Document Number: P1718R0
Date: 2019-06-16
Reply to: Marshall Clow
CppAlliance
mclow.lists@gmail.com

Mandating the Standard Library: Clause 25 - Algorithms library

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

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

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

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

— Drive-by fixes (hopefully very few)

This paper covers Clause 25 (Algorithms), and is based on N4810.

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

— algorithms.requirements 25.2
— alg.foreach 25.5.4
— alg.is.permutation 25.5.12
— alg.copy 25.6.1
— alg.move 25.6.2
— alg.swap 25.6.3
— alg.transform 25.6.4
— alg.replace 25.6.5
— alg.fill 25.6.6
— alg.generate 25.6.7
— alg.remove 25.6.8
— alg.unique 25.6.9
— alg.reverse 25.6.10
— alg.rotate 25.6.11
— alg.random.sample 25.6.12
— alg.random.shuffle 25.6.13
— alg.shift 25.6.14
— sort 25.7.1.1
— stable.sort 25.7.1.2
— partial.sort 25.7.1.3
— partial.sort.copy 25.7.1.4
— alg.nth.element 25.7.2
— lower.bound 25.7.3.1
— upper.bound 25.7.3.2
— equal.range 25.7.3.3
— binary.search 25.7.3.4
— alg.partitions 25.7.4
— alg.merge 25.7.5
— includes 25.7.6.1
— set.intersection 25.7.6.3
— set.difference 25.7.6.4
— set.symmetric.difference 25.7.6.5
— push.heap 25.7.7.1
— pop.heap 25.7.7.2
— make.heap 25.7.7.3
— sort.heap 25.7.7.4
— alg.min.max 25.7.8
— alg.clamp 25.7.9
— alg.3way 25.7.11
— alg.permutation.generators 25.7.12
— accumulate 25.9.2
— reduce 25.9.3
— inner.product 25.9.4
— transform.reduce 25.9.5
— exclusive.scan 25.9.7
— inclusive.scan 25.9.8
— transform.exclusive.scan 25.9.9
— transform.inclusive.scan 25.9.10
— adjacent.difference 25.9.11
— numeric.iota 25.9.12
— numeric.ops.gcd 25.9.13
— numeric.ops.lcm 25.9.14

Drive-by fixes:

— All of the _n algorithms now treat their `Size` parameter the same way. This also resolves LWG#3213.

— As I was "de-shalling" all the Requirements, I applied the guidance at https://github.com/cplusplus/draft/wiki/Specificat Style-Guidelines#requirements-expressed-by-concepts.

— Change "shall satisfy" to "shall meet" for old concepts.

Open questions:

— alg.unique 25.6.9 P7.2.3 is kind of a mess. Not quite sure how to "de-shall" it.

— alg.random.sample 25.6.12 has a `Distance` parameter that looks like the `Size` parameter from _n. Should it be specified the same way?

— Where should "XX is writeable to YY" and "the expression 'Foo/bar' is valid" go? Mandates? Expects? I put them in "Expects".

Thanks to Daniel Krügler and Tim Song for their advice and reviews.

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

```
git clone git@github.com:mclow/mandate.git
cd mandate
git diff master..chapter25 algorithms.tex
```

25 Algorithms library [algorithms]

25.1 General [algorithms.general]

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

2 The following subclauses describe components for non-modifying sequence operations, mutating sequence operations, sorting and related operations, and algorithms from the ISO C library, as summarized in Table 80.

Table 80 — Algorithms library summary

	Subclause	Header
25.2	Algorithms requirements	
25.3	Parallel algorithms	
25.5	Non-modifying sequence operations	`<algorithm>`
25.6	Mutating sequence operations	
25.7	Sorting and related operations	
25.9	Generalized numeric operations	`<numeric>`
25.10	C library algorithms	`<cstdlib>`

25.2 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 entities defined in the `std::ranges` namespace in this Clause are not found by argument-dependent name lookup (??). When found by unqualified (??) name lookup for the *postfix-expression* in a function call (??), they inhibit argument-dependent name lookup.

[*Example*:

```
void foo() {
  using namespace std::ranges;
  std::vector<int> vec{1,2,3};
  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 (??) than `std::ranges::find` since the former requires its first two parameters to have the same type. — *end example*]

3 For purposes of determining the existence of data races, algorithms shall not modify objects referenced through an iterator argument unless the specification requires such modification.

4 Throughout this Clause, where the template parameters are not constrained, the names of template parameters are used to express type requirements.

(4.1) — If an algorithm's template parameter is named `InputIterator`, `InputIterator1`, or `InputIterator2`, the template argument shall ~~satisfy~~meet the *Cpp17InputIterator* requirements (??).

(4.2) — If an algorithm's template parameter is named `OutputIterator`, `OutputIterator1`, or `OutputIterator2`, the template argument shall ~~satisfy~~meet the *Cpp17OutputIterator* requirements (??).

(4.3) — If an algorithm's template parameter is named `ForwardIterator`, `ForwardIterator1`, or `ForwardIterator2`, the template argument shall ~~satisfy~~meet the *Cpp17ForwardIterator* requirements (??).

(4.4) — If an algorithm's template parameter is named `BidirectionalIterator`, `BidirectionalIterator1`, or `BidirectionalIterator2`, the template argument shall ~~satisfy~~meet the *Cpp17BidirectionalIterator* requirements (??).

(4.5) — If an algorithm's template parameter is named `RandomAccessIterator`, `RandomAccessIterator1`, or `RandomAccessIterator2`, the template argument shall ~~satisfy~~meet the *Cpp17RandomAccessIterator* requirements (**??**).

5 If an algorithm's *Effects:* element specifies that a value pointed to by any iterator passed as an argument is modified, then that algorithm has an additional type requirement: The type of that argument shall satisfy the requirements of a mutable iterator (**??**). [*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.[234] When such a version is provided for *algorithm* it is called *algorithm*`_copy`. Algorithms that take predicates end with the suffix `_if` (which follows the suffix `_copy`).

7 When not otherwise constrained, the `Predicate` parameter is used whenever an algorithm expects a function object (**??**) that, when applied to the result of dereferencing the corresponding iterator, returns a value testable as `true`. In other words, if an algorithm takes `Predicate pred` as its argument and `first` as its iterator argument with value type `T`, it should work correctly in the construct `pred(*first)` contextually converted to `bool` (**??**). The function object `pred` shall not apply any non-constant function through the dereferenced iterator. Given a glvalue `u` of type (possibly `const`) `T` that designates the same object as `*first`, `pred(u)` shall be a valid expression that is equal to `pred(*first)`.

8 When not otherwise constrained, the `BinaryPredicate` parameter is used whenever an algorithm expects a function object that when applied to the result of dereferencing two corresponding iterators or to dereferencing an iterator and type `T` when `T` is part of the signature returns a value testable as `true`. In other words, if an algorithm takes `BinaryPredicate binary_pred` as its argument and `first1` and `first2` as its iterator arguments with respective value types `T1` and `T2`, it should work correctly in the construct `binary_pred(*first1, *first2)` contextually converted to `bool` (**??**). Unless otherwise specified, `BinaryPredicate` always takes the first iterator's `value_type` as its first argument, that is, in those cases when `T value` is part of the signature, it should work correctly in the construct `binary_pred(*first1, value)` contextually converted to `bool` (**??**). `binary_pred` shall not apply any non-constant function through the dereferenced iterators. Given a glvalue `u` of type (possibly `const`) `T1` that designates the same object as `*first1`, and a glvalue `v` of type (possibly `const`) `T2` that designates the same object as `*first2`, `binary_pred(u, *first2)`, `binary_pred(*first1, v)`, and `binary_pred(u, v)` shall each be a valid expression that is equal to `binary_pred(*first1, *first2)`, and `binary_pred(u, value)` shall be a valid expression that is equal to `binary_pred(*first1, value)`.

9 The parameters `UnaryOperation`, `BinaryOperation`, `BinaryOperation1`, and `BinaryOperation2` are used whenever an algorithm expects a function object (**??**).

10 [*Note*: Unless otherwise specified, algorithms that take function objects as arguments are permitted to copy those function objects freely. Programmers for whom object identity is important should consider using a wrapper class that points to a noncopied implementation object such as `reference_wrapper<T>` (**??**), or some equivalent solution. — *end note*]

11 When the description of an algorithm gives an expression such as `*first == value` for a condition, the expression shall evaluate to either `true` or `false` in boolean contexts.

12 In the description of the algorithms, operator `+` is used for some of the iterator categories for which it does not have to be defined. In these cases the semantics of `a + n` are the same as those of

```
auto tmp = a;
for (; n < 0; ++n) --tmp;
for (; n > 0; --n) ++tmp;
return tmp;
```

Similarly, operator `-` is used for some combinations of iterators and sentinel types for which it does not have to be defined. If `[a, b)` denotes a range, the semantics of `b - a` in these cases are the same as those of

```
iter_difference_t<remove_reference_t<decltype(a)>> n = 0;
for (auto tmp = a; tmp != b; ++tmp) ++n;
return n;
```

234) The decision whether to include a copying version was usually based on complexity considerations. When the cost of doing the operation dominates the cost of copy, the copying version is not included. For example, `sort_copy` is not included because the cost of sorting is much more significant, and users might as well do `copy` followed by `sort`.

and if `[b, a)` denotes a range, the same as those of

```
iter_difference_t<remove_reference_t<decltype(b)>> n = 0;
for (auto tmp = b; tmp != a; ++tmp) --n;
return n;
```

13 In the description of algorithm return values, a sentinel value `s` denoting the end of a range `[i, s)` is sometimes returned where an iterator is expected. In these cases, the semantics are as if the sentinel is converted into an iterator using `ranges::next(i, s)`.

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

15 The number and order of deducible template parameters for algorithm declarations are unspecified, except where explicitly stated otherwise. [*Note*: Consequently, the algorithms may not be called with explicitly-specified template argument lists. — *end note*]

16 The class templates `binary_transform_result`, `for_each_result`, `minmax_result`, `mismatch_result`, `copy_result`, and `partition_copy_result` have the template parameters, data members, and special members specified below. They have no base classes or members other than those specified.

25.3 Parallel algorithms [algorithms.parallel]

1 This subclause describes components that C++ programs may use to perform operations on containers and other sequences in parallel.

25.3.1 Terms and definitions [algorithms.parallel.defns]

1 A *parallel algorithm* is a function template listed in this document with a template parameter named `ExecutionPolicy`.

2 Parallel algorithms access objects indirectly accessible via their arguments by invoking the following functions:

(2.1) — All operations of the categories of the iterators that the algorithm is instantiated with.

(2.2) — Operations on those sequence elements that are required by its specification.

(2.3) — User-provided function objects to be applied during the execution of the algorithm, if required by the specification.

(2.4) — Operations on those function objects required by the specification. [*Note*: See 25.2. — *end note*]

These functions are herein called *element access functions*. [*Example*: The `sort` function may invoke the following element access functions:

(2.5) — Operations of the random-access iterator of the actual template argument (as per **??**), as implied by the name of the template parameter `RandomAccessIterator`.

(2.6) — The `swap` function on the elements of the sequence (as per the preconditions specified in 25.7.1.1).

(2.7) — The user-provided `Compare` function object.

— *end example*]

3 A standard library function is *vectorization-unsafe* if it is specified to synchronize with another function invocation, or another function invocation is specified to synchronize with it, and if it is not a memory allocation or deallocation function. [*Note*: Implementations must ensure that internal synchronization inside standard library functions does not prevent forward progress when those functions are executed by threads of execution with weakly parallel forward progress guarantees. — *end note*] [*Example*:

```
int x = 0;
std::mutex m;
void f() {
  int a[] = {1,2};
  std::for_each(std::execution::par_unseq, std::begin(a), std::end(a), [&](int) {
    std::lock_guard<mutex> guard(m);            // incorrect: lock_guard constructor calls m.lock()
  ++x;
  });
}
```

The above program may result in two consecutive calls to `m.lock()` on the same thread of execution (which may deadlock), because the applications of the function object are not guaranteed to run on different threads of execution. — *end example*]

25.3.2 Requirements on user-provided function objects [algorithms.parallel.user]

1 Unless otherwise specified, function objects passed into parallel algorithms as objects of type `Predicate`, `BinaryPredicate`, `Compare`, `UnaryOperation`, `BinaryOperation`, `BinaryOperation1`, `BinaryOperation2`, and the operators used by the analogous overloads to these parallel algorithms that could be formed by the invocation with the specified default predicate or operation (where applicable) shall not directly or indirectly modify objects via their arguments, nor shall they rely on the identity of the provided objects.

25.3.3 Effect of execution policies on algorithm execution [algorithms.parallel.exec]

1 Parallel algorithms have template parameters named `ExecutionPolicy` (**??**) which describe the manner in which the execution of these algorithms may be parallelized and the manner in which they apply the element access functions.

2 If an object is modified by an element access function, the algorithm will perform no other unsynchronized accesses to that object. The modifying element access functions are those which are specified as modifying the object. [*Note*: For example, `swap()`, `++`, `--`, `@=`, and assignments modify the object. For the assignment and `@=` operators, only the left argument is modified. — *end note*]

3 Unless otherwise stated, implementations may make arbitrary copies of elements (with type `T`) from sequences where `is_trivially_copy_constructible_v<T>` and `is_trivially_destructible_v<T>` are `true`. [*Note*: This implies that user-supplied function objects should not rely on object identity of arguments for such input sequences. Users for whom the object identity of the arguments to these function objects is important should consider using a wrapping iterator that returns a non-copied implementation object such as `reference_wrapper<T>` (**??**) or some equivalent solution. — *end note*]

4 The invocations of element access functions in parallel algorithms invoked with an execution policy object of type `execution::sequenced_policy` all occur in the calling thread of execution. [*Note*: The invocations are not interleaved; see **??**. — *end note*]

5 The invocations of element access functions in parallel algorithms invoked with an execution policy object of type `execution::unsequenced_policy` are permitted to execute in an unordered fashion in the calling thread of execution, unsequenced with respect to one another in the calling thread of execution. [*Note*: This means that multiple function object invocations may be interleaved on a single thread of execution, which overrides the usual guarantee from **??** that function executions do not overlap with one another. — *end note*] The behavior of a program is undefined if it invokes a vectorization-unsafe standard library function from user code called from a `execution::unsequenced_policy` algorithm. [*Note*: Because `execution::unsequenced_policy` allows the execution of element access functions to be interleaved on a single thread of execution, blocking synchronization, including the use of mutexes, risks deadlock. — *end note*]

6 The invocations of element access functions in parallel algorithms invoked with an execution policy object of type `execution::parallel_policy` are permitted to execute either in the invoking thread of execution or in a thread of execution implicitly created by the library to support parallel algorithm execution. If the threads of execution created by `thread` (**??**) provide concurrent forward progress guarantees (**??**), then a thread of execution implicitly created by the library will provide parallel forward progress guarantees; otherwise, the provided forward progress guarantee is implementation-defined. Any such invocations executing in the same thread of execution are indeterminately sequenced with respect to each other. [*Note*: It is the caller's responsibility to ensure that the invocation does not introduce data races or deadlocks. — *end note*] [*Example*:

```
int a[] = {0,1};
std::vector<int> v;
std::for_each(std::execution::par, std::begin(a), std::end(a), [&](int i) {
  v.push_back(i*2+1);                                  // incorrect: data race
});
```

The program above has a data race because of the unsynchronized access to the container `v`. — *end example*] [*Example*:

```
std::atomic<int> x{0};
int a[] = {1,2};
```

```
std::for_each(std::execution::par, std::begin(a), std::end(a), [&](int) {
  x.fetch_add(1, std::memory_order::relaxed);
  // spin wait for another iteration to change the value of x
  while (x.load(std::memory_order::relaxed) == 1) { } // incorrect: assumes execution order
});
```

The above example depends on the order of execution of the iterations, and will not terminate if both iterations are executed sequentially on the same thread of execution. — *end example*] [*Example*:

```
int x = 0;
std::mutex m;
int a[] = {1,2};
std::for_each(std::execution::par, std::begin(a), std::end(a), [&](int) {
  std::lock_guard<mutex> guard(m);
  ++x;
});
```

The above example synchronizes access to object `x` ensuring that it is incremented correctly. — *end example*]

7 The invocations of element access functions in parallel algorithms invoked with an execution policy object of type `execution::parallel_unsequenced_policy` are permitted to execute in an unordered fashion in unspecified threads of execution, and unsequenced with respect to one another within each thread of execution. These threads of execution are either the invoking thread of execution or threads of execution implicitly created by the library; the latter will provide weakly parallel forward progress guarantees. [*Note*: This means that multiple function object invocations may be interleaved on a single thread of execution, which overrides the usual guarantee from **??** that function executions do not overlap with one another. — *end note*] The behavior of a program is undefined if it invokes a vectorization-unsafe standard library function from user code called from a `execution::parallel_unsequenced_policy` algorithm. [*Note*: Because `execution::parallel_unsequenced_policy` allows the execution of element access functions to be interleaved on a single thread of execution, blocking synchronization, including the use of mutexes, risks deadlock. — *end note*]

8 [*Note*: The semantics of invocation with `execution::unsequenced_policy`, `execution::parallel_policy`, or `execution::parallel_unsequenced_policy` allow the implementation to fall back to sequential execution if the system cannot parallelize an algorithm invocation, e.g., due to lack of resources. — *end note*]

9 If an invocation of a parallel algorithm uses threads of execution implicitly created by the library, then the invoking thread of execution will either

(9.1) — temporarily block with forward progress guarantee delegation (**??**) on the completion of these library-managed threads of execution, or

(9.2) — eventually execute an element access function;

the thread of execution will continue to do so until the algorithm is finished. [*Note*: In blocking with forward progress guarantee delegation in this context, a thread of execution created by the library is considered to have finished execution as soon as it has finished the execution of the particular element access function that the invoking thread of execution logically depends on. — *end note*]

10 The semantics of parallel algorithms invoked with an execution policy object of implementation-defined type are implementation-defined.

25.3.4 Parallel algorithm exceptions [algorithms.parallel.exceptions]

1 During the execution of a parallel algorithm, if temporary memory resources are required for parallelization and none are available, the algorithm throws a `bad_alloc` exception.

2 During the execution of a parallel algorithm, if the invocation of an element access function exits via an uncaught exception, the behavior is determined by the `ExecutionPolicy`.

25.3.5 `ExecutionPolicy` algorithm overloads [algorithms.parallel.overloads]

1 Parallel algorithms are algorithm overloads. Each parallel algorithm overload has an additional template type parameter named `ExecutionPolicy`, which is the first template parameter. Additionally, each parallel algorithm overload has an additional function parameter of type `ExecutionPolicy&&`, which is the first function parameter. [*Note*: Not all algorithms have parallel algorithm overloads. — *end note*]

2 Unless otherwise specified, the semantics of `ExecutionPolicy` algorithm overloads are identical to their overloads without.

[3] Unless otherwise specified, the complexity requirements of `ExecutionPolicy` algorithm overloads are relaxed from the complexity requirements of the overloads without as follows: when the guarantee says "at most *expr*" or "exactly *expr*" and does not specify the number of assignments or swaps, and *expr* is not already expressed with $\mathcal{O}()$ notation, the complexity of the algorithm shall be $\mathcal{O}(expr)$.

[4] Parallel algorithms shall not participate in overload resolution unless `is_execution_policy_v<remove_cvref_t<ExecutionPolicy>>` is `true`.

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

```
#include <initializer_list>

namespace std {
  // 25.5, non-modifying sequence operations
  // 25.5.1, 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 25.3.5
                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 = {});
    template<InputRange R, class Proj = identity,
             IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
      constexpr bool all_of(R&& r, Pred pred, Proj proj = {});
  }

  // 25.5.2, 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 25.3.5
                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 = {});
    template<InputRange R, class Proj = identity,
             IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
      constexpr bool any_of(R&& r, Pred pred, Proj proj = {});
  }

  // 25.5.3, 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 25.3.5
                 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 = {});
    template<InputRange R, class Proj = identity,
             IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
      constexpr bool none_of(R&& r, Pred pred, Proj proj = {});
  }

  // 25.5.4, 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 25.3.5
                ForwardIterator first, ForwardIterator last, Function f);

namespace ranges {
  template<class I, class F>
  struct for_each_result {
    [[no_unique_address]] I in;
    [[no_unique_address]] F fun;

    template<class I2, class F2>
      requires ConvertibleTo<const I&, I2> && ConvertibleTo<const F&, F2>
      operator for_each_result<I2, F2>() const & {
        return {in, fun};
      }

    template<class I2, class F2>
      requires ConvertibleTo<I, I2> && ConvertibleTo<F, F2>
      operator for_each_result<I2, F2>() && {
        return {std::move(in), std::move(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 = {});
  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 = {});
}

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 25.3.5
                             ForwardIterator first, Size n, Function f);

// 25.5.5, 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 25.3.5
                       ForwardIterator first, ForwardIterator last,
                       const T& value);
template<class InputIterator, class Predicate>
  constexpr InputIterator find_if(InputIterator first, InputIterator last,
                                  Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator find_if(ExecutionPolicy&& exec, // see 25.3.5
                          ForwardIterator first, ForwardIterator last,
                          Predicate pred);
template<class InputIterator, class Predicate>
  constexpr InputIterator find_if_not(InputIterator first, InputIterator last,
                                      Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator find_if_not(ExecutionPolicy&& exec, // see 25.3.5
                              ForwardIterator first, ForwardIterator last,
                              Predicate pred);
```

```
namespace ranges {
  template<InputIterator I, Sentinel<I> S, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to, projected<I, Proj>, const T*>
    constexpr I find(I first, S last, const T& value, 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 = {});
  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 = {});
  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 = {});
  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 = {});
  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 = {});
}

// 25.5.6, 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 25.3.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1,
         class ForwardIterator2, class BinaryPredicate>
  ForwardIterator1
    find_end(ExecutionPolicy&& exec, // see 25.3.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);

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

```
// 25.5.7, 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 25.3.5
                  ForwardIterator1 first1, ForwardIterator1 last1,
                  ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1,
         class ForwardIterator2, class BinaryPredicate>
  ForwardIterator1
    find_first_of(ExecutionPolicy&& exec, // see 25.3.5
                  ForwardIterator1 first1, ForwardIterator1 last1,
                  ForwardIterator2 first2, ForwardIterator2 last2,
                  BinaryPredicate pred);

namespace ranges {
  template<InputIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2,
           class Proj1 = identity, class Proj2 = identity,
           IndirectRelation<projected<I1, Proj1>,
                            projected<I2, Proj2>> Pred = ranges::equal_to>
    constexpr I1 find_first_of(I1 first1, S1 last1, I2 first2, S2 last2,
                               Pred pred = {},
                               Proj1 proj1 = {}, 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 = {},
                    Proj1 proj1 = {}, Proj2 proj2 = {});
}

// 25.5.8, 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 25.3.5
                  ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class BinaryPredicate>
  ForwardIterator
    adjacent_find(ExecutionPolicy&& exec, // see 25.3.5
                  ForwardIterator first, ForwardIterator last,
                  BinaryPredicate pred);

namespace ranges {
  template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
           IndirectRelation<projected<I, Proj>> Pred = ranges::equal_to>
    constexpr I adjacent_find(I first, S last, Pred pred = {},
                              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 = {}, Proj proj = {});
}

// 25.5.9, 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 25.3.5
          ForwardIterator first, ForwardIterator last, const T& value);
template<class InputIterator, class Predicate>
  constexpr typename iterator_traits<InputIterator>::difference_type
    count_if(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  typename iterator_traits<ForwardIterator>::difference_type
    count_if(ExecutionPolicy&& exec, // see 25.3.5
             ForwardIterator first, ForwardIterator last, Predicate pred);

namespace ranges {
  template<InputIterator I, Sentinel<I> S, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to, projected<I, Proj>, const T*>
    constexpr iter_difference_t<I>
      count(I first, S last, const T& value, 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 = {});
  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 = {});
  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 = {});
}

// 25.5.10, 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 25.3.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2);
```

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec, // see 25.3.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec, // see 25.3.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec, // see 25.3.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);

namespace ranges {
  template<class I1, class I2>
  struct mismatch_result {
    [[no_unique_address]] I1 in1;
    [[no_unique_address]] I2 in2;

    template<class II1, class II2>
      requires ConvertibleTo<const I1&, II1> && ConvertibleTo<const I2&, II2>
      operator mismatch_result<II1, II2>() const & {
        return {in1, in2};
      }

    template<class II1, class II2>
      requires ConvertibleTo<I1, II1> && ConvertibleTo<I2, II2>
      operator mismatch_result<II1, II2>() && {
        return {std::move(in1), std::move(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 = {},
               Proj1 proj1 = {}, 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 = {},
               Proj1 proj1 = {}, Proj2 proj2 = {});
}

// 25.5.11, 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 25.3.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  bool equal(ExecutionPolicy&& exec, // see 25.3.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  bool equal(ExecutionPolicy&& exec, // see 25.3.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  bool equal(ExecutionPolicy&& exec, // see 25.3.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);

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

// 25.5.12, 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 = {},
                                  Proj1 proj1 = {}, 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 = {},
                                  Proj1 proj1 = {}, Proj2 proj2 = {});
}

// 25.5.13, 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 25.3.5
           ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  ForwardIterator1
    search(ExecutionPolicy&& exec, // see 25.3.5
           ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2,
           BinaryPredicate pred);

namespace ranges {
  template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
           Sentinel<I2> S2, class Pred = ranges::equal_to,
           class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
    constexpr subrange<I1>
      search(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = {},
             Proj1 proj1 = {}, 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 = {},
             Proj1 proj1 = {}, 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 25.3.5
             ForwardIterator first, ForwardIterator last,
             Size count, const T& value);
template<class ExecutionPolicy, class ForwardIterator, class Size, class T,
         class BinaryPredicate>
  ForwardIterator
    search_n(ExecutionPolicy&& exec, // see 25.3.5
             ForwardIterator first, ForwardIterator last,
             Size count, const T& value,
             BinaryPredicate pred);
```

```
namespace ranges {
  template<ForwardIterator I, Sentinel<I> S, class T,
           class Pred = ranges::equal_to, class Proj = identity>
    requires IndirectlyComparable<I, const T*, Pred, Proj>
    constexpr subrange<I>
      search_n(I first, S last, iter_difference_t<I> count,
               const T& value, Pred pred = {}, 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 = {}, Proj proj = {});
}

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

// 25.6, mutating sequence operations
// 25.6.1, 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 25.3.5
                        ForwardIterator1 first, ForwardIterator1 last,
                        ForwardIterator2 result);

namespace ranges {
  template<class I, class O>
  struct copy_result {
    [[no_unique_address]] I in;
    [[no_unique_address]] O out;

    template<class I2, class O2>
      requires ConvertibleTo<const I&, I2> && ConvertibleTo<const O&, O2>
      operator copy_result<I2, O2>() const & {
        return {in, out};
      }

    template<class I2, class O2>
      requires ConvertibleTo<I, I2> && ConvertibleTo<O, O2>
      operator copy_result<I2, O2>() && {
        return {std::move(in), std::move(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 25.3.5
```

```
                          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 25.3.5
                           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 = {});
  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 = {});
}

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);
}

// 25.6.2, move
template<class InputIterator, class OutputIterator>
  constexpr OutputIterator move(InputIterator first, InputIterator last,
                                OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1,
         class ForwardIterator2>
  ForwardIterator2 move(ExecutionPolicy&& exec, // see 25.3.5
```

```
                        ForwardIterator1 first, ForwardIterator1 last,
                        ForwardIterator2 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 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);
}

// 25.6.3, 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 25.3.5
                               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>
  constexpr void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

```
  // 25.6.4, 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 25.3.5
                ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 result, UnaryOperation op);
  template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
           class ForwardIterator, class BinaryOperation>
    ForwardIterator
      transform(ExecutionPolicy&& exec, // see 25.3.5
                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 = {});
    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 = {});

    template<class I1, class I2, class O>
    struct binary_transform_result {
      [[no_unique_address]] I1 in1;
      [[no_unique_address]] I2 in2;
      [[no_unique_address]] O  out;

      template<class II1, class II2, class OO>
        requires ConvertibleTo<const I1&, II1> &&
          ConvertibleTo<const I2&, II2> && ConvertibleTo<const O&, OO>
        operator binary_transform_result<II1, II2, OO>() const & {
          return {in1, in2, out};
        }

      template<class II1, class II2, class OO>
        requires ConvertibleTo<I1, II1> &&
          ConvertibleTo<I2, II2> && ConvertibleTo<O, OO>
        operator binary_transform_result<II1, II2, OO>() && {
          return {std::move(in1), std::move(in2), std::move(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 = {}, 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 = {}, Proj2 proj2 = {});
}

// 25.6.5, 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 25.3.5
               ForwardIterator first, ForwardIterator last,
               const T& old_value, const T& new_value);
template<class ForwardIterator, class Predicate, class T>
  constexpr void replace_if(ForwardIterator first, ForwardIterator last,
                            Predicate pred, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator, class Predicate, class T>
  void replace_if(ExecutionPolicy&& exec, // see 25.3.5
                  ForwardIterator first, ForwardIterator last,
                  Predicate pred, const T& new_value);

namespace ranges {
  template<InputIterator I, Sentinel<I> S, class T1, class T2, class Proj = identity>
    requires Writable<I, const T2&> &&
             IndirectRelation<ranges::equal_to, projected<I, Proj>, const T1*>
    constexpr I
      replace(I first, S last, const T1& old_value, const T2& new_value, 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 = {});
  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 = {});
  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 = {});
}

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 25.3.5
                                ForwardIterator1 first, ForwardIterator1 last,
                                ForwardIterator2 result,
                                const T& old_value, const T& new_value);
```

```
template<class InputIterator, class OutputIterator, class Predicate, class T>
  constexpr OutputIterator replace_copy_if(InputIterator first, InputIterator last,
                                           OutputIterator result,
                                           Predicate pred, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate, class T>
  ForwardIterator2 replace_copy_if(ExecutionPolicy&& exec, // see 25.3.5
                                   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 = {});
  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 = {});

  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 = {});
  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 = {});
}

// 25.6.6, 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 25.3.5
            ForwardIterator first, ForwardIterator last, const T& value);
template<class OutputIterator, class Size, class T>
  constexpr OutputIterator fill_n(OutputIterator first, Size n, const T& value);
template<class ExecutionPolicy, class ForwardIterator,
         class Size, class T>
  ForwardIterator fill_n(ExecutionPolicy&& exec, // see 25.3.5
                         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);
}

// 25.6.7, 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 25.3.5
                ForwardIterator first, ForwardIterator last,
                Generator gen);
template<class OutputIterator, class Size, class Generator>
  constexpr OutputIterator generate_n(OutputIterator first, Size n, Generator gen);
template<class ExecutionPolicy, class ForwardIterator, class Size, class Generator>
  ForwardIterator generate_n(ExecutionPolicy&& exec, // see 25.3.5
                             ForwardIterator first, Size n, Generator gen);

namespace ranges {
  template<Iterator O, Sentinel<O> S, CopyConstructible F>
    requires Invocable<F&> && Writable<O, invoke_result_t<F&>>
    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);
}

// 25.6.8, 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 25.3.5
                         ForwardIterator first, ForwardIterator last,
                         const T& value);
template<class ForwardIterator, class Predicate>
  constexpr ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                                      Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator remove_if(ExecutionPolicy&& exec, // see 25.3.5
                            ForwardIterator first, ForwardIterator last,
                            Predicate pred);

namespace ranges {
  template<Permutable I, Sentinel<I> S, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to, projected<I, Proj>, const T*>
    constexpr I remove(I first, S last, const T& value, 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 = {});
  template<Permutable I, Sentinel<I> S, class Proj = identity,
           IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr I remove_if(I first, S last, Pred pred, 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 = {});
}

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 25.3.5
                ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result, const T& value);
template<class InputIterator, class OutputIterator, class Predicate>
  constexpr OutputIterator
    remove_copy_if(InputIterator first, InputIterator last,
                   OutputIterator result, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate>
  ForwardIterator2
    remove_copy_if(ExecutionPolicy&& exec, // see 25.3.5
                   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 = {});
  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 = {});

  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 = {});
  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 = {});
}

// 25.6.9, 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 25.3.5
                         ForwardIterator first, ForwardIterator last);
```

```
template<class ExecutionPolicy, class ForwardIterator, class BinaryPredicate>
  ForwardIterator unique(ExecutionPolicy&& exec, // see 25.3.5
                         ForwardIterator first, ForwardIterator last,
                         BinaryPredicate pred);

namespace ranges {
  template<Permutable I, Sentinel<I> S, class Proj = identity,
           IndirectRelation<projected<I, Proj>> C = ranges::equal_to>
    constexpr I unique(I first, S last, C comp = {}, 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 = {}, 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 25.3.5
                ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  ForwardIterator2
    unique_copy(ExecutionPolicy&& exec, // see 25.3.5
                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 = {}, 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 = {}, Proj proj = {});
}

// 25.6.10, reverse
template<class BidirectionalIterator>
  constexpr void reverse(BidirectionalIterator first, BidirectionalIterator last);
template<class ExecutionPolicy, class BidirectionalIterator>
  void reverse(ExecutionPolicy&& exec, // see 25.3.5
               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 25.3.5
                 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);
}

// 25.6.11, rotate
template<class ForwardIterator>
  constexpr ForwardIterator rotate(ForwardIterator first,
                                   ForwardIterator middle,
                                   ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator rotate(ExecutionPolicy&& exec, // see 25.3.5
                         ForwardIterator first,
                         ForwardIterator middle,
                         ForwardIterator last);

namespace ranges {
  template<Permutable I, Sentinel<I> S>
    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 25.3.5
                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);
}

// 25.6.12, sample
template<class PopulationIterator, class SampleIterator,
         class Distance, class UniformRandomBitGenerator>
  SampleIterator sample(PopulationIterator first, PopulationIterator last,
                        SampleIterator out, Distance n,
                        UniformRandomBitGenerator&& g);

// 25.6.13, 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);
}

// 25.6.14, shift
template<class ForwardIterator>
  constexpr ForwardIterator
    shift_left(ForwardIterator first, ForwardIterator last,
               typename iterator_traits<ForwardIterator>::difference_type n);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator
    shift_left(ExecutionPolicy&& exec, // see 25.3.5
               ForwardIterator first, ForwardIterator last,
               typename iterator_traits<ForwardIterator>::difference_type n);
template<class ForwardIterator>
  constexpr ForwardIterator
    shift_right(ForwardIterator first, ForwardIterator last,
                typename iterator_traits<ForwardIterator>::difference_type n);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator
    shift_right(ExecutionPolicy&& exec, // see 25.3.5
                ForwardIterator first, ForwardIterator last,
                typename iterator_traits<ForwardIterator>::difference_type n);

// 25.7, sorting and related operations
// 25.7.1, 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 25.3.5
            RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  void sort(ExecutionPolicy&& exec, // see 25.3.5
            RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);

namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less,
           class Proj = identity>
    requires Sortable<I, Comp, Proj>
    constexpr I
      sort(I first, S last, Comp comp = {}, 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 = {}, 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 25.3.5
                   RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  void stable_sort(ExecutionPolicy&& exec, // see 25.3.5
                   RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);

namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less,
           class Proj = identity>
    requires Sortable<I, Comp, Proj>
    I stable_sort(I first, S last, Comp comp = {}, 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 = {}, 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 25.3.5
                    RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  void partial_sort(ExecutionPolicy&& exec, // see 25.3.5
                    RandomAccessIterator first,
                    RandomAccessIterator middle,
```

```
                    RandomAccessIterator last, Compare comp);

namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less,
           class Proj = identity>
    requires Sortable<I, Comp, Proj>
    constexpr I
      partial_sort(I first, I middle, S last, Comp comp = {}, 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 = {},
                   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 25.3.5
                      ForwardIterator first, ForwardIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last);
template<class ExecutionPolicy, class ForwardIterator, class RandomAccessIterator,
         class Compare>
  RandomAccessIterator
    partial_sort_copy(ExecutionPolicy&& exec,  // see 25.3.5
                      ForwardIterator first, ForwardIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last,
                      Compare comp);

namespace ranges {
  template<InputIterator I1, Sentinel<I1> S1, RandomAccessIterator I2, Sentinel<I2> S2,
           class Comp = ranges::less, class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyCopyable<I1, I2> && Sortable<I2, Comp, Proj2> &&
             IndirectStrictWeakOrder<Comp, projected<I1, Proj1>, projected<I2, Proj2>>
    constexpr I2
      partial_sort_copy(I1 first, S1 last, I2 result_first, S2 result_last,
                        Comp comp = {}, Proj1 proj1 = {}, 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 = {},
                        Proj1 proj1 = {}, 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 25.3.5
                 ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  bool is_sorted(ExecutionPolicy&& exec, // see 25.3.5
                 ForwardIterator first, ForwardIterator last,
                 Compare comp);

namespace ranges {
  template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
           IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less>
    constexpr bool is_sorted(I first, S last, Comp comp = {}, 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 = {}, 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 25.3.5
                    ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  ForwardIterator
    is_sorted_until(ExecutionPolicy&& exec, // see 25.3.5
                    ForwardIterator first, ForwardIterator last,
                    Compare comp);

namespace ranges {
  template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
           IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less>
    constexpr I is_sorted_until(I first, S last, Comp comp = {}, 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 = {}, Proj proj = {});
}

// 25.7.2, 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 25.3.5
                   RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  void nth_element(ExecutionPolicy&& exec, // see 25.3.5
                   RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last, Compare comp);
```

```
namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less,
           class Proj = identity>
    requires Sortable<I, Comp, Proj>
    constexpr I
      nth_element(I first, I nth, S last, Comp comp = {}, 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 = {}, Proj proj = {});
}

// 25.7.3, 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 = {},
                            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 = {}, 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 = {}, 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 = {}, 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 = {}, 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 = {}, 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 = {},
                                 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 = {},
                                 Proj proj = {});
}

// 25.7.4, 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 25.3.5
                      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 = {});
  template<InputRange R, class Proj = identity,
           IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr bool is_partitioned(R&& r, Pred pred, 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 25.3.5
                            ForwardIterator first,
                            ForwardIterator last,
                            Predicate pred);

namespace ranges {
  template<Permutable I, Sentinel<I> S, class Proj = identity,
           IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr I
      partition(I first, S last, Pred pred, 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 = {});
}

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 25.3.5
                                         BidirectionalIterator first,
                                         BidirectionalIterator last,
                                         Predicate pred);

namespace ranges {
  template<BidirectionalIterator I, Sentinel<I> S, class Proj = identity,
           IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires Permutable<I>
    I stable_partition(I first, S last, Pred pred, Proj proj = {});
  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 = {});
}

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 25.3.5
                   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 {
    [[no_unique_address]] I  in;
    [[no_unique_address]] O1 out1;
    [[no_unique_address]] O2 out2;

    template<class II, class OO1, class OO2>
      requires ConvertibleTo<const I&, II> &&
        ConvertibleTo<const O1&, OO1> && ConvertibleTo<const O2&, OO2>
      operator partition_copy_result<II, OO1, OO2>() const & {
        return {in, out1, out2};
      }

    template<class II, class OO1, class OO2>
      requires ConvertibleTo<I, II> &&
        ConvertibleTo<O1, OO1> && ConvertibleTo<O2, OO2>
      operator partition_copy_result<II, OO1, OO2>() && {
        return {std::move(in), std::move(out1), std::move(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 = {});
  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 = {});
}

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 = {});
  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 = {});
}

// 25.7.5, 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 25.3.5
          ForwardIterator1 first1, ForwardIterator1 last1,
          ForwardIterator2 first2, ForwardIterator2 last2,
          ForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  ForwardIterator
    merge(ExecutionPolicy&& exec, // see 25.3.5
          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 = {}, Proj1 proj1 = {}, 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 = {}, Proj1 proj1 = {}, 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 25.3.5
                     BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last);
template<class ExecutionPolicy, class BidirectionalIterator, class Compare>
  void inplace_merge(ExecutionPolicy&& exec, // see 25.3.5
                     BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last, Compare comp);

namespace ranges {
  template<BidirectionalIterator I, Sentinel<I> S, class Comp = ranges::less,
           class Proj = identity>
    requires Sortable<I, Comp, Proj>
    I inplace_merge(I first, I middle, S last, Comp comp = {}, 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 = {},
                    Proj proj = {});
}

// 25.7.6, 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 25.3.5
                ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Compare>
  bool includes(ExecutionPolicy&& exec, // see 25.3.5
                ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2, ForwardIterator2 last2,
                Compare comp);
```

```
namespace ranges {
  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
           class Proj1 = identity, class Proj2 = identity,
           IndirectStrictWeakOrder<projected<I1, Proj1>, projected<I2, Proj2>> Comp =
             ranges::less>
    constexpr bool includes(I1 first1, S1 last1, I2 first2, S2 last2, Comp comp = {},
                            Proj1 proj1 = {}, 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 = {},
                            Proj1 proj1 = {}, 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 25.3.5
              ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator2 last2,
              ForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  ForwardIterator
    set_union(ExecutionPolicy&& exec, // see 25.3.5
              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 = {},
                Proj1 proj1 = {}, 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 = {},
                Proj1 proj1 = {}, 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 25.3.5
                     ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2, ForwardIterator2 last2,
                     ForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  ForwardIterator
    set_intersection(ExecutionPolicy&& exec, // see 25.3.5
                     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 = {}, Proj1 proj1 = {}, 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 = {}, Proj1 proj1 = {}, 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 25.3.5
                   ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2, ForwardIterator2 last2,
                   ForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  ForwardIterator
    set_difference(ExecutionPolicy&& exec, // see 25.3.5
                   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 = {}, Proj1 proj1 = {}, 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 = {}, Proj1 proj1 = {}, 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 25.3.5
                             ForwardIterator1 first1, ForwardIterator1 last1,
                             ForwardIterator2 first2, ForwardIterator2 last2,
                             ForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  ForwardIterator
    set_symmetric_difference(ExecutionPolicy&& exec, // see 25.3.5
                             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 = {}, Proj1 proj1 = {},
                               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 = {},
                               Proj1 proj1 = {}, Proj2 proj2 = {});
}
```

```
// 25.7.7, 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 = {}, 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 = {}, 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 = {}, 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 = {}, 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 = {}, 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 = {}, 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 = {}, 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 = {}, 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 25.3.5
               RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  bool is_heap(ExecutionPolicy&& exec, // see 25.3.5
               RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);

namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
           IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less>
    constexpr bool is_heap(I first, S last, Comp comp = {}, 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 = {}, 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 25.3.5
                  RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  RandomAccessIterator
    is_heap_until(ExecutionPolicy&& exec, // see 25.3.5
                  RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp);

namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
           IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less>
    constexpr I is_heap_until(I first, S last, Comp comp = {}, 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 = {}, Proj proj = {});
}

// 25.7.8, minimum and maximum
template<class T> constexpr const T& min(const T& a, const T& b);
```

```
template<class T, class Compare>
  constexpr const T& min(const T& a, const T& b, Compare comp);
template<class T>
  constexpr T min(initializer_list<T> t);
template<class T, class Compare>
  constexpr T min(initializer_list<T> t, Compare comp);

namespace ranges {
  template<class T, class Proj = identity,
           IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less>
    constexpr const T& min(const T& a, const T& b, Comp comp = {}, 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 = {}, Proj proj = {});
  template<InputRange R, class Proj = identity,
           IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
    requires IndirectlyCopyableStorable<iterator_t<R>, iter_value_t<iterator_t<R>>*>
    constexpr iter_value_t<iterator_t<R>>
      min(R&& r, Comp comp = {}, Proj proj = {});
}

template<class T> constexpr const T& max(const T& a, const T& b);
template<class T, class Compare>
  constexpr const T& max(const T& a, const T& b, Compare comp);
template<class T>
  constexpr T max(initializer_list<T> t);
template<class T, class Compare>
  constexpr T max(initializer_list<T> t, Compare comp);

namespace ranges {
  template<class T, class Proj = identity,
           IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less>
    constexpr const T& max(const T& a, const T& b, Comp comp = {}, 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 = {}, Proj proj = {});
  template<InputRange R, class Proj = identity,
           IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
    requires IndirectlyCopyableStorable<iterator_t<R>, iter_value_t<iterator_t<R>>*>
    constexpr iter_value_t<iterator_t<R>>
      max(R&& r, Comp comp = {}, Proj proj = {});
}

template<class T> constexpr pair<const T&, const T&> minmax(const T& a, const T& b);
template<class T, class Compare>
  constexpr pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
template<class T>
  constexpr pair<T, T> minmax(initializer_list<T> t);
template<class T, class Compare>
  constexpr pair<T, T> minmax(initializer_list<T> t, Compare comp);

namespace ranges {
  template<class T>
  struct minmax_result {
    [[no_unique_address]] T min;
    [[no_unique_address]] T max;

    template<class T2>
      requires ConvertibleTo<const T&, T2>
      operator minmax_result<T2>() const & {
        return {min, max};
      }
```

```
    template<class T2>
      requires ConvertibleTo<T, T2>
      operator minmax_result<T2>() && {
        return {std::move(min), std::move(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 = {}, 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 = {}, Proj proj = {});
  template<InputRange R, class Proj = identity,
           IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
    requires IndirectlyCopyableStorable<iterator_t<R>, iter_value_t<iterator_t<R>>*>
    constexpr minmax_result<iter_value_t<iterator_t<R>>>
      minmax(R&& r, Comp comp = {}, 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 25.3.5
                              ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  ForwardIterator min_element(ExecutionPolicy&& exec, // see 25.3.5
                              ForwardIterator first, ForwardIterator last,
                              Compare comp);

namespace ranges {
  template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
           IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less>
    constexpr I min_element(I first, S last, Comp comp = {}, 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 = {}, 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 25.3.5
                              ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  ForwardIterator max_element(ExecutionPolicy&& exec, // see 25.3.5
                              ForwardIterator first, ForwardIterator last,
                              Compare comp);

namespace ranges {
  template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
           IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less>
    constexpr I max_element(I first, S last, Comp comp = {}, 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 = {}, 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 25.3.5
                   ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  pair<ForwardIterator, ForwardIterator>
    minmax_element(ExecutionPolicy&& exec, // see 25.3.5
                   ForwardIterator first, ForwardIterator last, Compare comp);

namespace ranges {
  template<class I>
  using minmax_element_result = minmax_result<I>;

  template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
           IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less>
    constexpr minmax_element_result<I>
      minmax_element(I first, S last, Comp comp = {}, Proj proj = {});
  template<ForwardRange R, class Proj = identity,
           IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
    constexpr minmax_element_result<safe_iterator_t<R>>
      minmax_element(R&& r, Comp comp = {}, Proj proj = {});
}

// 25.7.9, bounded value
template<class T>
  constexpr const T& clamp(const T& v, const T& lo, const T& hi);
template<class T, class Compare>
  constexpr const T& clamp(const T& v, const T& lo, const T& hi, Compare comp);

// 25.7.10, 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 25.3.5
                            ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Compare>
  bool
    lexicographical_compare(ExecutionPolicy&& exec, // see 25.3.5
                            ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2,
                            Compare comp);
```

```
namespace ranges {
  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
           class Proj1 = identity, class Proj2 = identity,
           IndirectStrictWeakOrder<projected<I1, Proj1>, projected<I2, Proj2>> Comp =
             ranges::less>
    constexpr bool
      lexicographical_compare(I1 first1, S1 last1, I2 first2, S2 last2,
                              Comp comp = {}, Proj1 proj1 = {}, 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 = {},
                              Proj1 proj1 = {}, Proj2 proj2 = {});
}

// 25.7.11, three-way comparison algorithms
template<class T, class U>
  constexpr auto compare_3way(const T& a, const U& b);
template<class InputIterator1, class InputIterator2, class Cmp>
  constexpr auto
    lexicographical_compare_3way(InputIterator1 b1, InputIterator1 e1,
                                 InputIterator2 b2, InputIterator2 e2,
                                 Cmp comp)
      -> common_comparison_category_t<decltype(comp(*b1, *b2)), strong_ordering>;
template<class InputIterator1, class InputIterator2>
  constexpr auto
    lexicographical_compare_3way(InputIterator1 b1, InputIterator1 e1,
                                 InputIterator2 b2, InputIterator2 e2);

// 25.7.12, 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 = {}, 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 = {}, 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 = {}, 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 = {}, Proj proj = {});
  }
}
```

25.5 Non-modifying sequence operations [alg.nonmodifying]

25.5.1 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);

template<InputIterator I, Sentinel<I> S, class Proj = identity,
         IndirectUnaryPredicate<projected<I, Proj>> Pred>
  constexpr bool ranges::all_of(I first, S last, Pred pred, Proj proj = {});
template<InputRange R, class Proj = identity,
         IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
  constexpr bool ranges::all_of(R&& r, Pred pred, 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 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.

25.5.2 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);

template<InputIterator I, Sentinel<I> S, class Proj = identity,
         IndirectUnaryPredicate<projected<I, Proj>> Pred>
  constexpr bool ranges::any_of(I first, S last, Pred pred, Proj proj = {});
template<InputRange R, class Proj = identity,
         IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
  constexpr bool ranges::any_of(R&& r, Pred pred, 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 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.

25.5.3 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);

template<InputIterator I, Sentinel<I> S, class Proj = identity,
         IndirectUnaryPredicate<projected<I, Proj>> Pred>
  constexpr bool ranges::none_of(I first, S last, Pred pred, Proj proj = {});
```

```
template<InputRange R, class Proj = identity,
         IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
  constexpr bool ranges::none_of(R&& r, Pred pred, 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 E 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.

25.5.4 For each [alg.foreach]

```
template<class InputIterator, class Function>
  constexpr Function for_each(InputIterator first, InputIterator last, Function f);
```

1 ~~*Requires:*~~ *Expects:* `Function` ~~shall satisfy~~meets the *Cpp17MoveConstructible* requirements (Table **??**). [*Note*: `Function` need not meet the requirements of *Cpp17CopyConstructible* (Table **??**). — *end note*]

2 *Effects:* Applies `f` to the result of dereferencing every iterator in the range `[first, last)`, starting from `first` and proceeding to `last - 1`. [*Note*: If the type of `first` satisfies the requirements of a mutable iterator, `f` may apply non-constant functions through the dereferenced iterator. — *end note*]

3 *Returns:* `f`.

4 *Complexity:* Applies `f` exactly `last - first` times.

5 *Remarks:* If `f` returns a result, the result is ignored.

```
template<class ExecutionPolicy, class ForwardIterator, class Function>
  void for_each(ExecutionPolicy&& exec,
                ForwardIterator first, ForwardIterator last,
                Function f);
```

6 ~~*Requires:*~~ *Expects:* `Function` ~~shall satisfy~~meets the *Cpp17CopyConstructible* requirements.

7 *Effects:* Applies `f` to the result of dereferencing every iterator in the range `[first, last)`. [*Note*: If the type of `first` satisfies the requirements of a mutable iterator, `f` may apply non-constant functions through the dereferenced iterator. — *end note*]

8 *Complexity:* Applies `f` exactly `last - first` times.

9 *Remarks:* If `f` returns a result, the result is ignored. Implementations do not have the freedom granted under 25.3.3 to make arbitrary copies of elements from the input sequence.

10 [*Note*: Does not return a copy of its `Function` parameter, since parallelization may not permit efficient state accumulation. — *end note*]

```
template<InputIterator I, Sentinel<I> S, class Proj = identity,
         IndirectUnaryInvocable<projected<I, Proj>> Fun>
  constexpr ranges::for_each_result<I, Fun>
    ranges::for_each(I first, S last, Fun f, Proj proj = {});
template<InputRange R, class Proj = identity,
         IndirectUnaryInvocable<projected<iterator_t<R>, Proj>> Fun>
  constexpr ranges::for_each_result<safe_iterator_t<R>, Fun>
    ranges::for_each(R&& r, Fun f, 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 non-constant functions. — *end note*]

12 *Returns:* `{last, std::move(f)}`.

13 *Complexity:* Applies `f` and `proj` exactly `last - first` times.

14 *Remarks:* If `f` returns a result, the result is ignored.

15 [*Note*: The overloads in namespace `ranges` require `Fun` to model `CopyConstructible`. — *end note*]

```
template<class InputIterator, class Size, class Function>
  constexpr InputIterator for_each_n(InputIterator first, Size n, Function f);
```

16 Let N be $\max(0, \texttt{n})$.

17 *Mandates:* The type `Size` is convertible to an integral type (??, ??).

18 ~~*Requires:*~~ *Expects:* `Function` ~~shall satisfy~~meets the *Cpp17MoveConstructible* requirements [*Note*: `Function` need not meet the requirements of *Cpp17CopyConstructible*. — *end note*]

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

20 *Effects:* Applies `f` to the result of dereferencing every iterator in the range `[first, first +` ~~`n`~~*N*`)` in order. [*Note*: If the type of `first` satisfies the requirements of a mutable iterator, `f` may apply non-constant functions through the dereferenced iterator. — *end note*]

21 *Returns:* `first +` ~~`n`~~*N*.

22 *Remarks:* If `f` returns a result, the result is ignored.

```
template<class ExecutionPolicy, class ForwardIterator, class Size, class Function>
  ForwardIterator for_each_n(ExecutionPolicy&& exec, ForwardIterator first, Size n,
                             Function f);
```

23 Let N be $\max(0, \texttt{n})$.

24 *Mandates:* The type `Size` is convertible to an integral type (??, ??).

25 ~~*Requires:*~~ *Expects:* `Function` ~~shall satisfy~~meets the *Cpp17CopyConstructible* requirements.

26 *Requires:* `n >= 0`.

27 *Effects:* Applies `f` to the result of dereferencing every iterator in the range `[first, first +` ~~`n`~~*N*`)`. [*Note*: If the type of `first` satisfies the requirements of a mutable iterator, `f` may apply non-constant functions through the dereferenced iterator. — *end note*]

28 *Returns:* `first +` ~~`n`~~*N*

29 *Remarks:* If `f` returns a result, the result is ignored. Implementations do not have the freedom granted under 25.3.3 to make arbitrary copies of elements from the input sequence.

25.5.5 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);

template<InputIterator I, Sentinel<I> S, class T, class Proj = identity>
  requires IndirectRelation<ranges::equal_to, projected<I, Proj>, const T*>
  constexpr I ranges::find(I first, S last, const T& value, 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>
    ranges::find(R&& r, const T& value, Proj proj = {});
template<InputIterator I, Sentinel<I> S, class Proj = identity,
         IndirectUnaryPredicate<projected<I, Proj>> Pred>
  constexpr I ranges::find_if(I first, S last, Pred pred, Proj proj = {});
```

```
template<InputRange R, class Proj = identity,
         IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
  constexpr safe_iterator_t<R>
    ranges::find_if(R&& r, Pred pred, Proj proj = {});
template<InputIterator I, Sentinel<I> S, class Proj = identity,
         IndirectUnaryPredicate<projected<I, Proj>> Pred>
  constexpr I ranges::find_if_not(I first, S last, Pred pred, Proj proj = {});
template<InputRange R, class Proj = identity,
         IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
  constexpr safe_iterator_t<R>
    ranges::find_if_not(R&& r, Pred pred, 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`. Returns `last` if no such iterator is found.

3 *Complexity:* At most `last - first` applications of the corresponding predicate and any projection.

25.5.6 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);

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>
    ranges::find_end(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = {},
             Proj1 proj1 = {}, 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>
    ranges::find_end(R1&& r1, R2&& r2, Pred pred = {},
```

```
        Proj1 proj1 = {}, Proj2 proj2 = {});
```

1 Let:

(1.1) — `pred` be `equal_to{}` for the overloads with no parameter `pred`.

(1.2) — *E* 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 `last1` if `[first2, last2)` is empty, or if `(last2 - first2) > (last1 - first1)` is `true`, or if there is no iterator in the range `[first1, last1 - (last2 - first2))` such that for every non-negative integer `n < (last2 - first2)`, *E* is `true`. Otherwise `i` is the last such iterator in `[first1, last1 - (last2 - first2))`.

2 *Returns:*

(2.1) — `i` for the overloads in namespace `std`, and

(2.2) — `{i, i + (i == last1 ? 0 : last2 - first2)}` for the overloads in namespace `ranges`.

3 *Complexity:* At most `(last2 - first2) * (last1 - first1 - (last2 - first2) + 1)` applications of the corresponding predicate and any projections.

25.5.7 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);

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 ranges::find_first_of(I1 first1, S1 last1, I2 first2, S2 last2,
                                     Pred pred = {},
                                     Proj1 proj1 = {}, 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>
    ranges::find_first_of(R1&& r1, R2&& r2,
                          Pred pred = {},
```

```
                    Proj1 proj1 = {}, 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 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. 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.

25.5.8 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);

template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
         IndirectRelation<projected<I, Proj>> Pred = ranges::equal_to>
  constexpr I ranges::adjacent_find(I first, S last, Pred pred = {}, Proj proj = {});
template<ForwardRange R, class Proj = identity,
         IndirectRelation<projected<iterator_t<R>, Proj>> Pred = ranges::equal_to>
  constexpr safe_iterator_t<R> ranges::adjacent_find(R&& r, Pred pred = {}, 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. Returns `last` if no such iterator is found.

3 *Complexity:* For the overloads with no `ExecutionPolicy`, exactly

$$\min((\texttt{i - first}) + 1, (\texttt{last - first}) - 1)$$

applications of the corresponding predicate, where `i` is `adjacent_find`'s return value. For the overloads with an `ExecutionPolicy`, $\mathcal{O}$(`last - first`) applications of the corresponding predicate, and no more than twice as many applications of any projection.

25.5.9 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);

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>
    ranges::count(I first, S last, const T& value, 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>>
    ranges::count(R&& r, const T& value, Proj proj = {});
template<InputIterator I, Sentinel<I> S, class Proj = identity,
         IndirectUnaryPredicate<projected<I, Proj>> Pred>
  constexpr iter_difference_t<I>
    ranges::count_if(I first, S last, Pred pred, Proj proj = {});
template<InputRange R, class Proj = identity,
         IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
  constexpr iter_difference_t<iterator_t<R>>
    ranges::count_if(R&& r, Pred pred, 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 with both parameters `proj` and `pred`.

2 *Effects:* Returns the number of iterators `i` in the range `[first, last)` for which E holds.

3 *Complexity:* Exactly `last - first` applications of the corresponding predicate and any projection.

25.5.10 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);

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 ranges::mismatch_result<I1, I2>
    ranges::mismatch(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = {},
                     Proj1 proj1 = {}, 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 ranges::mismatch_result<safe_iterator_t<R1>, safe_iterator_t<R2>>
    ranges::mismatch(R1&& r1, R2&& r2, Pred pred = {},
                     Proj1 proj1 = {}, Proj2 proj2 = {});
```

1 Let `last2` be `first2 + (last1 - first1)` for the overloads with no parameter `last2` or `r2`.

2 Let E be:

(2.1) — `!(*(first1 + n) == *(first2 + n))` for the overloads with no parameter `pred`,

(2.2) — `pred(*(first1 + n), *(first2 + n)) == false` for the overloads with a parameter `pred` and no parameter `proj1`,

(2.3) — `!invoke(pred, invoke(proj1, *(first1 + n)), invoke(proj2, *(first2 + n)))` for the overloads with both parameters `pred` and `proj1`.

3 Let N be min(`last1 - first1`, `last2 - first2`).

4 *Returns:* `{ first1 + n, first2 + n }`, where `n` is the smallest integer in `[0,` N`)` such that E holds, or N if no such integer exists.

5 *Complexity:* At most N applications of the corresponding predicate and any projections.

25.5.11 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);

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 ranges::equal(I1 first1, S1 last1, I2 first2, S2 last2,
                               Pred pred = {},
                               Proj1 proj1 = {}, 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 ranges::equal(R1&& r1, R2&& r2, Pred pred = {},
                               Proj1 proj1 = {}, Proj2 proj2 = {});
```

1 Let:

(1.1) — `last2` be `first2 + (last1 - first1)` for the overloads with no parameter `last2` or `r2`,

(1.2) — `pred` be `equal_to{}` for the overloads with no parameter `pred`,

(1.3) — E be:

(1.3.1) — `pred(*i, *(first2 + (i - first1)))` for the overloads with no parameter `proj1`,

(1.3.2) — `invoke(pred, invoke(proj1, *i), invoke(proj2, *(first2 + (i - first1))))` for the overloads with parameter `proj1`.

2 *Returns:* If `last1 - first1 != last2 - first2`, return `false`. Otherwise return `true` if E holds for every iterator `i` in the range `[first1, last1)` Otherwise, returns `false`.

3 *Complexity:* If the types of `first1`, `last1`, `first2`, and `last2`:

(3.1) — meet the *Cpp17RandomAccessIterator* requirements (**??**) for the overloads in namespace `std`, or

(3.2) — pairwise model `SizedSentinel` (**??**) for the overloads in namespace `ranges`,

and `last1 - first1 != last2 - first2`, then no applications of the corresponding predicate and each projection; otherwise,

(3.3) — For the overloads with no `ExecutionPolicy`, at most min(`last1 - first1`, `last2 - first2`) applications of the corresponding predicate and any projections.

(3.4) — For the overloads with an `ExecutionPolicy`, $\mathcal{O}$(min(`last1 - first1`, `last2 - first2`)) applications of the corresponding predicate.

25.5.12 Is permutation [alg.is.permutation]

```
template<class ForwardIterator1, class ForwardIterator2>
  constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                                ForwardIterator2 first2);
template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                                ForwardIterator2 first2, BinaryPredicate pred);
template<class ForwardIterator1, class ForwardIterator2>
  constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                                ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                                ForwardIterator2 first2, ForwardIterator2 last2,
                                BinaryPredicate pred);
```

1 *Requires:* `ForwardIterator1` and `ForwardIterator2` shall have the same value type. The comparison function shall be an equivalence relation.

2 *Mandates:* `ForwardIterator1` and `ForwardIterator2` have the same value type.

3 *Expects:* The comparison function is an equivalence relation.

4 *Remarks:* If `last2` was not given in the argument list, it denotes `first2 + (last1 - first1)` below.

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 `ForwardIterator2 begin`, such that `equal(first1, last1, begin)` returns `true` or `equal(first1, last1, begin, pred)` returns `true`; otherwise, returns `false`.

6 *Complexity:* No applications of the corresponding predicate if `ForwardIterator1` and `ForwardIterator2` meet the requirements of random access iterators and `last1 - first1 != last2 - first2`. Otherwise, exactly `last1 - first1` applications of the corresponding predicate if `equal(first1, last1, first2, last2)` would return `true` if `pred` was not given in the argument list or `equal(first1, last1, first2, last2, pred)` would return `true` if `pred` was given in the argument list; otherwise, at worst $\mathcal{O}(N^2)$, where N has the value `last1 - first1`.

```
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 ranges::is_permutation(I1 first1, S1 last1, I2 first2, S2 last2,
                                        Pred pred = {},
                                        Proj1 proj1 = {}, 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 ranges::is_permutation(R1&& r1, R2&& r2, Pred pred = {},
                                        Proj1 proj1 = {}, Proj2 proj2 = {});
```

7 *Returns:* If `last1 - first1 != last2 - first2`, return `false`. Otherwise return `true` if there exists a permutation of the elements in the range `[first2, last2)`, bounded by `[pfirst, plast)`, such that `ranges::equal(first1, last1, pfirst, plast, pred, proj1, proj2)` returns `true`; otherwise, returns `false`.

8 *Complexity:* No applications of the corresponding predicate and projections if:

(8.1) — `S1` and `I1` model `SizedSentinel`,

(8.2) — `S2` and `I2` model `SizedSentinel`, and

(8.3) — `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`.

25.5.13 Search [alg.search]

```
template<class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator1
    search(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    search(ExecutionPolicy&& exec,
           ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  constexpr ForwardIterator1
    search(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2,
           BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  ForwardIterator1
    search(ExecutionPolicy&& exec,
           ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2,
           BinaryPredicate pred);
```

1 *Returns:* The first iterator `i` in the range `[first1, last1 - (last2-first2))` such that for every non-negative integer `n` less than `last2 - first2` the following corresponding conditions hold: `*(i + n) == *(first2 + n), pred(*(i + n), *(first2 + n)) != false`. Returns `first1` if `[first2, last2)` is empty, otherwise returns `last1` if no such iterator is found.

2 *Complexity:* At most `(last1 - first1) * (last2 - first2)` applications of the corresponding predicate.

```
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>
    ranges::search(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = {},
                   Proj1 proj1 = {}, 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>
    ranges::search(R1&& r1, R2&& r2, Pred pred = {},
                   Proj1 proj1 = {}, Proj2 proj2 = {});
```

3 *Returns:*

(3.1) — `{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 condition

```
bool(invoke(pred, invoke(proj1, *(i + n)), invoke(proj2, *(first2 + n))))
```

is `true`:

(3.2) — Returns `{last1, last1}` if no such iterator exists.

4 *Complexity:* At most `(last1 - first1) * (last2 - first2)` applications of the corresponding predicate and projections.

```
template<class ForwardIterator, class Size, class T>
  constexpr ForwardIterator
    search_n(ForwardIterator first, ForwardIterator last,
             Size count, const T& value);
template<class ExecutionPolicy, class ForwardIterator, class Size, class T>
  ForwardIterator
    search_n(ExecutionPolicy&& exec,
             ForwardIterator first, ForwardIterator last,
             Size count, const T& value);

template<class ForwardIterator, class Size, class T,
         class BinaryPredicate>
  constexpr ForwardIterator
    search_n(ForwardIterator first, ForwardIterator last,
             Size count, const T& value,
             BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Size, class T,
         class BinaryPredicate>
  ForwardIterator
    search_n(ExecutionPolicy&& exec,
             ForwardIterator first, ForwardIterator last,
             Size count, const T& value,
             BinaryPredicate pred);
```

5 Let N be $\max(0, \texttt{count})$.

6 ~~*Requires:*~~ *Mandates:* The type `Size` ~~shall be~~is convertible to integral type (**??**, **??**).

7 *Returns:* The first iterator `i` in the range `[first, last-`~~`count`~~N`)` such that for every non-negative integer `n` less than ~~`count`~~N the following corresponding conditions hold: `*(i + n) == value, pred(*(i + n),value) != false`. Returns `last` if no such iterator is found.

8 *Complexity:* At most `last - first` applications of the corresponding predicate.

```
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>
    ranges::search_n(I first, S last, iter_difference_t<I> count,
                     const T& value, Pred pred = {}, 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>
    ranges::search_n(R&& r, iter_difference_t<iterator_t<R>> count,
                     const T& value, Pred pred = {}, Proj proj = {});
```

9 *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.

10 *Complexity:* At most `last - first` applications of the corresponding predicate and projection.

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

11 *Effects:* Equivalent to: `return searcher(first, last).first;`

12 *Remarks:* `Searcher` need not meet the *Cpp17CopyConstructible* requirements.

25.6 Mutating sequence operations [alg.modifying.operations]

25.6.1 Copy [alg.copy]

```
template<class InputIterator, class OutputIterator>
  constexpr OutputIterator copy(InputIterator first, InputIterator last,
                                OutputIterator result);
```

```
template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  constexpr ranges::copy_result<I, O> ranges::copy(I first, S last, O result);
template<InputRange R, WeaklyIncrementable O>
  requires IndirectlyCopyable<iterator_t<R>, O>
  constexpr ranges::copy_result<safe_iterator_t<R>, O> ranges::copy(R&& r, O result);
```

1 Let N be `last - first`.

2 ~~*Requires:*~~ *Expects:* `result` ~~shall~~is not be in the range `[first, last)`.

3 *Effects:* Copies elements in the range `[first, last)` into the range `[result, result + N)` starting from `first` and proceeding to `last`. For each non-negative integer $n < N$, performs `*(result + n) = *(first + n)`.

4 *Returns:*

(4.1) — `result + N` for the overload in namespace `std`, or

(4.2) — `{last, result + N}` for the overloads in namespace `ranges`.

5 *Complexity:* Exactly N assignments.

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 copy(ExecutionPolicy&& policy,
                        ForwardIterator1 first, ForwardIterator1 last,
                        ForwardIterator2 result);
```

6 ~~*Requires:*~~ *Expects:* The ranges `[first, last)` and `[result, result + (last - first))` ~~shall~~do not overlap.

7 *Effects:* Copies elements in the range `[first, last)` into the range `[result, result + (last - first))`. For each non-negative integer `n < (last - first)`, performs `*(result + n) = *(first + n)`.

8 *Returns:* `result + (last - first)`.

9 *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);

template<InputIterator I, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  constexpr ranges::copy_n_result<I, O>
    ranges::copy_n(I first, iter_difference_t<I> n, O result);
```

10 Let ~~*M* be max(n, 0)~~*N* be $\max(0, \texttt{n})$.

11 *Mandates:* The type `Size` is convertible to an integral type (??, ??).

12 *Effects:* For each non-negative integer ~~$i < M$~~$i < N$, performs `*(result + i) = *(first + i)`.

13 *Returns:*

(13.1) — `result +` ~~*M*~~*N* for the overloads in namespace `std`, or

(13.2) — `{first +` ~~*M*~~*N*`, result +` ~~*M*~~*N*`}` for the overload in namespace `ranges`.

14 *Complexity:* Exactly ~~*M*~~*N* assignments.

```
template<class InputIterator, class OutputIterator, class Predicate>
  constexpr OutputIterator copy_if(InputIterator first, InputIterator last,
                                   OutputIterator result, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate>
  ForwardIterator2 copy_if(ExecutionPolicy&& exec,
                           ForwardIterator1 first, ForwardIterator1 last,
```

```
                        ForwardIterator2 result, Predicate pred);

template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class Proj = identity,
         IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires IndirectlyCopyable<I, O>
  constexpr ranges::copy_if_result<I, O>
    ranges::copy_if(I first, S last, O result, Pred pred, Proj proj = {});
template<InputRange R, WeaklyIncrementable O, class Proj = identity,
         IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<R>, O>
  constexpr ranges::copy_if_result<safe_iterator_t<R>, O>
    ranges::copy_if(R&& r, O result, Pred pred, Proj proj = {});
```

15 Let E be:

(15.1) — `bool(pred(*i))` for the overloads in namespace `std`, or

(15.2) — `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.

16 ~~*Requires:*~~ *Expects:* The ranges `[first, last)` and `[result, result + (last - first))` ~~shall~~do not overlap. [*Note*: For the overload with an `ExecutionPolicy`, there may be a performance cost if `iterator_traits<ForwardIterator1>::value_type` is not *Cpp17MoveConstructible* (Table **??**). — *end note*]

17 *Effects:* Copies all of the elements referred to by the iterator `i` in the range `[first, last)` for which E is `true`.

18 *Returns:*

(18.1) — `result + N` for the overloads in namespace `std`, or

(18.2) — `{last, result + N}` for the overloads in namespace `ranges`.

19 *Complexity:* Exactly `last - first` applications of the corresponding predicate and any projection.

20 *Remarks:* Stable (**??**).

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
  constexpr BidirectionalIterator2
    copy_backward(BidirectionalIterator1 first,
                  BidirectionalIterator1 last,
                  BidirectionalIterator2 result);

template<BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
  requires IndirectlyCopyable<I1, I2>
  constexpr ranges::copy_backward_result<I1, I2>
    ranges::copy_backward(I1 first, S1 last, I2 result);
template<BidirectionalRange R, BidirectionalIterator I>
  requires IndirectlyCopyable<iterator_t<R>, I>
  constexpr ranges::copy_backward_result<safe_iterator_t<R>, I>
    ranges::copy_backward(R&& r, I result);
```

21 Let N be `last - first`.

22 ~~*Requires:*~~ *Expects:* `result` ~~shall not be~~is not in the range `(first, last]`.

23 *Effects:* Copies elements in the range `[first, last)` into the range `[result - N, result)` starting from `last - 1` and proceeding to `first`.[235] For each positive integer $n \leq N$, performs `*(result - n) = *(last - n)`.

24 *Returns:*

(24.1) — `result - N` for the overload in namespace `std`, or

(24.2) — `{last, result - N}` for the overloads in namespace `ranges`.

25 *Complexity:* Exactly N assignments.

235) `copy_backward` should be used instead of copy when `last` is in the range `[result - N, result)`.

25.6.2 Move [alg.move]

```
template<class InputIterator, class OutputIterator>
  constexpr OutputIterator move(InputIterator first, InputIterator last,
                                OutputIterator result);

template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyMovable<I, O>
  constexpr ranges::move_result<I, O>
    ranges::move(I first, S last, O result);
template<InputRange R, WeaklyIncrementable O>
  requires IndirectlyMovable<iterator_t<R>, O>
  constexpr ranges::move_result<safe_iterator_t<R>, O>
    ranges::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`.

Let N be `last - first`.

2 ~~*Requires:*~~ *Expects:* `result` ~~shall not be~~is not in the range `[first, last)`.

3 *Effects:* Moves elements in the range `[first, last)` into the range `[result, result + N)` starting from `first` and proceeding to `last`. For each non-negative integer $n < N$, performs `*(result + n) = E`.

4 *Returns:*

(4.1) — `result + N` for the overload in namespace `std`, or

(4.2) — `{last, result + N}` for the overloads in namespace `ranges`.

5 *Complexity:* Exactly N assignments.

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 move(ExecutionPolicy&& policy,
                        ForwardIterator1 first, ForwardIterator1 last,
                        ForwardIterator2 result);
```

6 Let N be `last - first`.

7 ~~*Requires:*~~ *Expects:* The ranges `[first, last)` and `[result, result + N)` ~~shall~~do not overlap.

8 *Effects:* Moves elements in the range `[first, last)` into the range `[result, result + N)`. For each non-negative integer $n < N$, performs `*(result + n) = std::move(*(first + n))`.

9 *Returns:* `result + N`.

10 *Complexity:* Exactly N assignments.

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
  constexpr BidirectionalIterator2
    move_backward(BidirectionalIterator1 first, BidirectionalIterator1 last,
                  BidirectionalIterator2 result);

template<BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
  requires IndirectlyMovable<I1, I2>
  constexpr ranges::move_backward_result<I1, I2>
    ranges::move_backward(I1 first, S1 last, I2 result);
template<BidirectionalRange R, BidirectionalIterator I>
  requires IndirectlyMovable<iterator_t<R>, I>
  constexpr ranges::move_backward_result<safe_iterator_t<R>, I>
    ranges::move_backward(R&& r, I result);
```

11 Let E be

(11.1) — `std::move(*(last - n))` for the overload in namespace `std`, or

(11.2) — `ranges::iter_move(last - n)` for the overloads in namespace `ranges`.

Let N be `last - first`.

12 ~~*Requires:*~~ *Expects:* `result` ~~shall not be~~is not in the range `(first, last]`.

13 *Effects:* Moves elements in the range `[first, last)` into the range `[result - N, result)` starting from `last - 1` and proceeding to `first`.[236] For each positive integer $n \leq N$, performs `*(result - n) = E`.

14 *Returns:*

(14.1) — `result - N` for the overload in namespace `std`, or

(14.2) — `{last, result - N}` for the overloads in namespace `ranges`.

15 *Complexity:* Exactly N assignments.

25.6.3 Swap [alg.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,
                ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2);

template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2>
  requires IndirectlySwappable<I1, I2>
  constexpr ranges::swap_ranges_result<I1, I2>
    ranges::swap_ranges(I1 first1, S1 last1, I2 first2, S2 last2);
template<InputRange R1, InputRange R2>
  requires IndirectlySwappable<iterator_t<R1>, iterator_t<R2>>
  constexpr ranges::swap_ranges_result<safe_iterator_t<R1>, safe_iterator_t<R2>>
    ranges::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:*~~ *Expects:* The two ranges `[first1, last1)` and `[first2, last2)` ~~shall~~do not overlap. For the overloads in namespace `std`, `*(first1 + n)` ~~shall be~~is swappable with (??) `*(first2 + n)`.

3 *Effects:* For each non-negative integer $n < M$ performs:

(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:*

(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 M swaps.

```
template<class ForwardIterator1, class ForwardIterator2>
  constexpr void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

6 ~~*Requires:*~~ *Expects:* `a` and `b` ~~shall be~~is dereferenceable. `*a` ~~shall be~~is swappable with (??) `*b`.

7 *Effects:* As if by `swap(*a, *b)`.

25.6.4 Transform [alg.transform]

```
template<class InputIterator, class OutputIterator,
         class UnaryOperation>
  constexpr OutputIterator
```

236) `move_backward` should be used instead of move when `last` is in the range `[result - N, result)`.

```
    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);

template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
         CopyConstructible F, class Proj = identity>
  requires Writable<O, indirect_result_t<F&, projected<I, Proj>>>
  constexpr ranges::unary_transform_result<I, O>
    ranges::transform(I first1, S last1, O result, F op, 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 ranges::unary_transform_result<safe_iterator_t<R>, O>
    ranges::transform(R&& r, O result, F op, 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 ranges::binary_transform_result<I1, I2, O>
    ranges::transform(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                      F binary_op, Proj1 proj1 = {}, 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 ranges::binary_transform_result<safe_iterator_t<R1>, safe_iterator_t<R2>, O>
    ranges::transform(R1&& r1, R2&& r2, O result,
                      F binary_op, Proj1 proj1 = {}, 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:*~~ *Expects:* `op` and `binary_op` ~~shall~~do not invalidate iterators or subranges, nor modify elements in the ranges

(2.1) — `[first1, first1 + N]`,

(2.2) — `[first2, first2 + N]`, and

(2.3) — `[result, result + N]`.[237]

3 *Effects:* Assigns through every iterator `i` in the range `[result, result + N)` a new corresponding value equal to E.

4 *Returns:*

(4.1) — `result + 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 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 `first1` or `first2`.

25.6.5 Replace [alg.replace]

```
template<class ForwardIterator, class T>
  constexpr void replace(ForwardIterator first, ForwardIterator last,
                         const T& old_value, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  void replace(ExecutionPolicy&& exec,
               ForwardIterator first, ForwardIterator last,
               const T& old_value, const T& new_value);

template<class ForwardIterator, class Predicate, class T>
  constexpr void replace_if(ForwardIterator first, ForwardIterator last,
                            Predicate pred, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator, class Predicate, class T>
  void replace_if(ExecutionPolicy&& exec,
                  ForwardIterator first, ForwardIterator last,
                  Predicate pred, const T& new_value);

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
    ranges::replace(I first, S last, const T1& old_value, const T2& new_value, 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>
    ranges::replace(R&& r, const T1& old_value, const T2& new_value, 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 ranges::replace_if(I first, S last, Pred pred, const T& new_value, 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>
    ranges::replace_if(R&& r, Pred pred, const T& new_value, Proj proj = {});
```

1 Let E be

(1.1) — `bool(*i == old_value)` for `replace`,

237) The use of fully closed ranges is intentional.

(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:*~~ *Expects:* The expression `*first = new_value` ~~shall be~~is valid.

3 *Effects:* Substitutes elements referred by the iterator `i` in the range `[first, last)` with `new_value`, when E is `true`.

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);

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 ranges::replace_copy_result<I, O>
    ranges::replace_copy(I first, S last, O result, const T1& old_value, const T2& new_value,
                         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 ranges::replace_copy_result<safe_iterator_t<R>, O>
    ranges::replace_copy(R&& r, O result, const T1& old_value, const T2& new_value,
                         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 ranges::replace_copy_if_result<I, O>
    ranges::replace_copy_if(I first, S last, O result, Pred pred, const T& new_value,
                            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 ranges::replace_copy_if_result<safe_iterator_t<R>, O>
    ranges::replace_copy_if(R&& r, O result, Pred pred, const T& new_value,
```

```
                    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:*~~ *Expects:* The results of the expressions `*first` and `new_value` ~~shall be~~are writable (??) to the `result` output iterator. The ranges `[first, last)` and `[result, result + (last - first))` ~~shall~~do 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.

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.

25.6.6 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);


template<class T, OutputIterator<const T&> O, Sentinel<O> S>
  constexpr O ranges::fill(O first, S last, const T& value);
template<class T, OutputRange<const T&> R>
  constexpr safe_iterator_t<R> ranges::fill(R&& r, const T& value);
template<class T, OutputIterator<const T&> O>
  constexpr O ranges::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 (??) to the output iterator. The type `Size` shall be convertible to an integral type (??, ??).

3 *Mandates:* The type `Size` is convertible to an integral type (??, ??).

4 *Expects:* The expression `value` is writable (??) to the output iterator.

5 *Effects:* Assigns `value` through all the iterators in the range `[first, first + N)`.

6 *Returns:* `first + N`.

7 *Complexity:* Exactly N assignments.

25.6.7 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);

template<Iterator O, Sentinel<O> S, CopyConstructible F>
  requires Invocable<F&> && Writable<O, invoke_result_t<F&>>
  constexpr O ranges::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> ranges::generate(R&& r, F gen);
template<Iterator O, CopyConstructible F>
  requires Invocable<F&> && Writable<O, invoke_result_t<F&>>
  constexpr O ranges::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:*~~ *Mandates:* `Size` ~~shall be~~is convertible to an integral type (**??**, **??**).

3 *Effects:* Assigns the result of successive evaluations of `gen()` through each iterator in the range `[first, first + N)`.

4 *Returns:* `first + N`.

5 *Complexity:* Exactly N evaluations of `gen()` and assignments.

25.6.8 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);

template<Permutable I, Sentinel<I> S, class T, class Proj = identity>
  requires IndirectRelation<ranges::equal_to, projected<I, Proj>, const T*>
  constexpr I ranges::remove(I first, S last, const T& value, 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>
    ranges::remove(R&& r, const T& value, Proj proj = {});
template<Permutable I, Sentinel<I> S, class Proj = identity,
         IndirectUnaryPredicate<projected<I, Proj>> Pred>
  constexpr I ranges::remove_if(I first, S last, Pred pred, 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>
```

`ranges::remove_if(R&& r, Pred pred, 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:*~~ *Expects:* For the algorithms in namespace `std`, the type of `*first` ~~shall meet~~meets the *Cpp17MoveAssignable* requirements (Table **??**).

3 *Effects:* Eliminates all the elements referred to by iterator `i` in the range `[first, last)` for which E holds.

4 *Returns:* The end of the resulting range.

5 *Remarks:* 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);

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 ranges::remove_copy_result<I, O>
    ranges::remove_copy(I first, S last, O result, const T& value, 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 ranges::remove_copy_result<safe_iterator_t<R>, O>
    ranges::remove_copy(R&& r, O result, const T& value, Proj proj = {});
template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
         class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires IndirectlyCopyable<I, O>
  constexpr ranges::remove_copy_if_result<I, O>
    ranges::remove_copy_if(I first, S last, O result, Pred pred, Proj proj = {});
template<InputRange R, WeaklyIncrementable O, class Proj = identity,
         IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<R>, O>
```

```
constexpr ranges::remove_copy_if_result<safe_iterator_t<R>, O>
  ranges::remove_copy_if(R&& r, O result, Pred pred, 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:*~~ *Expects:* The ranges `[first, last)` and `[result, result + (last - first))` ~~shall~~do not overlap. The expression `*result = *first` ~~shall be~~is valid. [*Note*: For the overloads with an `ExecutionPolicy`, there may be a performance cost if `iterator_traits<ForwardIterator1>::value_type` does not meet the *Cpp17MoveConstructible* (Table **??**) 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`.

12 *Returns:*

(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 (**??**).

25.6.9 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);

template<Permutable I, Sentinel<I> S, class Proj = identity,
         IndirectRelation<projected<I, Proj>> C = ranges::equal_to>
  constexpr I ranges::unique(I first, S last, C comp = {}, 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>
    ranges::unique(R&& r, C comp = {}, 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:*~~ *Expects:* For the overloads in namepace `std`, `pred` ~~shall~~is be an equivalence relation and the type of `*first` ~~shall meet~~meets the *Cpp17MoveAssignable* requirements (Table **??**).

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`.

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);

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 ranges::unique_copy_result<I, O>
    ranges::unique_copy(I first, S last, O result, C comp = {}, 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 ranges::unique_copy_result<safe_iterator_t<R>, O>
    ranges::unique_copy(R&& r, O result, C comp = {}, 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:*~~ *Expects:*

(7.1) — The ranges `[first, last)` and `[result, result+(last-first))` ~~shall~~do not overlap.

(7.2) — For the overloads in namespace `std`:

(7.2.1) — The comparison function ~~shall be~~is an equivalence relation.

(7.2.2) — The expression `*result = *first` ~~shall be~~is valid.

(7.2.3) — For the overloads with no `ExecutionPolicy`, let `T` be the value type of `InputIterator`. If `InputIterator` meets the *Cpp17ForwardIterator* requirements, then there are no additional requirements for `T`. Otherwise, if `OutputIterator` meets the *Cpp17ForwardIterator* requirements and its value type is the same as `T`, then `T` ~~shall meet~~meets the *Cpp17CopyAssignable* (Table **??**) requirements. Otherwise, `T` ~~shall meet~~meets both the *Cpp17CopyConstructible* (Table **??**) and *Cpp17CopyAssignable* requirements. [*Note*: For the overloads with an `ExecutionPolicy`, there may be a performance cost if the value type of `ForwardIterator1` does not meet both 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.

9 *Returns:*

(9.1) — `result + N` for the overloads in namespace `std`, or

(9.2) — `{last, result + N}` for the overloads in namespace `ranges`

10 *Complexity:* Exactly `last - first - 1` applications of the corresponding predicate and no more than twice as many applications of any projection.

25.6.10 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);

template<BidirectionalIterator I, Sentinel<I> S>
  requires Permutable<I>
  constexpr I ranges::reverse(I first, S last);
template<BidirectionalRange R>
  requires Permutable<iterator_t<R>>
  constexpr safe_iterator_t<R> ranges::reverse(R&& r);
```

1 ~~*Requires:*~~ *Expects:* For the overloads in namespace `std`, `BidirectionalIterator` ~~shall meet~~meets the *Cpp17ValueSwappable* 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);

template<BidirectionalIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  constexpr ranges::reverse_copy_result<I, O>
    ranges::reverse_copy(I first, S last, O result);
template<BidirectionalRange R, WeaklyIncrementable O>
  requires IndirectlyCopyable<iterator_t<R>, O>
  constexpr ranges::reverse_copy_result<safe_iterator_t<R>, O>
    ranges::reverse_copy(R&& r, O result);
```

5 Let N be `last - first`.

6 ~~*Requires:*~~ *Expects:* The ranges `[first, last)` and `[result, result + N)` ~~shall~~do not overlap.

7 *Effects:* Copies the range `[first, last)` to the range `[result, result + N)` such that for every non-negative integer `i < N` the following assignment takes place: `*(result + N - 1 - i) = *(first + i)`.

8 *Returns:*

(8.1) — `result + N` for the overloads in namespace `std`, or

(8.2) — `{last, result + N}` for the overloads in namespace `ranges`.

9 *Complexity:* Exactly N assignments.

25.6.11 Rotate [alg.rotate]

```
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);

template<Permutable I, Sentinel<I> S>
  constexpr subrange<I> ranges::rotate(I first, I middle, S last);
```

1 ~~*Requires:*~~ *Expects:* `[first, middle)` and `[middle, last)` ~~shall be~~are valid ranges. For the overloads in namespace `std`, `ForwardIterator` ~~shall meet~~meets the *Cpp17ValueSwappable* requirements (**??**), and the type of `*first` ~~shall meet~~meets the *Cpp17MoveConstructible* (Table **??**) and *Cpp17MoveAssignable* (Table **??**) 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 overload in namespace `ranges`.

4 *Complexity:* At most `last - first` swaps.

```
template<ForwardRange R>
  requires Permutable<iterator_t<R>>
  constexpr safe_subrange_t<R> ranges::rotate(R&& r, iterator_t<R> middle);
```

5 *Effects:* Equivalent to: `return ranges::rotate(ranges::begin(r), middle, ranges::end(r));`

```
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);

  template<ForwardIterator I, Sentinel<I> S, WeaklyIncrementable O>
    requires IndirectlyCopyable<I, O>
    constexpr ranges::rotate_copy_result<I, O>
      ranges::rotate_copy(I first, I middle, S last, O result);
```

6 Let N be `last - first`.

7 ~~*Requires:*~~ *Expects:* `[first, middle)` and `[middle, last)` ~~shall be~~are valid ranges. The ranges `[first, last)` and `[result, result + N)` ~~shall~~do not overlap.

8 *Effects:* Copies the range `[first, last)` to the range `[result, result + N)` such that for each non-negative integer $i < N$ the following assignment takes place: `*(result + i) = *(first + (i + (middle - first)) % N)`.

9 *Returns:*

(9.1) — `result + N` for the overloads in namespace `std`, or

(9.2) — `{last, result + N}` for the overload in namespace `ranges`.

10 *Complexity:* Exactly N assignments.

```
template<ForwardRange R, WeaklyIncrementable O>
  requires IndirectlyCopyable<iterator_t<R>, O>
  constexpr ranges::rotate_copy_result<safe_iterator_t<R>, O>
    ranges::rotate_copy(R&& r, iterator_t<R> middle, O result);
```

11 *Effects:* Equivalent to:

```
return ranges::rotate_copy(ranges::begin(r), middle, ranges::end(r), result);
```

25.6.12 Sample [alg.random.sample]

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

1 *Mandates:* `Distance` is an integer type.

2 ~~*Requires:*~~ *Expects:*

(2.1) — `PopulationIterator` ~~shall satisfy~~meets the *Cpp17InputIterator* requirements (**??**).

(2.2) — `SampleIterator` ~~shall satisfy~~meets the *Cpp17OutputIterator* requirements (**??**).

(2.3) — `SampleIterator` ~~shall satisfy~~meets the *Cpp17RandomAccessIterator* requirements (**??**) unless `PopulationIterator` satisfies the *Cpp17ForwardIterator* requirements (**??**).

(2.4) — `PopulationIterator`'s value type ~~shall be~~is writable (**??**) to `out`.

(2.5) — `Distance` shall be an integer type.

(2.6) — `remove_reference_t<UniformRandomBitGenerator>` ~~shall satisfy~~meets the requirements of a uniform random bit generator type (**??**) whose return type is convertible to `Distance`.

(2.7) — `out` ~~shall not be~~is not in the range `[first, last)`.

3 *Effects:* Copies min(`last - first`, `n`) elements (the *sample*) from `[first, last)` (the *population*) to `out` such that each possible sample has equal probability of appearance. [*Note*: Algorithms that obtain such effects include *selection sampling* and *reservoir sampling*. — *end note*]

4 *Returns:* The end of the resulting sample range.

5 *Complexity:* $\mathcal{O}$(`last - first`).

6 *Remarks:*

(6.1) — Stable if and only if `PopulationIterator` ~~satisfies~~meets the *Cpp17ForwardIterator* requirements.

(6.2) — To the extent that the implementation of this function makes use of random numbers, the object `g` shall serve as the implementation's source of randomness.

25.6.13 Shuffle [alg.random.shuffle]

```
template<class RandomAccessIterator, class UniformRandomBitGenerator>
  void shuffle(RandomAccessIterator first,
               RandomAccessIterator last,
               UniformRandomBitGenerator&& g);

template<RandomAccessIterator I, Sentinel<I> S, class Gen>
  requires Permutable<I> &&
           UniformRandomBitGenerator<remove_reference_t<Gen>>
  I ranges::shuffle(I first, S last, Gen&& g);
template<RandomAccessRange R, class Gen>
  requires Permutable<iterator_t<R>> &&
           UniformRandomBitGenerator<remove_reference_t<Gen>>
  safe_iterator_t<R> ranges::shuffle(R&& r, Gen&& g);
```

1 ~~*Requires:*~~ *Expects:* For the overload in namespace `std`:

(1.1) — `RandomAccessIterator` ~~shall meet~~meets the *Cpp17ValueSwappable* requirements (**??**).

(1.2) — The type `remove_reference_t<UniformRandomBitGenerator>` ~~shall meet~~meets the uniform random bit generator (**??**) requirements.

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.

25.6.14 Shift [alg.shift]

```
template<class ForwardIterator>
  constexpr ForwardIterator
    shift_left(ForwardIterator first, ForwardIterator last,
               typename iterator_traits<ForwardIterator>::difference_type n);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator
    shift_left(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
               typename iterator_traits<ForwardIterator>::difference_type n);
```

1 ~~*Requires:*~~ *Expects:* The type of `*first` ~~shall satisfy~~meets the *Cpp17MoveAssignable* requirements.

2 *Effects:* If `n <= 0` or `n >= last - first`, does nothing. Otherwise, moves the element from position `first + n + i` into position `first + i` for each non-negative integer `i < (last - first) - n`. In the first overload case, does so in order starting from `i = 0` and proceeding to `i = (last - first) - n - 1`.

3 *Returns:* `first + (last - first - n)` if `n` is positive and `n < last - first`, otherwise `first` if `n` is positive, otherwise `last`.

4 *Complexity:* At most `(last - first) - n` assignments.

```
template<class ForwardIterator>
  constexpr ForwardIterator
    shift_right(ForwardIterator first, ForwardIterator last,
                typename iterator_traits<ForwardIterator>::difference_type n);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator
    shift_right(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
                typename iterator_traits<ForwardIterator>::difference_type n);
```

5 ~~*Requires:*~~ *Expects:* The type of `*first` ~~shall satisfy~~meets the *Cpp17MoveAssignable* requirements. `ForwardIterator` ~~shall meet~~meets the *Cpp17BidirectionalIterator* requirements (**??**) or the *Cpp17ValueSwappable* requirements.

6 *Effects:* If `n <= 0` or `n >= last - first`, does nothing. Otherwise, moves the element from position `first + i` into `position first + n + i` for each non-negative integer `i < (last - first) - n`. In the first overload case, if `ForwardIterator` ~~satisfies~~meets the *Cpp17BidirectionalIterator* requirements, does so in order starting from `i = (last - first) - n - 1` and proceeding to `i = 0`.

7 *Returns:* `first + n` if `n` is positive and `n < last - first`, otherwise `last` if `n` is positive, otherwise `first`.

8 *Complexity:* At most `(last - first) - n` assignments or swaps.

25.7 Sorting and related operations [alg.sorting]

1 The operations in 25.7 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 (**??**) that meets the requirements for a template parameter named `BinaryPredicate` (25.2). The return value of the function call operation applied to an object of type `Compare`, when contextually converted to `bool` (**??**), 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.

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 25.7.3, `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`, and

(4.5) — the induced relation is a strict total ordering.

— *end note*]

5 A sequence is *sorted with respect to a `comp` and `proj`* for 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,

```
bool(invoke(comp, invoke(proj, *(i + n)), invoke(proj, *i)))
```

is `false`.

6 A sequence `[start, finish)` is *partitioned with respect to an expression* `f(e)` if there exists an integer `n` such that for all `0 <= i < (finish - start)`, `f(*(start + i))` is `true` if and only if `i < n`.

7 In the descriptions of the functions that deal with ordering relationships we frequently use a notion of equivalence to describe concepts such as stability. The equivalence to which we refer is not necessarily an `operator==`, but an equivalence relation induced by the strict weak ordering. That is, two elements `a` and `b` are considered equivalent if and only if `!(a < b) && !(b < a)`.

25.7.1 Sorting [alg.sort]

25.7.1.1 `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);

template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less,
         class Proj = identity>
  requires Sortable<I, Comp, Proj>
  constexpr I
    ranges::sort(I first, S last, Comp comp = {}, Proj proj = {});
template<RandomAccessRange R, class Comp = ranges::less, class Proj = identity>
  requires Sortable<iterator_t<R>, Comp, Proj>
  constexpr safe_iterator_t<R>
    ranges::sort(R&& r, Comp comp = {}, Proj proj = {});
```

1 Let `comp` be `less{}` and `proj` be `identity{}` for the overloads with no parameters by those names.

2 ~~*Requires:*~~ *Expects:* For the overloads in namespace `std`, `RandomAccessIterator` ~~shall meet~~meets the *Cpp17ValueSwappable* requirements (**??**) and the type of `*first` ~~shall meet~~meets the *Cpp17MoveConstructible* (Table **??**) and *Cpp17MoveAssignable* (Table **??**) requirements.

3 *Effects:* Sorts the elements in the range `[first, last)` with respect to `comp` and `proj`.

4 *Returns:* `last`, for the overloads in namespace `ranges`.

5 *Complexity:* Let N be `last - first`. $\mathcal{O}(N \log N)$ comparisons and projections.

25.7.1.2 `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);

template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less,
         class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I ranges::stable_sort(I first, S last, Comp comp = {}, Proj proj = {});
template<RandomAccessRange R, class Comp = ranges::less, class Proj = identity>
  requires Sortable<iterator_t<R>, Comp, Proj>
  safe_iterator_t<R>
    ranges::stable_sort(R&& r, Comp comp = {}, Proj proj = {});
```

1 Let `comp` be `less{}` and `proj` be `identity{}` for the overloads with no parameters by those names.

2 ~~*Requires:*~~ *Expects:* For the overloads in namespace `std`, `RandomAccessIterator` ~~shall meet~~meets the *Cpp17ValueSwappable* requirements (**??**) and the type of `*first` ~~shall meet~~meets the *Cpp17MoveConstructible* (Table **??**) and *Cpp17MoveAssignable* (Table **??**) requirements.

3 *Effects:* Sorts the elements in the range `[first, last)` with respect to `comp` and `proj`.

4 *Returns:* `last` for the overloads in namespace `ranges`.

5 *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 projections as the number of comparisons.

6 *Remarks:* Stable (**??**).

25.7.1.3 `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);
```

```
template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less,
         class Proj = identity>
  requires Sortable<I, Comp, Proj>
  constexpr I
    ranges::partial_sort(I first, I middle, S last, Comp comp = {}, Proj proj = {});
```

1 Let `comp` be `less{}` and `proj` be `identity{}` for the overloads with no parameters by those names.

2 ~~*Requires:*~~ *Expects:* `[first, middle)` and `[middle, last)` ~~shall be~~are valid ranges. For the overloads in namespace `std`, `RandomAccessIterator` ~~shall meet~~meets the *Cpp17ValueSwappable* requirements (**??**) and the type of `*first` ~~shall meet~~meets the *Cpp17MoveConstructible* (Table **??**) and *Cpp17MoveAssignable* (Table **??**) requirements.

3 *Effects:* Places the first `middle - first` elements from the range `[first, last)` as sorted with respect to `comp` and `proj` into the range `[first, middle)`. The rest of the elements in the range `[middle, last)` are placed in an unspecified order.

4 *Returns:* `last` for the overload in namespace `ranges`.

5 *Complexity:* Approximately `(last - first) * log(middle - first)` comparisons, and twice as many projections.

```
template<RandomAccessRange R, class Comp = ranges::less, class Proj = identity>
  requires Sortable<iterator_t<R>, Comp, Proj>
  constexpr safe_iterator_t<R>
    ranges::partial_sort(R&& r, iterator_t<R> middle, Comp comp = {}, Proj proj = {});
```

6 *Effects:* Equivalent to:

```
return ranges::partial_sort(ranges::begin(r), middle, ranges::end(r), comp, proj);
```

25.7.1.4 `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);

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
    ranges::partial_sort_copy(I1 first, S1 last, I2 result_first, S2 result_last,
                              Comp comp = {}, Proj1 proj1 = {}, 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>
    ranges::partial_sort_copy(R1&& r, R2&& result_r, Comp comp = {},
                              Proj1 proj1 = {}, Proj2 proj2 = {});
```

1 Let N be min(`last - first`, `result_last - result_first`). Let `comp` be `less{}`, and `proj1` and `proj2` be `identity{}` for the overloads with no parameters by those names.

2 ~~*Requires:*~~ *Expects:* For the overloads in namespace `std`, `RandomAccessIterator` ~~shall meet~~meets the *Cpp17ValueSwappable* requirements (**??**), the type of `*result_first` ~~shall meet~~meets the *Cpp17MoveConstructible* (Table **??**) and *Cpp17MoveAssignable* (Table **??**) requirements, and the expression `*first` ~~shall be~~is writable (**??**) to `result_first`.

3 *Expects:* For iterators `a1` and `b1` in `[first, last)`, and iterators `x2` and `y2` in `[result_first, result_last)`, after evaluating the assignment `*y2 = *b1`, let E be the value of

```
bool(invoke(comp, invoke(proj1, *a1), invoke(proj2, *y2))).
```

Then, after evaluating the assignment `*x2 = *a1`, E is equal to

```
bool(invoke(comp, invoke(proj2, *x2), invoke(proj2, *y2))).
```

[*Note*: Writing a value from the input range into the output range does not affect how it is ordered by `comp` and `proj1` or `proj2`. — *end note*]

4 *Effects:* Places the first N elements as sorted with respect to `comp` and `proj2` into the range `[result_first, result_first + N)`.

5 *Returns:* `result_first + N`.

6 *Complexity:* Approximately `(last - first) * log` N comparisons, and twice as many projections.

25.7.1.5 `is_sorted` [is.sorted]

```
template<class ForwardIterator>
  constexpr bool is_sorted(ForwardIterator first, ForwardIterator last);
```

1 *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 *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 *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 *Effects:* Equivalent to:

```
is_sorted_until(std::forward<ExecutionPolicy>(exec), first, last, comp) == last
```

```
template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
         IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less>
  constexpr bool ranges::is_sorted(I first, S last, Comp comp = {}, Proj proj = {});
```

```
template<ForwardRange R, class Proj = identity,
         IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
  constexpr bool ranges::is_sorted(R&& r, Comp comp = {}, Proj proj = {});
```

5 *Effects:* Equivalent to: `return ranges::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);

template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
         IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less>
  constexpr I ranges::is_sorted_until(I first, S last, Comp comp = {}, Proj proj = {});
template<ForwardRange R, class Proj = identity,
         IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
  constexpr safe_iterator_t<R>
    ranges::is_sorted_until(R&& r, Comp comp = {}, Proj proj = {});
```

6 Let `comp` be `less{}` and `proj` be `identity{}` for the overloads with no parameters by those names.

7 *Returns:* The last iterator `i` in `[first, last]` for which the range `[first, i)` is sorted with respect to `comp` and `proj`.

8 *Complexity:* Linear.

25.7.2 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);

template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less,
         class Proj = identity>
  requires Sortable<I, Comp, Proj>
  constexpr I
    ranges::nth_element(I first, I nth, S last, Comp comp = {}, Proj proj = {});
```

1 Let `comp` be `less{}` and `proj` be `identity{}` for the overloads with no parameters by those names.

2 ~~*Requires:*~~ *Expects:* `[first, nth)` and `[nth, last)` ~~shall be~~are valid ranges. For the overloads in namespace `std`, `RandomAccessIterator` ~~shall meet~~meets the *Cpp17ValueSwappable* requirements (**??**),

and the type of `*first` ~~shall meet~~meets the *Cpp17MoveConstructible* (Table **??**) and *Cpp17MoveAssignable* (Table **??**) 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 with respect to `comp` and `proj`, 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: `bool(invoke(comp, invoke(proj, *j), invoke(proj, *i)))` is `false`.

4 *Returns:* `last` for the overload 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`.

```
template<RandomAccessRange R, class Comp = ranges::less, class Proj = identity>
  requires Sortable<iterator_t<R>, Comp, Proj>
  constexpr safe_iterator_t<R>
    ranges::nth_element(R&& r, iterator_t<R> nth, Comp comp = {}, Proj proj = {});
```

6 *Effects:* Equivalent to:

```
return ranges::nth_element(ranges::begin(r), nth, ranges::end(r), comp, proj);
```

25.7.3 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 comparison function. They work on non-random access iterators minimizing the number of comparisons, which will be logarithmic for all types of iterators. They are especially appropriate for random access iterators, because these algorithms do a logarithmic number of steps through the data structure. For non-random access iterators they execute a linear number of steps.

25.7.3.1 `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);

template<ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
         IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = ranges::less>
  constexpr I ranges::lower_bound(I first, S last, const T& value, Comp comp = {},
                                  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>
    ranges::lower_bound(R&& r, const T& value, Comp comp = {}, Proj proj = {});
```

1 Let `comp` be `less{}` and `proj` be `identity{}` for overloads with no parameters by those names.

2 ~~*Requires:*~~ *Expects:* The elements `e` of `[first, last)` ~~shall be~~are partitioned with respect to the expression `bool(invoke(comp, invoke(proj, e), value))`.

3 *Returns:* The furthermost iterator `i` in the range `[first, last]` such that for every iterator `j` in the range `[first, i)`, `bool(invoke(comp, invoke(proj, *j), value))` is `true`.

4 *Complexity:* At most $\log_2(\texttt{last - first}) + \mathcal{O}(1)$ comparisons and projections.

25.7.3.2 `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);

template<ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
         IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = ranges::less>
  constexpr I ranges::upper_bound(I first, S last, const T& value, Comp comp = {}, 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>
    ranges::upper_bound(R&& r, const T& value, Comp comp = {}, Proj proj = {});
```

1 Let `comp` be `less{}` and `proj` be `identity{}` for overloads with no parameters by those names.

2 ~~*Requires:*~~ *Expects:* The elements `e` of `[first, last)` ~~shall be~~are partitioned with respect to the expression `!bool(invoke(comp, value, invoke(proj, e)))`.

3 *Returns:* The furthermost iterator `i` in the range `[first, last]` such that for every iterator `j` in the range `[first, i)`, `!bool(invoke(comp, value, invoke(proj, *j)))` is `true`.

4 *Complexity:* At most $\log_2(\texttt{last - first}) + \mathcal{O}(1)$ comparisons and projections.

25.7.3.3 `equal_range` [equal.range]

```
template<class ForwardIterator, class T>
  constexpr pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first,
                ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
  constexpr pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first,
                ForwardIterator last, const T& value,
                Compare comp);

template<ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
         IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = ranges::less>
  constexpr subrange<I>
    ranges::equal_range(I first, S last, const T& value, Comp comp = {}, 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>
    ranges::equal_range(R&& r, const T& value, Comp comp = {}, Proj proj = {});
```

1 Let `comp` be `less{}` and `proj` be `identity{}` for overloads with no parameters by those names.

2 ~~*Requires:*~~ *Expects:* The elements `e` of `[first, last)` ~~shall be~~are partitioned with respect to the expressions `bool(invoke(comp, invoke(proj, e), value))` and `!bool(invoke(comp, value, invoke(proj, e)))`. Also, for all elements `e` of `[first, last)`, `bool(comp(e, value))` ~~shall imply~~implies `!bool(comp(value, e))` for the overloads in namespace `std`.

3 *Returns:*

(3.1) — For the overloads in namespace `std`:

```
{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 $2 * \log_2(\texttt{last - first}) + \mathcal{O}(1)$ comparisons and projections.

25.7.3.4 `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);

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

1 Let `comp` be `less{}` and `proj` be `identity{}` for overloads with no parameters by those names.

2 ~~*Requires:*~~ *Expects:* The elements `e` of `[first, last)` ~~shall be~~are partitioned with respect to the expressions `bool(invoke(comp, invoke(proj, e), value))` and `!bool(invoke(comp, value, invoke(proj, e)))`. Also, for all elements `e` of `[first, last)`, `bool(comp(e, value))` ~~shall imply~~implies `!bool(comp(value, e))` for the overloads in namespace `std`.

3 *Returns:* `true` if and only if for some iterator `i` in the range `[first, last)`, `!bool(invoke(comp, invoke(proj, *i), value)) && !bool(invoke(comp, value, invoke(proj, *i)))` is `true`.

4 *Complexity:* At most $\log_2(\texttt{last - first}) + \mathcal{O}(1)$ comparisons and projections.

25.7.4 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);

template<InputIterator I, Sentinel<I> S, class Proj = identity,
         IndirectUnaryPredicate<projected<I, Proj>> Pred>
  constexpr bool ranges::is_partitioned(I first, S last, Pred pred, Proj proj = {});
template<InputRange R, class Proj = identity,
         IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
  constexpr bool ranges::is_partitioned(R&& r, Pred pred, Proj proj = {});
```

1 Let `proj` be `identity{}` for the overloads with no parameter named `proj`.

2 *Returns:* `true` if and only if the elements `e` of `[first, last)` are partitioned with respect to the expression `bool(invoke(pred, invoke(proj, e)))`.

3 *Complexity:* Linear. At most `last - first` applications of `pred` and `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,
              ForwardIterator first, ForwardIterator last, Predicate pred);

template<Permutable I, Sentinel<I> S, class Proj = identity,
         IndirectUnaryPredicate<projected<I, Proj>> Pred>
  constexpr I
    ranges::partition(I first, S last, Pred pred, 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>
    ranges::partition(R&& r, Pred pred, Proj proj = {});
```

4 Let `proj` be `identity{}` for the overloads with no parameter named `proj` and let $E(x)$ be `bool(invoke(pred, invoke(proj, `x`)))`.

5 ~~*Requires:*~~ *Expects:* For the overloads in namespace `std`, `ForwardIterator` ~~shall meet~~meets the *Cpp17ValueSwappable* requirements (**??**).

6 *Effects:* Places all the elements `e` in `[first, last)` that satisfy E(`e`) before all the elements that do not.

7 *Returns:* An iterator `i` such that E(`*j`) is `true` for every iterator `j` in `[first, i)` and `false` for every iterator `j` in `[i, last)`.

8 *Complexity:* Let $N =$ `last - first`:

(8.1) — For the overload with no `ExecutionPolicy`, exactly N applications of the predicate and projection. At most $N/2$ swaps if the type of `first` meets the *Cpp17BidirectionalIterator* requirements for the overloads in namespace `std` or models `BidirectionalIterator` for the overloads in namespace `ranges`, and at most N swaps otherwise.

(8.2) — 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);

template<BidirectionalIterator I, Sentinel<I> S, class Proj = identity,
         IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires Permutable<I>
  I ranges::stable_partition(I first, S last, Pred pred, Proj proj = {});
template<BidirectionalRange R, class Proj = identity,
         IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
  requires Permutable<iterator_t<R>>
  safe_iterator_t<R> ranges::stable_partition(R&& r, Pred pred, Proj proj = {});
```

9 Let `proj` be `identity{}` for the overloads with no parameter named `proj` and let $E(x)$ be `bool(invoke(pred, invoke(proj, `x`)))`.

~~*Requires:*~~ *Expects:* For the overloads in namespace `std`, `BidirectionalIterator` ~~shall meet~~meets the *Cpp17ValueSwappable* requirements (**??**) and the type of `*first` ~~shall meet~~meets the *Cpp17MoveConstructible* (Table **??**) and *Cpp17MoveAssignable* (Table **??**) requirements.

10 *Effects:* Places all the elements `e` in `[first, last)` that satisfy E(`e`) before all the elements that do not. The relative order of the elements in both groups is preserved.

11 *Returns:* An iterator `i` such that for every iterator `j` in `[first, i)`, E(`*j`) is `true`, and for every iterator `j` in the range `[i, last)`, E(`*j`) is `false`,

12 *Complexity:* Let $N =$ `last - first`:

(12.1) — 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 projection.

(12.2) — 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);

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 ranges::partition_copy_result<I, O1, O2>
    ranges::partition_copy(I first, S last, O1 out_true, O2 out_false, Pred pred,
                  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 ranges::partition_copy_result<safe_iterator_t<R>, O1, O2>
    ranges::partition_copy(R&& r, O1 out_true, O2 out_false, Pred pred, Proj proj = {});
```

13 Let `proj` be `identity{}` for the overloads with no parameter named `proj` and let $E(x)$ be `bool(invoke(pred, invoke(proj, `x`)))`.

14 ~~*Requires:*~~ *Expects:* The input range and output ranges ~~shall~~do not overlap. For the overloads in namespace `std`, the expression `*first` ~~shall be~~is writable (`??`) to `out_true` and `out_false`. [*Note*: For the overload with an `ExecutionPolicy`, there may be a performance cost if `first`'s value type does not meet the *Cpp17CopyConstructible* requirements. — *end note*]

15 *Effects:* For each iterator `i` in `[first, last)`, copies `*i` to the output range beginning with `out_true` if E(`*i`) is `true`, or to the output range beginning with `out_false` otherwise.

16 *Returns:* 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

(16.1) — `{o1, o2}` for the overloads in namespace `std`, or

(16.2) — `{last, o1, o2}` for the overloads in namespace `ranges`.

17 *Complexity:* Exactly `last - first` applications of `pred` and `proj`.

```
template<class ForwardIterator, class Predicate>
  constexpr ForwardIterator
    partition_point(ForwardIterator first, ForwardIterator last, Predicate pred);

template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
         IndirectUnaryPredicate<projected<I, Proj>> Pred>
  constexpr I ranges::partition_point(I first, S last, Pred pred, Proj proj = {});
template<ForwardRange R, class Proj = identity,
         IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
  constexpr safe_iterator_t<R>
    ranges::partition_point(R&& r, Pred pred, Proj proj = {});
```

18 Let `proj` be `identity{}` for the overloads with no parameter named `proj` and let $E(x)$ be `bool(invoke(pred, invoke(proj, `x`)))`.

19 ~~*Requires:*~~ *Expects:* The elements `e` of `[first, last)` ~~shall be~~are partitioned with respect to E(`e`).

20 *Returns:* An iterator `mid` such that E(`*i`) is `true` for all iterators `i` in `[first, mid)`, and `false` for all iterators `i` in `[mid, last)`

21 *Complexity:* $\mathcal{O}$(log(`last - first`)) applications of `pred` and `proj`.

25.7.5 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);

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 ranges::merge_result<I1, I2, O>
    ranges::merge(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                  Comp comp = {}, Proj1 proj1 = {}, 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 ranges::merge_result<safe_iterator_t<R1>, safe_iterator_t<R2>, O>
    ranges::merge(R1&& r1, R2&& r2, O result,
                  Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});
```

1 Let N be `(last1 - first1) + (last2 - first2)`. Let `comp` be `less{}`, `proj1` be `identity{}`, and `proj2` be `identity{}`, for the overloads with no parameters by those names.

2 ~~*Requires:*~~ *Expects:* The ranges `[first1, last1)` and `[first2, last2)` ~~shall be~~are sorted with respect to `comp` and `proj1` or `proj2`, respectively. The resulting range ~~shall~~does 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`. If an element `a` precedes `b` in an input range, `a` is copied into the output range before `b`. If `e1` is an element of `[first1, last1)` and `e2` of `[first2, last2)`, `e2` is copied into the output range before `e1` if and only if `bool(invoke(comp, invoke(proj2, e2), invoke(proj1, e1)))` is `true`.

4 *Returns:*

(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:*

(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 (**??**).

```
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);

template<BidirectionalIterator I, Sentinel<I> S, class Comp = ranges::less,
         class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I ranges::inplace_merge(I first, I middle, S last, Comp comp = {}, Proj proj = {});
```

7 Let `comp` be `less{}` and `proj` be `identity{}` for the overloads with no parameters by those names.

8 ~~*Requires:*~~ *Expects:* `[first, middle)` and `[middle, last)` ~~shall be~~are valid ranges sorted with respect to `comp` and `proj`. For the overloads in namespace `std`, `BidirectionalIterator` ~~shall meet~~meets the *Cpp17ValueSwappable* requirements (**??**) and the type of `*first` ~~shall meet~~meets the *Cpp17MoveConstructible* (Table **??**) and *Cpp17MoveAssignable* (Table **??**) 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 is sorted with respect to `comp` and `proj`.

10 *Returns:* `last` for the overload in namespace `ranges`.

11 *Complexity:* Let $N =$ `last - first`:

(11.1) — For the overloads with no `ExecutionPolicy`, and if enough additional memory is available, exactly $N - 1$ comparisons.

(11.2) — Otherwise, $\mathcal{O}(N \log N)$ comparisons.

In either case, twice as many projections as comparisons.

12 *Remarks:* Stable (**??**).

```
template<BidirectionalRange R, class Comp = ranges::less, class Proj = identity>
  requires Sortable<iterator_t<R>, Comp, Proj>
  safe_iterator_t<R>
    ranges::inplace_merge(R&& r, iterator_t<R> middle, Comp comp = {}, Proj proj = {});
```

13 *Effects:* Equivalent to:

```
return ranges::inplace_merge(ranges::begin(r), middle, ranges::end(r), comp, proj);
```

25.7.6 Set operations on sorted structures [alg.set.operations]

1 This subclause defines all the basic set operations on sorted structures. They also work with `multisets` (**??**) containing multiple copies of equivalent elements. The semantics of the set operations are generalized to `multisets` 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.

25.7.6.1 `includes` [includes]

```
template<class InputIterator1, class InputIterator2>
  constexpr bool includes(InputIterator1 first1, InputIterator1 last1,
                          InputIterator2 first2, InputIterator2 last2);
```

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  bool includes(ExecutionPolicy&& exec,
                ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2, ForwardIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
  constexpr bool includes(InputIterator1 first1, InputIterator1 last1,
                          InputIterator2 first2, InputIterator2 last2,
                          Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class Compare>
  bool includes(ExecutionPolicy&& exec,
                ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2, ForwardIterator2 last2,
                Compare comp);

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 ranges::includes(I1 first1, S1 last1, I2 first2, S2 last2, Comp comp = {},
                                  Proj1 proj1 = {}, 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 ranges::includes(R1&& r1, R2&& r2, Comp comp = {},
                                  Proj1 proj1 = {}, Proj2 proj2 = {});
```

1 Let `comp` be `less{}`, `proj1` be `identity{}`, and `proj2` be `identity{}`, for the overloads with no parameters by those names.

2 ~~*Requires:*~~ *Expects:* The ranges `[first1, last1)` and `[first2, last2)` ~~shall be~~are sorted with respect to `comp` and `proj1` or `proj2`, respectively.

3 *Returns:* `true` if and only if `[first2, last2)` is 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) + (last2 - first2)) - 1` comparisons and applications of each projection.

25.7.6.2 `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);

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 ranges::set_union_result<I1, I2, O>
    ranges::set_union(I1 first1, S1 last1, I2 first2, S2 last2, O result, Comp comp = {},
                      Proj1 proj1 = {}, 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 ranges::set_union_result<safe_iterator_t<R1>, safe_iterator_t<R2>, O>
    ranges::set_union(R1&& r1, R2&& r2, O result, Comp comp = {},
                      Proj1 proj1 = {}, Proj2 proj2 = {});
```

1 Let `comp` be `less{}`, and `proj1` and `proj2` be `identity{}` for the overloads with no parameters by those names.

2 ~~*Requires:*~~ *Expects:* The ranges `[first1, last1)` and `[first2, last2)` ~~shall be~~are sorted with respect to `comp` and `proj1` or `proj2`, respectively. The resulting range ~~shall~~does not overlap with either of the original ranges.

3 *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.

4 *Returns:* Let `result_last` be the end of the constructed range. Returns

(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:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons and applications of each projection.

6 *Remarks:* 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 are copied to the output range, in order, and then the final $\max(n - m, 0)$ elements from the second range are copied to the output range, in order.

25.7.6.3 `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);

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 ranges::set_intersection_result<I1, I2, O>
    ranges::set_intersection(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                             Comp comp = {}, Proj1 proj1 = {}, 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 ranges::set_intersection_result<safe_iterator_t<R1>, safe_iterator_t<R2>, O>
    ranges::set_intersection(R1&& r1, R2&& r2, O result,
                             Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});
```

1 Let `comp` be `less{}`, and `proj1` and `proj2` be `identity{}` for the overloads with no parameters by those names.

2 ~~*Requires:*~~ *Expects:* The ranges `[first1, last1)` and `[first2, last2)` ~~shall be~~are sorted with respect to `comp` and `proj1` or `proj2`, respectively. The resulting range ~~shall~~does not overlap with either of the original ranges.

3 *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.

4 *Returns:* Let `result_last` be the end of the constructed range. Returns

(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:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons and applications of each projection.

6 *Remarks:* 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 are copied from the first range to the output range, in order.

25.7.6.4 `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);

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 ranges::set_difference_result<I1, O>
    ranges::set_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                           Comp comp = {}, Proj1 proj1 = {}, 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 ranges::set_difference_result<safe_iterator_t<R1>, O>
    ranges::set_difference(R1&& r1, R2&& r2, O result,
                           Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});
```

1 Let `comp` be `less{}`, and `proj1` and `proj2` be `identity{}` for the overloads with no parameters by those names.

2 ~~*Requires:*~~ *Expects:* The ranges `[first1, last1)` and `[first2, last2)` ~~shall be~~are sorted with respect to `comp` and `proj1` or `proj2`, respectively. The resulting range ~~shall~~does not overlap with either of the original ranges.

3 *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.

4 *Returns:* Let `result_last` be the end of the constructed range. Returns

(4.1) — `result_last` for the overloads in namespace `std`, or

(4.2) — `{last1, result_last}` for the overloads in namespace `ranges`.

5 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons and applications of each projection.

6 *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)` is copied to the output range, in order.

25.7.6.5 `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);

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 ranges::set_symmetric_difference_result<I1, I2, O>
    ranges::set_symmetric_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                                     Comp comp = {}, Proj1 proj1 = {},
                                     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 ranges::set_symmetric_difference_result<safe_iterator_t<R1>, safe_iterator_t<R2>, O>
    ranges::set_symmetric_difference(R1&& r1, R2&& r2, O result, Comp comp = {},
                                     Proj1 proj1 = {}, Proj2 proj2 = {});
```

1 Let `comp` be `less{}`, and `proj1` and `proj2` be `identity{}` for the overloads with no parameters by those names.

2 ~~*Requires:*~~ *Expects:* The resulting range ~~shall~~does not overlap with either of the original ranges. The ranges `[first1, last1)` and `[first2, last2)` ~~shall be~~are sorted with respect to `comp` and `proj1` or `proj2`, respectively. ~~The resulting range shall not overlap with either of the original ranges.~~

3 *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.

4 *Returns:* Let `result_last` be the end of the constructed range. Returns

(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:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons and applications of each projection.

6 *Remarks:* 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 $|m - n|$ 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 $m < n$. In either case, the elements are copied in order.

25.7.7 Heap operations [alg.heap.operations]

1 A random access range `[a, b)` is a *heap with respect to `comp` and `proj`* for a comparator and projection `comp` and `proj` if its elements are organized such that:

(1.1) — With N `= b - a`, for all i, $0 < i < N$, `bool(invoke(comp, invoke(proj, a[`$\lfloor\frac{i-1}{2}\rfloor$`]), invoke(proj, a[`$i$`])))` is `false`.

(1.2) — `*a` may be removed by `pop_heap`, or a new element added by `push_heap`, in $\mathcal{O}(\log N)$ time.

2 These properties make heaps useful as priority queues.

3 `make_heap` converts a range into a heap and `sort_heap` turns a heap into a sorted sequence.

25.7.7.1 `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);

template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less,
         class Proj = identity>
  requires Sortable<I, Comp, Proj>
  constexpr I
    ranges::push_heap(I first, S last, Comp comp = {}, Proj proj = {});
template<RandomAccessRange R, class Comp = ranges::less, class Proj = identity>
  requires Sortable<iterator_t<R>, Comp, Proj>
  constexpr safe_iterator_t<R>
    ranges::push_heap(R&& r, Comp comp = {}, Proj proj = {});
```

1 Let `comp` be `less{}` and `proj` be `identity{}` for the overloads with no parameters by those names.

2 ~~*Requires:*~~ *Expects:* The range `[first, last - 1)` ~~shall be~~is a valid heap with respect to `comp` and `proj`. For the overloads in namespace `std`, the type of `*first` ~~shall meet~~meets the *Cpp17MoveConstructible* requirements (Table **??**) and the *Cpp17MoveAssignable* requirements (Table **??**).

3 *Effects:* Places the value in the location `last - 1` into the resulting heap `[first, last)`.

4 *Returns:* `last` for the overloads in namespace `ranges`.

5 *Complexity:* At most log(`last - first`) comparisons and twice as many projections.

25.7.7.2 `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);

template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less,
         class Proj = identity>
  requires Sortable<I, Comp, Proj>
  constexpr I
    ranges::pop_heap(I first, S last, Comp comp = {}, Proj proj = {});
template<RandomAccessRange R, class Comp = ranges::less, class Proj = identity>
  requires Sortable<iterator_t<R>, Comp, Proj>
  constexpr safe_iterator_t<R>
    ranges::pop_heap(R&& r, Comp comp = {}, Proj proj = {});
```

1 Let `comp` be `less{}` and `proj` be `identity{}` for the overloads with no parameters by those names.

2 ~~*Requires:*~~ *Expects:* The range `[first, last)` ~~shall be~~is a valid non-empty heap with respect to `comp` and `proj`. For the overloads in namespace `std`, `RandomAccessIterator` ~~shall meet~~meets the *Cpp17ValueSwappable* requirements (**??**) and the type of `*first` ~~shall meet~~meets the *Cpp17MoveConstructible* (Table **??**) and *Cpp17MoveAssignable* (Table **??**) requirements.

3 *Effects:* Swaps the value in the location `first` with the value in the location `last - 1` and makes `[first, last - 1)` into a heap with respect to `comp` and `proj`.

4 *Returns:* `last` for the overloads in namespace `ranges`.

5 *Complexity:* At most 2 log(`last - first`) comparisons and twice as many projections.

25.7.7.3 `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);
```

```
template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less,
         class Proj = identity>
  requires Sortable<I, Comp, Proj>
  constexpr I
    ranges::make_heap(I first, S last, Comp comp = {}, Proj proj = {});
template<RandomAccessRange R, class Comp = ranges::less, class Proj = identity>
  requires Sortable<iterator_t<R>, Comp, Proj>
  constexpr safe_iterator_t<R>
    ranges::make_heap(R&& r, Comp comp = {}, Proj proj = {});
```

1 Let `comp` be `less{}` and `proj` be `identity{}` for the overloads with no parameters by those names.

2 ~~*Requires:*~~ *Expects:* For the overloads in namespace `std`, the type of `*first` ~~shall meet~~meets the *Cpp17MoveConstructible* (Table **??**) and *Cpp17MoveAssignable* (Table **??**) requirements.

3 *Effects:* Constructs a heap with respect to `comp` and `proj` out of the range `[first, last)`.

4 *Returns:* `last` for the overloads in namespace `ranges`.

5 *Complexity:* At most 3(`last - first`) comparisons and twice as many projections.

25.7.7.4 `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);

template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less,
         class Proj = identity>
  requires Sortable<I, Comp, Proj>
  constexpr I
    ranges::sort_heap(I first, S last, Comp comp = {}, Proj proj = {});
template<RandomAccessRange R, class Comp = ranges::less, class Proj = identity>
  requires Sortable<iterator_t<R>, Comp, Proj>
  constexpr safe_iterator_t<R>
    ranges::sort_heap(R&& r, Comp comp = {}, Proj proj = {});
```

1 Let `comp` be `less{}` and `proj` be `identity{}` for the overloads with no parameters by those names.

2 ~~*Requires:*~~ *Expects:* The range `[first, last)` ~~shall be~~is a valid heap with respect to `comp` and `proj`. For the overloads in namespace `std`, `RandomAccessIterator` ~~shall meet~~meets the *Cpp17ValueSwappable* requirements (**??**) and the type of `*first` ~~shall meet~~meets the *Cpp17MoveConstructible* (Table **??**) and *Cpp17MoveAssignable* (Table **??**) requirements.

3 *Effects:* Sorts elements in the heap `[first, last)` with respect to `comp` and `proj`.

4 *Returns:* `last` for the overloads in namespace `ranges`.

5 *Complexity:* At most $2N \log N$ comparisons, where $N =$ `last - first`, and twice as many projections.

25.7.7.5 `is_heap` [is.heap]

```
template<class RandomAccessIterator>
  constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last);
```

1 *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 *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 *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 *Effects:* Equivalent to:

```
return is_heap_until(std::forward<ExecutionPolicy>(exec), first, last, comp) == last;
```

```
template<RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
         IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less>
  constexpr bool ranges::is_heap(I first, S last, Comp comp = {}, Proj proj = {});
template<RandomAccessRange R, class Proj = identity,
         IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
  constexpr bool ranges::is_heap(R&& r, Comp comp = {}, 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);

template<RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
         IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less>
  constexpr I ranges::is_heap_until(I first, S last, Comp comp = {}, Proj proj = {});
template<RandomAccessRange R, class Proj = identity,
         IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
  constexpr safe_iterator_t<R>
    ranges::is_heap_until(R&& r, Comp comp = {}, Proj proj = {});
```

6 Let `comp` be `less{}` and `proj` be `identity{}` for the overloads with no parameters by those names.

7 *Returns:* The last iterator `i` in `[first, last]` for which the range `[first, i)` is a heap with respect to `comp` and `proj`.

8 *Complexity:* Linear.

25.7.8 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);

template<class T, class Proj = identity,
         IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less>
```

```
  constexpr const T& ranges::min(const T& a, const T& b, Comp comp = {}, Proj proj = {});
```

1 ~~*Requires:*~~ *Expects:* For the first form, type `T` ~~shall be~~meets the *Cpp17LessThanComparable* requirements (Table **??**).

2 *Returns:* The smaller value.

3 *Remarks:* Returns the first argument when the arguments are equivalent. An invocation may explicitly specify an argument for the template parameter `T` of the overloads in namespace `std`.

4 *Complexity:* Exactly one comparison and two applications of the projection, if any.

```
template<class T>
  constexpr T min(initializer_list<T> r);
template<class T, class Compare>
  constexpr T min(initializer_list<T> r, Compare comp);

template<Copyable T, class Proj = identity,
         IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less>
  constexpr T ranges::min(initializer_list<T> r, Comp comp = {}, Proj proj = {});
template<InputRange R, class Proj = identity,
         IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
  requires IndirectlyCopyableStorable<iterator_t<R>, iter_value_t<iterator_t<R>>*>
  constexpr iter_value_t<iterator_t<R>>
    ranges::min(R&& r, Comp comp = {}, Proj proj = {});
```

5 ~~*Requires:*~~ *Expects:* `ranges::distance(r) > 0`. For the overloads in namespace `std`, `T` ~~shall be~~meets the *Cpp17CopyConstructible* requirements. For the first form, type `T` ~~shall be~~meets the *Cpp17LessThanComparable* requirements (Table **??**).

6 *Returns:* The smallest value in the input range.

7 *Remarks:* Returns a copy of the leftmost element when several elements are equivalent to the smallest. An invocation may explicitly specify an argument for the template parameter `T` of the overloads in namespace `std`.

8 *Complexity:* Exactly `ranges::distance(r) - 1` comparisons and twice as many applications of the projection, if any.

```
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, class Proj = identity,
         IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less>
  constexpr const T& ranges::max(const T& a, const T& b, Comp comp = {}, Proj proj = {});
```

9 ~~*Requires:*~~ *Expects:* For the first form, type `T` ~~shall be~~meets the *Cpp17LessThanComparable* requirements (Table **??**).

10 *Returns:* The larger value.

11 *Remarks:* Returns the first argument when the arguments are equivalent. An invocation may explicitly specify an argument for the template parameter `T` of the overloads in namespace `std`.

12 *Complexity:* Exactly one comparison and two applications of the projection, if any.

```
template<class T>
  constexpr T max(initializer_list<T> r);
template<class T, class Compare>
  constexpr T max(initializer_list<T> r, Compare comp);

template<Copyable T, class Proj = identity,
         IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less>
  constexpr T ranges::max(initializer_list<T> r, Comp comp = {}, Proj proj = {});
template<InputRange R, class Proj = identity,
         IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
  requires IndirectlyCopyableStorable<iterator_t<R>, iter_value_t<iterator_t<R>>*>
  constexpr iter_value_t<iterator_t<R>>
```

```
ranges::max(R&& r, Comp comp = {}, Proj proj = {});
```

13 ~~*Requires:*~~ *Expects:* `ranges::distance(r) > 0`. For the overloads in namespace `std`, `T` ~~shall be~~is *Cpp17CopyConstructible*. For the first form, type `T` ~~shall be~~meets the *Cpp17LessThanComparable* requirements (Table **??**).

14 *Returns:* The largest value in the input range.

15 *Remarks:* Returns a copy of the leftmost element when several elements are equivalent to the largest. An invocation may explicitly specify an argument for the template parameter `T` of the overloads in namespace `std`.

16 *Complexity:* Exactly `ranges::distance(r) - 1` comparisons and twice as many applications of the projection, if any.

```
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, class Proj = identity,
         IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less>
  constexpr ranges::minmax_result<const T&>
    ranges::minmax(const T& a, const T& b, Comp comp = {}, Proj proj = {});
```

17 ~~*Requires:*~~ *Expects:* For the first form, type `T` ~~shall be~~meets the *Cpp17LessThanComparable* requirements (Table **??**).

18 *Returns:* `{b, a}` if `b` is smaller than `a`, and `{a, b}` otherwise.

19 *Remarks:* An invocation may explicitly specify an argument for the template parameter `T` of the overloads in namespace `std`.

20 *Complexity:* Exactly one comparison and two applications of the projection, if any.

```
template<class T>
  constexpr pair<T, T> minmax(initializer_list<T> t);
template<class T, class Compare>
  constexpr pair<T, T> minmax(initializer_list<T> t, Compare comp);

template<Copyable T, class Proj = identity,
         IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less>
  constexpr ranges::minmax_result<T>
    ranges::minmax(initializer_list<T> r, Comp comp = {}, Proj proj = {});
template<InputRange R, class Proj = identity,
         IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
  requires IndirectlyCopyableStorable<iterator_t<R>, iter_value_t<iterator_t<R>>*>
  constexpr ranges::minmax_result<iter_value_t<iterator_t<R>>>
    ranges::minmax(R&& r, Comp comp = {}, Proj proj = {});
```

21 ~~*Requires:*~~ *Expects:* `ranges::distance(r) > 0`. For the overloads in namespace `std`, `T` ~~shall be~~is *Cpp17CopyConstructible*. For the first form, type `T` ~~shall be~~meets the *Cpp17LessThanComparable* requirements (Table **??**).

22 *Returns:* Let `X` be the return type. Returns `X{x, y}`, where `x` is a copy of the leftmost element with the smallest and `y` a copy of the rightmost element with the largest value in the input range.

23 *Remarks:* An invocation may explicitly specify an argument for the template parameter `T` of the overloads in namespace `std`.

24 *Complexity:* At most (3/2)`ranges::distance(r)` applications of the corresponding predicate and twice as many applications of the projection, if any.

```
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);

template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
         IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less>
  constexpr I ranges::min_element(I first, S last, Comp comp = {}, Proj proj = {});
template<ForwardRange R, class Proj = identity,
         IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
  constexpr safe_iterator_t<R>
    ranges::min_element(R&& r, Comp comp = {}, Proj proj = {});
```

25 Let `comp` be `less{}` and `proj` be `identity{}` for the overloads with no parameters by those names.

26 *Returns:* The first iterator `i` in the range `[first, last)` such that for every iterator `j` in the range `[first, last)`,

```
bool(invoke(comp, invoke(proj, *j), invoke(proj, *i)))
```

is `false`. Returns `last` if `first == last`.

27 *Complexity:* Exactly $\max(\texttt{last - first - 1}, 0)$ comparisons and twice as many projections.

```
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);

template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
         IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less>
  constexpr I ranges::max_element(I first, S last, Comp comp = {}, Proj proj = {});
template<ForwardRange R, class Proj = identity,
         IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
  constexpr safe_iterator_t<R>
    ranges::max_element(R&& r, Comp comp = {}, Proj proj = {});
```

28 Let `comp` be `less{}` and `proj` be `identity{}` for the overloads with no parameters by those names.

29 *Returns:* The first iterator `i` in the range `[first, last)` such that for every iterator `j` in the range `[first, last)`,

```
bool(invoke(comp, invoke(proj, *i), invoke(proj, *j)))
```

is `false`. Returns `last` if `first == last`.

30 *Complexity:* Exactly $\max(\texttt{last - first - 1}, 0)$ comparisons and twice as many projections.

```
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);

template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
         IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less>
  constexpr ranges::minmax_result<I>
    ranges::minmax_element(I first, S last, Comp comp = {}, Proj proj = {});
template<ForwardRange R, class Proj = identity,
         IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
  constexpr ranges::minmax_result<safe_iterator_t<R>>
    ranges::minmax_element(R&& r, Comp comp = {}, Proj proj = {});
```

31 *Returns:* `{first, first}` if `[first, last)` is empty, otherwise `{m, M}`, where `m` is the first iterator in `[first, last)` such that no iterator in the range refers to a smaller element, and where `M` is the last iterator[238] 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 the projection, if any.

25.7.9 Bounded value [alg.clamp]

```
template<class T>
  constexpr const T& clamp(const T& v, const T& lo, const T& hi);
template<class T, class Compare>
  constexpr const T& clamp(const T& v, const T& lo, const T& hi, Compare comp);
```

1 ~~*Requires:*~~ *Expects:* The value of `lo` ~~shall be~~is no greater than `hi`. For the first form, type `T` ~~shall be~~meets the *Cpp17LessThanComparable* requirements (Table ??).

2 *Returns:* `lo` if `v` is less than `lo`, `hi` if `hi` is less than `v`, otherwise `v`.

3 [*Note*: If NaN is avoided, `T` can be a floating-point type. — *end note*]

4 *Complexity:* At most two comparisons.

25.7.10 Lexicographical comparison [alg.lex.comparison]

```
template<class InputIterator1, class InputIterator2>
  constexpr bool
    lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  bool
    lexicographical_compare(ExecutionPolicy&& exec,
                            ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
  constexpr bool
    lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2,
                            Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Compare>
  bool
    lexicographical_compare(ExecutionPolicy&& exec,
                            ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2,
                            Compare comp);
```

238) This behavior intentionally differs from `max_element()`.

```
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
    ranges::lexicographical_compare(I1 first1, S1 last1, I2 first2, S2 last2,
                                    Comp comp = {}, Proj1 proj1 = {}, 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
    ranges::lexicographical_compare(R1&& r1, R2&& r2, Comp comp = {},
                                    Proj1 proj1 = {}, 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)`.

2 *Complexity:* At most 2 min(`last1 - first1`, `last2 - first2`) applications of the corresponding comparison and each projection, if any.

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 proper 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*: `ranges::lexicographical_compare(I1, S1, I2, S2, Comp, Proj1, Proj2)` could be implemented as:

```
for ( ; first1 != last1 && first2 != last2 ; ++first1, (void) ++first2) {
  if (invoke(comp, invoke(proj1, *first1), invoke(proj2, *first2))) return true;
  if (invoke(comp, invoke(proj2, *first2), invoke(proj1, *first1))) return false;
}
return first1 == last1 && first2 != last2;
```

— *end example*]

5 [*Note*: An empty sequence is lexicographically less than any non-empty sequence, but not less than any empty sequence. — *end note*]

25.7.11 Three-way comparison algorithms [alg.3way]

```
template<class T, class U> constexpr auto compare_3way(const T& a, const U& b);
```

1 *Effects:* Compares two values and produces a result of the strongest applicable comparison category type:

(1.1) — Returns `a <=> b` if that expression is well-formed.

(1.2) — Otherwise, if the expressions `a == b` and `a < b` are each well-formed and convertible to `bool`, returns `strong_ordering::equal` when `a == b` is `true`, otherwise returns `strong_ordering::less` when `a < b` is `true`, and otherwise returns `strong_ordering::greater`.

(1.3) — Otherwise, if the expression `a == b` is well-formed and convertible to `bool`, returns `strong_equality::equal` when `a == b` is `true`, and otherwise returns `strong_equality::nonequal`.

(1.4) — Otherwise, the function is defined as deleted.

```
template<class InputIterator1, class InputIterator2, class Cmp>
  constexpr auto
    lexicographical_compare_3way(InputIterator1 b1, InputIterator1 e1,
                                 InputIterator2 b2, InputIterator2 e2,
                                 Cmp comp)
      -> common_comparison_category_t<decltype(comp(*b1, *b2)), strong_ordering>;
```

2 ~~*Requires:*~~ *Expects:* `Cmp` ~~shall be~~is a function object type whose return type is a comparison category type.

3 *Effects:* Lexicographically compares two ranges and produces a result of the strongest applicable comparison category type. Equivalent to:

```
for ( ; b1 != e1 && b2 != e2; void(++b1), void(++b2) )
  if (auto cmp = comp(*b1,*b2); cmp != 0)
    return cmp;
return b1 != e1 ? strong_ordering::greater :
       b2 != e2 ? strong_ordering::less :
                  strong_ordering::equal;
```

```
template<class InputIterator1, class InputIterator2>
  constexpr auto
    lexicographical_compare_3way(InputIterator1 b1, InputIterator1 e1,
                                 InputIterator2 b2, InputIterator2 e2);
```

4 *Effects:* Equivalent to:

```
return lexicographical_compare_3way(b1, e1, b2, e2,
                                    [](const auto& t, const auto& u) {
                                      return compare_3way(t, u);
                                    });
```

25.7.12 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);

template<BidirectionalIterator I, Sentinel<I> S, class Comp = ranges::less,
         class Proj = identity>
  requires Sortable<I, Comp, Proj>
  constexpr bool
    ranges::next_permutation(I first, S last, Comp comp = {}, Proj proj = {});
template<BidirectionalRange R, class Comp = ranges::less,
         class Proj = identity>
  requires Sortable<iterator_t<R>, Comp, Proj>
  constexpr bool
    ranges::next_permutation(R&& r, Comp comp = {}, Proj proj = {});
```

1 Let `comp` be `less{}` and `proj` be `identity{}` for overloads with no parameters by those names.

2 ~~*Requires:*~~ *Expects:* For the overloads in namespace `std`, `BidirectionalIterator` ~~shall meet~~meets the *Cpp17ValueSwappable* requirements (**??**).

3 *Effects:* Takes a sequence defined by the range `[first, last)` and transforms it into the next permutation. The next permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `comp` and `proj`. If no such permutation exists, transforms the sequence into the first permutation; that is, the ascendingly-sorted one.

4 *Returns:* `true` if and only if a next permutation was found.

5 *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);
```

```
template<BidirectionalIterator I, Sentinel<I> S, class Comp = ranges::less,
         class Proj = identity>
  requires Sortable<I, Comp, Proj>
  constexpr bool
    ranges::prev_permutation(I first, S last, Comp comp = {}, Proj proj = {});
template<BidirectionalRange R, class Comp = ranges::less,
         class Proj = identity>
  requires Sortable<iterator_t<R>, Comp, Proj>
  constexpr bool
    ranges::prev_permutation(R&& r, Comp comp = {}, Proj proj = {});
```

6 Let `comp` be `less{}` and `proj` be `identity{}` for overloads with no parameters by those names.

7 ~~*Requires:*~~ *Expects:* For the overloads in namespace `std`, `BidirectionalIterator` ~~shall meet~~meets the *Cpp17ValueSwappable* requirements (**??**).

8 *Effects:* Takes a sequence defined by the range `[first, last)` and transforms it into the previous permutation. The previous permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `comp` and `proj`. If no such permutation exists, transforms the sequence into the last permutation; that is, the descendingly-sorted one.

9 *Returns:* `true` if and only if a previous permutation was found.

10 *Complexity:* At most `(last - first) / 2` swaps.

25.8 Header `<numeric>` synopsis [numeric.ops.overview]

```
namespace std {
  // 25.9.2, accumulate
  template<class InputIterator, class T>
    T accumulate(InputIterator first, InputIterator last, T init);
  template<class InputIterator, class T, class BinaryOperation>
    T accumulate(InputIterator first, InputIterator last, T init, BinaryOperation binary_op);

  // 25.9.3, reduce
  template<class InputIterator>
    typename iterator_traits<InputIterator>::value_type
      reduce(InputIterator first, InputIterator last);
  template<class InputIterator, class T>
    T reduce(InputIterator first, InputIterator last, T init);
  template<class InputIterator, class T, class BinaryOperation>
    T reduce(InputIterator first, InputIterator last, T init, BinaryOperation binary_op);
  template<class ExecutionPolicy, class ForwardIterator>
    typename iterator_traits<ForwardIterator>::value_type
      reduce(ExecutionPolicy&& exec, // see 25.3.5
             ForwardIterator first, ForwardIterator last);
  template<class ExecutionPolicy, class ForwardIterator, class T>
    T reduce(ExecutionPolicy&& exec, // see 25.3.5
             ForwardIterator first, ForwardIterator last, T init);
  template<class ExecutionPolicy, class ForwardIterator, class T, class BinaryOperation>
    T reduce(ExecutionPolicy&& exec, // see 25.3.5
             ForwardIterator first, ForwardIterator last, T init, BinaryOperation binary_op);

  // 25.9.4, inner product
  template<class InputIterator1, class InputIterator2, class T>
    T inner_product(InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2, T init);
  template<class InputIterator1, class InputIterator2, class T,
           class BinaryOperation1, class BinaryOperation2>
    T inner_product(InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2, T init,
                    BinaryOperation1 binary_op1,
                    BinaryOperation2 binary_op2);
```

```
// 25.9.5, transform reduce
template<class InputIterator1, class InputIterator2, class T>
  T transform_reduce(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2,
                     T init);
template<class InputIterator1, class InputIterator2, class T,
         class BinaryOperation1, class BinaryOperation2>
  T transform_reduce(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2,
                     T init,
                     BinaryOperation1 binary_op1,
                     BinaryOperation2 binary_op2);
template<class InputIterator, class T,
         class BinaryOperation, class UnaryOperation>
  T transform_reduce(InputIterator first, InputIterator last,
                     T init,
                     BinaryOperation binary_op, UnaryOperation unary_op);
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2, class T>
  T transform_reduce(ExecutionPolicy&& exec, // see 25.3.5
                     ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2,
                     T init);
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2, class T,
         class BinaryOperation1, class BinaryOperation2>
  T transform_reduce(ExecutionPolicy&& exec, // see 25.3.5
                     ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2,
                     T init,
                     BinaryOperation1 binary_op1,
                     BinaryOperation2 binary_op2);
template<class ExecutionPolicy,
         class ForwardIterator, class T,
         class BinaryOperation, class UnaryOperation>
  T transform_reduce(ExecutionPolicy&& exec, // see 25.3.5
                     ForwardIterator first, ForwardIterator last,
                     T init,
                     BinaryOperation binary_op, UnaryOperation unary_op);

// 25.9.6, partial sum
template<class InputIterator, class OutputIterator>
  OutputIterator partial_sum(InputIterator first,
                             InputIterator last,
                             OutputIterator result);
template<class InputIterator, class OutputIterator, class BinaryOperation>
  OutputIterator partial_sum(InputIterator first,
                             InputIterator last,
                             OutputIterator result,
                             BinaryOperation binary_op);

// 25.9.7, exclusive scan
template<class InputIterator, class OutputIterator, class T>
  OutputIterator exclusive_scan(InputIterator first, InputIterator last,
                                OutputIterator result,
                                T init);
template<class InputIterator, class OutputIterator, class T, class BinaryOperation>
  OutputIterator exclusive_scan(InputIterator first, InputIterator last,
                                OutputIterator result,
                                T init, BinaryOperation binary_op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T>
  ForwardIterator2 exclusive_scan(ExecutionPolicy&& exec, // see 25.3.5
                                  ForwardIterator1 first, ForwardIterator1 last,
                                  ForwardIterator2 result,
```

```
                                T init);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T,
         class BinaryOperation>
  ForwardIterator2 exclusive_scan(ExecutionPolicy&& exec, // see 25.3.5
                                ForwardIterator1 first, ForwardIterator1 last,
                                ForwardIterator2 result,
                                T init, BinaryOperation binary_op);

// 25.9.8, inclusive scan
template<class InputIterator, class OutputIterator>
  OutputIterator inclusive_scan(InputIterator first, InputIterator last,
                                OutputIterator result);
template<class InputIterator, class OutputIterator, class BinaryOperation>
  OutputIterator inclusive_scan(InputIterator first, InputIterator last,
                                OutputIterator result,
                                BinaryOperation binary_op);
template<class InputIterator, class OutputIterator, class BinaryOperation, class T>
  OutputIterator inclusive_scan(InputIterator first, InputIterator last,
                                OutputIterator result,
                                BinaryOperation binary_op, T init);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 inclusive_scan(ExecutionPolicy&& exec, // see 25.3.5
                                ForwardIterator1 first, ForwardIterator1 last,
                                ForwardIterator2 result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryOperation>
  ForwardIterator2 inclusive_scan(ExecutionPolicy&& exec, // see 25.3.5
                                ForwardIterator1 first, ForwardIterator1 last,
                                ForwardIterator2 result,
                                BinaryOperation binary_op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryOperation, class T>
  ForwardIterator2 inclusive_scan(ExecutionPolicy&& exec, // see 25.3.5
                                ForwardIterator1 first, ForwardIterator1 last,
                                ForwardIterator2 result,
                                BinaryOperation binary_op, T init);

// 25.9.9, transform exclusive scan
template<class InputIterator, class OutputIterator, class T,
         class BinaryOperation, class UnaryOperation>
  OutputIterator transform_exclusive_scan(InputIterator first, InputIterator last,
                                          OutputIterator result,
                                          T init,
                                          BinaryOperation binary_op,
                                          UnaryOperation unary_op);
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2, class T,
         class BinaryOperation, class UnaryOperation>
  ForwardIterator2 transform_exclusive_scan(ExecutionPolicy&& exec, // see 25.3.5
                                            ForwardIterator1 first, ForwardIterator1 last,
                                            ForwardIterator2 result,
                                            T init,
                                            BinaryOperation binary_op,
                                            UnaryOperation unary_op);

// 25.9.10, transform inclusive scan
template<class InputIterator, class OutputIterator,
         class BinaryOperation, class UnaryOperation>
  OutputIterator transform_inclusive_scan(InputIterator first, InputIterator last,
                                          OutputIterator result,
                                          BinaryOperation binary_op,
                                          UnaryOperation unary_op);
```

```
template<class InputIterator, class OutputIterator,
         class BinaryOperation, class UnaryOperation, class T>
  OutputIterator transform_inclusive_scan(InputIterator first, InputIterator last,
                                          OutputIterator result,
                                          BinaryOperation binary_op,
                                          UnaryOperation unary_op,
                                          T init);
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2,
         class BinaryOperation, class UnaryOperation>
  ForwardIterator2 transform_inclusive_scan(ExecutionPolicy&& exec, // see 25.3.5
                                            ForwardIterator1 first, ForwardIterator1 last,
                                            ForwardIterator2 result,
                                            BinaryOperation binary_op,
                                            UnaryOperation unary_op);
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2,
         class BinaryOperation, class UnaryOperation, class T>
  ForwardIterator2 transform_inclusive_scan(ExecutionPolicy&& exec, // see 25.3.5
                                            ForwardIterator1 first, ForwardIterator1 last,
                                            ForwardIterator2 result,
                                            BinaryOperation binary_op,
                                            UnaryOperation unary_op,
                                            T init);

// 25.9.11, adjacent difference
template<class InputIterator, class OutputIterator>
  OutputIterator adjacent_difference(InputIterator first,
                                     InputIterator last,
                                     OutputIterator result);
template<class InputIterator, class OutputIterator, class BinaryOperation>
  OutputIterator adjacent_difference(InputIterator first,
                                     InputIterator last,
                                     OutputIterator result,
                                     BinaryOperation binary_op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 adjacent_difference(ExecutionPolicy&& exec, // see 25.3.5
                                       ForwardIterator1 first,
                                       ForwardIterator1 last,
                                       ForwardIterator2 result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryOperation>
  ForwardIterator2 adjacent_difference(ExecutionPolicy&& exec, // see 25.3.5
                                       ForwardIterator1 first,
                                       ForwardIterator1 last,
                                       ForwardIterator2 result,
                                       BinaryOperation binary_op);

// 25.9.12, iota
template<class ForwardIterator, class T>
  void iota(ForwardIterator first, ForwardIterator last, T value);

// 25.9.13, greatest common divisor
template<class M, class N>
  constexpr common_type_t<M,N> gcd(M m, N n);

// 25.9.14, least common multiple
template<class M, class N>
  constexpr common_type_t<M,N> lcm(M m, N n);

// 25.9.15, midpoint
template<class T>
  constexpr T midpoint(T a, T b) noexcept;
```

```
  template<class T>
    constexpr T* midpoint(T* a, T* b);
}
```

25.9 Generalized numeric operations [numeric.ops]

1 [*Note*: The use of closed ranges as well as semi-open ranges to specify requirements throughout this subclause is intentional. — *end note*]

25.9.1 Definitions [numerics.defns]

1 Define *GENERALIZED_NONCOMMUTATIVE_SUM*(op, a1, ..., aN) as follows:

(1.1) — a1 when N is 1, otherwise

(1.2) — op(*GENERALIZED_NONCOMMUTATIVE_SUM*(op, a1, ..., aK), *GENERALIZED_NONCOMMUTATIVE_SUM*(op, aM, ..., aN)) for any K where $1 < K + 1 = M \leq N$.

2 Define *GENERALIZED_SUM*(op, a1, ..., aN) as *GENERALIZED_NONCOMMUTATIVE_SUM*(op, b1, ..., bN), where b1, ..., bN may be any permutation of a1, ..., aN.

25.9.2 Accumulate [accumulate]

```
template<class InputIterator, class T>
  T accumulate(InputIterator first, InputIterator last, T init);
template<class InputIterator, class T, class BinaryOperation>
  T accumulate(InputIterator first, InputIterator last, T init,
               BinaryOperation binary_op);
```

1 ~~*Requires:*~~ *Expects:* T ~~shall satisfy~~meets the *Cpp17CopyConstructible* (Table **??**) and *Cpp17CopyAssignable* (Table **??**) requirements. In the range [first, last), binary_op ~~shall neither modify elements nor invalidate iterators or subranges~~neither modifies elements nor invalidates iterators or subranges.[239]

2 *Effects:* Computes its result by initializing the accumulator acc with the initial value init and then modifies it with acc = std::move(acc) + *i or acc = binary_op(std::move(acc), *i) for every iterator i in the range [first, last) in order.[240]

25.9.3 Reduce [reduce]

```
template<class InputIterator>
  typename iterator_traits<InputIterator>::value_type
    reduce(InputIterator first, InputIterator last);
```

1 *Effects:* Equivalent to:

```
return reduce(first, last,
              typename iterator_traits<InputIterator>::value_type{});
```

```
template<class ExecutionPolicy, class ForwardIterator>
  typename iterator_traits<ForwardIterator>::value_type
    reduce(ExecutionPolicy&& exec,
           ForwardIterator first, ForwardIterator last);
```

2 *Effects:* Equivalent to:

```
return reduce(std::forward<ExecutionPolicy>(exec), first, last,
              typename iterator_traits<ForwardIterator>::value_type{});
```

```
template<class InputIterator, class T>
  T reduce(InputIterator first, InputIterator last, T init);
```

3 *Effects:* Equivalent to:

```
return reduce(first, last, init, plus<>());
```

239) The use of fully closed ranges is intentional.

240) accumulate is similar to the APL reduction operator and Common Lisp reduce function, but it avoids the difficulty of defining the result of reduction on an empty sequence by always requiring an initial value.

```
template<class ExecutionPolicy, class ForwardIterator, class T>
  T reduce(ExecutionPolicy&& exec,
           ForwardIterator first, ForwardIterator last, T init);
```

4 *Effects:* Equivalent to:

```
return reduce(std::forward<ExecutionPolicy>(exec), first, last, init, plus<>());
```

```
template<class InputIterator, class T, class BinaryOperation>
  T reduce(InputIterator first, InputIterator last, T init,
           BinaryOperation binary_op);
template<class ExecutionPolicy, class ForwardIterator, class T, class BinaryOperation>
  T reduce(ExecutionPolicy&& exec,
           ForwardIterator first, ForwardIterator last, T init,
           BinaryOperation binary_op);
```

5 *Mandates:* All of `binary_op(init, *first)`, `binary_op(*first, init)`, `binary_op(init, init)`, and `binary_op(*first, *first)` are convertible to `T`.

6 ~~*Requires:*~~ *Expects:*

(6.1) — `T` ~~shall be~~is *Cpp17MoveConstructible* (Table **??**).

(6.2) — All of `binary_op(init, *first)`, `binary_op(*first, init)`, `binary_op(init, init)`, and `binary_op(*first, *first)` shall be convertible to `T`.

(6.3) — `binary_op` ~~shall neither invalidate~~neither invalidates iterators or subranges, nor ~~modify~~modifies elements in the range `[first, last]`.

7 *Returns:* *`GENERALIZED_SUM`*`(binary_op, init, *i, ...)` for every `i` in `[first, last)`.

8 *Complexity:* $\mathcal{O}$(`last - first`) applications of `binary_op`.

9 [*Note*: The difference between `reduce` and `accumulate` is that `reduce` applies `binary_op` in an unspecified order, which yields a nondeterministic result for non-associative or non-commutative `binary_op` such as floating-point addition. — *end note*]

25.9.4 Inner product [inner.product]

```
template<class InputIterator1, class InputIterator2, class T>
  T inner_product(InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, T init);
template<class InputIterator1, class InputIterator2, class T,
         class BinaryOperation1, class BinaryOperation2>
  T inner_product(InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, T init,
                  BinaryOperation1 binary_op1,
                  BinaryOperation2 binary_op2);
```

1 ~~*Requires:*~~ *Expects:* `T` ~~shall satisfy~~meets the *Cpp17CopyConstructible* (Table **??**) and *Cpp17CopyAssignable* (Table **??**) requirements. In the ranges `[first1, last1]` and `[first2, first2 + (last1 - first1)]` `binary_op1` and `binary_op2` ~~shall neither modify elements nor invalidate iterators or subranges~~neither modifies elements nor invalidates iterators or subranges.[241]

2 *Effects:* Computes its result by initializing the accumulator `acc` with the initial value `init` and then modifying it with `acc = std::move(acc) + (*i1) * (*i2)` or `acc = binary_op1(std::move(acc), binary_op2(*i1, *i2))` for every iterator `i1` in the range `[first1, last1)` and iterator `i2` in the range `[first2, first2 + (last1 - first1))` in order.

25.9.5 Transform reduce [transform.reduce]

```
template<class InputIterator1, class InputIterator2, class T>
  T transform_reduce(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2,
                     T init);
```

1 *Effects:* Equivalent to:

```
return transform_reduce(first1, last1, first2, init, plus<>(), multiplies<>());
```

241) The use of fully closed ranges is intentional.

```
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2, class T>
  T transform_reduce(ExecutionPolicy&& exec,
                     ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2,
                     T init);
```

2 *Effects:* Equivalent to:

```
return transform_reduce(std::forward<ExecutionPolicy>(exec),
                        first1, last1, first2, init, plus<>(), multiplies<>());
```

```
template<class InputIterator1, class InputIterator2, class T,
         class BinaryOperation1, class BinaryOperation2>
  T transform_reduce(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2,
                     T init,
                     BinaryOperation1 binary_op1,
                     BinaryOperation2 binary_op2);
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2, class T,
         class BinaryOperation1, class BinaryOperation2>
  T transform_reduce(ExecutionPolicy&& exec,
                     ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2,
                     T init,
                     BinaryOperation1 binary_op1,
                     BinaryOperation2 binary_op2);
```

3 *Mandates:* All of

(3.1) — `binary_op1(init, init)`,

(3.2) — `binary_op1(init, binary_op2(*first1, *first2))`,

(3.3) — `binary_op1(binary_op2(*first1, *first2), init)`, and

(3.4) — `binary_op1(binary_op2(*first1, *first2), binary_op2(*first1, *first2))`

are convertible to `T`.

4 ~~*Requires:*~~ *Expects:*

(4.1) — `T` ~~shall be~~is *Cpp17MoveConstructible* (Table **??**).

(4.2) — All of

(4.2.1) — `binary_op1(init, init)`,

(4.2.2) — `binary_op1(init, binary_op2(*first1, *first2))`,

(4.2.3) — `binary_op1(binary_op2(*first1, *first2), init)`, and

(4.2.4) — `binary_op1(binary_op2(*first1, *first2), binary_op2(*first1, *first2))`

shall be convertible to `T`.

(4.3) — Neither `binary_op1` nor `binary_op2` ~~shall invalidate~~invalidates subranges, nor ~~modify~~modifies elements in the ranges `[first1, last1]` and `[first2, first2 + (last1 - first1)]`.

5 *Returns:*

```
GENERALIZED_SUM(binary_op1, init, binary_op2(*i, *(first2 + (i - first1))), ...)
```

for every iterator `i` in `[first1, last1)`.

6 *Complexity:* $\mathcal{O}$(`last1 - first1`) applications each of `binary_op1` and `binary_op2`.

```
template<class InputIterator, class T,
         class BinaryOperation, class UnaryOperation>
  T transform_reduce(InputIterator first, InputIterator last, T init,
                     BinaryOperation binary_op, UnaryOperation unary_op);
template<class ExecutionPolicy,
         class ForwardIterator, class T,
         class BinaryOperation, class UnaryOperation>
```

```
T transform_reduce(ExecutionPolicy&& exec,
                   ForwardIterator first, ForwardIterator last,
                   T init, BinaryOperation binary_op, UnaryOperation unary_op);
```

7 *Mandates:* All of

(7.1) — `binary_op(init, init)`,

(7.2) — `binary_op(init, unary_op(*first))`,

(7.3) — `binary_op(unary_op(*first), init)`, and

(7.4) — `binary_op(unary_op(*first), unary_op(*first))`

are convertible to `T`.

8 ~~*Requires:*~~ *Expects:*

(8.1) — `T` ~~shall be~~is *Cpp17MoveConstructible* (Table **??**).

(8.2) — All of

(8.2.1) — `binary_op(init, init)`,

(8.2.2) — `binary_op(init, unary_op(*first))`,

(8.2.3) — `binary_op(unary_op(*first), init)`, and

(8.2.4) — `binary_op(unary_op(*first), unary_op(*first))`

shall be convertible to `T`.

(8.3) — Neither `unary_op` nor `binary_op` ~~shall invalidate~~invalidates subranges, nor ~~modify~~modifies elements in the range `[first, last]`.

9 *Returns:*

GENERALIZED_SUM`(binary_op, init, unary_op(*i), ...)`

for every iterator `i` in `[first, last)`.

10 *Complexity:* $\mathcal{O}$(`last - first`) applications each of `unary_op` and `binary_op`.

11 [*Note*: `transform_reduce` does not apply `unary_op` to `init`. — *end note*]

25.9.6 Partial sum [partial.sum]

```
template<class InputIterator, class OutputIterator>
  OutputIterator partial_sum(
    InputIterator first, InputIterator last,
    OutputIterator result);
template<class InputIterator, class OutputIterator, class BinaryOperation>
  OutputIterator partial_sum(
    InputIterator first, InputIterator last,
    OutputIterator result, BinaryOperation binary_op);
```

1 *Mandates:* `InputIterator`'s value type is constructible from the type of `*first`. The result of the expression `std::move(acc) + *i` or `binary_op(std::move(acc), *i)` is implicitly convertible to `InputIterator`'s value type.

2 ~~*Requires:*~~ *Expects:* ~~`InputIterator`'s value type shall be constructible from the type of `*first`. The result of the expression `std::move(acc) + *i` or `binary_op(std::move(acc), *i)` shall be implicitly convertible to `InputIterator`'s value type.~~ `acc` ~~shall be~~is writable (**??**) to the `result` output iterator. In the ranges `[first, last]` and `[result, result + (last - first)]` `binary_op` ~~shall neither modify elements nor invalidate iterators or subranges~~neither modifies elements nor invalidates iterators or subranges.[242]

3 *Effects:* For a non-empty range, the function creates an accumulator `acc` whose type is `InputIterator`'s value type, initializes it with `*first`, and assigns the result to `*result`. For every iterator `i` in `[first + 1, last)` in order, `acc` is then modified by `acc = std::move(acc) + *i` or `acc = binary_op(std::move(acc), *i)` and the result is assigned to `*(result + (i - first))`.

4 *Returns:* `result + (last - first)`.

242) The use of fully closed ranges is intentional.

5 *Complexity:* Exactly `(last - first) - 1` applications of the binary operation.

6 *Remarks:* `result` may be equal to `first`.

25.9.7 Exclusive scan [exclusive.scan]

```
template<class InputIterator, class OutputIterator, class T>
  OutputIterator exclusive_scan(InputIterator first, InputIterator last,
                                OutputIterator result, T init);
```

1 *Effects:* Equivalent to:

```
return exclusive_scan(first, last, result, init, plus<>());
```

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T>
  ForwardIterator2 exclusive_scan(ExecutionPolicy&& exec,
                                  ForwardIterator1 first, ForwardIterator1 last,
                                  ForwardIterator2 result, T init);
```

2 *Effects:* Equivalent to:

```
return exclusive_scan(std::forward<ExecutionPolicy>(exec),
                      first, last, result, init, plus<>());
```

```
template<class InputIterator, class OutputIterator, class T, class BinaryOperation>
  OutputIterator exclusive_scan(InputIterator first, InputIterator last,
                                OutputIterator result, T init, BinaryOperation binary_op);
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2, class T, class BinaryOperation>
  ForwardIterator2 exclusive_scan(ExecutionPolicy&& exec,
                                  ForwardIterator1 first, ForwardIterator1 last,
                                  ForwardIterator2 result, T init, BinaryOperation binary_op);
```

3 *Mandates:* All of `binary_op(init, init)`, `binary_op(init, *first)`, and `binary_op(*first, *first)` are convertible to `T`.

4 ~~*Requires:*~~ *Expects:*

(4.1) — `T` ~~shall be~~is *Cpp17MoveConstructible* (Table **??**).

(4.2) — All of `binary_op(init, init)`, `binary_op(init, *first)`, and `binary_op(*first, *first)` shall be convertible to `T`.

(4.3) — `binary_op` ~~shall neither invalidate~~neither invalidates iterators or subranges, nor ~~modify~~modifies elements in the ranges `[first, last]` or `[result, result + (last - first)]`.

5 *Effects:* For each integer `K` in `[0, last - first)` assigns through `result + K` the value of:

```
GENERALIZED_NONCOMMUTATIVE_SUM(
    binary_op, init, *(first + 0), *(first + 1), ..., *(first + K - 1))
```

6 *Returns:* The end of the resulting range beginning at `result`.

7 *Complexity:* $\mathcal{O}$(`last - first`) applications of `binary_op`.

8 *Remarks:* `result` may be equal to `first`.

9 [*Note*: The difference between `exclusive_scan` and `inclusive_scan` is that `exclusive_scan` excludes the i^{th} input element from the i^{th} sum. If `binary_op` is not mathematically associative, the behavior of `exclusive_scan` may be nondeterministic. — *end note*]

25.9.8 Inclusive scan [inclusive.scan]

```
template<class InputIterator, class OutputIterator>
  OutputIterator inclusive_scan(InputIterator first, InputIterator last, OutputIterator result);
```

1 *Effects:* Equivalent to:

```
return inclusive_scan(first, last, result, plus<>());
```

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 inclusive_scan(ExecutionPolicy&& exec,
                                  ForwardIterator1 first, ForwardIterator1 last,
                                  ForwardIterator2 result);
```

2 *Effects:* Equivalent to:

```
return inclusive_scan(std::forward<ExecutionPolicy>(exec), first, last, result, plus<>());
```

```
template<class InputIterator, class OutputIterator, class BinaryOperation>
  OutputIterator inclusive_scan(InputIterator first, InputIterator last,
                                OutputIterator result, BinaryOperation binary_op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryOperation>
  ForwardIterator2 inclusive_scan(ExecutionPolicy&& exec,
                                  ForwardIterator1 first, ForwardIterator1 last,
                                  ForwardIterator2 result, BinaryOperation binary_op);

template<class InputIterator, class OutputIterator, class BinaryOperation, class T>
  OutputIterator inclusive_scan(InputIterator first, InputIterator last,
                                OutputIterator result, BinaryOperation binary_op, T init);
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2, class BinaryOperation, class T>
  ForwardIterator2 inclusive_scan(ExecutionPolicy&& exec,
                                  ForwardIterator1 first, ForwardIterator1 last,
                                  ForwardIterator2 result, BinaryOperation binary_op, T init);
```

3 *Mandates:* If `init` is provided, all of `binary_op(init, init)`, `binary_op(init, *first)`, and `binary_op(*first, *first)` are convertible to `T`; otherwise, `binary_op(*first, *first)` is convertible to `ForwardIterator1`'s value type.

4 ~~*Requires:*~~ *Expects:*

(4.1) — If `init` is provided, `T` ~~shall be~~is *Cpp17MoveConstructible* (Table **??**); otherwise, `ForwardIterator1`'s value type ~~shall be~~is *Cpp17MoveConstructible*.

(4.2) — If `init` is provided, all of `binary_op(init, init)`, `binary_op(init, *first)`, and `binary_op(*first, *first)` shall be convertible to `T`; otherwise, `binary_op(*first, *first)` shall be convertible to `ForwardIterator1`'s value type.

(4.3) — `binary_op` ~~shall neither invalidate~~neither invalidates iterators or subranges, nor ~~modify~~modifies elements in the ranges `[first, last]` or `[result, result + (last - first)]`.

5 *Effects:* For each integer `K` in `[0, last - first)` assigns through `result + K` the value of

(5.1) — *GENERALIZED_NONCOMMUTATIVE_SUM*(
`binary_op, init, *(first + 0), *(first + 1), ..., *(first + K)`)
if `init` is provided, or

(5.2) — *GENERALIZED_NONCOMMUTATIVE_SUM*(
`binary_op, *(first + 0), *(first + 1), ..., *(first + K)`)
otherwise.

6 *Returns:* The end of the resulting range beginning at `result`.

7 *Complexity:* $\mathcal{O}$(`last - first`) applications of `binary_op`.

8 *Remarks:* `result` may be equal to `first`.

9 [*Note*: The difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the i^{th} input element in the i^{th} sum. If `binary_op` is not mathematically associative, the behavior of `inclusive_scan` may be nondeterministic. — *end note*]

25.9.9 Transform exclusive scan [transform.exclusive.scan]

```
template<class InputIterator, class OutputIterator, class T,
         class BinaryOperation, class UnaryOperation>
  OutputIterator transform_exclusive_scan(InputIterator first, InputIterator last,
                                          OutputIterator result, T init,
                                          BinaryOperation binary_op,UnaryOperation unary_op);
```

```
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2, class T,
         class BinaryOperation, class UnaryOperation>
  ForwardIterator2 transform_exclusive_scan(ExecutionPolicy&& exec,
                                            ForwardIterator1 first, ForwardIterator1 last,
                                            ForwardIterator2 result, T init,
                                            BinaryOperation binary_op, UnaryOperation unary_op);
```

1 *Mandates:* All of

(1.1) — `binary_op(init, init)`,

(1.2) — `binary_op(init, unary_op(*first))`, and

(1.3) — `binary_op(unary_op(*first), unary_op(*first))`

are be convertible to `T`.

2 ~~*Requires:*~~ *Expects:*

(2.1) — `T` ~~shall be~~is *Cpp17MoveConstructible* (Table **??**).

(2.2) — All of

(2.2.1) — `binary_op(init, init)`,

(2.2.2) — `binary_op(init, unary_op(*first))`, and

(2.2.3) — `binary_op(unary_op(*first), unary_op(*first))`

shall be convertible to `T`.

(2.3) — Neither `unary_op` nor `binary_op` ~~shall invalidate~~invalidates iterators or subranges, nor ~~modify~~modifies elements in the ranges `[first, last]` or `[result, result + (last - first)]`.

3 *Effects:* For each integer `K` in `[0, last - first)` assigns through `result + K` the value of:

```
GENERALIZED_NONCOMMUTATIVE_SUM(
    binary_op, init,
    unary_op(*(first + 0)), unary_op(*(first + 1)), ..., unary_op(*(first + K - 1)))
```

4 *Returns:* The end of the resulting range beginning at `result`.

5 *Complexity:* $\mathcal{O}$(`last - first`) applications each of `unary_op` and `binary_op`.

6 *Remarks:* `result` may be equal to `first`.

7 [*Note*: The difference between `transform_exclusive_scan` and `transform_inclusive_scan` is that `transform_exclusive_scan` excludes the i^{th} input element from the i^{th} sum. If `binary_op` is not mathematically associative, the behavior of `transform_exclusive_scan` may be nondeterministic. `transform_exclusive_scan` does not apply `unary_op` to `init`. — *end note*]

25.9.10 Transform inclusive scan [transform.inclusive.scan]

```
template<class InputIterator, class OutputIterator,
         class BinaryOperation, class UnaryOperation>
  OutputIterator transform_inclusive_scan(InputIterator first, InputIterator last,
                                          OutputIterator result,
                                          BinaryOperation binary_op, UnaryOperation unary_op);
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2,
         class BinaryOperation, class UnaryOperation>
  ForwardIterator2 transform_inclusive_scan(ExecutionPolicy&& exec,
                                            ForwardIterator1 first, ForwardIterator1 last,
                                            ForwardIterator2 result,
                                            BinaryOperation binary_op, UnaryOperation unary_op);
template<class InputIterator, class OutputIterator,
         class BinaryOperation, class UnaryOperation, class T>
  OutputIterator transform_inclusive_scan(InputIterator first, InputIterator last,
                                          OutputIterator result,
                                          BinaryOperation binary_op, UnaryOperation unary_op,
                                          T init);
```

```
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2,
         class BinaryOperation, class UnaryOperation, class T>
  ForwardIterator2 transform_inclusive_scan(ExecutionPolicy&& exec,
                                            ForwardIterator1 first, ForwardIterator1 last,
                                            ForwardIterator2 result,
                                            BinaryOperation binary_op, UnaryOperation unary_op,
                                            T init);
```

1 *Mandates:* If `init` is provided, all of

(1.1) — `binary_op(init, init)`,

(1.2) — `binary_op(init, unary_op(*first))`, and

(1.3) — `binary_op(unary_op(*first), unary_op(*first))`

are convertible to `T`; otherwise, `binary_op(unary_op(*first), unary_op(*first))` are convertible to `ForwardIterator1`'s value type.

2 ~~*Requires:*~~ *Expects:*

(2.1) — If `init` is provided, `T` ~~shall be~~is *Cpp17MoveConstructible* (Table **??**); otherwise, `ForwardIterator1`'s value type ~~shall be~~is *Cpp17MoveConstructible*.

(2.2) — If `init` is provided, all of

(2.2.1) — `binary_op(init, init)`,

(2.2.2) — `binary_op(init, unary_op(*first))`, and

(2.2.3) — `binary_op(unary_op(*first), unary_op(*first))`

shall be convertible to `T`; otherwise, `binary_op(unary_op(*first), unary_op(*first))` shall be convertible to `ForwardIterator1`'s value type.

(2.3) — Neither `unary_op` nor `binary_op` ~~shall invalidate~~invalidates iterators or subranges, nor ~~modify~~modifies elements in the ranges `[first, last]` or `[result, result + (last - first)]`.

3 *Effects:* For each integer `K` in `[0, last - first)` assigns through `result + K` the value of

(3.1) — *GENERALIZED_NONCOMMUTATIVE_SUM*(
```
    binary_op, init,
    unary_op(*(first + 0)), unary_op(*(first + 1)), ..., unary_op(*(first + K)))
```
if `init` is provided, or

(3.2) — *GENERALIZED_NONCOMMUTATIVE_SUM*(
```
    binary_op,
    unary_op(*(first + 0)), unary_op(*(first + 1)), ..., unary_op(*(first + K)))
```
otherwise.

4 *Returns:* The end of the resulting range beginning at `result`.

5 *Complexity:* $\mathcal{O}$(`last - first`) applications each of `unary_op` and `binary_op`.

6 *Remarks:* `result` may be equal to `first`.

7 [*Note*: The difference between `transform_exclusive_scan` and `transform_inclusive_scan` is that `transform_inclusive_scan` includes the i^{th} input element in the i^{th} sum. If `binary_op` is not mathematically associative, the behavior of `transform_inclusive_scan` may be nondeterministic. `transform_inclusive_scan` does not apply `unary_op` to `init`. — *end note*]

25.9.11 Adjacent difference [adjacent.difference]

```
template<class InputIterator, class OutputIterator>
  OutputIterator
    adjacent_difference(InputIterator first, InputIterator last, OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2
    adjacent_difference(ExecutionPolicy&& exec,
                        ForwardIterator1 first, ForwardIterator1 last, ForwardIterator2 result);
```

```
template<class InputIterator, class OutputIterator, class BinaryOperation>
  OutputIterator
    adjacent_difference(InputIterator first, InputIterator last,
                        OutputIterator result, BinaryOperation binary_op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryOperation>
  ForwardIterator2
    adjacent_difference(ExecutionPolicy&& exec,
                        ForwardIterator1 first, ForwardIterator1 last,
                        ForwardIterator2 result, BinaryOperation binary_op);
```

1 Let `T` be the value type of `decltype(first)`. For the overloads that do not take an argument `binary_op`, let `binary_op` be an lvalue that denotes an object of type `minus<>`.

2 ~~*Requires:*~~ *Expects:*

(2.1) — For the overloads with no `ExecutionPolicy`, `T` ~~shall be~~is *Cpp17MoveAssignable* (Table **??**) and ~~shall be~~is constructible from the type of `*first`. `acc` (defined below) ~~shall be~~is writable (**??**) to the `result` output iterator. The result of the expression `binary_op(val, std::move(acc))` ~~shall be~~is writable to the `result` output iterator.

(2.2) — For the overloads with an `ExecutionPolicy`, the result of the expressions `binary_op(*first, *first)` and `*first` ~~shall be~~is writable to `result`.

(2.3) — For all overloads, in the ranges `[first, last]` and `[result, result + (last - first)]`, `binary_op` ~~shall neither modify~~neither modifies elements nor invalidate iterators or subranges.[243]

3 *Effects:* For the overloads with no `ExecutionPolicy` and a non-empty range, the function creates an accumulator `acc` of type `T`, initializes it with `*first`, and assigns the result to `*result`. For every iterator `i` in `[first + 1, last)` in order, creates an object `val` whose type is `T`, initializes it with `*i`, computes `binary_op(val, std::move(acc))`, assigns the result to `*(result + (i - first))`, and move assigns from `val` to `acc`.

4 For the overloads with an `ExecutionPolicy` and a non-empty range, performs `*result = *first`. Then, for every `d` in `[1, last - first - 1]`, performs `*(result + d) = binary_op(*(first + d), *(first + (d - 1)))`.

5 *Returns:* `result + (last - first)`.

6 *Complexity:* Exactly `(last - first) - 1` applications of the binary operation.

7 *Remarks:* For the overloads with no `ExecutionPolicy`, `result` may be equal to `first`. For the overloads with an `ExecutionPolicy`, the ranges `[first, last)` and `[result, result + (last - first))` shall not overlap.

25.9.12 Iota [numeric.iota]

```
template<class ForwardIterator, class T>
  void iota(ForwardIterator first, ForwardIterator last, T value);
```

1 ~~*Requires:*~~ *Mandates:* `T` ~~shall be~~is convertible to `ForwardIterator`'s value type. The expression `++val`, where `val` has type `T`, ~~shall be~~is well-formed.

2 *Effects:* For each element referred to by the iterator `i` in the range `[first, last)`, assigns `*i = value` and increments `value` as if by `++value`.

3 *Complexity:* Exactly `last - first` increments and assignments.

25.9.13 Greatest common divisor [numeric.ops.gcd]

```
template<class M, class N>
  constexpr common_type_t<M,N> gcd(M m, N n);
```

1 ~~*Requires:*~~ *Expects:* |`m`| and |`n`| ~~shall be~~are representable as a value of `common_type_t<M, N>`. [*Note*: These requirements ensure, for example, that `gcd(m, m)` = |`m`| is representable as a value of type `M`. — *end note*]

2 *Remarks:* If either `M` or `N` is not an integer type, or if either is *cv* `bool`, the program is ill-formed.

243) The use of fully closed ranges is intentional.

3 *Returns:* Zero when `m` and `n` are both zero. Otherwise, returns the greatest common divisor of |`m`| and |`n`|.

4 *Throws:* Nothing.

25.9.14 Least common multiple [numeric.ops.lcm]

```
template<class M, class N>
  constexpr common_type_t<M,N> lcm(M m, N n);
```

1 ~~*Requires:*~~ *Expects:* |`m`| and |`n`| ~~shall be~~are representable as a value of `common_type_t<M, N>`. The least common multiple of |`m`| and |`n`| shall be representable as a value of type `common_type_t<M,N>`.

2 *Remarks:* If either `M` or `N` is not an integer type, or if either is *cv* `bool` the program is ill-formed.

3 *Returns:* Zero when either `m` or `n` is zero. Otherwise, returns the least common multiple of |`m`| and |`n`|.

4 *Throws:* Nothing.

25.9.15 Midpoint [numeric.ops.midpoint]

```
template<class T>
  constexpr T midpoint(T a, T b) noexcept;
```

1 *Constraints:* `T` is an arithmetic type other than `bool`.

2 *Returns:* Half the sum of `a` and `b`. If `T` is an integer type and the sum is odd, the result is rounded towards `a`.

3 *Remarks:* No overflow occurs. If `T` is a floating-point type, at most one inexact operation occurs.

```
template<class T>
  constexpr T* midpoint(T* a, T* b);
```

4 *Constraints:* `T` is a complete object type.

5 *Expects:* `a` and `b` point to, respectively, elements `x`[i] and `x`[j] of the same array object `x`. [*Note*: An object that is not an array element is considered to belong to a single-element array for this purpose; see **??**. A pointer past the last element of an array `x` of n elements is considered to be equivalent to a pointer to a hypothetical element `x`[n] for this purpose; see **??**. — *end note*]

6 *Returns:* A pointer to `x`[$i + \frac{j-i}{2}$], where the result of the division is truncated towards zero.

25.10 C library algorithms [alg.c.library]

1 [*Note*: The header `<cstdlib>` (**??**) declares the functions described in this subclause. — *end note*]

```
void* bsearch(const void* key, const void* base, size_t nmemb, size_t size,
              c-compare-pred* compar);
void* bsearch(const void* key, const void* base, size_t nmemb, size_t size,
              compare-pred* compar);
void qsort(void* base, size_t nmemb, size_t size, c-compare-pred* compar);
void qsort(void* base, size_t nmemb, size_t size, compare-pred* compar);
```

2 *Effects:* These functions have the semantics specified in the C standard library.

3 *Remarks:* The behavior is undefined unless the objects in the array pointed to by `base` are of trivial type.

4 *Throws:* Any exception thrown by `compar()` (**??**).

See also: ISO C 7.22.5.