Concepts for the C++0x Standard Library: Algorithms (Revision 3)

Authors: Douglas Gregor, Indiana University
           Mat Marcus, Adobe Systems, Inc.
           Thomas Witt, Zephyr Associates, Inc.
           Andrew Lumsdaine, Indiana University
Document number: N2696=08-0206
Revises document number: N2625=08-0135
Date: 2008-06-30
Project: Programming Language C++, Library Working Group
Reply-to: Douglas Gregor <dgregor@osl.iu.edu>

Introduction

This document proposes changes to Chapter 25 of the C++ Standard Library in order to make full use of concepts [1]. We make every attempt to provide complete backward compatibility with the pre-concept Standard Library, and note each place where we have knowingly changed semantics.

This document is formatted in the same manner as the latest working draft of the C++ standard (N2588). Future versions of this document will track the working draft and the concepts proposal as they evolve. Wherever the numbering of a (sub)section matches a section of the working paper, the text in this document should be considered replacement text, unless editorial comments state otherwise. All editorial comments will have a gray background . Changes to the replacement text are categorized and typeset as additions, ~~removals~~, or ~~changes~~modifications.

Changes from N2625

— Replace uses of `Integral` with `IntegralLike`.

— Changed several iterator `value_type` associated types to `reference` associated types, in the following algorithms:

    — `remove_copy`, `remove_copy_if`

    — `unique_copy`

    — `reverse_copy`

    — `replace_copy`, `replace_copy_if`

— Changed `adjacent_find`'s `HasEqualTo` requirement to use `EqualityComparable`.

— Fixed types in the parameter types of `mismatch`

— Removed every use of the `Mutable` iterator concepts, which no longer exist.

— Removed the `SameType` constraints from `merge`, `includes`, `set_union`, `set_intersection`, `set_differ-ence`, and `set_symmetric_difference`, to match C++03 requirements.

— Introduce the use of the `ShuffleIterator` concept, for algorithms that require the ability to move elements between the sequence and a variable of the iterator `value_type` (e.g., `sort`).

Chapter 25   Algorithms library [algorithms]

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

2   The following subclauses describe components for non-modifying sequence operation, modifying sequence operations, sorting and related operations, and algorithms from the ISO C library, as summarized in Table 1.

<div align="center">

Table 1: Algorithms library summary

Subclause	Header(s)
25.1 Non-modifying sequence operations	
25.2 Mutating sequence operations	`<algorithm>`
25.3 Sorting and related operations	
[alg.c.library] C library algorithms	`<cstdlib>`

</div>

Header `<algorithm>` synopsis

```
namespace std {
  // 25.1, non-modifying sequence operations:
  template<InputIterator Iter, Callable<auto, Iter::reference> Function>
    requires CopyConstructible<Function>
    Function for_each(Iter first, Iter last, Function f);
  template<InputIterator Iter, class T>
    requires HasEqualTo<Iter::referencevalue_type, T>
    Iter find(Iter first, Iter last, const T& value);
  template<InputIterator Iter, Predicate<auto, Iter::referencevalue_type> Pred>
    requires CopyConstructible<Pred>
    Iter find_if(Iter first, Iter last, Pred pred);
  template<ForwardIterator Iter1, ForwardIterator Iter2>
    requires HasEqualTo<Iter1::referencevalue_type, Iter2::referencevalue_type>
    Iter1 find_end(Iter1 first1, Iter1 last1,
                   Iter2 first2, Iter2 last2);
  template<ForwardIterator Iter1, ForwardIterator Iter2,
           Predicate<auto, Iter1::referencevalue_type, Iter2::referencevalue_type> Pred>
    requires CopyConstructible<Pred>
    Iter1 find_end(Iter1 first1, Iter1 last1,
                   Iter2 first2, Iter2 last2,
                   Pred pred);
```

```cpp
template<InputIterator Iter1, ForwardIterator Iter2>
  requires HasEqualTo<Iter1::reference value_type, Iter2::reference value_type>
  Iter1 find_first_of(Iter1 first1, Iter1 last1,
                      Iter2 first2, Iter2 last2);
template<InputIterator Iter1, ForwardIterator Iter2,
         Predicate<auto, Iter1::reference value_type, Iter2::reference value_type> Pred>
  requires CopyConstructible<Pred>
  Iter1 find_first_of(Iter1 first1, Iter1 last1,
                      Iter2 first2, Iter2 last2,
                      Pred pred);

template<ForwardIterator Iter>
  requires EqualityComparable<Iter::reference value_type>
  Iter adjacent_find(Iter first, Iter last);
template<ForwardIterator Iter,
         Predicate<auto, Iter::reference value_type, Iter::reference value_type> Pred>
  requires CopyConstructible<Pred>
  Iter adjacent_find(Iter first, Iter last, Pred pred);

template<InputIterator Iter, class T>
  requires HasEqualTo<Iter::reference value_type, T>
  Iter::difference_type count(Iter first, Iter last, const T& value);
template<InputIterator Iter, Predicate<auto, Iter::reference value_type> Pred>
  requires CopyConstructible<Pred>
  Iter::difference_type count_if(Iter first, Iter last, Pred pred);

template<InputIterator Iter1, InputIterator Iter2>
  requires HasEqualTo<Iter1::reference value_type, Iter2::reference value_type>
  pair<Iter1, Iter2> mismatch(Iter1 first1, Iter1 last1,
                              Iter2 first2);
template<InputIterator Iter1, InputIterator Iter2,
         Predicate<auto, Iter1::reference value_type, Iter2::reference value_type> Pred>
  requires CopyConstructible<Pred>
  pair<Iter1, Iter2> mismatch(Iter1 first1, Iter1 last1,
                              Iter2 first2, Pred pred);

template<InputIterator Iter1, InputIterator Iter2>
  requires HasEqualTo<Iter1::reference value_type, Iter2::reference value_type>
  bool equal(Iter1 first1, Iter1 last1,
             Iter2 first2);
template<InputIterator Iter1, InputIterator Iter2,
         Predicate<auto, Iter1::reference value_type, Iter2::reference value_type> Pred>
  requires CopyConstructible<Pred>
  bool equal(Iter1 first1, Iter1 last1,
             Iter2 first2, Pred pred);

template<ForwardIterator Iter1, ForwardIterator Iter2>
  requires HasEqualTo<Iter1::reference value_type, Iter2::reference value_type>
  Iter1 search(Iter1 first1, Iter1 last1,
               Iter2 first2, Iter2 last2);
```

```
template<ForwardIterator Iter1, ForwardIterator Iter2,
         Predicate<auto, Iter1::referencevalue_type, Iter2::referencevalue_type> Pred>
  requires CopyConstructible<Pred>
  Iter1 search(Iter1 first1, Iter1 last1,
               Iter2 first2, Iter2 last2,
               Pred pred);
template<ForwardIterator Iter, class T>
  requires HasEqualTo<Iter::referencevalue_type, T>
  Iter search_n(Iter first, Iter last, Iter::difference_type count,
                const T& value);
template<ForwardIterator Iter, class T,
         Predicate<auto, Iter::referencevalue_type, T> Pred>
  requires CopyConstructible<Pred>
  Iter search_n(Iter first, Iter last, Iter::difference_type count,
                const T& value, Pred pred);

// 25.2, modifying sequence operations:
// 25.2.1, copy:
template<InputIterator InIter, OutputIterator<auto, InIter::reference> OutIter>
  OutIter copy(InIter first, InIter last,
               OutIter result);
template<BidirectionalIterator InIter, MutableBidirectionalIterator OutIter>
  requires HasCopyAssignOutputIterator<OutIter::reference, InIter::reference>
  OutIter copy_backward(InIter first, InIter last,
                        OutIter result);

// 25.2.2, move:
template<InputIterator InIter, OutputIterator<auto, InIter::value_type&&>typename OutIter>
  requires HasStdMove<InIter::reference>
        && OutputIterator<OutIter, HasStdMove<InIter::reference>::result_type>
  OutIter move(InIter first, InIter last,
               OutIter result);
template<BidirectionalIterator InIter, BidirectionalIterator OutIter>
  requires HasMoveAssign<OutIter::reference, InIter::value_type>
           HasStdMove<InIter::reference>
        && OutputIterator<OutIter, HasStdMove<InIter::reference>::result_type>
  OutIter move_backward(InIter first, InIter last,
                        OutIter result);

// 25.2.3, swap:
template<class T>
  requires MoveAssignable<T> && MoveConstructible<T> && NothrowDestructible<T>
  void swap(T& a, T& b);
template<MutableForwardIterator Iter1, MutableForwardIterator Iter2>
  requires HasSwap<Iter1::reference, Iter2::reference>
           SameType<Iter1::value_type, Iter2::value_type> &&
           SameType<Iter1::value_type&, Iter1::reference> &&
           SameType<Iter2::value_type&, Iter2::reference> &&
           Swappable<Iter1::value_type>
  Iter2 swap_ranges(Iter1 first1, Iter1 last1,
```

```
                        Iter2 first2);
template<~~Mutable~~ForwardIterator Iter1, ~~Mutable~~ForwardIterator Iter2>
  requires HasSwap<Iter1::reference, Iter2::reference>
          ~~SameType<Iter1::value_type, Iter2::value_type> &&~~
          ~~SameType<Iter1::value_type&, Iter1::reference> &&~~
          ~~SameType<Iter2::value_type&, Iter2::reference> &&~~
          ~~Swappable<Iter1::value_type>~~
  void iter_swap(Iter1 a, Iter2 b);

template<InputIterator InIter, class OutIter,
         Callable<auto, ~~InIter::reference~~const InIter::value_type&> Op>
  requires OutputIterator<OutIter, Op::result_type>
       && CopyConstructible<Op>
  OutIter transform(InIter first, InIter last,
                    OutIter result, Op op);
template<InputIterator InIter1, InputIterator InIter2,
         class OutIter,
         Callable<auto, ~~InIter1::reference~~const InIter1::value_type&,
                        ~~InIter2::reference~~const InIter2::value_type&> BinaryOp>
  requires OutputIterator<OutIter, BinaryOp::result_type>
       && CopyConstructible<BinaryOp>
  OutIter transform(InIter1 first1, InIter1 last1,
                    InIter2 first2, OutIter result,
                    BinaryOp binary_op);

template<~~Mutable~~ForwardIterator Iter, class T>
  requires OutputIterator<Iter, Iter::reference>
       && ~~HasCopyAssign~~OutputIterator<Iter~~::reference~~, ~~T~~const T&>
       && HasEqualTo<Iter::~~reference~~value_type, T>
  void replace(Iter first, Iter last,
               const T& old_value, const T&
               new_value);
template<~~Mutable~~ForwardIterator Iter, Predicate<auto, Iter::~~reference~~value_type> Pred, class T>
  requires OutputIterator<Iter, Iter::reference>
       && ~~HasCopyAssign~~OutputIterator<Iter~~::reference~~, ~~T~~const T&>
       && CopyConstructible<Pred>
  void replace_if(Iter first, Iter last,
                  Pred pred, const T& new_value);
template<InputIterator InIter, typename OutIter, class T>
  requires OutputIterator<OutIter, InIter::~~value_type~~reference>
       && ~~HasCopyAssign~~OutputIterator<Iter~~::reference~~, ~~T~~const T&>
       && HasEqualTo<InIter::~~reference~~value_type, T>
  OutIter replace_copy(InIter first, InIter last,
                       OutIter result,
                       const T& old_value, const T& new_value);
template<InputIterator InIter, typename OutIter,
         Predicate<auto, InIter::~~reference~~value_type> Pred, class T>
  requires OutputIterator<OutIter, InIter::~~value_type~~reference>
       && ~~HasCopyAssign~~OutputIterator<OutIter~~::reference~~, ~~T~~const T&>
       && CopyConstructible<Pred>
```

Draft

```
    OutIter replace_copy_if(InIter first, InIter last,
                            OutIter result,
                            Pred pred, const T& new_value);

template<MutableForwardIterator Iter, class T>
  requires HasCopyAssignOutputIterator<Iter::reference, Tconst T&>
  void fill(Iter first, Iter last, const T& value);
template<class Iter, IntegralLike Size, class T>
  requires OutputIterator<Iter, const T&>
  void fill_n(Iter first, Size n, const T& value);

template<MutableForwardIterator Iter, Callable Generator>
  requires HasCopyAssignOutputIterator<Iter::reference, Generator::result_type>
        && CopyConstructible<Generator>
  void generate(Iter first, Iter last,
                Generator gen);
template<class Iter, IntegralLike Size, Callable Generator>
  requires OutputIterator<Iter, Generator::result_type>
        && CopyConstructible<Generator>
  void generate_n(Iter first, Size n, Generator gen);

template<ForwardIterator Iter, class T>
  requires MoveAssignableOutputIterator<Iter, Iter::reference>
        && HasEqualTo<Iter::referencevalue_type, T>
  Iter remove(Iter first, Iter last,
              const T& value);
template<ForwardIterator Iter, Predicate<auto, Iter::referencevalue_type> Pred>
  requires MoveAssignableOutputIterator<Iter, Iter::reference>
        && CopyConstructible<Pred>
  Iter remove_if(Iter first, Iter last,
                 Pred pred);
template<InputIterator InIter, OutputIterator<auto, InIter::value_typereference> OutIter, class T>
  requires HasEqualTo<InIter::referencevalue_type, T>
  OutIter remove_copy(InIter first, InIter last,
                      OutIter result, const T& value);
template<InputIterator InIter, OutputIterator<auto, InIter::value_typereference> OutIter,
         Predicate<auto, InIter::referencevalue_type> Pred>
  requires CopyConstructible<Pred>
  OutIter remove_copy_if(InIter first, InIter last,
                         OutIter result, Pred pred);

template<ForwardIterator Iter>
  requires MoveAssignableOutputIterator<Iter, Iter::reference>
        && EqualityComparable<Iter::referencevalue_type>
  Iter unique(Iter first, Iter last);
template<ForwardIterator Iter,
         Predicate<auto, Iter::referencevalue_type, Iter::referencevalue_type> Pred>
  requires MoveAssignableOutputIterator<Iter, Iter::reference>
        && CopyConstructible<Pred>
  Iter unique(Iter first, Iter last,
```

```
                Pred pred);
    template<InputIterator InIter, ~~OutputIterator<auto, InIter::value_type>~~typename OutIter>
      requires OutputIterator<OutIter, InIter::reference>
            && OutputIterator<OutIter, const InIter::value_type&>
            && EqualityComparable<InIter::value_type>
            && CopyAssignable<InIter::value_type>
            && CopyConstructible<InIter::value_type>
            && !ForwardIterator<InIter>
            && !~~Mutable~~ForwardIterator<OutIter>
      OutIter unique_copy(InIter first, InIter last,
                          OutIter result);
    template<ForwardIterator InIter, OutputIterator<auto, InIter::~~value_type~~reference> OutIter>
      requires EqualityComparable<InIter::~~reference~~value_type>
      OutIter unique_copy(InIter first, InIter last,
                          OutIter result);
    template<InputIterator InIter, ~~Mutable~~ForwardIterator OutIter>
      requires ~~HasCopyAssign~~OutputIterator<OutIter~~::reference~~, InIter::reference>
            && HasEqualTo<OutIter::~~reference~~value_type, InIter::value_type>
            && !ForwardIterator<InIter>
      OutIter unique_copy(InIter first, InIter last,
                          OutIter result);
    template<InputIterator InIter, ~~OutputIterator<auto, InIter::value_type>~~typename OutIter,
             Predicate<auto, InIter::value_type, InIter::value_type> Pred>
      requires OutputIterator<OutIter, InIter::reference>
            && OutputIterator<OutIter, const InIter::value_type&>
            && CopyAssignable<InIter::value_type>
            && CopyConstructible<InIter::value_type>
            && CopyConstructible<Pred>
            && !ForwardIterator<InIter>
            && !~~Mutable~~ForwardIterator<OutIter>
      OutIter unique_copy(InIter first, InIter last,
                          OutIter result, Pred pred);
    template<ForwardIterator InIter, OutputIterator<auto, InIter::~~value_type~~reference> OutIter,
             Predicate<auto, InIter::~~reference~~value_type, InIter::~~reference~~value_type> Pred>
      requires CopyConstructible<Pred>
      OutIter unique_copy(InIter first, InIter last,
                          OutIter result);
    template<InputIterator InIter, ~~Mutable~~ForwardIterator OutIter,
             Predicate<auto, OutIter::~~reference~~value_type, InIter::~~reference~~value_type> Pred>
      requires ~~HasCopyAssign~~OutputIterator<OutIter~~::reference~~, InIter::reference>
            && CopyConstructible<Pred>
            && !ForwardIterator<InIter>
      OutIter unique_copy(InIter first, InIter last,
                          OutIter result, Pred pred);

    template<BidirectionalIterator Iter>
      requires ~~SwappableIterator<Iter>~~HasSwap<Iter::reference, Iter::reference>
      void reverse(Iter first, Iter last);
    template<BidirectionalIterator InIter, OutputIterator<auto, InIter::~~value_type~~reference> OutIter>
      OutIter reverse_copy(InIter first, InIter last, OutIter result);
```

```
template<ForwardIterator Iter>
  requires ~~SwappableIterator<Iter>~~HasSwap<Iter::reference, Iter::reference>
  void rotate(Iter first, Iter middle,
              Iter last);
template<ForwardIterator InIter, OutputIterator<auto, InIter::~~value_type~~reference> OutIter>
  OutIter rotate_copy(InIter first, InIter middle,
                      InIter last, OutIter result);

template<RandomAccessIterator Iter>
  requires ~~SwappableIterator<Iter>~~HasSwap<Iter::reference, Iter::reference>
  void random_shuffle(Iter first,
                      Iter last);
template<RandomAccessIterator Iter, Callable<auto, Iter::difference_type> Rand>
  requires ~~SwappableIterator<Iter>~~HasSwap<Iter::reference, Iter::reference>
        && Convertible<Rand::result_type, Iter::difference_type>
        && CopyConstructible<Rand>
  void random_shuffle(Iter first,
                      Iter last,
                      Rand& rand);
template<class RandomAccessIterator, class UniformRandomNumberGenerator>
  void random_shuffle(RandomAccessIterator first,
                      RandomAccessIterator last,
                      UniformRandomNumberGenerator& rand);

// 25.2.13, partitions:
template<BidirectionalIterator Iter, Predicate<auto, Iter::~~reference~~value_type> Pred>
  requires ~~SwappableIterator<Iter>~~HasSwap<Iter::reference, Iter::reference>
        && CopyConstructible<Pred>
  Iter partition(Iter first, Iter last, Pred pred);
template<BidirectionalIterator Iter, Predicate<auto, Iter::~~reference~~value_type> Pred>
  requires ~~SwappableIterator<Iter>~~ShuffleIterator<Iter>
        && CopyConstructible<Pred>
  Iter stable_partition(Iter first, Iter last, Pred pred);

// 25.3, sorting and related operations:
// 25.3.1, sorting:
template<RandomAccessIterator Iter>
  requires ~~SwappableIterator<Iter>~~ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void sort(Iter first, Iter last);
template<RandomAccessIterator Iter,
         ~~Predicate~~StrictWeakOrder<auto, Iter::value_type~~, Iter::value_type~~> Compare>
  requires ~~SwappableIterator<Iter>~~ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void sort(Iter first, Iter last,
            Compare comp);

template<RandomAccessIterator Iter>
  requires ~~SwappableIterator<Iter>~~ShuffleIterator<Iter>
```

```
          && LessThanComparable<Iter::value_type>
  void stable_sort(Iter first, Iter last);
template<RandomAccessIterator Iter,
         PredicateStrictWeakOrder<auto, Iter::value_type, Iter::value_type> Compare>
  requires SwappableIterator<Iter>ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void stable_sort(Iter first, Iter last,
                   Compare comp);


template<RandomAccessIterator Iter>
  requires SwappableIterator<Iter>ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void partial_sort(Iter first,
                    Iter middle,
                    Iter last);
template<RandomAccessIterator Iter,
         PredicateStrictWeakOrder<auto, Iter::value_type, Iter::value_type> Compare>
  requires SwappableIterator<Iter>ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void partial_sort(Iter first,
                    Iter middle,
                    Iter last,
                    Compare comp);
template<InputIterator InIter, RandomAccessIterator RAIter>
  requires HasCopyAssign<RAIter::reference, InIter::value_typereference> &&
           SwappableIterator<RAIter>Swappable<RAIter::reference> &&
           HasLess<InIter::value_type, RAIter::value_type> &&
           HasLess<RAIter::value_type, InIter::value_type> &&
           HasLess<RAIter::value_type>
  RAIter partial_sort_copy(InIter first, InIter last,
                           RAIter result_first, RAIter result_last);
template<InputIterator InIter, RandomAccessIterator RAIter, class Compare>
  requires HasCopyAssign<RAIter::reference, InIter::value_typereference> &&
           SwappableIterator<RAIter>Swappable<RAIter::reference> &&
           Predicate<Compare, InIter::value_type, RAIter::value_type> &&
           Predicate<Compare, RAIter::value_type, InIter::value_type> &&
           PredicateStrictWeakOrder<Compare, RAIter::value_type>}
  RAIter partial_sort_copy(InIter first, InIter last,
                           RAIter result_first, RAIter result_last,
                           Compare comp);


template<ForwardIterator Iter>
  requires LessThanComparable<Iter::value_type>
  bool is_sorted(Iter first, Iter last);
template<ForwardIterator Iter,
         PredicateStrictWeakOrder<auto, Iter::value_type, Iter::value_type> Compare>
  requires CopyConstructible<Compare>
  bool is_sorted(Iter first, Iter last,
                 Compare comp);
template<ForwardIterator Iter>
```

```
  requires LessThanComparable<Iter::value_type>
  Iter is_sorted_until(Iter first, Iter last);
template<ForwardIterator Iter,
         ~~Predicate~~StrictWeakOrder<auto, Iter::value_type~~, Iter::value_type~~> Compare>
  requires CopyConstructible<Compare>
  Iter is_sorted_until(Iter first, Iter last,
                       Compare comp);


template<RandomAccessIterator Iter>
  requires ~~SwappableIterator<Iter>~~ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void nth_element(Iter first, Iter nth,
                   Iter last);
template<~~SwappableIterator<Iter>~~RandomAccessIterator Iter,
         ~~Predicate~~StrictWeakOrder<auto, Iter::value_type~~, Iter::value_type~~> Compare>
  requires ~~SameType<Iter::value_type&, Iter::reference> && Swappable<Iter::value_type> &&~~
           ~~MoveConstructible<Iter::value_type> && MoveAssignable<Iter::value_type>~~
           ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void nth_element(Iter first, Iter nth,
                   Iter last,  Compare comp);


// 25.3.3, binary search:
template<ForwardIterator Iter, class T>
  requires HasLess<Iter::~~reference~~value_type, T>
  Iter lower_bound(Iter first, Iter last,
                   const T& value);
template<ForwardIterator Iter, class T, Predicate<auto, Iter::~~reference~~value_type, T> Compare>
  requires CopyConstructible<Compare>
  Iter lower_bound(Iter first, Iter last,
                   const T& value, Compare comp);


template<ForwardIterator Iter, class T>
  requires HasLess<T, Iter::~~reference~~value_type>
  Iter upper_bound(Iter first, Iter last,
                   const T& value);
template<ForwardIterator Iter, class T, Predicate<auto, T, Iter::~~reference~~value_type> Compare>
  requires CopyConstructible<Compare>
  Iter upper_bound(Iter first, Iter last,
                   const T& value, Compare comp);


template<ForwardIterator Iter, class T>
  requires HasLess<T, Iter::~~reference~~value_type>
        && HasLess<Iter::~~reference~~value_type, T>
  pair<Iter, Iter>
    equal_range(Iter first,
                Iter last, const T& value);
template<ForwardIterator Iter, class T, ~~class~~CopyConstructible Compare>
  requires Predicate<Compare, T, Iter::~~reference~~value_type>
        && Predicate<Compare, Iter::~~reference~~value_type, T>
```

```
    pair<Iter, Iter>
      equal_range(Iter first,
                  Iter last, const T& value,
                  Compare comp);

template<ForwardIterator Iter, class T>
  requires HasLess<T, Iter::referencevalue_type>
        && HasLess<Iter::referencevalue_type, T>
  bool binary_search(Iter first, Iter last,
                     const T& value);
template<ForwardIterator Iter, class T, classCopyConstructible Compare>
  requires Predicate<Compare, T, Iter::referencevalue_type>
        && Predicate<Compare, Iter::referencevalue_type, T>
  bool binary_search(Iter first, Iter last,
                     const T& value, Compare comp);

// 25.3.4, merge:
template<InputIterator InIter1, InputIterator InIter2,
         OutputIterator<auto, InIter1::value_type>typename OutIter>
  requires SameType<InIter1::value_type, InIter2::value_type>
           OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && HasLess<InIter2::value_type, InIter1::value_type>
        && LessThanComparable<InIter1::value_type>
        && LessThanComparable<InIter2::value_type>
  OutIter merge(InIter1 first1, InIter1 last1,
                InIter2 first2, InIter2 last2,
                OutIter result);
template<InputIterator InIter1, InputIterator InIter2,
         OutputIterator<auto, InIter1::value_type>typename OutIter,
         Predicate<auto, InIter12::value_type, InIter21::value_type> Compare>
  requires SameType<InIter1::value_type, InIter2::value_type>
           OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && CopyConstructible<Compare>
        && StrictWeakOrder<Compare, InIter1::value_type>
        && StrictWeakOrder<Compare, InIter2::value_type>
  OutIter merge(InIter1 first1, InIter1 last1,
                InIter2 first2, InIter2 last2,
                OutIter result, Compare comp);

template<BidirectionalIterator Iter>
  requires SwappableIterator<Iter>ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void inplace_merge(Iter first,
                     Iter middle,
                     Iter last);
template<BidirectionalIterator Iter,
         PredicateStrictWeakOrder<auto, Iter::value_type, Iter::value_type> Compare>
  requires SwappableIterator<Iter>ShuffleIterator<Iter>
```

```
                && CopyConstructible<Compare>
   void inplace_merge(Iter first,
                      Iter middle,
                      Iter last, Compare comp);


// 25.3.5, set operations:
template<InputIterator Iter1, InputIterator Iter2>
   requires SameType<Iter1::value_type, Iter2::value_type> && LessThanComparable<Iter1::value_type>
            HasLess<Iter1::value_type, Iter2::value_type>
         && HasLess<Iter2::value_type, Iter1::value_type>
         && LessThanComparable<Iter1::value_type>
         && LessThanComparable<Iter2::value_type>
   bool includes(Iter1 first1, Iter1 last1,
                 Iter2 first2, Iter2 last2);
template<InputIterator Iter1, InputIterator Iter2,
         Predicate<auto, Iter1::value_type, Iter2::value_type>typename Compare>
   requires SameType<Iter1::value_type, Iter2::value_type>
            Predicate<Compare, Iter1::value_type, Iter2::value_type>
         && Predicate<Compare, Iter2::value_type, Iter1::value_type>
         && StrictWeakOrder<Compare, Iter1::value_type>
         && StrictWeakOrder<Compare, Iter2::value_type>
   bool includes(Iter1 first1, Iter1 last1,
                 Iter2 first2, Iter2 last2,
                 Compare comp);


template<InputIterator InIter1, InputIterator InIter2,
         OutputIterator<auto, InIter1::value_type>typename OutIter>
   requires SameType<InIter1::value_type, InIter2::value_type>
         && LessThanComparable<InIter1::value_type>
            OutputIterator<OutIter, InIter1::reference>
         && OutputIterator<OutIter, InIter2::reference>
         && HasLess<InIter2::value_type, InIter1::value_type>
         && HasLess<InIter1::value_type, InIter2::value_type>
         && LessThanComparable<InIter1::value_type>
         && LessThanComparable<InIter2::value_type>
   OutIter set_union(InIter1 first1, InIter1 last1,
                     InIter2 first2, InIter2 last2,
                     OutIter result);
template<InputIterator InIter1, InputIterator InIter2,
         OutputIterator<auto, InIter1::value_type>typename OutIter,
         Predicate<auto, InIter1::value_type, InIter2::value_type>CopyConstructible Compare>
   requires SameType<InIter1::value_type, InIter2::value_type>
            OutputIterator<OutIter, InIter1::reference>
         && OutputIterator<OutIter, InIter2::reference>
         && Predicate<Compare, InIter1::value_type, InIter2::value_type>
         && Predicate<Compare, InIter2::value_type, InIter1::value_type>
         && StrictWeakOrder<Compare, InIter1::value_type>
         && StrictWeakOrder<Compare, InIter2::value_type>
   OutIter set_union(InIter1 first1, InIter1 last1,
                     InIter2 first2, InIter2 last2,
```

```
                      OutIter result, Compare comp);

template<InputIterator InIter1, InputIterator InIter2,
         OutputIterator<auto, InIter1::value_type>typename OutIter>
  requires SameType<InIter1::value_type, InIter2::value_type>
        && LessThanComparable<InIter1::value_type>
           OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && HasLess<InIter2::value_type, InIter1::value_type>
        && HasLess<InIter1::value_type, InIter2::value_type>
        && LessThanComparable<InIter1::value_type>
        && LessThanComparable<InIter2::value_type>
  OutIter set_intersection(InIter1 first1, InIter1 last1,
                           InIter2 first2, InIter2 last2,
                           OutIter result);
template<InputIterator InIter1, InputIterator InIter2,
         OutputIterator<auto, InIter1::value_type>typename OutIter,
         Predicate<auto, InIter1::value_type, InIter2::value_type>CopyConstructible Compare>
  requires SameType<InIter1::value_type, InIter2::value_type>
           OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && Predicate<Compare, InIter1::value_type, InIter2::value_type>
        && Predicate<Compare, InIter2::value_type, InIter1::value_type>
        && StrictWeakOrder<Compare, InIter1::value_type>
        && StrictWeakOrder<Compare, InIter2::value_type>
  OutIter set_intersection(InIter1 first1, InIter1 last1,
                           InIter2 first2, InIter2 last2,
                           OutIter result, Compare comp);

template<InputIterator InIter1, InputIterator InIter2,
         OutputIterator<auto, InIter1::value_type>typename OutIter>
  requires SameType<InIter1::value_type, InIter2::value_type>
        && LessThanComparable<InIter1::value_type>
           OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && HasLess<InIter2::value_type, InIter1::value_type>
        && HasLess<InIter1::value_type, InIter2::value_type>
        && LessThanComparable<InIter1::value_type>
        && LessThanComparable<InIter2::value_type>
  OutIter set_difference(InIter1 first1, InIter1 last1,
                         InIter2 first2, InIter2 last2,
                         OutIter result);
template<InputIterator InIter1, InputIterator InIter2,
         OutputIterator<auto, InIter1::value_type>typename OutIter,
         Predicate<auto, InIter1::value_type, InIter2::value_type>CopyConstructible Compare>
  requires SameType<InIter1::value_type, InIter2::value_type>
           OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && Predicate<Compare, InIter1::value_type, InIter2::value_type>
        && Predicate<Compare, InIter2::value_type, InIter1::value_type>
```

```
          && StrictWeakOrder<Compare, InIter1::value_type>
          && StrictWeakOrder<Compare, InIter2::value_type>
  OutIter set_difference(InIter1 first1, InIter1 last1,
                          InIter2 first2, InIter2 last2,
                          OutIter result, Compare comp);

template<InputIterator InIter1, InputIterator InIter2,
         ~~OutputIterator<auto, InIter1::value_type>~~typename OutIter>
  requires ~~SameType<InIter1::value_type, InIter2::value_type>~~
          ~~&& LessThanComparable<InIter1::value_type>~~
            OutputIterator<OutIter, InIter1::reference>
          && OutputIterator<OutIter, InIter2::reference>
          && HasLess<InIter2::value_type, InIter1::value_type>
          && HasLess<InIter1::value_type, InIter2::value_type>
          && LessThanComparable<InIter1::value_type>
          && LessThanComparable<InIter2::value_type>
  OutIter set_symmetric_difference(InIter1 first1, InIter1 last1,
                                    InIter2 first2, InIter2 last2,
                                    OutIter result);
template<InputIterator InIter1, InputIterator InIter2,
         ~~OutputIterator<auto, InIter1::value_type>~~typename OutIter,
         ~~Predicate<auto, InIter1::value_type, InIter2::value_type>~~CopyConstructible Compare>
  requires ~~SameType<InIter1::value_type, InIter2::value_type>~~
            OutputIterator<OutIter, InIter1::reference>
          && OutputIterator<OutIter, InIter2::reference>
          && Predicate<Compare, InIter1::value_type, InIter2::value_type>
          && Predicate<Compare, InIter2::value_type, InIter1::value_type>
          && StrictWeakOrder<Compare, InIter1::value_type>
          && StrictWeakOrder<Compare, InIter2::value_type>
  OutIter set_symmetric_difference(InIter1 first1, InIter1 last1,
                                    InIter2 first2, InIter2 last2,
                                    OutIter result, Compare comp);


// 25.3.6, heap operations:
template<RandomAccessIterator Iter>
  requires ~~SwappableIterator<Iter>~~ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void push_heap(Iter first, Iter last);
template<RandomAccessIterator Iter,
         ~~Predicate~~StrictWeakOrder<auto, Iter::value_type~~, Iter::value_type~~> Compare>
  requires ~~SwappableIterator<Iter>~~ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void push_heap(Iter first, Iter last,
                 Compare comp);

template<RandomAccessIterator Iter>
  requires ~~SwappableIterator<Iter>~~ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void pop_heap(Iter first, Iter last);
template<RandomAccessIterator Iter,
```

```
                   ~~Predicate~~StrictWeakOrder<auto, Iter::value_type~~, Iter::value_type~~> Compare>
    requires ~~SwappableIterator<Iter>~~ShuffleIterator<Iter>
          && CopyConstructible<Compare>
    void pop_heap(Iter first, Iter last,
                   Compare comp);

  template<RandomAccessIterator Iter>
    requires ~~SwappableIterator<Iter>~~ShuffleIterator<Iter>
          && LessThanComparable<Iter::value_type>
    void make_heap(Iter first, Iter last);
  template<RandomAccessIterator Iter,
           ~~Predicate~~StrictWeakOrder<auto, Iter::value_type~~, Iter::value_type~~> Compare>
    requires ~~SwappableIterator<Iter>~~ShuffleIterator<Iter>
          && CopyConstructible<Compare>
    void make_heap(Iter first, Iter last,
                   Compare comp);

  template<RandomAccessIterator Iter>
    requires ~~SwappableIterator<Iter>~~ShuffleIterator<Iter>
          && LessThanComparable<Iter::value_type>
    void sort_heap(Iter first, Iter last);
  template<RandomAccessIterator Iter,
           ~~Predicate~~StrictWeakOrder<auto, Iter::value_type~~, Iter::value_type~~> Compare>
    requires ~~SwappableIterator<Iter>~~ShuffleIterator<Iter>
          && CopyConstructible<Compare>
    void sort_heap(Iter first, Iter last,
                   Compare comp);

  template<RandomAccessIterator Iter>
    requires LessThanComparable<Iter::value_type>
    bool is_heap(Iter first, Iter last);
  template<RandomAccessIterator Iter,
           ~~Predicate~~StrictWeakOrder<auto, Iter::value_type~~, Iter::value_type~~> Compare>
    bool is_heap(Iter first, Iter last, Compare comp);
  template<RandomAccessIterator Iter>
    requires LessThanComparable<Iter::value_type>
    Iter is_heap_until(Iter first, Iter last);
  template<RandomAccessIterator Iter,
           ~~Predicate~~StrictWeakOrder<auto, Iter::value_type~~, Iter::value_type~~> Compare>
    requires CopyConstructible<Compare>
    Iter is_heap_until(Iter first, Iter last,
                       Compare comp);

  // 25.3.7, minimum and maximum:
  template<LessThanComparable T> const T& min(const T& a, const T& b);
  template<class T, ~~Predicate<auto, T, T>~~StrictWeakOrder<auto, T> Compare>
    requires !SameType<T, Compare> && CopyConstructible<Compare>
    const T& min(const T& a, const T& b, Compare comp);
  ~~template<LessThanComparable T> const T& min(const T& a, const T& b, const T& c);~~
  template<LessThanComparable T, LessThanComparable... Args>
```

```
    requires SameType<T, Args>...
      const T& min(const T& a, const Args&... args);
  template<class T, class U, class... Args>
      const T& min(const T& a, const U& b, const Args&... args);

  template<LessThanComparable T> const T& max(const T& a, const T& b);
  template<class T, Predicate<auto, T, T>StrictWeakOrder<auto, T> Compare>
      requires !SameType<T, Compare> && CopyConstructible<Compare>
      const T& max(const T& a, const T& b, Compare comp);
  template<LessThanComparable T> const T& max(const T& a, const T& b, const T& c);
  template<LessThanComparable T, LessThanComparable... Args>
      requires SameType<T, Args>...
      const T& max(const T& a, const Args&... args);
  template<class T, class U, class... Args>
      const T& max(const T& a, const U& b, const Args&... args);

  template<LessThanComparable T> pair<const T&, const T&> minmax(const T& a, const T& b);
  template<class T, Predicate<auto, T, T>StrictWeakOrder<auto, T> Compare>
      requires !SameType<T, Compare> && CopyConstructible<Compare>

      pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
  template<LessThanComparable T>
      pair<const T&, const T&> minmax(const T& a, const T& b, const T& c);
  template<LessThanComparable T, LessThanComparable... Args>
      requires SameType<T, Args>...
      pair<const T&, const T&> minmax(const T& a, const Args&... args);
  template<class T, class U, class... Args>
      pair<const T&, const T&> minmax(const T& a, const U& b, const Args&... args);

  template<ForwardIterator Iter>
      requires LessThanComparable<Iter::referencevalue_type>
      Iter min_element(Iter first, Iter last);
  template<ForwardIterator Iter,
            PredicateStrictWeakOrder<auto, Iter::referencevalue_type, Iter::reference> Compare>
      requires CopyConstructible<Compare>
      Iter min_element(Iter first, Iter last,
                       Compare comp);

  template<ForwardIterator Iter>
      requires LessThanComparable<Iter::referencevalue_type>
      Iter max_element(Iter first, Iter last);
  template<ForwardIterator Iter,
            PredicateStrictWeakOrder<auto, Iter::referencevalue_type, Iter::reference> Compare>
      requires CopyConstructible<Compare>
      Iter max_element(Iter first, Iter last,
                       Compare comp);

  template<ForwardIterator Iter>
      requires LessThanComparable<Iter::value_type>
      pair<Iter, Iter>
```

```
      minmax_element(Iter first, Iter last);
  template<ForwardIterator Iter,
           ~~Predicate~~StrictWeakOrder<auto, Iter::value_type~~, Iter::value_type~~> Compare>
    requires CopyConstructible<Compare>
    pair<Iter, Iter>
      minmax_element(Iter first, Iter last, Compare comp);

  template<InputIterator Iter1, InputIterator Iter2>
    requires HasLess<Iter1::~~reference~~value_type, Iter2::~~reference~~value_type>
          && HasLess<Iter2::~~reference~~value_type, Iter1::~~reference~~value_type>
    bool lexicographical_compare(Iter1 first1, Iter1 last1,
                                 Iter2 first2, Iter2 last2);

  template<InputIterator Iter1, InputIterator Iter2, ~~class~~CopyConstructible Compare>
    requires Predicate<Compare, Iter1::~~reference~~value_type, Iter2::~~reference~~value_type>
          && Predicate<Compare, Iter2::~~reference~~value_type, Iter1::~~reference~~value_type>
    bool lexicographical_compare(Iter1 first1, Iter1 last1,
                                 Iter2 first2, Iter2 last2,
                                 Compare comp);

  // 25.3.9, permutations:
  template<BidirectionalIterator Iter>
    requires ~~SwappableIterator<Iter>~~HasSwap<Iter::reference, Iter::reference>
          LessThanComparable<Iter::~~reference~~value_type>
    bool next_permutation(Iter first, Iter last);
  template<BidirectionalIterator Iter,
           ~~Predicate~~StrictWeakOrder<auto, Iter::~~reference~~value_type~~, Iter::reference~~> Compare>
    requires ~~SwappableIterator<Iter>~~HasSwap<Iter::reference, Iter::reference>
          && CopyConstructible<Compare>
    bool next_permutation(Iter first, Iter last, Compare comp);
  template<BidirectionalIterator Iter>
    requires ~~SwappableIterator<Iter>~~HasSwap<Iter::reference, Iter::reference>
          LessThanComparable<Iter::~~reference~~value_type>
    bool prev_permutation(Iter first, Iter last);
  template<BidirectionalIterator Iter,
           ~~Predicate~~StrictWeakOrder<auto, Iter::~~reference~~value_type~~, Iter::reference~~> Compare>
    requires ~~SwappableIterator<Iter>~~HasSwap<Iter::reference, Iter::reference>
          && CopyConstructible<Compare>
    bool prev_permutation(Iter first, Iter last, Compare comp);
}
```

3 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.

4 ~~Throughout this clause, the names of template parameters are used to express type requirements. If an algorithm's template parameter is InputIterator, InputIterator1, or InputIterator2, the actual template argument shall satisfy the requirements of an input iterator (24.1.1). If an algorithm's template parameter is OutputIterator, OutputIterator1, or OutputIterator2, the actual template argument shall satisfy the requirements of an output iterator (24.1.2). If an~~

algorithm's template parameter is ForwardIterator, ForwardIterator1, or ForwardIterator2, the actual template argument shall satisfy the requirements of a forward iterator (24.1.3). If an algorithm's template parameter is BidirectionalIterator, BidirectionalIterator1, or BidirectionalIterator2, the actual template argument shall satisfy the requirements of a bidirectional iterator (24.1.4). If an algorithm's template parameter is RandomAccessIterator, RandomAccessIterator1, or RandomAccessIterator2, the actual template argument shall satisfy the requirements of a random-access iterator (24.1.5).

5   If an algorithm's Effects section says 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 (24.1). [Note: this requirement does not affect arguments that are declared as OutputIterator, OutputIterator1, or OutputIterator2, because output iterators must always be mutable. — end note]

6   Both in-place and copying versions are provided for certain algorithms.[1] 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   The Predicate parameter is used whenever an algorithm expects a function object that when applied to the result of dereferencing the corresponding iterator returns a value testable as true. In other words, if an algorithm takes Predicate pred as its argument and first as its iterator argument, it should work correctly in the construct if (pred(*first)){...}. The function object pred shall not apply any non-constant function through the dereferenced iterator. This function object may be a pointer to function, or an object of a type with an appropriate function call operator.

8   The Predicate 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 Predicate binary_pred as its argument and first1 and first2 as its iterator arguments, it should work correctly in the construct if (binary_pred(*first1, *first2)){...}. Predicate always takes the first iterator type as its first argument, that is, in those cases when T value is part of the signature, it should work correctly in the context of if (binary_pred(*first1, value)){...}. binary_pred shall not apply any non-constant function through the dereferenced iterators.

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

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

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

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

and that of `b−a` is the same as of

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

---

[1] 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`.

25.1   Non-modifying sequence operations   [alg.nonmodifying]

25.1.1   For each   [alg.foreach]

The standard does not state whether the function object takes a value of the iterator's value type or reference type. The SGI STL documentation says it's the value type, but that conflicts with user expectations that they are operating on references. Also, "the result of dereferencing every iterator..." is the reference type, not the value type. Therefore, we have chosen to use the reference type.

```
template<InputIterator Iter, Callable<auto, Iter::reference> Function>
  requires CopyConstructible<Function>
  Function for_each(Iter first, Iter last, Function f);
```

1   *Effects:* Applies f to the result of dereferencing every iterator in the range [$first$, $last$), starting from $first$ and proceeding to $last - 1$.

2   *Returns:* f.

3   *Complexity:* Applies f exactly $last - first$ times.

4   ~~Notes: If f returns a result, the result is ignored.~~

25.1.2   Find   [alg.find]

```
template<InputIterator Iter, class T>
  requires HasEqualTo<Iter::referencevalue_type, T>
  Iter find(Iter first, Iter last, const T& value);

template<InputIterator Iter, Predicate<auto, Iter::referencevalue_type> Pred>
  requires CopyConstructible<Pred>
  Iter find_if(Iter first, Iter last, Pred pred);
```

1   *Returns:* The first iterator i in the range [$first$, $last$) for which the following corresponding conditions hold: *i == $value$, $pred$(*i) != false. Returns $last$ if no such iterator is found.

2   *Complexity:* At most $last - first$ applications of the corresponding predicate.

25.1.3   Find End   [alg.find.end]

```
template<ForwardIterator Iter1, ForwardIterator Iter2>
  requires HasEqualTo<Iter1::referencevalue_type, Iter2::referencevalue_type>
  Iter1 find_end(Iter1 first1, Iter1 last1,
                 Iter2 first2, Iter2 last2);

template<ForwardIterator Iter1, ForwardIterator Iter2,
         Predicate<auto, Iter1::referencevalue_type, Iter2::referencevalue_type> Pred>
  requires CopyConstructible<Pred>
  Iter1 find_end(Iter1 first1, Iter1 last1,
                 Iter2 first2, Iter2 last2,
                 Pred pred);
```

1   *Effects:* Finds a subsequence of equal values in a sequence.

2   *Returns:* The last iterator i in the range [*first1*, *last1* − (*last2* − *first2*)) such that for any non-
        negative integer n < (*last2* − *first2*), the following corresponding conditions hold: *(i + n) == *(
        first2 + n), *pred*(*(i + n), *(*first2* + n)) != false. Returns *last1* if no such iterator is found.

3   *Complexity:* At most (*last2* − *first2*) * (*last1* − *first1* − (*last2* − *first2*) + 1) applications
        of the corresponding predicate.

25.1.4   Find First   [alg.find.first.of]

This text assumes that the proposed resolution to DR 576 is accepted, weakening the requirements on the first type
parameter (`Iter1`) to `Input Iterator`.

```
template<InputIterator Iter1, ForwardIterator Iter2>
  requires HasEqualTo<Iter1::referencevalue_type, Iter2::referencevalue_type>
  Iter1 find_first_of(Iter1 first1, Iter1 last1,
                      Iter2 first2, Iter2 last2);

template<InputIterator Iter1, ForwardIterator Iter2,
         Predicate<auto, Iter1::referencevalue_type, Iter2::referencevalue_type> Pred>
  requires CopyConstructible<Pred>
  Iter1 find_first_of(Iter1 first1, Iter1 last1,
                      Iter2 first2, Iter2 last2,
                      Pred pred);
```

1   *Effects:* Finds an element that matches one of a set of values.

2   *Returns:* The first iterator i in the range [*first1*, *last1*) such that for some iterator j in the range [*first2*,
        last2) the following conditions hold: *i == *j, *pred*(*i,*j) != false. Returns *last1* if no such itera-
        tor is found.

3   *Complexity:* At most (*last1*−*first1*) * (*last2*−*first2*) applications of the corresponding predicate.

25.1.5   Adjacent find   [alg.adjacent.find]

```
template<ForwardIterator Iter>
  requires HasEqualToEqualityComparable<Iter::referencevalue_type>
  Iter adjacent_find(Iter first, Iter last);

template<ForwardIterator Iter,
         Predicate<auto, Iter::referencevalue_type, Iter::referencevalue_type> Pred>
  requires CopyConstructible<Pred>
  Iter adjacent_find(Iter first, Iter last, Pred pred);
```

1   *Returns:* The first iterator i such that both i and i + 1 are in the range [*first*, *last*) for which the following
        corresponding conditions hold: *i == *(i + 1), *pred*(*i, *(i + 1)) != false. Returns *last* if no
        such iterator is found.

2   *Complexity:* For a nonempty range, exactly `min((i - `*`first`*`) + 1, (`*`last`* `- `*`first`*`) - 1)` applications of the corresponding predicate, where `i` is `adjacent_find`'s return value.

25.1.6   Count   [alg.count]

```
template<InputIterator Iter, class T>
  requires HasEqualTo<Iter::referencevalue_type, T>
  Iter::difference_type count(Iter first, Iter last, const T& value);

template<InputIterator Iter, Predicate<auto, Iter::referencevalue_type> Pred>
  requires CopyConstructible<Pred>
  Iter::difference_type count_if(Iter first, Iter last, Pred pred);
```

1   *Effects:* Returns the number of iterators `i` in the range [*first*, *last*) for which the following corresponding conditions hold: `*i == `*`value`*, *`pred`*`(*i) != false`.

2   *Complexity:* Exactly *`last`* `- `*`first`* applications of the corresponding predicate.

25.1.7   Mismatch   [mismatch]

```
template<InputIterator Iter1, InputIterator Iter2>
  requires HasEqualTo<Iter1::referencevalue_type, Iter2::referencevalue_type>
  pair<Iter1, Iter2> mismatch(Iter1 first1, Iter1 last1,
                              Iter2 first2);

template<InputIterator Iter1, InputIterator Iter2,
         Predicate<auto, Iter1::referencevalue_type, Iter2::referencevalue_type> Pred>
  requires CopyConstructible<Pred>
  pair<Iter1, Iter2> mismatch(Iter1 first1, Iter1 last1,
                              Iter2 first2, Pred pred);
```

1   *Returns:* A pair of iterators `i` and `j` such that `j == `*`first2`* `+ (i - `*`first1`*`)` and `i` is the first iterator in the range [*first1*, *last1*) for which the following corresponding conditions hold:

```
!(*i == *(first2 + (i - first1)))
pred(*i, *(first2 + (i - first1))) == false
```

Returns the pair *`last1`* and *`first2`* `+ (`*`last1`* `- `*`first1`*`)` if such an iterator `i` is not found.

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

25.1.8   Equal   [alg.equal]

```
template<InputIterator Iter1, InputIterator Iter2>
  requires HasEqualTo<Iter1::referencevalue_type, Iter2::referencevalue_type>
  bool equal(Iter1 first1, Iter1 last1,
             Iter2 first2);

template<InputIterator Iter1, InputIterator Iter2,
```

```
          Predicate<auto, Iter1::referencevalue_type, Iter2::referencevalue_type> Pred>
    requires CopyConstructible<Pred>
    bool equal(Iter1 first1, Iter1 last1,
               Iter2 first2, Pred pred);
```

1   *Returns:* `true` if for every iterator `i` in the range [`first1`, `last1`) the following corresponding conditions hold:
     `*i == *(first2 + (i - first1))`, `pred(*i, *(first2 + (i - first1))) != false`. Otherwise,
     returns `false`.

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

25.1.9   Search   [alg.search]

```
template<ForwardIterator Iter1, ForwardIterator Iter2>
  requires HasEqualTo<Iter1::referencevalue_type, Iter2::referencevalue_type>
  Iter1 search(Iter1 first1, Iter1 last1,
               Iter2 first2, Iter2 last2);

template<ForwardIterator Iter1, ForwardIterator Iter2,
         Predicate<auto, Iter1::referencevalue_type, Iter2::referencevalue_type> Pred>
  requires CopyConstructible<Pred>
  Iter1 search(Iter1 first1, Iter1 last1,
               Iter2 first2, Iter2 last2,
               Pred pred);
```

1   *Effects:* Finds a subsequence of equal values in a sequence.

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

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

```
template<ForwardIterator Iter, class T>
  requires HasEqualTo<Iter::referencevalue_type, T>
  Iter search_n(Iter first, Iter last, Iter::difference_type count,
                const T& value);

template<ForwardIterator Iter, class T,
         Predicate<auto, Iter::referencevalue_type, T> Pred>
  requires CopyConstructible<Pred>
  Iter search_n(Iter first, Iter last, Iter::difference_type count,
                const T& value, Pred pred);
```

4   Requires: The type Size is convertible to integral type (4.7, 12.3).

> We have removed the `Size` parameter and instead chosen to use the `difference_type` of the iterator. This
> change can break existing code in two ways. First, if the `Size` parameter was originally bound to a type larger
> than `difference_type` and the `count` parameter contains a value outside of the range of `difference_type` (in
> which case, `search_n` always returns `last`). Second, if the user explicitly provides an argument for the `Size`
> parameter.

5   *Effects:* Finds a subsequence of equal values in a sequence.

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

7   *Complexity:* At most (*last* − *first*) * *count* applications of the corresponding predicate if *count* is posi-
        tive, or 0 otherwise.

25.2   Mutating sequence operations   [alg.modifying.operations]

25.2.1   Copy   [alg.copy]

```
template<InputIterator InIter, OutputIterator<auto, InIter::reference> OutIter>
  OutIter copy(InIter first, InIter last,
               OutIter result);
```

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

2   *Returns:* *result* + (*last* − *first*).

3   *Requires:* *result* shall not be in the range [*first*, *last*).

4   *Complexity:* Exactly *last* − *first* assignments.

```
template<BidirectionalIterator InIter, MutableBidirectionalIterator OutIter>
  requires HasCopyAssignOutputIterator<OutIter::reference, InIter::reference>
  OutIter copy_backward(InIter first, InIter last,
                        OutIter result);
```

5   *Effects:* Copies elements in the range [*first*, *last*) into the range [*result* − (*last* − *first*),*result*)
        starting from *last* − 1 and proceeding to *first*.[2] For each positive integer n <= (*last* − *first*), per-
        forms *(*result* − n) = *(*last* − n).

6   *Requires:* *result* shall not be in the range [*first*, *last*).

7   *Returns:* *result* − (*last* − *first*).

8   *Complexity:* Exactly *last* − *first* assignments.

25.2.2   Move   [alg.move]

```
template<InputIterator InIter, OutputIterator<auto, InIter::value_type&&>typename OutIter>
  requires HasStdMove<InIter::reference>
        && OutputIterator<OutIter, HasStdMove<InIter::reference>::result_type>
  OutIter move(InIter first, InIter last,
               OutIter result);
```

---
[2] copy_backward should be used instead of copy when *last* is in the range [*result* − (*last* − *first*),*result*).

1   *Effects:* Moves elements in the range [*first*, *last*) into the range [*result*, *result* + (*last* − *first*)) starting from *first* and proceeding to *last*. For each non-negative integer n < (*last*−*first*), performs *(result* + n) = ~~std::move~~std_move(*(*first* + n)).

2   *Returns:* *result* + (*last* − *first*).

3   *Requires:* *result* shall not be in the range [*first*, *last*).

4   *Complexity:* Exactly *last* − *first* move assignments.

```
template<BidirectionalIterator InIter, BidirectionalIterator OutIter>
  requires HasMoveAssign<OutIter::reference, InIter::value_type>
           HasStdMove<InIter::reference>
        && OutputIterator<OutIter, HasStdMove<InIter::reference>::result_type>
  OutIter move_backward(InIter first, InIter last,
                        OutIter result);
```

5   *Effects:* Moves elements in the range [*first*, *last*) into the range [*result* − (*last*−*first*), *result*) starting from *last* − 1 and proceeding to *first*.[3] For each positive integer n <= (*last* − *first*), performs *(result* − n) = ~~std::move~~std_move(*(*last* − n)).

6   *Requires:* *result* shall not be in the range [*first*, *last*).

7   *Returns:* *result* − (*last* − *first*).

8   *Complexity:* Exactly *last* − *first* assignments.

25.2.3   Swap   [alg.swap]

```
template<class T>
  requires MoveAssignable<T> && MoveConstructible<T> && NothrowDestructible<T>
  void swap(T& a, T& b);
```

1   ~~Requires: Type T is MoveConstructible (33) and MoveAssignable (35).~~

2   *Effects:* Exchanges values stored in two locations.

```
template<MutableForwardIterator Iter1, MutableForwardIterator Iter2>
  requires HasSwap<Iter1::reference, Iter2::reference>
           SameType<Iter1::value_type, Iter2::value_type> &&
           SameType<Iter1::value_type&, Iter1::reference> &&
           SameType<Iter2::value_type&, Iter2::reference> &&
           Swappable<Iter1::value_type>
  Iter2 swap_ranges(Iter1 first1, Iter1 last1,
                    Iter2 first2);
```

3   *Effects:* For each non-negative integer n < (*last1* − *first1*) performs: swap(*(*first1* + n), *(*first2* + n)).

---

[3] move_backward should be used instead of move when *last* is in the range [*result* − (*last* − *first*), *result*).

4   *Requires:* The two ranges [$first1$,$last1$) and [$first2$,$first2$ + ($last1$ − $first1$)) shall not over-
     lap. ~~The type of *first1 shall be the same as the type of *first2 and that type shall satisfy the Swappable~~
     ~~requirements (20.1.4).~~

5   *Returns:* $first2$ + ($last1$ − $first1$).

6   *Complexity:* Exactly $last1$ − $first1$ swaps.

```
template<~~Mutable~~ForwardIterator Iter1, ~~Mutable~~ForwardIterator Iter2>
  requires HasSwap<Iter1::reference, Iter2::reference>
          SameType<Iter1::value_type, Iter2::value_type> &&
          SameType<Iter1::value_type&, Iter1::reference> &&
          SameType<Iter2::value_type&, Iter2::reference> &&
          Swappable<Iter1::value_type>
  void iter_swap(Iter1 a, Iter2 b);
```

7   *Effects:* swap(*a, *b).

8   ~~Requires: The type of *a shall be the same as the type of *b and that type shall satisfy the Swappable requirements~~
     ~~(20.1.4).~~

25.2.4   Transform   [alg.transform]

```
template<InputIterator InIter, class OutIter,
        Callable<auto, ~~InIter::reference~~const InIter::value_type&> Op>
  requires OutputIterator<OutIter, Op::result_type>
        && CopyConstructible<Op>
  OutIter transform(InIter first, InIter last,
                    OutIter result, Op op);

template<InputIterator InIter1, InputIterator InIter2,
        class OutIter,
        Callable<auto, ~~InIter1::reference~~const InIter1::value_type&,
                       ~~InIter2::reference~~const InIter2::value_type&> BinaryOp>
  requires OutputIterator<OutIter, BinaryOp::result_type>
        && CopyConstructible<BinaryOp>
  OutIter transform(InIter1 first1, InIter1 last1,
                    InIter2 first2, OutIter result,
                    BinaryOp binary_op);
```

> There is some question here about whether the Callable arguments should be references or const value_type&s.
> The text says that the function object shall not modify elements (which implies const value_type&), and in other
> areas we have consistently used reference-to-const, e.g., for predicates.

1   *Effects:* Assigns through every iterator i in the range [$result$,$result$ + ($last1$ − $first1$)) a new cor-
     responding value equal to op(*($first1$ + (i − $result$)) or $binary\_op$(*($first1$ + (i − $result$),
     *($first2$ + (i − $result$))).

2   *Requires:* op and $binary\_op$ shall not invalidate iterators or subranges, or modify elements in the ranges [
     $first1$,$last1$], [$first2$,$first2$ + ($last1$ − $first1$)], and [$result$,$result$ + ($last1$ − $first1$)

].[4]

3   *Returns:* `result + (last1 - first1)`.

4   *Complexity:* Exactly `last1 - first1` applications of `op` or `binary_op`.

5   *Remarks:* `result` may be equal to `first` in case of unary transform, or to `first1` or `first2` in case of binary transform.

25.2.5   Replace   [alg.replace]

```
template<MutableForwardIterator Iter, class T>
  requires OutputIterator<Iter, Iter::reference>
        && HasCopyAssignOutputIterator<Iter::reference, Tconst T&>
        && HasEqualTo<Iter::referencevalue_type, T>
  void replace(Iter first, Iter last,
               const T& old_value, const T&
               new_value);

template<MutableForwardIterator Iter, Predicate<auto, Iter::referencevalue_type> Pred, class T>
  requires OutputIterator<Iter, Iter::reference>
        && HasCopyAssignOutputIterator<Iter::reference, Tconst T&>
        && CopyConstructible<Pred>
  void replace_if(Iter first, Iter last,
                  Pred pred, const T& new_value);
```

1   *Requires:* The expression `*first = new_value` must be valid.

2   *Effects:* Substitutes elements referred by the iterator i in the range `[first, last)` with `new_value`, when the following corresponding conditions hold: `*i == old_value`, `pred(*i) != false`.

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

```
template<InputIterator InIter, typename OutIter, class T>
  requires OutputIterator<OutIter, InIter::value_typereference>
        && HasCopyAssignOutputIterator<Iter::reference, Tconst T&>
        && HasEqualTo<InIter::referencevalue_type, T>
  OutIter replace_copy(InIter first, InIter last,
                       OutIter result,
                       const T& old_value, const T& new_value);

template<InputIterator InIter, typename OutIter,
         Predicate<auto, InIter::referencevalue_type> Pred, class T>
  requires OutputIterator<OutIter, InIter::value_typereference>
        && HasCopyAssignOutputIterator<OutIter::reference, Tconst T&>
        && CopyConstructible<Pred>
  OutIter replace_copy_if(InIter first, InIter last,
                          OutIter result,
                          Pred pred, const T& new_value);
```

---

[4]The use of fully closed ranges is intentional.

4   *Requires:* The results of the expressions *`*first`* and *`new_value`* shall be writable to the `result` output iterator. The ranges [*`first`*,*`last`*) and [*`result`*,*`result`* + (*`last`* − *`first`*)) shall not overlap.

5   *Effects:* Assigns to every iterator i in the range [*`result`*,*`result`* + (*`last`* − *`first`*)) either *`new_value`* or *`*(first`* + (i − *`result`*)) depending on whether the following corresponding conditions hold:

```
*(first + (i - result)) == old_value
pred(*(first + (i - result))) != false
```

6   *Returns:* `result` + (*`last`* − *`first`*).

7   *Complexity:* Exactly `last` − `first` applications of the corresponding predicate.

25.2.6   Fill   [alg.fill]

```
template<MutableForwardIterator Iter, class T>
  requires HasCopyAssignOutputIterator<Iter::reference, Tconst T&>
  void fill(Iter first, Iter last, const T& value);

template<class Iter, IntegralLike Size, class T>
  requires OutputIterator<Iter, const T&>
  void fill_n(Iter first, Size n, const T& value);
```

1   ~~Requires: The expression value shall be writable to the output iterator. The type Size is convertible to integral type (4.7, 12.3).~~[5]

2   *Effects:* The first algorithm assigns *`value`* through all the iterators in the range [*`first`*,*`last`*). The second algorithm assigns *`value`* through all the iterators in the range [*`first`*,*`first`* + *`n`*) if *`n`* is positive, otherwise it does nothing.

3   *Complexity:* Exactly `last` − `first`, *`n`*, or 0 assignments, respectively.

25.2.7   Generate   [alg.generate]

```
template<MutableForwardIterator Iter, Callable Generator>
  requires HasCopyAssignOutputIterator<Iter::reference, Generator::result_type>
        && CopyConstructible<Generator>
  void generate(Iter first, Iter last,
                Generator gen);

template<class Iter, IntegralLike Size, Callable Generator>
  requires OutputIterator<Iter, Generator::result_type>
        && CopyConstructible<Generator>
  void generate_n(Iter first, Size n, Generator gen);
```

---

[5] The "Convertible to integral type" requirements are very odd. We can model them with concepts if we need, but I've chosen the simpler route of requiring the `Size` parameter to model `IntegralLike`. This could break existing code, although it appears that such existing code would fail to compile with at least libstdc++.

1   *Effects:* The first algorithm invokes the function object *gen* and assigns the return value of *gen* through all the iterators in the range [*first*, *last*). The second algorithm invokes the function object *gen* and assigns the return value of *gen* through all the iterators in the range [*first*, *first* + *n*) if *n* is positive, otherwise it does nothing.

2   ~~Requires: gen takes no arguments, Size is convertible to integral type (4.7, 12.3).~~[6]

3   *Complexity:* Exactly *last* - *first*, *n*, or 0 invocations of *gen* and assignments, respectively.

25.2.8 Remove   [alg.remove]

```
template<ForwardIterator Iter, class T>
  requires MoveAssignableOutputIterator<Iter, Iter::reference>
        && HasEqualTo<Iter::referencevalue_type, T>
  Iter remove(Iter first, Iter last,
              const T& value);

template<ForwardIterator Iter, Predicate<auto, Iter::referencevalue_type> Pred>
  requires MoveAssignableOutputIterator<Iter, Iter::reference>
        && CopyConstructible<Pred>
  Iter remove_if(Iter first, Iter last,
                 Pred pred);
```

1   ~~Requires: The type of *first shall satisfy the MoveAssignable requirements (Table 35).~~

2   *Effects:* Eliminates all the elements referred to by iterator i in the range [*first*, *last*) for which the following corresponding conditions hold: *i == *value*, *pred*(*i) != false.

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

4   *Remarks:* Stable.

5   *Complexity:* Exactly *last* - *first* applications of the corresponding predicate.

```
template<InputIterator InIter, OutputIterator<auto, InIter::value_typereference> OutIter, class T>
  requires HasEqualTo<InIter::referencevalue_type, T>
  OutIter remove_copy(InIter first, InIter last,
                      OutIter result, const T& value);

template<InputIterator InIter, OutputIterator<auto, InIter::value_typereference> OutIter,
         Predicate<auto, InIter::referencevalue_type> Pred>
  requires CopyConstructible<Pred>
  OutIter remove_copy_if(InIter first, InIter last,
                         OutIter result, Pred pred);
```

6   *Requires:* ~~Type T is EqualityComparable (20.1.1).~~ The ranges [*first*, *last*) and [*result*, *result* + (*last* - *first*)) shall not overlap.

---

[6] The "Convertible to integral type" requirements are very odd. We can model them with concepts if we need, but I've chosen the simpler route of requiring the `Size` parameter to model `IntegralLike`. This could break existing code, although it appears that such existing code would fail to compile with at least libstdc++.

7   *Effects:* Copies all the elements referred to by the iterator i in the range [$first$, $last$) for which the following corresponding conditions do not hold: *i == $value$, $pred$(*i) != false.

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

9   *Complexity:* Exactly $last$ - $first$ applications of the corresponding predicate.

10   *Remarks:* Stable.

25.2.9   Unique   [alg.unique]

```
template<ForwardIterator Iter>
  requires ~~MoveAssignable~~OutputIterator<Iter, Iter::reference>
        && EqualityComparable<Iter::~~reference~~value_type>
  Iter unique(Iter first, Iter last);

template<ForwardIterator Iter, Predicate<auto, Iter::~~reference~~value_type, Iter::~~reference~~value_type> Pred>
  requires ~~MoveAssignable~~OutputIterator<Iter, Iter::reference>
        && CopyConstructible<Pred>
  Iter unique(Iter first, Iter last,
              Pred pred);
```

1   *Effects:* For a nonempty range, eliminates all but the first element from every consecutive group of equivalent elements referred to by the iterator i in the range [$first$ + 1, $last$) for which the following conditions hold: *(i - 1) == *i or $pred$(*(i - 1), *i) != false.

2   *Requires:* The comparison function shall be an equivalence relation.

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

4   *Complexity:* For nonempty ranges, exactly ($last$ - $first$) - 1 applications of the corresponding predicate.

```
template<InputIterator InIter, ~~OutputIterator<auto, InIter::value_type>~~typename OutIter>
  requires OutputIterator<OutIter, InIter::reference>
        && OutputIterator<OutIter, const InIter::value_type&>
        && EqualityComparable<InIter::value_type>
        && CopyAssignable<InIter::value_type>
        && CopyConstructible<InIter::value_type>
        && !ForwardIterator<InIter>
        && !~~Mutable~~ForwardIterator<OutIter>
  OutIter unique_copy(InIter first, InIter last,
                      OutIter result);

template<ForwardIterator InIter, OutputIterator<auto, InIter::~~value_type~~reference> OutIter>
  requires EqualityComparable<InIter::~~reference~~value_type>
  OutIter unique_copy(InIter first, InIter last,
                      OutIter result);

template<InputIterator InIter, ~~Mutable~~ForwardIterator OutIter>
  requires ~~HasCopyAssign~~OutputIterator<OutIter~~::reference~~, InIter::reference>
        && HasEqualTo<OutIter::~~reference~~value_type, InIter::value_type>
```

```
            && !ForwardIterator<InIter>
  OutIter unique_copy(InIter first, InIter last,
                      OutIter result);

template<InputIterator InIter, ~~OutputIterator<auto, InIter::value_type>~~ typename OutIter,
         Predicate<auto, InIter::value_type, InIter::value_type> Pred>
  requires OutputIterator<OutIter, InIter::reference>
        && OutputIterator<OutIter, const InIter::value_type&>
        && CopyAssignable<InIter::value_type>
        && CopyConstructible<InIter::value_type>
        && CopyConstructible<Pred>
        && !ForwardIterator<InIter>
        && !~~Mutable~~ForwardIterator<OutIter>
  OutIter unique_copy(InIter first, InIter last,
                      OutIter result, Pred pred);

template<ForwardIterator InIter, OutputIterator<auto, InIter::~~value_type~~reference> OutIter,
         Predicate<auto, InIter::~~reference~~value_type, InIter::~~reference~~value_type> Pred>
  requires CopyConstructible<Pred>
  OutIter unique_copy(InIter first, InIter last,
                      OutIter result);

template<InputIterator InIter, ~~Mutable~~ForwardIterator OutIter,
         Predicate<auto, OutIter::~~reference~~value_type, InIter::~~reference~~value_type> Pred>
  requires ~~HasCopyAssign~~OutputIterator<OutIter~~::reference~~, InIter::reference>
        && CopyConstructible<Pred>
        && !ForwardIterator<InIter>
  OutIter unique_copy(InIter first, InIter last,
                      OutIter result, Pred pred);
```

Note that we have split the two signatures of `unique_copy` into six signatures, to cover the actual variants required in the implementation and specify how partial ordering would occur.

5   *Requires:* The ranges [*first*, *last*) and [*result*, *result*+(*last*−*first*)) shall not overlap. ~~The expression~~ ~~*result = *first shall be valid. If neither InputIterator nor OutputIterator meets the requirements of forward iterator~~ ~~then the value type of InputIterator shall be CopyConstructible (20.1.3). Otherwise CopyConstructible is not~~ ~~required.~~

6   *Effects:* Copies only the first element from every consecutive group of equal elements referred to by the iterator i in the range [*first*, *last*) for which the following corresponding conditions hold: `*i == *(i - 1)` or `pred(*i, *(i - 1)) != false`.

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

8   *Complexity:* For nonempty ranges, exactly `last - first - 1` applications of the corresponding predicate.

25.2.10   Reverse   [alg.reverse]

```
template<BidirectionalIterator Iter>
  requires ~~SwappableIterator<Iter>~~HasSwap<Iter::reference, Iter::reference>
```

```
void reverse(Iter first, Iter last);
```

1   *Effects:* For each non-negative integer i <= (*last* − *first*)/2, applies `iter_swap` to all pairs of iterators *first* + i, (*last* − i) − 1.

2   ~~Requires: The type of *first shall satisfy the Swappable requirements (20.1.4).~~

3   *Complexity:* Exactly (*last* − *first*)/2 swaps.

```
template<BidirectionalIterator InIter, OutputIterator<auto, InIter::value_typereference> OutIter>
  OutIter reverse_copy(InIter first, InIter last, OutIter result);
```

4   *Effects:* Copies the range [*first*, *last*) to the range [*result*, *result*+(*last*−*first*)) such that for any non-negative integer i < (*last* − *first*) the following assignment takes place: *(*result* + (*last* − *first*) − i) = *(*first* + i).

5   *Requires:* The ranges [*first*, *last*) and [*result*, *result*+(*last*−*first*)) shall not overlap.

6   *Returns:* *result* + (*last* − *first*).

7   *Complexity:* Exactly *last* − *first* assignments.

25.2.11   Rotate   [alg.rotate]

```
template<ForwardIterator Iter>
  requires SwappableIterator<Iter>HasSwap<Iter::reference, Iter::reference>
  void rotate(Iter first, Iter middle,
              Iter last);
```

1   *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*).

2   *Remarks:* This is a left rotate.

3   *Requires:* [*first*, *middle*) and [*middle*, *last*) are valid ranges. ~~The type of *first shall satisfy the Swappable requirements (20.1.4), the `MoveConstructible` requirements (Table 33), and the `MoveAssignable` requirements (Table 35).~~

4   *Complexity:* At most *last* − *first* swaps.

```
template<ForwardIterator InIter, OutputIterator<auto, InIter::value_typereference> OutIter>
  OutIter rotate_copy(InIter first, InIter middle,
                      InIter last, OutIter result);
```

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

6   *Returns:* *result* + (*last* − *first*).

7   *Requires:* The ranges [*first*, *last*) and [*result*, *result* + (*last* − *first*)) shall not overlap.

8   *Complexity:* Exactly *last* − *first* assignments.

25.2.12   Random shuffle   [alg.random.shuffle]

```
template<RandomAccessIterator Iter>
  requires SwappableIterator<Iter>HasSwap<Iter::reference, Iter::reference>
  void random_shuffle(Iter first,
                      Iter last);

template<RandomAccessIterator Iter, Callable<auto, Iter::difference_type> Rand>
  requires SwappableIterator<Iter>HasSwap<Iter::reference, Iter::reference>
        && Convertible<Rand::result_type, Iter::difference_type>
        && CopyConstructible<Rand>
  void random_shuffle(Iter first,
                      Iter last,
                      Rand&& rand);

template<class RandomAccessIterator, class UniformRandomNumberGenerator>
  void random_shuffle(RandomAccessIterator first,
                      RandomAccessIterator last,
                      UniformRandomNumberGenerator& rand);
```

> TODO: We do not yet have the `UniformRandomNumberGenerator` concept, so we leave the third `random_shuffle` without concept constraints for now.

1   *Effects:* Shuffles the elements in the range [*first*, *last*) with uniform distribution.

2   ~~Requires: The type of *first shall satisfy the Swappable requirements (20.1.4).~~

3   *Complexity:* Exactly (*last* - *first*) - 1 swaps.

4   *Remarks:* The underlying source of random numbers for the first form of the function is implementation-defined. An implementation may use the `rand` function from the standard C library. The second form of the function takes a random number generating function object *rand* such that if n is an argument for rand, with a positive value, that has type ~~iterator_traits<RandomAccessIterator>::difference_type~~Iter::difference_type, then `rand(n)` returns a randomly chosen value, which lies in the interval (0,n]~~, and which has a type that is convertible to iterator_traits<RandomAccessIterator>:: difference_type~~.

25.2.13   Partitions   [alg.partitions]

```
template<BidirectionalIterator Iter, Predicate<auto, Iter::referencevalue_type> Pred>
  requires SwappableIterator<Iter>HasSwap<Iter::reference, Iter::reference>
        && CopyConstructible<Pred>
  Iter partition(Iter first, Iter last, Pred pred);
```

1   *Effects:* Places all the elements in the range [*first*, *last*) that satisfy *pred* before all the elements that do not satisfy it.

2   *Returns:* An iterator i such that for any iterator j in the range [*first*,i) *pred*(*j) != false, and for any iterator k in the range [i, *last*), *pred*(*k) == false.

3   ~~Requires: The type of *first shall satisfy the Swappable requirements (20.1.4), the `MoveConstructible` requirements (Table [moveconstructible]), and the the `MoveAssignable` requirements (Table [moveassignable]).~~

4   *Complexity:* At most (*last* − *first*)/2 swaps. Exactly *last* − *first* applications of the predicate are
        done.

```
template<BidirectionalIterator Iter, Predicate<auto, Iter::referencevalue_type> Pred>
  requires SwappableIterator<Iter>ShuffleIterator<Iter>
        && CopyConstructible<Pred>
  Iter stable_partition(Iter first, Iter last, Pred pred);
```

5   *Effects:* Places all the elements in the range [*first*, *last*) that satisfy *pred* before all the elements that do not
        satisfy it.

6   *Returns:* An iterator i such that for any iterator j in the range [*first*, i), *pred*(*j) != false, and for any
        iterator k in the range [i, *last*), *pred*(*k) == false. The relative order of the elements in both groups is
        preserved.

7   ~~Requires: The type of *first shall satisfy the Swappable requirements (20.1.4), the MoveConstructible requirements
        (Table [moveconstructible]), and the the MoveAssignable requirements (Table [moveassignable]).~~

8   *Complexity:* At most (*last* − *first*) * log(*last* − *first*) swaps, but only linear number of swaps if
        there is enough extra memory. Exactly *last* − *first* applications of the predicate.

25.3   Sorting and related operations   [alg.sorting]

25.3.1   Sorting   [alg.sort]

25.3.1.1   `sort`   [sort]

```
template<RandomAccessIterator Iter>
  requires SwappableIterator<Iter>ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void sort(Iter first, Iter last);

template<RandomAccessIterator Iter,
         PredicateStrictWeakOrder<auto, Iter::value_type, Iter::value_type> Compare>
  requires SwappableIterator<Iter>ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void sort(Iter first, Iter last,
            Compare comp);
```

1   *Effects:* Sorts the elements in the range [*first*, *last*).

2   ~~Requires: The type of *first shall satisfy the Swappable requirements (37), the MoveConstructible requirements
        (Table 33), and the the MoveAssignable requirements (Table 35).~~

3   *Complexity:* Approximately $N \log(N)$ (where N == *last* − *first*) comparisons on the average.[7]

25.3.1.2   `stable_sort`   [stable.sort]

---
[7] If the worst case behavior is important `stable_sort()` (25.3.1.2) or `partial_sort()` (25.3.1.3) should be used.

```
template<RandomAccessIterator Iter>
  requires SwappableIterator<Iter>ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void stable_sort(Iter first, Iter last);

template<RandomAccessIterator Iter,
        PredicateStrictWeakOrder<auto, Iter::value_type, Iter::value_type> Compare>
  requires SwappableIterator<Iter>ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void stable_sort(Iter first, Iter last,
                   Compare comp);
```

1   *Effects:* Sorts the elements in the range [*first*, *last*).

2   ~~Requires: The type of *first shall satisfy the Swappable requirements (20.1.4), the `MoveConstructible` requirements (Table 33), and the the `MoveAssignable` requirements (Table 35).~~

3   *Complexity:* It does at most $N \log^2(N)$ (where N == *last* - *first*) comparisons; if enough extra memory is available, it is $N \log(N)$.

4   *Remarks:* Stable.

25.3.1.3   partial_sort   [partial.sort]

```
template<RandomAccessIterator Iter>
  requires SwappableIterator<Iter>ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void partial_sort(Iter first,
                    Iter middle,
                    Iter last);
template<RandomAccessIterator Iter,
        PredicateStrictWeakOrder<auto, Iter::value_type, Iter::value_type> Compare>
  requires SwappableIterator<Iter>ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void partial_sort(Iter first,
                    Iter middle,
                    Iter last,
                    Compare comp);
```

1   *Effects:* Places the first *middle* - *first* sorted elements from the range [*first*, *last*) into the range [*first*, *middle*). The rest of the elements in the range [*middle*, *last*) are placed in an unspecified order.

2   ~~Requires: The type of *first shall satisfy the Swappable requirements (20.1.4), the `MoveConstructible` requirements (Table 33), and the the `MoveAssignable` requirements (Table 35).~~

3   *Complexity:* It takes approximately (*last* - *first*) * log(*middle* - *first*) comparisons.

25.3.1.4   partial_sort_copy   [partial.sort.copy]

```
template<InputIterator InIter, RandomAccessIterator RAIter>
  requires Has~~Copy~~Assign<RAIter::reference, InIter::~~value_type~~reference> &&
           ~~SwappableIterator<RAIter>~~Swappable<RAIter::reference> &&
           HasLess<InIter::value_type, RAIter::value_type> &&
           HasLess<RAIter::value_type, InIter::value_type> &&
           ~~HasLess~~LessThanComparable<RAIter::value_type>
  RAIter partial_sort_copy(InIter first, InIter last,
                           RAIter result_first, RAIter result_last);

template<InputIterator InIter, RandomAccessIterator RAIter, class Compare>
  requires Has~~Copy~~Assign<RAIter::reference, InIter::~~value_type~~reference> &&
           ~~SwappableIterator<RAIter>~~Swappable<RAIter::reference> &&
           Predicate<Compare, InIter::value_type, RAIter::value_type> &&
           Predicate<Compare, RAIter::value_type, InIter::value_type> &&
           ~~Predicate~~StrictWeakOrder<Compare, RAIter::value_type>
  RAIter partial_sort_copy(InIter first, InIter last,
                           RAIter result_first, RAIter result_last,
                           Compare comp);
```

1   *Effects:* Places the first min(`last - first`, `result_last - result_first`) sorted elements into the range [`result_first`, `result_first` + min(`last - first`, `result_last - result_first`)).

2   *Returns:* The smaller of: `result_last` or `result_first` + (`last - first`).

3   ~~*Requires:* The type of *result_first shall satisfy the Swappable requirements (37), the MoveConstructible requirements (Table 33), and the the MoveAssignable requirements (Table 35).~~

4   *Complexity:* Approximately (`last - first`) * log(min(`last - first`, `result_last - result_first`)) comparisons.

25.3.1.5   `is_sorted`   [is.sorted]

```
template<ForwardIterator Iter>
  requires LessThanComparable<Iter::value_type>
  bool is_sorted(Iter first, Iter last);
```

1   *Returns:* `is_sorted_until(first, last) == last`

```
template<ForwardIterator Iter,
         ~~Predicate~~StrictWeakOrder<auto, Iter::value_type~~, Iter::value_type~~> Compare>
  requires CopyConstructible<Compare>
  bool is_sorted(Iter first, Iter last,
                 Compare comp);
```

2   *Returns:* `is_sorted_until(first, last, comp) == last`

```
template<ForwardIterator Iter>
  requires LessThanComparable<Iter::value_type>
  Iter is_sorted_until(Iter first, Iter last);
template<ForwardIterator Iter,
         ~~Predicate~~StrictWeakOrder<auto, Iter::value_type~~, Iter::value_type~~> Compare>
```

```
requires CopyConstructible<Compare>
Iter is_sorted_until(Iter first, Iter last,
                     Compare comp);
```

3   *Returns:* If `distance(first, last) < 2`, returns `last`. Otherwise, returns the last iterator `i` in `[first,last]`
        for which the range `[first,i)` is sorted.

4   *Complexity:* Linear.

25.3.2   Nth element   [alg.nth.element]

```
template<RandomAccessIterator Iter>
  requires SwappableIterator<Iter>ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void nth_element(Iter first, Iter nth,
                   Iter last);

template<SwappableIterator<Iter>RandomAccessIterator Iter,
         PredicateStrictWeakOrder<auto, Iter::value_type, Iter::value_type> Compare>
  requires SameType<Iter::value_type&, Iter::reference> && Swappable<Iter::value_type> &&
           MoveConstructible<Iter::value_type> && MoveAssignable<Iter::value_type>
           ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void nth_element(Iter first, Iter nth,
                   Iter last,  Compare comp);
```

1   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. Also for any iterator `i` in the range `[first,nth)` and any iterator `j` in the range
        `[nth,last)` it holds that: `!(*i > *j)` or `comp(*j, *i) == false`.

2   ~~Requires: The type of *first shall satisfy the Swappable requirements (20.1.4), the `MoveConstructible` requirements
        (Table 33), and the the `MoveAssignable` requirements (Table 35).~~

3   *Complexity:* Linear on average.

25.3.3   Binary search   [alg.binary.search]

1   All of the algorithms in this section are versions of binary search and assume that the sequence being searched is
    partitioned with respect to an expression formed by binding the search key to an argument of the implied or explicit
    comparison function. They work on non-random access iterators minimizing the number of comparisons, which will
    be logarithmic for all types of iterators. They are especially appropriate for random access iterators, because these
    algorithms do a logarithmic number of steps through the data structure. For non-random access iterators they execute a
    linear number of steps.

25.3.3.1   `lower_bound`   [lower.bound]

```
template<ForwardIterator Iter, class T>
  requires HasLess<Iter::referencevalue_type, T>
  Iter lower_bound(Iter first, Iter last,
```

```
                    const T& value );
```

```
template<ForwardIterator Iter, class T, Predicate<auto, Iter::referencevalue_type, T> Compare>
  requires CopyConstructible<Compare>
  Iter lower_bound(Iter first, Iter last,
                   const T& value, Compare comp );
```

1   *Requires:* The elements e of [*first*, *last*) are partitioned with respect to the expression e < value or *comp*(e, value).

2   *Returns:* The furthermost iterator i in the range [*first*, *last*] such that for any iterator j in the range [*first*, i) the following corresponding conditions hold: *j < *value* or *comp*(*j, *value*) != false.

3   *Complexity:* At most log(*last* − *first*) + 1 comparisons.

25.3.3.2   upper_bound   [upper.bound]

```
template<ForwardIterator Iter, class T>
  requires HasLess<T, Iter::referencevalue_type>
  Iter upper_bound(Iter first, Iter last,
                   const T& value );
```

```
template<ForwardIterator Iter, class T, Predicate<auto, T, Iter::referencevalue_type> Compare>
  requires CopyConstructible<Compare>
  Iter upper_bound(Iter first, Iter last,
                   const T& value, Compare comp );
```

1   *Requires:* The elements e of [*first*, *last*) are partitioned with respect to the expression !(value < e) or !*comp*(value, e).

2   *Returns:* The furthermost iterator i in the range [*first*, *last*) such that for any iterator j in the range [*first*, i) the following corresponding conditions hold: !(value < *j) or *comp*(*value*, *j) == false.

3   *Complexity:* At most log(*last* − *first*) + 1 comparisons.

25.3.3.3   equal_range   [equal.range]

```
template<ForwardIterator Iter, class T>
  requires HasLess<T, Iter::referencevalue_type>
        && HasLess<Iter::referencevalue_type, T>
  pair<Iter, Iter>
    equal_range(Iter first,
                Iter last, const T& value );
```

```
template<ForwardIterator Iter, class T, classCopyConstructible Compare>
  requires Predicate<Compare, T, Iter::referencevalue_type>
        && Predicate<Compare, Iter::referencevalue_type, T>
  pair<Iter, Iter>
    equal_range(Iter first,
                Iter last, const T& value,
```

```
                    Compare comp );
```

1   *Requires:* The elements e of [*first*,*last*) are partitioned with respect to the expressions e < value and
     !(value < e) or *comp*(e, value) and !*comp*(value, e). Also, for all elements e of [*first*, *last*), e
     < value implies !(value < e) or *comp*(e, value) implies !*comp*(value, e).

2   *Returns:*

```
        make_pair(lower_bound(first, last, value),
                  upper_bound(first, last, value))
```

     or

```
        make_pair(lower_bound(first, last, value, comp),
                  upper_bound(first, last, value, comp))
```

3   *Complexity:* At most 2 * log(*last* − *first*) + 1 comparisons.

25.3.3.4   binary_search   [binary.search]

```
template<ForwardIterator Iter, class T>
  requires HasLess<T, Iter::referencevalue_type>
        && HasLess<Iter::referencevalue_type, T>
  bool binary_search(Iter first, Iter last,
                      const T& value );

template<ForwardIterator Iter, class T, classCopyConstructible Compare>
  requires Predicate<Compare, T, Iter::referencevalue_type>
        && Predicate<Compare, Iter::referencevalue_type, T>
  bool binary_search(Iter first, Iter last,
                      const T& value , Compare comp );
```

1   *Requires:* The elements e of [*first*,*last*) are partitioned with respect to the expressions e < value and
     !(value < e) or *comp*(e, value) and !*comp*(value, e). Also, for all elements e of [*first*, *last*), e
     < value implies !(value < e) or *comp*(e, value) implies !*comp*(value, e).

2   *Returns:* true if there is an iterator i in the range [*first*,*last*) that satisfies the corresponding condi-
     tions: !(*i < *value*) && !(*value* < *i) or *comp*(*i, *value*) == false && *comp*(*value*, *i) ==
     false.

3   *Complexity:* At most log(*last* − *first*) + 2 comparisons.

25.3.4   Merge   [alg.merge]

```
template<InputIterator InIter1, InputIterator InIter2,
         OutputIterator<auto, InIter1::value_type>typename OutIter>
  requires SameType<InIter1::value_type, InIter2::value_type>
          OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && HasLess<InIter2::value_type, InIter1::value_type>
```

```
            && LessThanComparable<InIter1::value_type>
            && LessThanComparable<InIter2::value_type>
    OutIter merge(InIter1 first1, InIter1 last1,
                  InIter2 first2, InIter2 last2,
                  OutIter result);

template<InputIterator InIter1, InputIterator InIter2,
         OutputIterator<auto, InIter1::value_type>typename OutIter,
         Predicate<auto, InIter12::value_type, InIter21::value_type> Compare>
    requires SameType<InIter1::value_type, InIter2::value_type>
            OutputIterator<OutIter, InIter1::reference>
         && OutputIterator<OutIter, InIter2::reference>
         && CopyConstructible<Compare>
         && StrictWeakOrder<Compare, InIter1::value_type>
         && StrictWeakOrder<Compare, InIter2::value_type>
    OutIter merge(InIter1 first1, InIter1 last1,
                  InIter2 first2, InIter2 last2,
                  OutIter result, Compare comp);
```

1   *Effects:* Merges two sorted ranges [*first1*,*last1*) and [*first2*,*last2*) into the range [*result*,*result* + (*last1* - *first1*) + (*last2* - *first2*)).

2   The resulting range shall not overlap with either of the original ranges. The list will be sorted in non-decreasing order according to the ordering defined by *comp*; that is, for every iterator i in [*first*,*last*) other than *first*, the condition *i < *(i - 1) or *comp*(*i, *(i - 1)) will be false.

3   *Returns:* *result* + (*last1* - *first1*) + (*last2* - *first2*).

4   *Complexity:* At most (*last1* - *first1*) + (*last2* - *first2*) - 1 comparisons.

5   *Remarks:* Stable.

```
template<BidirectionalIterator Iter>
  requires SwappableIterator<Iter>ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void inplace_merge(Iter first,
                     Iter middle,
                     Iter last);

template<BidirectionalIterator Iter,
         PredicateStrictWeakOrder<auto, Iter::value_type, Iter::value_type> Compare>
  requires SwappableIterator<Iter>ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void inplace_merge(Iter first,
                     Iter middle,
                     Iter last, Compare comp);
```

6   *Effects:* Merges two sorted consecutive ranges [*first*,*middle*) and [*middle*,*last*), putting the result of the merge into the range [*first*,*last*). The resulting range will be in non-decreasing order; that is, for every iterator i in [*first*,*last*) other than *first*, the condition *i < *(i - 1) or, respectively, *comp*(*i, *(i - 1)) will be false.

7   ~~Requires: The type of *first shall satisfy the Swappable requirements (20.1.4), the `MoveConstructible` requirements (Table 33), and the the `MoveAssignable` requirements (Table 35).~~

8   *Complexity:* When enough additional memory is available, (`last - first`) - 1 comparisons. If no additional memory is available, an algorithm with complexity $N \log(N)$ (where `N` is equal to `last - first`) may be used.

9   *Remarks:* Stable.

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

1   This section defines all the basic set operations on sorted structures. They also work with `multisets` ([multiset]) 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.

25.3.5.1   `includes`   [includes]

```
template<InputIterator Iter1, InputIterator Iter2>
  requires SameType<Iter1::value_type, Iter2::value_type> && LessThanComparable<Iter1::value_type>
          HasLess<Iter1::value_type, Iter2::value_type>
       && HasLess<Iter2::value_type, Iter1::value_type>
       && LessThanComparable<Iter1::value_type>
       && LessThanComparable<Iter2::value_type>
  bool includes(Iter1 first1, Iter1 last1,
                Iter2 first2, Iter2 last2);

template<InputIterator Iter1, InputIterator Iter2,
         Predicate<auto, Iter1::value_type, Iter2::value_type>typename Compare>
  requires SameType<Iter1::value_type, Iter2::value_type>
          Predicate<Compare, Iter1::value_type, Iter2::value_type>
       && Predicate<Compare, Iter2::value_type, Iter1::value_type>
       && StrictWeakOrder<Compare, Iter1::value_type>
       && StrictWeakOrder<Compare, Iter2::value_type>
  bool includes(Iter1 first1, Iter1 last1,
                Iter2 first2, Iter2 last2,
                Compare comp);
```

1   *Returns:* `true` if every element in the range [`first2`,`last2`) is contained in the range [`first1`,`last1`). Returns `false` otherwise.

2   *Complexity:* At most 2 * ((`last1 - first1`) + (`last2 - first2`)) - 1 comparisons.

25.3.5.2   `set_union`   [set.union]

```
template<InputIterator InIter1, InputIterator InIter2,
         OutputIterator<auto, InIter1::value_type>typename OutIter>
  requires SameType<InIter1::value_type, InIter2::value_type>
       && LessThanComparable<InIter1::value_type>
          OutputIterator<OutIter, InIter1::reference>
```

```
          && OutputIterator<OutIter, InIter2::reference>
          && HasLess<InIter2::value_type, InIter1::value_type>
          && HasLess<InIter1::value_type, InIter2::value_type>
          && LessThanComparable<InIter1::value_type>
          && LessThanComparable<InIter2::value_type>
      OutIter set_union(InIter1 first1, InIter1 last1,
                        InIter2 first2, InIter2 last2,
                        OutIter result);

  template<InputIterator InIter1, InputIterator InIter2,
          OutputIterator<auto, InIter1::value_type>typename OutIter,
          Predicate<auto, InIter1::value_type, InIter2::value_type>CopyConstructible Compare>
    requires SameType<InIter1::value_type, InIter2::value_type>
             OutputIterator<OutIter, InIter1::reference>
          && OutputIterator<OutIter, InIter2::reference>
          && Predicate<Compare, InIter1::value_type, InIter2::value_type>
          && Predicate<Compare, InIter2::value_type, InIter1::value_type>
          && StrictWeakOrder<Compare, InIter1::value_type>
          && StrictWeakOrder<Compare, InIter2::value_type>
      OutIter set_union(InIter1 first1, InIter1 last1,
                        InIter2 first2, InIter2 last2,
                        OutIter result, Compare comp);
```

1   *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.

2   *Requires:* The resulting range shall not overlap with either of the original ranges.

3   *Returns:* The end of the constructed range.

4   *Complexity:* At most 2 * ((`last1` - `first1`) + (`last2` - `first2`)) - 1 comparisons.

5   *Remarks:* If [`first1`,`last1`) contains m elements that are equivalent to each other and [`first2`,`last2`) contains n elements that are equivalent to them, then all m elements from the first range shall be copied to the output range, in order, and then $\max(n-m,0)$ elements from the second range shall be copied to the output range, in order.

25.3.5.3 `set_intersection`   **[set.intersection]**

```
  template<InputIterator InIter1, InputIterator InIter2,
          OutputIterator<auto, InIter1::value_type>typename OutIter>
    requires SameType<InIter1::value_type, InIter2::value_type>
          && LessThanComparable<InIter1::value_type>
             OutputIterator<OutIter, InIter1::reference>
          && OutputIterator<OutIter, InIter2::reference>
          && HasLess<InIter2::value_type, InIter1::value_type>
          && HasLess<InIter1::value_type, InIter2::value_type>
          && LessThanComparable<InIter1::value_type>
          && LessThanComparable<InIter2::value_type>
      OutIter set_intersection(InIter1 first1, InIter1 last1,
```

```
                             InIter2 first2, InIter2 last2,
                             OutIter result);

template<InputIterator InIter1, InputIterator InIter2,
        OutputIterator<auto, InIter1::value_type>typename OutIter,
        Predicate<auto, InIter1::value_type, InIter2::value_type>CopyConstructible Compare>
  requires SameType<InIter1::value_type, InIter2::value_type>
          OutputIterator<OutIter, InIter1::reference>
       && OutputIterator<OutIter, InIter2::reference>
       && Predicate<Compare, InIter1::value_type, InIter2::value_type>
       && Predicate<Compare, InIter2::value_type, InIter1::value_type>
       && StrictWeakOrder<Compare, InIter1::value_type>
       && StrictWeakOrder<Compare, InIter2::value_type>
  OutIter set_intersection(InIter1 first1, InIter1 last1,
                           InIter2 first2, InIter2 last2,
                           OutIter result, Compare comp);
```

1   *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.

2   *Requires:* The resulting range shall not overlap with either of the original ranges.

3   *Returns:* The end of the constructed range.

4   *Complexity:* At most 2 * ((*last1* − *first1*) + (*last2* − *first2*)) − 1 comparisons.

5   *Remarks:* If [`first1,last1`) contains *m* elements that are equivalent to each other and [`first2,last2`) contains *n* elements that are equivalent to them, the first $\min(m,n)$ elements shall be copied from the first range to the output range, in order.

25.3.5.4 `set_difference`   [set.difference]

```
template<InputIterator InIter1, InputIterator InIter2,
        OutputIterator<auto, InIter1::value_type>typename OutIter>
  requires SameType<InIter1::value_type, InIter2::value_type>
       && LessThanComparable<InIter1::value_type>
          OutputIterator<OutIter, InIter1::reference>
       && OutputIterator<OutIter, InIter2::reference>
       && HasLess<InIter2::value_type, InIter1::value_type>
       && HasLess<InIter1::value_type, InIter2::value_type>
       && LessThanComparable<InIter1::value_type>
       && LessThanComparable<InIter2::value_type>
  OutIter set_difference(InIter1 first1, InIter1 last1,
                         InIter2 first2, InIter2 last2,
                         OutIter result);

template<InputIterator InIter1, InputIterator InIter2,
        OutputIterator<auto, InIter1::value_type>typename OutIter,
        Predicate<auto, InIter1::value_type, InIter2::value_type>CopyConstructible Compare>
  requires SameType<InIter1::value_type, InIter2::value_type>
```

```
        OutputIterator<OutIter, InIter1::reference>
    && OutputIterator<OutIter, InIter2::reference>
    && Predicate<Compare, InIter1::value_type, InIter2::value_type>
    && Predicate<Compare, InIter2::value_type, InIter1::value_type>
    && StrictWeakOrder<Compare, InIter1::value_type>
    && StrictWeakOrder<Compare, InIter2::value_type>
  OutIter set_difference(InIter1 first1, InIter1 last1,
                         InIter2 first2, InIter2 last2,
                         OutIter result, Compare comp);
```

1   *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.

2   *Requires:* The resulting range shall not overlap with either of the original ranges.

3   *Returns:* The end of the constructed range.

4   *Complexity:* At most 2 * ((*last1* - *first1*) + (*last2* - *first2*)) - 1 comparisons.

5   *Remarks:* If [`first1`,`last1`) contains m elements that are equivalent to each other and [`first2`,`last2`) contains n elements that are equivalent to them, the last $\max(m-n,0)$ elements from [`first1`,`last1`) shall be copied to the output range.

25.3.5.5 `set_symmetric_difference`   [set.symmetric.difference]

```
template<InputIterator InIter1, InputIterator InIter2,
         OutputIterator<auto, InIter1::value_type>typename OutIter>
  requires SameType<InIter1::value_type, InIter2::value_type>
       && LessThanComparable<InIter1::value_type>
         OutputIterator<OutIter, InIter1::reference>
    && OutputIterator<OutIter, InIter2::reference>
    && HasLess<InIter2::value_type, InIter1::value_type>
    && HasLess<InIter1::value_type, InIter2::value_type>
    && LessThanComparable<InIter1::value_type>
    && LessThanComparable<InIter2::value_type>
  OutIter set_symmetric_difference(InIter1 first1, InIter1 last1,
                                   InIter2 first2, InIter2 last2,
                                   OutIter result);

template<InputIterator InIter1, InputIterator InIter2,
         OutputIterator<auto, InIter1::value_type>typename OutIter,
         Predicate<auto, InIter1::value_type, InIter2::value_type>CopyConstructible Compare>
  requires SameType<InIter1::value_type, InIter2::value_type>
         OutputIterator<OutIter, InIter1::reference>
    && OutputIterator<OutIter, InIter2::reference>
    && Predicate<Compare, InIter1::value_type, InIter2::value_type>
    && Predicate<Compare, InIter2::value_type, InIter1::value_type>
    && StrictWeakOrder<Compare, InIter1::value_type>
    && StrictWeakOrder<Compare, InIter2::value_type>
  OutIter set_symmetric_difference(InIter1 first1, InIter1 last1,
```

```
                              InIter2 first2, InIter2 last2,
                              OutIter result, Compare comp);
```

1   *Effects:* Copies the elements of the range [*first1*, *last1*) which are not present in the range [*first2*, *last2*), and the elements of the range [*first2*, *last2*) which are not present in the range [*first1*, *last1*) to the range beginning at `result`. The elements in the constructed range are sorted.

2   *Requires:* The resulting range shall not overlap with either of the original ranges.

3   *Returns:* The end of the constructed range.

4   *Complexity:* At most 2 * ((*last1* - *first1*) + (*last2* - *first2*)) - 1 comparisons.

5   *Remarks:* If [first1,last1) contains *m* elements that are equivalent to each other and [first2,last2) contains *n* elements that are equivalent to them, then $|m - n|$ 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 $m < n$.

25.3.6   Heap operations   [alg.heap.operations]

1   A *heap* is a particular organization of elements in a range between two random access iterators [`a`,`b`). Its two key properties are:

(1) There is no element greater than `*a` in the range and

(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.

25.3.6.1   push_heap   [push.heap]

```
template<RandomAccessIterator Iter>
  requires SwappableIterator<Iter>ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void push_heap(Iter first, Iter last);

template<RandomAccessIterator Iter,
         PredicateStrictWeakOrder<auto, Iter::value_type, Iter::value_type> Compare>
  requires SwappableIterator<Iter>ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void push_heap(Iter first, Iter last,
                 Compare comp);
```

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

2   *Requires:* The range [*first*, *last* - 1) shall be a valid heap. ~~The type of *first shall satisfy the MoveConstructible requirements (Table [moveconstructible]) and the the MoveAssignable requirements (Table [moveassignable]).~~

3   *Complexity:* At most log(*last* - *first*) comparisons.

25.3.6.2 `pop_heap`   **[pop.heap]**

```
template<RandomAccessIterator Iter>
  requires SwappableIterator<Iter>ShuffleIterator<Iter> && LessThanComparable<Iter::value_type>
  void pop_heap(Iter first, Iter last);

template<RandomAccessIterator Iter,
         PredicateStrictWeakOrder<auto, Iter::value_type, Iter::value_type> Compare>
  requires SwappableIterator<Iter>ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void pop_heap(Iter first, Iter last,
                Compare comp);
```

1   *Effects:* Swaps the value in the location *first* with the value in the location *last* - 1 and makes [*first*,*last*
      - 1) into a heap.

2   *Requires:* The range [*first*,*last*) shall be a valid heap.   ~~The type of *first shall satisfy the Swappable
      requirements (20.1.4), the MoveConstructible requirements (Table [moveconstructible]), and the the MoveAssignable
      requirements (Table [moveassignable]).~~

3   *Complexity:* At most 2 * log(*last* - *first*) comparisons.

25.3.6.3 `make_heap`   **[make.heap]**

```
template<RandomAccessIterator Iter>
  requires SwappableIterator<Iter>ShuffleIterator<Iter> &&
           LessThanComparable<Iter::value_type>
  void make_heap(Iter first, Iter last);

template<RandomAccessIterator Iter,
         PredicateStrictWeakOrder<auto, Iter::value_type, Iter::value_type> Compare>
  requires SwappableIterator<Iter>ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void make_heap(Iter first, Iter last,
                 Compare comp);
```

1   *Effects:* Constructs a heap out of the range [*first*,*last*).

2   *~~Requires:~~* ~~The type of *first shall satisfy the MoveConstructible requirements (Table [moveconstructible])
      and the the MoveAssignable requirements (Table [moveassignable]).~~

3   *Complexity:* At most 3 * (*last* - *first*) comparisons.

25.3.6.4 `sort_heap`   **[sort.heap]**

```
template<RandomAccessIterator Iter>
  requires SwappableIterator<Iter>ShuffleIterator<Iter> && LessThanComparable<Iter::value_type>
  void sort_heap(Iter first, Iter last);

template<RandomAccessIterator Iter,
```

```
            PredicateStrictWeakOrder<auto, Iter::value_type, Iter::value_type> Compare>
    requires SwappableIterator<Iter>ShuffleIterator<Iter>
        && CopyConstructible<Compare>
    void sort_heap(Iter first, Iter last,
                    Compare comp);
```

1   *Effects:* Sorts elements in the heap [*first*, *last*).

2   ~~Requires: The type of *first shall satisfy the Swappable requirements (20.1.4), the MoveConstructible requirements~~
        ~~(Table [moveconstructible]), and the the MoveAssignable requirements (Table [moveassignable]).~~

3   *Complexity:* At most $N \log(N)$ comparisons (where N == *last* - *first*).

25.3.6.5   is_heap   **[is.heap]**

```
template<RandomAccessIterator Iter>
  requires LessThanComparable<Iter::value_type>
  bool is_heap(Iter first, Iter last);
```

1   *Returns:* is_heap_until(first, last) == last

```
template<RandomAccessIterator Iter,
        PredicateStrictWeakOrder<auto, Iter::value_type, Iter::value_type> Compare>
  requires CopyConstructible<Compare>
  bool is_heap(Iter first, Iter last, Compare comp);
```

2   *Returns:* is_heap_until(first, last, comp) == last

```
template<RandomAccessIterator Iter>
  Iter is_heap_until(Iter first, Iter last);
template<RandomAccessIterator Iter,
        PredicateStrictWeakOrder<auto, Iter::value_type, Iter::value_type> Compare>
  requires CopyConstructible<Compare>
  Iter is_heap_until(Iter first, Iter last,
                    Compare comp);
```

3   *Returns:* If distance(first, last) < 2, returns last. Otherwise, returns the last iterator i in [first,last]
        for which the range [first,i) is a heap.

4   *Complexity:* Linear.

25.3.7   Minimum and maximum   **[alg.min.max]**

```
template<LessThanComparable T> const T& min(const T& a, const T& b);
template<class T, Predicate<auto, T, T>StrictWeakOrder<auto, T> Compare>
  requires !SameType<T, Compare> && CopyConstructible<Compare>
  const T& min(const T& a, const T& b, Compare comp);
```

1   ~~Requires: Type T is LessThanComparable (20.1.2).~~

2   *Returns:* The smaller value.

3   *Remarks:* Returns the first argument when the arguments are equivalent.

```
template<LessThanComparable T> const T& min(const T& a, const T& b, const T& c);
template<LessThanComparable T, LessThanComparable... Args>
  requires SameType<T, Args>...
  const T& min(const T& a, const Args&... args);
```

4   *Requires:* Type T is LessThanComparable, and all types forming Args... are the same as T.

5   *Returns:* The smallest value in the set of all arguments.

6   *Remarks:* Returns the leftmost argument when several arguments are equivalent to the smallest. Returns *a* if
           sizeof...(Args) is 0.

```
template<class T, class U, class... Args>
  const T& min(const T& a, const U& b, const Args&... args);
```

At present, we do not know how to write concept constraints for this variant of min, because we cannot directly express
the idea of splitting Args into its first N-1 arguments (all of which are the same at T) and its Nth argument (the binary
predicate). Most likely this is possible with what is effectively metaprogramming of the concept system to "walk"
through the arguments, but such an implementation would make a poor specification. For now, we leave this version
unconstrained.

7   *Requires:* The types of all arguments except the last one are the same as T. The last argument is a binary predicate
           over T.

8   *Returns:* The first element in a partial ordering of all the arguments except the last one, where the ordering is
           defined by the predicate.

9   *Remarks:* Returns the leftmost argument when several arguments are equivalent to the first element in the ordering.
           Returns *a* if sizeof...(Args) is 0.

```
template<LessThanComparable T> const T& max(const T& a, const T& b);
template<class T, Predicate<auto, T, T>StrictWeakOrder<auto, T> Compare>
  requires !SameType<T, Compare> && CopyConstructible<Compare>
  const T& max(const T& a, const T& b, Compare comp);
```

10   Requires: Type T is LessThanComparable (20.1.2).

11   *Returns:* The larger value.

12   *Remarks:* Returns the first argument when the arguments are equivalent.

```
template<LessThanComparable T> const T& max(const T& a, const T& b, const T& c);
template<LessThanComparable T, LessThanComparable... Args>
  requires SameType<T, Args>...
  const T& max(const T& a, const Args&... args);
```

13   *Requires:* Type T is LessThanComparable, and all types forming Args... are the same as T.

14   *Returns:* The largest value in the set of all arguments.

15   *Remarks:* Returns the leftmost argument when several arguments are equivalent to the largest. Returns *a* if
           sizeof...(Args) is 0.

```
template<class T, class U, class... Args>
  const T& max(const T& a, const U& b, const Args&... args);
```

As with the corresponding min function, we do not know how to write the constraints, so we leave this version unconstrained.

16   *Requires:* The types of all arguments except the last one are the same as T. The last argument is a binary predicate over T.

17   *Returns:* The last element in a partial ordering of all the arguments except the last one, where the ordering is defined by the predicate.

18   *Remarks:* Returns the leftmost argument when several arguments are equivalent to the last element in the ordering. Returns *a* if sizeof...(Args) is 0.

```
template<LessThanComparable T> pair<const T&, const T&> minmax(const T& a, const T& b);
template<class T, Predicate<auto, T, T>StrictWeakOrder<auto, T> Compare>
  requires !SameType<T, Compare> && CopyConstructible<Compare>
  pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
```

19   *Requires:* Type T shall be LessThanComparable ([lessthancomparable]).

20   *Returns:* pair<const T&, const T&>(b, a) if b is smaller than a, and pair<const T&, const T&>(a, b) otherwise.

21   *Remarks:* Returns <pair<const T&, const T&>(a, b) when the arguments are equivalent.

22   *Complexity:* Exactly one comparison.

```
template<LessThanComparable T>
  pair<const T&, const T&> minmax(const T& a, const T& b, const T& c);
template<LessThanComparable T, LessThanComparable... Args>
  requires SameType<T, Args>...
  pair<const T&, const T&> minmax(const T& a, const Args&... args);
```

23   *Requires:* Type T is LessThanComparable, and all types forming Args... are the same as T.

24   *Returns:* pair<const T&, const T&>(x, y) where x is the first element and y the last element in a partial ordering of all the arguments.

25   *Remarks:* x is the leftmost argument when several arguments are equivalent to the smallest. y is the rightmost argument when several arguments are equivalent to the largest. Returns pair<const T&, const T&>(a, a) if sizeof...(Args) is 0.

26   *Complexity:* At most (3/2)sizeof...(Args) applications of the corresponding predicate.

```
template<class T, class U, class...  Args>
  pair<const T&, const T&> minmax(const T& a, const U& b, const Args&...  args);
```

As with the corresponding min and max functions, we do not know how to write the constraints, so we leave this version unconstrained.

27   *Requires:* The types of all arguments except the last one are the same as T. The last argument is a binary predicate over T.

28   *Returns:* pair<const T&, const T&>(x, y) where x is the first element and y the last element in a partial
         ordering of all the arguments defined by the predicate.

29   *Remarks:* x is the leftmost argument when several arguments would order equivalent as the first in the ordering.
         y is the rightmost argument when several arguments would order equivalent as the last in the ordering. Returns
         pair<const T&, const T&>(a, a) if sizeof...(Args) is 0.

30   *Complexity:* At most (3/2)sizeof...(Args) applications of the corresponding predicate.

```
template<ForwardIterator Iter>
  requires LessThanComparable<Iter::referencevalue_type>
  Iter min_element(Iter first, Iter last);

template<ForwardIterator Iter,
         PredicateStrictWeakOrder<auto, Iter::referencevalue_type, Iter::reference> Compare>
  requires CopyConstructible<Compare>
  Iter min_element(Iter first, Iter last,
                   Compare comp);
```

31   *Returns:* The first iterator i in the range [*first*, *last*) such that for any iterator j in the range [*first*, *last*
) the following corresponding conditions hold: !(*j < *i) or *comp*(*j, *i) == false. Returns *last* if
         first == *last*.

32   *Complexity:* Exactly max((*last* - *first*) - 1, 0) applications of the corresponding comparisons.

```
template<ForwardIterator Iter>
  requires LessThanComparable<Iter::referencevalue_type>
  Iter max_element(Iter first, Iter last);

template<ForwardIterator Iter,
         PredicateStrictWeakOrder<auto, Iter::referencevalue_type, Iter::reference> Compare>
  requires CopyConstructible<Compare>
  Iter max_element(Iter first, Iter last,
                   Compare comp);
```

33   *Returns:* The first iterator i in the range [*first*, *last*) such that for any iterator j in the range [*first*, *last*
) the following corresponding conditions hold: !(*i < *j) or *comp*(*i, *j) == false. Returns *last* if
         first == *last*.

34   *Complexity:* Exactly max((*last* - *first*) - 1, 0) applications of the corresponding comparisons.

```
template<ForwardIterator Iter>
  requires LessThanComparable<Iter::value_type>
  pair<Iter, Iter>
    minmax_element(Iter first, Iter last);
template<ForwardIterator Iter,
         PredicateStrictWeakOrder<auto, Iter::value_type, Iter::value_type> Compare>
  requires CopyConstructible<Compare>
  pair<Iter, Iter>
    minmax_element(Iter first, Iter last, Compare comp);
```

35   *Returns:* make_pair(m, M), where m is min_element(first, last) or min_element(first, last, comp)

and M is `max_element(first, last)` or `max_element(first, last, comp)`.

36   *Complexity:* At most `max(2 * (last - first) - 2, 0)` applications of the corresponding comparisons.

25.3.8   Lexicographical comparison   [alg.lex.comparison]

```
template<InputIterator Iter1, InputIterator Iter2>
  requires HasLess<Iter1::referencevalue_type, Iter2::referencevalue_type>
        && HasLess<Iter2::referencevalue_type, Iter1::referencevalue_type>
  bool lexicographical_compare(Iter1 first1, Iter1 last1,
                               Iter2 first2, Iter2 last2);

template<InputIterator Iter1, InputIterator Iter2, classCopyConstructible Compare>
  requires Predicate<Compare, Iter1::referencevalue_type, Iter2::referencevalue_type>
        && Predicate<Compare, Iter2::referencevalue_type, Iter1::referencevalue_type>
  bool lexicographical_compare(Iter1 first1, Iter1 last1,
                               Iter2 first2, Iter2 last2,
                               Compare comp);
```

1   *Returns:* `true` if the sequence of elements defined by the range `[first1, last1)` is lexicographically less than the sequence of elements defined by the range `[first2, last2)`.

Returns `false` otherwise.

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

3   *Remarks:* If two sequences have the same number of elements and their corresponding elements are equivalent, then neither sequence is lexicographically less than the other. If one sequence is a prefix of the other, then the shorter sequence is lexicographically less than the longer sequence. Otherwise, the lexicographical comparison of the sequences yields the same result as the comparison of the first corresponding pair of elements that are not equivalent.

```
for ( ; first1 != last1 && first2 != last2 ; ++first1, ++first2) {
  if (*first1 < *first2) return true;
  if (*first2 < *first1) return false;
}
return first1 == last1 && first2 != last2;
```

25.3.9   Permutation generators   [alg.permutation.generators]

```
template<BidirectionalIterator Iter>
  requires SwappableIterator<Iter>HasSwap<Iter::reference, Iter::reference>
        LessThanComparable<Iter::referencevalue_type>
  bool next_permutation(Iter first, Iter last);

template<BidirectionalIterator Iter,
        PredicateStrictWeakOrder<auto, Iter::referencevalue_type, Iter::reference> Compare>
  requires SwappableIterator<Iter>HasSwap<Iter::reference, Iter::reference>
```

```
        && CopyConstructible<Compare>
  bool next_permutation(Iter first, Iter last, Compare comp);
```

1   *Effects:* Takes a sequence defined by the range [*first*, *last*) and transforms it into the next permutation. The next permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `operator<` or *comp*. If such a permutation exists, it returns `true`. Otherwise, it transforms the sequence into the smallest permutation, that is, the ascendingly sorted one, and returns `false`.

2   ~~Requires: The type of first shall satisfy the Swappable requirements (20.1.4).~~

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

```
template<BidirectionalIterator Iter>
  requires SwappableIterator<Iter>HasSwap<Iter::reference, Iter::reference>
          LessThanComparable<Iter::referencevalue_type>
  bool prev_permutation(Iter first, Iter last);

template<BidirectionalIterator Iter,
        PredicateStrictWeakOrder<auto, Iter::referencevalue_type, Iter::reference> Compare>
  requires SwappableIterator<Iter>HasSwap<Iter::reference, Iter::reference>
        && CopyConstructible<Compare>
  bool prev_permutation(Iter first, Iter last, Compare comp);
```

4   *Effects:* Takes a sequence defined by the range [*first*, *last*) and transforms it into the previous permutation. The previous permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `operator<` or *comp*.

5   *Returns:* `true` if such a permutation exists. Otherwise, it transforms the sequence into the largest permutation, that is, the descendingly sorted one, and returns `false`.

6   ~~Requires: The type of *first shall satisfy the Swappable requirements (20.1.4).~~

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

Bibliography

[1] Douglas Gregor, Bjarne Stroustrup, James Widman, and Jeremy Siek. Proposed wording for concepts (revision 6). Technical Report N2676=08-0186, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, June 2008.

Draft