# Source:

## http://www.cplusplus.com/reference/algorithm/

# Standard Template Library: Algorithms

The header **<algorithm>** defines a collection of functions especially designed to be used on ranges of elements.

A range is any sequence of objects that can be accessed through **iterators or pointers**, such as an array or an instance of some of the [STL containers](http://www.cplusplus.com/stl).

Notice though, that algorithms operate through iterators directly on the values, not affecting in any way the structure of any possible container (it never affects the size or storage allocation of the container).

## Functions in <algorithm>

**Non-modifying sequence operations**:

[**all\_of**](http://www.cplusplus.com/reference/algorithm/all_of/)

Test condition on all elements in range (function template )

[**any\_of**](http://www.cplusplus.com/reference/algorithm/any_of/)

Test if any element in range fulfills condition (function template )

[**none\_of**](http://www.cplusplus.com/reference/algorithm/none_of/)

Test if no elements fulfill condition (function template )

[**for\_each**](http://www.cplusplus.com/reference/algorithm/for_each/)

Apply function to range (function template )

[**find**](http://www.cplusplus.com/reference/algorithm/find/)

Find value in range (function template )

[**find\_if**](http://www.cplusplus.com/reference/algorithm/find_if/)

Find element in range (function template )

[**find\_if\_not**](http://www.cplusplus.com/reference/algorithm/find_if_not/)

Find element in range (negative condition) (function template )

[**find\_end**](http://www.cplusplus.com/reference/algorithm/find_end/)

Find last subsequence in range (function template )

[**find\_first\_of**](http://www.cplusplus.com/reference/algorithm/find_first_of/)

Find element from set in range (function template )

[**adjacent\_find**](http://www.cplusplus.com/reference/algorithm/adjacent_find/)

Find equal adjacent elements in range (function template )

[**count**](http://www.cplusplus.com/reference/algorithm/count/)

Count appearances of value in range (function template )

[**count\_if**](http://www.cplusplus.com/reference/algorithm/count_if/)

Return number of elements in range satisfying condition (function template )

[**mismatch**](http://www.cplusplus.com/reference/algorithm/mismatch/)

Return first position where two ranges differ (function template )

[**equal**](http://www.cplusplus.com/reference/algorithm/equal/)

Test whether the elements in two ranges are equal (function template )

[**is\_permutation**](http://www.cplusplus.com/reference/algorithm/is_permutation/)

Test whether range is permutation of another (function template )

[**search**](http://www.cplusplus.com/reference/algorithm/search/)

Search range for subsequence (function template )

[**search\_n**](http://www.cplusplus.com/reference/algorithm/search_n/)

Search range for elements (function template )

**Modifying sequence operations**:

[**copy**](http://www.cplusplus.com/reference/algorithm/copy/)

Copy range of elements (function template )

[**copy\_n**](http://www.cplusplus.com/reference/algorithm/copy_n/)

Copy elements (function template )

[**copy\_if**](http://www.cplusplus.com/reference/algorithm/copy_if/)

Copy certain elements of range (function template )

[**copy\_backward**](http://www.cplusplus.com/reference/algorithm/copy_backward/)

Copy range of elements backward (function template )

[**move**](http://www.cplusplus.com/reference/algorithm/move/)

Move range of elements (function template )

[**move\_backward**](http://www.cplusplus.com/reference/algorithm/move_backward/)

Move range of elements backward (function template )

[**swap**](http://www.cplusplus.com/reference/algorithm/swap/)

Exchange values of two objects (function template )

[**swap\_ranges**](http://www.cplusplus.com/reference/algorithm/swap_ranges/)

Exchange values of two ranges (function template )

[**iter\_swap**](http://www.cplusplus.com/reference/algorithm/iter_swap/)

Exchange values of objects pointed to by two iterators (function template )

[**transform**](http://www.cplusplus.com/reference/algorithm/transform/)

Transform range (function template )

[**replace**](http://www.cplusplus.com/reference/algorithm/replace/)

Replace value in range (function template )

[**replace\_if**](http://www.cplusplus.com/reference/algorithm/replace_if/)

Replace values in range (function template )

[**replace\_copy**](http://www.cplusplus.com/reference/algorithm/replace_copy/)

Copy range replacing value (function template )

[**replace\_copy\_if**](http://www.cplusplus.com/reference/algorithm/replace_copy_if/)

Copy range replacing value (function template )

[**fill**](http://www.cplusplus.com/reference/algorithm/fill/)

Fill range with value (function template )

[**fill\_n**](http://www.cplusplus.com/reference/algorithm/fill_n/)

Fill sequence with value (function template )

[**generate**](http://www.cplusplus.com/reference/algorithm/generate/)

Generate values for range with function (function template )

[**generate\_n**](http://www.cplusplus.com/reference/algorithm/generate_n/)

Generate values for sequence with function (function template )

[**remove**](http://www.cplusplus.com/reference/algorithm/remove/)

Remove value from range (function template )

[**remove\_if**](http://www.cplusplus.com/reference/algorithm/remove_if/)

Remove elements from range (function template )

[**remove\_copy**](http://www.cplusplus.com/reference/algorithm/remove_copy/)

Copy range removing value (function template )

[**remove\_copy\_if**](http://www.cplusplus.com/reference/algorithm/remove_copy_if/)

Copy range removing values (function template )

[**unique**](http://www.cplusplus.com/reference/algorithm/unique/)

Remove consecutive duplicates in range (function template )

[**unique\_copy**](http://www.cplusplus.com/reference/algorithm/unique_copy/)

Copy range removing duplicates (function template )

[**reverse**](http://www.cplusplus.com/reference/algorithm/reverse/)

Reverse range (function template )

[**reverse\_copy**](http://www.cplusplus.com/reference/algorithm/reverse_copy/)

Copy range reversed (function template )

[**rotate**](http://www.cplusplus.com/reference/algorithm/rotate/)

Rotate left the elements in range (function template )

[**rotate\_copy**](http://www.cplusplus.com/reference/algorithm/rotate_copy/)

Copy range rotated left (function template )

[**random\_shuffle**](http://www.cplusplus.com/reference/algorithm/random_shuffle/)

Randomly rearrange elements in range (function template )

[**shuffle**](http://www.cplusplus.com/reference/algorithm/shuffle/)

Randomly rearrange elements in range using generator (function template )

**Partitions**:

[**is\_partitioned**](http://www.cplusplus.com/reference/algorithm/is_partitioned/)

Test whether range is partitioned (function template )

[**partition**](http://www.cplusplus.com/reference/algorithm/partition/)

Partition range in two (function template )

[**stable\_partition**](http://www.cplusplus.com/reference/algorithm/stable_partition/)

Partition range in two - stable ordering (function template )

[**partition\_copy**](http://www.cplusplus.com/reference/algorithm/partition_copy/)

Partition range into two (function template )

[**partition\_point**](http://www.cplusplus.com/reference/algorithm/partition_point/)

Get partition point (function template )

**Sorting**:

[**sort**](http://www.cplusplus.com/reference/algorithm/sort/)

Sort elements in range (function template )

[**stable\_sort**](http://www.cplusplus.com/reference/algorithm/stable_sort/)

Sort elements preserving order of equivalents (function template )

[**partial\_sort**](http://www.cplusplus.com/reference/algorithm/partial_sort/)

Partially sort elements in range (function template )

[**partial\_sort\_copy**](http://www.cplusplus.com/reference/algorithm/partial_sort_copy/)

Copy and partially sort range (function template )

[**is\_sorted**](http://www.cplusplus.com/reference/algorithm/is_sorted/)

Check whether range is sorted (function template )

[**is\_sorted\_until**](http://www.cplusplus.com/reference/algorithm/is_sorted_until/)

Find first unsorted element in range (function template )

[**nth\_element**](http://www.cplusplus.com/reference/algorithm/nth_element/)

Sort element in range (function template )

**Binary search** (operating on partitioned/sorted ranges):

[**lower\_bound**](http://www.cplusplus.com/reference/algorithm/lower_bound/)

Return iterator to lower bound (function template )

[**upper\_bound**](http://www.cplusplus.com/reference/algorithm/upper_bound/)

Return iterator to upper bound (function template )

[**equal\_range**](http://www.cplusplus.com/reference/algorithm/equal_range/)

Get subrange of equal elements (function template )

[**binary\_search**](http://www.cplusplus.com/reference/algorithm/binary_search/)

Test if value exists in sorted sequence (function template )

**Merge** (operating on sorted ranges):

[**merge**](http://www.cplusplus.com/reference/algorithm/merge/)

Merge sorted ranges (function template )

[**inplace\_merge**](http://www.cplusplus.com/reference/algorithm/inplace_merge/)

Merge consecutive sorted ranges (function template )

[**includes**](http://www.cplusplus.com/reference/algorithm/includes/)

Test whether sorted range includes another sorted range (function template )

[**set\_union**](http://www.cplusplus.com/reference/algorithm/set_union/)

Union of two sorted ranges (function template )

[**set\_intersection**](http://www.cplusplus.com/reference/algorithm/set_intersection/)

Intersection of two sorted ranges (function template )

[**set\_difference**](http://www.cplusplus.com/reference/algorithm/set_difference/)

Difference of two sorted ranges (function template )

[**set\_symmetric\_difference**](http://www.cplusplus.com/reference/algorithm/set_symmetric_difference/)

Symmetric difference of two sorted ranges (function template )

**Heap**:

[**push\_heap**](http://www.cplusplus.com/reference/algorithm/push_heap/)

Push element into heap range (function template )

[**pop\_heap**](http://www.cplusplus.com/reference/algorithm/pop_heap/)

Pop element from heap range (function template )

[**make\_heap**](http://www.cplusplus.com/reference/algorithm/make_heap/)

Make heap from range (function template )

[**sort\_heap**](http://www.cplusplus.com/reference/algorithm/sort_heap/)

Sort elements of heap (function template )

[**is\_heap**](http://www.cplusplus.com/reference/algorithm/is_heap/)

Test if range is heap (function template )

[**is\_heap\_until**](http://www.cplusplus.com/reference/algorithm/is_heap_until/)

Find first element not in heap order (function template )

**Min/max**:

[**min**](http://www.cplusplus.com/reference/algorithm/min/)

Return the smallest (function template )

[**max**](http://www.cplusplus.com/reference/algorithm/max/)

Return the largest (function template )

[**minmax**](http://www.cplusplus.com/reference/algorithm/minmax/)

Return smallest and largest elements (function template )

[**min\_element**](http://www.cplusplus.com/reference/algorithm/min_element/)

Return smallest element in range (function template )

[**max\_element**](http://www.cplusplus.com/reference/algorithm/max_element/)

Return largest element in range (function template )

[**minmax\_element**](http://www.cplusplus.com/reference/algorithm/minmax_element/)

Return smallest and largest elements in range (function template )

**Other**:

[**lexicographical\_compare**](http://www.cplusplus.com/reference/algorithm/lexicographical_compare/)

Lexicographical less-than comparison (function template )

[**next\_permutation**](http://www.cplusplus.com/reference/algorithm/next_permutation/)

Transform range to next permutation (function template )

[**prev\_permutation**](http://www.cplusplus.com/reference/algorithm/prev_permutation/)

Transform range to previous permutation (function template )

# for\_each

**Apply function to range**

Applies function *fn* to each of the elements in the range [first,last).  
  
The behavior of this template function is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | template<class InputIterator, class Function>  Function for\_each(InputIterator first, InputIterator last, Function fn)  {  while (first!=last) {  fn (\*first);  ++first;  }  return fn; // or, since C++11: return move(fn);  } |

**template <class InputIterator, class Function>**

**Function for\_each (InputIterator first, InputIterator last, Function fn);**

### Parameters

first, last

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions in a sequence. The range used is [first,last), which contains all the elements between first and last, including the element pointed by first but not the element pointed by last.

fn

Unary function that accepts an element in the range as argument.  
This can either be a function pointer or a [move constructible](http://www.cplusplus.com/is_move_constructible) function object.  
Its return value, if any, is ignored.

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 | // for\_each example  #include <iostream> // std::cout  #include <algorithm> // std::for\_each  #include <vector> // std::vector  void myfunction (int i) { // function:  std::cout << ' ' << i;  }  struct myclass { // function object type:  void operator() (int i) {std::cout << ' ' << i;}  } myobject;  int main () {  std::vector<int> myvector;  myvector.push\_back(10);  myvector.push\_back(20);  myvector.push\_back(30);  std::cout << "myvector contains:";  for\_each (myvector.begin(), myvector.end(), myfunction);  std::cout << '\n';  // or:  std::cout << "myvector contains:";  for\_each (myvector.begin(), myvector.end(), myobject);  std::cout << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/for_each/) |

Output:

|  |
| --- |
| myvector contains: 10 20 30  myvector contains: 10 20 30 |

# find

template <class InputIterator, class T>

InputIterator find (InputIterator first, InputIterator last, const T& val);

**Find value in range**

Returns an iterator to the first element in the range [first,last) that compares equal to *val*. If no such element is found, the function returns *last*.  
  
The function uses operator== to compare the individual elements to *val*.

### Parameters

first, last

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions in a sequence. The range searched is [first,last), which contains all the elements between first and last, including the element pointed by first but not the element pointed by last.

val

Value to search for in the range.  
T shall be a type supporting comparisons with the elements pointed by InputIterator using operator== (with the elements as left-hand side operands, and val as right-hand side).

### Return value

An iterator to the first element in the range that compares equal to val.  
If no elements match, the function returns last.

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 | // find example  #include <iostream> // std::cout  #include <algorithm> // std::find  #include <vector> // std::vector  int main () {  // using std::find with array and pointer:  int myints[] = { 10, 20, 30, 40 };  int \* p;  p = std::find (myints, myints+4, 30);  if (p != myints+4)  std::cout << "Element found in myints: " << \*p << '\n';  else  std::cout << "Element not found in myints\n";  // using std::find with vector and iterator:  std::vector<int> myvector (myints,myints+4);  std::vector<int>::iterator it;  it = find (myvector.begin(), myvector.end(), 30);  if (it != myvector.end())  std::cout << "Element found in myvector: " << \*it << '\n';  else  std::cout << "Element not found in myvector\n";  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/find/) |

Output:

|  |
| --- |
| Element found in myints: 30  Element found in myvector: 30 |

# search

**Search range for subsequence**

Searches the range [first1,last1) for the first occurrence of the sequence defined by [first2,last2), and returns an iterator to its first element, or last1 if no occurrences are found.  
  
The elements in both ranges are compared sequentially using operator== (or pred, in version *(2)*): A subsequence of [first1,last1) is considered a match only when this is true for **all** the elements of [first2,last2).  
  
This function returns the first of such occurrences. For an algorithm that returns the last instead, see [find\_end](http://www.cplusplus.com/find_end).

|  |  |
| --- | --- |
| *equality (1)* | template <class ForwardIterator1, class ForwardIterator2>  ForwardIterator1 search (ForwardIterator1 first1, ForwardIterator1 last1,  ForwardIterator2 first2, ForwardIterator2 last2); |
| *predicate (2)* | template <class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>  ForwardIterator1 search (ForwardIterator1 first1, ForwardIterator1 last1,  ForwardIterator2 first2, ForwardIterator2 last2,  BinaryPredicate pred); |

### Parameters

first1, last1

[Forward iterators](http://www.cplusplus.com/ForwardIterator) to the initial and final positions of the searched sequence. The range used is [first1,last1), which contains all the elements between first1 and last1, including the element pointed by first1 but not the element pointed by last1.

first2, last2

[Forward iterators](http://www.cplusplus.com/ForwardIterator) to the initial and final positions of the sequence to be searched for. The range used is [first2,last2).  
For *(1)*, the elements in both ranges shall be of types comparable using operator== (with the elements of the first range as left-hand side operands, and those of the second as right-hand side operands).

pred

Binary function that accepts two elements as arguments (one of each of the two sequences, in the same order), and returns a value convertible to bool. The returned value indicates whether the elements are considered to match in the context of this function.  
The function shall not modify any of its arguments.  
This can either be a function pointer or a function object.

### Return value

An iterator to the first element of the first occurrence of [first2,last2) in [first1,last1).  
If the sequence is not found, the function returns last1.

If [first2,last2) is an empty range, the function returns first1.

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 | // search algorithm example  #include <iostream> // std::cout  #include <algorithm> // std::search  #include <vector> // std::vector  bool mypredicate (int i, int j) {  return (i==j);  }  int main () {  std::vector<int> haystack;  // set some values: haystack: 10 20 30 40 50 60 70 80 90  for (int i=1; i<10; i++) haystack.push\_back(i\*10);  // using default comparison:  int needle1[] = {40,50,60,70};  std::vector<int>::iterator it;  it = std::search (haystack.begin(), haystack.end(), needle1, needle1+4);  if (it!=haystack.end())  std::cout << "needle1 found at position " << (it-haystack.begin()) << '\n';  else  std::cout << "needle1 not found\n";  // using predicate comparison:  int needle2[] = {20,30,50};  it = std::search (haystack.begin(), haystack.end(), needle2, needle2+3, mypredicate);  if (it!=haystack.end())  std::cout << "needle2 found at position " << (it-haystack.begin()) << '\n';  else  std::cout << "needle2 not found\n";  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/search/) |

Output:

|  |
| --- |
| needle1 found at position 3  needle2 not found |

# copy

**Copy range of elements**

Copies the elements in the range [first,last) into the range beginning at result.  
  
The function returns an iterator to the end of the destination range (which points to the element following the last element copied).  
  
The ranges shall not overlap in such a way that result points to an element in the range [first,last). For such cases, see [copy\_backward](http://www.cplusplus.com/copy_backward).

template <class InputIterator, class OutputIterator>

OutputIterator copy (InputIterator first, InputIterator last, OutputIterator result);

### Parameters

first, last

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions in a sequence to be copied. The range used is [first,last), which contains all the elements between first and last, including the element pointed by first but not the element pointed by last.

result

[Output iterator](http://www.cplusplus.com/OutputIterator) to the initial position in the destination sequence.  
This shall not point to any element in the range [first,last).

### Return value

An iterator to the end of the destination range where elements have been copied.

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | // copy algorithm example  #include <iostream> // std::cout  #include <algorithm> // std::copy  #include <vector> // std::vector  int main () {  int myints[]={10,20,30,40,50,60,70};  std::vector<int> myvector (7);  std::copy ( myints, myints+7, myvector.begin() );  std::cout << "myvector contains:";  for (std::vector<int>::iterator it = myvector.begin(); it!=myvector.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/copy/) |

Output:

|  |
| --- |
| myvector contains: 10 20 30 40 50 60 70 |

# swap

**Exchange values of two objects**

Exchanges the values of *a* and *b*.

|  |  |
| --- | --- |
| *header* | // moved from <algorithm> to <utility> in C++11 |
| *non-array (1)* | template <class T> void swap (T& a, T& b)  noexcept (is\_nothrow\_move\_constructible<T>::value && is\_nothrow\_move\_assignable<T>::value); |
| *array (2)* | template <class T, size\_t N> void swap(T (&a)[N], T (&b)[N])  noexcept (noexcept(swap(\*a,\*b))); |

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | // swap algorithm example (C++98)  #include <iostream> // std::cout  #include <algorithm> // std::swap  #include <vector> // std::vector  int main () {  int x=10, y=20; // x:10 y:20  std::swap(x,y); // x:20 y:10  std::vector<int> foo (4,x), bar (6,y); // foo:4x20 bar:6x10  std::swap(foo,bar); // foo:6x10 bar:4x20  std::cout << "foo contains:";  for (std::vector<int>::iterator it=foo.begin(); it!=foo.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/swap/) |

Output:

|  |
| --- |
| foo contains: 10 10 10 10 10 10 |

# reverse

**Reverse range**

Reverses the order of the elements in the range [first,last).  
  
The function calls [iter\_swap](http://www.cplusplus.com/iter_swap) to swap the elements to their new locations.

template <class BidirectionalIterator>

void reverse (BidirectionalIterator first, BidirectionalIterator last);

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 | // reverse algorithm example  #include <iostream> // std::cout  #include <algorithm> // std::reverse  #include <vector> // std::vector  int main () {  std::vector<int> myvector;  // set some values:  for (int i=1; i<10; ++i) myvector.push\_back(i); // 1 2 3 4 5 6 7 8 9  std::reverse(myvector.begin(),myvector.end()); // 9 8 7 6 5 4 3 2 1  // print out content:  std::cout << "myvector contains:";  for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/reverse/) |

Output:

|  |
| --- |
| myvector contains: 9 8 7 6 5 4 3 2 1 |

# replace

**Replace value in range**

Assigns *new\_value* to all the elements in the range [first,last) that compare equal to *old\_value*.  
  
The function uses operator== to compare the individual elements to *old\_value*.

**template <class ForwardIterator, class T>**

**void replace (ForwardIterator first, ForwardIterator last,**

**const T& old\_value, const T& new\_value);**

### Parameters

first, last

[Forward iterators](http://www.cplusplus.com/ForwardIterator) to the initial and final positions in a sequence of elements that support being compared and assigned a value of type T. The range used is [first,last), which contains all the elements between first and last, including the element pointed by first but not the element pointed by last.

old\_value

Value to be replaced.

new\_value

Replacement value.

### Return value

none

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | // replace algorithm example  #include <iostream> // std::cout  #include <algorithm> // std::replace  #include <vector> // std::vector  int main () {  int myints[] = { 10, 20, 30, 30, 20, 10, 10, 20 };  std::vector<int> myvector (myints, myints+8); // 10 20 30 30 20 10 10 20  std::replace (myvector.begin(), myvector.end(), 20, 99); // 10 99 30 30 99 10 10 99  std::cout << "myvector contains:";  for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/replace/) |

Output:

|  |
| --- |
| myvector contains: 10 99 30 30 99 10 10 99 |

# generate

**Generate values for range with function**

Assigns the value returned by successive calls to *gen* to the elements in the range [first,last).

**template <class ForwardIterator, class Generator>**

**void generate (ForwardIterator first, ForwardIterator last, Generator gen);**

### Parameters

first, last

[Forward iterators](http://www.cplusplus.com/ForwardIterator) to the initial and final positions in a sequence. The range affected is [first,last), which contains all the elements between first and last, including the element pointed by first but not the element pointed by last.

gen

Generator function that is called with no arguments and returns some value of a type convertible to those pointed by the iterators.  
This can either be a function pointer or a function object.

### Return value

None

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 | // generate algorithm example  #include <iostream> // std::cout  #include <algorithm> // std::generate  #include <vector> // std::vector  #include <ctime> // std::time  #include <cstdlib> // std::rand, std::srand  // function generator:  int RandomNumber () { return (std::rand()%100); }  // class generator:  struct c\_unique {  int current;  c\_unique() {current=0;}  int operator()() {return ++current;}  } UniqueNumber;  int main () {  std::srand ( unsigned ( std::time(0) ) );  std::vector<int> myvector (8);  std::generate (myvector.begin(), myvector.end(), RandomNumber);  std::cout << "myvector contains:";  for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  std::generate (myvector.begin(), myvector.end(), UniqueNumber);  std::cout << "myvector contains:";  for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';    return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/generate/) |

A possible output:

|  |
| --- |
| myvector contains: 57 87 76 66 85 54 17 15  myvector contains: 1 2 3 4 5 6 7 8 |

# unique

**Remove consecutive duplicates in range**

Removes all but the first element from every consecutive group of equivalent elements in the range [first,last).

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 | // unique algorithm example  #include <iostream> // std::cout  #include <algorithm> // std::unique, std::distance  #include <vector> // std::vector  bool myfunction (int i, int j) {  return (i==j);  }  int main () {  int myints[] = {10,20,20,20,30,30,20,20,10}; // 10 20 20 20 30 30 20 20 10  std::vector<int> myvector (myints,myints+9);  // using default comparison:  std::vector<int>::iterator it;  it = std::unique (myvector.begin(), myvector.end()); // 10 20 30 20 10 ? ? ? ?  // ^  myvector.resize( std::distance(myvector.begin(),it) ); // 10 20 30 20 10  // using predicate comparison:  std::unique (myvector.begin(), myvector.end(), myfunction); // (no changes)  // print out content:  std::cout << "myvector contains:";  for (it=myvector.begin(); it!=myvector.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/unique/) |

Output:

|  |
| --- |
| myvector contains: 10 20 30 20 10 |

# rotate

**Rotate left the elements in range**

Rotates the order of the elements in the range [first,last), in such a way that the element pointed by *middle*becomes the new first element.

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 | // rotate algorithm example  #include <iostream> // std::cout  #include <algorithm> // std::rotate  #include <vector> // std::vector  int main () {  std::vector<int> myvector;  // set some values:  for (int i=1; i<10; ++i) myvector.push\_back(i); // 1 2 3 4 5 6 7 8 9  std::rotate(myvector.begin(),myvector.begin()+3,myvector.end());  // 4 5 6 7 8 9 1 2 3  // print out content:  std::cout << "myvector contains:";  for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/rotate/) |

Output:

|  |
| --- |
| myvector contains: 4 5 6 7 8 9 1 2 3 |

# partition

**Partition range in two**

Rearranges the elements from the range [first,last), in such a way that all the elements for which pred returns trueprecede all those for which it returns false. The iterator returned points to the first element of the second group.  
  
The relative ordering within each group is not necessarily the same as before the call. See [stable\_partition](http://www.cplusplus.com/stable_partition) for a function with a similar behavior but with stable ordering within each group.

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 | // partition algorithm example  #include <iostream> // std::cout  #include <algorithm> // std::partition  #include <vector> // std::vector  bool IsOdd (int i) { return (i%2)==1; }  int main () {  std::vector<int> myvector;  // set some values:  for (int i=1; i<10; ++i) myvector.push\_back(i); // 1 2 3 4 5 6 7 8 9  std::vector<int>::iterator bound;  bound = std::partition (myvector.begin(), myvector.end(), IsOdd);  // print out content:  std::cout << "odd elements:";  for (std::vector<int>::iterator it=myvector.begin(); it!=bound; ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  std::cout << "even elements:";  for (std::vector<int>::iterator it=bound; it!=myvector.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/partition/) |

Possible output:

|  |
| --- |
| odd elements: 1 9 3 7 5  even elements: 6 4 8 2 |

# sort

**Sort elements in range**

Sorts the elements in the range [first,last) into ascending order.  
  
The elements are compared using operator< for the first version, and *comp* for the second.

Equivalent elements are not guaranteed to keep their original relative order (see [stable\_sort](http://www.cplusplus.com/stable_sort)).

|  |  |
| --- | --- |
| *default (1)* | template <class RandomAccessIterator>  void sort (RandomAccessIterator first, RandomAccessIterator last); |
| *custom (2)* | template <class RandomAccessIterator, class Compare>  void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp); |

### Parameters

first, last

[Random-access iterators](http://www.cplusplus.com/RandomAccessIterator) to the initial and final positions of the sequence to be sorted. The range used is [first,last), which contains all the elements between first and last, including the element pointed by first but not the element pointed by last.  
RandomAccessIterator shall point to a type for which [swap](http://www.cplusplus.com/swap) is properly defined and which is both [*move-constructible*](http://www.cplusplus.com/is_move_constructible) and [*move-assignable*](http://www.cplusplus.com/is_move_assignable).

comp

Binary function that accepts two elements in the range as arguments, and returns a value convertible to bool. The value returned indicates whether the element passed as first argument is considered to go before the second in the specific *strict weak ordering* it defines.  
The function shall not modify any of its arguments.  
This can either be a function pointer or a function object.

### Return value

none

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 | // sort algorithm example  #include <iostream> // std::cout  #include <algorithm> // std::sort  #include <vector> // std::vector  bool myfunction (int i,int j) { return (i<j); }  struct myclass {  bool operator() (int i,int j) { return (i<j);}  } myobject;  int main () {  int myints[] = {32,71,12,45,26,80,53,33};  std::vector<int> myvector (myints, myints+8); // 32 71 12 45 26 80 53 33  // using default comparison (operator <):  std::sort (myvector.begin(), myvector.begin()+4); //(12 32 45 71)26 80 53 33  // using function as comp  std::sort (myvector.begin()+4, myvector.end(), myfunction); // 12 32 45 71(26 33 53 80)  // using object as comp  std::sort (myvector.begin(), myvector.end(), myobject); //(12 26 32 33 45 53 71 80)  // print out content:  std::cout << "myvector contains:";  for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/sort/) |

Output:

|  |
| --- |
| myvector contains: 12 26 32 33 45 53 71 80 |

# stable\_sort

**Sort elements preserving order of equivalents**

Sorts the elements in the range [first,last) into ascending order, like [sort](http://www.cplusplus.com/sort), but [stable\_sort](http://www.cplusplus.com/stable_sort) preserves the relative order of the elements with equivalent values.  
  
The elements are compared using operator< for the first version, and comp for the second.

template <class RandomAccessIterator>

void stable\_sort ( RandomAccessIterator first, RandomAccessIterator last );

template <class RandomAccessIterator, class Compare>

void stable\_sort ( RandomAccessIterator first, RandomAccessIterator last,

Compare comp );

### Parameters

first, last

[Random-access iterators](http://www.cplusplus.com/RandomAccessIterator) to the initial and final positions of the sequence to be sorted. The range used is [first,last), which contains all the elements between first and last, including the element pointed by first but not the element pointed by last.  
RandomAccessIterator shall point to a type for which [swap](http://www.cplusplus.com/swap) is properly defined and which is both [*move-constructible*](http://www.cplusplus.com/is_move_constructible) and [*move-assignable*](http://www.cplusplus.com/is_move_assignable).

comp

Binary function that accepts two elements in the range as arguments, and returns a value convertible to bool. The value returned indicates whether the element passed as first argument is considered to go before the second in the specific *strict weak ordering* it defines.  
The function shall not modify any of its arguments.  
This can either be a function pointer or a function object.

### Return value

none

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 | // stable\_sort example  #include <iostream> // std::cout  #include <algorithm> // std::stable\_sort  #include <vector> // std::vector  bool compare\_as\_ints (double i,double j)  {  return (int(i)<int(j));  }  int main () {  double mydoubles[] = {3.14, 1.41, 2.72, 4.67, 1.73, 1.32, 1.62, 2.58};  std::vector<double> myvector;  myvector.assign(mydoubles,mydoubles+8);  std::cout << "using default comparison:";  std::stable\_sort (myvector.begin(), myvector.end());  for (std::vector<double>::iterator it=myvector.begin(); it!=myvector.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  myvector.assign(mydoubles,mydoubles+8);  std::cout << "using 'compare\_as\_ints' :";  std::stable\_sort (myvector.begin(), myvector.end(), compare\_as\_ints);  for (std::vector<double>::iterator it=myvector.begin(); it!=myvector.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/stable_sort/) |

compare\_as\_ints is a function that compares only the integral part of the elements, therefore, elements with the same integral part are considered equivalent. stable\_sort preserves the relative order these had before the call.  
  
Possible output:

|  |
| --- |
| using default comparison: 1.32 1.41 1.62 1.73 2.58 2.72 3.14 4.67  using 'compare\_as\_ints' : 1.41 1.73 1.32 1.62 2.72 2.58 3.14 4.67 |

# is\_sorted (C++11)

**Check whether range is sorted**

Returns true if the range [first,last) is sorted into ascending order.  
  
The elements are compared using operator< for the first version, and *comp* for the second.

|  |  |
| --- | --- |
| *default (1)* | template <class ForwardIterator>  bool is\_sorted (ForwardIterator first, ForwardIterator last); |
| *custom (2)* | template <class ForwardIterator, class Compare>  bool is\_sorted (ForwardIterator first, ForwardIterator last, Compare comp); |

### Parameters

first, last

[Forward iterators](http://www.cplusplus.com/ForwardIterator) to the initial and final positions of the sequence. The range checked is [first,last), which contains all the elements between first and last, including the element pointed by first but not the element pointed by last.

comp

Binary function that accepts two elements in the range as arguments, and returns a value convertible to bool. The value returned indicates whether the element passed as first argument is considered to go before the second in the specific *strict weak ordering* it defines.  
The function shall not modify any of its arguments.  
This can either be a function pointer or a function object.

### Return value

true if the range [first,last) is sorted into ascending order, false otherwise.  
  
If the range [first,last) contains less than two elements, the function always returns true.

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 | // is\_sorted example  #include <iostream> // std::cout  #include <algorithm> // std::is\_sorted, std::prev\_permutation  #include <array> // std::array  int main () {  std::array<int,4> foo {2,4,1,3};  do {  // try a new permutation:  std::prev\_permutation(foo.begin(),foo.end());  // print range:  std::cout << "foo:";  for (int& x:foo) std::cout << ' ' << x;  std::cout << '\n';  } while (!std::is\_sorted(foo.begin(),foo.end()));  std::cout << "the range is sorted!\n";  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/is_sorted/) |

Output:

|  |
| --- |
| foo: 2 3 4 1  foo: 2 3 1 4  foo: 2 1 4 3  foo: 2 1 3 4  foo: 1 4 3 2  foo: 1 4 2 3  foo: 1 3 4 2  foo: 1 3 2 4  foo: 1 2 4 3  foo: 1 2 3 4  the range is sorted! |

# min

**Return the smallest**

Returns the smallest of a and b. If both are equivalent, a is returned.  
  
The versions for *initializer lists (3)* return the smallest of all the elements in the list. Returning the first of them if these are more than one.  
  
The function uses operator< (or comp, if provided) to compare the values.

|  |  |
| --- | --- |
| *default (1)* | template <class T> const T& min (const T& a, const T& b); |
| *custom (2)* | template <class T, class Compare>  const T& min (const T& a, const T& b, Compare comp); |
| *initializer list (3)* | template <class T> T min (initializer\_list<T> il);  template <class T, class Compare>  T min (initializer\_list<T> il, Compare comp); |

### Parameters

a, b

Values to compare.

comp

Binary function that accepts two values of type T as arguments, and returns a value convertible to bool. The value returned indicates whether the element passed as first argument is considered less than the second.  
The function shall not modify any of its arguments.  
This can either be a function pointer or a function object.

il

An [initializer\_list](http://www.cplusplus.com/initializer_list) object.  
These objects are automatically constructed from *initializer list* declarators.

T shall support being compared with operator<.

### Return value

The lesser of the values passed as arguments.

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | // min example  #include <iostream> // std::cout  #include <algorithm> // std::min  int main () {  std::cout << "min(1,2)==" << std::min(1,2) << '\n';  std::cout << "min(2,1)==" << std::min(2,1) << '\n';  std::cout << "min('a','z')==" << std::min('a','z') << '\n';  std::cout << "min(3.14,2.72)==" << std::min(3.14,2.72) << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/min/) |

Output:

|  |
| --- |
| min(1,2)==1  min(2,1)==1  min('a','z')==a  min(3.14,2.72)==2.72 |

# max

**Return the largest**

Returns the largest of *a* and *b*. If both are equivalent, *a* is returned.  
  
The versions for *initializer lists (3)* return the largest of all the elements in the list. Returning the first of them if these are more than one.  
  
The function uses operator< (or *comp*, if provided) to compare the values.

|  |  |
| --- | --- |
| *default (1)* | template <class T> const T& max (const T& a, const T& b); |
| *custom (2)* | template <class T, class Compare>  const T& max (const T& a, const T& b, Compare comp); |
| *initializer list (3)* | template <class T> T max (initializer\_list<T> il);  template <class T, class Compare>  T max (initializer\_list<T> il, Compare comp); |

### Parameters

a, b

Values to compare.

comp

Binary function that accepts two values of type T as arguments, and returns a value convertible to bool. The value returned indicates whether the element passed as first argument is considered less than the second.  
The function shall not modify any of its arguments.  
This can either be a function pointer or a function object.

il

An [initializer\_list](http://www.cplusplus.com/initializer_list) object.  
These objects are automatically constructed from *initializer list* declarators.

T shall support being compared with operator<.

### Return value

The largest of the values passed as arguments.

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | // max example  #include <iostream> // std::cout  #include <algorithm> // std::max  int main () {  std::cout << "max(1,2)==" << std::max(1,2) << '\n';  std::cout << "max(2,1)==" << std::max(2,1) << '\n';  std::cout << "max('a','z')==" << std::max('a','z') << '\n';  std::cout << "max(3.14,2.73)==" << std::max(3.14,2.73) << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/max/) |

Output:

|  |
| --- |
| max(1,2)==2  max(2,1)==2  max('a','z')==z  max(3.14,2.73)==3.14 |

# minmax

**Return smallest and largest elements**

Returns a [pair](http://www.cplusplus.com/pair) with the smallest of a and b as first element, and the largest as second. If both are equivalent, the function returns [make\_pair](http://www.cplusplus.com/make_pair)(a,b).

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | // minmax example  #include <iostream> // std::cout  #include <algorithm> // std::minmax  int main () {  auto result = std::minmax({1,2,3,4,5});  std::cout << "minmax({1,2,3,4,5}): ";  std::cout << result.first << ' ' << result.second << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/minmax/) |

Output:

|  |
| --- |
| minmax({1,2,3,4,5}): 1 5 |

# min\_element

**Return smallest element in range**

Returns an iterator pointing to the element with the smallest value in the range [first,last).  
  
The comparisons are performed using either operator< for the first version, or *comp* for the second; An element is the smallest if no other element compares less than it. If more than one element fulfills this condition, the iterator returned points to the first of such elements.

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 | // min\_element/max\_element example  #include <iostream> // std::cout  #include <algorithm> // std::min\_element, std::max\_element  bool myfn(int i, int j) { return i<j; }  struct myclass {  bool operator() (int i,int j) { return i<j; }  } myobj;  int main () {  int myints[] = {3,7,2,5,6,4,9};  // using default comparison:  std::cout << "The smallest element is " << \*std::min\_element(myints,myints+7) << '\n';  std::cout << "The largest element is " << \*std::max\_element(myints,myints+7) << '\n';  // using function myfn as comp:  std::cout << "The smallest element is " << \*std::min\_element(myints,myints+7,myfn) << '\n';  std::cout << "The largest element is " << \*std::max\_element(myints,myints+7,myfn) << '\n';  // using object myobj as comp:  std::cout << "The smallest element is " << \*std::min\_element(myints,myints+7,myobj) << '\n';  std::cout << "The largest element is " << \*std::max\_element(myints,myints+7,myobj) << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/min_element/) |

Output:

|  |
| --- |
| The smallest element is 2  The largest element is 9  The smallest element is 2  The largest element is 9  The smallest element is 2  The largest element is 9 |

# max\_element

**Return largest element in range**

Returns an iterator pointing to the element with the largest value in the range [first,last).  
  
The comparisons are performed using either operator< for the first version, or *comp* for the second; An element is the largest if no other element does not compare less than it. If more than one element fulfills this condition, the iterator returned points to the first of such elements.

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 | // min\_element/max\_element example  #include <iostream> // std::cout  #include <algorithm> // std::min\_element, std::max\_element  bool myfn(int i, int j) { return i<j; }  struct myclass {  bool operator() (int i,int j) { return i<j; }  } myobj;  int main () {  int myints[] = {3,7,2,5,6,4,9};  // using default comparison:  std::cout << "The smallest element is " << \*std::min\_element(myints,myints+7) << '\n';  std::cout << "The largest element is " << \*std::max\_element(myints,myints+7) << '\n';  // using function myfn as comp:  std::cout << "The smallest element is " << \*std::min\_element(myints,myints+7,myfn) << '\n';  std::cout << "The largest element is " << \*std::max\_element(myints,myints+7,myfn) << '\n';  // using object myobj as comp:  std::cout << "The smallest element is " << \*std::min\_element(myints,myints+7,myobj) << '\n';  std::cout << "The largest element is " << \*std::max\_element(myints,myints+7,myobj) << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/max_element/) |

Output:

|  |
| --- |
| The smallest element is 2  The largest element is 9  The smallest element is 2  The largest element is 9  The smallest element is 2  The largest element is 9 |

# minmax\_element

**Return smallest and largest elements in range**

Returns a [pair](http://www.cplusplus.com/pair) with an iterator pointing to the element with the smallest value in the range [first,last) as first element, and the largest as second.  
  
The comparisons are performed using either operator< for the first version, or comp for the second.  
  
If more than one equivalent element has the smallest value, the first iterator points to the first of such elements.  
  
If more than one equivalent element has the largest value, the second iterator points to the last of such elements.

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | // minmax\_element  #include <iostream> // std::cout  #include <algorithm> // std::minmax\_element  #include <array> // std::array  int main () {  std::array<int,7> foo {3,7,2,9,5,8,6};  auto result = std::minmax\_element (foo.begin(),foo.end());  // print result:  std::cout << "min is " << \*result.first;  std::cout << ", at position " << (result.first-foo.begin()) << '\n';  std::cout << "max is " << \*result.second;  std::cout << ", at position " << (result.second-foo.begin()) << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/minmax_element/) |

Output:

|  |
| --- |
| min is 2, at position 2  max is 9, at position 3 |

# next\_permutation

**Transform range to next permutation**

Rearranges the elements in the range [first,last) into the next  [*lexicographically*](http://www.cplusplus.com/lexicographical_compare)*greater* permutation.

The comparisons of individual elements are performed using either operator< for the first version, or comp for the second.

If the function can determine the next higher permutation, it rearranges the elements as such and returns true. If that was not possible (because it is already at the largest possible permutation), it rearranges the elements according to the first permutation (sorted in ascending order) and returns false.

|  |  |
| --- | --- |
| *default (1)* | template <class BidirectionalIterator>  bool next\_permutation (BidirectionalIterator first,  BidirectionalIterator last); |
| *custom (2)* | template <class BidirectionalIterator, class Compare>  bool next\_permutation (BidirectionalIterator first,  BidirectionalIterator last, Compare comp); |

### Parameters

first, last

[Bidirectional iterators](http://www.cplusplus.com/BidirectionalIterator) to the initial and final positions of the sequence. The range used is [first,last), which contains all the elements between first and last, including the element pointed by first but not the element pointed by last.  
BidirectionalIterator shall point to a type for which [swap](http://www.cplusplus.com/swap) is properly defined.

comp

Binary function that accepts two arguments of the type pointed by BidirectionalIterator, and returns a value convertible to bool. The value returned indicates whether the first argument is considered to go before the second in the specific *strict weak ordering* it defines.  
The function shall not modify any of its arguments.  
This can either be a function pointer or a function object.

### Return value

true if the function could rearrange the object as a lexicographicaly greater permutation.  
Otherwise, the function returns false to indicate that the arrangement is not greater than the previous, but the lowest possible (sorted in ascending order).

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | // next\_permutation example  #include <iostream> // std::cout  #include <algorithm> // std::next\_permutation, std::sort  int main () {  int myints[] = {1,2,3};  std::sort (myints,myints+3);  std::cout << "The 3! possible permutations with 3 elements:\n";  do {  std::cout << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';  } while ( std::next\_permutation(myints,myints+3) );  std::cout << "After loop: " << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/next_permutation/) |

Output:

|  |
| --- |
| The 3! possible permutations with 3 elements:  1 2 3  1 3 2  2 1 3  2 3 1  3 1 2  3 2 1  After loop: 1 2 3 |

# prev\_permutation

**Transform range to previous permutation**

Rearranges the elements in the range [first,last) into the previous [*lexicographically*](http://www.cplusplus.com/lexicographical_compare)*-ordered* permutation.

### The comparisons of individual elements are performed using either operator< for the first version, or comp for the second. If the function can determine the previous permutation, it rearranges the elements as such and returns true. If that was not possible (because it is already at the lowest possible permutation), it rearranges the elements according to the last permutation (sorted in descending order) and returns false. Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | // next\_permutation example  #include <iostream> // std::cout  #include <algorithm> // std::next\_permutation, std::sort, std::reverse  int main () {  int myints[] = {1,2,3};  std::sort (myints,myints+3);  std::reverse (myints,myints+3);  std::cout << "The 3! possible permutations with 3 elements:\n";  do {  std::cout << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';  } while ( std::prev\_permutation(myints,myints+3) );  std::cout << "After loop: " << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/prev_permutation/) |

Output:

|  |
| --- |
| 3 2 1  3 1 2  2 3 1  2 1 3  1 3 2  1 2 3  After loop: 3 2 1 |

# set\_union

**Union of two sorted ranges**

Constructs a sorted range beginning in the location pointed by *result* with the *set union* of the two sorted ranges [first1,last1) and [first2,last2).  
  
The *union* of two sets is formed by the elements that are present in either one of the sets, or in both. Elements from the second range that have an equivalent element in the first range are not copied to the resulting range.

|  |  |
| --- | --- |
| *default (1)* | template <class InputIterator1, class InputIterator2, class OutputIterator>  OutputIterator set\_union (InputIterator1 first1, InputIterator1 last1,  InputIterator2 first2, InputIterator2 last2,  OutputIterator result); |
| *custom (2)* | template <class InputIterator1, class InputIterator2,  class OutputIterator, class Compare>  OutputIterator set\_union (InputIterator1 first1, InputIterator1 last1,  InputIterator2 first2, InputIterator2 last2,  OutputIterator result, Compare comp); |

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 | // set\_union example  #include <iostream> // std::cout  #include <algorithm> // std::set\_union, std::sort  #include <vector> // std::vector  int main () {  int first[] = {5,10,15,20,25};  int second[] = {50,40,30,20,10};  std::vector<int> v(10); // 0 0 0 0 0 0 0 0 0 0  std::vector<int>::iterator it;  std::sort (first,first+5); // 5 10 15 20 25  std::sort (second,second+5); // 10 20 30 40 50  it=std::set\_union (first, first+5, second, second+5, v.begin());  // 5 10 15 20 25 30 40 50 0 0  v.resize(it-v.begin()); // 5 10 15 20 25 30 40 50  std::cout << "The union has " << (v.size()) << " elements:\n";  for (it=v.begin(); it!=v.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/set_union/) |

Output:

|  |
| --- |
| The union has 8 elements:  5 10 15 20 25 30 40 50 |

# set\_intersection

**Intersection of two sorted ranges**

Constructs a sorted range beginning in the location pointed by *result* with the *set intersection* of the two sorted ranges [first1,last1) and [first2,last2).  
  
The *intersection* of two sets is formed only by the elements that are present in both sets. The elements copied by the function come always from the first range, in the same order.

|  |  |
| --- | --- |
| *default (1)* | template <class InputIterator1, class InputIterator2, class OutputIterator>  OutputIterator set\_intersection (InputIterator1 first1, InputIterator1 last1,  InputIterator2 first2, InputIterator2 last2,  OutputIterator result); |
| *custom (2)* | template <class InputIterator1, class InputIterator2,  class OutputIterator, class Compare>  OutputIterator set\_intersection (InputIterator1 first1, InputIterator1 last1,  InputIterator2 first2, InputIterator2 last2,  OutputIterator result, Compare comp); |

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 | // set\_intersection example  #include <iostream> // std::cout  #include <algorithm> // std::set\_intersection, std::sort  #include <vector> // std::vector  int main () {  int first[] = {5,10,15,20,25};  int second[] = {50,40,30,20,10};  std::vector<int> v(10); // 0 0 0 0 0 0 0 0 0 0  std::vector<int>::iterator it;  std::sort (first,first+5); // 5 10 15 20 25  std::sort (second,second+5); // 10 20 30 40 50  it=std::set\_intersection (first, first+5, second, second+5, v.begin());  // 10 20 0 0 0 0 0 0 0 0  v.resize(it-v.begin()); // 10 20  std::cout << "The intersection has " << (v.size()) << " elements:\n";  for (it=v.begin(); it!=v.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/set_intersection/) |

Output:

|  |
| --- |
| The intersection has 2 elements:  10 20 |

# set\_difference

**Difference of two sorted ranges**

Constructs a sorted range beginning in the location pointed by result with the *set difference* of the sorted range [first1,last1) with respect to the sorted range [first2,last2).  
  
The difference of two sets is formed by the elements that are present in the first set, but not in the second one. The elements copied by the function come always from the first range, in the same order.  
  
For containers supporting multiple occurrences of a value, the *difference* includes as many occurrences of a given value as in the first range, minus the amount of matching elements in the second, preserving order.

|  |  |
| --- | --- |
| *default (1)* | template <class InputIterator1, class InputIterator2, class OutputIterator>  OutputIterator set\_difference (InputIterator1 first1, InputIterator1 last1,  InputIterator2 first2, InputIterator2 last2,  OutputIterator result); |
| *custom (2)* | template <class InputIterator1, class InputIterator2,  class OutputIterator, class Compare>  OutputIterator set\_difference (InputIterator1 first1, InputIterator1 last1,  InputIterator2 first2, InputIterator2 last2,  OutputIterator result, Compare comp); |

Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 | // set\_difference example  #include <iostream> // std::cout  #include <algorithm> // std::set\_difference, std::sort  #include <vector> // std::vector  int main () {  int first[] = {5,10,15,20,25};  int second[] = {50,40,30,20,10};  std::vector<int> v(10); // 0 0 0 0 0 0 0 0 0 0  std::vector<int>::iterator it;  std::sort (first,first+5); // 5 10 15 20 25  std::sort (second,second+5); // 10 20 30 40 50  it=std::set\_difference (first, first+5, second, second+5, v.begin());  // 5 15 25 0 0 0 0 0 0 0  v.resize(it-v.begin()); // 5 15 25  std::cout << "The difference has " << (v.size()) << " elements:\n";  for (it=v.begin(); it!=v.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/set_difference/) |

Output:

|  |
| --- |
| The difference has 3 elements:  5 15 25 |

# set\_symmetric\_difference

**Symmetric difference of two sorted ranges**

Constructs a sorted range beginning in the location pointed by *result* with the *set symmetric difference* of the two sorted ranges [first1,last1) and [first2,last2).  
  
The *symmetric difference* of two sets is formed by the elements that are present in one of the sets, but not in the other. Among the equivalent elements in each range, those discarded are those that appear before in the existent order before the call. The existing order is also preserved for the copied elements.

### Example

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 | // set\_symmetric\_difference example  #include <iostream> // std::cout  #include <algorithm> // std::set\_symmetric\_difference, std::sort  #include <vector> // std::vector  int main () {  int first[] = {5,10,15,20,25};  int second[] = {50,40,30,20,10};  std::vector<int> v(10); // 0 0 0 0 0 0 0 0 0 0  std::vector<int>::iterator it;  std::sort (first,first+5); // 5 10 15 20 25  std::sort (second,second+5); // 10 20 30 40 50  it=std::set\_symmetric\_difference (first, first+5, second, second+5, v.begin());  // 5 15 25 30 40 50 0 0 0 0  v.resize(it-v.begin()); // 5 15 25 30 40 50  std::cout << "The symmetric difference has " << (v.size()) << " elements:\n";  for (it=v.begin(); it!=v.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/algorithm/set_symmetric_difference/) |

Output:

|  |
| --- |
| The symmetric difference has 6 elements:  5 15 25 30 40 50 |