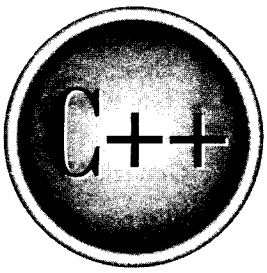
The Complete Reference



Chapter 34

The STL Algorithms

The algorithms defined by the standard template library are described here. These algorithms operate on containers through iterators. All of the algorithms are template functions. Here are descriptions of the generic type names used by the algorithms.

Generic Name	Represents
BiIter	Bidirectional iterator
ForIter	Forward iterator
InIter	Input iterator
OutIter	Output iterator
RandIter	Random access iterator
T	Some type of data
Size	Some type of integer
Func	Some type of function
Generator	A function that generates objects
BinPred	Binary predicate
UnPred	Unary predicate
Comp	Comparison function

adjacent_find

```
template <class ForIter>
  ForIter adjacent_find(ForIter start, ForIter end);
template <class ForIter, class BinPred>
  ForIter adjacent_find(ForIter start, ForIter end, BinPred pfn);
```

The **adjacent_find( )** algorithm searches for adjacent matching elements within a sequence specified by *start* and *end* and returns an iterator to the first element. If no adjacent pair is found, *end* is returned. The first version looks for equivalent elements. The second version lets you specify your own method for determining matching elements.

binary_search

```
template <class ForIter, class T>
  bool binary_search(ForIter start, ForIter end, const T &val);
```

```
template <class ForIter, class T, class Comp>
  bool binary_search(ForIter start, ForIter end, const T &val,
                     Comp cmpfn);
```

The **binary_search( )** algorithm performs a binary search on an ordered sequence beginning at *start* and ending with *end* for the value specified by *val*. It returns true if *val* is found and false otherwise. The first version compares the elements in the specified sequence for equality. The second version allows you to specify your own comparison function.

copy

```
template <class InIter, class OutIter>
  OutIter copy(InIter start, InIter end, OutIter result);
```

The **copy( )** algorithm copies a sequence beginning at *start* and ending with *end*, putting the result into the sequence pointed to by *result*. It returns an iterator to the end of the resulting sequence. The range to be copied must not overlap with *result*.

copy_backward

```
template <class BiIter1, class BiIter2>
  BiIter2 copy_backward(BiIter1 start, BiIter1 end, BiIter2 result);
```

The **copy_backward( )** algorithm is the same as **copy( )** except that it moves the elements from the end of the sequence first.

count

```
template <class InIter, class T>
  ptrdiff_t count(InIter start, InIter end, const T &val);
```

The **count( )** algorithm returns the number of elements in the sequence beginning at *start* and ending at *end* that match *val*.

count_if

```
template <class InIter, class UnPred>
  ptrdiff_t count(InIter start, InIter end, UnPred pfn);
```

The **count_if( )** algorithm returns the number of elements in the sequence beginning at *start* and ending at *end* for which the unary predicate *pfn* returns true. The type **ptrdiff_t** is defined as some form of integer.

equal

```
template <class InIter1, class InIter2>
  bool equal(InIter1 start1, InIter1 end1, InIter2 start2);
template <class InIter1, class InIter2, class BinPred>
  bool equal(InIter1 start1, InIter1 end1, InIter2 start2,
             BinPred pfn);
```

The **equal( )** algorithm determines if two ranges are the same. The range determined by *start1* and *end1* is tested against the sequence pointed to by *start2*. If the ranges are the same, true is returned. Otherwise, false is returned.

The second form allows you to specify a binary predicate that determines when two elements are equal.

equal_range

```
template <class ForIter, class T>
  pair<ForIter, ForIter> equal_range(ForIter start, ForIter end,
                                     const T &val);
template <class ForIter, class T, class Comp>
  pair<ForIter, ForIter> equal_range(ForIter start, ForIter end,
                                     const T &val, Comp cmpfn);
```

The **equal_range( )** algorithm returns a range in which an element can be inserted into a sequence without disrupting the ordering of the sequence. The region in which to search for such a range is specified by *start* and *end*. The value is passed in *val*. To specify your own search criteria, specify the comparison function *cmpfn*.

The template class **pair** is a utility class that can hold a pair of objects in its **first** and **second** members.

fill and fill_n

```
template <class ForIter, class T>
  void fill(ForIter start, ForIter end, const T &val);
template <class OutIter, class Size, class T>
  void fill_n(OutIter start, Size num, const T &val);
```

The **fill( )** and **fill_n( )** algorithms fill a range with the value specified by *val*. For **fill( )** the range is specified by *start* and *end*. For **fill_n( )**, the range begins at *start* and runs for *num* elements.

find

```
template <class InIter, class T>
  InIter find(InIter start, InIter end, const T &val);
```

The **find( )** algorithm searches the range *start* to *end* for the value specified by *val*. It returns an iterator to the first occurrence of the element or to *end* if the value is not in the sequence.

find_end

```
template <class ForIter1, class ForIter2>
  ForIter1 find_end(ForIter1 start1, ForIter1 end1,
                    ForIter2 start2, ForIter2 end2);
template <class ForIter1, class ForIter2, class BinPred>
  ForIter1 find_end(ForIter1 start1, ForIter1 end1,
                    ForIter2 start2, ForIter2 end2, BinPred pfn);
```

The **find_end( )** algorithm finds the last subsequence defined by *start2* and *end2* within the range *start1* and *end1*. If the sequence is found, an iterator to the first element in the sequence is returned. Otherwise, the iterator *end1* is returned.

The second form allows you to specify a binary predicate that determines when elements match.

find_first_of

```
template <class ForIter1, class ForIter2>
  ForIter1 find_first_of(ForIter1 start1, ForIter1 end1,
```

```
                          ForIter2 start2, ForIter2 end2);
template <class ForIter1, class ForIter2, class BinPred>
  ForIter1 find_first_of(ForIter1 start1, ForIter1 end1,
                          ForIter2 start2, ForIter2 end2,
                          BinPred pfn);
```

The **find_first_of( )** algorithm finds the first element within the sequence defined by *start1* and *end1* that matches an element within the range *start1* and *end1*. If no matching element is found, the iterator *end1* is returned.

The second form allows you to specify a binary predicate that determines when elements match.

find_if

```
template <class InIter, class UnPred>
  InIter find_if(InIter start, InIter end, UnPred pfn);
```

The **find_if( )** algorithm searches the range *start* to *end* for an element for which the unary predicate *pfn* returns true. It returns an iterator to the first occurrence of the element or to *end* if the value is not in the sequence.

for_each

```
template<class InIter, class Func>
  Func for_each(InIter start, InIter end, Func fn);
```

The **for_each( )** algorithm applies the function *fn* to the range of elements specified by *start* and *end*. It returns *fn*.

generate and generate_n

```
template <class ForIter, class Generator>
  void generate(ForIter start, ForIter end, Generator fngen);
template <class ForIter, class Size, class Generator>
  void generate_n(OutIter start, Size num, Generator fngen);
```

The algorithms **generate( )** and **generate_n( )** assign to elements in a range the values returned by a generator function. For **generate( )**, the range being assigned is

specified by *start* and *end*. For **generate_n( )**, the range begins at *start* and runs for *num* elements. The generator function is passed in *fngen*. It has no parameters.

includes

```
template <class InIter1, class InIter2>
  bool includes(InIter1 start1, InIter1 end1,
                InIter2 start2, InIter2 end2);
template <class InIter1, class InIter2, class Comp>
  bool includes(InIter1 start1, InIter1 end1,
                InIter2 start2, InIter2 end2, Comp cmpfn);
```

The **includes( )** algorithm determines if the sequence defined by *start1* and *end1* includes all of the elements in the sequence defined by *start2* and *end2*. It returns true if the elements are all found and false otherwise.

The second form allows you to specify a comparison function that determines when one element is less than another.

inplace_merge

```
template <class BiIter>
  void inplace_merge(BiIter start, BiIter mid, BiIter end);
template <class BiIter, class Comp>
  void inplace_merge(BiIter start, BiIter mid, BiIter end, Comp cmpfn);
```

Within a single sequence, the **inplace_merge( )** algorithm merges the range defined by *start* and *mid* with the range defined by *mid* and *end*. Both ranges must be sorted in increasing order. After executing, the resulting sequence is sorted in increasing order.

The second form allows you to specify a comparison function that determines when one element is less than another.

iter_swap

```
template <class ForIter1, class ForIter2>
  void iter_swap(ForIter1 i, ForIter2 j)
```

The **iter_swap( )** algorithm exchanges the values pointed to by its two iterator arguments.

lexicographical_compare

```
template <class InIter1, class InIter2>
  bool lexicographical_compare(InIter1 start1, InIter1 end1,
                               InIter2 start2, InIter2 end2);
template <class InIter1, class InIter2, class Comp>
  bool lexicographical_compare(InIter1 start1, InIter1 end1,
                               InIter2 start2, InIter2 end2,
                               Comp cmpfn);
```

The **lexicographical_compare( )** algorithm alphabetically compares the sequence defined by *start1* and *end1* with the sequence defined by *start2* and *end2*. It returns true if the first sequence is lexicographically less than the second (that is, if the first sequence would come before the second using dictionary order).

The second form allows you to specify a comparison function that determines when one element is less than another.

lower_bound

```
template <class ForIter, class T>
  ForIter lower_bound(ForIter start, ForIter end, const T &val);
template <class ForIter, class T, class Comp>
  ForIter lower_bound(ForIter start, ForIter end, const T &val,
                      Comp cmpfn);
```

The **lower_bound( )** algorithm finds the first point in the sequence defined by *start* and *end* that is not less than *val*. It returns an iterator to this point.

The second form allows you to specify a comparison function that determines when one element is less than another.

make_heap

```
template <class RandIter>
  void make_heap(RandIter start, RandIter end);
template <class RandIter, class Comp>
  void make_heap(RandIter start, RandIter end, Comp cmpfn);
```

The **make_heap( )** algorithm constructs a heap from the sequence defined by *start* and *end*.

The second form allows you to specify a comparison function that determines when one element is less than another.

max

```
template <class T>
  const T &max(const T &i, const T &j);
template <class T, class Comp>
  const T &max(const T &i, const T &j, Comp cmpfn);
```

The **max( )** algorithm returns the maximum of two values.

The second form allows you to specify a comparison function that determines when one element is less than another.

max_element

```
template <class ForIter>
  ForIter max_element(ForIter start, ForIter last);
template <class ForIter, class Comp>
  ForIter max_element(ForIter start, ForIter last, Comp cmpfn);
```

The **max_element( )** algorithm returns an iterator to the maximum element within the range *start* and *last*.

The second form allows you to specify a comparison function that determines when one element is less than another.

merge

```
template <class InIter1, class InIter2, class OutIter>
  OutIter merge(InIter1 start1, InIter1 end1,
                InIter2 start2, InIter2 end2,
                OutIter result);
template <class InIter1, class InIter2, class OutIter, class Comp>
  OutIter merge(InIter1 start1, InIter1 end1,
                InIter2 start2, InIter2 end2,
                OutIter result, Comp cmpfn);
```

The **merge( )** algorithm merges two ordered sequences, placing the result into a third sequence. The sequences to be merged are defined by *start1*, *end1* and *start2*, *end2*.

The result is put into the sequence pointed to by *result*. An iterator to the end of the resulting sequence is returned.

The second form allows you to specify a comparison function that determines when one element is less than another.

min

```
template <class T>
  const T &min(const T &i, const T &j);
template <class T, class Comp>
  const T &min(const T &i, const T &j, Comp cmpfn);
```

The **min( )** algorithm returns the minimum of two values.

The second form allows you to specify a comparison function that determines when one element is less than another.

min_element

```
template <class ForIter>
  ForIter min_element(ForIter start, ForIter last);
template <class ForIter, class Comp>
  ForIter min_element(ForIter start, ForIter last, Comp cmpfn);
```

The **min_element( )** algorithm returns an iterator to the minimum element within the range *start* and *last*.

The second form allows you to specify a comparison function that determines when one element is less than another.

mismatch

```
template <class InIter1, class InIter2>
  pair<InIter1, InIter2> mismatch(InIter1 start1, InIter1 end1,
                                  InIter2 start2);
template <class InIter1, class InIter2, class BinPred>
  pair<InIter1, InIter2> mismatch(InIter1 start1, InIter1 end1,
                                  InIter2 start2, BinPred pfn);
```

The **mismatch( )** algorithm finds the first mismatch between the elements in two sequences. Iterators to the two elements are returned. If no mismatch is found, iterators to the end of the sequence are returned.

The second form allows you to specify a binary predicate that determines when one element is equal to another.

The **pair** template class contains two data members called **first** and **second** that hold the pair of values.

next_permutation

```
template <class BiIter>
  bool next_permutation(BiIter start, BiIter end);
template <class BiIter, class Comp>
  bool next_permutation(BiIter start, BiIter end, Comp cmfn);
```

The **next_permutation( )** algorithm constructs the next permutation of a sequence. The permutations are generated assuming a sorted sequence: from low to high represents the first permutation. If the next permutation does not exist, **next_permutation( )** sorts the sequence as its first permutation and returns false. Otherwise, it returns true.

The second form allows you to specify a comparison function that determines when one element is less than another.

nth_element

```
template <class RandIter>
  void nth_element(RandIter start, RandIter element, RandIter end);
template <class RandIter, class Comp>
  void nth_element(RandIter start, RandIter element,
                   RandIter end, Comp cmpfn);
```

The **nth_element( )** algorithm arranges the sequence specified by *start* and *end* such that all elements less than *element* come before that element and all elements greater than *element* come after it.

The second form allows you to specify a comparison function that determines when one element is greater than another.

partial_sort

```
template <class RandIter>
  void partial_sort(RandIter start, RandIter mid, RandIter end);
template <class RandIter, class Comp>
  void partial_sort(RandIter start, RandIter mid,
                    RandIter end, Comp cmpfn);
```

The **partial_sort( )** algorithm sorts the range *start* to *end*. However, after execution, only elements in the range *start* to *mid* will be in sorted order.

The second form allows you to specify a comparison function that determines when one element is less than another.

partial_sort_copy

```
template <class InIter, class RandIter>
  RandIter partial_sort_copy(InIter start, InIter end,
                             RandIter res_start, RandIter res_end);
template <class InIter, class RandIter, class Comp>
  RandIter partial_sort_copy(InIter start, InIter end,
                             RandIter res_start, RandIter res_end,
                             Comp cmpfn);
```

The **partial_sort_copy( )** algorithm sorts the range *start* to *end* and then copies as many elements as will fit into the resulting sequence defined by *res_start* and *res_end*. It returns an iterator to one past the last element copied into the resulting sequence.

The second form allows you to specify a comparison function that determines when one element is less than another.

partition

```
template <class BiIter, class UnPred>
  BiIter partition(BiIter start, BiIter end, UnPred pfn);
```

The **partition( )** algorithm arranges the sequence defined by *start* and *end* such that all elements for which the predicate specified by *pfn* returns true come before those for which the predicate returns false. It returns an iterator to the beginning of the elements for which the predicate is false.

pop_heap

```
template <class RandIter>
  void pop_heap(RandIter start, RandIter end);
template <class RandIter, class Comp>
  void pop_heap(RandIter start, RandIter end, Comp cmpfn);
```

The **pop_heap( )** algorithm exchanges the *first* and *last*–1 elements and then rebuilds the heap.

The second form allows you to specify a comparison function that determines when one element is less than another.

prev_permutation

```
template <class BiIter>
  bool prev_permutation(BiIter start, BiIter end);
template <class BiIter, class Comp>
  bool prev_permutation(BiIter start, BiIter end, Comp cmpfn);
```

The **prev_permutation( )** algorithm constructs the previous permutation of a sequence. The permutations are generated assuming a sorted sequence: from low to high represents the first permutation. If the next permutation does not exist, **prev_permutation( )** sorts the sequence as its final permutation and returns false. Otherwise, it returns true.

The second form allows you to specify a comparison function that determines when one element is less than another.

push_heap

```
template <class RandIter>
  void push_heap(RandIter start, RandIter end);
template <class RandIter, class Comp>
  void push_heap(RandIter start, RandIter end, Comp cmpfn);
```

The **push_heap( )** algorithm pushes an element onto the end of a heap. The range specified by *start* and *end* is assumed to represent a valid heap.

The second form allows you to specify a comparison function that determines when one element is less than another.

random_shuffle

```
template <class RandIter>
  void random_shuffle(RandIter start, RandIter end);
template <class RandIter, class Generator>
  void random_shuffle(RandIter start, RandIter end, Generator rand_gen);
```

The **random_shuffle( )** algorithm randomizes the sequence defined by *start* and *end*.

The second form specifies a custom random number generator. This function must have the following general form:

rand_gen(*num*);

It must return a random number between zero and *num*.

remove, remove_if, remove_copy, and remove_copy_if

```
template <class ForIter, class T>
  ForIter remove(ForIter start, ForIter end, const T &val);
template <class ForIter, class UnPred>
  ForIter remove_if(ForIter start, ForIter end,  UnPred pfn);
template <class InIter, class OutIter, class T>
  OutIter remove_copy(InIter start, InIter end,
                      OutIter result, const T &val);
template <class InIter, class OutIter, class UnPred>
  OutIter remove_copy_if(InIter start, InIter end,
                         OutIter result, UnPred pfn);
```

The **remove( )** algorithm removes elements from the specified range that are equal to *val*. It returns an iterator to the end of the remaining elements.

The **remove_if( )** algorithm removes elements from the specified range for which the predicate *pfn* is true. It returns an iterator to the end of the remaining elements.

The **remove_copy( )** algorithm copies elements from the specified range that are equal to *val* and puts the result into the sequence pointed to by *result*. It returns an iterator to the end of the result.

The **remove_copy_if( )** algorithm copies elements from the specified range for which the predicate *pfn* is true and puts the result into the sequence pointed to by *result*. It returns an iterator to the end of the result.

replace, replace_copy, replace_if, and replace_copy_if

```
template <class ForIter, class T>
  void replace(ForIter start, ForIter end,
               const T &old, const T &new);
template <class ForIter, class UnPred, class T>
  void replace_if(ForIter start, ForIter end,
                  UnPred pfn, const T &new);
template <class InIter, class OutIter, class T>
```

```
OutIter replace_copy(InIter start, InIter end, OutIter result,
                     const T &old, const T &new);
template <class InIter, class OutIter, class UnPred, class T>
  OutIter replace_copy_if(InIter start, InIter end, OutIter result,
                          UnPred pfn, const T &new);
```

Within the specified range, the **replace( )** algorithm replaces elements with the value *old* with elements that have the value *new*.

Within the specified range, the **replace_if( )** algorithm replaces those elements for which the predicate *pfn* is true with elements that have the value *new*.

Within the specified range, the **replace_copy( )** algorithm copies elements to *result*. In the process it replaces elements that have the value *old* with elements that have the value *new*. The original range is unchanged. An iterator to the end of *result* is returned.

Within the specified range, the **replace_copy_if( )** algorithm copies elements to *result*. In the process it replaces elements for which the predicate *pfn* returns true with elements that have the value *new*. The original range is unchanged. An iterator to the end of *result* is returned.

reverse and reverse_copy

```
template <class BiIter>
  void reverse(BiIter start, BiIter end);
template <class BiIter, class OutIter>
  OutIter reverse_copy(BiIter start, BiIter end, OutIter result);
```

The **reverse( )** algorithm reverses the order of the range specified by *start* and *end*.

The **reverse_copy( )** algorithm copies in reverse order the range specified by *start* and *end* and stores the result in *result*. It returns an iterator to the end of *result*.

rotate and rotate_copy

```
template <class ForIter>
  void rotate(ForIter start, ForIter mid, ForIter end);
template <class ForIter, class OutIter>
  OutIter rotate_copy(ForIter start, ForIter mid, ForIter end,
                      OutIter result);
```

The **rotate( )** algorithm left-rotates the elements in the range specified by *start* and *end* so that the element specified by *mid* becomes the new first element.

The **rotate_copy( )** algorithm copies the range specified by *start* and *end*, storing the result in *result*. In the process it left-rotates the elements so that the element specified by *mid* becomes the new first element. It returns an iterator to the end of *result*.

search

```
template <class ForIter1, class ForIter2>
  ForIter1 search(ForIter1 start1, ForIter1 end1,
                  ForIter2 start2, ForIter2 end2);
template <class ForIter1, class ForIter2, class BinPred>
  ForIter1 search(ForIter1 start1, ForIter1 end1,
                  ForIter2 start2, ForIter2 end2, BinPred pfn);
```

The **search( )** algorithm searches for a subsequence within a sequence. The sequence being searched is defined by *start1* and *end1*. The subsequence being searched is specified by *start2* and *end2*. If the subsequence is found, an iterator to its beginning is returned. Otherwise, *end1* is returned.

The second form allows you to specify a binary predicate that determines when one element is equal to another.

search_n

```
template <class ForIter, class Size, class T>
  ForIter search_n(ForIter start, ForIter end,
                   Size num, const T &val);
template <class ForIter, class Size, class T, class BinPred>
  ForIter search_n(ForIter start, ForIter end,
                   Size num, const T &val, BinPred pfn);
```

The **search_n( )** algorithm searches for a sequence of *num* elements equal to *val* within a sequence. The sequence being searched is defined by *start1* and *end1*. If the subsequence is found, an iterator to its beginning is returned. Otherwise, *end* is returned.

The second form allows you to specify a binary predicate that determines when one element is equal to another.

set_difference

```
template <class InIter1, class InIter2, class OutIter>
  OutIter set_difference(InIter1 start1, InIter1 end1,
```

```
            InIter2 start2, InIter2 end2, OutIter result);
template <class InIter1, class InIter2, class OutIter, class Comp>
  OutIter set_difference(InIter1 start1, InIter1 end1,
            InIter2 start2, InIter2 end2,
            OutIter result, Comp cmpfn);
```

The **set_difference( )** algorithm produces a sequence that contains the difference between the two ordered sets defined by *start1, end1* and *start2, end2*. That is, the set defined by *start2, end2* is subtracted from the set defined by *start1, end1*. The result is ordered and put into *result*. It returns an iterator to the end of the result.

The second form allows you to specify a comparison function that determines when one element is less than another.

set_intersection

```
template <class InIter1, class InIter2, class OutIter>
  OutIter set_intersection(InIter1 start1, InIter1 end1,
            InIter2 start2, InIter2 end2, OutIter result);
template <class InIter1, class InIter2, class OutIter, class Comp>
  OutIter set_intersection(InIter1 start1, InIter1 end1,
            InIter2 start2, InIter2 end2,
            OutIter result, Comp cmpfn);
```

The **set_intersection( )** algorithm produces a sequence that contains the intersection of the two ordered sets defined by *start1, end1* and *start2, end2*. These are the elements common to both sets. The result is ordered and put into *result*. It returns an iterator to the end of the result.

The second form allows you to specify a comparison function that determines when one element is less than another.

set_symmetric_difference

```
template <class InIter1, class InIter2, class OutIter>
  OutIter set_symmetric_difference(InIter1 start1, InIter1 end1,
            InIter2 start2, InIter2 end2, OutIter result);
template <class InIter1, class InIter2, class OutIter, class Comp>
  OutIter set_symmetric_difference(InIter1 start1, InIter1 end1,
            InIter2 start2, InIter2 end2, OutIter result,
            Comp cmpfn);
```

The **set_symmetric_difference( )** algorithm produces a sequence that contains the symmetric difference between the two ordered sets defined by *start1, end1* and *start2, end2*. That is, the resultant set contains only those elements that are not common to both sets. The result is ordered and put into *result*. It returns an iterator to the end of the result.

The second form allows you to specify a comparison function that determines when one element is less than another.

set_union

```
template <class InIter1, class InIter2, class OutIter>
  OutIter set_union(InIter1 start1, InIter1 end1,
            InIter2 start2, InIter2 end2, OutIter result);
template <class InIter1, class InIter2, class OutIter, class Comp>
  OutIter set_union(InIter1 start1, InIter1 end1,
            InIter2 start2, InIter2 end2, OutIter result,
            Comp cmpfn);
```

The **set_union( )** algorithm produces a sequence that contains the union of the two ordered sets defined by *start1, end1* and *start2, end2*. Thus, the resultant set contains those elements that are in both sets. The result is ordered and put into *result*. It returns an iterator to the end of the result.

The second form allows you to specify a comparison function that determines when one element is less than another.

sort

```
template <class RandIter>
  void sort(RandIter start, RandIter end);
template <class RandIter, classComp>
  void sort(RandIter start, RandIter end, Comp cmpfn);
```

The **sort( )** algorithm sorts the range specified by *start* and *end*.

The second form allows you to specify a comparison function that determines when one element is less than another.

sort_heap

```
template <class RandIter>
  void sort_heap(RandIter start, RandIter end);
```

```
template <class RandIter, class Comp>
  void sort_heap(RandIter start, RandIter end, Comp cmpfn);
```

The **sort_heap( )** algorithm sorts a heap within the range specified by *start* and *end*.

The second form allows you to specify a comparison function that determines when one element is less than another.

stable_partition

```
template <class BiIter, class UnPred>
  BiIter stable_partition(BiIter start, BiIter end, UnPred pfn);
```

The **stable_partition( )** algorithm arranges the sequence defined by *start* and *end* such that all elements for which the predicate specified by *pfn* returns true come before those for which the predicate returns false. The partitioning is stable. This means that the relative ordering of the sequence is preserved. It returns an iterator to the beginning of the elements for which the predicate is false.

stable_sort

```
template <class RandIter>
  void stable_sort(RandIter start, RandIter end);
template <class RandIter, class Comp>
  void stable_sort(RandIter start, RandIter end, Comp cmpfn);
```

The **sort( )** algorithm sorts the range specified by *start* and *end*. The sort is stable. This means that equal elements are not rearranged.

The second form allows you to specify a comparison function that determines when one element is less than another.

swap

```
template <class T>
  void swap(T &i, T &j);
```

The **swap( )** algorithm exchanges the values referred to by *i* and *j*.

swap_ranges

```
template <class ForIter1, class ForIter2>
  ForIter2 swap_ranges(ForIter1 start1, ForIter1 end1,
                       ForIter2 start2);
```

The **swap_ranges( )** algorithm exchanges elements in the range specified by *start1* and *end1* with elements in the sequence beginning at *start2*. It returns an iterator to the end of the sequence specified by *start2*.

transform

```
template <class InIter, class OutIter, class Func>
  OutIter transform(InIter start, InIter end,
                    OutIter result, Func unaryfunc);
template <class InIter1, class InIter2, class OutIter, class Func>
  OutIter transform(InIter1 start1, InIter1 end1,
                    InIter2 start2, OutIter result,
                    Func binaryfunc);
```

The **transform( )** algorithm applies a function to a range of elements and stores the outcome in *result*. In the first form, the range is specified by *start* and *end*. The function to be applied is specified by *unaryfunc*. This function receives the value of an element in its parameter and it must return its transformation.

In the second form, the transformation is applied using a binary operator function that receives the value of an element from the sequence to be transformed in its first parameter and an element from the second sequence as its second parameter.

Both versions return an iterator to the end of the resulting sequence.

unique and unique_copy

```
template <class ForIter>
  ForIter unique(ForIter start, ForIter end);
template <class ForIter, class BinPred>
  ForIter unique(ForIter start, ForIter end, BinPred pfn);
template <class ForIter, class OutIter>
  OutIter unique_copy(ForIter start, ForIter end, OutIter result);
template <class ForIter, class OutIter, class BinPred>
  OutIter unique_copy(ForIter start, ForIter end, OutIter result,
                      BinPred pfn);
```

The **unique( )** algorithm eliminates duplicate elements from the specified range. The second form allows you to specify a binary predicate that determines when one element is equal to another. **unique( )** returns an iterator to the end of the range.

The **unique_copy( )** algorithm copies the range specified by *start1* and *end1*, eliminating duplicate elements in the process. The outcome is put into *result*. The second form allows you to specify a binary predicate that determines when one element is equal to another. **unique_copy( )** returns an iterator to the end of the range.

upper_bound

```
template <class ForIter, class T>
  ForIter upper_bound(ForIter start, ForIter end, const T &val);
template <class ForIter, class T, class Comp>
  ForIter upper_bound(ForIter start, ForIter end, const T &val,
                      Comp cmpfn);
```

The **upper_bound( )** algorithm finds the last point in the sequence defined by *start* and *end* that is not greater than *val*. It returns an iterator to this point.

The second form allows you to specify a comparison function that determines when one element is less than another.