnaryInvocable = see below;

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

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

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

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

template<class F, class... Is>
  requires (Readable<Is> && ...) && Invocable<F, iter_reference_t<Is>...>
    using indirect_result_t = invoke_result_t<F, iter_reference_t<Is>...>;

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

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

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

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

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

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

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

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

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

// 22.3.7.7 Mergeable
template<class I1, class I2, class Out,
    class R = ranges::less<>, class P1 = identity, class P2 = identity>
  concept Mergeable = see below;

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

```
// 22.4, primitives
// 22.4.1, iterator tags
struct input_iterator_tag { };
struct output_iterator_tag { };
struct forward_iterator_tag: public input_iterator_tag { };
struct bidirectional_iterator_tag: public forward_iterator_tag { };
struct random_access_iterator_tag: public bidirectional_iterator_tag { };
struct contiguous_iterator_tag: public random_access_iterator_tag { };

// 22.4.2, iterator operations
template<class InputIterator, class Distance>
  constexpr void
    advance(InputIterator& i, Distance n);
template<class InputIterator>
  constexpr typename iterator_traits<InputIterator>::difference_type
    distance(InputIterator first, InputIterator last);
template<class InputIterator>
  constexpr InputIterator
    next(InputIterator x,
         typename iterator_traits<InputIterator>::difference_type n = 1);
template<class BidirectionalIterator>
  constexpr BidirectionalIterator
    prev(BidirectionalIterator x,
         typename iterator_traits<BidirectionalIterator>::difference_type n = 1);
// 22.4.3, range iterator operations
namespace ranges {
  // 22.4.3.1, ranges::advance
  template<Iterator I>
    constexpr void advance(I& i, iter_difference_t<I> n);
  template<Iterator I, Sentinel<I> S>
    constexpr void advance(I& i, S bound);
  template<Iterator I, Sentinel<I> S>
    constexpr iter_difference_t<I> advance(I& i, iter_difference_t<I> n, S bound);

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

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

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

// 22.5, predefined iterators and sentinels
// 22.5.1, reverse iterators
template<class Iterator> class reverse_iterator;
```

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

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

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

// 22.5.2, insert iterators
template<class Container> class back_insert_iterator;
template<class Container>
  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::iteratoriterator_t<Container> i);

// 22.5.3, move iterators and sentinels
template<class Iterator> class move_iterator;
template<class Iterator1, class Iterator2>
  constexpr bool operator==(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template<class Iterator1, class Iterator2>
  constexpr bool operator!=(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template<class Iterator1, class Iterator2>
  constexpr bool operator<(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

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

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

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

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

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

  // 22.5.5, default sentinels
  class default_sentinel;

  // 22.5.6, 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>>;

  // 22.5.7, unreachable sentinels
  class unreachable;

  // 22.6, stream iterators
  [...]
}
```

22.3 Iterator requirements [iterator.requirements]

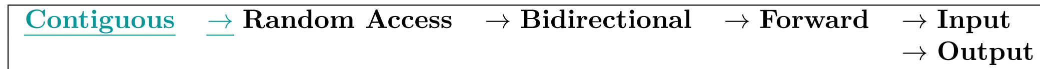
22.3.1 In general [iterator.requirements.general]

1 Iterators are a generalization of pointers that allow a C++ program to work with different data structures (for example, containers and ranges) in a uniform manner. To be able to construct template algorithms that work correctly and efficiently on different types of data structures, the library formalizes not just the interfaces but also the semantics and complexity assumptions of iterators. An input iterator `i` supports the expression `*i`, resulting in a value of some object type `T`, called the *value type* of the iterator. An output iterator `i` has a non-empty set of types that are *writable* to the iterator; for each such type `T`, the expression `*i = o` is valid where `o` is a value of type `T`. ~~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 74.

Table 74 — Relations among iterator categories



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

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

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

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

6 In addition to the requirements in this subclause, the nested *typedef-names* specified in 22.3.2.3 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. 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.

[*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> (22.3.2.3). — *end note*]

22.3.2 Associated types [iterator.assoc.types]

22.3.2.1 Incrementable traits [incrementable.traits]

1 To implement algorithms only in terms of incrementable types, it is often necessary to determine the difference type that corresponds to a particular incrementable type. Accordingly, it is required that if WI is the name of a type that models the WeaklyIncrementable concept (22.3.4.4), the type

```
iter_difference_t<WI>
```

be defined as the incrementable type's difference type.

2 iter_difference_t is implemented as if:

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

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

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.

```
  };

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

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

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

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

3 If `iterator_traits<I>` is a program-defined specialization, then `iter_difference_t<I>` denotes `iterator_traits<I>::difference_type`; otherwise, it denotes `incrementable_traits<I>::difference_type`.

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

22.3.2.2 Readable traits [readable.traits]

1 To implement algorithms only in terms of readable types, it is often necessary to determine the value type that corresponds to a particular readable type. Accordingly, it is required that if `R` is the name of a type that models the `Readable` concept (22.3.4.2), the type

```
iter_value_t<R>
```

be defined as the readable type's value type.

2 `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 readable_traits { };

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

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

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

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

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

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

3 If `iterator_traits<I>` is a program-defined specialization, then `iter_value_t<I>` denotes `iterator_traits<I>::value_type`; otherwise, it denotes `readable_traits<I>::value_type`.

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

5 [*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*]

6 [*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: Relocate [iterator.traits] here and modify as follows:]

22.3.2.3 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<IteratorI>::iterator_category
```

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

```
iterator_traits<IteratorI>::reference
iterator_traits<IteratorI>::pointer
```

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

```
iterator_traits<IteratorI>::difference_type
iterator_traits<IteratorI>::value_type
iterator_traits<IteratorI>::reference
iterator_traits<Iterator>::pointer
```

may be defined as `void`.

2 The member types of non-program-defined specializations of `iterator_traits` are computed as defined below. The definition below uses several exposition-only concepts equivalent to the following:

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

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

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

If the *qualified-id* `I::iterator_category` is valid and denotes 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, program-defined 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 22.3.4. If specified, it should be an alias for one of the standard iterator tag types (22.4.1), 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>
    requires is_object_v<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*]

22.3.3 Customization points [iterator.custpoints]

22.3.3.1 `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) — `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, `*E`.

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

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

22.3.3.2 `iter_swap` [iterator.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(I1, I2) = delete;
```

(1.2) — Otherwise, if the types of `E1` and `E2` both model `Readable`, and if the reference type of `E1` is swappable with (17.4.11) the reference type of `E2`, then `ranges::swap(*E1, *E2)`

(1.3) — Otherwise, if the types `T1` and `T2` of `E1` and `E2` model `IndirectlyMovableStorable<T1, T2>` and `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_t<remove_reference_t<X>> iter-exchange-move(X&& x, Y&& y)
    noexcept(see below noexcept(iter_value_t<remove_reference_t<X>>(iter_move(x))) &&
     noexcept(*x = iter_move(y)));
```

4 *Effects:* Equivalent to:

```
iter_value_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_t<T1>, iter_rvalue_reference_t<T1>> &&
is_nothrow_assignable_v<iter_value_t<T1>&, iter_rvalue_reference_t<T1>> &&
is_nothrow_assignable_v<iter_reference_t<T1>, iter_rvalue_reference_t<T2>> &&
is_nothrow_assignable_v<iter_reference_t<T1>, iter_value_t<T2>> &&
is_nothrow_move_constructible_v<iter_value_t<T1>> &&
noexcept(ranges::iter_move(declval<T1&>()))
```

22.3.4 Iterator concepts [iterator.concepts]

22.3.4.1 General [iterator.concepts.general]

1 Many of the concepts defined in this subclause (22.3.4) use the exposition-only type function *`ITER_CONCEPT`* in their specifications.

2 For a type `I`, let *`ITER_TRAITS`*`(I)` denote the type `I` if `iterator_traits<I>` names an instantiation of the primary template. Otherwise, *`ITER_TRAITS`*`(I)` denotes `iterator_traits<I>`.

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

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

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

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

22.3.4.2 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 Readable =
    requires {
      typename iter_value_t<In>;
      typename iter_reference_t<In>;
      typename iter_rvalue_reference_t<In>;
    } &&
    CommonReference<iter_reference_t<In>&&, iter_value_t<In>&> &&
    CommonReference<iter_reference_t<In>&&, iter_rvalue_reference_t<In>&&> &&
    CommonReference<iter_rvalue_reference_t<In>&&, const iter_value_t<In>&>;
```

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

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

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

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

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

(2.1) — If `Readable<Out> && Same<iter_value_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*]

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

2 Let `i` be an object of type `I`. When `i` is in the domain of both pre- and post-increment, `i` is said to be *incrementable*. `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*]

22.3.4.5 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 Incrementable =
    Regular<I> &&
    WeaklyIncrementable<I> &&
    requires(I i) {
      i++; requires Same<decltype(i++), I>;
    };
```

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

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

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

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

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

1 The `Iterator` concept forms the basis of the iterator concept taxonomy; every iterator 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 (22.3.4.7), to read (22.3.4.9) or write (22.3.4.10) values, or to provide a richer set of iterator movements (22.3.4.11, 22.3.4.12, 22.3.4.13).)

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

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

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

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

(2.1) — `i == s` is well-defined.

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

3 The domain of `==` 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.

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

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

```
template<class S, class I>
  concept SizedSentinel =
    Sentinel<S, I> &&
    !disable_sized_sentinel<remove_cv_t<S>, remove_cv_t<I>> &&
    requires(const I& i, const S& s) {
      s - i; requires Same<decltype(s - i), iter_difference_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`. `S` and `I` model `SizedSentinel<S, I>` only if

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

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

3 [*Note:* `disable_sized_sentinel` 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 model `SizedSentinel` is ill-formed with no diagnostic required ([structure.requirements]). — *end note*]

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

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

1 The `InputIterator` concept is a refinement of `Iterator` (22.3.4.6). It defines requirements for a type whose referenced values can be read (from the requirement for `Readable` (22.3.4.2)) and which can be both pre- and post-incremented. [*Note:* Unlike the input iterator requirements in 22.3.5.2, the `InputIterator` concept does not require equality comparison. — *end note*]

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

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

1 The `OutputIterator` concept is a refinement of `Iterator` (22.3.4.6). It defines requirements for a type that can be used to write values (from the requirement for `Writable` (22.3.4.3)) and which can be both pre- and post-incremented. However, output iterators are not required to satisfy `EqualityComparable`.

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

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

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

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

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

1 The `ForwardIterator` concept refines `InputIterator` (22.3.4.9), adding equality comparison and the multi-pass guarantee, specified below.

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

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

3 Pointers and references obtained from a forward iterator into a range `[i, s)` shall remain valid while `[i, s)` continues to denote a range.

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

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

(4.2) — The expression `([](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*]

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

1 The `BidirectionalIterator` concept refines `ForwardIterator` (22.3.4.11), and adds the ability to move an iterator backward as well as forward.

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

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

1 The `RandomAccessIterator` concept refines `BidirectionalIterator` (22.3.4.12) 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 RandomAccessIterator =
    BidirectionalIterator<I> &&
    DerivedFrom<ITER_CONCEPT(I), random_access_iterator_tag> &&
    StrictTotallyOrdered<I> &&
    SizedSentinel<I, I> &&
    requires(I i, const I j, const iter_difference_t<I> n) {
      { i += n } -> Same<I>&;
```

```
    j + n; requires Same<decltype(j + n), I>;
    n + j; requires Same<decltype(n + j), I>;
    { i -= 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_t<I>` such that after `n` applications of `++a`, then `bool(a == b)`. `I` models `RandomAccessIterator` only if

(2.1) — `(a += n)` is equal to `b`.

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

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

(2.4) — For any two positive 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)` is `true`.

22.3.4.14 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<ITER_CONCEPT(I), contiguous_iterator_tag> &&
    is_lvalue_reference_v<iter_reference_t<I>> &&
    Same<iter_value_t<I>, remove_cvref_t<iter_reference_t<I>>>;
```

2 Let `a` and `b` be dereferenceable iterators of type `I` such that `b` is reachable from `a`. `I` models `ContiguousIterator` only if

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

is `true`.

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

1 In the following sections, `a` and `b` denote values of type `X` or `const X`, `difference_type` and `reference` refer to the types `iterator_traits<X>::difference_type` and `iterator_traits<X>::reference`, respectively, `n` denotes a value of `difference_type`, `u`, `tmp`, and `m` denote identifiers, `r` denotes a value of `X&`, `t` denotes a value of value type `T`, `o` denotes a value of some type that is writable to the output iterator. [*Note:* For an iterator type `X` there must be an instantiation of `iterator_traits<X>` (22.3.2.3). — *end note*]

[Editor's note: Relocate [iterator.iterators] here:]

22.3.5.1 *Cpp17Iterator* [iterator.iterators]

[...]

[Editor's note: Relocate [input.iterators] here:]

22.3.5.2 Input iterators [input.iterators]

[...]

[Editor's note: Relocate [output.iterators] here:]

22.3.5.3 Output iterators [output.iterators]

[...]

[Editor's note: Relocate [forward.iterators] here:]

22.3.5.4 Forward iterators [forward.iterators]

[...]

[Editor's note: Relocate [bidirectional.iterators] here:]

22.3.5.5 Bidirectional iterators [bidirectional.iterators]

[...]

[Editor's note: Relocate [random.access.iterators] here:]

22.3.5.6 Random access iterators [random.access.iterators]

[...]

22.3.6 Indirect callable requirements [indirectcallable]

22.3.6.1 General [indirectcallable.general]

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

22.3.6.2 Indirect callables [indirectcallable.indirectinvocable]

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

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

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

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

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

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

22.3.6.3 Class template `projected` [projected]

1 Class template `projected` is intended for use when specifying the constraints of algorithms that accept callable objects and projections (15.3.18). 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_cvref_t<indirect_result_t<Proj&, I>>;
    indirect_result_t<Proj&, I> operator*() const;
  };

  template<WeaklyIncrementable I, class Proj>
  struct incrementable_traits<projected<I, Proj>> {
    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*]

22.3.7 Common algorithm requirements [commonalgoreq]

22.3.7.1 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::less<>` (19.14.8) function object type used in the concepts below imposes constraints on their arguments in addition to those that appear explicitly in the concepts' bodies. The function call operator of `ranges::less<>` requires its arguments to model `StrictTotallyOrderedWith` ([concept.stricttotallyordered]). — *end note*]

22.3.7.2 Concept `IndirectlyMovable` [commonalgoreq.indirectlymovable]

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

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

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

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

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

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

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

22.3.7.3 Concept `IndirectlyCopyable` [commonalgoreq.indirectlycopyable]

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

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

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

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

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

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

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

22.3.7.4 Concept `IndirectlySwappable` [commonalgoreq.indirectlyswappable]

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

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

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

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

22.3.7.6 Concept Permutable [commonalgoreq.permutable]

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

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

22.3.7.7 Concept Mergeable [commonalgoreq.mergeable]

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

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

22.3.7.8 Concept Sortable [commonalgoreq.sortable]

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

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

22.4 Iterator primitives [iterator.primitives]

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

22.4.1 Standard iterator tags [std.iterator.tags]

1 It is often desirable for a function template specialization to find out what is the most specific category of its iterator argument, so that the function can select the most efficient algorithm at compile time. To facilitate this, the library introduces *category tag* classes which are used as compile time tags for algorithm selection. They are: `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_concept` may be used to opt in or out of conformance to the iterator concepts defined in (22.3.4).

```
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 { };
}
```

[...]

22.4.2 Iterator operations [iterator.operations]

[...]

22.4.3 Range iterator operations [range.iterator.operations]

1 Since only types that model `RandomAccessIterator` provide the `+` operator, and types that model `SizedSentinel` provide the `-` operator, the library provides function templates `advance`, `distance`, `next`, and `prev`. These function templates use `+` and `-` for random access iterators and ranges that model `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*]

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

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

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

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

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

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

4 *Effects:*

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

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

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

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

5 *Expects:* If `n > 0`, `[i, bound)` 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, `I` shall model `BidirectionalIterator`, and `I` and `S` shall model `Same<I, S>`.

6 *Effects:*

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

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

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

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

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

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

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

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

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

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

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

```
return ranges::size(r); // 23.5.1
```

Otherwise, equivalent to:

```
return ranges::distance(ranges::begin(r), ranges::end(r)); // 23.4
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

22.5 Iterator adaptors [predef.iterators]

22.5.1 Reverse iterators [reverse.iterators]

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

22.5.1.1 Class template `reverse_iterator` [reverse.iterator]

```
namespace std {
  template<class Iterator>
  class reverse_iterator {
  public:
```

```
    using iterator_type     = Iterator;
    using iterator_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 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           = typename iterator_traits<Iterator>::pointer;
    using reference         = typename iterator_traits<Iterator>::reference;
    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 requires see below;

    [...]

    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 derived from `random_access_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.

22.5.1.2 `reverse_iterator` requirements [reverse.iter.requirements]

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

[2] Additionally, `Iterator` shall ~~satisfy~~either meet the requirements of a *Cpp17RandomAccessIterator* (22.3.5.6) or model `RandomAccessIterator` (22.3.4.13) if any of the members `operator+`, `operator-`, `operator+=`, `operator-=` (22.5.1.6), `operator[]` (22.5.1.5), or the non-member operators (22.5.1.7) `operator<`, `operator>`, `operator<=`, `operator>=`, `operator-`, or `operator+` (22.5.1.8) are referenced in a way that requires instantiation ([temp.inst]).

22.5.1.5 `reverse_iterator` element access [reverse.iter.elem]

[...]

[Editor's note: This change incorporates the PR of LWG 1052):]

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

2 ~~*Returns:* addressof(operator*()).~~ *Effects:*

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

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

[...]

22.5.1.6 `reverse_iterator` navigation [reverse.iter.nav]

[...]

22.5.1.7 `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 and implicitly convertible to `bool`.

[...]

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

[...]

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

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

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

4 *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);
```

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

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

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

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

[...]

22.5.2 Insert iterators [insert.iterators]

[...]

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

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

```
  public:
    using iterator_category = output_iterator_tag;
    using value_type        = void;
    using difference_type   = voidptrdiff_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);
}
```

[...]

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

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

  public:
    using iterator_category = output_iterator_tag;
    using value_type        = void;
    using difference_type   = voidptrdiff_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);
}
```

[...]

22.5.2.3 Class template `insert_iterator` [insert.iterator]

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

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

    insert_iterator() = default;
    insert_iterator(Container& x, typename Container::iteratoriterator_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::iteratoriterator_t<Container> i);
}
```

22.5.2.3.1 insert_iterator operations [insert.iter.ops]

```
insert_iterator(Container& x, typename Container::iteratoriterator_t<Container> i);
```

[...]

22.5.2.3.2 inserter [inserter]

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

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

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

22.5.3 Move iterators and sentinels [move.iterators]

[...]

22.5.3.1 Class template move_iterator [move.iterator]

```
namespace std {
  template<class Iterator>
  class move_iterator {
  public:
    using iterator_type     = Iterator;
    using iterator_category = typename iterator_traits<Iterator>::iterator_categorysee below;
    using value_type        = typename iterator_traits<Iterator>::value_typeiter_value_t<Iterator>;
    using difference_type   = typename iterator_traits<Iterator>::difference_typeiter_difference_t<Iterator>;
    using pointer           = Iterator;
    using reference         = see belowiter_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_iteratordecltype(auto) operator++(int);
    constexpr move_iterator& operator--();

    [...]
```

```
    constexpr move_iterator operator-(difference_type n) const;
    constexpr move_iterator& operator-=(difference_type n);
    constexpr unspecifiedreference operator[](difference_type n) const;

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

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

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

  private:
    Iterator current;     // exposition only
  };

  [...]

  template<class Iterator>
    constexpr move_iterator<Iterator> operator+(
      typename move_iterator<Iterator>::difference_typeiter_difference_t<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*.~~

2 The member *typedef-name* `iterator_category` denotes `random_access_iterator_tag` if `iterator_traits<Iterator>::iterator_category` is derived from `random_access_iterator_tag`, and `iterator_traits<Iterator>::iterator_category` otherwise.

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

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

[...]

22.5.3.5 `move_iterator` element access [move.iter.elem]

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

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

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

[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;
```

2 *Returns:* `current`.

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

3 ~~*Returns:* std::move(current[n]).~~

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

22.5.3.6 `move_iterator` navigation [move.iter.nav]

[...]

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`.

[...]

22.5.3.7 `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)`.

[...]

22.5.3.8 `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()`.

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

3 *Returns:* `x + n`.

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

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

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

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

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

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

22.5.3.9 Class template `move_sentinel` [move.sentinel]

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

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

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

— *end example*]

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

    constexpr S base() const;

  private:
    S last; // exposition only
  };
}
```

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

```
constexpr move_sentinel();
```

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

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

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

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

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

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

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

22.5.4 Common iterators [iterators.common]

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*]

22.5.4.1 Class template `common_iterator` [common.iterator]

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

    constexpr common_iterator() = default;
    constexpr common_iterator(I i);
    constexpr common_iterator(S s);
    template<ConvertibleTo<I> I2, ConvertibleTo<S> S2>
      constexpr common_iterator(const common_iterator<I2, S2>& x);

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

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

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

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

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

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

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

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

  template<InputIterator I, class S>
  struct iterator_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 iterator_category = see below;
    using iterator_concept = see below;
  };
}
```

22.5.4.2 `iterator_traits` for `common_iterator` [common.iterator.traits]

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

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

3 Let `C` denote the type `iterator_traits<I>::iterator_category`. If `C` models `DerivedFrom<forward_iterator_tag>`, `iterator_category` denotes `forward_iterator_tag`. Otherwise, `iterator_category` denotes `input_iterator_tag`.

4 `iterator_concept` denotes `forward_iterator_tag` if `I` models `ForwardIterator`; otherwise it denotes `input_iterator_tag`.

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

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

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

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

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

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

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

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

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

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

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

6 *Effects:* Equivalent to:

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

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

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

7 *Returns:* `*this`

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

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

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

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

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

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

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

2 *Effects:*

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

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

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

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

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

3 The expression in the requires clause is equivalent to:

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

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

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

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

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

3 *Returns:* `*this`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
    requires InputIterator<I>;
```

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

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

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

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

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

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

22.5.5 Default sentinels [default.sentinels]

22.5.5.1 Class `default_sentinel` [default.sent]

```
namespace std {
  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` (22.5.6.1)).

22.5.6 Counted iterators [iterators.counted]

22.5.6.1 Class template `counted_iterator` [counted.iterator]

1 Class template `counted_iterator` is an iterator adaptor with the same behavior as the underlying iterator except that it keeps track of 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;
// 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 tcodecounted_iterator<I2> refer to elements of the same sequence if and only if `next(i1.base(), i1.count())` and `next(i2.base(), i2.count())` refer to the same (possibly past-the-end) element.

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

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

    constexpr I base() const;
    constexpr iter_difference_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>;
friend constexpr counted_iterator operator+(
  iter_difference_t<I> n, const counted_iterator& x)
    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<Common<I> I2>
  friend constexpr iter_difference_t<I2> operator-(
    const counted_iterator& x, const counted_iterator<I2>& y);
friend constexpr iter_difference_t<I> operator-(
  const counted_iterator& x, default_sentinel);
friend constexpr iter_difference_t<I> operator-(
  default_sentinel, const counted_iterator& y);
constexpr counted_iterator& operator-=(difference_type n)
  requires RandomAccessIterator<I>;

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

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

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

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

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

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

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

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

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

22.5.6.2 counted_iterator operations [counted.iter.ops]

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

```
constexpr counted_iterator();
```

1 *Effects:* Value-initializes `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_t<I> n);
```

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

3 *Effects:* Initializes `current` with `i` and `cnt` with `n`.

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

4 *Effects:* Initializes `current` with `x.current` and `cnt` with `x.cnt`.

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

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

22.5.6.2.2 counted_iterator conversion [counted.iter.op.conv]

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

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

22.5.6.2.3 counted_iterator count [counted.iter.op.cnt]

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

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

22.5.6.2.4 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;`

22.5.6.2.5 counted_iterator::operator++ [counted.iter.op.incr]

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

1 *Expects:* `cnt > 0`.

2 *Effects:* Equivalent to:

```
++current;
--cnt;
```

3 *Returns:* `*this`.

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

4 *Expects:* `cnt > 0`.

5 *Effects:* Equivalent to:

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

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

6 *Expects:* `cnt > 0`.

7 *Effects:* Equivalent to:

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

22.5.6.2.6 `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;
```

22.5.6.2.7 `counted_iterator::operator+` [counted.iter.op.+]

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

1 *Expects:* `n <= cnt`.

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

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

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

22.5.6.2.8 `counted_iterator::operator+=` [counted.iter.op.+=]

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

1 *Expects:* `n <= cnt`.

2 *Effects:* Equivalent to:

```
current += n;
cnt -= n;
```

3 *Returns:* `*this`.

22.5.6.2.9 `counted_iterator::operator-` [counted.iter.op.-]

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

1 *Expects:* `-n <= cnt`.

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

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

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

4 *Effects:* Equivalent to: `return y.cnt - x.cnt;`

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

5 *Effects:* Equivalent to: `return -x.cnt;`

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

6 *Effects:* Equivalent to: `return y.cnt;`

22.5.6.2.10 `counted_iterator::operator-=` [counted.iter.op.-=]

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

1 *Expects:* `-n <= cnt`.

2 *Effects:* Equivalent to:

```
current -= n;
cnt += n;
```

3 *Returns:* `*this`.

22.5.6.2.11 `counted_iterator::operator[]` [counted.iter.op.index]

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

1 *Expects:* `n <= cnt`.

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

22.5.6.2.12 `counted_iterator` comparisons [counted.iter.op.comp]

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

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

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

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

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

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

```
friend constexpr bool operator!=(
  default_sentinel x, const counted_iterator& y);
```

4 *Expects:* For the first overload, `x` and `y` shall refer to elements of the same sequence (22.5.6.1).

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

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

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

7 *Effects:* Equivalent to: `return y.cnt < x.cnt;`

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

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

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

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

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

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

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

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

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

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

22.5.6.2.13 `counted_iterator` customizations [counted.iter.nonmember]

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

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

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

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

22.5.7 Unreachable sentinel [unreachable.sentinels]

22.5.7.1 Class `unreachable` [unreachable.sentinel]

[Editor's note: This wording integrates the PR for stl2#507):]

1 Class `unreachable` is a placeholder type that can be used with any `WeaklyIncrementable` type to denote the "upper bound" of an open interval. Comparing 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 {
  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;
  };
}
```

22.5.7.2 unreachable operations [unreachable.sentinel.ops]

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

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

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

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

22.6 Stream iterators [stream.iterators]

[...]

22.6.1 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:
    [...]
  };

  [...]
}
```

22.6.1.1 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`.

[...]

22.6.1.2 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)`

22.6.2 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   = voidptrdiff_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
  };
}
```

[...]

22.6.3 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
  };

  [...]
}
```

[...]

22.6.3.2 istreambuf_iterator constructors [istreambuf.iterator.cons]

[...]

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

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

[...]

22.6.3.3 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)`.

22.6.4 Class template ostreambuf_iterator [ostreambuf.iterator]

```
namespace std {
  template<class charT, class traits = char_traits<charT>>
  class ostreambuf_iterator {
  public:
    using iterator_category = output_iterator_tag;
    using value_type        = void;
```

```
    using difference_type   = 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:]

23 Ranges library [range]

23.1 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 75.

Table 75 — Ranges library summary

	Subclause	Header(s)
23.4	Range access	`<ranges>`
23.5	Range primitives	
23.6	Requirements	
23.7	Range utilities	
23.8	Range adaptors	

23.2 decay_copy [range.decaycopy]

1 Several places in this clause use the expression *DECAY_COPY*(x), which is expression-equivalent to:

```
decay_t<decltype((x))>(x)
```

23.3 Header <ranges> synopsis [range.syn]

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

namespace std {
  namespace ranges {
    inline namespace unspecified {
      // 23.4, 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;
```

```
  // 23.5, 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 { };

template<Rangefowarding-range R>
  using safe_iterator_t = decltype(ranges::begin(declval<R>()))iterator_t<R>;

// 23.6, range requirements

// 23.6.2, Range
template<class T>
  concept Range = see below;

// 23.6.3, SizedRange
template<class T>
  concept SizedRange = see below;

// 23.6.4, View
template<class T>
  concept View = see below;

// 23.6.5, CommonRange
template<class T>
  concept CommonRange = see below;

// 23.6.6, InputRange
template<class T>
  concept InputRange = see below;

// 23.6.7, OutputRange
template<class R, class T>
  concept OutputRange = see below;

// 23.6.8, ForwardRange
template<class T>
  concept ForwardRange = see below;

// 23.6.9, BidirectionalRange
template<class T>
  concept BidirectionalRange = see below;

// 23.6.10, RandomAccessRange
template<class T>
  concept RandomAccessRange = see below;
```

```
// 23.6.11, ContiguousRange
template<class T>
  concept ContiguousRange = see below;

// 23.6.12
template<class T>
  concept ViewableRange = see below;

// 23.7.1
template<class D>
  requires is_class_v<D>
class view_interface;

// 23.7.2.1
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;

template<forwarding-range R>
  using safe_subrange_t = subrange<iterator_t<R>>;

// 23.8.4
namespace view { inline constexpr unspecified all = unspecified; }

template<ViewableRange R>
  using all_view = decltype(view::all(declval<R>()));

// 23.8.5
template<InputRange R, IndirectUnaryPredicate<iterator_t<R>> Pred>
  requires View<R>
class filter_view;

namespace view { inline constexpr unspecified filter = unspecified; }

// 23.8.7
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; }

// 23.8.9
template<WeaklyIncrementable I, Semiregular Bound = unreachable>
  requires weakly-equality-comparable-with<I, Bound>
class iota_view;

namespace view { inline constexpr unspecified iota = unspecified; }

// 23.8.11
template<View> class take_view;

namespace view { inline constexpr unspecified take = unspecified; }

// 23.8.13
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_t<iterator_t<R>>>)
class join_view;

namespace view { inline constexpr unspecified join = unspecified; }
```

```
    // 23.8.15
    template<class T>
      requires is_object_v<T>
    class empty_view;

    namespace view {
      template<class T>
        inline constexpr empty_view<T> empty {};
    }

    // 23.8.16
    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;

    // 23.8.18
    template<InputRange R, ForwardRange Pattern>
      requires View<R> && View<Pattern> &&
          IndirectlyComparable<iterator_t<R>, iterator_t<Pattern>, ranges::equal_to<>> &&
          (ForwardRange<R> || tiny-range<Pattern>)
    class split_view;

    namespace view { inline constexpr unspecified split = unspecified; }

    // 23.8.20
    namespace view { inline constexpr unspecified counted = unspecified; }

    // 23.8.21
    template<View R>
      requires !CommonRange<R>
    class common_view;

    namespace view { inline constexpr unspecified common = unspecified; }

    // 23.8.23
    template<View R>
      requires BidirectionalRange<R>
    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;
  };
}
```

23.4 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 `<ranges>` header, the customization point objects in 23.4 are available when `<iterator>` is included.

23.4.1 `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) — `(E) + 0` if `E` is an lvalue of array type ([basic.compound]).

(1.2) — Otherwise, if `E` is an lvalue, *`DECAY_COPY`*`((E).begin())` if it is a valid expression and its type `I` models `Iterator`.

(1.3) — Otherwise, *`DECAY_COPY`*`(begin(E))` if it is a valid expression and its type `I` models `Iterator` with overload resolution performed in a context that includes the declarations:

```
template<class T> void begin(T&&) = delete;
template<class T> void begin(initializer_list<T>&&) = delete;
```

and does not include a declaration of `ranges::begin`.

(1.4) — Otherwise, `ranges::begin(E)` is ill-formed.

2 [*Note:* Whenever `ranges::begin(E)` is a valid expression, its type models `Iterator`. — *end note*]

23.4.2 `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) — `(E) + extent_v<T>` if `E` is an lvalue of array type ([basic.compound]) `T`.

(1.2) — Otherwise, if `E` is an lvalue, *`DECAY_COPY`*`((E).end())` if it is a valid expression and its type `S` models `Sentinel<decltype(ranges::begin(E))>`.

(1.3) — Otherwise, *`DECAY_COPY`*`(end(E))` if it is a valid expression and its type `S` models `Sentinel<decltype( ranges::begin(E))>` with overload resolution performed in a context that includes the declarations:

```
template<class T> void end(T&&) = delete;
template<class T> void end(initializer_list<T>&&) = delete;
```

and does not include a declaration of `ranges::end`.

(1.4) — 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)` model `Sentinel`. — *end note*]

23.4.3 `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 [*Note:* Whenever `ranges::cbegin(E)` is a valid expression, its type models `Iterator`. — *end note*]

23.4.4 `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 [*Note:* Whenever `ranges::cend(E)` is a valid expression, the types of `ranges::cend(E)` and `ranges::cbegin(E)` model `Sentinel`. — *end note*]

23.4.5 `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) — If `E` is an lvalue, *`DECAY_COPY`*`((E).rbegin())` if it is a valid expression and its type `I` models `Iterator`.

(1.2) — 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`.

(1.3) — 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 `BidirectionalIterator` (22.3.4.12).

(1.4) — Otherwise, `ranges::rbegin(E)` is ill-formed.

2 [*Note:* Whenever `ranges::rbegin(E)` is a valid expression, its type models `Iterator`. — *end note*]

23.4.6 `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) — If `E` is an lvalue, *`DECAY_COPY`*`((E).rend())` if it is a valid expression and its type `S` models `Sentinel<decltype(ranges::rbegin(E))>`.

(1.2) — 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`.

(1.3) — 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 models `BidirectionalIterator` (22.3.4.12).

(1.4) — 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)` model `Sentinel`. — *end note*]

23.4.7 `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 [*Note:* Whenever `ranges::crbegin(E)` is a valid expression, its type models `Iterator`. — *end note*]

23.4.8 `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 [*Note:* Whenever `ranges::crend(E)` is a valid expression, the types of `ranges::crend(E)` and `ranges::crbegin(E)` model `Sentinel`. — *end note*]

23.5 Range primitives [range.primitives]

1 In addition to being available via inclusion of the `<ranges>` header, the customization point objects in 23.5 are available when `<iterator>` is included.

23.5.1 `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>)` if `T` is an array type ([basic.compound]).

(1.2) — Otherwise, *`DECAY_COPY`*`(E.size())` if it is a valid expression and its type `I` models `Integral` and `disable_sized_range<remove_cvref_t<T>>` (23.6.3) is `false`.

(1.3) — Otherwise, *`DECAY_COPY`*`(size(E))` if it is a valid expression and its type `I` models `Integral` with overload resolution performed in a context that includes the declaration:

```
template<class T> void size(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::end(E) - ranges::begin(E))`, except that `E` is only evaluated once, if it is a valid expression and the types `I` and `S` of `ranges::begin(E)` and `ranges::end(E)` model `SizedSentinel<S, I>` (22.3.4.8) and `ForwardIterator<I>`.

(1.5) — Otherwise, `ranges::size(E)` is ill-formed.

2 [*Note:* Whenever `ranges::size(E)` is a valid expression, its type models `Integral`. — *end note*]

23.5.2 `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)` 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*]

23.5.3 `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) — If `E` is an lvalue, *`DECAY_COPY`*`((E).data())` if it is a valid expression of pointer to object type.

(1.2) — Otherwise, `ranges::begin(E)` if it is a valid expression of pointer to object type.

(1.3) — 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.

(1.4) — 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*]

23.5.4 `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 [*Note:* Whenever `ranges::cdata(E)` is a valid expression, it has pointer to object type. — *end note*]

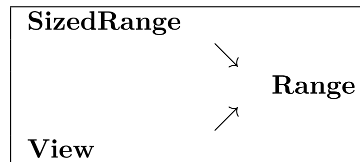
23.6 Range requirements [range.requirements]

23.6.1 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 (22.3.4.6) and a sentinel (22.3.4.7). 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 76.

Table 76 — Relations among range categories



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. *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 models `RandomAccessIterator` (22.3.4.13). The range categories *bidirectional ranges*, *forward ranges*, *input ranges*, and *output ranges* are defined similarly.

23.6.2 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 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 expression `E` such that `decltype((E))` is `T`, `T` models *`range-impl`* only if

(2.1) — `[ranges::begin(E), ranges::end(E))` denotes a range.

(2.2) — Both `ranges::begin(E)` and `ranges::end(E)` are amortized constant time and non-modifying. [*Note:* `ranges::begin(E)` and `ranges::end(E)` do not require implicit expression variations ([concepts.equality]). — *end note*]

(2.3) — If the type of `ranges::begin(E)` models `ForwardIterator`, `ranges::begin(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 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*]

23.6.3 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 SizedRange =
    Range<T> &&
    !disable_sized_range<remove_cvref_t<T>> &&
    requires(T& t) {
      { ranges::size(t) } -> ConvertibleTo<iter_difference_t<iterator_t<T>>>;
    };
```

2 Given an lvalue `t` of type `remove_reference_t<T>`, `T` models `SizedRange` 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>` 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_cvref_t<R>>` evaluates to `true` ([structure.requirements]). — *end note*]

23.6.4 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 21) 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 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` is valid and denotes a type ([temp.deduct]), `enable_view<T>::type::value`;

(4.2) — Otherwise, if `DerivedFrom<T, view_base>` is `true`, `true`;

(4.3) — Otherwise, if `T` is a 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`.

23.6.5 Common ranges [range.common]

1 The `CommonRange` concept specifies requirements of a `Range` type for which `begin` and `end` return objects of the same type. [*Note:* The standard containers ([containers]) model `CommonRange`. — *end note*]

```
template<class T>
  concept CommonRange =
    Range<T> && Same<iterator_t<T>, sentinel_t<T>>;
```

23.6.6 Input ranges [range.input]

1 The `InputRange` concept specifies requirements of a `Range` type for which `begin` returns a type that models `InputIterator` (22.3.4.9).

```
template<class T>
  concept InputRange =
    Range<T> && InputIterator<iterator_t<T>>;
```

23.6.7 Output ranges [range.output]

1 The `OutputRange` concept specifies requirements of a `Range` type for which `begin` returns a type that models `OutputIterator` (22.3.4.10).

```
template<class R, class T>
  concept OutputRange =
    Range<R> && OutputIterator<iterator_t<R>, T>;
```

23.6.8 Forward ranges [range.forward]

1 The `ForwardRange` concept specifies requirements of an `InputRange` type for which `begin` returns a type that models `ForwardIterator` (22.3.4.11).

```
template<class T>
  concept ForwardRange =
    InputRange<T> && ForwardIterator<iterator_t<T>>;
```

23.6.9 Bidirectional ranges [range.bidirectional]

1 The `BidirectionalRange` concept specifies requirements of a `ForwardRange` type for which `begin` returns a type that models `BidirectionalIterator` (22.3.4.12).

```
template<class T>
  concept BidirectionalRange =
    ForwardRange<T> && BidirectionalIterator<iterator_t<T>>;
```

23.6.10 Random access ranges [range.random.access]

1 The `RandomAccessRange` concept specifies requirements of a `BidirectionalRange` type for which `begin` returns a type that models `RandomAccessIterator` (22.3.4.13).

```
template<class T>
  concept RandomAccessRange =
    BidirectionalRange<T> && RandomAccessIterator<iterator_t<T>>;
```

23.6.11 Contiguous ranges [range.contiguous]

1 The `ContiguousRange` concept specifies requirements of a `RandomAccessRange` type for which `begin` returns a type that models `ContiguousIterator` (22.3.4.14).

```
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>>>>;
    };
```

23.6.12 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> && (forwarding-range<T> || View<decay_t<T>>);
```

23.7 Range utilities [range.utility]

1 The components in this subclause are general utilities for representing and manipulating ranges.

23.7.1 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 {
  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>>;

    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_t<iterator_t<R>> n);
      template<RandomAccessRange R = const D>
        constexpr decltype(auto) operator[](iter_difference_t<iterator_t<R>> n) const;

      template<ForwardRange C>
          requires !View<C> &&
            ConvertibleTo<iter_reference_t<iterator_t<const D>>,
              iter_value_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.

23.7.1.1 `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>>;
```

3 *Effects:* Equivalent to:

```
return ranges::empty(derived()) ? nullptr : addressof(*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 *Expects:* `!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 *Expects:* `!empty()`.

8 *Effects:* Equivalent to: `return *ranges::prev(ranges::end(derived()));`

```
template<RandomAccessRange R = D>
  constexpr decltype(auto) operator[](iter_difference_t<iterator_t<R>> n);
```

```
template<RandomAccessRange R = const D>
  constexpr decltype(auto) operator[](iter_difference_t<iterator_t<R>> n) const;
```

9 *Effects:* Equivalent to: `return ranges::begin(derived())[n];`

```
template<ForwardRange C>
  requires !View<C> &&
    ConvertibleTo<iter_reference_t<iterator_t<const D>>,
      iter_value_t<iterator_t<C>>> &&
    Constructible<C, range-common-iterator<const D>,
      range-common-iterator<const D>>
operator C() const;
```

10 *Effects:* Equivalent to:

```
using I = range-common-iterator<const D>;
return C(I{ranges::begin(derived())}, I{ranges::end(derived())});
```

23.7.2 Sub-ranges [range.subranges]

1 The `subrange` class template bundles together an iterator and a sentinel into a single object that models the `View` concept. Additionally, it models the `SizedRange` concept when the final template parameter is `subrange_kind::sized`.

23.7.2.1 `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 =
        SizedSentinel<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 =  // exposition only
        K == subrange_kind::sized && !SizedSentinel<S, I>;
```

```
  I begin_ {}; // exposition only
  S end_ {};   // exposition only
  iter_difference_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_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_t<I> n)
    requires K == subrange_kind::sized;

  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_t<I> size() const
    requires K == subrange_kind::sized;

  [[nodiscard]] constexpr subrange next(iter_difference_t<I> n = 1) const;
  [[nodiscard]] constexpr subrange prev(iter_difference_t<I> n = 1) const
    requires BidirectionalIterator<I>;
  constexpr subrange& advance(iter_difference_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_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_t<tuple_element_t<0, P>>) ->
    subrange<tuple_element_t<0, P>, tuple_element_t<1, P>, subrange_kind::sized>;
```

```
    template<forwarding-range R>
      subrange(R&&) -> subrange<iterator_t<R>, sentinel_t<R>,
        (SizedRange<R> || SizedSentinel<sentinel_t<R>, iterator_t<R>>)
          ? subrange_kind::sized : subrange_kind::unsized>;

    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;
}
```

23.7.2.1.1 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_t<I> n)
  requires K == subrange_kind::sized;
```

2 *Expects:* `n == ranges::distance(i, s)`.

3 *Effects:* Initializes `begin_` with `i`, `end_` with `s`. If `StoreSize` is `true`, initializes `size_` with `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>;
```

4 *Effects:* Equivalent to:

(4.1) — If `StoreSize` is `true`, `subrange{ranges::begin(r), ranges::end(r), ranges::size(r)}`.

(4.2) — 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;
```

5 *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;
```

6 *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_t<I> n)
  requires K == subrange_kind::sized;
```

7 *Effects:* Equivalent to:

```
subrange{get<0>(std::forward<PairLike>(r)), get<1>(std::forward<PairLike>(r)), n}
```

23.7.2.1.2 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_);`

23.7.2.1.3 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:*

(4.1) — If `StoreSize` is `true`, equivalent to: `return size_;`

(4.2) — Otherwise, equivalent to: `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;
```

23.7.2.1.4 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();
```

23.8 Range adaptors [range.adaptors]

1 This subclause 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*]

23.8.1 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))` 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(range, args...)
adaptor(args...)(range)
range | adaptor(args...)
```

In this case, `adaptor(args...)` is a range adaptor closure object.

23.8.2 Semiregular wrapper [range.adaptor.semiregular_wrapper]

1 Many of the types in this subclause 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>)
  : semiregular{in_place}
{ }
```

(1.3) — If `Assignable<T&, const T&>` is not satisfied, the copy assignment operator is equivalent to:

```
semiregular& operator=(const semiregular& that)
  noexcept(is_nothrow_copy_constructible_v<T>)
{
  if (that) emplace(*that);
  else reset();
  return *this;
}
```

(1.4) — If `Assignable<T&, T>` is not satisfied, the move assignment operator is equivalent to:

```
semiregular& operator=(semiregular&& that)
  noexcept(is_nothrow_move_constructible_v<T>)
{
  if (that) emplace(std::move(*that));
  else reset();
  return *this;
}
```

23.8.3 Helper concepts [range.adaptor.helpers]

1 Many of the types in this subclause are specified in terms of the following exposition-only concepts:

```
template<class R>
  concept simple-view =
    View<R> && Range<const R> &&
    Same<iterator_t<R>, iterator_t<const R>> &&
    Same<sentinel_t<R>, sentinel_t<const R>>;

template<InputIterator I>
  concept has-arrow =
    is_pointer_v<I> || requires(I i) { i.operator->(); };
```

23.8.4 `view::all` [range.adaptors.all]

1 `view::all` returns a `View` that includes all elements of its `Range` argument.

2 The name `view::all` denotes a range adaptor object (23.8.1). The expression `view::all(E)` for some subexpression `E` is expression-equivalent to:

- (2.1) — *`DECAY_COPY`*`(E)` if the decayed type of `E` 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.
- (2.4) — Otherwise, `view::all(E)` is ill-formed.

[*Note:* Whenever `view::all(E)` is a valid expression, it is a prvalue whose type models `View`. — *end note*]

23.8.4.1 *ref-view* [range.view.ref]

```
namespace std::ranges {
  template<Range R>
    requires std::is_object_v<R>
  class ref_view : public view_interface<ref_view<R>> {
  private:
    R* r_ = nullptr; // exposition only
  public:
    constexpr ref_view() noexcept = default;
    constexpr ref_view(R& r) noexcept;

    constexpr R& base() const;

    constexpr iterator_t<R> begin() const
      noexcept(noexcept(ranges::begin(*r_)));
    constexpr sentinel_t<R> end() const
      noexcept(noexcept(ranges::end(*r_)));

    constexpr bool empty() const
      noexcept(noexcept(ranges::empty(*r_)))
      requires requires { ranges::empty(*r_); };

    constexpr auto size() const
      noexcept(noexcept(ranges::size(*r_)))
      requires SizedRange<R>;
```

```
    constexpr auto data() const
      noexcept(noexcept(ranges::data(*r_)))
      requires ContiguousRange<R>;

    friend constexpr iterator_t<R> begin(ref_view&& r)
      noexcept(noexcept(r.begin()));
    friend constexpr sentinel_t<R> end(ref_view&& r)
      noexcept(noexcept(r.end()));
  };
}
```

23.8.4.1.1 *ref-view* operations [range.view.ref.ops]

```
constexpr ref_view(R& r) noexcept;
```

1 *Effects:* Initializes `r_` with `addressof(r)`.

```
constexpr R& base() const;
```

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

```
constexpr iterator_t<R> begin() const
  noexcept(noexcept(ranges::begin(*r_)));
friend constexpr iterator_t<R> begin(ref_view&& r)
  noexcept(noexcept(r.begin()));
```

3 *Effects:* Equivalent to: `return ranges::begin(*r_);` or `return r.begin();`, respectively.

```
constexpr sentinel_t<R> end() const
  noexcept(noexcept(ranges::end(*r_)));
friend constexpr sentinel_t<R> end(ref_view&& r)
  noexcept(noexcept(r.end()));
```

4 *Effects:* Equivalent to: `return ranges::end(*r_);` or `return r.end();`, respectively.

```
constexpr bool empty() const
  noexcept(noexcept(ranges::empty(*r_)))
  requires requires { ranges::empty(*r_); };
```

5 *Effects:* Equivalent to: `return ranges::empty(*r_);`

```
constexpr auto size() const
  noexcept(noexcept(ranges::size(*r_)))
  requires SizedRange<R>;
```

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

```
constexpr auto data() const
  noexcept(noexcept(ranges::data(*r_)))
  requires ContiguousRange<R>;
```

7 *Effects:* Equivalent to: `return ranges::data(*r_);`

23.8.5 Class template `filter_view` [range.adaptors.filter_view]

1 `filter_view` presents a `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 {
  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<class R, class Pred>
    filter_view(R&&, Pred) -> filter_view<all_view<R>, Pred>;
}
```

23.8.5.1 filter_view operations [range.adaptors.filter__view.ops]

23.8.5.1.1 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)`.

23.8.5.1.2 filter_view conversion [range.adaptors.filter__view.conv]

```
constexpr R base() const;
```

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

23.8.5.1.3 filter_view range begin [range.adaptors.filter__view.begin]

```
constexpr iterator begin();
```

1 *Returns:* `{*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.

23.8.5.1.4 filter_view range end [range.adaptors.filter__view.end]

```
constexpr sentinel end();
```

1 *Effects:* Equivalent to: `return sentinel{*this};`

```
constexpr iterator end() requires CommonRange<R>;
```

2 *Effects:* Equivalent to: `return iterator{*this, ranges::end(base_)};`

23.8.5.2 Class template filter_view::iterator [range.adaptors.filter__view.iterator]

```
namespace std::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_t<iterator_t<R>>;
    using difference_type   = iter_difference_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(noexcept(ranges::iter_move(i.current_)));
    friend constexpr void iter_swap(const iterator& x, const iterator& y)
      noexcept(noexcept(ranges::iter_swap(x.current_, y.current_)))
      requires IndirectlySwappable<iterator_t<R>>;
  };
}
```

1 `iterator::iterator_category` is defined as follows:

- (1.1) — Let `C` denote the type `iterator_traits<iterator_t<R>>::iterator_category`.
- (1.2) — If `C` models `DerivedFrom<bidirectional_iterator_tag>`, then `iterator_category` denotes `bidirectional_iterator_tag`.
- (1.3) — Otherwise, if `C` models `DerivedFrom<forward_iterator_tag>`, then `iterator_category` denotes `forward_iterator_tag`.
- (1.4) — Otherwise, `iterator_category` denotes `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`.

23.8.5.2.1 `filter_view::iterator` operations [range.adaptors.filter_view.iterator.ops]

23.8.5.2.1.1 `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)`.

23.8.5.2.1.2 `filter_view::iterator` conversion [range.adaptors.filter_view.iterator.conv]

```
constexpr iterator_t<R> base() const;
```

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

23.8.5.2.1.3 `filter_view::iterator::operator*` [range.adaptors.filter_view.iterator.star]

```
constexpr iter_reference_t<iterator_t<R>> operator*() const;
```

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

23.8.5.2.1.4 `filter_view::iterator::operator->` [range.adaptors.filter_view.iterator.arrow]

```
constexpr iterator_t<R> operator->() const
  requires has-arrow<iterator_t<R>>;
```

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

23.8.5.2.1.5 `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 `++*this`.

```
constexpr iterator operator++(int) requires ForwardRange<R>;
```

3 *Effects:* Equivalent to:

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

23.8.5.2.1.6 `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;
```

23.8.5.2.1.7 `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 *Effects:* Equivalent to: `return x.current_ == y.current_;`

```
friend constexpr bool operator!=(const iterator& x, const iterator& y)
  requires EqualityComparable<iterator_t<R>>;
```

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

23.8.5.2.2 `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(noexcept(ranges::iter_move(i.current_)));
```

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

```
friend constexpr void iter_swap(const iterator& x, const iterator& y)
  noexcept(noexcept(ranges::iter_swap(x.current_, y.current_)))
  requires IndirectlySwappable<iterator_t<R>>;
```

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

23.8.5.3 Class template `filter_view::sentinel` [range.adaptors.filter_view.sentinel]

```
namespace std::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);
  };
}
```

23.8.5.3.1 `filter_view::sentinel` constructors [range.adaptors.filter_view.sentinel.ctor]

```
explicit constexpr sentinel(filter_view& parent);
```

1 *Effects:* Initializes `end_` with `ranges::end(parent)`.

23.8.5.3.2 `filter_view::sentinel` conversion [range.adaptors.filter_view.sentinel.conv]

```
constexpr sentinel_t<R> base() const;
```

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

23.8.5.3.3 `filter_view::sentinel` comparison [range.adaptors.filter_view.sentinel.comp]

```
friend constexpr bool operator==(const iterator& x, const sentinel& y);
```

1 *Effects:* Equivalent to: `return x.current_ == y.end_;`

```
friend constexpr bool operator==(const sentinel& x, const iterator& y);
```

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

```
friend constexpr bool operator!=(const iterator& x, const sentinel& y);
```

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

```
friend constexpr bool operator!=(const sentinel& x, const iterator& y);
```

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

23.8.6 `view::filter` [range.adaptors.filter]

1 The name `view::filter` denotes a range adaptor object (23.8.1). The expression `view::filter(E, P)` for subexpressions `E` and `P` is expression-equivalent to `filter_view{E, P}`.

23.8.7 Class template `transform_view` [range.adaptors.transform_view]

1 `transform_view` presents a `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 {
  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> struct iterator;      // exposition only
    template<bool> 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>;
}
```

23.8.7.1 transform_view operations [range.adaptors.transform__view.ops]

23.8.7.1.1 transform_view constructors [range.adaptors.transform__view.ctor]

```
constexpr transform_view(R base, F fun);
```

1 *Effects:* Initializes `base_` with `std::move(base)` and `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 `fun_` with `std::move(fun)`.

23.8.7.1.2 transform_view conversion [range.adaptors.transform__view.conv]

```
constexpr R base() const;
```

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

23.8.7.1.3 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.

23.8.7.1.4 `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.

23.8.7.1.5 `transform_view` range size [range.adaptors.transform_view.size]

```
constexpr auto size() requires SizedRange<R>;
constexpr auto size() const requires SizedRange<const R>;
```

1 *Effects:* Equivalent to: `return ranges::size(base_);`

23.8.7.2 Class template `transform_view::iterator` [range.adaptors.transform_view.iterator]

```
namespace std::ranges {
  template<class R, class F>
  template<bool Const>
  class transform_view<R, F>::iterator { // exposition only
  private:
    using Parent =                  // exposition only
      conditional_t<Const, const transform_view, transform_view>;
    using Base   = conditional_t<  // exposition only
    iterator_t<Base> current_ {};  // exposition only
    Parent* parent_ = nullptr;     // exposition only
  public:
    using iterator_category = see below;
    using iterator_concept  = see below;
    using value_type        =
      remove_cvref_t<invoke_result_t<F&, iter_reference_t<iterator_t<Base>>>>;
    using difference_type   = iter_difference_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(noexcept(invoke(*i.parent_->fun_, *i.current_)));
    friend constexpr void iter_swap(const iterator& x, const iterator& y)
      noexcept(noexcept(ranges::iter_swap(x.current_, y.current_)))
        requires IndirectlySwappable<iterator_t<Base>>;
  };
}
```

1 Let C denote the type iterator_traits<iterator_t<Base>>::iterator_category. If C models DerivedFrom<contiguous_iterator_tag>, then iterator_category denotes random_access_iterator_tag; otherwise, iterator_category denotes C.

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.

23.8.7.2.1 transform_view::iterator operations [range.adaptors.transform_view.iterator.ops]

23.8.7.2.1.1 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 parent_ with addressof(parent).

```
constexpr iterator(iterator<!Const> i)
  requires Const && ConvertibleTo<iterator_t<R>, iterator_t<Base>>;
```

2 *Effects:* Initializes `current_` with `i.current_` and `parent_` with `i.parent_`.

23.8.7.2.1.2 `transform_view::iterator` conversion [range.adaptors.transform_view.iterator.conv]

```
constexpr iterator_t<Base> base() const;
```

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

23.8.7.2.1.3 `transform_view::iterator::operator*` [range.adaptors.transform_view.iterator.star]

```
constexpr decltype(auto) operator*() const;
```

1 *Effects:* Equivalent to: `return invoke(*parent_->fun_, *current_);`

23.8.7.2.1.4 `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;
```

23.8.7.2.1.5 `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;
```

23.8.7.2.1.6 `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;
```

23.8.7.2.1.7 `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]);`

23.8.7.2.2 `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 *Effects:* Equivalent to: `return x.current_ == y.current_;`

```
friend constexpr bool operator!=(const iterator& x, const iterator& y)
  requires EqualityComparable<iterator_t<Base>>;
```

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

```
friend constexpr bool operator<(const iterator& x, const iterator& y)
  requires RandomAccessRange<Base>;
```

3 *Effects:* Equivalent to: `return x.current_ < y.current_;`

```
friend constexpr bool operator>(const iterator& x, const iterator& y)
  requires RandomAccessRange<Base>;
```

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

```
friend constexpr bool operator<=(const iterator& x, const iterator& y)
  requires RandomAccessRange<Base>;
```

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

```
friend constexpr bool operator>=(const iterator& x, const iterator& y)
  requires RandomAccessRange<Base>;
```

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

23.8.7.2.3 `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 *Effects:* Equivalent to: `return iterator{*i.parent_, i.current_ + n};`

```
friend constexpr iterator operator-(iterator i, difference_type n)
  requires RandomAccessRange<Base>;
```

2 *Effects:* Equivalent to: `return iterator{*i.parent_, i.current_ - n};`

```
friend constexpr difference_type operator-(const iterator& x, const iterator& y)
  requires RandomAccessRange<Base>;
```

3 *Effects:* Equivalent to: `return x.current_ - y.current_;`

```
friend constexpr decltype(auto) iter_move(const iterator& i)
  noexcept(noexcept(invoke(*i.parent_->fun_, *i.current_)));
```

4 *Effects:*

(4.1) — If the expression `*i` is an lvalue, equivalent to: `return std::move(*i);`

(4.2) — Otherwise, equivalent to: `return *i;`

```
friend constexpr void iter_swap(const iterator& x, const iterator& y)
  noexcept(noexcept(ranges::iter_swap(x.current_, y.current_)))
    requires IndirectlySwappable<iterator_t<Base>>;
```

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

23.8.7.3 Class template `transform_view::sentinel` [range.adaptors.transform_view.sentinel]

```
namespace std::ranges {
  template<class R, class F>
  template<bool Const>
  class transform_view<R, F>::sentinel { // exposition only
  private:
    using Parent =                                  // exposition only
      conditional_t<Const, const transform_view, transform_view>;
    using Base = conditional_t<Const, const R, R>; // exposition only
    sentinel_t<Base> end_ {};                       // exposition only
  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_t<iterator_t<Base>>
      operator-(const iterator<Const>& x, const sentinel& y)
        requires SizedSentinel<sentinel_t<Base>, iterator_t<Base>>;
    friend constexpr iter_difference_t<iterator_t<Base>>
      operator-(const sentinel& y, const iterator<Const>& x)
        requires SizedSentinel<sentinel_t<Base>, iterator_t<Base>>;
  };
}
```

23.8.7.4 `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_`.

23.8.7.5 `transform_view::sentinel` conversion [range.adaptors.transform_view.sentinel.conv]

```
constexpr sentinel_t<Base> base() const;
```

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

23.8.7.6 `transform_view::sentinel` comparison [range.adaptors.transform_view.sentinel.comp]

```
friend constexpr bool operator==(const iterator<Const>& x, const sentinel& y);
```

1 *Effects:* Equivalent to: `return x.current_ == y.end_;`

```
friend constexpr bool operator==(const sentinel& x, const iterator<Const>& y);
```

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

```
friend constexpr bool operator!=(const iterator<Const>& x, const sentinel& y);
```

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

```
friend constexpr bool operator!=(const sentinel& x, const iterator<Const>& y);
```

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

23.8.7.7 `transform_view::sentinel` non-member functions [range.adaptors.transform_view.sentinel.nonmember]

```
friend constexpr iter_difference_t<iterator_t<Base>>
  operator-(const iterator<Const>& x, const sentinel& y)
    requires SizedSentinel<sentinel_t<Base>, iterator_t<Base>>;
```

1 *Effects:* Equivalent to: `return x.current_ - y.end_;`

```
friend constexpr iter_difference_t<iterator_t<Base>>
  operator-(const sentinel& y, const iterator<Const>& x)
    requires SizedSentinel<sentinel_t<Base>, iterator_t<Base>>;
```

2 *Effects:* Equivalent to: `return x.end_ - y.current_;`

23.8.8 `view::transform` [range.adaptors.transform]

1 The name `view::transform` denotes a range adaptor object (23.8.1). The expression `view::transform(E, F)` for subexpressions `E` and `F` is expression-equivalent to `transform_view{E, F}`.

23.8.9 Class template `iota_view` [range.adaptors.iota_view]

1 `iota_view` generates a sequence of elements by repeatedly incrementing an initial value.

2 [*Example:*

```
for (int i : iota_view{1, 10})
  cout << i << ' '; // prints: 1 2 3 4 5 6 7 8 9
```

— *end example*]

```
namespace std::ranges {
  template<class I>
    concept Decrementable = // exposition only
      see below;
  template<class I>
    concept Advanceable = // exposition only
      see below;

  template<WeaklyIncrementable I, Semiregular Bound = unreachable>
    requires weakly-equality-comparable-with<I, Bound>
  class iota_view : public view_interface<iota_view<I, Bound>> {
  private:
    struct iterator; // exposition only
    struct sentinel; // exposition only
    I value_ {};     // exposition only
    Bound bound_ {}; // 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<class I, class 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* 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_t<I>; } &&
    requires(I i, const I j, const iter_difference_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_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_t<I>` initialized with M, `0`, and `1`, respectively. Then if M is representable by `iter_difference_t<I>`, `I` models *Advanceable* 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` is `true`.

23.8.9.1 iota_view operations [range.adaptors.iota_view.ops]

23.8.9.1.1 iota_view constructors [range.adaptors.iota_view.ctor]

```
constexpr explicit iota_view(I value);
```

1 *Expects:* `Bound` denotes `unreachable` or `Bound{}` is reachable from `value`.

2 *Effects:* Initializes `value_` with `value`.

```
constexpr iota_view(I value, Bound bound);
```

3 *Expects:* `Bound` denotes `unreachable` or `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]).

23.8.9.1.2 iota_view range begin [range.adaptors.iota_view.begin]

```
constexpr iterator begin() const;
```

1 *Effects:* Equivalent to: `return iterator{value_};`

23.8.9.1.3 iota_view range end [range.adaptors.iota_view.end]

```
constexpr sentinel end() const;
```

1 *Effects:* Equivalent to: `return sentinel{bound_};`

```
constexpr iterator end() const requires Same<I, Bound>;
```

2 *Effects:* Equivalent to: `return iterator{bound_};`

23.8.9.1.4 iota_view range size [range.adaptors.iota_view.size]

```
constexpr auto size() const requires
  (Same<I, Bound> && Advanceable<I>) ||
  (Integral<I> && Integral<Bound>) ||
  SizedSentinel<Bound, I>;
```

1 *Effects:* Equivalent to: `return bound_ - value_;`

23.8.9.2 Class iota_view::iterator [range.adaptors.iota_view.iterator]

```
namespace std::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_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 `iterator::iterator_category` is defined as follows:

(1.1) — If `I` models *`Advanceable`*, then `iterator_category` is `random_access_iterator_tag`.

(1.2) — Otherwise, if `I` models *`Decrementable`*, then `iterator_category` is `bidirectional_iterator_tag`.

(1.3) — Otherwise, if `I` models `Incrementable`, then `iterator_category` is `forward_iterator_tag`.

(1.4) — Otherwise, `iterator_category` is `input_iterator_tag`.

2 [*Note:* Overloads for `iter_move` and `iter_swap` are omitted intentionally. — *end note*]

23.8.9.2.1 `iota_view::iterator` operations [range.adaptors.iota_view.iterator.ops]

23.8.9.2.1.1 `iota_view::iterator` constructors [range.adaptors.iota_view.iterator.ctor]

```
explicit constexpr iterator(I value);
```

1 *Effects:* Initializes `value_` with `value`.

23.8.9.2.1.2 `iota_view::iterator::operator*` [range.adaptors.iota_view.iterator.star]

```
constexpr I operator*() const noexcept(is_nothrow_copy_constructible_v<I>);
```

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

2 [*Note:* The `noexcept` clause is needed by the default `iter_move` implementation. — *end note*]

23.8.9.2.1.3 `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;
```

23.8.9.2.1.4 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;
```

23.8.9.2.1.5 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;
```

23.8.9.2.1.6 iota_view::iterator index [range.adaptors.iota_view.iterator.idx]

```
constexpr I operator[](difference_type n) const
  requires Advanceable<I>;
```

1 *Effects:* Equivalent to: `return value_ + n;`

23.8.9.2.2 iota_view::iterator comparisons [range.adaptors.iota_view.iterator.cmp]

```
friend constexpr bool operator==(const iterator& x, const iterator& y)
  requires EqualityComparable<I>;
```

1 *Effects:* Equivalent to: `return x.value_ == y.value_;`

```
friend constexpr bool operator!=(const iterator& x, const iterator& y)
  requires EqualityComparable<I>;
```

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

```
friend constexpr bool operator<(const iterator& x, const iterator& y)
  requires StrictTotallyOrdered<I>;
```

3 *Effects:* Equivalent to: `return x.value_ < y.value_;`

```
friend constexpr bool operator>(const iterator& x, const iterator& y)
  requires StrictTotallyOrdered<I>;
```

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

```
friend constexpr bool operator<=(const iterator& x, const iterator& y)
  requires StrictTotallyOrdered<I>;
```

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

```
friend constexpr bool operator>=(const iterator& x, const iterator& y)
  requires StrictTotallyOrdered<I>;
```

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

23.8.9.2.3 `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 *Effects:* Equivalent to: `return iterator{*i + n};`

```
friend constexpr iterator operator+(difference_type n, iterator i)
  requires Advanceable<I>;
```

2 *Effects:* Equivalent to: `return i + n;`

```
friend constexpr iterator operator-(iterator i, difference_type n)
  requires Advanceable<I>;
```

3 *Effects:* Equivalent to: `return i + -n;`

```
friend constexpr difference_type operator-(const iterator& x, const iterator& y)
  requires Advanceable<I>;
```

4 *Effects:* Equivalent to: `return *x - *y;`

23.8.9.3 Class `iota_view::sentinel` [range.adaptors.iota_view.sentinel]

```
namespace std::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);
  };
}
```

23.8.9.3.1 `iota_view::sentinel` constructors [range.adaptors.iota_view.sentinel.ctor]

```
constexpr explicit sentinel(Bound bound);
```

1 *Effects:* Initializes `bound_` with `bound`.

23.8.9.3.2 `iota_view::sentinel` comparisons [range.adaptors.iota_view.sentinel.cmp]

```
friend constexpr bool operator==(const iterator& x, const sentinel& y);
```

1 *Effects:* Equivalent to: `return x.value_ == y.bound_;`

```
friend constexpr bool operator==(const sentinel& x, const iterator& y);
```

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

```
friend constexpr bool operator!=(const iterator& x, const sentinel& y);
```

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

```
friend constexpr bool operator!=(const sentinel& x, const iterator& y);
```

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

23.8.10 `view::iota` [range.adaptors.iota]

1 The name `view::iota` denotes a customization point object ([customization.point.object]). The expressions `view::iota(E)` and `view::iota(E, F)` for some subexpressions `E` and `F` are expression-equivalent to `iota_view{E}` and `iota_view{E, F}`, respectively.

23.8.11 Class template `take_view` [range.adaptors.take_view]

1 `take_view` produces a `View` of the first *N* elements from another `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 {
  template<View R>
  class take_view : public view_interface<take_view<R>> {
  private:
    R base_ {};                                    // exposition only
    iter_difference_t<iterator_t<R>> count_ {}; // exposition only
    template<bool> struct sentinel;               // exposition only
  public:
    take_view() = default;
    constexpr take_view(R base, iter_difference_t<iterator_t<R>> count);
    template<ViewableRange O>
      requires Constructible<R, all_view<O>>
    constexpr take_view(O&& o, iter_difference_t<iterator_t<R>> count);

    constexpr R base() const;

    constexpr auto begin() requires !simple-view<R>;
    constexpr auto begin() const requires Range<const R>;

    constexpr auto end() requires !simple-view<R>;
    constexpr auto end() const requires Range<const R>;

    constexpr auto size() requires SizedRange<R>;
    constexpr auto size() const requires SizedRange<const R>;
  };

  template<Range R>
    take_view(R&& base, iter_difference_t<iterator_t<R>> n)
      -> take_view<all_view<R>>;
}
```

23.8.11.1 `take_view` operations [range.adaptors.take_view.ops]

```
constexpr take_view(R base, iter_difference_t<iterator_t<R>> count);
```

1 *Effects:* Initializes `base_` with `std::move(base)` and `count_` with `count`.

```
template<ViewableRange O>
  requires 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 `count_` with `count`.

```
constexpr R base() const;
```

3 *Effects:* Equivalent to: `return base_;`

```
constexpr auto begin() requires !simple-view<R>;
constexpr auto begin() const requires Range<const R>;
```

4 Let `T` be `R` for the first overload and `const R` for the second.

5 *Effects:*

(5.1) — If T models both SizedRange and RandomAccessRange, equivalent to: `return ranges::begin(base_);`

(5.2) — Otherwise, if T models SizedRange, equivalent to: `return counted_iterator{ranges::begin(base_), size()};`

(5.3) — Otherwise, equivalent to: `return counted_iterator{ranges::begin(base_), count_};`

```
constexpr auto end() requires !simple-view<R>;
constexpr auto end() const requires Range<const R>;
```

6 Let T be R for the first overload and `const R` for the second.

7 *Effects:*

(7.1) — If T models both SizedRange and RandomAccessRange, equivalent to: `return ranges::begin(base_) + size();`

(7.2) — Otherwise, if T models SizedRange, equivalent to: `return default_sentinel{};`

(7.3) — Otherwise, equivalent to: `return sentinel<is_const_v<T>>{ranges::end(base_)};`

```
constexpr auto size() requires SizedRange<R>;
constexpr auto size() const requires SizedRange<const R>;
```

8 *Effects:* Equivalent to:

```
auto n = ranges::size(base_);
return ranges::min(n, static_cast<decltype(n)>(count_));
```

23.8.11.2 Class template take_view::sentinel [range.adaptors.take_view.sentinel]

```
namespace std::ranges {
  template<class R>
  template<bool Const>
  class take_view<R>::sentinel { // exposition only
  private:
    using Base = conditional_t<Const, const R, R>; // exposition only
    using CI = counted_iterator<iterator_t<Base>>; // exposition only
    sentinel_t<Base> end_ {};                      // exposition only
  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);
    friend constexpr bool operator==(const CI& y, const sentinel& x);
    friend constexpr bool operator!=(const sentinel& x, const CI& y);
    friend constexpr bool operator!=(const CI& y, const sentinel& x);
  };
}
```

23.8.11.2.1 take_view::sentinel operations [range.adaptors.take_view.sentinel.ops]

```
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_`.

```
constexpr sentinel_t<Base> base() const;
```

3 *Effects:* Equivalent to: `return end_;`

```
friend constexpr bool operator==(const sentinel& x, const CI& y);
friend constexpr bool operator==(const CI& y, const sentinel& x);
```

4 *Effects:* Equivalent to: `return y.count() == 0 || y.base() == x.end_;`

```
friend constexpr bool operator!=(const sentinel& x, const CI& y);
friend constexpr bool operator!=(const CI& y, const sentinel& x);
```

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

23.8.12 `view::take` [range.adaptors.take]

1 The name `view::take` denotes a range adaptor object (23.8.1). The expression `view::take(E, F)` for some subexpressions `E` and `F` is expression-equivalent to `take_view{E, F}`.

23.8.13 Class template `join_view` [range.adaptors.join_view]

1 `join_view` flattens a `View` of ranges into a `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 {
  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_t<iterator_t<R>>>)
  class join_view : public view_interface<join_view<R>> {
  private:
    using InnerRng =                // exposition only
      iter_reference_t<iterator_t<R>>;
    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);

    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<class R>
    explicit join_view(R&&) -> join_view<all_view<R>>;
}
```

23.8.13.1 join_view operations [range.adaptors.join_view.ops]

23.8.13.1.1 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))`.

23.8.13.1.2 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<`C`>{*this, ranges::begin(base_)};` where C is `simple-view<R>` or `true` for the first and second overloads, respectively.

23.8.13.1.3 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<`C`>{*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<`C`>{*this, ranges::end(base_)};` where C is `simple-view<R>` or `true` for the first and second overloads, respectively.

23.8.13.2 Class template join_view::iterator [range.adaptors.join_view.iterator]

1

```
namespace std::ranges {
template<class R>
  template<bool Const>
  struct join_view<R>::iterator { // exposition only
  private:
    using Parent =                                        // exposition only
      conditional_t<Const, const join_view, join_view>;
    using Base   = conditional_t<Const, const R, R>;      // exposition only

    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_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 ref_is_glvalue && ForwardRange<Base> &&
        ForwardRange<iter_reference_t<iterator_t<Base>>>;

    constexpr iterator& operator--();
      requires ref_is_glvalue && BidirectionalRange<Base> &&
        BidirectionalRange<iter_reference_t<iterator_t<Base>>>;

    constexpr iterator operator--(int)
      requires ref_is_glvalue && BidirectionalRange<Base> &&
        BidirectionalRange<iter_reference_t<iterator_t<Base>>>;

    friend constexpr bool operator==(const iterator& x, const iterator& y)
      requires 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 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(noexcept(ranges::iter_move(i.inner_)));

    friend constexpr void iter_swap(const iterator& x, const iterator& y)
      noexcept(noexcept(ranges::iter_swap(x.inner_, y.inner_)));
  };
}
```

2 `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 *OUTERC* and *INNERC* each model `DerivedFrom<bidirectional_iterator_tag>`, `iterator_category` denotes `bidirectional_iterator_tag`.

(2.2.2) — Otherwise, if *OUTERC* and *INNERC* each model `DerivedFrom<forward_iterator_tag>`, `iterator_category` denotes `forward_iterator_tag`.

(2.3) — Otherwise, `iterator_category` denotes `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 `iterator::difference_type` denotes the type:

```
common_type_t<
  iter_difference_t<iterator_t<Base>>,
  iter_difference_t<iterator_t<iter_reference_t<iterator_t<Base>>>>>
```

23.8.13.2.1 `join_view::iterator` operations [range.adaptors.join_view.iterator.ops]

23.8.13.2.1.1 `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:* 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_ = ranges::begin(inner);
  if (inner_ != ranges::end(inner))
    return;
}
if constexpr (ref_is_glvalue)
  inner_ = iterator_t<iter_reference_t<iterator_t<Base>>>{};
```

23.8.13.2.1.2 `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 `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_`, `inner_` with `i.inner_`, and `parent_` with `i.parent_`.

23.8.13.2.1.3 `join_view::iterator::operator*` [range.adaptors.join_view.iterator.star]

```
constexpr decltype(auto) operator*() const;
```

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

23.8.13.2.1.4 `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_;`

23.8.13.2.1.5 `join_view::iterator::operator++` [range.adaptors.join_view.iterator.inc]

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

1 Let *inner-range* be:

(1.1) — If `ref_is_glvalue` is `true`, `*outer_`.

(1.2) — Otherwise, `parent_->inner_`.

2 *Effects:* Equivalent to:

```
if (++inner_ == ranges::end(inner-range)) {
  ++outer_;
  satisfy();
}
return *this;
```

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

3 *Effects:* Equivalent to: `++*this`.

```
constexpr iterator operator++(int)
  requires ref_is_glvalue && ForwardRange<Base> &&
    ForwardRange<iter_reference_t<iterator_t<Base>>>;
```

4 *Effects:* Equivalent to:

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

23.8.13.2.1.6 `join_view::iterator::operator--` [range.adaptors.join_view.iterator.dec]

```
constexpr iterator& operator--();
  requires 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 ref_is_glvalue && BidirectionalRange<Base> &&
    BidirectionalRange<iter_reference_t<iterator_t<Base>>>;
```

2 *Effects:* Equivalent to:

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

23.8.13.2.2 `join_view::iterator` comparisons [range.adaptors.join_view.iterator.comp]

```
friend constexpr bool operator==(const iterator& x, const iterator& y)
  requires ref_is_glvalue && EqualityComparable<iterator_t<Base>> &&
    EqualityComparable<iterator_t<iter_reference_t<iterator_t<Base>>>>;
```

1 *Effects:* Equivalent to: `return x.outer_ == y.outer_ && x.inner_ == y.inner_;`

```
friend constexpr bool operator!=(const iterator& x, const iterator& y)
  requires ref_is_glvalue && EqualityComparable<iterator_t<Base>> &&
    EqualityComparable<iterator_t<iter_reference_t<iterator_t<Base>>>>;
```

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

23.8.13.2.3 join_view::iterator non-member functions [range.adaptors.join_view.iterator.nonmember]

```
friend constexpr decltype(auto) iter_move(const iterator& i)
  noexcept(noexcept(ranges::iter_move(i.inner_)));
```

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

```
friend constexpr void iter_swap(const iterator& x, const iterator& y)
  noexcept(noexcept(ranges::iter_swap(x.inner_, y.inner_)));
```

2 *Effects:* Equivalent to: `return ranges::iter_swap(x.inner_, y.inner_);`

23.8.13.3 Class template join_view::sentinel [range.adaptors.join_view.sentinel]

```
namespace std::ranges {
  template<class R>
  template<bool Const>
  struct join_view<R>::sentinel { // exposition only
  private:
    using Parent =                                    // exposition only
      conditional_t<Const, const join_view, join_view>;
    using Base   = conditional_t<Const, const R, R>; // exposition only
    sentinel_t<Base> end_ {};                         // exposition only
  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);
  };
}
```

23.8.13.3.1 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 *Effects:* Equivalent to: `return x.outer_ == y.end_;`

```
friend constexpr bool operator==(const sentinel& x, const iterator<Const>& y);
```

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

```
friend constexpr bool operator!=(const iterator<Const>& x, const sentinel& y);
```

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

```
friend constexpr bool operator!=(const sentinel& x, const iterator<Const>& y);
```

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

23.8.14 view::join [range.adaptors.join]

1 The name `view::join` denotes a range adaptor object (23.8.1). The expression `view::join(E)` for some subexpression `E` is expression-equivalent to `join_view{E}`.

23.8.15 Class template `empty_view` [range.adaptors.empty_view]

1 `empty_view` produces 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 {
  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 T* data() noexcept { return nullptr; }
    constexpr static ptrdiff_t size() noexcept { return 0; }
    constexpr static bool empty() noexcept { return true; }

    friend constexpr T* begin(empty_view) noexcept { return nullptr; }
    friend constexpr T* end(empty_view) noexcept { return nullptr; }
  };
}
```

23.8.16 Class template `single_view` [range.adaptors.single_view]

1 `single_view` produces a `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 {
  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 T* begin() noexcept;
    constexpr const T* begin() const noexcept;
    constexpr T* end() noexcept;
    constexpr const T* end() const noexcept;
    constexpr static ptrdiff_t size() noexcept;
    constexpr T* data() noexcept;
    constexpr const T* data() const noexcept;
  };

  template<class T>
    explicit single_view(T&&) -> single_view<decay_t<T>>;
}
```

23.8.16.1 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 T* begin() noexcept;
constexpr const T* begin() const noexcept;
```

4 *Effects:* Equivalent to: `return data();`

```
constexpr T* end() noexcept;
constexpr const T* end() const noexcept;
```

5 *Effects:* Equivalent to: `return data() + 1;`

```
constexpr static ptrdiff_t size() noexcept;
```

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

```
constexpr T* data() noexcept;
constexpr const T* data() const noexcept;
```

7 *Effects:* Equivalent to: `return value_.operator->();`

23.8.17 view::single [range.adaptors.single]

1 The name `view::single` denotes a customization point object ([customization.point.object]). The expression `view::single(E)` for some subexpression `E` is expression-equivalent to `single_view{E}`.

23.8.18 Class template split_view [range.adaptors.split_view]

1 `split_view` takes a `View` and a delimiter, and splits the `View` into subranges on the delimiter. The delimiter can be a single element or a `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 {
  template<class R>
  concept tiny-range = // exposition only
    SizedRange<R> && (remove_reference_t<R>::size() <= 1);

  template<InputRange R, ForwardRange Pattern>
    requires View<R> && View<Pattern> &&
      IndirectlyComparable<iterator_t<R>, iterator_t<Pattern>, ranges::equal_to<>> &&
      (ForwardRange<R> || tiny-range<Pattern>)
  class split_view : public view_interface<split_view<R, Pattern>> {
  private:
    R base_ {};                              // exposition only
    Pattern pattern_ {};                     // exposition only
    iterator_t<R> current_ {};               // exposition only, only present if !ForwardRange<R>
    template<bool Const> struct outer_iterator; // exposition only
```

```
    template<bool Const> struct inner_iterator; // exposition only
  public:
    split_view() = default;
    constexpr split_view(R base, Pattern pattern);

    template<InputRange O, ForwardRange P>
      requires
        Constructible<R, all_view<O>> &&
        Constructible<Pattern, all_view<P>>
    constexpr split_view(O&& o, P&& p);

    template<InputRange O>
      requires
        Constructible<R, all_view<O>> &&
        Constructible<Pattern, single_view<iter_value_t<iterator_t<O>>>>
    constexpr split_view(O&& o, iter_value_t<iterator_t<O>> e);

    constexpr auto begin();
    constexpr auto begin() const
      requires ForwardRange<R> && ForwardRange<const R>;

    constexpr auto end()
      requires ForwardRange<R> && CommonRange<R>;
    constexpr auto end() const;
  };

  template<class O, class P>
    split_view(O&&, P&&) -> split_view<all_view<O>, all_view<P>>;

  template<InputRange O>
    split_view(O&&, iter_value_t<iterator_t<O>>)
      -> split_view<all_view<O>, single_view<iter_value_t<iterator_t<O>>>>;
}
```

23.8.18.1 split_view operations [range.adaptors.split__view.ops]

```
constexpr split_view(R base, Pattern pattern);
```

1 *Effects:* Initializes `base_` with `std::move(base)`, and `pattern_` with `std::move(pattern)`.

```
template<InputRange O, ForwardRange P>
  requires
    Constructible<R, all_view<O>> &&
    Constructible<Pattern, all_view<P>>
constexpr split_view(O&& o, P&& p);
```

2 *Effects:* Initializes `base_` with `view::all(std::forward<O>(o))` and `pattern_` with `view::all(std::forward<P>(p))`.

```
template<InputRange O>
  requires
    Constructible<R, all_view<O>> &&
    Constructible<Pattern, single_view<iter_value_t<iterator_t<O>>>>
constexpr split_view(O&& o, iter_value_t<iterator_t<O>> e);
```

3 *Effects:* Initializes `base_` with `view::all(std::forward<O>(o))` and `pattern_` with `single_view{std::move(e)}`.

```
constexpr auto begin();
```

4 *Effects:* If `R` models `ForwardRange`, equivalent to:

```
return outer_iterator<simple-view<R>>{*this, ranges::begin(base_)};
```

Otherwise, equivalent to:

```
current_ = ranges::begin(base_);
return outer_iterator<false>{*this};
```

```
constexpr auto begin() requires ForwardRange<R>;
constexpr auto begin() const
  requires ForwardRange<R> && ForwardRange<const R>;
```

5 *Effects:* Equivalent to: `return outer_iterator<true>{*this, ranges::begin(base_)};`

```
constexpr auto end()
  requires ForwardRange<R> && CommonRange<R>;
```

6 *Effects:* Equivalent to: `return outer_iterator<simple-view<R>>{*this, ranges::end(base_)};`

```
constexpr auto end() const;
```

7 *Effects:* If `R` models `ForwardRange<R> && ForwardRange<const R> && CommonRange<const R>`, equivalent to: `return outer_iterator<true>{*this, ranges::end(base_)};` Otherwise, equivalent to: `return default_sentinel{};`

23.8.18.2 Class template `split_view::outer_iterator` [range.adaptors.split__view.outer__iterator]

```
namespace std::ranges {
  template<class R, class Pattern>
  template<bool Const>
  struct split_view<R, Pattern>::outer_iterator { // exposition only
  private:
    using Parent =                // exposition only
      conditional_t<Const, const split_view, split_view>;
    using Base   =                // exposition only
      conditional_t<Const, const R, R>;
    Parent* parent_ = nullptr;    // exposition only
    iterator_t<Base> current_ {}; // exposition only, present only if R models ForwardRange
  public:
    using iterator_category = input_iterator_tag;
    using iterator_concept  =
      conditional_t<ForwardRange<Base>, forward_iterator_tag, input_iterator_tag>;
    using difference_type   = iter_difference_t<iterator_t<Base>>;
    struct value_type;

    outer_iterator() = default;
    constexpr explicit outer_iterator(Parent& parent)
      requires !ForwardRange<Base>;
    constexpr outer_iterator(Parent& parent, iterator_t<Base> current)
      requires ForwardRange<Base>;
    constexpr outer_iterator(outer_iterator<!Const> i) requires Const &&
      ConvertibleTo<iterator_t<R>, iterator_t<const R>>;

    constexpr value_type operator*() const;

    constexpr outer_iterator& operator++();
    constexpr decltype(auto) operator++(int);

    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>;

    friend constexpr bool operator==(const outer_iterator& x, default_sentinel);
    friend constexpr bool operator==(default_sentinel, const outer_iterator& x);
    friend constexpr bool operator!=(const outer_iterator& x, default_sentinel y);
    friend constexpr bool operator!=(default_sentinel y, const outer_iterator& x);
  };
}
```

1 Many of the following specifications refer to the notional member *`current`* of `outer_iterator`. *`current`* is equivalent to `current_` if `R` models `ForwardRange`, and `parent_->current_` otherwise.

23.8.18.3 split_view::outer_iterator operations [range.adaptors.split_view.outer_iterator.ops]

23.8.18.3.1 split_view::outer_iterator constructors [range.adaptors.split_view.outer_iterator.ctor]

```
constexpr explicit outer_iterator(Parent& parent)
  requires !ForwardRange<Base>;
```

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<R>, iterator_t<const R>>;
```

3 *Effects:* Initializes `parent_` with `i.parent_` and `current_` with `i.current_`.

23.8.18.3.2 split_view::outer_iterator::operator* [range.adaptors.split_view.outer_iterator.star]

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

1 *Effects:* Equivalent to: `return value_type{*this};`

23.8.18.3.3 split_view::outer_iterator::operator++ [range.adaptors.split_view.outer_iterator.inc]

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

1 *Effects:* Equivalent to:

```
const auto end = ranges::end(parent_->base_);
if (current == end) return *this;
const auto [pbegin, pend] = subrange{parent_->pattern_};
if (pbegin == pend) ++current;
else {
  do {
    const auto [b, p] = ranges::mismatch(current, end, pbegin, pend);
    if (p == pend) {
      current = b; // The pattern matched; skip it
      break;
    }
  } while (++current != end);
}
return *this;
```

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

2 *Effects:* If `Base` models `ForwardRange`, equivalent to:

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

Otherwise, equivalent to `++*this`.

23.8.18.3.4 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);`

```
friend constexpr bool operator==(const outer_iterator& x, default_sentinel);
friend constexpr bool operator==(default_sentinel, const outer_iterator& x);
```

3 *Effects:* Equivalent to: `return x.current == ranges::end(x.parent_->base_);`

```
friend constexpr bool operator!=(const outer_iterator& x, default_sentinel y);
friend constexpr bool operator!=(default_sentinel y, const outer_iterator& x);
```

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

23.8.18.4 Class split_view::outer_iterator::value_type [range.adaptors.split__view.outer__iterator.value__type]

```
namespace std::ranges {
  template<class R, class Pattern>
  template<bool Const>
  struct split_view<R, Pattern>::outer_iterator<Const>::value_type {
  private:
    outer_iterator i_ {}; // exposition only
  public:
    value_type() = default;
    constexpr explicit value_type(outer_iterator i);

    constexpr inner_iterator<Const> begin() const;
    constexpr default_sentinel end() const;
  };
}
```

23.8.18.4.1 split_view::outer_iterator::value_type operations [range.adaptors.split__view.outer__iterator.value__type.ops]

```
constexpr explicit value_type(outer_iterator i);
```

1 *Effects:* Initializes `i_` with `i`.

```
constexpr inner_iterator<Const> begin() const;
```

2 *Effects:* Equivalent to: `return inner_iterator<Const>{i_};`

```
constexpr default_sentinel end() const;
```

3 *Effects:* Equivalent to: `return {};`

23.8.18.5 Class template split_view::inner_iterator [range.adaptors.split__view.inner__iterator]

```
namespace std::ranges {
  template<class R, class Pattern>
  template<bool Const>
  struct split_view<R, Pattern>::inner_iterator { // exposition only
  private:
    using Base =
      conditional_t<Const, const R, R>; // exposition only
    outer_iterator<Const> i_ {};        // exposition only
    bool zero_ = false;                 // exposition only
  public:
    using iterator_category = see below;
    using iterator_concept  = typename outer_iterator<Const>::iterator_concept;
    using difference_type   = iter_difference_t<iterator_t<Base>>;
    using value_type        = iter_value_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 decltype(auto) operator++(int);
```

```
    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 bool operator==(const inner_iterator& x, default_sentinel);
    friend constexpr bool operator==(default_sentinel, const inner_iterator& x);
    friend constexpr bool operator!=(const inner_iterator& x, default_sentinel y);
    friend constexpr bool operator!=(default_sentinel y, const inner_iterator& x);

    friend constexpr decltype(auto) iter_move(const inner_iterator& i)
      noexcept(noexcept(ranges::iter_move(i.i_.current)));
    friend constexpr void iter_swap(const inner_iterator& x, const inner_iterator& y)
      noexcept(noexcept(ranges::iter_swap(x.i_.current, y.i_.current)))
        requires IndirectlySwappable<iterator_t<Base>>;
  };
}
```

1 The *typedef-name* `iterator_category` denotes `forward_iterator_tag` if `iterator_traits<iterator_t<Base>>::iterator_category` models `DerivedFrom<forward_iterator_tag>`, and `input_iterator_tag` otherwise.

23.8.18.5.1 `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`.

23.8.18.5.2 `split_view::inner_iterator::operator*` [range.adaptors.split_view.inner_iterator.star]

```
constexpr decltype(auto) operator*() const;
```

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

23.8.18.5.3 `split_view::inner_iterator::operator++` [range.adaptors.split_view.inner_iterator.inc]

```
constexpr inner_iterator& operator++() const;
```

1 *Effects:* Equivalent to:

```
zero_ = true;
if constexpr (!ForwardRange<Base>) {
  if constexpr (Pattern::size() == 0) {
    return *this;
  }
}
++i_.current;
return *this;
```

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

2 *Effects:* If `R` models `ForwardRange`, equivalent to:

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

Otherwise, equivalent to `++*this`.

23.8.18.5.4 `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);`

```
friend constexpr bool operator==(const inner_iterator& x, default_sentinel);
friend constexpr bool operator==(default_sentinel, const inner_iterator& x);
```

3 *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!=(const inner_iterator& x, default_sentinel y);
friend constexpr bool operator!=(default_sentinel y, const inner_iterator& x);
```

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

23.8.18.5.5 `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(noexcept(ranges::iter_move(i.i_.current)));
```

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

```
friend constexpr void iter_swap(const inner_iterator& x, const inner_iterator& y)
  noexcept(noexcept(ranges::iter_swap(x.i_.current, y.i_.current)))
    requires IndirectlySwappable<iterator_t<Base>>;
```

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

23.8.19 `view::split` [range.adaptors.split]

1 The name `view::split` denotes a range adaptor object (23.8.1). The expression `view::split(E, F)` for some subexpressions `E` and `F` is expression-equivalent to `split_view{E, F}`.

23.8.20 `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) — If `T` models `Iterator` and `U` models `ConvertibleTo<iter_difference_t<T>>`,

(1.1.1) — `subrange{E, E + F}` if `T` models `ContiguousIterator`.

(1.1.2) — Otherwise, `subrange{counted_iterator(E, static_cast<iter_difference_t<T>>(F)), default_sentinel{}}`.

(1.2) — Otherwise, `view::counted(E, F)` is ill-formed.

23.8.21 Class template `common_view` [range.adaptors.common__view]

1 `common_view` takes a `View` which has different types for its iterator and sentinel and turns it into an equivalent `View` 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::ranges {
  template<View R>
    requires !CommonRange<R>
  class common_view : public view_interface<common_view<R>> {
  private:
    R base_ {}; // exposition only
  public:
    common_view() = default;

    explicit constexpr common_view(R r);

    template<ViewableRange O>
      requires !CommonRange<O> && Constructible<R, all_view<O>>
    explicit constexpr common_view(O&& o);

    constexpr R base() const;

    constexpr auto size() const requires SizedRange<const R>;

    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<R>;
  };

  template<class O>
    common_view(O&&) -> common_view<all_view<O>>;
}
```

23.8.21.1 common_view operations [range.adaptors.common_view.ops]

```
explicit constexpr common_view(R base);
```

1 *Effects:* Initializes `base_` with `std::move(base)`.

```
template<ViewableRange O>
  requires !CommonRange<O> && Constructible<R, all_view<O>>
explicit constexpr common_view(O&& o);
```

2 *Effects:* Initializes `base_` with `view::all(std::forward<O>(o))`.

```
constexpr R base() const;
```

3 *Effects:* Equivalent to: `return base_;`

```
constexpr auto size() const requires SizedRange<const R>;
```

4 *Effects:* Equivalent to: `return ranges::size(base_);`

```
constexpr auto begin();
constexpr auto begin() const requires Range<const R>;
```

5 Let `T` be `R` or `const R` for the first and second overloads, respectively.

6 *Effects:* Equivalent to:

```
return common_iterator<iterator_t<T>, sentinel_t<T>>(ranges::begin(base_));
```

```
constexpr auto begin()
  requires RandomAccessRange<R> && SizedRange<R>;
constexpr auto begin() const
  requires RandomAccessRange<const R> && SizedRange<const R>;
```

7 *Effects:* Equivalent to: `return ranges::begin(base_);`

```
constexpr auto end();
constexpr auto end() const requires Range<const R>;
```

8 Let `T` be `R` or `const R` for the first and second overloads, respectively.

9 *Effects:* Equivalent to:

```
return common_iterator<iterator_t<T>, sentinel_t<T>>(ranges::end(base_));
```

```
constexpr auto end()
  requires RandomAccessRange<R> && SizedRange<R>;
constexpr auto end() const
  requires RandomAccessRange<const R> && SizedRange<const R>;
```

10 *Effects:* Equivalent to: `return ranges::begin(base_) + ranges::size(base_);`

23.8.22 `view::common` [range.adaptors.common]

1 The name `view::common` denotes a range adaptor object (23.8.1). The expression `view::common(E)` is expression-equivalent to:

(1.1) — If `decltype((E))` models `CommonRange`, `view::all(E)`, if that is a well-formed expression.

(1.2) — Otherwise, `common_view{E}`, if that is a well-formed expression.

(1.3) — Otherwise, `view::common(E)` is ill-formed.

23.8.23 Class template `reverse_view` [range.adaptors.reverse_view]

1 `reverse_view` takes a bidirectional `View` and produces another `View` 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::ranges {
  template<View R>
    requires BidirectionalRange<R>
  class reverse_view : public view_interface<reverse_view<R>> {
  private:
    R base_ {}; // exposition only
  public:
    reverse_view() = default;

    explicit constexpr reverse_view(R r);

    template<ViewableRange O>
      requires BidirectionalRange<O> && Constructible<R, all_view<O>>
    explicit constexpr reverse_view(O&& o);

    constexpr R base() const;
```

```
    constexpr auto begin();
    constexpr auto begin() requires CommonRange<R>;
    constexpr auto begin() const requires CommonRange<const R>;

    constexpr auto end();
    constexpr auto end() const requires CommonRange<const R>;

    constexpr auto size() const requires SizedRange<const R>;
  };

  template<class O>
    reverse_view(O&&) -> reverse_view<all_view<O>>;
}
```

23.8.23.1 `reverse_view` operations [range.adaptors.reverse_view.ops]

```
explicit constexpr reverse_view(R base);
```

1 *Effects:* Initializes `base_` with `std::move(base)`.

```
template<ViewableRange O>
  requires BidirectionalRange<O> && Constructible<R, all_view<O>>
explicit constexpr reverse_view(O&& o);
```

2 *Effects:* Initializes `base_` with `view::all(std::forward<O>(o))`.

```
constexpr R base() const;
```

3 *Effects:* Equivalent to: `return base_;`

```
constexpr auto begin();
```

4 *Returns:* `reverse_iterator{ranges::next(ranges::begin(base_), ranges::end(base_))}`.

5 *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<R>;
constexpr auto begin() const requires CommonRange<const R>;
```

6 *Effects:* Equivalent to: `return reverse_iterator{ranges::end(base_)};`

```
constexpr auto end() requires CommonRange<R>;
constexpr auto end() const requires CommonRange<const R>;
```

7 *Effects:* Equivalent to: `return reverse_iterator{ranges::begin(base_)};`

```
constexpr auto size() const requires SizedRange<const R>;
```

8 *Effects:* Equivalent to: `return ranges::size(base_);`

23.8.24 `view::reverse` [range.adaptors.reverse]

1 The name `view::reverse` denotes a range adaptor object (23.8.1). The expression `view::reverse(E)` for some subexpression `E` is expression-equivalent to `reverse_view{E}`.

24 Algorithms library [algorithms]

24.1 General [algorithms.general]

[...]

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

```
#include <initializer_list>
```

```
namespace std {
  // 24.5, non-modifying sequence operations
  // 24.5.1, 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 R, class Proj = identity,
        IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
      constexpr bool all_of(R&& r, Pred pred, Proj proj = Proj{});
  }

  // 24.5.2, 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 R, class Proj = identity,
        IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
      constexpr bool any_of(R&& r, Pred pred, Proj proj = Proj{});
  }

  // 24.5.3, 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 R, class Proj = identity,
        IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
      constexpr bool none_of(R&& r, Pred pred, Proj proj = Proj{});
  }

  // 24.5.4, 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>
    constexpr for_each_result<I, Fun>
      for_each(I first, S last, Fun f, Proj proj = Proj{});
  template<InputRange R, class Proj = identity,
      IndirectUnaryInvocable<projected<iterator_t<R>, Proj>> Fun>
    constexpr for_each_result<safe_iterator_t<R>, Fun>
      for_each(R&& r, 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);

// 24.5.5, 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 R, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to<>, projected<iterator_t<R>, Proj>, const T*>
    constexpr safe_iterator_t<R>
      find(R&& r, 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 R, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr safe_iterator_t<R>
      find_if(R&& r, 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 R, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr safe_iterator_t<R>
      find_if_not(R&& r, Pred pred, Proj proj = Proj{});
}
```

// *24.5.6, 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>
    requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
    constexpr subrange<I1>
      find_end(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<ForwardRange R1, ForwardRange R2,
      class Pred = ranges::equal_to<>, class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2>
    constexpr safe_subrange_t<R1>
      find_end(R1&& r1, R2&& r2, Pred pred = Pred{},
               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

// *24.5.7, 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 R1, ForwardRange R2, class Proj1 = identity,
      class Proj2 = identity,
      IndirectRelation<projected<iterator_t<R1>, Proj1>,
        projected<iterator_t<R2>, Proj2>> Pred = ranges::equal_to<>>
    constexpr safe_iterator_t<R1>
      find_first_of(R1&& r1, R2&& r2,
                    Pred pred = Pred{},
                    Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

// 24.5.8, 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 R, class Proj = identity,
      IndirectRelation<projected<iterator_t<R>, Proj>> Pred = ranges::equal_to<>>
    constexpr safe_iterator_t<R>
      adjacent_find(R&& r, Pred pred = Pred{}, Proj proj = Proj{});
}

// 24.5.9, 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_t<I>
      count(I first, S last, const T& value, Proj proj = Proj{});
```

```
  template<InputRange R, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to<>, projected<iterator_t<R>, Proj>, const T*>
    constexpr iter_difference_t<iterator_t<R>>
      count(R&& r, const T& value, Proj proj = Proj{});
  template<InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr iter_difference_t<I>
      count_if(I first, S last, Pred pred, Proj proj = Proj{});
  template<InputRange R, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr iter_difference_t<iterator_t<R>>
      count_if(R&& r, Pred pred, Proj proj = Proj{});
}

// 24.5.10, 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<>>
    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 R1, InputRange R2,
      class Proj1 = identity, class Proj2 = identity,
      IndirectRelation<projected<iterator_t<R1>, Proj1>,
        projected<iterator_t<R2>, Proj2>> Pred = ranges::equal_to<>>
    constexpr mismatch_result<safe_iterator_t<R1>, safe_iterator_t<R2>>
      mismatch(R1&& r1, R2&& r2, Pred pred = Pred{},
               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

// 24.5.11, 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 R1, InputRange R2, class Pred = ranges::equal_to<>,
      class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2>
    constexpr bool equal(R1&& r1, R2&& r2, Pred pred = Pred{},
                         Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

```
// 24.5.12, 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 R1, ForwardRange R2, class Pred = ranges::equal_to<>,
      class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2>
    constexpr bool is_permutation(R1&& r1, R2&& r2, Pred pred = Pred{},
                                  Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

// 24.5.13, 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>
      search(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<ForwardRange R1, ForwardRange R2, class Pred = ranges::equal_to<>,
      class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2>
    constexpr safe_subrange_t<R1>
```

```
    search(R1&& r1, R2&& r2, 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_t<I> count,
               const T& value, Pred pred = Pred{}, Proj proj = Proj{});
  template<ForwardRange R, class T, class Pred = ranges::equal_to<>,
      class Proj = identity>
    requires IndirectlyComparable<iterator_t<R>, const T*, Pred, Proj>
    constexpr safe_subrange_t<R>
      search_n(R&& r, iter_difference_t<iterator_t<R>> 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);

// 24.6, mutating sequence operations
// 24.6.1, 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>
    constexpr copy_result<I, O>
```

```
    copy(I first, S last, O result);
  template<InputRange R, WeaklyIncrementable O>
    requires IndirectlyCopyable<iterator_t<R>, O>
    constexpr copy_result<safe_iterator_t<R>, O>
      copy(R&& r, 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>
    constexpr copy_n_result<I, O>
      copy_n(I first, iter_difference_t<I> n, O result);
}

template<class InputIterator, class OutputIterator, class Predicate>
  constexpr OutputIterator copy_if(InputIterator first, InputIterator last,
                                   OutputIterator result, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate>
  ForwardIterator2 copy_if(ExecutionPolicy&& exec, // see [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>
    constexpr copy_if_result<I, O>
      copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});
  template<InputRange R, WeaklyIncrementable O, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    requires IndirectlyCopyable<iterator_t<R>, O>
    constexpr copy_if_result<safe_iterator_t<R>, O>
      copy_if(R&& r, 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<I1, I2>;

  template<BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
    requires IndirectlyCopyable<I1, I2>
    constexpr copy_backward_result<I1, I2>
      copy_backward(I1 first, S1 last, I2 result);
  template<BidirectionalRange R, BidirectionalIterator I>
    requires IndirectlyCopyable<iterator_t<R>, I>
    constexpr copy_backward_result<safe_iterator_t<R>, I>
```

```
    copy_backward(R&& r, I result);
}

// 24.6.2, move
template<class InputIterator, class OutputIterator>
  constexpr OutputIterator move(InputIterator first, InputIterator last,
                                OutputIterator result);

namespace ranges {
  template<class I, class O>
  using move_result = copy_result<I, O>;

  template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
    requires IndirectlyMovable<I, O>
    constexpr move_result<I, O>
      move(I first, S last, O result);
  template<InputRange R, WeaklyIncrementable O>
    requires IndirectlyMovable<iterator_t<R>, O>
    constexpr move_result<safe_iterator_t<R>, O>
      move(R&& r, 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>
    constexpr move_backward_result<I1, I2>
      move_backward(I1 first, S1 last, I2 result);
  template<BidirectionalRange R, BidirectionalIterator I>
    requires IndirectlyMovable<iterator_t<R>, I>
    constexpr move_backward_result<safe_iterator_t<R>, I>
      move_backward(R&& r, I result);
}

// 24.6.3, swap
template<class ForwardIterator1, class ForwardIterator2>
  constexpr 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<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2>
    requires IndirectlySwappable<I1, I2>
    constexpr swap_ranges_result<I1, I2>
      swap_ranges(I1 first1, S1 last1, I2 first2, S2 last2);
```

```
  template<InputRange R1, InputRange R2>
    requires IndirectlySwappable<iterator_t<R1>, iterator_t<R2>>
    constexpr swap_ranges_result<safe_iterator_t<R1>, safe_iterator_t<R2>>
      swap_ranges(R1&& r1, R2&& r2);
}

template<class ForwardIterator1, class ForwardIterator2>
  void iter_swap(ForwardIterator1 a, ForwardIterator2 b);

// 24.6.4, transform
template<class InputIterator, class OutputIterator, class UnaryOperation>
  constexpr OutputIterator
    transform(InputIterator first1, InputIterator last1,
              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 first1, ForwardIterator1 last1,
              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_t<F&, projected<I, Proj>>>
    constexpr unary_transform_result<I, O>
      transform(I first1, S last1, O result, F op, Proj proj = Proj{});
  template<InputRange R, WeaklyIncrementable O, CopyConstructible F,
      class Proj = identity>
    requires Writable<O, indirect_result_t<F&, projected<iterator_t<R>, Proj>>>
    constexpr unary_transform_result<safe_iterator_t<R>, O>
      transform(R&& r, 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_t<F&, projected<I1, Proj1>,
      projected<I2, Proj2>>>
    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 R1, InputRange R2, WeaklyIncrementable O,
      CopyConstructible F, class Proj1 = identity, class Proj2 = identity>
    requires Writable<O, indirect_result_t<F&,
      projected<iterator_t<R1>, Proj1>, projected<iterator_t<R2>, Proj2>>>
    constexpr binary_transform_result<safe_iterator_t<R1>, safe_iterator_t<R2>, O>
      transform(R1&& r1, R2&& r2, O result,
                F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

// 24.6.5, 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 R, class T1, class T2, class Proj = identity>
    requires Writable<iterator_t<R>, const T2&> &&
      IndirectRelation<ranges::equal_to<>, projected<iterator_t<R>, Proj>, const T1*>
    constexpr safe_iterator_t<R>
      replace(R&& r, 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 R, class T, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    requires Writable<iterator_t<R>, const T&>
    constexpr safe_iterator_t<R>
      replace_if(R&& r, 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*>
    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 R, class T1, class T2, OutputIterator<const T2&> O,
      class Proj = identity>
    requires IndirectlyCopyable<iterator_t<R>, O> &&
      IndirectRelation<ranges::equal_to<>, projected<iterator_t<R>, Proj>, const T1*>
    constexpr replace_copy_result<safe_iterator_t<R>, O>
      replace_copy(R&& r, 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>
    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 R, class T, OutputIterator<const T&> O, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    requires IndirectlyCopyable<iterator_t<R>, O>
    constexpr replace_copy_if_result<safe_iterator_t<R>, O>
      replace_copy_if(R&& r, O result, Pred pred, const T& new_value,
                      Proj proj = Proj{});
}

// 24.6.6, 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&> R>
    constexpr safe_iterator_t<R> fill(R&& r, const T& value);
  template<class T, OutputIterator<const T&> O>
    constexpr O fill_n(O first, iter_difference_t<O> n, const T& value);
}

// 24.6.7, 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, invoke_result_t<F&>>
    constexpr O generate(O first, S last, F gen);
  template<class R, CopyConstructible F>
      requires Invocable<F&> && OutputRange<R, invoke_result_t<F&>>
    constexpr safe_iterator_t<R> generate(R&& r, F gen);
  template<Iterator O, CopyConstructible F>
      requires Invocable<F&> && Writable<O, invoke_result_t<F&>>
    constexpr O generate_n(O first, iter_difference_t<O> n, F gen);
}

// 24.6.8, 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<ForwardIteratorPermutable 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 R, class T, class Proj = identity>
    requires Permutable<iterator_t<R>> &&
      IndirectRelation<ranges::equal_to<>, projected<iterator_t<R>, Proj>, const T*>
    constexpr safe_iterator_t<R>
      remove(R&& r, const T& value, Proj proj = Proj{});
  template<ForwardIteratorPermutable 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 R, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    requires Permutable<iterator_t<R>>
    constexpr safe_iterator_t<R>
      remove_if(R&& r, 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*>
    constexpr remove_copy_result<I, O>
      remove_copy(I first, S last, O result, const T& value, Proj proj = Proj{});
  template<InputRange R, WeaklyIncrementable O, class T, class Proj = identity>
    requires IndirectlyCopyable<iterator_t<R>, O> &&
      IndirectRelation<ranges::equal_to<>, projected<iterator_t<R>, Proj>, const T*>
    constexpr remove_copy_result<safe_iterator_t<R>, O>
      remove_copy(R&& r, 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>
    constexpr remove_copy_if_result<I, O>
      remove_copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});
  template<InputRange R, WeaklyIncrementable O, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    requires IndirectlyCopyable<iterator_t<R>, O>
    constexpr remove_copy_if_result<safe_iterator_t<R>, O>
      remove_copy_if(R&& r, O result, Pred pred, Proj proj = Proj{});
}

// 24.6.9, 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~~Permutable I, Sentinel<I> S, class Proj = identity,
      IndirectRelation<projected<I, Proj>> C = ranges::equal_to<>>
```

```
    requires Permutable<I>
    constexpr I unique(I first, S last, C comp = C{}, Proj proj = Proj{});
  template<ForwardRange R, class Proj = identity,
      IndirectRelation<projected<iterator_t<R>, Proj>> C = ranges::equal_to<>>
    requires Permutable<iterator_t<R>>
    constexpr safe_iterator_t<R>
      unique(R&& r, C comp = C{}, 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>> C = ranges::equal_to<>>
    requires IndirectlyCopyable<I, O> &&
      (ForwardIterator<I> ||
      (InputIterator<O> && Same<iter_value_t<I>, iter_value_t<O>>) ||
      IndirectlyCopyableStorable<I, O>)
    constexpr unique_copy_result<I, O>
      unique_copy(I first, S last, O result, C comp = C{}, Proj proj = Proj{});
  template<InputRange R, WeaklyIncrementable O, class Proj = identity,
      IndirectRelation<projected<iterator_t<R>, Proj>> C = ranges::equal_to<>>
    requires IndirectlyCopyable<iterator_t<R>, O> &&
      (ForwardIterator<iterator_t<R>> ||
      (InputIterator<O> && Same<iter_value_t<iterator_t<R>>, iter_value_t<O>>) ||
      IndirectlyCopyableStorable<iterator_t<R>, O>)
    constexpr unique_copy_result<safe_iterator_t<R>, O>
      unique_copy(R&& r, O result, C comp = C{}, Proj proj = Proj{});
}

// 24.6.10, 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 R>
    requires Permutable<iterator_t<R>>
    constexpr safe_iterator_t<R> reverse(R&& r);
```

```
}

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>
    constexpr reverse_copy_result<I, O>
      reverse_copy(I first, S last, O result);
  template<BidirectionalRange R, WeaklyIncrementable O>
    requires IndirectlyCopyable<iterator_t<R>, O>
    constexpr reverse_copy_result<safe_iterator_t<R>, O>
      reverse_copy(R&& r, O result);
}

// 24.6.11, rotate
template<class ForwardIterator>
  constexpr 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~~Permutable I, Sentinel<I> S>
    ~~requires Permutable<I>~~
    constexpr subrange<I> rotate(I first, I middle, S last);
  template<ForwardRange R>
    requires Permutable<iterator_t<R>>
    constexpr safe_subrange_t<R> rotate(R&& r, iterator_t<R> 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>
    constexpr rotate_copy_result<I, O>
      rotate_copy(I first, I middle, S last, O result);
```

```
  template<ForwardRange R, WeaklyIncrementable O>
    requires IndirectlyCopyable<iterator_t<R>, O>
    constexpr rotate_copy_result<safe_iterator_t<R>, O>
      rotate_copy(R&& r, iterator_t<R> middle, O result);
}

// 24.6.12, sample
[...]

// 24.6.13, shuffle
template<class RandomAccessIterator, class UniformRandomBitGenerator>
  void shuffle(RandomAccessIterator first,
               RandomAccessIterator last,
               UniformRandomBitGenerator&& g);

namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Gen>
    requires Permutable<I> &&
      UniformRandomBitGenerator<remove_reference_t<Gen>> &&
      ConvertibleTo<invoke_result_t<Gen&>, iter_difference_t<I>>
    I shuffle(I first, S last, Gen&& g);
  template<RandomAccessRange R, class Gen>
    requires Permutable<iterator_t<R>> &&
      UniformRandomBitGenerator<remove_reference_t<Gen>> &&
      ConvertibleTo<invoke_result_t<Gen&>, iter_difference_t<iterator_t<R>>>
    safe_iterator_t<R>
      shuffle(R&& r, Gen&& g);
}

// 24.6.14, shift
[...]

// 24.7, sorting and related operations
// 24.7.1, sorting
template<class RandomAccessIterator>
  constexpr void sort(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  constexpr 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 R, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    constexpr safe_iterator_t<R>
      sort(R&& r, 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 R, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    safe_iterator_t<R>
      stable_sort(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class RandomAccessIterator>
  constexpr void partial_sort(RandomAccessIterator first,
                              RandomAccessIterator middle,
                              RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  constexpr 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 R, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    constexpr safe_iterator_t<R>
      partial_sort(R&& r, iterator_t<R> middle, Comp comp = Comp{},
                   Proj proj = Proj{});
}

template<class InputIterator, class RandomAccessIterator>
  constexpr RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last);
template<class InputIterator, class RandomAccessIterator, class Compare>
  constexpr 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 R1, RandomAccessRange R2, class Comp = ranges::less<>,
      class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyCopyable<iterator_t<R1>, iterator_t<R2>> &&
        Sortable<iterator_t<R2>, Comp, Proj2> &&
        IndirectStrictWeakOrder<Comp, projected<iterator_t<R1>, Proj1>,
          projected<iterator_t<R2>, Proj2>>
    constexpr safe_iterator_t<R2>
      partial_sort_copy(R1&& r, R2&& result_r, 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 R, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    constexpr bool is_sorted(R&& r, 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 R, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    constexpr safe_iterator_t<R>
      is_sorted_until(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}

// 24.7.2, Nth element
template<class RandomAccessIterator>
  constexpr void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                             RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  constexpr 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 R, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    constexpr safe_iterator_t<R>
      nth_element(R&& r, iterator_t<R> nth, Comp comp = Comp{}, Proj proj = Proj{});
}

// 24.7.3, 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 R, class T, class Proj = identity,
      IndirectStrictWeakOrder<const T*, projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    constexpr safe_iterator_t<R>
      lower_bound(R&& r, 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 R, class T, class Proj = identity,
      IndirectStrictWeakOrder<const T*, projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    constexpr safe_iterator_t<R>
      upper_bound(R&& r, 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<>>
    constexpr subrange<I>
      equal_range(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
  template<ForwardRange R, class T, class Proj = identity,
      IndirectStrictWeakOrder<const T*, projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    constexpr safe_subrange_t<R>
      equal_range(R&& r, 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 R, class T, class Proj = identity,
      IndirectStrictWeakOrder<const T*, projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    constexpr bool binary_search(R&& r, const T& value, Comp comp = Comp{},
                                 Proj proj = Proj{});
}

// 24.7.4, 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 R, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr bool is_partitioned(R&& r, Pred pred, Proj proj = Proj{});
}

template<class ForwardIterator, class Predicate>
  constexpr 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~~Permutable 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 R, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    requires Permutable<iterator_t<R>>
    constexpr safe_iterator_t<R>
      partition(R&& r, 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 R, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    requires Permutable<iterator_t<R>>
    safe_iterator_t<R> stable_partition(R&& r, 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>
    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 R, WeaklyIncrementable O1, WeaklyIncrementable O2,
      class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    requires IndirectlyCopyable<iterator_t<R>, O1> &&
      IndirectlyCopyable<iterator_t<R>, O2>
    constexpr partition_copy_result<safe_iterator_t<R>, O1, O2>
      partition_copy(R&& r, 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 R, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr safe_iterator_t<R>
      partition_point(R&& r, Pred pred, Proj proj = Proj{});
}

// 24.7.5, 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>
    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 R1, InputRange R2, WeaklyIncrementable O, class Comp = ranges::less<>,
      class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<R1>, iterator_t<R2>, O, Comp, Proj1, Proj2>
    constexpr merge_result<safe_iterator_t<R1>, safe_iterator_t<R2>, O>
      merge(R1&& r1, R2&& r2, 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 R, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    safe_iterator_t<R>
      inplace_merge(R&& r, iterator_t<R> middle, Comp comp = Comp{},
                    Proj proj = Proj{});
}

// 24.7.6, 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 R1, InputRange R2, class Proj1 = identity,
      class Proj2 = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R1>, Proj1>,
        projected<iterator_t<R2>, Proj2>> Comp = ranges::less<>>
    constexpr bool includes(R1&& r1, R2&& r2, 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>
    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 R1, InputRange R2, WeaklyIncrementable O,
      class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<R1>, iterator_t<R2>, O, Comp, Proj1, Proj2>
    constexpr set_union_result<safe_iterator_t<R1>, safe_iterator_t<R2>, O>
      set_union(R1&& r1, R2&& r2, 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<class I1, class I2, class O>
  using set_intersection_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>
    constexpr set_intersection_result<I1, I2, O>
      set_intersection(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                       Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<InputRange R1, InputRange R2, WeaklyIncrementable O,
      class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<R1>, iterator_t<R2>, O, Comp, Proj1, Proj2>
    constexpr set_intersection_result<safe_iterator_t<R1>, safe_iterator_t<R2>, O>
      set_intersection(R1&& r1, R2&& r2, 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>
    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 R1, InputRange R2, WeaklyIncrementable O,
      class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<R1>, iterator_t<R2>, O, Comp, Proj1, Proj2>
    constexpr set_difference_result<safe_iterator_t<R1>, O>
      set_difference(R1&& r1, R2&& r2, 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>
    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 R1, InputRange R2, WeaklyIncrementable O,
      class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<R1>, iterator_t<R2>, O, Comp, Proj1, Proj2>
    constexpr set_symmetric_difference_result<safe_iterator_t<R1>, safe_iterator_t<R2>, O>
      set_symmetric_difference(R1&& r1, R2&& r2, O result, Comp comp = Comp{},
                               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

// 24.7.7, heap operations
template<class RandomAccessIterator>
  constexpr void push_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  constexpr 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 R, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    constexpr safe_iterator_t<R>
      push_heap(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class RandomAccessIterator>
  constexpr void pop_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  constexpr 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 R, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    constexpr safe_iterator_t<R>
      pop_heap(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}

template<class RandomAccessIterator>
  constexpr void make_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  constexpr 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 R, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    constexpr safe_iterator_t<R>
      make_heap(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}
```

```
template<class RandomAccessIterator>
  constexpr void sort_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  constexpr 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 R, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    constexpr safe_iterator_t<R>
      sort_heap(R&& r, 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 R, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    constexpr bool is_heap(R&& r, 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 R, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    constexpr safe_iterator_t<R>
      is_heap_until(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}
```

```
// 24.7.8, 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> r);
template<class T, class Compare>
  constexpr T min(initializer_list<T> r, 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> r, Comp comp = Comp{}, Proj proj = Proj{});
  template<InputRange R, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    requires Copyable<iter_value_t<iterator_t<R>>>
    constexpr iter_value_t<iterator_t<R>>
      min(R&& r, 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> r);
template<class T, class Compare>
  constexpr T max(initializer_list<T> r, 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> r, Comp comp = Comp{}, Proj proj = Proj{});
  template<InputRange R, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    requires Copyable<iter_value_t<iterator_t<R>>>
    constexpr iter_value_t<iterator_t<R>>
      max(R&& r, 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> r);
template<class T, class Compare>
  constexpr pair<T, T> minmax(initializer_list<T> r, 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 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 minmax_result<T>
      minmax(initializer_list<T> r, Comp comp = Comp{}, Proj proj = Proj{});
  template<InputRange R, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    requires Copyable<iter_value_t<iterator_t<R>>>
    constexpr minmax_result<iter_value_t<iterator_t<R>>>
      minmax(R&& r, 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 R, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    constexpr safe_iterator_t<R>
      min_element(R&& r, 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 R, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    constexpr safe_iterator_t<R>
      max_element(R&& r, 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<>>
    constexpr minmax_result<I>
      minmax_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<ForwardRange R, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    constexpr minmax_result<safe_iterator_t<R>>
      minmax_element(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}

// 24.7.9, bounded value
[...]

// 24.7.10, 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 R1, InputRange R2, class Proj1 = identity,
      class Proj2 = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R1>, Proj1>,
        projected<iterator_t<R2>, Proj2>> Comp = ranges::less<>>
    constexpr bool
      lexicographical_compare(R1&& r1, R2&& r2, Comp comp = Comp{},
                              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}

// 24.7.11, three-way comparison algorithms
[...]

// 24.7.12, permutations
template<class BidirectionalIterator>
```

```
    constexpr bool next_permutation(BidirectionalIterator first,
                                    BidirectionalIterator last);
  template<class BidirectionalIterator, class Compare>
    constexpr 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 R, class Comp = ranges::less<>,
        class Proj = identity>
      requires Sortable<iterator_t<R>, Comp, Proj>
      constexpr bool
        next_permutation(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
  }

  template<class BidirectionalIterator>
    constexpr bool prev_permutation(BidirectionalIterator first,
                                    BidirectionalIterator last);
  template<class BidirectionalIterator, class Compare>
    constexpr 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 R, class Comp = ranges::less<>,
        class Proj = identity>
      requires Sortable<iterator_t<R>, Comp, Proj>
      constexpr bool
        prev_permutation(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
  }
}
```

24.3 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 Clause 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
Change: satisfymeet the requirements of a mutable iterator (22.3). [*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 (23.6.2) 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.

14 The number and order of template parameters for algorithm declarations is unspecified, except where explicitly stated otherwise.

15 The class templates `binary_transform_result`, `copy_result`, `for_each_result`, `minmax_result`, `mismatch_result`, and `partition_copy_result` have the template parameters, data members, and special members specified above. They have no base classes or members other than those specified.

24.5 Non-modifying sequence operations [alg.nonmodifying]

24.5.1 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 R, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr bool all_of(R&& r, 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 any projection.

24.5.2 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 R, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr bool any_of(R&& r, 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 any projection.

24.5.3 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 R, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr bool none_of(R&& r, 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 any projection.

24.5.4 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>
    constexpr for_each_result<I, Fun>
      for_each(I first, S last, Fun f, Proj proj = Proj{});
  template<InputRange R, class Proj = identity,
      IndirectUnaryInvocable<projected<iterator_t<R>, Proj>> Fun>
    constexpr for_each_result<safe_iterator_t<R>, Fun>
      for_each(R&& r, 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.

[...]

24.5.5 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 R, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to<>, projected<iterator_t<R>, Proj>, const T*>
    constexpr safe_iterator_t<R>
      find(R&& r, 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 R, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr safe_iterator_t<R>
      find_if(R&& r, 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 R, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr safe_iterator_t<R>
      find_if_not(R&& r, 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 any projection.

24.5.6 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>
    requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
    constexpr subrange<I1>
      find_end(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<ForwardRange R1, ForwardRange R2,
      class Pred = ranges::equal_to<>, class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2>
    constexpr safe_subrange_t<R1>
      find_end(R1&& r1, R2&& r2, 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 any projections.

24.5.7 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 R1, ForwardRange R2, class Proj1 = identity,
      class Proj2 = identity,
      IndirectRelation<projected<iterator_t<R1>, Proj1>,
        projected<iterator_t<R2>, Proj2>> Pred = ranges::equal_to<>>
    constexpr safe_iterator_t<R1>
      find_first_of(R1&& r1, R2&& r2,
                    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 any projections.

24.5.8 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 R, class Proj = identity,
      IndirectRelation<projected<iterator_t<R>, Proj>> Pred = ranges::equal_to<>>
    constexpr safe_iterator_t<R>
      adjacent_find(R&& r, 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 any projection. For the overloads with an `ExecutionPolicy`, $\mathscr{O}$(`last - first`) applications of the corresponding predicate.

24.5.9 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_t<I>
      count(I first, S last, const T& value, Proj proj = Proj{});
  template<InputRange R, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to<>, projected<iterator_t<R>, Proj>, const T*>
    constexpr iter_difference_t<iterator_t<R>>
      count(R&& r, const T& value, Proj proj = Proj{});
  template<InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr iter_difference_t<I>
      count_if(I first, S last, Pred pred, Proj proj = Proj{});
  template<InputRange R, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr iter_difference_t<iterator_t<R>>
      count_if(R&& r, 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 any projection.

24.5.10 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<>>
    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 R1, InputRange R2,
      class Proj1 = identity, class Proj2 = identity,
      IndirectRelation<projected<iterator_t<R1>, Proj1>,
        projected<iterator_t<R2>, Proj2>> Pred = ranges::equal_to<>>
    constexpr mismatch_result<safe_iterator_t<R1>, safe_iterator_t<R2>>
      mismatch(R1&& r1, R2&& r2, 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 `last2` be `first2 + (last1 - first1)` for the overloads with no parameter `last2` or `r2`.

3 Let E be:

(3.1) — `!(*(first1 + n) == *(first2 + n))` for the overloads with no parameter `pred`,

(3.2) — `pred(*(first1 + n), *(first2 + n)) == false` for the overloads with a parameter `pred` and no parameter `proj1`,

(3.3) — `!invoke(pred, invoke(proj1, *(first1 + n)), invoke(proj2, *(first2 + n)))` for the overloads with both parameters `pred` and `proj1`.

4 *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,~~

(4.1) — ~~`!(*(first1 + n) == *(first2 + n))` or~~

(4.2) — ~~`pred(*(first1 + n), *(first2 + n)) == false`,~~

or `min(last1 - first1, last2 - first2)` if no such integer exists.

5 *Complexity:* At most `min(last1 - first1, last2 - first2)` applications of the corresponding predicate and any projections.

24.5.11 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 R1, InputRange R2, class Pred = ranges::equal_to<>,
      class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2>
    constexpr bool equal(R1&& r1, R2&& r2, 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 `r2`,
- (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 (22.3.5.6) for the overloads in namespace `std`, or

(4.2) — pairwise model `SizedSentinel` (22.3.4.8) 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 any projections.

(4.3.1) — ~~if `InputIterator1` and `InputIterator2` meet the *Cpp17RandomAccessIterator* requirements (22.3.5.6) 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,~~

24.5.12 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 R1, ForwardRange R2, class Pred = ranges::equal_to<>,
      class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2>
    constexpr bool is_permutation(R1&& r1, R2&& r2, 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`,

(6.2) — `S2` and `I2` model `SizedSentinel`, and

(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`.

24.5.13 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>
      search(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<ForwardRange R1, ForwardRange R2, class Pred = ranges::equal_to<>,
      class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2>
    constexpr safe_subrange_t<R1>
      search(R1&& r1, R2&& r2, 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 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))).`

(5.3) — Returns `{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_t<I> count,
               const T& value, Pred pred = Pred{}, Proj proj = Proj{});
  template<ForwardRange R, class T, class Pred = ranges::equal_to<>,
      class Proj = identity>
    requires IndirectlyComparable<iterator_t<R>, const T*, Pred, Proj>
    constexpr safe_subrange_t<R>
      search_n(R&& r, iter_difference_t<iterator_t<R>> 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 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)`. Returns `{last, last}` if no such iterator is found.

13 *Complexity:* At most `last - first` applications of the corresponding predicate and projection.

[...]

24.6 Mutating sequence operations [alg.modifying.operations]

24.6.1 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>
    constexpr copy_result<I, O>
      copy(I first, S last, O result);
  template<InputRange R, WeaklyIncrementable O>
    requires IndirectlyCopyable<iterator_t<R>, O>
    constexpr copy_result<safe_iterator_t<R>, O>
      copy(R&& r, 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)`~~ $n <$ `(last - first)`, performs `*(result +` ~~`n`~~ n`) = *(first +` ~~`n`~~ n`)`.

3 *Returns:*

(3.1) — `result + (last - first)` for the overload in namespace `std`, or

(3.2) — `{last, result + (last - first)}` for the overloads 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)`~~ $n <$ `(last - first)`, performs `*(result +` ~~`n`~~ n`) = *(first +` ~~`n`~~ 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>
    constexpr copy_n_result<I, O>
      copy_n(I first, iter_difference_t<I> n, O result);
}
```

[Editor's note: This wording incorporates the PR for stl2#498).]

9 Let M be $\max(\texttt{n}, 0)$.

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 + M, result + 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>
    constexpr copy_if_result<I, O>
      copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});
  template<InputRange R, WeaklyIncrementable O, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    requires IndirectlyCopyable<iterator_t<R>, O>
    constexpr copy_if_result<safe_iterator_t<R>, O>
      copy_if(R&& r, O result, Pred pred, Proj proj = Proj{});
}
```

13 Let E be:

(13.1) — `bool(pred(*i))` for the overloads in namespace `std`, or

(13.2) — `bool(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 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 *Cpp17MoveConstructible* ([tab:moveconstructible]). — *end note*]

15 *Effects:* Copies all of the elements referred to by the iterator `i` in the range `[first, last)` for which ~~`pred(*i)`~~E 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 any 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>
    constexpr copy_backward_result<I1, I2>
      copy_backward(I1 first, S1 last, I2 result);
  template<BidirectionalRange R, BidirectionalIterator I>
    requires IndirectlyCopyable<iterator_t<R>, I>
    constexpr copy_backward_result<safe_iterator_t<R>, I>
      copy_backward(R&& r, 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)~~ $n \leq$ (last - first), performs *(result - ~~n~~n) = *(last - ~~n~~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.

24.6.2 Move [alg.move]

```
template<class InputIterator, class OutputIterator>
  constexpr OutputIterator move(InputIterator first, InputIterator last,
                                OutputIterator result);

namespace ranges {
  template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
    requires IndirectlyMovable<I, O>
    constexpr move_result<I, O>
      move(I first, S last, O result);
  template<InputRange R, WeaklyIncrementable O>
    requires IndirectlyMovable<iterator_t<R>, O>
    constexpr move_result<safe_iterator_t<R>, O>
      move(R&& r, O result);
}
```

1 Let E be

(1.1) — std::move(*(first + n)) for the overload in namespace std, or

(1.2) — ranges::iter_move(first + n) for the overloads in namespace ranges.

2 *Requires:* result shall not be in the range [first, last).

3 *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)~~ $n <$ (last - first), performs ~~*(result + n) = std::move(*(first + n))~~ *(result + n) = E.

4 *Returns:*

(4.1) — result + (last - first) for the overload in namespace std, or

(4.2) — {last, result + (last - first)} for the overloads in namespace ranges.

5 *Complexity:* Exactly last - first ~~move~~ assignments.

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 move(ExecutionPolicy&& policy,
                        ForwardIterator1 first, ForwardIterator1 last,
                        ForwardIterator2 result);
```

6 *Requires:* The ranges [first, last) and [result, result + (last - first)) shall not overlap.

7 *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)).

8 *Returns:* result + (last - first).

9 *Complexity:* Exactly last - first assignments.

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
  constexpr BidirectionalIterator2
    move_backward(BidirectionalIterator1 first, BidirectionalIterator1 last,
                  BidirectionalIterator2 result);
```

5) copy_backward should be used instead of copy when last is in the range [result - (last - first), result).

```
namespace ranges {
  template<BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
    requires IndirectlyMovable<I1, I2>
    constexpr move_backward_result<I1, I2>
      move_backward(I1 first, S1 last, I2 result);
  template<BidirectionalRange R, BidirectionalIterator I>
    requires IndirectlyMovable<iterator_t<R>, I>
    constexpr move_backward_result<safe_iterator_t<R>, I>
      move_backward(R&& r, I result);
}
```

10 Let E be

(10.1) — `std::move(*(last - n))` for the overload in namespace `std`, or

(10.2) — `ranges::iter_move(last - n)` for the overloads in namespace `ranges`.

11 *Requires:* `result` shall not be in the range `(first, last]`.

12 *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)`~~ $n \leq$ `(last - first)`, performs ~~`*(result - n) = std::move(*(last - n))`~~ `*(result - n) = E`.

13 *Returns:*

(13.1) — `result - (last - first)` for the overload in namespace `std`, or

(13.2) — `{last, result - (last - first)}` for the overloads in namespace `ranges`.

14 *Complexity:* Exactly `last - first` assignments.

24.6.3 Swap [alg.swap]

[Editor's note: This wording integrates the PR for stl2#415.]

```
template<class ForwardIterator1, class ForwardIterator2>
  constexpr 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);
```

```
namespace ranges {
  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2>
    requires IndirectlySwappable<I1, I2>
    constexpr swap_ranges_result<I1, I2>
      swap_ranges(I1 first1, S1 last1, I2 first2, S2 last2);
  template<InputRange R1, InputRange R2>
    requires IndirectlySwappable<iterator_t<R1>, iterator_t<R2>>
    constexpr swap_ranges_result<safe_iterator_t<R1>, safe_iterator_t<R2>>
      swap_ranges(R1&& r1, R2&& r2);
}
```

1 Let:

(1.1) — `last2` be `first2 + (last1 - first1)` for the overloads with no parameter named `last2`, and

(1.2) — M be $min($`last1 - first1`, `last2 - first2`$)$.

2 *Requires:* The two ranges `[first1, last1)` and `[first2,` ~~`first2 + (last1 - first1)`~~ `last2)` shall not overlap. For the overloads in namespace `std`, `*(first1 + `~~`n`~~n`)` shall be swappable with ([swappable.requirements]) `*(first2 + `~~`n`~~n`)`.

3 *Effects:* For each non-negative integer ~~`n < (last1 - first1)`~~ $n < M$ performs: ~~`swap(*(first1 + n), *(first2 + n))`~~

6) `move_backward` should be used instead of move when ~~last~~`last` is in the range `[result - (last - first), result)`.

(3.1) — `swap(*(first1 + n), *(first2 + n))` for the overloads in namespace `std`, or

(3.2) — `ranges::iter_swap(first1 + n, first2 + n)` for the overloads in namespace `ranges`.

4 *Returns:* ~~`first2 + (last1 - first1)`~~

(4.1) — `last2` for the overloads in namespace `std`, or

(4.2) — `{first1 + M, first2 + M}` for the overloads in namespace `ranges`.

5 *Complexity:* Exactly ~~`last1 - first1`~~M swaps.

```
template<class ForwardIterator1, class ForwardIterator2>
  constexpr void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

[...]

24.6.4 Transform [alg.transform]

```
template<class InputIterator, class OutputIterator,
         class UnaryOperation>
  constexpr OutputIterator
    transform(InputIterator first1, InputIterator last1,
              OutputIterator result, UnaryOperation op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class UnaryOperation>
  ForwardIterator2
    transform(ExecutionPolicy&& exec,
              ForwardIterator1 first1, ForwardIterator1 last1,
              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);

namespace ranges {
  template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
      CopyConstructible F, class Proj = identity>
    requires Writable<O, indirect_result_t<F&, projected<I, Proj>>>
    constexpr unary_transform_result<I, O>
      transform(I first1, S last1, O result, F op, Proj proj = Proj{});
  template<InputRange R, WeaklyIncrementable O, CopyConstructible F,
      class Proj = identity>
    requires Writable<O, indirect_result_t<F&,
      projected<iterator_t<R>, Proj>>>
    constexpr unary_transform_result<safe_iterator_t<R>, O>
      transform(R&& r, O result, F op, Proj proj = Proj{});
  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      WeaklyIncrementable O, CopyConstructible F, class Proj1 = identity,
      class Proj2 = identity>
    requires Writable<O, indirect_result_t<F&, projected<I1, Proj1>,
      projected<I2, Proj2>>>
    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 R1, InputRange R2, WeaklyIncrementable O,
      CopyConstructible F, class Proj1 = identity, class Proj2 = identity>
    requires Writable<O, indirect_result_t<F&,
      projected<iterator_t<R1>, Proj1>, projected<iterator_t<R2>, Proj2>>>
    constexpr binary_transform_result<safe_iterator_t<R1>, safe_iterator_t<R2>, O>
      transform(R1&& r1, R2&& r2, O result,
                F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

1 Let:

- (1.1) `last2` be `first2 + (last1 - first1)` for the overloads with parameter `first2` but no parameter `last2`,
- (1.2) N be `last1 - first1` for unary transforms, or min(`last1 - first1`, `last2 - first2`) for binary transforms, and
- (1.3) E be
  - (1.3.1) `op(*(first1 + (i - result)))` for unary transforms defined in namespace `std`,
  - (1.3.2) `binary_op(*(first1 + (i - result)), *(first2 + (i - result)))` for binary transforms defined in namespace `std`,
  - (1.3.3) `invoke(op, invoke(proj, *(first1 + (i - result))))` for unary transforms defined in namespace `ranges`, or
  - (1.3.4) `invoke(binary_op, invoke(proj1, *(first1 + (i - result))), invoke(proj2, *(first2 + (i - result))))` for binary transforms defined in namespace `ranges`.

2 *Requires:* `op` and `binary_op` shall not invalidate iterators or subranges, or modify elements in the ranges

- (2.1) `[first1, ~~last1~~first1 + N]`,
- (2.2) `[first2, first2 + ~~(last1 - first1)~~N]`, and
- (2.3) `[result, result + ~~(last1 - first1)~~N]`.[7]

3 *Effects:* Assigns through every iterator `i` in the range `[result, result + ~~(last1 - first1)~~N)` a new corresponding value equal to E ~~`op(*(first1 + (i - result)))` or `binary_op(*(first1 + (i - result)), *(first2 + (i - result)))`~~.

4 *Returns:*

- (4.1) `result + ~~(last1 - first1)~~N` for the overloads defined in namespace `std`,
- (4.2) `{first1 + N, result + N}` for unary transforms defined in namespace `ranges`, or
- (4.3) `{first1 + N, first2 + N, result + N}` for binary transforms defined in namespace `ranges`.

5 *Complexity:* Exactly ~~`last1 - first1`~~N applications of `op` or `binary_op`, and any projections. This requirement also applies to the overload with an `ExecutionPolicy`.

6 *Remarks:* `result` may be equal ~~to first in case of unary transform, or~~ to `first1` or `first2` ~~in case of binary transform~~.

24.6.5 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);
```

7) The use of fully closed ranges is intentional.

```
template<class ExecutionPolicy, class ForwardIterator, class Predicate, class T>
  void replace_if(ExecutionPolicy&& exec,
                  ForwardIterator first, ForwardIterator last,
                  Predicate pred, const T& new_value);

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 R, class T1, class T2, class Proj = identity>
    requires Writable<iterator_t<R>, const T2&> &&
      IndirectRelation<ranges::equal_to<>, projected<iterator_t<R>, Proj>, const T1*>
    constexpr safe_iterator_t<R>
      replace(R&& r, 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 R, class T, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    requires Writable<iterator_t<R>, const T&>
    constexpr safe_iterator_t<R>
      replace_if(R&& r, Pred pred, const T& new_value, Proj proj = Proj{});
}
```

1 Let E be

- (1.1) — `bool(*i == old_value)` for the `replace`,
- (1.2) — `bool(pred(*i))` for `replace_if`,
- (1.3) — `bool(invoke(proj, *i) == old_value)` for `ranges::replace`, or
- (1.4) — `bool(invoke(pred, invoke(proj, *i)))` for `ranges::replace_if`.

2 *Requires:* The expression `*first = new_value` shall be valid.

3 *Effects:* Substitutes elements referred by the iterator `i` in the range `[first, last)` with `new_value`, when E is `true` ~~the following corresponding conditions hold: `*i == old_value`, `pred(*i) != false`~~.

4 *Returns:* `last` for the overloads in namespace `ranges`.

5 *Complexity:* Exactly `last - first` applications of the corresponding predicate and any 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);

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*>
    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 R, class T1, class T2, OutputIterator<const T2&> O,
      class Proj = identity>
    requires IndirectlyCopyable<iterator_t<R>, O> &&
      IndirectRelation<ranges::equal_to<>, projected<iterator_t<R>, Proj>, const T1*>
    constexpr replace_copy_result<safe_iterator_t<R>, O>
      replace_copy(R&& r, 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>
    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 R, class T, OutputIterator<const T&> O, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    requires IndirectlyCopyable<iterator_t<R>, O>
    constexpr replace_copy_if_result<safe_iterator_t<R>, O>
      replace_copy_if(R&& r, O result, Pred pred, const T& new_value,
                      Proj proj = Proj{});
}
```

6 Let E be

(6.1) — `bool(*(first + (i - result)) == old_value)` for `replace_copy`,

(6.2) — `bool(pred(*(first + (i - result))))` for `replace_copy_if`,

(6.3) — `bool(invoke(proj, *(first + (i - result))) == old_value)` for `ranges::replace_copy`,

(6.4) — `bool(invoke(pred, invoke(proj, *(first + (i - result)))))` for `ranges::replace_copy_if`.

7 *Requires:* The results of the expressions `*first` and `new_value` shall be writable (22.3.1) 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 E holds. the following corresponding conditions hold:

```
*(first + (i - result)) == old_value
pred(*(first + (i - result))) != false
```

9 *Returns:*

(9.1) — `result + (last - first)` for the overloads in namespace `std`, or

(9.2) — `{last, result + (last - first)}` for the overloads in namespace `ranges`.

10 *Complexity:* Exactly `last - first` applications of the corresponding predicate and any projection.

24.6.6 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);

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&> R>
    constexpr safe_iterator_t<R> fill(R&& r, const T& value);
  template<class T, OutputIterator<const T&> O>
    constexpr O fill_n(O first, iter_difference_t<O> n, const T& value);
}
```

1 Let N be $\max(0, \texttt{n})$ for the `fill_n` algorithms, and `last - first` for the `fill` algorithms.

2 *Requires:* The expression `value` shall be writable (22.3.1) to the output iterator. The type `Size` shall be convertible to an integral type ([conv.integral], [class.conv]).

3 *Effects:* Assigns `value` through all the iterators in the range `[first,` first + N). ~~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.~~

4 *Returns:* `first +` N. ~~`fill_n` returns `first + n` for non-negative values of `n` and `first` for negative values.~~

5 *Complexity:* Exactly N ~~`last - first`, `n`, or 0~~ assignments~~, respectively~~.

24.6.7 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);

namespace ranges {
  template<Iterator O, Sentinel<O> S, CopyConstructible F>
      requires Invocable<F&> && Writable<O, invoke_result_t<F&>>
    constexpr O generate(O first, S last, F gen);
  template<class R, CopyConstructible F>
      requires Invocable<F&> && OutputRange<R, invoke_result_t<F&>>
    constexpr safe_iterator_t<R> generate(R&& r, F gen);
  template<Iterator O, CopyConstructible F>
      requires Invocable<F&> && Writable<O, invoke_result_t<F&>>
    constexpr O generate_n(O first, iter_difference_t<O> n, F gen);
}
```

1 Let N be $\max(0, \texttt{n})$ for the `generate_n` algorithms, and `last - first` for the `generate` algorithms.

2 *Requires:* `gen` takes no arguments, `Size` shall be convertible to an integral type ([conv.integral], [class.conv]).

3 *Effects:* Invokes the function object `gen` and assigns the return value through all the iterators in the range `[first,` first + N).

~~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.~~

4 *Returns:* `first +` N.

~~generate_n returns first + n for non-negative values of n and first for negative values.~~

5 *Complexity:* Exactly N ~~last - first, n, or 0~~ invocations of `gen` and assignments~~, respectively~~.

24.6.8 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);

namespace ranges {
  template<ForwardIteratorPermutable 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 R, class T, class Proj = identity>
    requires Permutable<iterator_t<R>> &&
      IndirectRelation<ranges::equal_to<>, projected<iterator_t<R>, Proj>, const T*>
    constexpr safe_iterator_t<R>
      remove(R&& r, const T& value, Proj proj = Proj{});
  template<ForwardIteratorPermutable 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 R, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    requires Permutable<iterator_t<R>>
    constexpr safe_iterator_t<R>
      remove_if(R&& r, Pred pred, Proj proj = Proj{});
}
```

1 Let E be

(1.1) — `bool(*i == value)` for `remove`,

(1.2) — `bool(pred(*i))` for `remove_if`,

(1.3) — `bool(invoke(proj, *i) == value)` for `ranges::remove`, or

(1.4) — `bool(invoke(pred, invoke(proj, *i)))` for `ranges::remove_if`.

2 *Requires:* ~~The~~For the algorithms in namespace `std`, the type of `*first` shall ~~satisfy~~meet the *Cpp17MoveAssignable* requirements ([tab:moveassignable]).

3 *Effects:* Eliminates all the elements referred to by iterator `i` in the range `[first, last)` for which E holds ~~the following corresponding conditions hold: *i == value, pred(*i) != false~~.

4 *Returns:* The end of the resulting range.

5 *Remarks:* Stable ([algorithm.stable]).

6 *Complexity:* Exactly `last - first` applications of the corresponding predicate and any projection.

7 [*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);

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*>
    constexpr remove_copy_result<I, O>
      remove_copy(I first, S last, O result, const T& value, Proj proj = Proj{});
  template<InputRange R, WeaklyIncrementable O, class T, class Proj = identity>
    requires IndirectlyCopyable<iterator_t<R>, O> &&
      IndirectRelation<ranges::equal_to<>, projected<iterator_t<R>, Proj>, const T*>
    constexpr remove_copy_result<safe_iterator_t<R>, O>
      remove_copy(R&& r, 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>
    constexpr remove_copy_if_result<I, O>
      remove_copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});
  template<InputRange R, WeaklyIncrementable O, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    requires IndirectlyCopyable<iterator_t<R>, O>
    constexpr remove_copy_if_result<safe_iterator_t<R>, O>
      remove_copy_if(R&& r, O result, Pred pred, Proj proj = Proj{});
}
```

8 Let E be

(8.1) — `bool(*i == value)` for `remove_copy`,

(8.2) — `bool(pred(*i))` for `remove_copy_if`,

(8.3) — `bool(invoke(proj, *i) == value)` for `ranges::remove_copy`, or

(8.4) — `bool(invoke(pred, invoke(proj, *i)))` for `ranges::remove_copy_if`.

9 Let N be the number of elements in `[first, last)` for which E is `false`.

10 *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~~does not meet the *Cpp17MoveConstructible* ([tab:moveconstructible]) requirements. — *end note*]

11 *Effects:* Copies all the elements referred to by the iterator `i` in the range `[first, last)` for which E is `false` the following corresponding conditions do not hold: `*i == value, pred(*i) != false`.

12 *Returns:* The end of the resulting range.

(12.1) — `result + ` N, for the algorithms in namespace `std`, or

(12.2) — `{last, result + ` N`}`, for the algorithms in namespace `ranges`.

13 *Complexity:* Exactly `last - first` applications of the corresponding predicate and any projection.

14 *Remarks:* Stable ([algorithm.stable]).

24.6.9 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);

namespace ranges {
  template<ForwardIteratorPermutable I, Sentinel<I> S, class Proj = identity,
      IndirectRelation<projected<I, Proj>> C = ranges::equal_to<>>
    requires Permutable<I>
    constexpr I unique(I first, S last, C comp = C{}, Proj proj = Proj{});
  template<ForwardRange R, class Proj = identity,
      IndirectRelation<projected<iterator_t<R>, Proj>> C = ranges::equal_to<>>
    requires Permutable<iterator_t<R>>
    constexpr safe_iterator_t<R>
      unique(R&& r, C comp = C{}, Proj proj = Proj{});
}
```

1 Let `pred` be `equal_to<>{}` for the overloads with no parameter `pred`, and let E be

(1.1) — `bool(pred(*(i - 1), *i))` for the overloads in namespace `std`, or

(1.2) — `bool(invoke(comp, invoke(proj, *(i - 1)), invoke(proj, *i)))` for the overloads in namespace `ranges`.

2 *Requires:* The comparison function For the overloads in namepace `std`, `pred` shall be an equivalence relation. The and the type of `*first` shall satisfymeet the *Cpp17MoveAssignable* requirements ([tab:moveassignable]).

3 *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 E is `true`. the following conditions hold: `*(i - 1) == *i` or `pred(*(i - 1), *i) != false`.

4 *Returns:* The end of the resulting range.

5 *Complexity:* For nonempty ranges, exactly `(last - first) - 1` applications of the corresponding predicate and no more than twice as many applications of any 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);

namespace ranges {
  template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
      class Proj = identity, IndirectRelation<projected<I, Proj>> C = ranges::equal_to<>>
    requires IndirectlyCopyable<I, O> &&
      (ForwardIterator<I> ||
      (InputIterator<O> && Same<iter_value_t<I>, iter_value_t<O>>) ||
      IndirectlyCopyableStorable<I, O>)
    constexpr unique_copy_result<I, O>
      unique_copy(I first, S last, O result, C comp = C{}, Proj proj = Proj{});
  template<InputRange R, WeaklyIncrementable O, class Proj = identity,
      IndirectRelation<projected<iterator_t<R>, Proj>> C = ranges::equal_to<>>
    requires IndirectlyCopyable<iterator_t<R>, O> &&
      (ForwardIterator<iterator_t<R>> ||
      (InputIterator<O> && Same<iter_value_t<iterator_t<R>>, iter_value_t<O>>) ||
      IndirectlyCopyableStorable<iterator_t<R>, O>)
    constexpr unique_copy_result<safe_iterator_t<R>, O>
      unique_copy(R&& r, O result, C comp = C{}, Proj proj = Proj{});
}
```

6 Let `pred` be `equal_to<>{}` for the overloads in namespace `std` with no parameter `pred`, and let E be

(6.1) — `bool(pred(*i, *(i - 1)))` for the overloads in namespace `std`, or

(6.2) — `bool(invoke(comp, invoke(proj, *i), invoke(proj, *(i - 1))))` for the overloads in namespace `ranges`.

7 *Requires:*

(7.1) — ~~The comparison function shall be an equivalence relation.~~

(7.2) — The ranges `[first, last)` and `[result, result + (last - first))` shall not overlap.

(7.3) — For the overloads in namespace `std`:

(7.3.1) — The comparison function shall be an equivalence relation.

(7.3.2) — The expression `*result = *first` shall be valid.

(7.3.3) — For the overloads with no `ExecutionPolicy`, let `T` be the value type of `InputIterator`. If `InputIterator` meets the ~~forward iterator~~ *Cpp17ForwardIterator* requirements, then there are no additional requirements for `T`. Otherwise, if `OutputIterator` meets the ~~forward iterator~~ *Cpp17ForwardIterator* requirements and its value type is the same as `T`, then `T` shall ~~be~~meet the *Cpp17CopyAssignable* ([tab:copyassignable]) requirements. Otherwise, `T` shall ~~be both~~meet the *Cpp17CopyConstructible* ([tab:copyconstructible]) and *Cpp17CopyAssignable* requirements. [*Note:* For the overloads with an `ExecutionPolicy`, there may be a performance cost if the value type of `ForwardIterator1` ~~is not both~~ does not meet the *Cpp17CopyConstructible* and *Cpp17CopyAssignable* requirements. — *end note*]

8 *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 E holds. ~~the following corresponding conditions hold: `*i == *(i - 1)` or `pred(*i, *(i - 1)) != false`.~~

9 *Returns:* ~~The end of the resulting range.~~

(9.1) — `result +` N for the overloads in namespace `std`, or

(9.2) — `{last, result +` N`}` for the overloads in namespace `ranges`

where N is the number of groups of equal elements.

10 *Complexity:* For nonempty ranges, exactly `last - first - 1` applications of the corresponding predicate and no more than twice as many applications of any projection.

24.6.10 Reverse [alg.reverse]

```
template<class BidirectionalIterator>
  constexpr void reverse(BidirectionalIterator first, BidirectionalIterator last);
template<class ExecutionPolicy, class BidirectionalIterator>
  void reverse(ExecutionPolicy&& exec,
               BidirectionalIterator first, BidirectionalIterator last);

namespace ranges {
  template<BidirectionalIterator I, Sentinel<I> S>
    requires Permutable<I>
    constexpr I reverse(I first, S last);
  template<BidirectionalRange R>
    requires Permutable<iterator_t<R>>
    constexpr safe_iterator_t<R> reverse(R&& r);
}
```

1 *Requires:* For the overloads in namespace `std`, `BidirectionalIterator` shall ~~satisfy~~meet the *Cpp17ValueSwappable* requirements ([swappable.requirements]).

2 *Effects:* For each non-negative integer `i < (last - first) / 2`, applies `std::iter_swap`, or `ranges::iter_swap` for the overloads in namespace `ranges`, to all pairs of iterators `first + i, (last - i) - 1`.

3 *Returns:* `last` for the overloads in namespace `ranges`.

4 *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);

namespace ranges {
  template<BidirectionalIterator I, Sentinel<I> S, WeaklyIncrementable O>
    requires IndirectlyCopyable<I, O>
    constexpr reverse_copy_result<I, O>
      reverse_copy(I first, S last, O result);
  template<BidirectionalRange R, WeaklyIncrementable O>
    requires IndirectlyCopyable<iterator_t<R>, O>
    constexpr reverse_copy_result<safe_iterator_t<R>, O>
      reverse_copy(R&& r, O result);
}
```

5 *Requires:* The ranges `[first, last)` and `[result, result + (last - first))` shall not overlap.

6 *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)`.

7 *Returns:*

(7.1) — `result + (last - first)` for the overloads in namespace `std`, or

(7.2) — `{last, result + (last - first)}` for the overloads in namespace `ranges`.

8 *Complexity:* Exactly `last - first` assignments.

24.6.11 Rotate [alg.rotate]

[Editor's note: This wording incorporates the PR for stl2#526).]

```
template<class ForwardIterator>
  constexpr ForwardIterator
    rotate(ForwardIterator first, ForwardIterator middle, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator
    rotate(ExecutionPolicy&& exec,
           ForwardIterator first, ForwardIterator middle, ForwardIterator last);

namespace ranges {
  template<ForwardIteratorPermutable I, Sentinel<I> S>
    requires Permutable<I>
    constexpr subrange<I> rotate(I first, I middle, S last);
  template<ForwardRange R>
    requires Permutable<iterator_t<R>>
    constexpr safe_subrange_t<R> rotate(R&& r, iterator_t<R> middle);
}
```

1 *Requires:* `[first, middle)` and `[middle, last)` shall be valid ranges. For the overloads in namespace `std`, `ForwardIterator` shall ~~satisfy~~meet the *Cpp17ValueSwappable* requirements ([swappable.requirements])~~.~~ ~~The~~, and the type of `*first` shall ~~satisfy~~meet 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)`. [*Note:* This is a left rotate. — *end note*]

3 *Returns:*

(3.1) — `first + (last - middle)` for the overloads in namespace `std`, or

(3.2) — `{first + (last - middle), last}` for the overloads in namespace `ranges`.

4 ~~*Remarks:* This is a left rotate.~~

5 *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);

namespace ranges {
  template<ForwardIterator I, Sentinel<I> S, WeaklyIncrementable O>
    requires IndirectlyCopyable<I, O>
    constexpr rotate_copy_result<I, O>
      rotate_copy(I first, I middle, S last, O result);
  template<ForwardRange R, WeaklyIncrementable O>
    requires IndirectlyCopyable<iterator_t<R>, O>
    constexpr rotate_copy_result<safe_iterator_t<R>, O>
      rotate_copy(R&& r, iterator_t<R> middle, O result);
}
```

6 *Requires:* The ranges `[first, last)` and `[result, result + (last - first))` shall not overlap.

7 *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))`.

8 *Returns:*

(8.1) — `result + (last - first)` for the overloads in namespace `std`, or

(8.2) — `{last, result + (last - first)}` for the overloads in namespace `ranges`.

9 *Complexity:* Exactly `last - first` assignments.

24.6.12 Sample [alg.random.sample]

[...]

24.6.13 Shuffle [alg.random.shuffle]

```
template<class RandomAccessIterator, class UniformRandomBitGenerator>
  void shuffle(RandomAccessIterator first,
               RandomAccessIterator last,
               UniformRandomBitGenerator&& g);

namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Gen>
    requires Permutable<I> &&
      UniformRandomBitGenerator<remove_reference_t<Gen>> &&
      ConvertibleTo<invoke_result_t<Gen&>, iter_difference_t<I>>
    I shuffle(I first, S last, Gen&& g);
  template<RandomAccessRange R, class Gen>
    requires Permutable<iterator_t<R>> &&
      UniformRandomBitGenerator<remove_reference_t<Gen>> &&
      ConvertibleTo<invoke_result_t<Gen&>, iter_difference_t<iterator_t<R>>>
    safe_iterator_t<R>
      shuffle(R&& r, Gen&& g);
}
```

1 *Requires:* For the overloads in namespace `std`:

(1.1) — `RandomAccessIterator` shall ~~satisfy~~meet the *Cpp17ValueSwappable* requirements ([swappable.requirements]).

(1.2) — The type `remove_reference_t<UniformRandomBitGenerator>` shall ~~satisfy the requirements of~~ ~~a~~meet the uniform random bit generator ([rand.req.urng]) requirements ~~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 *Returns:* `last` for the overloads in namespace `ranges`.

4 *Complexity:* Exactly `(last - first) - 1` swaps.

5 *Remarks:* To the extent that the implementation of this function makes use of random numbers, the object referenced by `g` shall serve as the implementation's source of randomness.

24.6.14 Shift [alg.shift]

[...]

24.7 Sorting and related operations [alg.sorting]

1 ~~All the~~The operations in 24.7 defined 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 24.7.3, `comp` shall induce a strict weak ordering on the values.

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)`.

24.7.1 Sorting [alg.sort]

24.7.1.1 `sort` [sort]

```
template<class RandomAccessIterator>
  constexpr void sort(RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator>
  void sort(ExecutionPolicy&& exec,
            RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  constexpr 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);

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 R, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    constexpr safe_iterator_t<R>
      sort(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}
```

1 *Requires:* For the overloads in namespace `std`, `RandomAccessIterator` shall ~~satisfy~~meet the *Cpp17ValueSwappable* requirements ([swappable.requirements])~~. The~~ and the type of `*first` shall ~~satisfy~~meet the *Cpp17MoveConstructible* ([tab:moveconstructible]) and *Cpp17MoveAssignable* ([tab:moveassignable]) requirements.

2 *Effects:* Sorts the elements in the range `[first, last)`.

3 *Returns:* `last`, for the overloads in namespace `ranges`.

4 *Complexity:* Let N be `last - first`. $\mathcal{O}(N \log(N))$ comparisons and twice as many applications of any projection.~~, where N = `last - first`.~~

24.7.1.2 `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);

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 R, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    safe_iterator_t<R>
      stable_sort(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}
```

1 *Requires:* For the overloads in namespace `std`, `RandomAccessIterator` shall ~~satisfy~~meet the *Cpp17ValueSwappable* requirements ([swappable.requirements])~~. The~~ and the type of `*first` shall ~~satisfy~~meet the *Cpp17MoveConstructible* ([tab:moveconstructible]) and *Cpp17MoveAssignable* ([tab:moveassignable]) requirements.

2 *Effects:* Sorts the elements in the range `[first, last)`.

3 *Returns:* `last` for the overloads in namespace `ranges`.

4 *Complexity:* Let N be `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 any projection as the number of comparisons. ~~At most $N \log^2(N)$ comparisons, where $N =$ `last - first`, but only $N \log N$ comparisons if there is enough extra memory.~~

5 *Remarks:* Stable ([algorithm.stable]).

24.7.1.3 `partial_sort` [partial.sort]

```
template<class RandomAccessIterator>
  constexpr 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>
  constexpr 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);

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 R, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    constexpr safe_iterator_t<R>
      partial_sort(R&& r, iterator_t<R> middle, Comp comp = Comp{},
                   Proj proj = Proj{});
}
```

1 *Requires:* For the overloads in namespace `std`, `RandomAccessIterator` shall ~~satisfy~~meet the *Cpp17ValueSwappable* requirements ([swappable.requirements])~~. The~~ and the type of `*first` shall ~~satisfy~~meet 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 *Returns:* `last` for the overloads in namespace `ranges`.

4 *Complexity:* Approximately `(last - first) * log(middle - first)` comparisons, and exactly twice as many applications of any projection.

24.7.1.4 `partial_sort_copy` [partial.sort.copy]

```
template<class InputIterator, class RandomAccessIterator>
  constexpr 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>
  constexpr 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);

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 R1, RandomAccessRange R2, class Comp = ranges::less<>,
      class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyCopyable<iterator_t<R1>, iterator_t<R2>> &&
        Sortable<iterator_t<R2>, Comp, Proj2> &&
        IndirectStrictWeakOrder<Comp, projected<iterator_t<R1>, Proj1>,
          projected<iterator_t<R2>, Proj2>>
    constexpr safe_iterator_t<R2>
```

```
    partial_sort_copy(R1&& r, R2&& result_r, Comp comp = Comp{},
                      Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

1 *Requires:* For the overloads in namespace `std`, `RandomAccessIterator` shall ~~satisfy~~meet the *Cpp17ValueSwappable* requirements ([swappable.requirements])~~. The~~ and the type of `*result_first` shall ~~satisfy~~meet 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, and exactly twice as many applications of any projection.

24.7.1.5 `is_sorted` [is.sorted]

```
template<class ForwardIterator>
  constexpr bool is_sorted(ForwardIterator first, ForwardIterator last);
```

1 ~~*Returns:*~~ *Effects:* Equivalent to: `return is_sorted_until(first, last) == last;`~~.~~

```
template<class ExecutionPolicy, class ForwardIterator>
  bool is_sorted(ExecutionPolicy&& exec,
                 ForwardIterator first, ForwardIterator last);
```

2 ~~*Returns:*~~ *Effects:* Equivalent to:

```
return 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:*~~ *Effects:* Equivalent to: `return 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:*~~ *Effects:* Equivalent to:

```
return 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 R, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    constexpr bool is_sorted(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}
```

5 *Effects:* Equivalent to: `return 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);

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 R, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    constexpr safe_iterator_t<R>
      is_sorted_until(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}
```

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.

24.7.2 Nth element [alg.nth.element]

```
template<class RandomAccessIterator>
  constexpr 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>
  constexpr 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);

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 R, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    constexpr safe_iterator_t<R>
      nth_element(R&& r, iterator_t<R> nth, Comp comp = Comp{}, Proj proj = Proj{});
}
```

1 Let `comp` be `std::less<>{}` for the overloads with no parameter named `comp`.

2 *Requires:* For the overloads in namespace `std`, `RandomAccessIterator` shall ~~satisfy~~meet the *Cpp17ValueSwappable* requirements ([swappable.requirements])~~. The~~and the type of `*first` shall ~~satisfy~~meet the *Cpp17MoveConstructible* ([tab:moveconstructible]) and *Cpp17MoveAssignable* ([tab:moveassignable]) requirements.

3 *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`.~~

(3.1) — `bool(comp(*j, *i))` is `false` for the overloads in namespace `std`, or

(3.2) — `bool(invoke(comp, invoke(proj, *j), invoke(proj, *i)))` is `false` for the overloads in namespace `ranges`.

4 *Returns:* `last` for the overloads in namespace `ranges`.

5 *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`.

24.7.3 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.

24.7.3.1 `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);

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 R, class T, class Proj = identity,
      IndirectStrictWeakOrder<const T*, projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    constexpr safe_iterator_t<R>
      lower_bound(R&& r, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
}
```

1 Let `comp` be `std::less<>{}` for the overload with no parameter named `comp`.

2 *Requires:* The elements `e` of `[first, last)` shall be partitioned with respect to the expression ~~`e < value` or~~:

(2.1) — `bool(comp(e, value))`, for the overloads in namespace `std`, or

(2.2) — `bool(invoke(comp, invoke(proj, e), value))`, for the overloads in namespace `ranges`.

3 *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~~

(3.1) — `bool(comp(*j, value))` ~~`!= false`~~ for the overloads in namespace `std`, or

(3.2) — `bool(invoke(comp, invoke(proj, *j), value))`, for the overloads in namespace `ranges`.

4 *Complexity:* At most $\log_2($`last - first`$) + \mathcal{O}(1)$ comparisons and applications of any projection.

24.7.3.2 `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);
```

```
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 R, class T, class Proj = identity,
      IndirectStrictWeakOrder<const T*, projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    constexpr safe_iterator_t<R>
      upper_bound(R&& r, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
}
```

1 Let `comp` be `std::less<>{}` for the overload with no parameter named `comp`.

2 *Requires:* The elements `e` of `[first, last)` shall be partitioned with respect to the expression ~~`!(value < e)` or~~

(2.1) — `!bool(comp(value, e))` for the overloads in namespace `std`, or

(2.2) — `!invoke(comp, value, invoke(proj, e))` for the overloads in namespace `ranges`.

3 *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~~

(3.1) — `!bool(comp(value, *j))` ~~`== false`~~ for the overloads in namespace `std`, or

(3.2) — `!invoke(comp, value, invoke(proj, *j))` for the overloads in namespace `ranges`.

4 *Complexity:* At most $\log_2(\texttt{last - first}) + \mathcal{O}(1)$ comparisons and applications of any projection.

24.7.3.3 `equal_range` [equal.range]

[Editor's note: This wording incorporates the PR for stl2#526).]

```
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<>>
    constexpr subrange<I>
      equal_range(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
  template<ForwardRange R, class T, class Proj = identity,
      IndirectStrictWeakOrder<const T*, projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    constexpr safe_subrange_t<R>
      equal_range(R&& r, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
}
```

1 Let `comp` be `std::less<>{}` for the overload with no parameter named `comp`.

2 *Requires:* The elements `e` of `[first, last)` shall be partitioned with respect to the expressions ~~`e < value` and `!(value < e)` or~~

(2.1) — `bool(comp(e, value))` and `!bool(comp(value, e))` for the overloads in namespace `std`, or

(2.2) — `invoke(comp, invoke(proj, e), value)` and `!invoke(comp, value, invoke(proj, e))`. for the overloads in namespace `ranges`.

Also, for all elements `e` of `[first, last)`, ~~`e < value` shall imply `!(value < e)` or~~

(2.3) — `bool(comp(e, value))` shall imply `!bool(comp(value, e))` for the overloads in namespace `std`, or

(2.4) — `bool(invoke(comp, invoke(proj, e), value))` shall imply `!invoke(comp, value, invoke(proj, e))` for the overloads in namespace `ranges`.

3 *Returns:*

(3.1) — For the overloads in namespace `std`:

```
~~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)})
```

(3.2) — For the overloads in namespace `ranges`:

```
{ranges::lower_bound(first, last, value, comp, proj),
 ranges::upper_bound(first, last, value, comp, proj)}
```

4 *Complexity:* At most $2 * \log_2($`last - first`$) + \mathcal{O}(1)$ comparisons and applications of any projection.

24.7.3.4 `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 Let `comp` be `std::less<>{}` for the overload with no parameter named `comp`.

2 *Requires:* The elements `e` of `[first, last)` shall be partitioned with respect to the expressions ~~`e < value` and `!(value < e)` or~~ `bool(comp(e, value))` and `!bool(comp(value, e))`. Also, for all elements `e` of `[first, last)`, ~~`e < value` shall imply `!(value < e)` or~~ `bool(comp(e, value))` shall imply `!bool(comp(value, e))`.

3 *Returns:* `true` if and only if there is an iterator `i` in the range `[first, last)` that satisfies the corresponding conditions: ~~`!(*i < value) && !(value < *i)` or~~ `!bool(comp(*i, value))` ~~`== false`~~ `&&` `!bool(comp(value, *i))` ~~`== false`~~.

4 *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 R, class T, class Proj = identity,
      IndirectStrictWeakOrder<const T*, projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    constexpr bool binary_search(R&& r, const T& value, Comp comp = Comp{},
                                 Proj proj = Proj{});
}
```

5 *Requires:* The elements `e` of `[first, last)` are partitioned with respect to the expressions `bool(invoke(comp, invoke(proj, e), value))` and `!invoke(comp, value, invoke(proj, e))`. Also, for all elements `e` of `[first, last)`, `bool(invoke(comp, invoke(proj, e), value))` shall imply `!invoke(comp, value, invoke(proj, e))`.

6 *Returns:* `true` if and only if there is an iterator `i` in the range `[first, last)` that satisfies the corresponding conditions: `!invoke(comp, invoke(proj, *i), value) && !invoke(comp, value, invoke(proj, *i))`.

7 *Complexity:* At most $\log_2($`last - first`$) + \mathcal{O}(1)$ applications of the comparison function and projection.

24.7.4 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);

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 R, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr bool is_partitioned(R&& r, Pred pred, Proj proj = Proj{});
}
```

1 Let E(`i`) be

(1.1) — `bool(pred(*i))` for the overloads in namespace `std`, or

(1.2) — `bool(invoke(pred, invoke(proj, *i)))` for the overloads in namespace `ranges`.

2 *Requires:* For the overloads in namespace `std`, E(`first`) shall be a well-formed expression.

3 *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.

4 *Returns:* `true` if and only if `[first, last)` is ~~empty or if the elements e of [first, last) are~~ partitioned with respect to the expression ~~`pred(e)`~~ E(`i`).

5 *Complexity:* Linear. At most `last - first` applications of `pred` and any projection.

```
template<class ForwardIterator, class Predicate>
  constexpr 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);

namespace ranges {
  template<ForwardIteratorPermutable 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 R, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    requires Permutable<iterator_t<R>>
    constexpr safe_iterator_t<R>
      partition(R&& r, Pred pred, Proj proj = Proj{});
}
```

6 Let E(`j`) be

(6.1) — `bool(pred(*j))` for the overloads in namespace `std`, or

(6.2) — `bool(invoke(pred, invoke(proj, *j)))` for the overloads in namespace `ranges`.

7 *Requires:* For the overloads in namespace `std`, E(`first`) shall be a valid expression and `ForwardIterator` shall ~~satisfy~~ meet the *Cpp17ValueSwappable* requirements ([swappable.requirements]).

8 *Effects:* Places all the elements in the range `[first, last)` that satisfy ~~`pred`~~ E before all the elements that do not satisfy it.

9 *Returns:* An iterator `i` such that E(`j`) is `true` for every iterator `j` in ~~the range~~ `[first, i)` ~~`pred(*j) != false`,~~ and `false` for every iterator ~~`k`~~ in ~~the range~~ `[i, last)`~~, `pred(*k) == false` is false~~.

10 *Complexity:* Let N = `last - first`:

(10.1) — For the overloads with no `ExecutionPolicy`, exactly N applications of the predicate and any projection. At most $N/2$ swaps if `ForwardIterator` meets the *Cpp17BidirectionalIterator* require-

ments for the overloads in namespace `std` or models `BidirectionalIterator` for the overloads in namespace `ranges`, and at most N swaps otherwise.

(10.2) — For the overload with an `ExecutionPolicy`, $\mathcal{O}(N \log N)$ swaps and $\mathcal{O}(N)$ applications of the predicate.

```
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);

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

11 Let E(`j`) be

(11.1) — `bool(pred(*j))` for the overloads in namespace `std`, or

(11.2) — `bool(invoke(pred, invoke(proj, *j)))` for the overloads in namespace `ranges`.

12 *Requires:* For the overloads in namespace `std`, `BidirectionalIterator` shall ~~satisfy~~meet the *Cpp17ValueSwappable* requirements ([swappable.requirements])~~. The~~ and the type of `*first` shall ~~satisfy~~ meet the *Cpp17MoveConstructible* ([tab:moveconstructible]) and *Cpp17MoveAssignable* ([tab:moveassignable]) requirements.

13 *Effects:* Places all the elements in the range `[first, last)` that satisfy ~~`pred`~~ E before all the elements that do not satisfy it.

14 *Returns:* An iterator `i` such that for every iterator `j` in the range `[first, i)`, ~~`pred(*j) != false`~~ E`(j)` is `true`, and for every iterator `k` in the range `[i, last)`, ~~`pred(*k) == false`~~ E`(k)` is `false`. The relative order of the elements in both groups is preserved.

15 *Complexity:* Let N = `last - first`:

(15.1) — For the overloads with no `ExecutionPolicy`, at most $N \log N$ swaps, but only $\mathcal{O}(N)$ swaps if there is enough extra memory. Exactly N applications of the predicate and any projection.

(15.2) — For the overload with an `ExecutionPolicy`, $\mathcal{O}(N \log N)$ swaps and $\mathcal{O}(N)$ applications of the predicate.

```
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);

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>
    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 R, WeaklyIncrementable O1, WeaklyIncrementable O2,
      class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    requires IndirectlyCopyable<iterator_t<R>, O1> &&
      IndirectlyCopyable<iterator_t<R>, O2>
    constexpr partition_copy_result<safe_iterator_t<R>, O1, O2>
      partition_copy(R&& r, O1 out_true, O2 out_false, Pred pred, Proj proj = Proj{});
}
```

16 Let *E* be

(16.1) — `bool(pred(*i))` for the overloads in namespace `std`, or

(16.2) — `bool(invoke(pred, invoke(proj, *i)))` for the overloads in namespace `ranges`.

17 *Requires:* The input range shall not overlap with either of the output ranges. For the overloads in namespace `std`, `pred(*first)` shall be a valid expression and the expression `*first` shall be writable (22.3.1) to `out_true` and `out_false`. [*Note:* For the overload with an `ExecutionPolicy`, there may be a performance cost if `first`'s value type is not *Cpp17CopyConstructible*. — *end note*]

(17.1) — For the overload with no `ExecutionPolicy`, `InputIterator`'s value type shall be *Cpp17CopyAssignable* ([tab:copyassignable]), and shall be writable (22.3.1) to the `out_true` and `out_false` `OutputIterator`s, and shall be convertible to `Predicate`'s argument type.

(17.2) — For the overload with an `ExecutionPolicy`, `ForwardIterator`'s value type shall be *Cpp17CopyAssignable*, and shall be writable to the `out_true` and `out_false` `ForwardIterator`s, and shall be convertible to `Predicate`'s argument type. [*Note:* There may be a performance cost if `ForwardIterator`'s value type is not *Cpp17CopyConstructible*. — *end note*]

(17.3) — For both overloads, the input range shall not overlap with either of the output ranges.

18 *Effects:* For each iterator `i` in `[first, last)`, copies `*i` to the output range beginning with `out_true` if ~~`pred(*i)`~~ *E*`(i)` is `true`, or to the output range beginning with `out_false` otherwise.

19 *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`.~~ Let `o1` be the end of the output range beginning at `out_true`, and `o2` the end of the output range beginning at `out_false` Returns:

(19.1) — `{o1, o2}` for the overloads in namespace `std`, or

(19.2) — `{last, o1, o2}` for the overloads in namespace `ranges`.

20 *Complexity:* Exactly `last - first` applications of `pred` and any projection.

```
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 R, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr safe_iterator_t<R>
      partition_point(R&& r, Pred pred, Proj proj = Proj{});
}
```

21 Let *E*`(i)` be

(21.1) — `bool(pred(*i))` for the overload in namespace `std`, or

(21.2) — `bool(invoke(pred, invoke(proj, *i)))` for the overloads in namespace `ranges`.

22 *Requires:* The sequence `[first, last)` shall be partitioned with respect to the expression *E*`(i)`. For the overload in namespace `std`, *E*`(first)` shall be a valid expression.

23 ~~*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)`.~~

24 *Returns:* An iterator `mid` such that ~~`all_of(first, mid, pred)` and `none_of(mid, last, pred)` are both `true`~~ *E*`(i)` is `true` for all iterators `i` in `[first, mid)`, and `false` for all iterators in `[mid, last)`.

25 *Complexity:* $\mathcal{O}(\log($`last - first`$))$ applications of `pred` and any projection.

26 *Returns:* An iterator `mid` such that `all_of(first, mid, pred, proj)` and `none_of(mid, last, pred, proj)` are both true.

24.7.5 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);

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 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 R1, InputRange R2, WeaklyIncrementable O, class Comp = ranges::less<>,
      class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<R1>, iterator_t<R2>, O, Comp, Proj1, Proj2>
    constexpr merge_result<safe_iterator_t<R1>, safe_iterator_t<R2>, O>
      merge(R1&& r1, R2&& r2, O result,
            Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

1 Let `comp` be `std::less<>{}` for the overloads with no parameter `comp`, and let N be `(last1 - first1) + (last2 - first2)`.

2 *Requires:* The ranges `[first1, last1)` and `[first2, last2)` shall be sorted with respect to ~~operator< or~~ `comp`, and the corresponding projection `proj1` or `proj2` for the overloads in namespace `ranges`. The resulting range shall not overlap with either of the original ranges.

3 *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 +` N ~~`(last1 - first1) + (last2 - first2)`, such that the resulting range satisfies `is_sorted(result, result_last)` or `is_sorted(result, result_last, comp)`, respectively~~. 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

(3.1) — `bool(comp(e2, e1))` for the overloads in namespace `std`, or

(3.2) — `bool(invoke(comp, invoke(proj2, e2), invoke(proj1, e1)))` for the overloads in namespace `ranges`

is `true`.

4 *Returns:* ~~`result + (last1 - first1) + (last2 - first2).`~~

(4.1) — `result_last` for the overloads in namespace `std`, or

(4.2) — `{last1, last2, result_last}` for the overloads in namespace `ranges`.

5 *Complexity:* ~~Let $N = $ `(last1 - first1) + (last2 - first2)`:~~

(5.1) — For the overloads with no `ExecutionPolicy`, at most $N - 1$ comparisons and applications of each projection.

(5.2) — For the overloads with an `ExecutionPolicy`, $\mathcal{O}(N)$ comparisons.

6 *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);

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 R, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    safe_iterator_t<R>
      inplace_merge(R&& r, iterator_t<R> middle, Comp comp = Comp{},
                    Proj proj = Proj{});
}
```

7 Let `comp` be `std::less<>{}` for the overloads with no parameter `comp`.

8 *Requires:* The ranges `[first, middle)` and `[middle, last)` shall be sorted with respect to ~~`operator<` or~~ `comp`, and `proj` for the overloads in namespace `ranges`. For the overloads in namespace `std`, `BidirectionalIterator` shall ~~satisfy~~meet the *Cpp17ValueSwappable* requirements ([swappable.requirements])~~.~~ ~~The~~ and the type of `*first` shall ~~satisfy~~meet the *Cpp17MoveConstructible* ([tab:moveconstructible]) and *Cpp17MoveAssignable* ([tab:moveassignable]) requirements.

9 *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,~~

(9.1) — `bool(comp(*i, *(i - 1)))` for the overloads in namespace `std`, or

(9.2) — `bool(invoke(comp, invoke(proj, *i), invoke(proj, *(i - 1))))` for the overloads in namespace `ranges`

will be `false`.

10 *Returns:* `last` for the overloads in namespace `ranges`.

11 *Complexity:* Let $N =$ `last - first`:

- (11.1) — For the overloads with no `ExecutionPolicy`, if enough additional memory is available, exactly $N - 1$ comparisons.
- (11.2) — ~~For the overloads with no `ExecutionPolicy` if no additional memory is available, $\mathcal{O}(N \log N)$ comparisons.~~
- (11.3) — Otherwise ~~For the overloads with an `ExecutionPolicy`~~, $\mathcal{O}(N \log N)$ comparisons.

In any case, exactly twice as many applications of any projection as the number of comparisons.

12 *Remarks:* Stable ([algorithm.stable]).

24.7.6 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.

24.7.6.1 `includes` [includes]

[Editor's note: This wording includes the proposed resolution for LWG 3115.]

```
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);

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 R1, InputRange R2, class Proj1 = identity,
      class Proj2 = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R1>, Proj1>,
        projected<iterator_t<R2>, Proj2>> Comp = ranges::less<>>
    constexpr bool includes(R1&& r1, R2&& r2, Comp comp = Comp{},
                            Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

1 Let `comp` be `std::less<>{}` for the overloads with no parameter `comp`.

2 *Requires:* The ranges `[first1, last1)` and `[first2, last2)` shall be sorted with respect to `comp`, and `proj` for the overloads in namespace `ranges`.

3 *Returns:* `true` if and only if `[first2, last2)` is ~~empty or if every element in the range `[first2, last2)` is contained in the range `[first1, last1)`. Returns `false` otherwise.~~ a subsequence of `[first1, last1)`. [*Note:* A sequence S is a subsequence of another sequence T if S can be obtained from T

by removing some, all, or none of T's elements and keeping the remaining elements in the same order. — *end note*]

4 *Complexity:* At most `2 * (last1 - first1)` comparisons and applications of any projections.

24.7.6.2 `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);

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 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 R1, InputRange R2, WeaklyIncrementable O,
      class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<R1>, iterator_t<R2>, O, Comp, Proj1, Proj2>
    constexpr set_union_result<safe_iterator_t<R1>, safe_iterator_t<R2>, O>
      set_union(R1&& r1, R2&& r2, O result, Comp comp = Comp{},
                Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

1 *Requires:* The ranges `[first1, last1)` and `[first2, last2)` shall be sorted with respect to `comp`, and the corresponding projection `proj1` or `proj2` for the overloads in namespace `ranges`. 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:* ~~T~~Let `result_last` be the end of the constructed range. Returns

- (3.1) — `result_last` for the overloads in namespace `std`, or
- (3.2) — `{last1, last2, result_last}` for the overloads in namespace `ranges`.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons and applications of any projections.

5 *Remarks:* Stable ([algorithm.stable]). 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 the final $\max(n - m, 0)$ elements from the second range shall be copied to the output range, in order.

24.7.6.3 `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);

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 set_intersection_result<I1, I2, O>
      set_intersection(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                       Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
  template<InputRange R1, InputRange R2, WeaklyIncrementable O,
      class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<R1>, iterator_t<R2>, O, Comp, Proj1, Proj2>
    constexpr set_intersection_result<safe_iterator_t<R1>, safe_iterator_t<R2>, O>
      set_intersection(R1&& r1, R2&& r2, O result,
                       Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

1 *Requires:* The ranges `[first1, last1)` and `[first2, last2)` shall be sorted with respect to `comp`, and the corresponding projection `proj1` or `proj2` for the overloads in namespace `ranges`. 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:* ~~T~~Let `result_last` be the end of the constructed range. Returns

- (3.1) — `result_last` for the overloads in namespace `std`, or
- (3.2) — `{last1, last2, result_last}` for the overloads in namespace `ranges`.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons and applications of any projections.

5 *Remarks:* Stable ([algorithm.stable]). 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.

24.7.6.4 `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);

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 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 R1, InputRange R2, WeaklyIncrementable O,
      class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<R1>, iterator_t<R2>, O, Comp, Proj1, Proj2>
    constexpr set_difference_result<safe_iterator_t<R1>, O>
      set_difference(R1&& r1, R2&& r2, O result,
                     Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

1 *Requires:* The ranges `[first1, last1)` and `[first2, last2)` shall be sorted with respect to `comp`, and the corresponding projection `proj1` or `proj2` for the overloads in namespace `ranges`. 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:* ~~T~~Let `result_last` be the end of the constructed range. Returns

(3.1) — `result_last` for the overloads in namespace `std`, or

(3.2) — `{last1, result_last}` for the overloads in namespace `ranges`.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons and applications of any projections.

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.

24.7.6.5 `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);

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 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 R1, InputRange R2, WeaklyIncrementable O,
      class Comp = ranges::less<>, class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<iterator_t<R1>, iterator_t<R2>, O, Comp, Proj1, Proj2>
    constexpr set_symmetric_difference_result<safe_iterator_t<R1>, safe_iterator_t<R2>, O>
      set_symmetric_difference(R1&& r1, R2&& r2, O result, Comp comp = Comp{},
                               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

1 *Requires:* The ranges `[first1, last1)` and `[first2, last2)` shall be sorted with respect to `comp`, and the corresponding projection `proj1` or `proj2` for the overloads in namespace `ranges`. 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:* ~~T~~Let `result_last` be the end of the constructed range. Returns

(3.1) — `result_last` for the overloads in namespace `std`, or

(3.2) — `{last1, last2, result_last}` for the overloads in namespace `ranges`.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons and applications of any projections.

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 $m < n$.

24.7.7 Heap operations [alg.heap.operations]

[...]

24.7.7.1 push_heap [push.heap]

```
template<class RandomAccessIterator>
  constexpr void push_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  constexpr 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 R, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    constexpr safe_iterator_t<R>
      push_heap(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}
```

1 *Requires:* The range `[first, last - 1)` shall be a valid heap. ~~T~~For the overloads in namespace `std`, the type of `*first` shall ~~satisfy~~meet 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 *Returns:* `last` for the overloads in namespace `ranges`.

4 *Complexity:* At most $\log(\texttt{last - first})$ comparisons and twice as many applications of any projection.

24.7.7.2 pop_heap [pop.heap]

```
template<class RandomAccessIterator>
  constexpr void pop_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  constexpr 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 R, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    constexpr safe_iterator_t<R>
      pop_heap(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}
```

1 *Requires:* The range `[first, last)` shall be a valid non-empty heap. For the overloads in namespace `std`, `RandomAccessIterator` shall ~~satisfy~~meet the *Cpp17ValueSwappable* requirements ([swappable.requirements])~~.~~ ~~The~~ and the type of `*first` shall ~~satisfy~~meet 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 *Returns:* `last` for the overloads in namespace `ranges`.

4 *Complexity:* At most $2\log(\texttt{last - first})$ comparisons and twice as many applications of any projection.

24.7.7.3 make_heap [make.heap]

```
template<class RandomAccessIterator>
  constexpr void make_heap(RandomAccessIterator first, RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class Compare>
  constexpr 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 R, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    constexpr safe_iterator_t<R>
      make_heap(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}
```

1 *Requires:* For the overloads in namespace `std`, t~~T~~he 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 *Returns:* `last` for the overloads in namespace `ranges`.

4 *Complexity:* At most 3(`last - first`) comparisons and twice as many applications of any projection.

24.7.7.4 `sort_heap` [sort.heap]

```
template<class RandomAccessIterator>
  constexpr void sort_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  constexpr 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 R, class Comp = ranges::less<>, class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    constexpr safe_iterator_t<R>
      sort_heap(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}
```

1 *Requires:* The range `[first, last)` shall be a valid heap. For the overloads in namespace `std`, `RandomAccessIterator` shall ~~satisfy~~meet the *Cpp17ValueSwappable* requirements ([swappable.requirements])~~.~~ ~~The~~ and the type of `*first` shall ~~satisfy~~meet the *Cpp17MoveConstructible* ([tab:moveconstructible]) and *Cpp17MoveAssignable* ([tab:moveassignable]) requirements.

2 *Effects:* Sorts elements in the heap `[first, last)`.

3 *Returns:* `last` for the overloads in namespace `ranges`.

4 *Complexity:* At most $2N \log N$ comparisons, where $N =$ `last - first`, and exactly twice as many applications of any projection.

24.7.7.5 `is_heap` [is.heap]

```
template<class RandomAccessIterator>
  constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last);
```

1 ~~*Returns:*~~ *Effects:* Equivalent to: `return is_heap_until(first, last) == last;`~~.~~

```
template<class ExecutionPolicy, class RandomAccessIterator>
  bool is_heap(ExecutionPolicy&& exec,
               RandomAccessIterator first, RandomAccessIterator last);
```

2 ~~*Returns:*~~ *Effects:* Equivalent to:

```
return 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:*~~ *Effects:* Equivalent to: `return 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:*~~ *Effects:* Equivalent to:

```
return 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 R, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    constexpr bool is_heap(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
```

5 *Effects:* Equivalent to: `return ranges::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);

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 R, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    constexpr safe_iterator_t<R>
      is_heap_until(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}
```

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.

24.7.8 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);
```

```
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{});
}
```

1 *Requires:* For the first form, type `T` shall be *Cpp17LessThanComparable* ([tab:lessthancomparable]).

2 *Returns:* The smaller value.

3 *Remarks:* Returns the first argument when the arguments are equivalent.

4 *Complexity:* Exactly one comparison and two applications of any projection.

```
template<class T>
  constexpr T min(initializer_list<T> tr);
template<class T, class Compare>
  constexpr T min(initializer_list<T> tr, Compare comp);
```

```
namespace ranges {
  template<Copyable T, class Proj = identity,
      IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr T min(initializer_list<T> r, Comp comp = Comp{}, Proj proj = Proj{});
  template<InputRange R, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    requires Copyable<iter_value_t<iterator_t<R>>>
    constexpr iter_value_t<iterator_t<R>>
      min(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}
```

5 *Requires:* `ranges::distance(r) > 0`. For the overloads in namespace `std`, `T` shall be *Cpp17CopyConstructible* ~~and `t.size() > 0`~~. For the first form, type `T` shall be *Cpp17LessThanComparable*.

6 *Returns:* The smallest value in the ~~initializer list~~ input range.

7 *Remarks:* Returns a copy of the leftmost ~~argument~~ element when several ~~argument~~ elements are equivalent to the smallest.

8 *Complexity:* Exactly ~~`t.size()`~~`ranges::distance(r) - 1` comparisons and twice as many applications of any projection.

```
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);
```

```
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{});
}
```

9 *Requires:* For the first form, type `T` shall be *Cpp17LessThanComparable* ([tab:lessthancomparable]).

10 *Returns:* The larger value.

11 *Remarks:* Returns the first argument when the arguments are equivalent.

12 *Complexity:* Exactly one comparison and two applications of any projection.

```
template<class T>
  constexpr T max(initializer_list<T> tr);
template<class T, class Compare>
  constexpr T max(initializer_list<T> tr, Compare comp);
```

```
namespace ranges {
  template<Copyable T, class Proj = identity,
      IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr T max(initializer_list<T> r, Comp comp = Comp{}, Proj proj = Proj{});
  template<InputRange R, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    requires Copyable<iter_value_t<iterator_t<R>>>
```

```
    constexpr iter_value_t<iterator_t<R>>
      max(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}
```

13 *Requires:* ranges::distance(r) > 0. For the overloads in namespace std, T shall be *Cpp17CopyConstructible* ~~and t.size() > 0~~. For the first form, type T shall be *Cpp17LessThanComparable*.

14 *Returns:* The largest value in the ~~initializer list~~ input range.

15 *Remarks:* Returns a copy of the leftmost ~~argument~~ element when several ~~argument~~ elements are equivalent to the largest.

16 *Complexity:* Exactly ~~t.size()~~ ranges::distance(r) - 1 comparisons and twice as many applications of any projection.

```
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);

namespace ranges {
  template<class T, class Proj = identity,
      IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr minmax_result<const T&>
      minmax(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
}
```

17 *Requires:* For the first form, type T shall be *Cpp17LessThanComparable* ([tab:lessthancomparable]).

18 *Returns:* Let X be the return type of the overload in question. Returns ~~pair<const T&, const T&>(~~X{b, a}~~)~~ if b is smaller than a, and ~~pair<const T&, const T&>(~~X{a, b}~~)~~ otherwise.

19 ~~*Remarks:* Returns pair<const T&, const T&>(a, b) when the arguments are equivalent.~~

20 *Complexity:* Exactly one comparison and two applications of any projection.

```
template<class T>
  constexpr pair<T, T> minmax(initializer_list<T> tr);
template<class T, class Compare>
  constexpr pair<T, T> minmax(initializer_list<T> tr, Compare comp);

namespace ranges {
  template<Copyable T, class Proj = identity,
      IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr minmax_result<T>
      minmax(initializer_list<T> r, Comp comp = Comp{}, Proj proj = Proj{});
  template<InputRange R, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    requires Copyable<iter_value_t<iterator_t<R>>>
    constexpr minmax_result<iter_value_t<iterator_t<R>>>
      minmax(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}
```

21 *Requires:* ranges::distance(r) > 0. For the overloads in namespace std, T shall be *Cpp17CopyConstructible* ~~and t.size() > 0~~. For the first form, type T shall be *Cpp17LessThanComparable*.

22 *Returns:* Let X be the return type of the overload in question. Returns ~~pair<T, T>(~~X{x, y}~~)~~, where x ~~has~~ is a copy of the leftmost element with the smallest value and y ~~has~~ a copy of the rightmost element with the largest value in the ~~initializer list~~ input range.

23 ~~*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.~~

24 *Complexity:* At most (3/2)~~t.size()~~ranges::distance(r) applications of the corresponding predicate and twice as many applications of any 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);

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 R, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    constexpr safe_iterator_t<R>
      min_element(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}
```

25 Let `comp` be `std::less<>{}` for the overloads with no parameter `comp`, and let E`(x, y)` be

(25.1) — `bool(comp(*x, *y))` for the overloads in namespace `std`, or

(25.2) — `bool(invoke(comp, invoke(proj, *x), invoke(proj, *y)))` for the overloads in namespace `ranges`.

26 *Returns:* The first iterator `i` in the range `[first, last)` such that for every iterator `j` in the range `[first, last)` E`(j, i)` is `false`. ~~the following corresponding conditions hold: `!(*j < *i)` or `comp(*j, *i) == false`.~~ Returns `last` if `first == last`.

27 *Complexity:* Exactly $\max(\texttt{last - first} - 1, 0)$ ~~applications of the corresponding~~ comparisons and twice as many applications of any 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);

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 R, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    constexpr safe_iterator_t<R>
      max_element(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}
```

28 Let `comp` be `std::less<>{}` for the overloads with no parameter `comp`, and let E`(x, y)` be

(28.1) — `bool(comp(*x, *y))` for the overloads in namespace `std`, or

(28.2) — `bool(invoke(comp, invoke(proj, *x), invoke(proj, *y)))` for the overloads in namespace `ranges`.

29 *Returns:* The first iterator `i` in the range `[first, last)` such that for every iterator `j` in the range `[first, last)` *E*`(i, j)` is false. ~~the following corresponding conditions hold: `!(*i < *j)` or `comp(*i, *j) == false`.~~ Returns `last` if `first == last`.

30 *Complexity:* Exactly max(`last - first - 1`, 0) ~~applications of the corresponding~~ comparisons and twice as many applications of any 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);

namespace ranges {
  template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less<>>
    constexpr minmax_result<I>
      minmax_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
  template<ForwardRange R, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less<>>
    constexpr minmax_result<safe_iterator_t<R>>
      minmax_element(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}
```

31 *Returns:* Let `X` be the return type of the overload in question. Returns ~~`make_pair(`~~`X{first, first}`~~`)`~~ if `[first, last)` is empty, otherwise ~~`make_pair(`~~`X{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[8] in `[first, last)` such that no iterator in the range refers to a larger element.

32 *Complexity:* Let N be `last - first`. At most $\max(\lfloor\frac{3}{2}(N-1)\rfloor, 0)$ comparisons and twice as many applications of any projection. ~~applications of the corresponding predicate, where N is `last - first`.~~

24.7.9 Bounded value [alg.clamp]

[...]

24.7.10 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);
```

8) This behavior intentionally differs from `max_element()`.

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Compare>
  bool
    lexicographical_compare(ExecutionPolicy&& exec,
                            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 R1, InputRange R2, class Proj1 = identity,
      class Proj2 = identity,
      IndirectStrictWeakOrder<projected<iterator_t<R1>, Proj1>,
        projected<iterator_t<R2>, Proj2>> Comp = ranges::less<>>
    constexpr bool
      lexicographical_compare(R1&& r1, R2&& r2, Comp comp = Comp{},
                              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
}
```

1 *Returns:* `true` if and only 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 and any projections.

3 *Remarks:* If two sequences have the same number of elements and their corresponding elements (if any) are equivalent, then neither sequence is lexicographically less than the other. If one sequence is a prefix of the other, then the shorter sequence is lexicographically less than the longer sequence. Otherwise, the lexicographical comparison of the sequences yields the same result as the comparison of the first corresponding pair of elements that are not equivalent.

4 [*Example:* Let `comp` be `std::less<>{}` for the overloads with no parameter `comp`, and let `proj1` and `proj2` be `std::identity{}` for the overloads with no parameters `proj1` and `proj2`. 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;
  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;
```

— *end example*]

5 [*Note:* An empty sequence is lexicographically less than any non-empty sequence, but not less than any empty sequence. — *end note*]

24.7.11 Three-way comparison algorithms [alg.3way]

[...]

24.7.12 Permutation generators [alg.permutation.generators]

```
template<class BidirectionalIterator>
  constexpr bool next_permutation(BidirectionalIterator first,
                                  BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
  constexpr 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 R, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    constexpr bool
      next_permutation(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}
```

1 *Requires:* For the overloads in namespace `std`, `BidirectionalIterator` shall ~~satisfy~~meet 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` (and `proj`, if present).

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.

```
template<class BidirectionalIterator>
  constexpr bool prev_permutation(BidirectionalIterator first,
                                  BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
  constexpr 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 R, class Comp = ranges::less<>,
      class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    constexpr bool
      prev_permutation(R&& r, Comp comp = Comp{}, Proj proj = Proj{});
}
```

5 *Requires:* For the overloads in namespace `std`, `BidirectionalIterator` shall ~~satisfy~~meet the *Cpp17ValueSwappable* requirements ([swappable.requirements]).

6 *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` (and `proj`, if present).

7 *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`.

8 *Complexity:* At most `(last - first) / 2` swaps.

[...]

25 Numerics library [numerics]

[...]

25.8 Numeric arrays [numarray]

[...]

25.8.10 `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 (22.3.5.6) ~~and of a contiguous iterator (22.3.1)~~ and models `ContiguousIterator`, 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 (22.3.5.6) ~~and of a contiguous iterator (22.3.1)~~ and models `ContiguousIterator`, whose `value_type` is the template parameter `T` and whose `reference` type is `const T&`.

[...]

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

`<algorithm>`, 121

constant iterator, 30
constexpr iterators, 31
container
    contiguous, 23
contiguous container, 23
contiguous iterators, 30

iterator
    constexpr, 31

`<memory>`, 9
multi-pass guarantee, 40
mutable iterator, 30

projection, 5

requirements
    iterator, 29

swappable, 8
swappable with, 8

unspecified, 185

writable, 29

Index of library names

ref-view
    ref-view, 86

adjacent_find, 161
advance, 46
    subrange, 83
all_of, 156
any_of, 156
at
    view_interface, 80

back
    view_interface, 79
back_insert_iterator, 49
base
    ref-view, 86
    common_view, 119
    counted_iterator, 61
    filter_view, 87
    filter_view::iterator, 88
    filter_view::sentinel, 90
    reverse_view, 121
    take_view, 103
    take_view::sentinel, 104
    transform_view, 91
    transform_view::iterator, 94
    transform_view::sentinel, 96
basic_string_view, 22
    const_iterator, 22
    const_pointer, 22
    const_reference, 22
    const_reverse_iterator, 22
    difference_type, 22
    iterator, 22
    pointer, 22
    reference, 22
    reverse_iterator, 22
    size_type, 22
    traits_type, 22
    value_type, 22
begin
    ref-view, 86
    common_view, 120
    filter_view, 87
    iota_view, 99
    join_view, 106
    reverse_view, 121
    single_view, 112
    split_view, 113, 114
    split_view::outer_iterator::value_type, 116
    subrange, 83
    take_view, 103
    transform_view, 91
bidirectional_iterator_tag, 45
BidirectionalIterator, 40
BidirectionalRange, 77
binary_search, 190

common_iterator, 55
    constructor, 57
    iter_move, 58
    iter_swap, 59
    operator!=, 58
    operator*, 57
    operator++, 58
    operator-, 58
    operator->, 57
    operator=, 57
    operator==, 58
common_view
    common_view, 119
CommonRange, 77
<concepts>, 6
const_iterator
    basic_string_view, 22
const_pointer
    basic_string_view, 22
const_reference
    basic_string_view, 22
const_reverse_iterator
    basic_string_view, 22
contiguous_iterator_tag, 45
ContiguousIterator, 41
ContiguousRange, 78
copy, 167
copy_backward, 168
copy_if, 168
copy_n, 167
count, 161
    counted_iterator, 61
count_if, 161
counted_iterator, 59
    base, 61
    constructor, 61
    count, 61
    counted_iterator, 61
    iter_move, 64
    iter_swap, 64
    operator!=, 63
    operator*, 61
    operator+, 62
    operator++, 61, 62
    operator+=, 62
    operator-, 63
    operator-=, 63
    operator--, 62
    operator<, 64

operator<=, 64
operator=, 61
operator==, 63
operator>, 64
operator>=, 64
operator[], 63

data
ref-view, 86
single_view, 112
view_interface, 79
default_sentinel, 59
destroy, 18
destroy_at, 18
destroy_n, 19
difference_type
basic_string_view, 22
distance, 47

empty
ref-view, 86
subrange, 83
view_interface, 79
end
ref-view, 86
common_view, 120
filter_view, 87
iota_view, 99
join_view, 106
reverse_view, 121
single_view, 112
split_view, 114
split_view::outer_iterator::value_type, 116
subrange, 83
take_view, 104
transform_view, 92
equal, 163
equal_range, 189
equal_to, 20
equal_to<>, 20

fill, 174
fill_n, 174
filter_view
filter_view, 87
find, 158
find_end, 159
find_first_of, 160
find_if, 158
find_if_not, 158
for_each, 157
forward_iterator_tag, 45
ForwardIterator, 40
ForwardRange, 77
front
view_interface, 79
front_insert_iterator, 50

generate, 175
generate_n, 175
get
subrange, 83
greater, 20
greater<>, 21
greater_equal, 20
greater_equal<>, 21

includes, 196
Incrementable, 38
incrementable_traits, 31
IndirectlyComparable, 44
IndirectlyCopyable, 44
IndirectlyCopyableStorable, 44
IndirectlyMovable, 43
IndirectlyMovableStorable, 44
IndirectlySwappable, 44
IndirectRegularUnaryInvocable, 42
IndirectRelation, 42
IndirectStrictWeakOrder, 42
IndirectUnaryInvocable, 42
IndirectUnaryPredicate, 42
inner_iterator
split_view::inner_iterator, 117
inplace_merge, 195
input_iterator_tag, 45
InputIterator, 39
InputRange, 77
insert_iterator, 50
constructor, 51
inserter, 51
iota_view
iota_view, 99
is_heap, 202, 203
is_heap_until, 203
is_partitioned, 190
is_permutation, 165
is_sorted, 186
is_sorted_until, 186
istream_iterator
constructor, 66
operator!=, 66
operator==, 66
istreambuf_iterator, 66
constructor, 67
operator!=, 67
operator==, 67
iter_difference_t, 31
iter_move
common_iterator, 58
counted_iterator, 64
filter_view::iterator, 89
join_view::iterator, 110
move_iterator, 54
reverse_iterator, 49
split_view::inner_iterator, 118
transform_view::iterator, 95
iter_swap, 171
common_iterator, 59

- counted_iterator, 64
- filter_view::iterator, 90
- join_view::iterator, 110
- move_iterator, 54
- reverse_iterator, 49
- split_view::inner_iterator, 118
- transform_view::iterator, 96

iter_value_t, 32
Iterator, 38
<iterator>, 24
iterator
- basic_string_view, 22
- filter_view, 87
- filter_view::iterator, 88
- iota_view::iterator, 100
- join_view::iterator, 108
- transform_view::iterator, 93, 94

iterator_category
- iterator_traits, 33

iterator_traits, 33
- iterator_category, 33
- pointer, 33
- reference, 33

join_view
- join_view, 106

less, 20
less<>, 21
less_equal, 20
less_equal<>, 22
lexicographical_compare, 207
lower_bound, 188

make_heap, 201
make_move_iterator, 54
make_reverse_iterator, 49
max, 204
max_element, 206
merge, 194
Mergeable, 45
min, 203, 204
min_element, 205
minmax, 205
minmax_element, 207
mismatch, 162
move, 169
- algorithm, 169

move_backward, 169
move_iterator, 51
- iter_move, 54
- iter_swap, 54
- operator!=, 53
- operator*, 53
- operator+, 54
- operator++, 53
- operator-, 53
- operator->, 53
- operator==, 53
- operator[], 53

move_sentinel, 54
- constructor, 55
- move_sentinel, 55
- operator=, 55

next, 47
- subrange, 83

next_permutation, 208
none_of, 157
not_equal_to, 20
not_equal_to<>, 21
nth_element, 187

operator *PairLike*
- subrange, 83

operator bool
- view_interface, 79

operator!=
- common_iterator, 58
- counted_iterator, 63
- istream_iterator, 66
- istreambuf_iterator, 67
- move_iterator, 53
- unreachable, 65

operator*
- common_iterator, 57
- counted_iterator, 61
- filter_view::iterator, 89
- iota_view::iterator, 100
- join_view::iterator, 108
- move_iterator, 53
- split_view::inner_iterator, 117
- split_view::outer_iterator, 115
- transform_view::iterator, 94

operator+
- counted_iterator, 62
- iota_view::iterator, 102
- move_iterator, 54
- transform_view::iterator, 95

operator++
- common_iterator, 58
- counted_iterator, 61, 62
- filter_view::iterator, 89
- iota_view::iterator, 100, 101
- join_view::iterator, 109
- move_iterator, 53
- split_view::inner_iterator, 117
- split_view::outer_iterator, 115
- transform_view::iterator, 94

operator+=
- counted_iterator, 62
- iota_view::iterator, 101
- transform_view::iterator, 94

operator-
- common_iterator, 58
- counted_iterator, 63
- iota_view::iterator, 102
- move_iterator, 53

  - transform_view::iterator, 95
- operator-=
  - counted_iterator, 63
  - iota_view::iterator, 101
  - transform_view::iterator, 95
- operator->
  - common_iterator, 57
  - filter_view::iterator, 89
  - join_view::iterator, 108
  - move_iterator, 53
  - reverse_iterator, 49
- operator--
  - counted_iterator, 62
  - filter_view::iterator, 89
  - iota_view::iterator, 101
  - join_view::iterator, 109
  - transform_view::iterator, 94
- operator<
  - counted_iterator, 64
  - iota_view::iterator, 101
  - transform_view::iterator, 95
- operator<=
  - counted_iterator, 64
  - iota_view::iterator, 101
  - transform_view::iterator, 95
- operator=
  - common_iterator, 57
  - counted_iterator, 61
  - move_sentinel, 55
- operator==
  - common_iterator, 58
  - counted_iterator, 63
  - filter_view::iterator, 89
  - filter_view::sentinel, 90
  - iota_view::iterator, 101
  - iota_view::sentinel, 102
  - istream_iterator, 66
  - istreambuf_iterator, 67
  - join_view::iterator, 109
  - join_view::sentinel, 110
  - move_iterator, 53
  - split_view::inner_iterator, 117, 118
  - split_view::outer_iterator, 115, 116
  - take_view::sentinel, 105
  - transform_view::iterator, 95
  - transform_view::sentinel, 96
  - unreachable, 65
- operator>
  - counted_iterator, 64
  - iota_view::iterator, 101
  - transform_view::iterator, 95
- operator>=
  - counted_iterator, 64
  - iota_view::iterator, 102
  - transform_view::iterator, 95
- operator[]
  - counted_iterator, 63
  - iota_view::iterator, 101
  - move_iterator, 53
  - transform_view::iterator, 95
  - view_interface, 79
- ostreambuf_iterator, 67
- outer_iterator
  - split_view::outer_iterator, 115
- output_iterator_tag, 45
- OutputIterator, 39
- OutputRange, 77

- partial_sort, 184
- partial_sort_copy, 185
- partition, 191
- partition_copy, 192
- partition_point, 193
- Permutable, 45
- pointer
  - basic_string_view, 22
  - iterator_traits, 33
- pop_heap, 201
- prev, 47
  - subrange, 83
- prev_permutation, 209
- projected, 43
- push_heap, 201

- random_access_iterator_tag, 45
- RandomAccessIterator, 40
- RandomAccessRange, 77
- Range, 75
- <ranges>, 68
- ranges::swap, 7
- Readable, 37
- readable_traits, 32
- reference
  - basic_string_view, 22
  - iterator_traits, 33
- remove, 176
- remove_copy, 177
- remove_copy_if, 177
- remove_if, 176
- replace, 172
- replace_copy, 173
- replace_copy_if, 173
- replace_if, 172
- reverse, 180
- reverse_copy, 180
- reverse_iterator, 47
  - basic_string_view, 22
  - iter_move, 49
  - iter_swap, 49
  - make_reverse_iterator non-member function, 49
  - operator->, 49
- reverse_view
  - reverse_view, 121
- rotate, 180
- rotate_copy, 181

- search, 166

search_n, 166
Sentinel, 38
sentinel
    filter_view, 90
    filter_view::sentinel, 90
    iota_view::sentinel, 102
    join_view::sentinel, 110
    take_view::sentinel, 104
    transform_view::sentinel, 96
set_difference, 199
set_intersection, 198
set_symmetric_difference, 199
set_union, 197
shuffle, 182
single_view
    single_view, 112
size
    ref-view, 86
    common_view, 119
    iota_view, 99
    reverse_view, 121
    single_view, 112
    subrange, 83
    take_view, 104
    transform_view, 92
    view_interface, 79
size_type
    basic_string_view, 22
SizedRange, 76
SizedSentinel, 39
sort, 183
sort_heap, 202
Sortable, 45
span, 23
split_view
    split_view, 113
stable_partition, 192
stable_sort, 183
subrange, 80
    subrange, 82
swap_ranges, 170
Swappable, 7
SwappableWith, 8

take_view
    take_view, 103
traits_type
    basic_string_view, 22
transform, 171
transform_view
    transform_view, 91

uninitialized_copy, 15
uninitialized_copy_n, 16
uninitialized_default_construct, 14
uninitialized_default_construct_n, 14
uninitialized_fill, 17
uninitialized_fill_n, 18
uninitialized_move, 16
uninitialized_move_n, 17
uninitialized_value_construct, 15
uninitialized_value_construct_n, 15
unique, 178
unique_copy, 178
unreachable, 64
    operator!=, 65
    operator==, 65
upper_bound, 188

value_type
    basic_string_view, 22
    split_view::outer_iterator::value_type, 116
View, 76
view_interface, 78
ViewableRange, 78

WeaklyIncrementable, 37
Writable, 37

Index of implementation-defined behavior

The entries in this section are rough descriptions; exact specifications are at the indicated page in the general text.

type of `basic_string_view::const_iterator`, 22