Concepts for the C++0x Standard Library: Algorithms

Douglas Gregor, Jeremiah Willcock, and Andrew Lumsdaine
Open Systems Laboratory
Indiana University
Bloomington, IN 47405
{dgregor, jewillco, lums}@cs.indiana.edu

Document number: N2040=06-0110
Date: 2006-06-23
Project: Programming Language C++, Library Working Group
Reply-to: Douglas Gregor <dgregor@cs.indiana.edu>

Introduction

This document proposes changes to Chapter 25 of the C++ Standard Library in order to make full use of concepts [1]. Unless otherwise specified, all changes in this document have been verified to work with ConceptGCC and its modified Standard Library implementation. 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 working draft of the C++ standard (N2009). 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.

Chapter 25 Algorithms library [lib.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	`<algorithm>`
25.3 Sorting and related operations	
?? C library algorithms	`<cstdlib>`

Header `<algorithm>` synopsis

Note: Provide an updated synopsis!

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 BinaryPredicate parameter is used whenever an algorithm expects a function object that when applied to the result of dereferencing two corresponding iterators or to dereferencing an iterator and type T when T is part of the signature returns a value testable as true. In other words, if an algorithm takes BinaryPredicate binary_pred as its argument and first1 and first2 as its iterator arguments, it should work correctly in the construct if (binary_pred(*first1, *first2)){...}. BinaryPredicate always takes the first iterator type as its first argument, that is, in those cases when T value is part of the signature, it should work correctly in the context of if (binary_pred(*first1, value)){...}. binary_pred shall not apply any non-constant function through the dereferenced iterators.~~

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

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

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

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

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

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

25.1 Non-modifying sequence operations [lib.alg.nonmodifying]

25.1.1 For each [lib.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, Callable1<Iter::reference> Function>
  Function for_each(Iter first, Iter last, Function f);
```

[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`.

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

2 *Returns:* `f`.

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

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

25.1.2 Find [lib.alg.find]

```
template<InputIterator Iter, class T>
  where EqualityComparable<Iter::reference, T>
  Iter find(Iter first, Iter last, const T& value);

template<InputIterator Iter, Predicate<Iter::reference> Pred>
  Iter find_if(Iter first, Iter last, Pred pred);
```

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

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

25.1.3 Find End [lib.alg.find.end]

```
template<ForwardIterator Iter1, ForwardIterator Iter2>
  where EqualityComparable<Iter1::reference, Iter2::reference>
  Iter1 find_end(Iter1 first1, Iter1 last1,
                 Iter2 first2, Iter2 last2);

template<ForwardIterator Iter1, ForwardIterator Iter2,
         BinaryPredicate<Iter1::reference, Iter2::reference> Pred>
  Iter1 find_end(Iter1 first1, Iter1 last1,
                 Iter2 first2, Iter2 last2,
                 Pred pred);
```

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

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

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

25.1.4 Find First [lib.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>
  where EqualityComparable<Iter1::reference, Iter2::reference>
  Iter1 find_first_of(Iter1 first1, Iter1 last1,
                      Iter2 first2, Iter2 last2);

template<InputIterator Iter1, ForwardIterator Iter2,
         BinaryPredicate<Iter1::reference, Iter2::reference> 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.

25.1.5 Adjacent find [lib.alg.adjacent.find]

```
template<ForwardIterator Iter>
  where EqualityComparable<Iter::reference>
  Iter adjacent_find(Iter first, Iter last);

template<ForwardIterator Iter, BinaryPredicate<Iter1::reference, Iter2::reference> Pred>
  Iter adjacent_find(Iter first, Iter last, Pred pred);
```

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

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

25.1.6 Count [lib.alg.count]

```
template<InputIterator Iter, class T>
  where EqualityComparable<Iter::reference, T>
  Iter::difference_type count(Iter first, Iter last, const T& value);

template<InputIterator Iter, Predicate<Iter::reference> Pred>
  Iter::difference_type count_if(Iter first, Iter last, Pred pred);
```

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

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

25.1.7 Mismatch [lib.mismatch]

```
template<InputIterator Iter1, InputIterator Iter2>
  where EqualityComparable<Iter1::reference, Iter2::reference>
  pair<Iter1, Iter2> mismatch(Iter1 first1, Iter1 last1,
                              Iter first2);

template<InputIterator Iter1, InputIterator Iter2,
         BinaryPredicate<Iter1::reference, Iter2::reference> Pred>
  pair<Iter1, Iter2> mismatch(Iter1 first1, Iter2 last1,
                              Iter2 first2, Pred pred);
```

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

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

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

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

25.1.8 Equal [lib.alg.equal]

```
template<InputIterator Iter1, InputIterator Iter2>
  where EqualityComparable<Iter1::reference, Iter2::reference>
  bool equal(Iter1 first1, Iter1 last1,
             Iter2 first2);

template<InputIterator Iter1, InputIterator Iter2,
         BinaryPredicate<Iter1::reference, Iter2::reference> Pred>
  bool equal(Iter1 first1, Iter1 last1,
             Iter2 first2, Pred pred);
```

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

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

25.1.9 Search [lib.alg.search]

```
template<ForwardIterator Iter1, ForwardIterator Iter2>
  where EqualityComparable<Iter1::reference, Iter2::reference>
  Iter1 search(Iter1 first1, Iter1 last1,
               Iter2 first2, Iter2 last2);

template<ForwardIterator Iter1, ForwardIterator Iter2,
         BinaryPredicate<Iter1::reference, Iter2::reference> 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>
  where EqualityComparable<Iter::reference, T>
  Iter search_n(Iter first, Iter last, Iter::difference_type count,
                const T& value);

template<ForwardIterator Iter, class T,
         BinaryPredicate<Iter::reference, T> 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. Note: This change has not yet been reflected in libstdc++.

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 [lib.alg.modifying.operations]

25.2.1 Copy [lib.alg.copy]

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

Note: Due to some optimizations in libstdc++ that we have not yet modeled with concepts, ConceptGCC does not yet use this definition of `copy`. However, it does type-check with ConceptGCC.

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

`*(result + n) = *(first + n)`.

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

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

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

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

Note: Due to some optimizations in libstdc++ that we have not yet modeled with concepts, ConceptGCC does not yet use this definition of `copy_backward`. However, it does type-check with ConceptGCC.

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

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

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

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

25.2.2 Swap [lib.alg.swap]

```
template<class T>
  where Assignable<T> && CopyConstructible<T>
  void swap(T& a, T& b);
```

1 ~~Requires: Type T is CopyConstructible (20.1.3) and Assignable (23.1).~~

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

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

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

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.

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

```
template<MutableForwardIterator Iter1, MutableForwardIterator Iter2>
  where SameType<Iter1::reference, Iter2::reference> && Swappable<Iter1::value_type>
  void iter_swap(Iter1 a, Iter2 b);
```

Note: ConceptGCC is a little bit fuzzy on this, because it doesn't actually type-check (although it should) and doesn't contain the Swappable requirement.

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

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

25.2.3 Transform [lib.alg.transform]

```
template<InputIterator InIter, class OutIter,
         Callable1<InIter::reference> Op>
  where OutputIterator<OutIter, Op::result_type>
  OutIter transform(InIter first, InIter last,
                    OutIter result, Op op);

template<InputIterator InIter1, InputIterator InIter2,
         class OutIter, Callable2<InIter1::reference, InIter2::reference> BinaryOp>
  where OutputIterator<OutIter, BinaryOp::result_type>
  OutIter transform(InIter1 first1, InIter1 last1,
                    InIter2 first2, OutIter result,
                    BinaryOp binary_op);
```

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)]`.[3)]

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.

25.2.4 Replace [lib.alg.replace]

```
template<MutableForwardIterator Iter, class T>
  where EqualityComparable<Iter::reference, T> && Assignable<Iter::reference, T>
  void replace(Iter first, Iter last,
               const T& old_value, const T& new_value);
```

3) The use of fully closed ranges is intentional.

```
template<MutableForwardIterator Iter, Predicate<Iter::reference> Pred, class T>
  where Assignable<Iter::reference, T>
  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, OutputIterator<InIter::reference> OutIter, class T>
  where Assignable<OutIter::reference, T> && EqualityComparable<InIter::reference, T>
  OutIter replace_copy(InIter first, InIter last,
                       OutIter result,
                       const T& old_value, const T& new_value);

template<InputIterator InIter, OutputIterator<InIter::reference> OutIter,
         Predicate<InIter::reference> Pred, class T>
  where Assignable<OutIter::reference, T>
  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.

25.2.5 Fill [lib.alg.fill]

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

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

Note: Due to some optimizations in libstdc++ that we have not yet modeled with concepts, ConceptGCC does not yet use these definitions of `fill` or `fill_n`.

1 ~~Requires: The expression value shall be writable to the output iterator. The type Size is convertible to integral type (4.7, 12.3).~~[4]

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.

25.2.6 Generate [lib.alg.generate]

```
template<MutableForwardIterator Iter, Callable0 Generator>
  where Assignable<Iter::reference, Generator::result_type>
  void generate(Iter first, Iter last,
                Generator gen);

template<class Iter, Integral Size, Callable0 Generator>
  where OutputIterator<Iter, Generator::result_type>
  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).~~[5]

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

25.2.7 Remove [lib.alg.remove]

```
template<MutableForwardIterator Iter, class T>
  where EqualityComparable<Iter::reference, T>
  Iter remove(Iter first, Iter last,
              const T& value);

template<MutableForwardIterator Iter, Predicate<Iter::reference> Pred>
  Iter remove_if(Iter first, Iter last,
                 Pred pred);
```

Note: ConceptGCC is not fully type-checking `remove` and `remove_if` at this time.

1 ~~Requires: The type of *first shall satisfy the Assignable requirements (23.1).~~

4) 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 `Integral`. This could break existing code, although it appears that such existing code would fail to compile with at least libstdc++.

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 `Integral`. This could break existing code, although it appears that such existing code would fail to compile with at least libstdc++.

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

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

25.2.8 Unique [lib.alg.unique]

```
template<MutableForwardIterator Iter>
  where EqualityComparable<Iter::reference>
  Iter unique(Iter first, Iter last);

template<MutableForwardIterator Iter, BinaryPredicate<Iter::reference, Iter::reference> Pred>
  Iter unique(Iter first, Iter last,
              Pred pred);
```

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

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

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

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

```
template<InputIterator InIter, OutputIterator<InIter::value_type> OutIter>
  where EqualityComparable<InIter::value_type> && Assignable<InIter::value_type> &&
        CopyConstructible<InIter::value_type> && !ForwardIterator<InIter> &&
        !MutableForwardIterator<OutIter>
  OutIter
    unique_copy(InIter first, InIter last,
                OutIter result);

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

template<InputIterator InIter, MutableForwardIterator OutIter>
  where EqualityComparable<OutIter::reference, InIter::value_type> &&
        Assignable<OutIter::reference, InIter::reference> &&
        !ForwardIterator<InIter>
  OutIter unique_copy(InIter first, InIter last,
                      OutIter result);

template<InputIterator InIter, OutputIterator<InIter::value_type> OutIter,
         BinaryPredicate<InIter::value_type, InIter::value_type> Pred>
  where Assignable<InIter::value_type> && CopyConstructible<InIter::value_type> &&
        !ForwardIterator<InIter> && !MutableForwardIterator<OutIter>
  OutIter unique_copy(InIter first, InIter last,
                      OutIter result, Pred pred);

template<ForwardIterator InIter, OutputIterator<InIter::value_type> OutIter,
         BinaryPredicate<InIter::reference, InIter::reference> Pred>
  OutIter unique_copy(InIter first, InIter last,
                      OutIter result);

template<InputIterator InIter, MutableForwardIterator OutIter,
         BinaryPredicate<OutIter::reference, InIter::reference> Pred>
  where Assignable<OutIter::reference, InIter::reference> &&
        !ForwardIterator<InIter>
  OutIter unique_copy(InIter first, InIter last,
                      OutIter result, Pred pred);
```

We assume (and require!) the proposed resolution to DR 538, which adds the `Assignable` requirement.

Note that we have split the two signatures of `unique_copy` into six signatures, to cover the actual variants required in the implementation.

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.

25.2.9 Reverse [lib.alg.reverse]

```
template<MutableBidirectionalIterator Iter>
  where Swappable<Iter::value_type>
  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<InIter::value_type> 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.

25.2.10 Rotate [lib.alg.rotate]

```
template<MutableForwardIterator Iter>
  where Swappable<Iter>
  void rotate(Iter first, Iter middle,
              Iter last);
```

1 *Effects:* For each non-negative integer `i < (last - first)`, places the element from the position `first + i` into position `first + (i + (last - middle)) % (last - first)`.

2 *Remarks:* This is a left rotate.

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

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

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

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

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

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

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

25.2.11 Random shuffle [lib.alg.random.shuffle]

```
template<MutableRandomAccessIterator Iter>
  where Swappable<Iter::value_type>
  void random_shuffle(Iter first,
                      Iter last);

template<MutableRandomAccessIterator Iter, Callable1<Iter::difference_type> Rand>
  where Swappable<Iter::value_type> && Convertible<Rand::result_type, Iter::difference_type>
  void random_shuffle(Iter first,
                      Iter last,
                      Rand& rand);
```

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

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

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

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

25.2.12 Partitions [lib.alg.partitions]

```
template<MutableBidirectionalIterator Iter, Predicate<Iter::reference> Pred>
  where Swappable<Iter::value_type>
  Iter partition(Iter first,
                 Iter last, Pred pred);
```

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

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

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

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

```
template<MutableBidirectionalIterator Iter, Predicate<Iter::reference> Pred>
  where Swappable<Iter::value_type>
  Iter stable_partition(Iter first,
                        Iter last, Pred pred);
```

ConceptGCC cannot currently type-check the libstdc++ definition of stable_partition, so these concept requirements have not been verified.

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

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

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

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

25.3 Sorting and related operations [lib.alg.sorting]

1 All the operations in 25.3 have two versions: one that takes a function object of type `Compare` and one that uses an `operator<`.

2 `Compare` is used as a function object which returns `true` if the first argument is less than the second, and `false` otherwise. `Compare comp` is used throughout for algorithms assuming an ordering relation. It is assumed that `comp` will not apply any non-constant function through the dereferenced iterator.

3 For all algorithms that take `Compare`, there is a version that uses `operator<` instead. That is, `comp(*i, *j) != false` defaults to `*i < *j != false`. For algorithms other than those described in 25.3.3 to work correctly, `comp` has to induce a strict weak ordering on the values.

4 The term *strict* refers to the requirement of an irreflexive relation (`!comp(x, x)` for all `x`), and the term *weak* to requirements that are not as strong as those for a total ordering, but stronger than those for a partial ordering. If we define `equiv(a, b)` as `!comp(a, b) && !comp(b, a)`, then the requirements are that `comp` and `equiv` both be transitive relations:

— `comp(a, b) && comp(b, c)` implies `comp(a, c)`

— `equiv(a, b) && equiv(b, c)` implies `equiv(a, c)` [*Note:* Under these conditions, it can be shown that

  — `equiv` is an equivalence relation

  — `comp` induces a well-defined relation on the equivalence classes determined by `equiv`

  — The induced relation is a strict total ordering. —*end note*]

5 A sequence is *sorted with respect to a comparator* `comp` if for any iterator `i` pointing to the sequence and any non-negative integer `n` such that `i + n` is a valid iterator pointing to an element of the sequence, `comp(*(i + n), *i) == false`.

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

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

25.3.1 Sorting [lib.alg.sort]

25.3.1.1 `sort` [lib.sort]

```
template<MutableRandomAccessIterator Iter>
  where LessThanComparable<Iter::value_type> && Assignable<Iter::reference> &&
        Swappable<Iter::value_type> && CopyConstructible<Iter::value_type>
  void sort(Iter first, Iter last);

template<MutableRandomAccessIterator Iter, BinaryPredicate<Iter::value_type, Iter::value_type> Compare>
  where Assignable<Iter::reference> && Swappable<Iter::value_type> &&
        CopyConstructible<Iter::value_type>
  void sort(Iter first, Iter last,
            Compare comp);
```

It is possible that these concept requirements could be simplified. These requirements were pushed up from the libstdc++ implementation, which may overconstrain the problem.

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

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

3 *Complexity:* Approximately $N\log(N)$ (where N `== last - first`) comparisons on the average.[6)]

25.3.1.2 `stable_sort` [lib.stable.sort]

```
template<MutableRandomAccessIterator Iter>
  where LessThanComparable<Iter::value_type> && Swappable<Iter::value_type> &&
        Assignable<Iter::value_type> && CopyConstructible<Iter::value_type>
  void stable_sort(Iter first, Iter last);

template<MutableRandomAccessIterator Iter, BinaryPredicate<Iter::value_type, Iter::value_type> Compare>
  where Swappable<Iter::value_type> && Assignable<Iter::value_type> &&
        CopyConstructible<Iter::value_type>
  void stable_sort(Iter first, Iter last,
                   Compare comp);
```

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

ConceptGCC can not currently type-check the libstdc++ definition of stable_sort, so these concept requirements are speculative.

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

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

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` [lib.partial.sort]

```
template<MutableRandomAccessIterator Iter>
  where CopyConstructible<Iter::value_type> && Swappable<Iter::value_type> &&
        LessThanComparable<Iter::value_type>
  void partial_sort(Iter first,
                    Iter middle,
                    Iter last);

template<MutableRandomAccessIterator Iter, BinaryPredicate<Iter::value_type, Iter::value_type> Compare>
  where CopyConstructible<Iter::value_type> && Swappable<Iter::value_type>
  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).~~

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

25.3.1.4 `partial_sort_copy` [lib.partial.sort.copy]

Note: This does not currently type-check with ConceptGCC.

```
template<class InputIterator, class RandomAccessIterator>
  RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last);

template<class InputIterator, class RandomAccessIterator,
         class Compare>
  RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last,
                      Compare comp);
```

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 (**??**).

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

25.3.2 Nth element [lib.alg.nth.element]

```
template<MutableRandomAccessIterator Iter>
  where Swappable<Iter::value_type> && Assignable<Iter::reference> &&
        CopyConstructible<Iter::value_type> && LessThanComparable<Iter::value_type>
  void nth_element(Iter first, Iter nth,
                   Iter last);

template<MutableRandomAccessIterator Iter, BinaryPredicate<Iter::value_type, Iter::value_type> Compare>
  where Swappable<Iter::value_type> && Assignable<Iter::reference> &&
        CopyConstructible<Iter::value_type>
  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).~~

3 *Complexity:* Linear on average.

25.3.3 Binary search [lib.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` [lib.lower.bound]

```
template<ForwardIterator Iter, class T>
  where LessThanComparable<Iter::value_type, T>
  Iter lower_bound(Iter first, Iter last,
                   const T& value);
```

```
template<ForwardIterator Iter, class T, BinaryPredicate<Iter::reference, T> 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` [lib.upper.bound]

```
template<ForwardIterator Iter, class T>
  where LessThanComparable<T, Iter::reference>
  Iter upper_bound(Iter first, Iter last,
                   const T& value);

template<ForwardIterator Iter, class T, BinaryPredicate<T, Iter::reference> 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` [lib.equal.range]

```
template<ForwardIterator Iter, class T>
  where LessThanComparable<T, Iter::reference> &&
        LessThanComparable<Iter::reference, T>
  pair<Iter, Iter>
    equal_range(Iter first,
                Iter last, const T& value);

template<ForwardIterator Iter, class T, class Compare>
  where BinaryPredicate<Compare, T, Iter::reference> &&
        BinaryPredicate<Compare, Iter::reference, 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` [lib.binary.search]

```
template<ForwardIterator Iter, class T>
  where LessThanComparable<T, Iter::reference> &&
        LessThanComparable<Iter::reference, T>
  bool binary_search(ForwardIterator first, ForwardIterator last,
                     const T& value);

template<ForwardIterator Iter, class T, class Compare>
  where BinaryPredicate<Compare, T, Iter::reference> &&
        BinaryPredicate<Compare, Iter::reference, 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 [lib.alg.merge]

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

```
template<InputIterator InIter1, InputIterator InIter2,
         OutputIterator<InIter1::value_type> OutIter,
         BinaryPredicate<InIter1::value_type, InIter2::value_type> Compare>
  where SameType<InIter1::value_type, InIter2::value_type>
  OutIter
    merge(InIter1 first1, InIter1 last1,
          InIter2 first2, InIter2 last2,
          OutIter result, Compare comp);
```

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

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

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

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

5 *Remarks:* Stable.

```
template<MutableBidirectionalIterator Iter>
  where Swappable<Iter::value_type> && CopyConstructible<Iter::value_type> &&
        Assignable<Iter::value_type> && LessThanComparable<Iter::value_type>
  void inplace_merge(Iter first,
                     Iter middle,
                     Iter last);

template<MutableBidirectionalIterator Iter, BinaryPredicate<Iter::value_type, Iter::value_type> Compare>
  where Swappable<Iter::value_type> && CopyConstructible<Iter::value_type> &&
        Assignable<Iter::value_type>
  void inplace_merge(Iter first,
                     Iter middle,
                     Iter last, Compare comp);
```

At present, ConceptGCC can not type-check this algorithm, so the concept constraints are speculative.

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).~~

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 [lib.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` [lib.includes]

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

template<InputIterator Iter1, InputIterator Iter2, BinaryPredicate<Iter1::value_type, Iter2::value_type> Compare>
  where SameType<Iter1::value_type, Iter2::value_type>
  bool includes(Iter1 first1, Iter1 last1,
                Iter2 first2, Iter2 last2,
                Compare comp);
```

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

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

25.3.5.2 `set_union` [lib.set.union]

```
template<InputIterator InIter1, InputIterator InIter2,
         OutputIterator<InIter1::value_type> OutIter>
  where SameType<InIter1::value_type, InIter2::value_type> &&
        LessThanComparable<InIter1::value_type>
  OutIter
    set_union(InIter1 first1, InIter1 last1,
              InIter2 first2, InIter2 last2,
              OutIter result);

template<InputIterator InIter1, InputIterator InIter2,
         OutputIterator<InIter1::value_type> OutIter,
         BinaryPredicate<InIter1::value_type, InIter2::value_type> Compare>
  where SameType<InIter1::value_type, InIter2::value_type>
  OutIter
    set_union(InIter1 first1, InIter1 last1,
              InIter2 first2, InIter2 last2,
              OutIter result, Compare comp);
```

1 *Effects:* Constructs a sorted union of the elements from the two ranges; that is, the set of elements that are present in one or both of the ranges.

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

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

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

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

25.3.5.3 `set_intersection` [lib.set.intersection]

```
template<InputIterator InIter1, InputIterator InIter2,
         OutputIterator<InIter1::value_type> OutIter>
  where SameType<InIter1::value_type, InIter2::value_type> &&
        LessThanComparable<InIter1::value_type>
  OutIter
    set_intersection(InIter1 first1, InIter1 last1,
                     InIter2 first2, InIter2 last2,
                     OutIter result);

template<InputIterator InIter1, InputIterator InIter2,
         OutputIterator<InIter1::value_type> OutIter,
         BinaryPredicate<InIter1::value_type, InIter2::value_type> Compare>
  where SameType<InIter1::value_type, InIter2::value_type>
  OutIter
    set_intersection(InIter1 first1, InIter1 last1,
                     InIter2 first2, InIter2 last2,
                     OutIter result, Compare comp);
```

1 *Effects:* Constructs a sorted intersection of the elements from the two ranges; that is, the set of elements that are present in both of the ranges.

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

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

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

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

25.3.5.4 `set_difference` [lib.set.difference]

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

template<InputIterator InIter1, InputIterator InIter2,
         OutputIterator<InIter1::value_type> OutIter,
         BinaryPredicate<InIter1::value_type, InIter2::value_type> Compare>
  where SameType<InIter1::value_type, InIter2::value_type>
  OutIter
    set_difference(InIter1 first1, InIter1 last1,
                  InIter2 first2, InIter2 last2,
                  OutIter result, Compare comp);
```

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

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

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

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

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

25.3.5.5 `set_symmetric_difference` [lib.set.symmetric.difference]

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

template<InputIterator InIter1, InputIterator InIter2,
         OutputIterator<InIter1::value_type> OutIter,
         BinaryPredicate<InIter1::value_type, InIter2::value_type> Compare>
  where SameType<InIter1::value_type, InIter2::value_type>
  OutIter
    set_symmetric_difference(InIter1 first1, InIter1 last1,
                             InIter2 first2, InIter2 last2,
                             OutIter result, Compare comp);
```

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

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

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

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

5 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2,last2)` contains n elements that are equivalent to them, then $|m-n|$ of those elements shall be copied to the output range: the last $m-n$ of these elements from `[first1,last1)` if $m>n$, and the last $n-m$ of these elements from `[first2,last2)` if $m<n$.

25.3.6 Heap operations [lib.alg.heap.operations]

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

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

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

2 These properties make heaps useful as priority queues.

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

25.3.6.1 `push_heap` [lib.push.heap]

```
template<MutableRandomAccessIterator Iter>
  where CopyConstructible<Iter::value_type> && LessThanComparable<Iter::value_type>
  void push_heap(Iter first, Iter last);

template<MutableRandomAccessIterator Iter, BinaryPredicate<Iter::value_type, Iter::value_type> Compare>
  where CopyConstructible<Iter::value_type>
  void push_heap(Iter first, Iter last,
                 Compare comp);
```

Note: It seems like this should use Swappable, and make the CopyConstructible requirement unnecessary.

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

2 *Requires:* The range `[first,last - 1)` shall be a valid heap.

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

25.3.6.2 `pop_heap` [lib.pop.heap]

```
template<MutableRandomAccessIterator Iter>
  where CopyConstructible<Iter::value_type> && Swappable<Iter::value_type> &&
        LessThanComparable<Iter::value_type>
  void pop_heap(Iter first, Iter last);

template<MutableRandomAccessIterator Iter, BinaryPredicate<Iter::value_type, Iter::value_type> Compare>
  where CopyConstructible<Iter::value_type> && Swappable<Iter::value_type>
  void pop_heap(Iter first, Iter last,
                Compare comp);
```

The libstdc++ implementation of this routine does not use swap() like it should, but it does type-check. Note: We may be able to eliminate the CopyConstructible requirement.

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

2 *Requires:* The range `[first, last)` shall be a valid heap. ~~The type of *first shall satisfy the Swappable requirements (20.1.4).~~

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

25.3.6.3 `make_heap` [lib.make.heap]

```
template<MutableRandomAccessIterator Iter>
  where CopyConstructible<Iter::value_type> &&
        LessThanComparable<Iter::value_type>
  void make_heap(Iter first, Iter last);

template<MutableRandomAccessIterator Iter, BinaryPredicate<Iter::value_type, Iter::value_type> Compare>
  where CopyConstructible<Iter::value_type>
  void make_heap(Iter first, Iter last,
                 Compare comp);
```

Note: Should we use Swappable instead of CopyConstructible here?

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

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

25.3.6.4 `sort_heap` [lib.sort.heap]

```
template<MutableRandomAccessIterator Iter>
  where CopyConstructible<Iter::value_type> && Swappable<Iter::value_type> &&
        LessThanComparable<Iter::value_type>
  void sort_heap(Iter first, Iter last);

template<MutableRandomAccessIterator Iter, BinaryPredicate<Iter::value_type, Iter::value_type> Compare>
  where CopyConstructible<Iter::value_type> && Swappable<Iter::value_type>
  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).~~

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

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

```
template<LessThanComparable T> const T& min(const T& a, const T& b);
template<class T, BinaryPredicate<T, T> 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& max(const T& a, const T& b);
template<class T, BinaryPredicate<T, T> Compare>
  const T& max(const T& a, const T& b, Compare comp);
```

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

5 *Returns:* The larger value.

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

```
template<ForwardIterator Iter>
  where LessThanComparable<Iter::reference>
  Iter min_element(Iter first, Iter last);

template<ForwardIterator Iter, BinaryPredicate<Iter::reference, Iter::reference> Compare>
  Iter min_element(Iter first, Iter last,
                   Compare comp);
```

7 *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`.

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

```
template<ForwardIterator Iter>
  where LessThanComparable<Iter::reference>
  Iter max_element(Iter first, Iter last);

template<ForwardIterator Iter, BinaryPredicate<Iter::reference, Iter::reference> Compare>
  Iter max_element(Iter first, Iter last,
                   Compare comp);
```

9 *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`.

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

25.3.8 Lexicographical comparison [lib.alg.lex.comparison]

```
template<InputIterator Iter1, InputIterator Iter2>
  where LessThanComparable<Iter1::reference, Iter2::reference> &&
        LessThanComparable<Iter2::reference, Iter1::reference>
  bool
    lexicographical_compare(Iter1 first1, Iter1 last1,
                            Iter2 first2, Iter2 last2);

template<InputIterator Iter1, InputIterator Iter2, class Compare>
  where BinaryPredicate<Compare, Iter1::reference, Iter2::reference> &&
        BinaryPredicate<Compare, Iter2::reference, Iter1::reference>
  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 [lib.alg.permutation.generators]

```
template<MutableBidirectionalIterator Iter>
  where Swappable<Iter::value_type> &&
        LessThanComparable<Iter::reference>
  bool next_permutation(Iter first,
                        Iter last);

template<MutableBidirectionalIterator Iter, BinaryPredicate<Iter::reference, Iter::reference> Compare>
  where Swappable<Iter::value_type>
  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<MutableBidirectionalIterator Iter>
  where Swappable<Iter::value_type> &&
        LessThanComparable<Iter::reference>
  bool prev_permutation(Iter first,
                        Iter last);

template<MutableBidirectionalIterator Iter, BinaryPredicate<Iter::reference, Iter::reference> Compare>
  where Swappable<Iter::value_type>
  bool prev_permutation(Iter first,
                        Iter last, Compare comp);
```

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

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

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

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

Bibliography

[1] Douglas Gregor and Bjarne Stroustrup. Concepts. Technical Report N2042=06-0112, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, June 2006.