APPENDIX H

STL Algorithms

This appendix gives a summary of each of the function templates provided by the Standard Template Library (STL). These function templates are defined in the `<algorithm>` header file, and can be categorized as follows:

- **Min/max algorithms:**
  Algorithms that determine the smallest (min) and largest (max) values in a range of elements
- **Sorting algorithms:**
  Algorithms that sort a range of elements, or determine whether a range is sorted
- **Search algorithms:**
  Algorithms that perform various searching operations on a sorted range of elements
- **Read-only sequence algorithms:**
  Algorithms that iterate over a range of elements, performing various operations that do not modify the elements
- **Copying and moving algorithms:**
  Algorithms use various techniques to copy or move ranges of elements.
- **Swapping algorithms:**
  Algorithms that swap values, ranges of elements, or the values pointed to by iterators
- **Replacement algorithms:**
  Algorithms that replace elements of a specified value
- **Removal algorithms:**
  Algorithms that remove elements
- **Reversal algorithms:**
  Algorithms that reverse the order of elements in a range
- **Fill algorithms:**
  Algorithms that fill the elements in a range with values
- **Rotation algorithms:**
  Algorithms that rotate the elements in a range with values
- **Shuffling algorithms:**
  Algorithms that shuffle the elements in a range
- **Set algorithms:**
  Algorithms that perform common mathematical set operations on a range of elements
- **Transformation algorithms:**
  Algorithms that iterate over ranges of elements, performing some operation using each element

- **Partition algorithms:**
  Algorithms that partition a range of elements into two groups
- **Merge algorithms:**
  Algorithms that merge ranges of elements
- **Permutation algorithms:**
  Algorithms that rearrange of a range of elements into all of its different possible permutations
- **Heap algorithms:**
  Algorithms that create and work with a heap data structure
- **Lexicographical comparison algorithm:**
  An algorithm that makes a lexicographical comparison between two ranges of elements

Min/Max Algorithms

The min/max algorithms determine the smaller or larger of two values. The functions are described in Table H-1.

Table H-1 Min/Max Functions

Function	Description
`min(value1, value2)`	Returns the smaller of *value1* or *value2*.
`max(value1, value2)`	Returns the larger of *value1* or *value2*.
`minmax(value1, value2)`	Returns a `pair` object. The `pair` object's `first` member will be the smaller of *value1* or *value2*, and its `second` member will be the larger of *value1* or *value2*.
`min_element(iterator1, iterator2)`	*iterator1* and *iterator2* each point to elements. The function returns an iterator pointing to the smaller of the two elements.
`max_element(iterator1, iterator2)`	*iterator1* and *iterator2* each point to elements. The function returns an iterator pointing to the larger of the two elements.
`minmax_element(iterator1, iterator2)`	*iterator1* and *iterator2* each point to elements. The function returns a `pair` object. The `pair` object's `first` member will be an iterator pointing to the smaller of the elements, and its `second` member will be an iterator pointing to the larger of the elements.

Sorting Algorithms

The sorting algorithms perform operations related to sorting a range of elements. Most of the functions are described in Table H-2.

Table H-2 Sorting Functions

Function	Description
`sort(iterator1, iterator2)`	*iterator1* and *iterator2* mark the beginning and end of a range of elements. The function sorts the range of elements in ascending order.
`stable_sort(iterator1, iterator2)`	*iterator1* and `iterator2` mark the beginning and end of a range of elements. The function sorts the range of elements in ascending order, but does not change the relative order of elements that are equal in value.

Table H-2 *(continued)*

Function	Description
`partial_sort (firstIt, stopSortIt, lastIt)`	*firstIt* and *lastIt* are iterators that mark the beginning and end of a range of elements. *stopSortIt* is an iterator that points to an element somewhere within the range. The function partially sorts the range. When the function is finished, the elements that appear before *stopSortIt* will contain the smallest values in the range, and they will be sorted in ascending order. The remaining elements will not be sorted.
`is_sorted(iterator1, iterator2)`	*iterator1* and *iterator2* mark the beginning and end of a range of elements. The function returns `true` if the range is sorted in ascending order, or `false` otherwise.
`is_sorted_until (iterator1, iterator2)`	*iterator1* and *iterator2* mark the beginning and end of a range of elements. The function steps through the range until it finds an element that is out of ascending order, and it returns an iterator pointing to that element. If the entire range is sorted in ascending order, the function returns an iterator pointing to the end of the range.
`nth_element (firstIt, nthIt, lastIt)`	*firstIt* and *lastIt* are iterators that mark the beginning and end of a range of elements. *nthIt* is an iterator that points to an element somewhere within the range. The function rearranges the values in the range so that the *nthIt* element contains the value that would be in that position if the entire range were sorted in ascending order. The remaining elements will not be sorted.

Search Algorithms

The search algorithms perform various searching operations on a range of elements. The functions are described in Table H-3.

Table H-3 Search Functions

Function Template	Description
`adjacent_find(iterator1, iterator2)`	The *iterator1* and *iterator2* arguments mark the beginning and end of a range of elements. The function searches for two consecutive elements that have the same value. Returns an iterator to the first element of the first occurrence. If no occurrences are found, the function returns *iterator2*. (This function uses the == operator to compare elements. If the elements contain your own class objects, be sure to overload that operator.)
`binary_search (iterator1, iterator2, value)`	The *iterator1* and *iterator2* arguments mark the beginning and end of a sorted range of elements, and *value* is the value to search for. The function returns `true` if the *value* is found in the range of elements, or `false` otherwise. (This function uses the < operator to compare elements. If the elements contain your own class objects, be sure to overload that operator.)

(table continues)

Table H-3 *(continued)*

Function Template	Description
`equal_range(`*`iterator1`*`, `*`iterator2`*`, `*`value`*`)`	The *`iterator1`* and *`iterator2`* arguments mark the beginning and end of a sorted range of elements, and *`value`* is the value to search for. Returns a `pair` object. The `pair` object's `first` member is an iterator pointing to the first element in the `set` that matches the specified *`value`*. The `pair` object's `second` member is an iterator pointing to the position *after* the last element that matches the specified *`value`*. If the specified *`value`* is not found, both iterators will point to the element that would naturally appear after the element that was searched for. (This function uses the `<` operator to compare elements. If the elements contain your own class objects, be sure to overload that operator.)
`find(`*`iterator1`*`, `*`iterator2`*`, `*`value`*`)`	The function returns `true` if the *`value`* is found in the range of elements, or `false` otherwise. (This function uses the `==` operator to compare elements. If the elements contain your own class objects, be sure to overload that operator.)
`find_if(`*`iterator1`*`, `*`iterator2`*`, `*`function`*`)`	The *`iterator1`* and *`iterator2`* arguments mark the beginning and end of a range of elements. The *`function`* argument is the address of a function, or is a function object, that accepts an element as its argument, and returns either `true` or `false`. The function starts iterating over the range of elements, passing each element as an argument to *`function`*. The `find_if` function returns an iterator to the first element that causes *`function`* to return `true`.
`find_if_not(`*`iterator1`*`, `*`iterator2`*`, `*`function`*`)`	Works the same as `find_if`, except that it returns an iterator to the first element that causes *`function`* to return `false`.
`find_end(`*`iterator1`*`, `*`iterator2`*`, `*`iterator3`*`, `*`iterator4`*`)`	Searches for a range within a range. The *`iterator1`* and *`iterator2`* arguments mark the beginning and end of the range to search, and *`iterator3`* and *`iterator4`* mark the beginning and end of the range to search for. The function returns an iterator pointing to the first element of the range-within-a-range. If the range is found multiple times, the function returns an iterator to the first element of the last occurrence. If the range is not found, the function returns *`iterator2`*. (This function uses the `==` operator to compare elements. If the elements contain your own class objects, be sure to overload that operator.)

Table H-3 *(continued)*

Function Template	Description
`find_first_of(iterator1, iterator2, iterator3, iterator4)`	The *iterator1* and *iterator2* arguments mark the beginning and end of range #1, and *iterator3* and *iterator4* mark the beginning and end of range #2. The function returns an iterator pointing to the first element in range #1 that contains any of the values in range #2. If none of the values in range #2 are found in range #1, the function returns *iterator2*. (This function uses the == operator to compare elements. If the elements contain your own class objects, be sure to overload that operator.)
`lower_bound(iterator1, iterator2, value)`	The *iterator1* and *iterator2* arguments mark the beginning and end of a sorted range of elements, and *value* is the value for which to search. Returns an iterator pointing to the first element with a value that is equal to or greater than *value*. (This function uses the < operator to compare elements. If the elements contain your own class objects, be sure to overload that operator.)
`upper_bound(iterator1, iterator2, value)`	The *iterator1* and *iterator2* arguments mark the beginning and end of a sorted range of elements, and *value* is the value for which to search. Returns an iterator pointing to the first element with a value that is greater than *value*. (This function uses the < operator to compare elements. If the elements contain your own class objects, be sure to overload that operator.)
`search(iterator1, iterator2, iterator3, iterator4)`	Searches for a range within a range. The *iterator1* and *iterator2* arguments mark the beginning and end of the range to search, and *iterator3* and *iterator4* mark the beginning and end of the range for which to search. The function returns an iterator pointing to the first element of the first occurrence of the range-within-a-range. If the range is not found, the function returns *iterator2*. (This function uses the == operator to compare elements. If the elements contain your own class objects, be sure to overload that operator.)
`search_n(iterator1, iterator2, count, value)`	The *iterator1* and *iterator2* arguments mark the beginning and end of the range to search. The function searches for a sequence of *count* elements containing *value*. If such a sequence is found, the function returns an iterator pointing to the first element of the sequence. Otherwise, *iterator2* is returned. (This function uses the == operator to compare elements. If the elements contain your own class objects, be sure to overload that operator.)

Read-Only Sequence Algorithms

The read-only sequence algorithms iterate over a range of elements, performing some operation that does not modify the elements. The functions are described in Table H-4.

Table H-4 Read-Only Sequence Functions

Function Template	Description
`all_of(`*`iterator1`*`, `*`iterator2`*`, `*`function`*`)`	The *`iterator1`* and *`iterator2`* arguments mark the beginning and end of a range of elements. The *`function`* argument is the address of a function, or is a function object, that accepts an element as its argument, and returns either `true` or `false`. (The function should not make changes to the element.) The `all_of` function iterates over the range of elements, passing each element as an argument to *`function`*. The `all_of` function returns `true` if *`function`* returns `true` for all of the elements in the range, or `false` otherwise.
`any_of(`*`iterator1`*`, `*`iterator2`*`, `*`function`*`)`	The *`iterator1`* and *`iterator2`* arguments mark the beginning and end of a range of elements. The *`function`* argument is the address of a function, or is a function object, that accepts an element as its argument, and returns either `true` or `false`. (The function should not make changes to the element.) The `any_of` function iterates over the range of elements, passing each element as an argument to *`function`*. The `any_of` function returns `true` if *`function`* returns `true` for any of the elements in the range, or `false` otherwise.
`none_of(`*`iterator1`*`, `*`iterator2`*`, `*`function`*`)`	The *`iterator1`* and *`iterator2`* arguments mark the beginning and end of a range of elements. The *`function`* argument is the address of a function, or is a function object, that accepts an element as its argument, and returns either `true` or `false`. (The function should not make changes to the element.) The `none_of` function iterates over the range of elements, passing each element as an argument to *`function`*. The `none_of` function returns `true` if *`function`* returns `false` for all of the elements in the range, or `false` otherwise.
`for_each(`*`iterator1`*`, `*`iterator2`*`, `*`function`*`)`	The *`iterator1`* and *`iterator2`* arguments mark the beginning and end of a range of elements. The *`function`* argument is the address of a function, or is a function object, that accepts an element as its argument (Any value returned from *`function`* is ignored). The `for_each` function iterates over the range of elements, passing each element as an argument to *`function`*.

Table H-4 *(continued)*

Function Template	Description
count(*iterator1*, *iterator2*, *value*)	The *iterator1* and *iterator2* arguments mark the beginning and end of a range of elements, and value is a value for which to search. Returns the number of elements in the range that match *value*. (This function uses the == operator to compare elements. If the elements contain your own class objects, be sure to overload that operator.)
count_if(*iterator1*, *iterator2*, *function*)	The *iterator1* and *iterator2* arguments mark the beginning and end of a range of elements. The *function* argument is the address of a function, or is a function object, that accepts an element as its argument, and returns either true or false. (The function should not make changes to the element.) The count_if function iterates over the range of elements, passing each element as an argument to *function*. The count_if function returns the number of elements for which *function* returns true.
mismatch(*iterator1*, *iterator2*, *iterator3*)	The *iterator1* and *iterator2* arguments mark the beginning and end of the 1st range of elements. The *iterator3* argument marks the beginning of the 2nd range of elements, assumed to have the same number of elements as the 1st range. The function compares the elements of the two ranges, watching for two elements that do not match. The function returns a pair object containing iterators. If two nonmatching elements were found, the pair object's first member points to the mismatched element in the first range, and the second member points to the corresponding element in the 2nd range. If all of the elements in both ranges match, the first member will point to *iterator2* and second will point to its corresponding element in the 2nd range. If none of the elements match, the pair object will contain *iterator1* and *iterator2*. (This function uses the == operator to compare elements. If the elements contain your own class objects, be sure to overload that operator.)
equal(*iterator1*, *iterator2*, *iterator3*)	The *iterator1* and *iterator2* arguments mark the beginning and end of the 1st range of elements. The *iterator3* argument marks the beginning of the 2nd range of elements, assumed to have the same number of elements as the 1st range. The function returns true if both ranges are equal to each other. (This function uses the == operator to compare elements. If the elements contain your own class objects, be sure to overload that operator.)

Copying and Moving Algorithms

The copying and moving algorithms use various techniques to copy or move ranges of elements. The functions are described in Table H-5.

Table H-5 Copying and Moving Functions

Function Template	Description

`copy(`*`iterator1`*`,`
`iterator2``,`
`iterator3``)`
	The *`iterator1`* and *`iterator2`* arguments mark the beginning and end of the 1st range of elements. The *`iterator3`* argument marks the beginning of a 2nd range of elements. The function copies all of the elements from the 1st range to the 2nd range. The function returns an iterator pointing to the end of the 2nd range.

`copy(`*`iterator1`*`,` *`n`*`,`
`iterator2``,`
`iterator3``)`
	The *`iterator1`* and *`iterator2`* arguments mark the beginning and end of the 1st range of elements. The *`iterator3`* argument marks the beginning of a 2nd range of elements. The function copies the first *`n`* elements from the 1st range to the 2nd range. The function returns an iterator pointing to the end of the 2nd range.

`copy_if(`*`iterator1`*`,`
`iterator2``,`
`iterator3``,`
`function``)`
	The *`iterator1`* and *`iterator2`* arguments mark the beginning and end of the 1st range of elements. The *`iterator3`* argument marks the beginning of a 2nd range of elements. The *`function`* argument is the address of a function, or is a function object, that accepts an element as its argument, and returns either `true` or `false`. The `copy_if` function passes each element of the 1st range as an argument to *`function`*. Each element that causes *`function`* to return `true` is copied to the 2nd range. The `copy_if` function returns an iterator pointing to the end of the 2nd range.

`copy_backward(`*`iterator1`*`,`
`iterator2``,`
`iterator3``)`
	The *`iterator1`* and *`iterator2`* arguments mark the beginning and end of the 1st range of elements. The *`iterator3`* argument marks the *end* of a 2nd range of elements. The function copies all of the elements from the 1st range to the 2nd range, and the result is the same as that of the copy function. However, this function's copy operation takes place from the end of each range, to the beginning of each range. The function returns an iterator pointing to the first element of the 2nd range.

`move(`*`iterator1`*`,`
`iterator2``,`
`iterator3``)`
	The *`iterator1`* and *`iterator2`* arguments mark the beginning and end of the 1st range of elements. The *`iterator3`* argument marks the beginning of a 2nd range of elements. The function moves all of the elements from the 1st range to the 2nd range. The function returns an iterator pointing to the end of the 2nd range. After the function executes, the 1st range will contain unspecified values.

`move_backward(`*`iterator1`*`,`
`iterator2``,`
`iterator3``)`
	Works like the `copy_backward` function, but the elements are moved from the 1st range to the 2nd range instead of being copied. The function returns an iterator pointing to the first element of the 2nd range. After the function executes, the 1st range will contain unspecified values.

Swapping Algorithms

The swapping algorithms swap two values in memory. They can be primitive values, objects, ranges of elements, or the values pointed to by iterators. The functions are described in Table H-6.

Table H-6 Swapping Functions

Function Template	Description
`swap(value1, value2)`	Swaps the values of *value1* and *value2*.
`swap(iterator1, iterator2, iterator3)`	The *iterator1* and *iterator2* arguments mark the beginning and end of the 1st range of elements. The *iterator3* argument marks the beginning of a 2nd range of elements. The function swaps the values of the elements in the two ranges.
`iter_swap(iterator1, iterator2)`	Swaps the contents of the elements pointed to by the two iterators.

Replacement Algorithms

The replacement algorithms replace elements of a specified value. The functions are described in Table H-7.

Table H-7 Replacement Functions

Function Template	Description
`replace(iterator1, iterator2, oldValue, newValue)`	The *iterator1* and *iterator2* arguments mark the beginning and end of a range of elements. The function finds all of the elements containing *oldValue* in the range, and changes their value to *newValue*.
`replace_if(iterator1, iterator2, function, newValue)`	The *iterator1* and *iterator2* arguments mark the beginning and end of a range of elements. The *function* argument is the address of a function, or is a function object, that accepts an element as its argument, and returns either `true` or `false`. (The function should not modify the element.) The `replace_if` function passes each element of the range as an argument to *function*. Each element that causes *function* to return `true` is assigned the value *newValue*.
`replace_copy(iterator1, iterator2, iterator3, oldValue, newValue)`	The *iterator1* and *iterator2* arguments mark the beginning and end of the 1st range of elements. The *iterator3* argument marks the beginning of a 2nd range of elements. The function copies the elements from the 1st range to the 2nd range, replacing each element that has the value *oldValue* with *newValue*. The function returns an iterator pointing to the end of the 2nd range.

(table continues)

Table H-7 *(continued)*

Function Template	Description
`replace_copy_if(iterator1, iterator2, iterator3, function, newValue)`	The *iterator1* and *iterator2* arguments mark the beginning and end of the 1st range of elements. The *iterator3* argument marks the beginning of a 2nd range of elements. The *function* argument is the address of a function, or is a function object, that accepts an element as its argument, and returns either `true` or `false`. (The *function* should not modify the element.) The `copy_if` function copies the elements from the 1st range to the 2nd range, passing each element an argument to *function*. If an element causes *function* to return `true`, its value in the 2nd range is changed to *newValue*. The `copy_if` function returns an iterator pointing to the end of the 2nd range.

Removal Algorithms

The removal algorithms remove elements from a range. The functions are described in Table H-8.

Table H-8 Removal Functions

Function Template	Description
`remove(iterator1, iterator2, value)`	The *iterator1* and *iterator2* arguments mark the beginning and end of a range of elements. The function removes all elements matching *value* from the range. The function returns an iterator to the end of the range.
`remove_if(iterator1, iterator2, function)`	The *iterator1* and *iterator2* arguments mark the beginning and end of a range of elements. The *function* argument is the address of a function, or is a function object, that accepts an element as its argument, and returns either `true` or `false`. (The function should not modify the element.) The `remove_if` function passes each element of the range as an argument to *function*. Each element that causes *function* to return `true` is removed from the range. The function returns an iterator to the end of the range.
`remove_copy(iterator1, iterator2, iterator3, value)`	The *iterator1* and *iterator2* arguments mark the beginning and end of the 1st range of elements. The *iterator3* argument marks the beginning of a 2nd range of elements. The function copies the elements from the 1st range to the 2nd range, except those elements with a value matching *value*. The function returns an iterator pointing to the end of the 2nd range.

Table H-8 *(continued)*

Function Template	Description
remove_copy_if(*iterator1*, *iterator2*, *iterator3*, *function*)	The *iterator1* and *iterator2* arguments mark the beginning and end of the 1st range of elements. The *iterator3* argument marks the beginning of a 2nd range of elements. The *function* argument is the address of a function, or is a function object, that accepts an element as its argument, and returns either true or false. (The *function* should not modify the element.) The remove_copy_if function copies the elements from the 1st range to the 2nd range, passing each element an argument to *function*. If an element causes *function* to return true, it is not copied. The copy_if function returns an iterator pointing to the end of the 2nd range.
unique(*iterator1*, *iterator2*)	The *iterator1* and *iterator2* arguments mark the beginning and end of a range of elements. If the range contains any consecutive elements that are duplicates, all are removed except the first one. The function returns an iterator to the end of the range.
unique_copy(*iterator1*, *iterator2*, *iterator3*)	The *iterator1* and *iterator2* arguments mark the beginning and end of the 1st range of elements. The *iterator3* argument marks the beginning of a 2nd range of elements. The function copies the elements from the 1st range to the 2nd range, except consecutive duplicate elements. The function returns an iterator pointing to the end of the 2nd range.

Reversal Algorithms

The reversal algorithms reverse the order of elements in a range. The functions are described in Table H-9.

Table H-9 Reversal Functions

Function Template	Description
reverse(*iterator1*, *iterator2*)	The *iterator1* and *iterator2* arguments mark the beginning and end of a range of elements. The function reverses the order of the elements in the range.
reverse_copy(*iterator1*, *iterator2*, *iterator3*)	The *iterator1* and *iterator2* arguments mark the beginning and end of the 1st range of elements. The *iterator3* argument marks the beginning of a 2nd range of elements. The function copies the elements from the 1st range to the 2nd range, in reverse order.

Fill Algorithms

The fill algorithms fill the elements in a range with a value. The functions are described in Table H-10.

Table H-10 Fill Functions

Function Template	Description
`fill(`*`iterator1`*`, `*`iterator2`*`, `*`value`*`)`	The *`iterator1`* and *`iterator2`* arguments mark the beginning and end of a range of elements. The function assigns *`value`* to each element in the range.
`fill(`*`iterator1`*`, `*`n`*`, `*`value`*`)`	The *`iterator1`* argument points to the first element in a range of elements, and the *`n`* argument is an integer. The function assigns *`value`* to *`n`* elements, beginning at *`iterator1`*.
`generate(`*`iterator1`*`, `*`iterator2`*`, `*`function`*`)`	The *`iterator1`* and *`iterator2`* arguments mark the beginning and end of a range of elements. The *`function`* argument is the address of a function, or is a function object, that accepts no argument, and returns a value of a type that is compatible with the elements in the range. For each element in the range, *`function`* is called, and the value it returns is assigned to the element.
`generate_n(`*`iterator1`*`, `*`n`*`, `*`function`*`)`	The *`iterator1`* argument points to the first element in a range of elements, and the *`n`* argument is an integer. The *`function`* argument is the address of a function, or is a function object, that accepts no argument, and returns a value of a type that is compatible with the elements in the range. For *`n`* elements in the range, *`function`* is called, and the value it returns is assigned to the element.

Rotation Algorithms

The rotation algorithms rotate the elements in a range. The functions are described in Table H-11.

Table H-11 Rotation Functions

Function Template	Description
`rotate(`*`firstIt`*`, `*`middleIt`*`, `*`lastIt`*`)`	*`firstIt`* and *`lastIt`* are iterators that mark the beginning and end of a range of elements. *`middleIt`* is an iterator that points to an element somewhere within the range. The function rotates the elements such that the element pointed to by *`middleIt`* becomes the first element in the range. The function returns an iterator pointing to the element containing the value that was first in the range, before the rotation took place.

Table H-11 *(continued)*

Function Template	Description
`rotate(firstIt, middleIt, lastIt, outputIt)`	`firstIt` and `lastIt` are iterators that mark the beginning and end of range #1. `middleIt` is an iterator that points to an element somewhere within range #1. `outputIt` is an iterator pointing to the first element in range #2. The function copies the elements from range #1 to range #2. The elements in range #2 are rotated such that the element pointed to by `middleIt` becomes the first element in the range. The function returns an iterator pointing to the end of range #2.

Shuffling Algorithms

The shuffling algorithms shuffle the elements in a range. The functions are described in Table H-12.

Table H-12 Rotation Functions

Function Template	Description
`random_shuffle(iterator1, iterator2)`	The `iterator1` and `iterator2` arguments mark the beginning and end of a range of elements. The function randomly shuffles the elements in the range.
`shuffle(iterator1, iterator2, generator)`	The `iterator1` and `iterator2` arguments mark the beginning and end of a range of elements. The `generator` argument is a function that serves as a uniform random number generator. The function shuffles the elements in the range, using the `generator` function to determine the randomness.

Set Algorithms

The STL provides a number of function templates for performing the basic set operations that you probably learned about in math class. Table H-13 summarizes these methods.

Table H-13 Set Functions

Function Template	Description
`set_union(iterator1, iterator2, iterator3, iterator4, iterator5)`	Finds the union of two sorted sets. The union of two sets is a set that contains all the elements of both sets, excluding duplicates. The `iterator1` and `iterator2` arguments mark the beginning and end of the first sorted set. The `iterator3` and `iterator4` arguments mark the beginning and end of the second sorted set. The `iterator5` argument marks the beginning of the container that will hold the union of the two sets. The function returns an iterator pointing to the end of the range of elements in the union.

(table continues)

Table H-13 *(continued)*

Function Template	Description
set_intersection (*iterator1*, *iterator2*, *iterator3*, *iterator4*, *iterator5*)	Finds the intersection of two sorted sets. The intersection of two sets is a set that contains only the elements that are found in both sets. The *iterator1* and *iterator2* arguments mark the beginning and end of the first sorted set. The *iterator3* and *iterator4* arguments mark the beginning and end of the second sorted set. The *iterator5* argument marks the beginning of the container that will hold the intersection of the two sets. The function returns an iterator pointing to the end of range of elements in the intersection.
set_difference(*iterator1*, *iterator2*, *iterator3*, *iterator4*, *iterator5*)	Finds the difference of two sorted sets. The difference of two sets is the set of elements that appear in one set, but not the other. The *iterator1* and *iterator2* arguments mark the beginning and end of the first sorted set. The *iterator3* and *iterator4* arguments mark the beginning and end of the second sorted set. The *iterator5* argument marks the beginning of the container that will hold the difference of the two sets. The function returns an iterator pointing to the end of the range of elements in the difference.
set_symmetric_difference (*iterator1*, *iterator2*, *iterator3*, *iterator4*, *iterator5*)	Finds the symmetric difference of two sorted sets. The symmetric difference of two sets is the set of elements that are in one set, but not in both. The *iterator1* and *iterator2* arguments mark the beginning and end of the first sorted set. The *iterator3* and *iterator4* arguments mark the beginning and end of the second sorted set. The *iterator5* argument marks the beginning of the container that will hold the symmetric difference of the two sets. The function returns an iterator pointing to the end of the range of elements in the symmetric difference.
includes(*iterator1*, *iterator2*, *iterator3*, *iterator4*)	Determines whether one set includes another set. The *iterator1* and *iterator2* arguments mark the beginning and end of the first sorted set. The *iterator3* and *iterator4* arguments mark the beginning and end of the second sorted set. The function returns true if the first set contains all of the elements of the second set. Otherwise, the function returns false.

Transformation Algorithms

The transformation algorithms iterate over ranges of elements, performing some programmer-defined operation on each element. Table H-14 lists these functions.

Table H-14 Range Modification Functions

Function Template	Description
`for_each(`*`iterator1`*`, `*`iterator2`*`, `*`function`*`)`	The *`iterator1`* and *`iterator2`* arguments mark the beginning and end of a range of elements. The *`function`* argument is the address of a function, or is a function object, that accepts an element as its argument. (The *`function`*'s return value, if it has any, is ignored.) The `for_each` function iterates over each element in the range, calling *`function`*, passing the element as an argument.
`transform(`*`iterator1`*`, `*`iterator2`*`, `*`iterator3`*`, `*`function`*`)`	The *`iterator1`* and *`iterator2`* arguments mark the beginning and end of the 1st range of elements. The *`iterator3`* argument marks the beginning of the 2nd range of elements, assumed to have the same number of elements as the 1st range. The *`function`* argument is the address of a function, or is a function object, that can accept an element of the 1st range as its argument, and returns a value of a data type that is compatible with the elements in the 2nd range. The `transform` function iterates over the elements of the 1st range, passing each element of the 1st range to *`function`*, and storing *`function`*'s return value in the 2nd range.
`transform(`*`iterator1`*`, `*`iterator2`*`, `*`iterator3`*`, `*`iterator4`*`, `*`function`*`)`	The *`iterator1`* and *`iterator2`* arguments mark the beginning and end of the 1st range of elements. The *`iterator3`* argument marks the beginning of the 2nd range of elements, assumed to have the same number of elements as the 1st range. The *`iterator4`* argument marks the beginning of the 3rd range of elements, also assumed to have the same number of elements as the 1st range. The *`function`* argument is the address of a function, or is a function object, that can accept two arguments: an element of the 1st range and an element of the 2nd range. *`function`* returns a value of a data type that is compatible with the elements in the 3rd range. The `transform` function iterates over the elements of the 1st and 2nd ranges, passing those elements to *`function`*, and storing *`function`*'s return value in the 3rd range.

Partition Algorithms

The partition algorithms partition a range of elements into two groups. Table H-15 lists these functions.

Table H-15 Partition Functions

Function Template	Description
`partition(`*`iterator1`*`, `*`iterator2`*`, `*`function`*`)`	The *`iterator1`* and *`iterator2`* arguments mark the beginning and end of a range of elements. The *`function`* argument is the address of a function, or is a function object, that accepts an element as its argument, and returns `true` or `false`. The `partition` function iterates over each element in the range, calling *`function`*, passing the element as an argument. After the `partition` function executes, all of the elements that caused *`function`* to return `true` will be stored before the elements that caused *`function`* to return `false`.
`stable_partition(`*`iterator1`*`, `*`iterator2`*`, `*`function`*`)`	The *`iterator1`* and *`iterator2`* arguments mark the beginning and end of a range of elements. The *`function`* argument is the address of a function, or is a function object, that accepts an element as its argument, and returns `true` or `false`. The `stable_partition` function iterates over each element in the range, calling *`function`*, passing the element as an argument. After the `stable_partition` function executes, all of the elements that caused *`function`* to return `true` will be stored before the elements that caused *`function`* to return `false`. Within the two “groups” of elements, the relative ordering of elements will be the same as before the partitioning occurred.
`partition_copy(`*`iterator1`*`, `*`iterator2`*`, `*`iterator3`*`, `*`iterator4`*`, `*`function`*`)`	The *`iterator1`* and *`iterator2`* arguments mark the beginning and end of the 1st range of elements. The *`iterator3`* argument marks the beginning of the 2nd range of elements, assumed to have the same number of elements as the 1st range. The *`iterator4`* argument marks the beginning of the 3rd range of elements, also assumed to have the same number of elements as the 1st range. The *`function`* argument is the address of a function, or is a function object, that accepts an element as its argument, and returns `true` or `false`. The `partition_copy` function iterates over the elements of the 1st range, passing those elements to *`function`*. Each element that caused *`function`* to return `true` is copied to the 2nd range, and each element that caused *`function`* to return `false` is copied to the 3rd range.

Table H-15 *(continued)*

Function Template	Description
`partition_point(`*`iterator1`*`, `*`iterator2`*`, `*`function`*`)`	The *`iterator1`* and *`iterator2`* arguments mark the beginning and end of a range of elements. The *`function`* argument is the address of a function, or is a function object, that accepts an element as its argument, and returns `true` or `false`. The `partition_point` function iterates over each element in the range, calling *`function`*, passing the element as an argument. The `partition_point` function returns an iterator pointing to the first element that caused *`function`* to return `false`. (This element is the partition point.)
`is_partitioned(`*`iterator1`*`, `*`iterator2`*`, `*`function`*`)`	The *`iterator1`* and *`iterator2`* arguments mark the beginning and end of a range of elements. The *`function`* argument is the address of a function, or is a function object, that accepts an element as its argument, and returns `true` or `false`. The `is_partitioned` function iterates over each element in the range, calling *`function`*, passing the element as an argument. The `is_partitioned` function returns `true` if the range is already portioned (if all of the elements that caused *`function`* to return `true` are stored before the elements that caused function to return `false`.)

Merge Algorithms

The merge algorithms merge two sorted ranges of elements to create a 3rd sorted range of elements. Table H-16 lists these functions.

Table H-16 Merge Functions

Function Template	Description
`merge(`*`iterator1`*`, `*`iterator2`*`, `*`iterator3`*`, `*`iterator4`*`, `*`iterator5`*`)`	The *`iterator1`* and *`iterator2`* arguments mark the beginning and end of the 1st sorted range of elements. The *`iterator3`* and *`iterator4`* arguments mark the beginning and end of the 2nd sorted range of elements. The *`iterator5`* argument marks the beginning of the 3rd range, assumed to have enough elements to hold both the 1st and 2nd ranges combined. The function combines the 1st and 2nd ranges, storing the result in the 3rd range. (This function uses the < operator to compare elements. If the elements contain your own class objects, be sure to overload that operator.)
`inplace_merge(`*`firstIt`*`, `*`middleIt`*`, `*`lastIt`*`)`	The *`firstIt`* and *`middleIt`* arguments mark the beginning and end of the 1st sorted range of elements. The *`middleIt`* and *`lastIt`* arguments mark the beginning and end of the 2nd sorted range of elements. The function combines the 1st and 2nd ranges, storing the results at *`firstIt`*. (This function uses the < operator to compare elements. If the elements contain your own class objects, be sure to overload that operator.)

Permutation Algorithms

If a range has *N* elements, there are *N*! possible arrangements of those elements. Each arrangement is called a *permutation*. The different permutations of a group of elements can be compared lexicographically to determine an order, from lowest to highest. The STL's permutation algorithms allow you to get the different possible permutations of a range of elements, and determine whether one range of elements is a permutation of another range of elements. Table H-17 lists these functions.

Table H-17 Permutation Functions

Function Template	Description
`next_permutation(iterator1, iterator2)`	The *iterator1* and *iterator2* arguments mark the beginning and end of a range of elements. The function rearranges the elements into their next permutation, according to their lexicographical order. (This function uses the < operator to compare elements. If the elements contain your own class objects, be sure to overload that operator.)
`prev_permutation(iterator1, iterator2)`	The *iterator1* and *iterator2* arguments mark the beginning and end of a range of elements. The function rearranges the elements into their previous permutation, according to their lexicographical order. (This function uses the < operator to compare elements. If the elements contain your own class objects, be sure to overload that operator.)
`is_permutation(iterator1, iterator2, iterator3)`	The *iterator1* and *iterator2* arguments mark the beginning and end of the 1st range of elements. The *iterator3* argument marks the beginning of the 2nd range of elements, assumed to have the same number of elements as the 1st range. The function returns `true` if the 2nd range is a permutation of the 1st range, or `false` otherwise. (This function uses the == operator to compare elements. If the elements contain your own class objects, be sure to overload that operator.)

Heap Algorithms

The heap algorithms perform operations that you typically need when working with a heap. A heap is a data structure in which the highest value is always stored on top. Table H-18 lists these functions.

Table H-18 Heap Functions

Function Template	Description
`make_heap(iterator1, iterator2)`	The *iterator1* and *iterator2* arguments mark the beginning and end of a range of elements. The function rearranges the elements so they become a heap, with the greatest element stored at the beginning of the range. (This function uses the < operator to compare elements. If the elements contain your own class objects, be sure to overload that operator.)

Table H-18 *(continued)*

Function Template	Description
`push_heap(iterator1, iterator2)`	The *iterator1* and *iterator2* arguments mark the beginning and end of a range of elements. Within that range, the elements from *iterator1* through *iterator2* – 1 are considered a heap. This function extends the heap to include the element at *iterator2*.
`pop_heap(iterator1, iterator2)`	The *iterator1* and *iterator2* arguments mark the beginning and end of a range of elements that are considered to be a heap. This function moves the element with the highest value to *iterator2* – 1 and shortens the heap by one element.
`sort_heap(iterator1, iterator2)`	The *iterator1* and *iterator2* arguments mark the beginning and end of a range of elements that are considered to be a heap. This function sorts the elements into ascending order. (This function uses the < operator to compare elements. If the elements contain your own class objects, be sure to overload that operator.)
`is_heap(iterator1, iterator2)`	The *iterator1* and *iterator2* arguments mark the beginning and end of a range of elements. This function returns `true` if the range is arranged in a way that can be considered a heap. (This function uses the < operator to compare elements. If the elements contain your own class objects, be sure to overload that operator.)
`is_heap_until(iterator1, iterator2)`	The *iterator1* and *iterator2* arguments mark the beginning and end of a range of elements. This function returns an iterator pointing to the first element that is not in the correct position for this range to be considered a heap. If all the elements are in valid positions, the function returns *iterator2*. (This function uses the < operator to compare elements. If the elements contain your own class objects, be sure to overload that operator.)

Lexicographical Comparison Algorithm

The words in the dictionary are sorted according to their lexicographic order. For example, "aardvark" is less than "android", and "pastry" is less than "pizza". The STL provides the function described in Table H-19 to determine whether one range of elements is lexicographically less than another range.

Table H-19 Lexicographical Comparison Function

Function Template	Description
`lexicographical_compare (iterator1, iterator2, iterator3, iterator4)`	The *iterator1* and *iterator2* arguments mark the beginning and end of the 1st range of elements. The *iterator3* and *iterator4* arguments mark the beginning and end of the 2nd range of elements. This function returns `true` if the 1st range is lexicographically less than the 2nd range, or `false` otherwise. (This function uses the < operator to compare elements. If the elements contain your own class objects, be sure to overload that operator.)