Concepts for the C++0x Standard Library: Algorithms (Revision 5)

Authors: Douglas Gregor, Indiana University
Mat Marcus, Adobe Systems, Inc.
Thomas Witt, Zephyr Associates, Inc.
Andrew Lumsdaine, Indiana University
Document number: N2759=08-0269
Revises document number: N2740=08-0250
Date: 2008-09-19
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 (N2691). 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 N2740

— Eliminate the use of the term "iterator category", instead referring to the appropriate iterator concept (Clause **??**). Thanks to Alisdair Meredith for pointing out the inconsistency.

— Added a placeholder `UniformRandomNumberGenerator` concept, which is used to constrain the third `random_shuffle` overload. Made the third `random_shuffle` accept its random number generator by rvalue reference.

— Now that `ShuffleIterator` requires `ForwardIterator`, simplified the `rotate` algorithm to only directly require `ShuffleIterator`.

Proposed Wording

Issues resolved by concepts

The following LWG are resolved by concepts. These issues should be resolved as NAD following the application of this proposal to the wording paper:

Issue 631. Conflicting requirements for BinaryPredicate. The concepts specification for `lower_bound` and `upper_bound` specify an exact order for the operands, following the principle that we specify minimal requirements for an algorithm (within reason).

Issue 779. Resolution of #283 incomplete. The concepts proposal spells out the exact requirements for the `remove` algorithm.

Chapter 25 Algorithms library [algorithms]

1 This clause describes components that C++ programs may use to perform algorithmic operations on containers (clause **??**) 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.

Table 1: Algorithms library summary

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

Header `<algorithm>` synopsis

```
namespace std {
  // 25.1, non-modifying sequence operations:
  template <InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
    requires CopyConstructible<Pred>
    bool all_of(Iter first, Iter last, Pred pred);
  template <InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
    requires CopyConstructible<Pred>
    bool any_of(Iter first, Iter last, Pred pred);
  template <InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
    requires CopyConstructible<Pred>
    bool none_of(Iter first, Iter last, Pred pred);

  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::value_type, T>
    Iter find(Iter first, Iter last, const T& value);
  template<InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
    requires CopyConstructible<Pred>
    Iter find_if(Iter first, Iter last, Pred pred);
  template<InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
```

```
  requires CopyConstructible<Pred>
  Iter find_if_not(Iter first, Iter last, Pred pred);
template<ForwardIterator Iter1, ForwardIterator Iter2>
  requires HasEqualTo<Iter1::value_type, Iter2::value_type>
  Iter1 find_end(Iter1 first1, Iter1 last1,
                 Iter2 first2, Iter2 last2);
template<ForwardIterator Iter1, ForwardIterator Iter2,
         Predicate<auto, Iter1::value_type, Iter2::value_type> Pred>
  requires CopyConstructible<Pred>
  Iter1 find_end(Iter1 first1, Iter1 last1,
                 Iter2 first2, Iter2 last2,
                 Pred pred);

template<InputIterator Iter1, ForwardIterator Iter2>
  requires HasEqualTo<Iter1::value_type, Iter2::value_type>
  Iter1 find_first_of(Iter1 first1, Iter1 last1,
                      Iter2 first2, Iter2 last2);
template<InputIterator Iter1, ForwardIterator Iter2,
         Predicate<auto, Iter1::value_type, Iter2::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::value_type>
  Iter adjacent_find(Iter first, Iter last);
template<ForwardIterator Iter, EquivalenceRelation<auto, Iter::value_type> Pred>
  requires CopyConstructible<Pred>
  Iter adjacent_find(Iter first, Iter last, Pred pred);

template<InputIterator Iter, class T>
  requires HasEqualTo<Iter::value_type, T>
  Iter::difference_type count(Iter first, Iter last, const T& value);
template<InputIterator Iter, Predicate<auto, Iter::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::value_type, Iter2::value_type>
  pair<Iter1, Iter2> mismatch(Iter1 first1, Iter1 last1,
                              Iter2 first2);
template<InputIterator Iter1, InputIterator Iter2,
         Predicate<auto, Iter1::value_type, Iter2::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::value_type, Iter2::value_type>
```

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

template<ForwardIterator Iter1, ForwardIterator Iter2>
  requires HasEqualTo<Iter1::value_type, Iter2::value_type>
  Iter1 search(Iter1 first1, Iter1 last1,
               Iter2 first2, Iter2 last2);
template<ForwardIterator Iter1, ForwardIterator Iter2,
         Predicate<auto, Iter1::value_type, Iter2::value_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::value_type, T>
  Iter search_n(Iter first, Iter last, Iter::difference_type count,
                const T& value);
template<ForwardIterator Iter, class T,
         Predicate<auto, Iter::value_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<InputIterator InIter, OutputIterator<auto, InIter::reference> OutIter>
  OutIter copy_n(InIter first, InIter::difference_type n,
                 OutIter result);
template<InputIterator InIter, OutputIterator<auto, InIter::reference> OutIter,
         Predicate<auto, InIter::value_type> Pred>
  requires CopyConstructible<Pred>
  OutIter copy_if(InIter first, InIter last,
                  OutIter result, Pred pred);
template<BidirectionalIterator InIter, BidirectionalIterator OutIter>
  requires OutputIterator<OutIter, InIter::reference>
  OutIter copy_backward(InIter first, InIter last,
                        OutIter result);

// 25.2.2, move:
template<InputIterator InIter, typename OutIter>
  requires OutputIterator<OutIter, RvalueOf<InIter::reference>::type>
  OutIter move(InIter first, InIter last,
```

```
              OutIter result);
template<BidirectionalIterator InIter, BidirectionalIterator OutIter>
  requires OutputIterator<OutIter, RvalueOf<InIter::reference>::type>
  OutIter move_backward(InIter first, InIter last,
                        OutIter result);

// 25.2.3, swap:
template<class T>
  requires MoveAssignable<T> && MoveConstructible<T>
  void swap(T& a, T& b);
template<ForwardIterator Iter1, ForwardIterator Iter2>
  requires HasSwap<Iter1::reference, Iter2::reference>
  Iter2 swap_ranges(Iter1 first1, Iter1 last1,
                    Iter2 first2);
template<Iterator Iter1, Iterator Iter2>
  requires HasSwap<Iter1::reference, Iter2::reference>
  void iter_swap(Iter1 a, Iter2 b);

template<InputIterator InIter, class OutIter,
         Callable<auto, 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, const InIter1::value_type&,
                        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<ForwardIterator Iter, class T>
  requires OutputIterator<Iter, Iter::reference>
        && OutputIterator<Iter, const T&>
        && HasEqualTo<Iter::value_type, T>
  void replace(Iter first, Iter last,
               const T& old_value, const T& new_value);
template<ForwardIterator Iter, Predicate<auto, Iter::value_type> Pred, class T>
  requires OutputIterator<Iter, Iter::reference>
        && OutputIterator<Iter::reference, 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::reference>
        && OutputIterator<OutIter, const T&>
        && HasEqualTo<InIter::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::value_type> Pred, class T>
  requires OutputIterator<OutIter, InIter::reference>
        && OutputIterator<OutIter, const T&>
        && CopyConstructible<Pred>
  OutIter replace_copy_if(InIter first, InIter last,
                          OutIter result,
                          Pred pred, const T& new_value);

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

template<ForwardIterator Iter>
```

```
  requires OutputIterator<Iter, RvalueOf<Iter::reference>::type>
        && EqualityComparable<Iter::value_type>
  Iter unique(Iter first, Iter last);
template<ForwardIterator Iter, EquivalenceRelation<auto, Iter::value_type> Pred>
  requires OutputIterator<Iter, RvalueOf<Iter::reference>::type>
        && CopyConstructible<Pred>
  Iter unique(Iter first, Iter last,
              Pred pred);
template<InputIterator InIter, 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>
        && !ForwardIterator<OutIter>
  OutIter unique_copy(InIter first, InIter last,
                      OutIter result);
template<ForwardIterator InIter, OutputIterator<auto, InIter::reference> OutIter>
  requires EqualityComparable<InIter::value_type>
  OutIter unique_copy(InIter first, InIter last,
                      OutIter result);
template<InputIterator InIter, ForwardIterator OutIter>
  requires OutputIterator<OutIter, InIter::reference>
        && HasEqualTo<OutIter::value_type, InIter::value_type>
        && !ForwardIterator<InIter>
  OutIter unique_copy(InIter first, InIter last,
                      OutIter result);
template<InputIterator InIter, typename OutIter,
         EquivalenceRelation<auto, 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>
        && !ForwardIterator<OutIter>
  OutIter unique_copy(InIter first, InIter last,
                      OutIter result, Pred pred);
template<ForwardIterator InIter, OutputIterator<auto, InIter::reference> OutIter,
         EquivalenceRelation<auto, InIter::value_type> Pred>
  requires CopyConstructible<Pred>
  OutIter unique_copy(InIter first, InIter last,
                      OutIter result, Pred pred);
template<InputIterator InIter, ForwardIterator OutIter,
         Predicate<auto, OutIter::value_type, InIter::value_type> Pred>
  requires OutputIterator<OutIter, InIter::reference>
        && CopyConstructible<Pred>
        && !ForwardIterator<InIter>
  OutIter unique_copy(InIter first, InIter last,
```

```
                    OutIter result, Pred pred);

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

template<ShuffleIterator Iter>
  Iter rotate(Iter first, Iter middle,
              Iter last);
template<ForwardIterator InIter, OutputIterator<auto, InIter::reference> OutIter>
  OutIter rotate_copy(InIter first, InIter middle,
                      InIter last, OutIter result);

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

concept UniformRandomNumberGenerator<typename Rand> { }
template<RandomAccessIterator Iter, UniformRandomNumberGenerator Rand>
  void random_shuffle(Iter first,
                      Iter last,
                      Rand&& g);

// 25.2.13, partitions:
template <InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires CopyConstructible<Pred>
  bool is_partitioned(Iter first, Iter last, Pred pred);

template<BidirectionalIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Pred>
  Iter partition(Iter first, Iter last, Pred pred);
template<BidirectionalIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Pred>
  Iter stable_partition(Iter first, Iter last, Pred pred);
template <InputIterator InIter, OutputIterator<auto, InIter::reference> OutIter1,
          OutputIterator<auto, InIter::reference> OutIter2, Predicate<auto, InIter::value_type> Pred>
  requires CopyConstructible<Pred>
  pair<OutIter1, OutIter2>
  partition_copy(InIter first, InIter last,
```

```
                OutIter1 out_true, OutIter2 out_false,
                Pred pred);
template<ForwardIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires CopyConstructible<Pred>
  Iter partition_point(Iter first, Iter last, Pred pred);

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

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

template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void partial_sort(Iter first,
                    Iter middle,
                    Iter last);
template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void partial_sort(Iter first,
                    Iter middle,
                    Iter last,
                    Compare comp);
template<InputIterator InIter, RandomAccessIterator RAIter>
  requires ShuffleIterator<RAIter>
        && OutputIterator<RAIter, InIter::reference>
        && HasLess<InIter::value_type, RAIter::value_type>
        && 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 ShuffleIterator<RAIter>
        && OutputIterator<RAIter, InIter::reference>
        && Predicate<Compare, InIter::value_type, RAIter::value_type>
        && StrictWeakOrder<Compare, RAIter::value_type>}
        && CopyConstructible<Compare>
  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,
         StrictWeakOrder<auto, 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,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires CopyConstructible<Compare>
  Iter is_sorted_until(Iter first, Iter last,
                       Compare comp);

template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void nth_element(Iter first, Iter nth,
                   Iter last);
template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires 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::value_type, T>
  Iter lower_bound(Iter first, Iter last,
                   const T& value);
template<ForwardIterator Iter, class T, Predicate<auto, Iter::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::value_type>
  Iter upper_bound(Iter first, Iter last,
                   const T& value);
template<ForwardIterator Iter, class T, Predicate<auto, T, Iter::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::value_type>
        && HasLess<Iter::value_type, T>
  pair<Iter, Iter>
    equal_range(Iter first,
                Iter last, const T& value);
template<ForwardIterator Iter, class T, CopyConstructible Compare>
  requires Predicate<Compare, T, Iter::value_type>
        && Predicate<Compare, Iter::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::value_type>
        && HasLess<Iter::value_type, T>
  bool binary_search(Iter first, Iter last,
                     const T& value);
template<ForwardIterator Iter, class T, CopyConstructible Compare>
  requires Predicate<Compare, T, Iter::value_type>
        && Predicate<Compare, Iter::value_type, T>
  bool binary_search(Iter first, Iter last,
                     const T& value, Compare comp);

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

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

template<BidirectionalIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void inplace_merge(Iter first,
                     Iter middle,
                     Iter last);
template<BidirectionalIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires 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 HasLess<Iter1::value_type, Iter2::value_type>
        && HasLess<Iter2::value_type, Iter1::value_type>
  bool includes(Iter1 first1, Iter1 last1,
                Iter2 first2, Iter2 last2);
template<InputIterator Iter1, InputIterator Iter2,
         typename Compare>
  requires Predicate<Compare, Iter1::value_type, Iter2::value_type>
        && Predicate<Compare, Iter2::value_type, Iter1::value_type>
  bool includes(Iter1 first1, Iter1 last1,
                Iter2 first2, Iter2 last2,
                Compare comp);

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

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

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

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

```
  requires OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && Predicate<Compare, InIter1::value_type, InIter2::value_type>
        && Predicate<Compare, InIter2::value_type, InIter1::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 ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void push_heap(Iter first, Iter last);
template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void push_heap(Iter first, Iter last,
                 Compare comp);

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

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

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

template<ForwardIterator Iter>
  requires LessThanComparable<Iter::value_type>
  Iter max_element(Iter first, Iter last);
template<ForwardIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> 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,
         StrictWeakOrder<auto, 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::value_type, Iter2::value_type>
        && HasLess<Iter2::value_type, Iter1::value_type>
  bool lexicographical_compare(Iter1 first1, Iter1 last1,
                               Iter2 first2, Iter2 last2);

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

// 25.3.9, permutations:
template<BidirectionalIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  bool next_permutation(Iter first, Iter last);
```

```
  template<BidirectionalIterator Iter,
           StrictWeakOrder<auto, Iter::value_type> Compare>
    requires ShuffleIterator<Iter>
          && CopyConstructible<Compare>
    bool next_permutation(Iter first, Iter last, Compare comp);
  template<BidirectionalIterator Iter>
    requires ShuffleIterator<Iter>
          && LessThanComparable<Iter::value_type>
    bool prev_permutation(Iter first, Iter last);
  template<BidirectionalIterator Iter,
           StrictWeakOrder<auto, Iter::value_type> Compare>
    requires ShuffleIterator<Iter>
          && 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~~

[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`.

~~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.~~

[*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*]

~~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.~~

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~~with iterators that do not necessarily define these operators. 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);
```

25.1 Non-modifying sequence operations [alg.nonmodifying]

25.1.1 All of [alg.all_of]

```
template <InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires CopyConstructible<Pred>
  bool all_of(Iter first, Iter last, Pred pred);
```

Returns: `true` if `pred(*i)` is `true` for every iterator `i` in the range `[first,last)`, and `false` otherwise.

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

25.1.2 Any of [alg.any_of]

```
template <InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires CopyConstructible<Pred>
  bool any_of(Iter first, Iter last, Pred pred);
```

Returns: `true` if there exists an iterator `i` in the range `[first,last)` such that `pred(*i)` is `true`, and `false` otherwise.

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

25.1.3 None of [alg.none_of]

```
template <InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires CopyConstructible<Pred>
  bool none_of(Iter first, Iter last, Pred pred);
```

1 *Returns:* `true` if `pred(*i)` is `false` for every iterator `i` in the range `[first,last)`, and `false` otherwise.

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

25.1.4 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.~~

25.1.5 Find [alg.find]

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

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

template<InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires CopyConstructible<Pred>
  Iter find_if_not(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`, *pred*`(*i) == false`. Returns *last* if no such iterator is found.

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

25.1.6 Find End [alg.find.end]

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

template<ForwardIterator Iter1, ForwardIterator Iter2,
         Predicate<auto, Iter1::value_type, Iter2::value_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.

25.1.7 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::value_type, Iter2::value_type>
  Iter1 find_first_of(Iter1 first1, Iter1 last1,
                      Iter2 first2, Iter2 last2);

template<InputIterator Iter1, ForwardIterator Iter2,
         Predicate<auto, Iter1::value_type, Iter2::value_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 iterator is found.

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

25.1.8 Adjacent find [alg.adjacent.find]

```
template<ForwardIterator Iter>
  requires EqualityComparable<Iter::value_type>
  Iter adjacent_find(Iter first, Iter last);

template<ForwardIterator Iter, EquivalenceRelation<auto, Iter::value_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.

25.1.9 Count [alg.count]

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

template<InputIterator Iter, Predicate<auto, Iter::value_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.

25.1.10 Mismatch [mismatch]

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

template<InputIterator Iter1, InputIterator Iter2,
         Predicate<auto, Iter1::value_type, Iter2::value_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.

25.1.11 Equal [alg.equal]

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

template<InputIterator Iter1, InputIterator Iter2,
         Predicate<auto, Iter1::value_type, Iter2::value_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.

25.1.12 Search [alg.search]

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

template<ForwardIterator Iter1, ForwardIterator Iter2,
         Predicate<auto, Iter1::value_type, Iter2::value_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::value_type, T>
  Iter search_n(Iter first, Iter last, Iter::difference_type count,
                const T& value);

template<ForwardIterator Iter, class T,
```

```
        Predicate<auto, Iter::value_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 positive, 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<InputIterator InIter, OutputIterator<auto, InIter::reference> OutIter>
  OutIter copy_n(InIter first, InIter::difference_type n,
                 OutIter result);
```

As with `fill_n`, we have eliminated the `Size` parameter and instead have used the `difference_type` of the input iterator, which is a better choice for measuring distances within the input iterator sequence.

5 *Effects:* For each non-negative integer `i < n`, performs `*(result + i) = *(first + i)`.

6 *Returns:* `result + n`.

7 *Complexity:* Exactly `n` assignments.

```
template<InputIterator InIter, OutputIterator<auto, InIter::reference> OutIter,
         Predicate<auto, InIter::value_type> Pred>
  requires CopyConstructible<Pred>
  OutIter copy_if(InIter first, InIter last,
                  OutIter result, Pred pred);
```

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

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

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

11 *Remarks:* Stable.

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

12 *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)`, performs `*(result - n) = *(last - n)`.

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

14 *Returns:* `result - (last - first)`.

15 *Complexity:* Exactly `last - first` assignments.

25.2.2 Move [alg.move]

```
template<InputIterator InIter, typename OutIter>
  requires OutputIterator<OutIter, RvalueOf<InIter::reference>::type>
  OutIter move(InIter first, InIter last,
               OutIter 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 (*(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 OutputIterator<OutIter, RvalueOf<InIter::reference>::type>
  OutIter move_backward(InIter first, InIter last,
```

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

```
                    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(*(*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>
  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<ForwardIterator Iter1, ForwardIterator Iter2>
  requires HasSwap<Iter1::reference, Iter2::reference>
  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)).

4 *Requires:* The two ranges [*first1*,*last1*) and [*first2*,*first2* + (*last1* - *first1*)) shall not overlap. ~~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<Iterator Iter1, Iterator Iter2>
  requires HasSwap<Iter1::reference, Iter2::reference>
  void iter_swap(Iter1 a, Iter2 b);
```

We have loosened the requirements on iter_swap from ForwardIterator (required in C++03, which needed true references and copy-constructible/copy-assignable value types) to just Iterator, which represents the minimum requirement for C++0x iterators.

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).~~

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

25.2.4 Transform [alg.transform]

```
template<InputIterator InIter, class OutIter,
         Callable<auto, 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, const InIter1::value_type&,
                        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 corresponding 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<ForwardIterator Iter, class T>
  requires OutputIterator<Iter, Iter::reference>
        && OutputIterator<Iter, const T&>
        && HasEqualTo<Iter::value_type, T>
  void replace(Iter first, Iter last,
               const T& old_value, const T& new_value);

template<ForwardIterator Iter, Predicate<auto, Iter::value_type> Pred, class T>
```

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

```
  requires OutputIterator<Iter, Iter::reference>
        && OutputIterator<Iter, const 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::reference>
        && OutputIterator<OutIter, const T&>
        && HasEqualTo<InIter::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::value_type> Pred, class T>
  requires OutputIterator<OutIter, InIter::reference>
        && OutputIterator<OutIter, const T&>
        && CopyConstructible<Pred>
  OutIter replace_copy_if(InIter first, InIter last,
                          OutIter result,
                          Pred pred, const T& new_value);
```

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<ForwardIterator Iter, class T>
  requires OutputIterator<Iter, const 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<ForwardIterator Iter, Callable Generator>
  requires OutputIterator<Iter, 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);
```

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 OutputIterator<Iter, RvalueOf<Iter::reference>::type>
        && HasEqualTo<Iter::value_type, T>
  Iter remove(Iter first, Iter last,
              const T& value);

template<ForwardIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires OutputIterator<Iter, RvalueOf<Iter::reference>::type>
```

[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++.

[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++.

```
      && 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::reference> OutIter, class T>
  requires HasEqualTo<InIter::value_type, T>
  OutIter remove_copy(InIter first, InIter last,
                      OutIter result, const T& value);

template<InputIterator InIter, OutputIterator<auto, InIter::reference> OutIter,
         Predicate<auto, InIter::value_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.

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 OutputIterator<Iter, Iter::reference>
        && EqualityComparable<Iter::value_type>
  Iter unique(Iter first, Iter last);

template<ForwardIterator Iter, EquivalenceRelation<auto, Iter::value_type> Pred>
  requires OutputIterator<Iter, RvalueOf<Iter::reference>::type>
        && 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, 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>
        && !ForwardIterator<OutIter>
  OutIter unique_copy(InIter first, InIter last,
                      OutIter result);

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

template<InputIterator InIter, ForwardIterator OutIter>
  requires OutputIterator<OutIter, InIter::reference>
        && HasEqualTo<OutIter::value_type, InIter::value_type>
        && !ForwardIterator<InIter>
  OutIter unique_copy(InIter first, InIter last,
                      OutIter result);

template<InputIterator InIter, typename OutIter,
         EquivalenceRelation<auto, 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>
        && !ForwardIterator<OutIter>
  OutIter unique_copy(InIter first, InIter last,
                      OutIter result, Pred pred);

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

```
template<InputIterator InIter, ForwardIterator OutIter,
         Predicate<auto, OutIter::value_type, InIter::value_type> Pred>
  requires OutputIterator<OutIter, 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 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::reference> 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<ShuffleIterator Iter>
  Iter 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 *Returns:* `first + (last - middle)`.

3 *Remarks:* This is a left rotate.

4 *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).~~

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

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

6 *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))`.

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

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

9 *Complexity:* Exactly `last - first` assignments.

25.2.12 Random shuffle [alg.random.shuffle]

```
template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
  void random_shuffle(Iter first,
                      Iter last);

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

concept UniformRandomNumberGenerator<typename Rand> { }

template<RandomAccessIterator Iter, UniformRandomNumberGenerator Rand>
  void random_shuffle(Iter first,
                      Iter last,
```

```
                    Rand&& g);
```

In the third overload of `random_shuffle`, we use the placeholder concept `UniformRandomNumberGenerator`. When concepts are provided for this concept, the placeholder above will be removed and the third `random_shuffle` will refer to that `UniformRandomNumberGenerator`. Additionally, we have opted to make the third argument an rvalue reference rather than an lvalue reference, to synchronize it with the second `random_shuffle`.

1 *Effects:* Permutes the elements in the range `[`*`first`*`,`*`last`*`)` such that each possible permutation of those elements has equal probability of appearance.

2 ~~Requires: The type of *first shall satisfy the Swappable requirements (Table 37). The random number generating function object rand shall have a return type that is convertible to iterator_traits<RandomAccessIterator>::difference_ and the~~The call `rand(n)` shall return a randomly chosen value in the interval `[0,n)`, for `n > 0` ~~of type iterator_traits<Rando~~ ~~The function object g shall meet the requirements of uniform random number generator (??).~~

Paragraphs 3 and 4 are unchanged.

25.2.13 Partitions [alg.partitions]

```
template <InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires CopyConstructible<Pred>
  bool is_partitioned(Iter first, Iter last, Pred pred);
```

1 ~~*Requires:* `InputIterator`'s value type shall be convertible to `Predicate`'s argument type.~~

2 *Returns:* `true` if `[first,last)` is partitioned by `pred`, i.e. if all elements that satisfy `pred` appear before those that do not.

3 *Complexity:* Linear. At most `last - first` applications of `pred`.

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

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

5 *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`.

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

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

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

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

9 *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.

10 ~~Requires: The type of *first shall satisfy the Swappable requirements (20.1.4), the MoveConstructible requirements (Table ??), and the the MoveAssignable requirements (Table ??).~~

11 *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.

```
template <InputIterator InIter, OutputIterator<auto, InIter::reference> OutIter1,
          OutputIterator<auto, InIter::reference> OutIter2, Predicate<auto, InIter::value_type> Pred>
  requires CopyConstructible<Pred>
  pair<OutIter1, OutIter2>
  partition_copy(InIter first, InIter last,
                 OutIter1 out_true, OutIter2 out_false,
                 Pred pred);
```

12 ~~*Requires:* InputIterator's value type shall be Assignable, and shall be writable to the out_true and out_false OutputIterators, and shall be convertible to PredicateâĂŹs argument type. The input range shall not overlap with either of the output ranges.~~

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

14 *Returns:* A pair p such that p.first is the end of the output range beginning at out_true and p.second is the end of the output range beginning at out_false.

15 *Complexity:* Exactly last - first applications of pred.

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

16 ~~*Requires:* ForwardIterator's value type shall be convertible to Predicate's argument type. [first,last) shall be partitioned by pred, i.e. all elements that satisfy pred shall appear before those that do not.~~

17 *Returns:* An iterator mid such that all_of(first, mid, pred) and none_of(mid, last, pred) are both true.

18 *Complexity:* $\mathcal{O}(log(last - first))$ applications of pred.

25.3 Sorting and related operations [alg.sorting]

25.3.1 Sorting [alg.sort]

25.3.1.1 sort [sort]

```
template<RandomAccessIterator Iter>
```

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

template<RandomAccessIterator Iter,
          StrictWeakOrder<auto, Iter::value_type> Compare>
  requires 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]

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

template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires 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 ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void partial_sort(Iter first,
```

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

```
                    Iter middle,
                    Iter last);
template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires 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 ShuffleIterator<RAIter>
        && OutputIterator<RAIter, InIter::reference>
        && HasLess<InIter::value_type, RAIter::value_type>
        && 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 ShuffleIterator<RAIter>
        && OutputIterator<RAIter, InIter::reference>
        && Predicate<Compare, InIter::value_type, RAIter::value_type>
        && StrictWeakOrder<Compare, RAIter::value_type>}
        && CopyConstructible<Compare>
  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,
         StrictWeakOrder<auto, 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,
         StrictWeakOrder<auto, 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 ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void nth_element(Iter first, Iter nth,
                   Iter last);

template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires 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::value_type, T>
  Iter lower_bound(Iter first, Iter last,
                   const T& value);

template<ForwardIterator Iter, class T, Predicate<auto, Iter::value_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::value_type>
  Iter upper_bound(Iter first, Iter last,
                   const T& value);

template<ForwardIterator Iter, class T, Predicate<auto, T, Iter::value_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::value_type>
        && HasLess<Iter::value_type, T>
  pair<Iter, Iter>
    equal_range(Iter first,
                Iter last, const T& value);

template<ForwardIterator Iter, class T, CopyConstructible Compare>
  requires Predicate<Compare, T, Iter::value_type>
        && Predicate<Compare, Iter::value_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::value_type>
        && HasLess<Iter::value_type, T>
  bool binary_search(Iter first, Iter last,
                     const T& value);

template<ForwardIterator Iter, class T, CopyConstructible Compare>
  requires Predicate<Compare, T, Iter::value_type>
        && Predicate<Compare, Iter::value_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 conditions: `!(*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,
         typename OutIter>
  requires
           OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && HasLess<InIter2::value_type, InIter1::value_type>
  OutIter merge(InIter1 first1, InIter1 last1,
                InIter2 first2, InIter2 last2,
                OutIter result);

template<InputIterator InIter1, InputIterator InIter2,
         typename OutIter,
         Predicate<auto, InIter2::value_type, InIter1::value_type> Compare>
  requires
           OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && CopyConstructible<Compare>
  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 ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void inplace_merge(Iter first,
                     Iter middle,
                     Iter last);

template<BidirectionalIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
```

```
requires 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` (**??**) 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 HasLess<Iter1::value_type, Iter2::value_type>
        && HasLess<Iter2::value_type, Iter1::value_type>
  bool includes(Iter1 first1, Iter1 last1,
                Iter2 first2, Iter2 last2);

template<InputIterator Iter1, InputIterator Iter2,
         typename Compare>
  requires Predicate<Compare, Iter1::value_type, Iter2::value_type>
        && Predicate<Compare, Iter2::value_type, Iter1::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,
         typename OutIter>
  requires OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && HasLess<InIter2::value_type, InIter1::value_type>
        && HasLess<InIter1::value_type, InIter2::value_type>
  OutIter set_union(InIter1 first1, InIter1 last1,
                    InIter2 first2, InIter2 last2,
                    OutIter result);

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

template<InputIterator InIter1, InputIterator InIter2,
         typename OutIter,
         CopyConstructible Compare>
```

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

template<InputIterator InIter1, InputIterator InIter2,
         typename OutIter,
         CopyConstructible Compare>
  requires OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && Predicate<Compare, InIter1::value_type, InIter2::value_type>
        && Predicate<Compare, InIter2::value_type, InIter1::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.

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,
         typename OutIter>
  requires OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && HasLess<InIter2::value_type, InIter1::value_type>
        && HasLess<InIter1::value_type, InIter2::value_type>
  OutIter set_symmetric_difference(InIter1 first1, InIter1 last1,
                                   InIter2 first2, InIter2 last2,
                                   OutIter result);

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

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.

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

Returns: The end of the constructed range.

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

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]

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.

These properties make heaps useful as priority queues.

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

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

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

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

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

25.3.6.2 `pop_heap` [pop.heap]

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

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

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

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 ??), and the the `MoveAssignable` requirements (Table ??).~~

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

25.3.6.3 `make_heap` [make.heap]

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

template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires 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 ??) and the the MoveAssignable requirements (Table ??).~~

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

25.3.6.4 sort_heap [sort.heap]

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

template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires 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 ??), and the the MoveAssignable requirements (Table ??).~~

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,
         StrictWeakOrder<auto, 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,
         StrictWeakOrder<auto, 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, 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);~~
```

We have removed this version of the variadic `min` function because its requirements cannot be specified in any natural way. The fundamental problem is that the parameter pack `args` is used to contain N-1 arguments followed by the comparison operator, unless `args` is empty, in which case `b` is the comparison operator and is unused. Such a specification requires significant metaprogramming that would need to be exposed in the specification itself. The standard library concepts drafting group strongly believes that, in light of concepts, the complexity of this routine far outweighs its benefits.

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

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

~~*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, StrictWeakOrder<auto, T> Compare>
  requires !SameType<T, Compare> && CopyConstructible<Compare>
  const T& max(const T& a, const T& b, Compare comp);
```

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

Returns: The larger value.

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);
```

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

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

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 have removed this variant of the `max` function.

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

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

~~*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, StrictWeakOrder<auto, T> Compare>
  requires !SameType<T, Compare> && CopyConstructible<Compare>
  pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
```

~~*Requires:* Type `T` shall be `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 have removed this variant of `minmax`.

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::value_type>
  Iter min_element(Iter first, Iter last);

template<ForwardIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> 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::value_type>
  Iter max_element(Iter first, Iter last);

template<ForwardIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> 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,
         StrictWeakOrder<auto, 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::value_type, Iter2::value_type>
        && HasLess<Iter2::value_type, Iter1::value_type>
  bool lexicographical_compare(Iter1 first1, Iter1 last1,
                               Iter2 first2, Iter2 last2);

template<InputIterator Iter1, InputIterator Iter2, CopyConstructible Compare>
  requires Predicate<Compare, Iter1::value_type, Iter2::value_type>
        && Predicate<Compare, Iter2::value_type, Iter1::value_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 ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  bool next_permutation(Iter first, Iter last);

template<BidirectionalIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
        && 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 ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  bool prev_permutation(Iter first, Iter last);

template<BidirectionalIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
```

```
      && 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.

Acknowledgments

Chris Jefferson provided fixes for the `partial_sort_copy` algorithm. Sean Parent described the view of permutation algorithms as cycles of data movement, which gave us an implementation-agnostic way to determine which mutating algorithms required `ShuffleIterator` and which required `HasSwap` on the iterator's reference types. Daniel Krügler provided conceptualized versions of the algorithms in N2666.

Bibliography

[1] Douglas Gregor, Bjarne Stroustrup, James Widman, and Jeremy Siek. Proposed wording for concepts (revision 8). Technical Report N2741=08-0251, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, August 2008.