library

<algorithm>

**<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 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 )

function template

<algorithm>

**std::all\_of**

template <class InputIterator, class UnaryPredicate>

bool all\_of (InputIterator first, InputIterator last, UnaryPredicate pred);

Test condition on all elements in range

Returns true if *pred* returns true for all the elements in the range [first,last) or if the range is empty, and false otherwise.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | *template*<*class* InputIterator, *class* UnaryPredicate>  *bool* all\_of (InputIterator first, InputIterator last, UnaryPredicate pred)  {  *while* (first!=last) {  *if* (!pred(\*first)) *return* *false*;  ++first;  }  *return* *true*;  } |

**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*.

pred

Unary function that accepts an element in the range as argument and returns a value convertible to bool. The value returned indicates whether the element fulfills the condition checked by this function.  
The function shall not modify its argument.  
This can either be a function pointer or a function object.

**Return value**

true if *pred* returns true for all the elements in the range or if the range is empty, and false otherwise.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 | *// all\_of example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::all\_of*  *#include <array> // std::array*  *int* main () {  std::array<*int*,8> foo = {3,5,7,11,13,17,19,23};  *if* ( std::all\_of(foo.begin(), foo.end(), [](*int* i){*return* i%2;}) )  std::cout << "All the elements are odd numbers.\n";  *return* 0;  } |

Output:

|  |
| --- |
| All the elements are odd numbers. |

**Complexity**

Up to linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Calls *pred* for each element until a mismatch is found.

**Data races**

Some (or all) of the objects in the range [first,last) are accessed (once at most).

**Exceptions**

Throws if either *pred* or an operation on an iterator throws.  
Note that invalid parameters cause *undefined behavior*.

function template

<algorithm>

**std::any\_of**

template <class InputIterator, class UnaryPredicate>

bool any\_of (InputIterator first, InputIterator last, UnaryPredicate pred);

Test if any element in range fulfills condition

Returns true if *pred* returns true for any of the elements in the range [first,last), and false otherwise.  
  
If [first,last) is an empty range, the function returns false.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | *template*<*class* InputIterator, *class* UnaryPredicate>  *bool* any\_of (InputIterator first, InputIterator last, UnaryPredicate pred)  {  *while* (first!=last) {  *if* (pred(\*first)) *return* *true*;  ++first;  }  *return* *false*;  } |

**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*.

pred

Unary function that accepts an element in the range as argument and returns a value convertible to bool. The value returned indicates whether the element fulfills the condition checked by this function.  
The function shall not modify its argument.  
This can either be a function pointer or a function object.

**Return value**

true if *pred* returns true for any of the elements in the range [first,last), and false otherwise.  
  
If [first,last) is an empty range, the function returns false.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 | *// any\_of example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::any\_of*  *#include <array> // std::array*  *int* main () {  std::array<*int*,7> foo = {0,1,-1,3,-3,5,-5};  *if* ( std::any\_of(foo.begin(), foo.end(), [](*int* i){*return* i<0;}) )  std::cout << "There are negative elements in the range.\n";  *return* 0;  } |

Output:

|  |
| --- |
| There are negative elements in the range. |

**Complexity**

Up to linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Calls *pred* for each element until a match is found.

**Data races**

Some (or all) of the objects in the range [first,last) are accessed (once at most).

**Exceptions**

Throws if either *pred* or an operation on an iterator throws.  
Note that invalid parameters cause *undefined behavior*.

function template

<algorithm>

**std::none\_of**

template <class InputIterator, class UnaryPredicate>

bool none\_of (InputIterator first, InputIterator last, UnaryPredicate pred);

Test if no elements fulfill condition

Returns true if *pred* returns false for all the elements in the range [first,last) or if the range is empty, and false otherwise.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | *template*<*class* InputIterator, *class* UnaryPredicate>  *bool* none\_of (InputIterator first, InputIterator last, UnaryPredicate pred)  {  *while* (first!=last) {  *if* (pred(\*first)) *return* *false*;  ++first;  }  *return* *true*;  } |

**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*.

pred

Unary function that accepts an element in the range as argument and returns a value convertible to bool. The value returned indicates whether the element fulfills the condition checked by this function.  
The function shall not modify its argument.  
This can either be a function pointer or a function object.

**Return value**

true if *pred* returns false for all the elements in the range [first,last) or if the range is empty, and false otherwise.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 | *// none\_of example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::none\_of*  *#include <array> // std::array*  *int* main () {  std::array<*int*,8> foo = {1,2,4,8,16,32,64,128};  *if* ( std::none\_of(foo.begin(), foo.end(), [](*int* i){*return* i<0;}) )  std::cout << "There are no negative elements in the range.\n";  *return* 0;  } |

Output:

|  |
| --- |
| There are no negative elements in the range. |

**Complexity**

Up to linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Calls *pred* for each element until a match is found.

**Data races**

Some (or all) of the objects in the range [first,last) are accessed (once at most).

**Exceptions**

Throws if either *pred* or an operation on an iterator throws.  
Note that invalid parameters cause *undefined behavior*.

function template

<algorithm>

**std::for\_each**

template <class InputIterator, class Function>

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

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);*  } |

**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.

**Return value**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

Returns *fn*.

**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;  } |

Output:

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

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Applies *fn* to each element.

**Data races**

The objects in the range [first,last) are accessed (each object is accessed exactly once).  
These objects may be modified if InputIterator is a *mutable iterator* type and *fn* is not a constant function.

**Exceptions**

Throws if *fn* throws or if any of the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::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*.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | *template*<*class* InputIterator, *class* T>  InputIterator find (InputIterator first, InputIterator last, *const* T& val)  {  *while* (first!=last) {  *if* (\*first==val) *return* first;  ++first;  }  *return* last;  } |

**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 | *// find example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::find*  *#include <vector> // std::vector*  *int* main () {  *int* myints[] = { 10, 20, 30 ,40 };  *int* \* p;  *// pointer to array element:*  p = std::find (myints,myints+4,30);  ++p;  std::cout << "The element following 30 is " << \*p << '\n';  std::vector<*int*> myvector (myints,myints+4);  std::vector<*int*>::iterator it;  *// iterator to vector element:*  it = find (myvector.begin(), myvector.end(), 30);  ++it;  std::cout << "The element following 30 is " << \*it << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| The element following 30 is 40  The element following 30 is 40 |

**Complexity**

Up to linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Compares elements until a match is found.

**Data races**

Some (or all) of the objects in the range [first,last) are accessed (once at most).

**Exceptions**

Throws if either an element comparison or an operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::find\_if**

template <class InputIterator, class UnaryPredicate>

InputIterator find\_if (InputIterator first, InputIterator last, UnaryPredicate pred);

Find element in range

Returns an iterator to the first element in the range [first,last) for which *pred* returns true. If no such element is found, the function returns *last*.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | *template*<*class* InputIterator, *class* UnaryPredicate>  InputIterator find\_if (InputIterator first, InputIterator last, UnaryPredicate pred)  {  *while* (first!=last) {  *if* (pred(\*first)) *return* first;  ++first;  }  *return* last;  } |

**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*.

pred

Unary function that accepts an element in the range as argument and returns a value convertible to bool. The value returned indicates whether the element is considered a match in the context of this function.  
The function shall not modify its argument.  
This can either be a function pointer or a function object.

**Return value**

An iterator to the first element in the range for which *pred* does not return false.  
If *pred* is false for all elements, 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 | *// find\_if example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::find\_if*  *#include <vector> // std::vector*  *bool* IsOdd (*int* i) {  *return* ((i%2)==1);  }  *int* main () {  std::vector<*int*> myvector;  myvector.push\_back(10);  myvector.push\_back(25);  myvector.push\_back(40);  myvector.push\_back(55);  std::vector<*int*>::iterator it = std::find\_if (myvector.begin(), myvector.end(), IsOdd);  std::cout << "The first odd value is " << \*it << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| The first odd value is 25 |

**Complexity**

Up to linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Calls *pred* for each element until a match is found.

**Data races**

Some (or all) of the objects in the range [first,last) are accessed (once at most).

**Exceptions**

Throws if either *pred* or an operation on an iterator throws.  
Note that invalid parameters cause *undefined behavior*.

You are using a version without Ads of this website. *Please, consider donating:*

Top of Form



Bottom of Form

[[hide](javascript:donate.hide())]

function template

<algorithm>

**std::find\_if\_not**

template <class InputIterator, class UnaryPredicate>

InputIterator find\_if\_not (InputIterator first, InputIterator last, UnaryPredicate pred);

Find element in range (negative condition)

Returns an iterator to the first element in the range [first,last) for which *pred* returns false. If no such element is found, the function returns *last*.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | *template*<*class* InputIterator, *class* UnaryPredicate>  InputIterator find\_if\_not (InputIterator first, InputIterator last, UnaryPredicate pred)  {  *while* (first!=last) {  *if* (!pred(\*first)) *return* first;  ++first;  }  *return* last;  } |

**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*.

pred

Unary function that accepts an element in the range as argument and returns a value convertible to bool. The value returned indicates whether the element is considered a match in the context of this function.  
The function shall not modify its argument.  
This can either be a function pointer or a function object.

**Return value**

An iterator to the first element in the range for which *pred* returns false.  
If *pred* is true for all elements, the function returns *last*.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | *// find\_if\_not example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::find\_if\_not*  *#include <array> // std::array*  *int* main () {  std::array<*int*,5> foo = {1,2,3,4,5};  std::array<*int*,5>::iterator it =  std::find\_if\_not (foo.begin(), foo.end(), [](*int* i){*return* i%2;} );  std::cout << "The first even value is " << \*it << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| The first even value is 2 |

**Complexity**

Up to linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Calls *pred* for each element until a mismatch is found.

**Data races**

Some (or all) of the objects in the range [first,last) are accessed (once at most).

**Exceptions**

Throws if either *pred* or an operation on an iterator throws.  
Note that invalid parameters cause *undefined behavior*.

function template

<algorithm>

**std::find\_end**

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

Find last subsequence in range

Searches the range [first1,last1) for the last 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 last of such occurrences. For an algorithm that returns the first instead, see [search](http://www.cplusplus.com/search).  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 | *template*<*class* ForwardIterator1, *class* ForwardIterator2>  ForwardIterator1 find\_end (ForwardIterator1 first1, ForwardIterator1 last1,  ForwardIterator2 first2, ForwardIterator2 last2)  {  *if* (first2==last2) *return* last1; *// specified in C++11*  ForwardIterator1 ret = last1;  *while* (first1!=last1)  {  ForwardIterator1 it1 = first1;  ForwardIterator2 it2 = first2;  *while* (\*it1==\*it2) { *// or: while (pred(\*it1,\*it2)) for version (2)*  ++it1; ++it2;  *if* (it2==last2) { ret=first1; *break*; }  *if* (it1==last1) *return* ret;  }  ++first1;  }  *return* ret;  } |

**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 last occurrence of [first2,last2) in [first1,last1).  
If the sequence is not found, the function returns *last1*.

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

If [first2,last2) is an empty range, the result is unspecified.

**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 | *// find\_end example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::find\_end*  *#include <vector> // std::vector*  *bool* myfunction (*int* i, *int* j) {  *return* (i==j);  }  *int* main () {  *int* myints[] = {1,2,3,4,5,1,2,3,4,5};  std::vector<*int*> haystack (myints,myints+10);  *int* needle1[] = {1,2,3};  *// using default comparison:*  std::vector<*int*>::iterator it;  it = std::find\_end (haystack.begin(), haystack.end(), needle1, needle1+3);  *if* (it!=haystack.end())  std::cout << "needle1 last found at position " << (it-haystack.begin()) << '\n';  *int* needle2[] = {4,5,1};  *// using predicate comparison:*  it = std::find\_end (haystack.begin(), haystack.end(), needle2, needle2+3, myfunction);  *if* (it!=haystack.end())  std::cout << "needle2 last found at position " << (it-haystack.begin()) << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| needle1 found at position 5  needle2 found at position 3 |

**Complexity**

Up to linear in count2\*(1+count1-count2), where *countX* is the [distance](http://www.cplusplus.com/distance) between *firstX* and *lastX*: Compares elements until the last matching subsequence is found.

**Data races**

Some (or all) of the objects in both ranges are accessed (possibly more than once).

**Exceptions**

Throws if any element comparison (or call to *pred*) throws or if any of the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::find\_first\_of**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

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

Find element from set in range

Returns an iterator to the first element in the range [first1,last1) that matches any of the elements in [first2,last2). If no such element is found, the function returns *last1*.  
  
The elements in [first1,last1) are sequentially compared to each of the values in [first2,last2) using operator== (or *pred*, in version *(2)*), until a pair matches.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 | *template*<*class* InputIterator, *class* ForwardIterator>  InputIterator find\_first\_of ( InputIterator first1, InputIterator last1,  ForwardIterator first2, ForwardIterator last2)  {  *while* (first1!=last1) {  *for* (ForwardIterator it=first2; it!=last2; ++it) {  *if* (\*it==\*first1) *// or: if (pred(\*it,\*first)) for version (2)*  *return* first1;  }  ++first1;  }  *return* last1;  } |

**Parameters**

first1, last1

* [C++98](javascript:switch2.select(1))
* [C++11](javascript:switch2.select(2))

[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 element values 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 value returned 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 in [first1,last1) that is part of [first2,last2).  
If no matches are found, the function returns *last1*.

* [C++98](javascript:switch3.select(1))
* [C++11](javascript:switch3.select(2))

If [first2,last2) is an empty range, the result is unspecified.

**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 | *// find\_first\_of example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::find\_first\_of*  *#include <vector> // std::vector*  *#include <cctype> // std::tolower*  *bool* comp\_case\_insensitive (*char* c1, *char* c2) {  *return* (std::tolower(c1)==std::tolower(c2));  }  *int* main () {  *int* mychars[] = {'a','b','c','A','B','C'};  std::vector<*char*> haystack (mychars,mychars+6);  std::vector<*char*>::iterator it;  *int* needle[] = {'A','B','C'};  *// using default comparison:*  it = find\_first\_of (haystack.begin(), haystack.end(), needle, needle+3);  *if* (it!=haystack.end())  std::cout << "The first match is: " << \*it << '\n';  *// using predicate comparison:*  it = find\_first\_of (haystack.begin(), haystack.end(),  needle, needle+3, comp\_case\_insensitive);  *if* (it!=haystack.end())  std::cout << "The first match is: " << \*it << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| The first match is: A  The first match is: a |

**Complexity**

Up to linear in count1\*count2 (where *countX* is the [distance](http://www.cplusplus.com/distance) between *firstX* and *lastX*): Compares elements until a match is found.

**Data races**

Some (or all) of the objects in both ranges are accessed (once at most in the case of [first1,last1), and possibly more than once in [first2,last2)).

**Exceptions**

Throws if any element comparison (or *pred*) throws or if any of the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::adjacent\_find**

|  |  |
| --- | --- |
| **equality (1)** | template <class ForwardIterator>  ForwardIterator adjacent\_find (ForwardIterator first, ForwardIterator last); |
| **predicate (2)** | template <class ForwardIterator, class BinaryPredicate>  ForwardIterator adjacent\_find (ForwardIterator first, ForwardIterator last,  BinaryPredicate pred); |

Find equal adjacent elements in range

Searches the range [first,last) for the first occurrence of two consecutive elements that match, and returns an iterator to the first of these two elements, or *last* if no such pair is found.  
  
Two elements match if they compare equal using operator== (or using *pred*, in version *(2)*).  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | *template* <*class* ForwardIterator>  ForwardIterator adjacent\_find (ForwardIterator first, ForwardIterator last)  {  *if* (first != last)  {  ForwardIterator next=first; ++next;  *while* (next != last) {  *if* (\*first == \*next) *// or: if (pred(\*first,\*next)), for version (2)*  *return* first;  ++first; ++next;  }  }  *return* last;  } |

**Parameters**

first, last

[Forward iterators](http://www.cplusplus.com/ForwardIterator) to the initial and final positions of the searched 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*.

pred

Binary function that accepts two elements as arguments, 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 pair of matching consecutive elements in the range [first,last).  
If no such pair is found, 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 | *// adjacent\_find example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::adjacent\_find*  *#include <vector> // std::vector*  *bool* myfunction (*int* i, *int* j) {  *return* (i==j);  }  *int* main () {  *int* myints[] = {5,20,5,30,30,20,10,10,20};  std::vector<*int*> myvector (myints,myints+8);  std::vector<*int*>::iterator it;  *// using default comparison:*  it = std::adjacent\_find (myvector.begin(), myvector.end());  *if* (it!=myvector.end())  std::cout << "the first pair of repeated elements are: " << \*it << '\n';  *//using predicate comparison:*  it = std::adjacent\_find (++it, myvector.end(), myfunction);  *if* (it!=myvector.end())  std::cout << "the second pair of repeated elements are: " << \*it << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| the first pair of repeated elements are: 30  the second pair of repeated elements are: 10 |

**Complexity**

Up to linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Compares elements until a match is found.

**Data races**

Some (or all) of the objects in the range [first,last) are accessed (once at most).

**Exceptions**

Throws if any element comparison (or *pred*) throws or if any of the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::count**

template <class InputIterator, class T>

typename iterator\_traits<InputIterator>::difference\_type

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

Count appearances of value in range

Returns the number of elements in the range [first,last) that compare equal to *val*.  
  
The function uses operator== to compare the individual elements to *val*.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | *template* <*class* InputIterator, *class* T>  *typename* iterator\_traits<InputIterator>::difference\_type  count (InputIterator first, InputIterator last, *const* T& val)  {  *typename* iterator\_traits<InputIterator>::difference\_type ret = 0;  *while* (first!=last) {  *if* (\*first == val) ++ret;  ++first;  }  *return* ret;  } |

**Parameters**

first, last

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the sequence of elements. 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*.

val

Value to match.  
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**

The number of elements in the range [first,last) that compare equal to *val*.  
The return type ([iterator\_traits](http://www.cplusplus.com/iterator_traits)<InputIterator>::difference\_type) is a signed integral type.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | *// count algorithm example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::count*  *#include <vector> // std::vector*  *int* main () {  *// counting elements in array:*  *int* myints[] = {10,20,30,30,20,10,10,20}; *// 8 elements*  *int* mycount = std::count (myints, myints+8, 10);  std::cout << "10 appears " << mycount << " times.\n";  *// counting elements in container:*  std::vector<*int*> myvector (myints, myints+8);  mycount = std::count (myvector.begin(), myvector.end(), 20);  std::cout << "20 appears " << mycount << " times.\n";  *return* 0;  } |

Output:

|  |
| --- |
| 10 appears 3 times.  20 appears 3 times. |

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Compares once each element.

**Data races**

The objects in the range [first,last) are accessed (each object is accessed exactly once).

**Exceptions**

Throws if either an element comparison or an operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::count\_if**

template <class InputIterator, class Predicate>

typename iterator\_traits<InputIterator>::difference\_type

count\_if (InputIterator first, InputIterator last, UnaryPredicate pred);

Return number of elements in range satisfying condition

Returns the number of elements in the range [first,last) for which *pred* is true.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | *template* <*class* InputIterator, *class* UnaryPredicate>  *typename* iterator\_traits<InputIterator>::difference\_type  count\_if (InputIterator first, InputIterator last, UnaryPredicate pred)  {  *typename* iterator\_traits<InputIterator>::difference\_type ret = 0;  *while* (first!=last) {  *if* (pred(\*first)) ++ret;  ++first;  }  *return* ret;  } |

**Parameters**

first, last

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the sequence of elements. 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*.

pred

Unary function that accepts an element in the range as argument, and returns a value convertible to bool. The value returned indicates whether the element is counted by this function.  
The function shall not modify its argument.  
This can either be a function pointer or a function object.

**Return value**

The number of elements in the range [first,last) for which *pred* does not return false.  
The return type ([iterator\_traits](http://www.cplusplus.com/iterator_traits)<InputIterator>::difference\_type) is a signed integral type.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | *// count\_if example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::count\_if*  *#include <vector> // std::vector*  *bool* IsOdd (*int* i) { *return* ((i%2)==1); }  *int* main () {  std::vector<*int*> myvector;  *for* (*int* i=1; i<10; i++) myvector.push\_back(i); *// myvector: 1 2 3 4 5 6 7 8 9*  *int* mycount = count\_if (myvector.begin(), myvector.end(), IsOdd);  std::cout << "myvector contains " << mycount << " odd values.\n";  *return* 0;  } |

Output:

|  |
| --- |
| myvector contains 5 odd values. |

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Calls *pred* once for each element.

**Data races**

The objects in the range [first,last) are accessed (each object is accessed exactly once).

**Exceptions**

Throws if *pred* trhows or if any of the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::mismatch**

|  |  |
| --- | --- |
| **equality (1)** | template <class InputIterator1, class InputIterator2>  pair<InputIterator1, InputIterator2>  mismatch (InputIterator1 first1, InputIterator1 last1,  InputIterator2 first2); |
| **predicate (2)** | template <class InputIterator1, class InputIterator2, class BinaryPredicate>  pair<InputIterator1, InputIterator2>  mismatch (InputIterator1 first1, InputIterator1 last1,  InputIterator2 first2, BinaryPredicate pred); |

Return first position where two ranges differ

Compares the elements in the range *[first1,last1)* with those in the range beginning at *first2*, and returns the first element of both sequences that does not match.  
  
The elements are compared using operator== (or *pred*, in version *(2)*).  
  
The function returns a [pair](http://www.cplusplus.com/pair) of iterators to the first element in each range that does not match.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | *template* <*class* InputIterator1, *class* InputIterator2>  pair<InputIterator1, InputIterator2>  mismatch (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2 )  {  *while* ( (first1!=last1) && (\*first1==\*first2) ) *// or: pred(\*first1,\*first2), for version 2*  { ++first1; ++first2; }  *return* std::make\_pair(first1,first2);  } |

**Parameters**

first1, last1

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the first 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

[Input iterator](http://www.cplusplus.com/InputIterator) to the initial position of the second sequence. Up to as many elements as in the range [first1,last1) can be accessed by the function.

pred

Binary function that accepts two elements as argument (one of each of the two sequences, in the same order), and returns a value convertible to bool. The value returned 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**

A [pair](http://www.cplusplus.com/pair), where its members first and second point to the first element in both sequences that did not compare equal to each other.  
If the elements compared in both sequences have all matched, the function returns a [pair](http://www.cplusplus.com/pair) with first set to *last1* and second set to the element in that same relative position in the second sequence.  
If none matched, it returns [make\_pair](http://www.cplusplus.com/make_pair)(first1,first2).

**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 | *// mismatch algorithm example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::mismatch*  *#include <vector> // std::vector*  *#include <utility> // std::pair*  *bool* mypredicate (*int* i, *int* j) {  *return* (i==j);  }  *int* main () {  std::vector<*int*> myvector;  *for* (*int* i=1; i<6; i++) myvector.push\_back (i\*10); *// myvector: 10 20 30 40 50*  *int* myints[] = {10,20,80,320,1024}; *// myints: 10 20 80 320 1024*  std::pair<std::vector<*int*>::iterator,*int*\*> mypair;  *// using default comparison:*  mypair = std::mismatch (myvector.begin(), myvector.end(), myints);  std::cout << "First mismatching elements: " << \*mypair.first;  std::cout << " and " << \*mypair.second << '\n';  ++mypair.first; ++mypair.second;  *// using predicate comparison:*  mypair = std::mismatch (mypair.first, myvector.end(), mypair.second, mypredicate);  std::cout << "Second mismatching elements: " << \*mypair.first;  std::cout << " and " << \*mypair.second << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| First mismatching elements: 30 and 80  Second mismatching elements: 40 and 320 |

**Complexity**

Up to linear in the [distance](http://www.cplusplus.com/distance) between *first1* and *last1*: Compares elements until a mismatch is found.

**Data races**

Some (or all) of the objects in both ranges are accessed (once at most).

**Exceptions**

Throws if any element comparison (or *pred*) throws or if any of the operations on iterators throws.  
Note that invalid parameters cause *undefined behavior*.

function template

<algorithm>

**std::equal**

|  |  |
| --- | --- |
| **equality (1)** | template <class InputIterator1, class InputIterator2>  bool equal (InputIterator1 first1, InputIterator1 last1,  InputIterator2 first2); |
| **predicate (2)** | template <class InputIterator1, class InputIterator2, class BinaryPredicate>  bool equal (InputIterator1 first1, InputIterator1 last1,  InputIterator2 first2, BinaryPredicate pred); |

Test whether the elements in two ranges are equal

Compares the elements in the range [first1,last1) with those in the range beginning at *first2*, and returns true if all of the elements in both ranges match.  
  
The elements are compared using operator== (or *pred*, in version *(2)*).  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 | *template* <*class* InputIterator1, *class* InputIterator2>  *bool* equal ( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2 )  {  *while* (first1!=last1) {  *if* (!(\*first1 == \*first2)) *// or: if (!pred(\*first1,\*first2)), for version 2*  *return* *false*;  ++first1; ++first2;  }  *return* *true*;  } |

**Parameters**

first1, last1

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the first 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

[Input iterator](http://www.cplusplus.com/InputIterator) to the initial position of the second sequence. The comparison includes up to as many elements of this sequence as those in the range [first1,last1).

pred

Binary function that accepts two elements as argument (one of each of the two sequences, in the same order), and returns a value convertible to bool. The value returned 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**

true if all the elements in the range [first1,last1) compare equal to those of the range starting at *first2*, and false otherwise.

**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 | *// equal algorithm example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::equal*  *#include <vector> // std::vector*  *bool* mypredicate (*int* i, *int* j) {  *return* (i==j);  }  *int* main () {  *int* myints[] = {20,40,60,80,100}; *// myints: 20 40 60 80 100*  std::vector<*int*>myvector (myints,myints+5); *// myvector: 20 40 60 80 100*  *// using default comparison:*  *if* ( std::equal (myvector.begin(), myvector.end(), myints) )  std::cout << "The contents of both sequences are equal.\n";  *else*  std::cout << "The contents of both sequences differ.\n";  myvector[3]=81; *// myvector: 20 40 60 81 100*  *// using predicate comparison:*  *if* ( std::equal (myvector.begin(), myvector.end(), myints, mypredicate) )  std::cout << "The contents of both sequences are equal.\n";  *else*  std::cout << "The contents of both sequences differ.\n";  *return* 0;  } |

Output:

|  |
| --- |
| The contents of both sequences are equal.  The contents of both sequence differ. |

**Complexity**

Up to linear in the [distance](http://www.cplusplus.com/distance) between *first1* and *last1*: Compares elements until a mismatch is found.

**Data races**

Some (or all) of the objects in both ranges are accessed (once at most).

**Exceptions**

Throws if any of the element comparisons (or *pred*) throws, or if any of the operations on iterators throws.  
Note that invalid parameters cause *undefined behavior*.

function template

<algorithm>

**std::is\_permutation**

|  |  |
| --- | --- |
| **equality (1)** | template <class ForwardIterator1, class ForwardIterator2>  bool is\_permutation (ForwardIterator1 first1, ForwardIterator1 last1,  ForwardIterator2 first2); |
| **predicate (2)** | template <class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>  bool is\_permutation (ForwardIterator1 first1, ForwardIterator1 last1,  ForwardIterator2 first2, BinaryPredicate pred); |

Test whether range is permutation of another

Compares the elements in the range [first1,last1) with those in the range beginning at *first2*, and returns true if all of the elements in both ranges match, even in a different order.  
  
The elements are compared using operator== (or *pred*, in version *(2)*).  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | *template* <*class* InputIterator1, *class* InputIterator2>  *bool* is\_permutation (InputIterator1 first1, InputIterator1 last1,  InputIterator2 first2)  {  std::tie (first1,first2) = std::mismatch (first1,last1,first2);  *if* (first1==last1) *return* *true*;  InputIterator2 last2 = first2; std::advance (last2,std::distance(first1,last1));  *for* (InputIterator1 it1=first1; it1!=last1; ++it1) {  *if* (std::find(first1,it1,\*it1)==it1) {  *auto* n = std::count (first2,last2,\*it1);  *if* (n==0 || std::count (it1,last1,\*it1)!=n) *return* *false*;  }  }  *return* *true*;  } |

**Parameters**

first1, last1

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the first 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

[Input iterator](http://www.cplusplus.com/InputIterator) to the initial position of the second sequence.  
The function considers as many elements of this sequence as those in the range [first1,last1).  
If this sequence is shorter, it causes *undefined behavior*.

pred

Binary function that accepts two elements as argument (one of each of the two sequences, in the same order), and returns a value convertible to bool. The value returned 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.

InputIterator1 and InputIterator2 shall point to the same type.

**Return value**

true if all the elements in the range [first1,last1) compare equal to those of the range starting at *first2* in any order, and false otherwise.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | *// is\_permutation example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::is\_permutation*  *#include <array> // std::array*  *int* main () {  std::array<*int*,5> foo = {1,2,3,4,5};  std::array<*int*,5> bar = {3,1,4,5,2};  *if* ( std::is\_permutation (foo.begin(), foo.end(), bar.begin()) )  std::cout << "foo and bar contain the same elements.\n";  *return* 0;  } |

Output:

|  |
| --- |
| foo and bar contain the same elements. |

**Complexity**

If both sequence are [equal](http://www.cplusplus.com/equal) (with the elements in the same order), linear in the [distance](http://www.cplusplus.com/distance) between *first1* and *last1*.  
Otherwise, up to quadratic: Performs at most N2 element comparisons until the result is determined (where *N* is the [distance](http://www.cplusplus.com/distance) between *first1* and *last1*).

**Data races**

Some (or all) of the objects in both ranges are accessed (possibly multiple times each).

**Exceptions**

Throws if any of the element comparisons (or *pred*) throws, or if any of the operations on iterators throws.  
Note that invalid parameters cause *undefined behavior*.

function template

<algorithm>

**std::search**

|  |  |
| --- | --- |
| **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); |

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).  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | *template*<*class* ForwardIterator1, *class* ForwardIterator2>  ForwardIterator1 search ( ForwardIterator1 first1, ForwardIterator1 last1,  ForwardIterator2 first2, ForwardIterator2 last2)  {  *if* (first2==last2) *return* first1; *// specified in C++11*    *while* (first1!=last1)  {  ForwardIterator1 it1 = first1;  ForwardIterator2 it2 = first2;  *while* (\*it1==\*it2) { *// or: while (pred(\*it1,\*it2)) for version 2*  ++it1; ++it2;  *if* (it2==last2) *return* first1;  *if* (it1==last1) *return* last1;  }  ++first1;  }  *return* last1;  } |

**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*.

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

If [first2,last2) is an empty range, the result is unspecified.

**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;  } |

Output:

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

**Complexity**

Up to linear in count1\*count2 (where *countX* is the [distance](http://www.cplusplus.com/distance) between *firstX* and *lastX*): Compares elements until a matching subsequence is found.

**Data races**

Some (or all) of the objects in both ranges are accessed (possibly more than once).

**Exceptions**

Throws if any of the element comparisons (or *pred*) throws or if any of the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::search\_n**

|  |  |
| --- | --- |
| **equality (1)** | template <class ForwardIterator, class Size, class T>  ForwardIterator search\_n (ForwardIterator first, ForwardIterator last,  Size count, const T& val); |
| **predicate (2)** | template <class ForwardIterator, class Size, class T, class BinaryPredicate>  ForwardIterator search\_n ( ForwardIterator first, ForwardIterator last,  Size count, const T& val, BinaryPredicate pred ); |

Search range for elements

Searches the range [first,last) for a sequence of *count* elements, each comparing equal to *val* (or for which *pred* returns true).  
  
The function returns an iterator to the first of such elements, or *last* if no such sequence is found.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | *template*<*class* ForwardIterator, *class* Size, *class* T>  ForwardIterator search\_n (ForwardIterator first, ForwardIterator last,  Size count, *const* T& val)  {  ForwardIterator it, limit;  Size i;  limit=first; std::advance(limit,std::distance(first,last)-count);  *while* (first!=limit)  {  it = first; i=0;  *while* (\*it==val) *// or: while (pred(\*it,val)) for the pred version*  { ++it; *if* (++i==count) *return* first; }  ++first;  }  *return* last;  } |

**Parameters**

first, last

Forward iterators to the initial and final positions of the searched 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*.

count

Minimum number of successive elements to match.  
Size shall be (convertible to) an integral type.

val

Individual value to be compared, or to be used as argument for *pred* (in the second version).  
for the first version, T shall be a type supporting comparisons with the elements pointed by InputIterator using operator== (with the elements as left-hand size operands, and *val* as right-hand side).

pred

Binary function that accepts two arguments (one element from the sequence as first, and *val* as second), and returns a value convertible to bool. The value returned indicates whether the element is considered a 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 sequence.  
If no such sequence is found, 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 29 30 31 32 33 | *// search\_n example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::search\_n*  *#include <vector> // std::vector*  *bool* mypredicate (*int* i, *int* j) {  *return* (i==j);  }  *int* main () {  *int* myints[]={10,20,30,30,20,10,10,20};  std::vector<*int*> myvector (myints,myints+8);  std::vector<*int*>::iterator it;  *// using default comparison:*  it = std::search\_n (myvector.begin(), myvector.end(), 2, 30);  *if* (it!=myvector.end())  std::cout << "two 30s found at position " << (it-myvector.begin()) << '\n';  *else*  std::cout << "match not found\n";  *// using predicate comparison:*  it = std::search\_n (myvector.begin(), myvector.end(), 2, 10, mypredicate);  *if* (it!=myvector.end())  std::cout << "two 10s found at position " << *int*(it-myvector.begin()) << '\n';  *else*  std::cout << "match not found\n";  *return* 0;  } |

Output:

|  |
| --- |
| Two 30s found at position 2  Two 10s found at position 5 |

**Complexity**

Up to linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Compares elements until a matching subsequence is found.

**Data races**

Some (or all) of the objects in the range [first,last) are accessed (once at most).

**Exceptions**

Throws if any of the element comparisons (or *pred*) throws or if any of the operations on iterators throws.  
Note that invalid parameters cause *undefined behavior*.

function template

<algorithm>

**std::copy**

template <class InputIterator, class OutputIterator>

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

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).  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | *template*<*class* InputIterator, *class* OutputIterator>  OutputIterator copy (InputIterator first, InputIterator last, OutputIterator result)  {  *while* (first!=last) {  \*result = \*first;  ++result; ++first;  }  *return* 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;  } |

Output:

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

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Performs an assignment operation for each element in the range.

**Data races**

The objects in the range [first,last) are accessed (each object is accessed exactly once).  
The objects in the range between *result* and the returned value are modified (each object is modified exactly once).

**Exceptions**

Throws if either an element assignment or an operation on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::copy\_n**

template <class InputIterator, class Size, class OutputIterator>

OutputIterator copy\_n (InputIterator first, Size n, OutputIterator result);

Copy elements

Copies the first *n* elements from the range beginning at *first* into the range beginning at *result*.  
  
The function returns an iterator to the end of the destination range (which points to one past the last element copied).  
  
If *n* is negative, the function does nothing.  
  
If the ranges overlap, some of the elements in the range pointed by *result* may have undefined but valid values.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 | *template*<*class* InputIterator, *class* Size, *class* OutputIterator>  OutputIterator copy\_n (InputIterator first, Size n, OutputIterator result)  {  *while* (n>0) {  \*result = \*first;  ++result; ++first;  --n;  }  *return* result;  } |

**Parameters**

first

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial position in a sequence of at least *n* elements to be copied.  
InputIterator shall point to a type [assignable](http://www.cplusplus.com/is_assignable) to the elements pointed by OutputIterator.

n

Number of elements to copy.  
If this value is negative, the function does nothing.  
Size shall be (convertible to) an integral type.

result

[Output iterator](http://www.cplusplus.com/OutputIterator) to the initial position in the destination sequence of at least *n* elements.  
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 20 21 | *// copy\_n 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;  myvector.resize(7); *// allocate space for 7 elements*  std::copy\_n ( 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;  } |

Output:

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

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Performs an assignment operation for each element in the range.

**Data races**

The objects in the range of *n* elements pointed by *first* are accessed (each object is accessed exactly once).  
The objects in the range between *result* and the returned value are modified (each object is modified exactly once).

**Exceptions**

Throws if either an element assignment or an operation on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::copy\_if**

template <class InputIterator, class OutputIterator, class UnaryPredicate>

OutputIterator copy\_if (InputIterator first, InputIterator last,

OutputIterator result, UnaryPredicate pred);

Copy certain elements of range

Copies the elements in the range [first,last) for which *pred* returns true to the range beginning at *result*.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 | *template* <*class* InputIterator, *class* OutputIterator, *class* UnaryPredicate>  OutputIterator copy\_if (InputIterator first, InputIterator last,  OutputIterator result, UnaryPredicate pred)  {  *while* (first!=last) {  *if* (pred(\*first)) {  \*result = \*first;  ++result;  }  ++first;  }  *return* result;  } |

**Parameters**

first, last

Input iterators to the initial and final positions in a sequence. The range copied 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*.  
InputIterator shall point to a type [assignable](http://www.cplusplus.com/is_assignable) to the elements pointed by OutputIterator.

result

[Output iterator](http://www.cplusplus.com/OutputIterator) to the initial position of the range where the resulting sequence is stored. The range includes as many elements as [first,last).

pred

Unary function that accepts an element in the range as argument, and returns a value convertible to bool. The value returned indicates whether the element is to be copied (if true, it is copied).  
The function shall not modify any of its arguments.  
This can either be a function pointer or a function object.

The ranges shall not overlap.

**Return value**

An iterator pointing to the element that follows the last element written in the result sequence.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | *// copy\_if example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::copy\_if, std::distance*  *#include <vector> // std::vector*  *int* main () {  std::vector<*int*> foo = {25,15,5,-5,-15};  std::vector<*int*> bar (foo.size());  *// copy only positive numbers:*  *auto* it = std::copy\_if (foo.begin(), foo.end(), bar.begin(), [](*int* i){*return* !(i<0);} );  bar.resize(std::distance(bar.begin(),it)); *// shrink container to new size*  std::cout << "bar contains:";  *for* (*int*& x: bar) std::cout << ' ' << x;  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| bar contains: 25 15 5 |

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Applies *pred* to each element in the range and performs at most that many assignments.

**Data races**

The objects in the range [first,last) are accessed.  
The objects in the range between *result* and the returned value are modified.

**Exceptions**

Throws if any of *pred*, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::copy\_backward**

template <class BidirectionalIterator1, class BidirectionalIterator2>

BidirectionalIterator2 copy\_backward (BidirectionalIterator1 first,

BidirectionalIterator1 last,

BidirectionalIterator2 result);

Copy range of elements backward

Copies the elements in the range [first,last) starting from the end into the range terminating at *result*.  
  
The function returns an iterator to the first element in the destination range.  
  
The resulting range has the elements in the exact same order as [first,last). To reverse their order, see [reverse\_copy](http://www.cplusplus.com/reverse_copy).  
  
The function begins by copying \*(last-1) into \*(result-1), and then follows backward by the elements preceding these, until *first* is reached (and including it).  
  
The ranges shall not overlap in such a way that *result* (which is the *past-the-end element* in the destination range) points to an element in the range (first,last]. For such cases, see [copy](http://www.cplusplus.com/copy).  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | *template*<*class* BidirectionalIterator1, *class* BidirectionalIterator2>  BidirectionalIterator2 copy\_backward ( BidirectionalIterator1 first,  BidirectionalIterator1 last,  BidirectionalIterator2 result )  {  *while* (last!=first) \*(--result) = \*(--last);  *return* result;  } |

**Parameters**

first, last

[Bidirectional iterators](http://www.cplusplus.com/BidirectionalIterator) 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

[Bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) to the *past-the-end* position in the destination sequence.  
This shall not point to any element in the range (first,last].

**Return value**

An iterator to the first element of the destination sequence 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 20 21 22 23 | *// copy\_backward example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::copy\_backward*  *#include <vector> // std::vector*  *int* main () {  std::vector<*int*> myvector;  *// set some values:*  *for* (*int* i=1; i<=5; i++)  myvector.push\_back(i\*10); *// myvector: 10 20 30 40 50*  myvector.resize(myvector.size()+3); *// allocate space for 3 more elements*  std::copy\_backward ( myvector.begin(), myvector.begin()+5, myvector.end() );  std::cout << "myvector contains:";  *for* (std::vector<*int*>::iterator it=myvector.begin(); it!=myvector.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  *return* 0;  } |

Output:

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

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Performs an assignment operation for each element in the range.

**Data races**

The objects in the range [first,last) are accessed (each object is accessed exactly once).  
The objects in the range between the returned value and *result* are modified (each object is modified exactly once).

**Exceptions**

Throws if either an element assignment or an operation on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::move**

template <class InputIterator, class OutputIterator>

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

Move range of elements

Moves the elements in the range [first,last) into the range beginning at *result*.  
  
The value of the elements in the [first,last) is transferred to the elements pointed by *result*. After the call, the elements in the range [first,last) are left in an unspecified but valid state.  
  
The ranges shall not overlap in such a way that *result* points to an element in the range *[first,last)*. For such cases, see [move\_backward](http://www.cplusplus.com/move_backward).  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | *template*<*class* InputIterator, *class* OutputIterator>  OutputIterator move (InputIterator first, InputIterator last, OutputIterator result)  {  *while* (first!=last) {  \*result = std::move(\*first);  ++result; ++first;  }  *return* result;  } |

**Parameters**

first, last

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions in a sequence to be moved. 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 moved.

**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 | *// move algorithm example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::move (ranges)*  *#include <utility> // std::move (objects)*  *#include <vector> // std::vector*  *#include <string> // std::string*  *int* main () {  std::vector<std::string> foo = {"air","water","fire","earth"};  std::vector<std::string> bar (4);  *// moving ranges:*  std::cout << "Moving ranges...\n";  std::move ( foo.begin(), foo.begin()+4, bar.begin() );  std::cout << "foo contains " << foo.size() << " elements:";  std::cout << " (each in an unspecified but valid state)";  std::cout << '\n';  std::cout << "bar contains " << bar.size() << " elements:";  *for* (std::string& x: bar) std::cout << " [" << x << "]";  std::cout << '\n';  *// moving container:*  std::cout << "Moving container...\n";  foo = std::move (bar);  std::cout << "foo contains " << foo.size() << " elements:";  *for* (std::string& x: foo) std::cout << " [" << x << "]";  std::cout << '\n';  std::cout << "bar is in an unspecified but valid state";  std::cout << '\n';  *return* 0;  } |

Possible output:

|  |
| --- |
| Moving ranges...  foo contains 4 elements: (each in an unspecified but valid state)  bar contains 4 elements: [air] [water] [fire] [earth]  Moving container...  foo contains 4 elements: [air] [water] [fire] [earth]  bar is in an unspecified but valid state |

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Performs a move-assignment for each element in the range.

**Data races**

The objects in both ranges are modified.

**Exceptions**

Throws if either an element move-assignment or an operation on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::move\_backward**

template <class BidirectionalIterator1, class BidirectionalIterator2>

BidirectionalIterator2 move\_backward (BidirectionalIterator1 first,

BidirectionalIterator1 last,

BidirectionalIterator2 result);

Move range of elements backward

Moves the elements in the range [first,last) starting from the end into the range terminating at *result*.  
  
The function returns an iterator to the first element in the destination range.  
  
The resulting range has the elements in the exact same order as [first,last). To reverse their order, see [reverse](http://www.cplusplus.com/reverse).  
  
The function begins by moving \*(last-1) into \*(result-1), and then follows backward by the elements preceding these, until *first* is reached (and including it).  
  
The ranges shall not overlap in such a way that *result* (which is the *past-the-end element* in the destination range) points to an element in the range (first,last]. For such cases, see [move](http://www.cplusplus.com/move).  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | *template*<*class* BidirectionalIterator1, *class* BidirectionalIterator2>  BidirectionalIterator2 move\_backward ( BidirectionalIterator1 first,  BidirectionalIterator1 last,  BidirectionalIterator2 result )  {  *while* (last!=first) \*(--result) = std::move(\*(--last));  *return* result;  } |

**Parameters**

first, last

[Bidirectional iterators](http://www.cplusplus.com/BidirectionalIterator) to the initial and final positions in a sequence to be moved. 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

[Bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) to the *past-the-end* position in the destination sequence.  
This shall not point to any element in the range (first,last].

**Return value**

An iterator to the first element of the destination sequence where elements have been moved.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | *// move\_backward example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::move\_backward*  *#include <string> // std::string*  *int* main () {  std::string elems[10] = {"air","water","fire","earth"};  *// insert new element at the beginning:*  std::move\_backward (elems,elems+4,elems+5);  elems[0]="ether";  std::cout << "elems contains:";  *for* (*int* i=0; i<10; ++i)  std::cout << " [" << elems[i] << "]";  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| elems contains: [ether] [air] [water] [fire] [earth] [] [] [] [] [] |

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Performs a move-assignment for each element in the range.

**Data races**

The objects in both ranges are modified.

**Exceptions**

Throws if either an element move-assignment or an operation on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

C++98: <algorithm>, C++11: <utility>

**std::swap**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

template <class T> void swap (T& a, T& b);

Exchange values of two objects

Exchanges the values of *a* and *b*.

* [C++98](javascript:switch2.select(1))
* [C++11](javascript:switch2.select(2))

The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 | *template* <*class* T> *void* swap ( T& a, T& b )  {  T c(a); a=b; b=c;  } |

Notice how this function involves a copy construction and two assignment operations, which may not be the most efficient way of swapping the contents of classes that store large quantities of data, since each of these operations generally operate in linear time on their size.  
  
Large data types can provide an overloaded version of this function optimizing its performance. Notably, all [standard containers](http://www.cplusplus.com/stl) specialize it in such a way that only a few internal pointers are swapped instead of their entire contents, making them operate in constant time.

Many components of the standard library (within std) call swap in an *unqualified* manner to allow custom overloads for non-fundamental types to be called instead of this generic version: Custom overloads of swap declared in the same namespace as the type for which they are provided get selected through *argument-dependent lookup* over this generic version.

**Parameters**

a, b

Two objects, whose contents are swapped.

* [C++98](javascript:switch3.select(1))
* [C++11](javascript:switch3.select(2))

Type T shall be *copy-constructible* and *assignable*.

**Return value**

none

**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;  } |

Output:

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

**Complexity**

**Non-array:** Constant: Performs exactly one construction and two assignments (although notice that each of these operations works on its own complexity).  
**Array:** Linear in *N*: performs a swap operation per element.

**Data races**

Both *a* and *b* are modified.

**Exceptions**

Throws if the construction or assignment of type T throws.  
Never throws if T is [*nothrow-move-constructible*](http://www.cplusplus.com/is_nothrow_move_constructible) and [*nothrow-move-assignable*](http://www.cplusplus.com/is_nothrow_move_assignable).  
Note that if T does not fulfill the requirements specified above (in [parameters](http://www.cplusplus.com/reference/algorithm/swap/#parameters)), it causes *undefined behavior*.

function template

<algorithm>

**std::swap\_ranges**

template <class ForwardIterator1, class ForwardIterator2>

ForwardIterator2 swap\_ranges (ForwardIterator1 first1, ForwardIterator1 last1,

ForwardIterator2 first2);

Exchange values of two ranges

Exchanges the values of each of the elements in the range [first1,last1) with those of their respective elements in the range beginning at *first2*.  
  
The function calls [swap](http://www.cplusplus.com/swap) (unqualified) to exchange the elements.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 | *template*<*class* ForwardIterator1, *class* ForwardIterator2>  ForwardIterator2 swap\_ranges (ForwardIterator1 first1, ForwardIterator1 last1,  ForwardIterator2 first2)  {  *while* (first1!=last1) {  swap (\*first1, \*first2);  ++first1; ++first2;  }  *return* first2;  } |

**Parameters**

first1, last1

[Forward iterators](http://www.cplusplus.com/ForwardIterator) to the initial and final positions in one of the sequences to be swapped. 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

[Forward iterator](http://www.cplusplus.com/ForwardIterator) to the initial position in the other sequence to be swapped. The range used includes the same number of elements as the range *[first1,last1)*.  
The two ranges shall not overlap.

The ranges shall not overlap.  
[swap](http://www.cplusplus.com/swap) shall be defined to exchange the types pointed by both iterator types symmetrically (in both orders).

**Return value**

An iterator to the last element swapped in the second sequence.

**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 | *// swap\_ranges example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::swap\_ranges*  *#include <vector> // std::vector*  *int* main () {  std::vector<*int*> foo (5,10); *// foo: 10 10 10 10 10*  std::vector<*int*> bar (5,33); *// bar: 33 33 33 33 33*  std::swap\_ranges(foo.begin()+1, foo.end()-1, bar.begin());  *// print out results of swap:*  std::cout << "foo contains:";  *for* (std::vector<*int*>::iterator it=foo.begin(); it!=foo.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  std::cout << "bar contains:";  *for* (std::vector<*int*>::iterator it=bar.begin(); it!=bar.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| foo contains: 10 33 33 33 10  bar contains: 10 10 10 33 33 |

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Performs a swap operation for each element in the range.

**Data races**

The objects in both ranges are modified.

**Exceptions**

Throws if either an element assignment or an operation on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::iter\_swap**

template <class ForwardIterator1, class ForwardIterator2>

void iter\_swap (ForwardIterator1 a, ForwardIterator2 b);

Exchange values of objects pointed by two iterators

Swaps the elements pointed by *a* and *b*.  
  
The function calls [swap](http://www.cplusplus.com/swap) (unqualified) to exchange the elements.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 | *template* <*class* ForwardIterator1, *class* ForwardIterator2>  *void* iter\_swap (ForwardIterator1 a, ForwardIterator2 b)  {  swap (\*a, \*b);  } |

**Parameters**

a, b

[Forward iterators](http://www.cplusplus.com/ForwardIterator) to the objects to swap.  
[swap](http://www.cplusplus.com/swap) shall be defined to exchange values of the type pointed by the iterators.

**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 | *// iter\_swap example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::iter\_swap*  *#include <vector> // std::vector*  *int* main () {  *int* myints[]={10,20,30,40,50 }; *// myints: 10 20 30 40 50*  std::vector<*int*> myvector (4,99); *// myvector: 99 99 99 99*  std::iter\_swap(myints,myvector.begin()); *// myints: [99] 20 30 40 50*  *// myvector: [10] 99 99 99*  std::iter\_swap(myints+3,myvector.begin()+2); *// myints: 99 20 30 [99]*  *// myvector: 10 99 [40] 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;  } |

Output:

|  |
| --- |
| myvector contains: 10 99 40 99 |

**Complexity**

Constant: Calls [swap](http://www.cplusplus.com/swap) once.

**Data races**

The objects pointed by both iterators are modified.

**Exceptions**

Throws if the call to [swap](http://www.cplusplus.com/swap) throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::transform**

|  |  |
| --- | --- |
| **unary operation(1)** | template <class InputIterator, class OutputIterator, class UnaryOperation>  OutputIterator transform (InputIterator first1, InputIterator last1,  OutputIterator result, UnaryOperation op); |
| **binary operation(2)** | template <class InputIterator1, class InputIterator2,  class OutputIterator, class BinaryOperation>  OutputIterator transform (InputIterator1 first1, InputIterator1 last1,  InputIterator2 first2, OutputIterator result,  BinaryOperation binary\_op); |

Transform range

Applies an operation sequentially to the elements of one (1) or two (2) ranges and stores the result in the range that begins at *result*.

(1) unary operation

Applies *op* to each of the elements in the range [first1,last1) and stores the value returned by each operation in the range that begins at *result*.

(2) binary operation

Calls *binary\_op* using each of the elements in the range [first1,last1) as first argument, and the respective argument in the range that begins at *first2* as second argument. The value returned by each call is stored in the range that begins at *result*.

The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 | *template* <*class* InputIterator, *class* OutputIterator, *class* UnaryOperator>  OutputIterator transform (InputIterator first1, InputIterator last1,  OutputIterator result, UnaryOperator op)  {  *while* (first1 != last1) {  \*result = op(\*first1); *// or: \*result=binary\_op(\*first1,\*first2++);*  ++result; ++first1;  }  *return* result;  } |

The function allows for the destination range to be the same as one of the input ranges to make transformations *in place*.

**Parameters**

first1, last1

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the first 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

[Input iterator](http://www.cplusplus.com/InputIterator) to the initial position of the second range. The range includes as many elements as [first1,last1).

result

[Output iterator](http://www.cplusplus.com/OutputIterator) to the initial position of the range where the operation results are stored. The range includes as many elements as [first1,last1).

op

Unary function that accepts one element of the type pointed by InputIterator as argument, and returns some result value convertible to the type pointed by OutputIterator.  
This can either be a function pointer or a function object.

binary\_op

Binary function that accepts two elements as argument (one of each of the two sequences), and returns some result value convertible to the type pointed by OutputIterator.  
This can either be a function pointer or a function object.

Neither *op* nor *binary\_op* should directly modify the elements passed as its arguments: These are indirectly modified by the algorithm (using the return value) if the same range is specified for *result*.

**Return value**

An iterator pointing to the element that follows the last element written in the *result* sequence.

**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 | *// transform algorithm example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::transform*  *#include <vector> // std::vector*  *#include <functional> // std::plus*  *int* op\_increase (*int* i) { *return* ++i; }  *int* main () {  std::vector<*int*> foo;  std::vector<*int*> bar;  *// set some values:*  *for* (*int* i=1; i<6; i++)  foo.push\_back (i\*10); *// foo: 10 20 30 40 50*  bar.resize(foo.size()); *// allocate space*  std::transform (foo.begin(), foo.end(), bar.begin(), op\_increase);  *// bar: 11 21 31 41 51*  *// std::plus adds together its two arguments:*  std::transform (foo.begin(), foo.end(), bar.begin(), foo.begin(), std::plus<*int*>());  *// foo: 21 41 61 81 101*  std::cout << "foo contains:";  *for* (std::vector<*int*>::iterator it=foo.begin(); it!=foo.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| foo contains: 21 41 61 81 101 |

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first1* and *last1*: Performs one assignment and one application of *op* (or *binary\_op*) per element.

**Data races**

The objects in the range [first1,last1) (and eventually those in the range beginning at *first2*) are accessed (each object is accessed exactly once).  
The objects in the range beginning at *result* are modified.

**Exceptions**

Throws if any of the function calls, the assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::replace**

template <class ForwardIterator, class T>

void replace (ForwardIterator first, ForwardIterator last,

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

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*.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | *template* <*class* ForwardIterator, *class* T>  *void* replace (ForwardIterator first, ForwardIterator last,  *const* T& old\_value, *const* T& new\_value)  {  *while* (first!=last) {  *if* (\*first == old\_value) \*first=new\_value;  ++first;  }  } |

**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;  } |

Output:

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

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Compares each element and assigns to those matching.

**Data races**

The objects in the range [first,last) are accessed and potentially modified.

**Exceptions**

Throws if any of the element comparisons, element assignments or operations on iterators throw.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::replace\_if**

template <class ForwardIterator, class UnaryPredicate, class T>

void replace\_if (ForwardIterator first, ForwardIterator last,

UnaryPredicate pred, const T& new\_value );

Replace values in range

Assigns *new\_value* to all the elements in the range *[first,last)* for which *pred* returns true.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | *template* < *class* ForwardIterator, *class* UnaryPredicate, *class* T >  *void* replace\_if (ForwardIterator first, ForwardIterator last,  UnaryPredicate pred, *const* T& new\_value)  {  *while* (first!=last) {  *if* (pred(\*first)) \*first=new\_value;  ++first;  }  } |

**Parameters**

first, last

[Forward iterators](http://www.cplusplus.com/ForwardIterator) to the initial and final positions in a sequence of elements that support being 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*.

pred

Unary function that accepts an element in the range as argument, and returns a value convertible to bool. The value returned indicates whether the element is to be replaced (if true, it is replaced).  
The function shall not modify its argument.  
This can either be a function pointer or a function object.

new\_value

Value to assign to replaced elements.

**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 | *// replace\_if example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::replace\_if*  *#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::replace\_if (myvector.begin(), myvector.end(), IsOdd, 0); *// 0 2 0 4 0 6 0 8 0*  std::cout << "myvector contains:";  *for* (std::vector<*int*>::iterator it=myvector.begin(); it!=myvector.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| myvector contains: 0 2 0 4 0 6 0 8 0 |

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Applies *pred* to each element and assigns to those matching.

**Data races**

The objects in the range [first,last) are accessed and potentially modified.

**Exceptions**

Throws if any of *pred*, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::replace\_copy**

template <class InputIterator, class OutputIterator, class T>

OutputIterator replace\_copy (InputIterator first, InputIterator last,

OutputIterator result,

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

Copy range replacing value

Copies the elements in the range [first,last) to the range beginning at *result*, replacing the appearances of *old\_value* by *new\_value*.  
  
The function uses operator== to compare the individual elements to *old\_value*.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 | *template* <*class* InputIterator, *class* OutputIterator, *class* T>  OutputIterator replace\_copy (InputIterator first, InputIterator last,  OutputIterator result, *const* T& old\_value, *const* T& new\_value)  {  *while* (first!=last) {  \*result = (\*first==old\_value)? new\_value: \*first;  ++first; ++result;  }  *return* result;  } |

**Parameters**

first, last

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions in a sequence. The range copied 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 of the range where the resulting sequence is stored. The range includes as many elements as [first,last).  
The pointed type shall support being assigned a value of type T.

old\_value

Value to be replaced.

new\_value

Replacement value.

The ranges shall not overlap.

**Return value**

An iterator pointing to the element that follows the last element written in the result sequence.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | *// replace\_copy example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::replace\_copy*  *#include <vector> // std::vector*  *int* main () {  *int* myints[] = { 10, 20, 30, 30, 20, 10, 10, 20 };  std::vector<*int*> myvector (8);  std::replace\_copy (myints, myints+8, myvector.begin(), 20, 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;  } |

Output:

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

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Performs a comparison and an assignment for each element.

**Data races**

The objects in the range [first,last) are accessed.  
The objects in the range between *result* and the returned value are modified.

**Exceptions**

Throws if any of the element comparisons, element assignments or operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::replace\_copy\_if**

template <class InputIterator, class OutputIterator, class UnaryPredicate, class T>

OutputIterator replace\_copy\_if (InputIterator first, InputIterator last,

OutputIterator result, UnaryPredicate pred,

const T& new\_value);

Copy range replacing value

Copies the elements in the range [first,last) to the range beginning at *result*, replacing those for which *pred* returns true by *new\_value*.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | *template* <*class* InputIterator, *class* OutputIterator, *class* UnaryPredicate, *class* T>  OutputIterator replace\_copy\_if (InputIterator first, InputIterator last,  OutputIterator result, UnaryPredicate pred,  *const* T& new\_value)  {  *while* (first!=last) {  \*result = (pred(\*first))? new\_value: \*first;  ++first; ++result;  }  *return* result;  } |

**Parameters**

first, last

Input iterators to the initial and final positions in a sequence. The range copied 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 of the range where the resulting sequence is stored. The range includes as many elements as [first,last).  
The pointed type shall support being assigned a value of type T.

pred

Unary function that accepts an element in the range as argument, and returns a value convertible to bool. The value returned indicates whether the element is to be replaced in the copy (if true, it is replaced).  
The function shall not modify its argument.  
This can either be a function pointer or a function object.

new\_value

Value to assign to replaced values.

The ranges shall not overlap.

**Return value**

An iterator pointing to the element that follows the last element written in the result sequence.

**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 | *// replace\_copy\_if example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::replace\_copy\_if*  *#include <vector> // std::vector*  *bool* IsOdd (*int* i) { *return* ((i%2)==1); }  *int* main () {  std::vector<*int*> foo,bar;  *// set some values:*  *for* (*int* i=1; i<10; i++) foo.push\_back(i); *// 1 2 3 4 5 6 7 8 9*  bar.resize(foo.size()); *// allocate space*  std::replace\_copy\_if (foo.begin(), foo.end(), bar.begin(), IsOdd, 0);  *// 0 2 0 4 0 6 0 8 0*  std::cout << "bar contains:";  *for* (std::vector<*int*>::iterator it=bar.begin(); it!=bar.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| second contains: 0 2 0 4 0 6 0 8 0 |

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Applies *pred* and performs an assignment for each element.

**Data races**

The objects in the range [first,last) are accessed.  
The objects in the range between *result* and the returned value are modified.

**Exceptions**

Throws if any of *pred*, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::fill**

template <class ForwardIterator, class T>

void fill (ForwardIterator first, ForwardIterator last, const T& val);

Fill range with value

Assigns *val* to all the elements in the range [first,last).  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | *template* <*class* ForwardIterator, *class* T>  *void* fill (ForwardIterator first, ForwardIterator last, *const* T& val)  {  *while* (first != last) {  \*first = val;  ++first;  }  } |

**Parameters**

first, last

[Forward iterators](http://www.cplusplus.com/ForwardIterator) to the initial and final positions in a sequence of elements that support being assigned a value of type T. The range filled 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 assign to the elements in the filled range.

**Return value**

none

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | *// fill algorithm example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::fill*  *#include <vector> // std::vector*  *int* main () {  std::vector<*int*> myvector (8); *// myvector: 0 0 0 0 0 0 0 0*  std::fill (myvector.begin(),myvector.begin()+4,5); *// myvector: 5 5 5 5 0 0 0 0*  std::fill (myvector.begin()+3,myvector.end()-2,8); *// myvector: 5 5 5 8 8 8 0 0*  std::cout << "myvector contains:";  *for* (std::vector<*int*>::iterator it=myvector.begin(); it!=myvector.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| myvector contains: 5 5 5 8 8 8 0 0 |

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Assigns a value to each element.

**Data races**

The objects in the range [first,last) are modified (each object is accessed exactly once).

**Exceptions**

Throws if either an element assignment or an operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::fill\_n**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

template <class OutputIterator, class Size, class T>

void fill\_n (OutputIterator first, Size n, const T& val);

Fill sequence with value

Assigns *val* to the first *n* elements of the sequence pointed by *first*.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | *template* <*class* OutputIterator, *class* Size, *class* T>  OutputIterator fill\_n (OutputIterator first, Size n, *const* T& val)  {  *while* (n>0) {  \*first = val;  ++first; --n;  }  *return* first; *// since C++11*  } |

**Parameters**

first

[Output iterators](http://www.cplusplus.com/OutputIterator) to the initial position in a sequence of at least *n* elements that support being assigned a value of type T.

n

Number of elements to fill.

* [C++98](javascript:switch2.select(1))
* [C++11](javascript:switch2.select(2))

This value shall not be negative.

Size shall be (convertible to) an integral type.

val

Value to be used to fill the range.

**Return value**

* [C++98](javascript:switch3.select(1))
* [C++11](javascript:switch3.select(2))

none

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | *// fill\_n example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::fill\_n*  *#include <vector> // std::vector*  *int* main () {  std::vector<*int*> myvector (8,10); *// myvector: 10 10 10 10 10 10 10 10*  std::fill\_n (myvector.begin(),4,20); *// myvector: 20 20 20 20 10 10 10 10*  std::fill\_n (myvector.begin()+3,3,33); *// myvector: 20 20 20 33 33 33 10 10*  std::cout << "myvector contains:";  *for* (std::vector<*int*>::iterator it=myvector.begin(); it!=myvector.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| myvector contains: 20 20 20 33 33 33 10 10 |

**Complexity**

Linear in *n*: Assigns a value to each element.

**Data races**

The *n* first objects at the range pointed by *first* are modified (each object is modified exactly once).

**Exceptions**

Throws if either an element assignment or an operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::generate**

template <class ForwardIterator, class Generator>

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

Generate values for range with function

Assigns the value returned by successive calls to *gen* to the elements in the range [first,last).  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | *template* <*class* ForwardIterator, *class* Generator>  *void* generate ( ForwardIterator first, ForwardIterator last, Generator gen )  {  *while* (first != last) {  \*first = gen();  ++first;  }  } |

**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);  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;  } |

A possible output:

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

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Calls *gen* and performs an assignment for each element.

**Data races**

The objects in the range [first,last) are modified (each object is accessed exactly once).

**Exceptions**

Throws if any of *gen*, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::generate\_n**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

template <class OutputIterator, class Size, class Generator>

void generate\_n (OutputIterator first, Size n, Generator gen);

Generate values for sequence with function

Assigns the value returned by successive calls to *gen* to the first *n* elements of the sequence pointed by *first*.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | *template* <*class* OutputIterator, *class* Size, *class* Generator>  *void* generate\_n ( OutputIterator first, Size n, Generator gen )  {  *while* (n>0) {  \*first = gen();  ++first; --n;  }  } |

**Parameters**

first

Output iterators to the initial positions in a sequence of at least *n* elements that support being assigned a value of the type returned by *gen*.

n

Number of values to generate.

* [C++98](javascript:switch2.select(1))
* [C++11](javascript:switch2.select(2))

This value shall not be negative.

Size shall be (convertible to) an integral type.

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**

* [C++98](javascript:switch3.select(1))
* [C++11](javascript:switch3.select(2))

none

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | *// generate\_n example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::generate\_n*  *int* current = 0;  *int* UniqueNumber () { *return* ++current; }  *int* main () {  *int* myarray[9];  std::generate\_n (myarray, 9, UniqueNumber);  std::cout << "myarray contains:";  *for* (*int* i=0; i<9; ++i)  std::cout << ' ' << myarray[i];  std::cout << '\n';  *return* 0;  } |

A possible output:

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

**Complexity**

Linear in *n*: Calls *gen* and performs an assignment for each element.

**Data races**

The *n* first objects at the range pointed by *first* are modified (each object is modified exactly once).

**Exceptions**

Throws if any of *gen*, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::remove**

template <class ForwardIterator, class T>

ForwardIterator remove (ForwardIterator first, ForwardIterator last, const T& val);

Remove value from range

[Note: This is the reference for algorithm remove. See [remove](http://www.cplusplus.com/cstdio:remove) for [<cstdio>](http://www.cplusplus.com/%3Ccstdio%3E)'s remove.]  
  
Transforms the range [first,last) into a range with all the elements that compare equal to *val* removed, and returns an iterator to the new end of that range.  
  
The function cannot alter the properties of the object containing the range of elements (i.e., it cannot alter the size of an array or a container): The removal is done by replacing the elements that compare equal to *val* by the next element that does not, and signaling the new size of the shortened range by returning an iterator to the element that should be considered its new *past-the-end* element.  
  
The relative order of the elements not removed is preserved, while the elements between the returned iterator and *last* are left in a valid but unspecified state.  
  
The function uses operator== to compare the individual elements to *val*.

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 | *template* <*class* ForwardIterator, *class* T>  ForwardIterator remove (ForwardIterator first, ForwardIterator last, *const* T& val)  {  ForwardIterator result = first;  *while* (first!=last) {  *if* (!(\*first == val)) {  \*result = \*first;  ++result;  }  ++first;  }  *return* result;  } |

**Parameters**

first, last

[Forward iterators](http://www.cplusplus.com/ForwardIterator) to the initial and final positions in a sequence of [move-assignable](http://www.cplusplus.com/is_move_assignable) elements supporting being compared to 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*.

val

Value to be removed.

**Return value**

An iterator to the element that follows the last element not removed.  
The range between *first* and this iterator includes all the elements in the sequence that do not compare equal to *val*.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | *// remove algorithm example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::remove*  *int* main () {  *int* myints[] = {10,20,30,30,20,10,10,20}; *// 10 20 30 30 20 10 10 20*  *// bounds of range:*  *int*\* pbegin = myints; *// ^*  *int*\* pend = myints+*sizeof*(myints)/*sizeof*(*int*); *// ^ ^*  pend = std::remove (pbegin, pend, 20); *// 10 30 30 10 10 ? ? ?*  *// ^ ^*  std::cout << "range contains:";  *for* (*int*\* p=pbegin; p!=pend; ++p)  std::cout << ' ' << \*p;  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| range contains: 10 30 30 10 10 |

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Compares each element, and possibly performs assignments on some of them.

**Data races**

The objects in the range [first,last) are accessed and potentially modified.

**Exceptions**

Throws if any of the element comparisons, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::remove\_if**

template <class ForwardIterator, class UnaryPredicate>

ForwardIterator remove\_if (ForwardIterator first, ForwardIterator last,

UnaryPredicate pred);

Remove elements from range

Transforms the range [first,last) into a range with all the elements for which *pred* returns true removed, and returns an iterator to the new end of that range.  
  
The function cannot alter the properties of the object containing the range of elements (i.e., it cannot alter the size of an array or a container): The removal is done by replacing the elements for which *pred* returns true by the next element for which it does not, and signaling the new size of the shortened range by returning an iterator to the element that should be considered its new *past-the-end* element.  
  
The relative order of the elements not removed is preserved, while the elements between the returned iterator and *last* are left in a valid but unspecified state.

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | *template* <*class* ForwardIterator, *class* UnaryPredicate>  ForwardIterator remove\_if (ForwardIterator first, ForwardIterator last,  UnaryPredicate pred)  {  ForwardIterator result = first;  *while* (first!=last) {  *if* (!pred(\*first)) {  \*result = \*first;  ++result;  }  ++first;  }  *return* result;  } |

**Parameters**

first, last

[Forward iterators](http://www.cplusplus.com/ForwardIterator) to the initial and final positions in a sequence of [*move-assignable*](http://www.cplusplus.com/is_move_assignable) elements. 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*.

pred

Unary function that accepts an element in the range as argument, and returns a value convertible to bool. The value returned indicates whether the element is to be removed (if true, it is removed).  
The function shall not modify its argument.  
This can either be a function pointer or a function object.

**Return value**

An iterator to the element that follows the last element not removed.  
The range between *first* and this iterator includes all the elements in the sequence for which *pred* does not return true.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | *// remove\_if example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::remove\_if*  *bool* IsOdd (*int* i) { *return* ((i%2)==1); }  *int* main () {  *int* myints[] = {1,2,3,4,5,6,7,8,9}; *// 1 2 3 4 5 6 7 8 9*  *// bounds of range:*  *int*\* pbegin = myints; *// ^*  *int*\* pend = myints+*sizeof*(myints)/*sizeof*(*int*); *// ^ ^*  pend = std::remove\_if (pbegin, pend, IsOdd); *// 2 4 6 8 ? ? ? ? ?*  *// ^ ^*  std::cout << "the range contains:";  *for* (*int*\* p=pbegin; p!=pend; ++p)  std::cout << ' ' << \*p;  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| the range contains: 2 4 6 8 |

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Applies *pred* to each element, and possibly performs assignments on some of them.

**Data races**

The objects in the range [first,last) are accessed and potentially modified.

**Exceptions**

throws if any of *pred*, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::remove\_copy**

template <class InputIterator, class OutputIterator, class T>

OutputIterator remove\_copy (InputIterator first, InputIterator last,

OutputIterator result, const T& val);

Copy range removing value

Copies the elements in the range [first,last) to the range beginning at *result*, except those elements that compare equal to *val*.  
  
The resulting range is shorter than [first,last) by as many elements as matches in the sequence, which are "removed".  
  
The function uses operator== to compare the individual elements to *val*.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 | *template* <*class* InputIterator, *class* OutputIterator, *class* T>  OutputIterator remove\_copy (InputIterator first, InputIterator last,  OutputIterator result, *const* T& val)  {  *while* (first!=last) {  *if* (!(\*first == val)) {  \*result = \*first;  ++result;  }  ++first;  }  *return* result;  } |

**Parameters**

first, last

[Forward iterators](http://www.cplusplus.com/ForwardIterator) to the initial and final positions in a sequence of elements supporting being compared to 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*.

result

[Output iterator](http://www.cplusplus.com/OutputIterator) to the initial position of the range where the resulting sequence is stored.  
The pointed type shall support being assigned the value of an element in the range [first,last).

val

Value to be removed.

The ranges shall not overlap.

**Return value**

An iterator pointing to the end of the copied range, which includes all the elements in [first,last) except those that compare equal to *val*.

**Example**

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

Output:

|  |
| --- |
| myvector contains: 10 30 30 10 10 0 0 0 |

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Compares each element, and performs an assignment operation for those not removed.

**Data races**

The objects in the range [first,last) are accessed.  
The objects in the range between *result* and the returned value are modified.

**Exceptions**

Throws if any of the element comparisons, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::remove\_copy\_if**

template <class InputIterator, class OutputIterator, class UnaryPredicate>

OutputIterator remove\_copy\_if (InputIterator first, InputIterator last,

OutputIterator result, UnaryPredicate pred);

Copy range removing values

Copies the elements in the range [first,last) to the range beginning at *result*, except those elements for which *pred* returns true.  
  
The resulting range is shorter than [first,last) by as many elements as matches, which are "removed".  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 | *template* <*class* InputIterator, *class* OutputIterator, *class* UnaryPredicate>  OutputIterator remove\_copy\_if (InputIterator first, InputIterator last,  OutputIterator result, UnaryPredicate pred)  {  *while* (first!=last) {  *if* (!pred(\*first)) {  \*result = \*first;  ++result;  }  ++first;  }  *return* result;  } |

**Parameters**

first, last

[Forward iterators](http://www.cplusplus.com/ForwardIterator) 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*.

result

[Output iterator](http://www.cplusplus.com/OutputIterator) to the initial position of the range where the resulting sequence is stored.  
The pointed type shall support being assigned the value of an element in the range [first,last).

pred

Unary function that accepts an element in the range as argument, and returns a value convertible to bool. The value returned indicates whether the element is to be removed from the copy (if true, it is not copied).  
The function shall not modify its argument.  
This can either be a function pointer or a function object.

The ranges shall not overlap.

**Return value**

An iterator pointing to the end of the copied range, which includes all the elements in [first,last) except those for which *pred* returns true.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | *// remove\_copy\_if example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::remove\_copy\_if*  *#include <vector> // std::vector*  *bool* IsOdd (*int* i) { *return* ((i%2)==1); }  *int* main () {  *int* myints[] = {1,2,3,4,5,6,7,8,9};  std::vector<*int*> myvector (9);  std::remove\_copy\_if (myints,myints+9,myvector.begin(),IsOdd);  std::cout << "myvector contains:";  *for* (std::vector<*int*>::iterator it=myvector.begin(); it!=myvector.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| myvector contains: 2 4 6 8 0 0 0 0 0 |

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Applies *pred* to each element, and performs an assignment operation for those not removed.

**Data races**

The objects in the range [first,last) are accessed.  
The objects in the range between *result* and the returned value are modified.

**Exceptions**

Throws if any of *pred*, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::unique**

|  |  |
| --- | --- |
| **equality (1)** | template <class ForwardIterator>  ForwardIterator unique (ForwardIterator first, ForwardIterator last); |
| **predicate (2)** | template <class ForwardIterator, class BinaryPredicate>  ForwardIterator unique (ForwardIterator first, ForwardIterator last,  BinaryPredicate pred); |

Remove consecutive duplicates in range

Removes all but the first element from every consecutive group of equivalent elements in the range [first,last).  
  
The function cannot alter the properties of the object containing the range of elements (i.e., it cannot alter the size of an array or a container): The removal is done by replacing the duplicate elements by the next element that is not a duplicate, and signaling the new size of the shortened range by returning an iterator to the element that should be considered its new *past-the-end* element.  
  
The relative order of the elements not removed is preserved, while the elements between the returned iterator and *last* are left in a valid but unspecified state.  
  
The function uses operator== to compare the pairs of elements (or *pred*, in version *(2)*).  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 | *template* <*class* ForwardIterator>  ForwardIterator unique (ForwardIterator first, ForwardIterator last)  {  *if* (first==last) *return* last;  ForwardIterator result = first;  *while* (++first != last)  {  *if* (!(\*result == \*first)) *// or: if (!pred(\*result,\*first)) for version (2)*  \*(++result)=\*first;  }  *return* ++result;  } |

**Parameters**

first, last

[Forward iterators](http://www.cplusplus.com/ForwardIterator) to the initial and final positions of the sequence of [*move-assignable*](http://www.cplusplus.com/is_move_assignable) elements. 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*.

pred

Binary function that accepts two elements in the range as argument, and returns a value convertible to bool. The value returned indicates whether both arguments are considered equivalent (if true, they are equivalent and one of them is removed).  
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 element that follows the last element not removed.  
The range between *first* and this iterator includes all the elements in the sequence that were not considered duplicates.

**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;  } |

Output:

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

**Complexity**

For non-empty ranges, linear in one less than the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Compares each pair of elements, and possibly performs assignments on some of them.

**Data races**

The objects in the range [first,last) are accessed and potentially modified.

**Exceptions**

Throws if any of *pred*, the element comparisons, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::unique\_copy**

|  |  |
| --- | --- |
| **equality (1)** | template <class InputIterator, class OutputIterator>  OutputIterator unique\_copy (InputIterator first, InputIterator last,  OutputIterator result); |
| **predicate (2)** | template <class InputIterator, class OutputIterator, class BinaryPredicate>  OutputIterator unique\_copy (InputIterator first, InputIterator last,  OutputIterator result, BinaryPredicate pred); |

Copy range removing duplicates

Copies the elements in the range [first,last) to the range beginning at *result*, except consecutive duplicates (elements that compare equal to the element preceding).  
  
Only the first element from every consecutive group of equivalent elements in the range [first,last) is copied.  
  
The comparison between elements is performed by either applying operator==, or the template parameter *pred* (for the second version) between them.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | *template* <*class* InputIterator, *class* OutputIterator>  OutputIterator unique\_copy (InputIterator first, InputIterator last,  OutputIterator result)  {  *if* (first==last) *return* result;  \*result = \*first;  *while* (++first != last) {  *typename* iterator\_traits<InputIterator>::value\_type val = \*first;  *if* (!(\*result == val)) *// or: if (!pred(\*result,val)) for version (2)*  \*(++result)=val;  }  *return* ++result;  } |

**Parameters**

first, last

[Forward iterators](http://www.cplusplus.com/ForwardIterator) 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*.  
If InputIterator is a *single-pass iterator*, the type it points to shall be [*copy-constructible*](http://www.cplusplus.com/is_copy_constructible) and [*copy-assignable*](http://www.cplusplus.com/is_copy_assignable).

result

Output iterator to the initial position of the range where the resulting range of values is stored.  
The pointed type shall support being assigned the value of an element in the range [first,last).

pred

Binary function that accepts two elements in the range as argument, and returns a value convertible to bool. The value returned indicates whether both arguments are considered equivalent (if true, they are equivalent and one of them is removed).  
The function shall not modify any of its arguments.  
This can either be a function pointer or a function object.

The ranges shall not overlap.

**Return value**

An iterator pointing to the end of the copied range, which contains no consecutive duplicates.

**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 | *// unique\_copy example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::unique\_copy, std::sort, 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};  std::vector<*int*> myvector (9); *// 0 0 0 0 0 0 0 0 0*  *// using default comparison:*  std::vector<*int*>::iterator it;  it=std::unique\_copy (myints,myints+9,myvector.begin()); *// 10 20 30 20 10 0 0 0 0*  *// ^*  std::sort (myvector.begin(),it); *// 10 10 20 20 30 0 0 0 0*  *// ^*  *// using predicate comparison:*  it=std::unique\_copy (myvector.begin(), it, myvector.begin(), myfunction);  *// 10 20 30 20 30 0 0 0 0*  *// ^*  myvector.resize( std::distance(myvector.begin(),it) ); *// 10 20 30*  *// print out content:*  std::cout << "myvector contains:";  *for* (it=myvector.begin(); it!=myvector.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  *return* 0;  } |

Output:

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

**Complexity**

Up to linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Compares each pair of elements, and performs an assignment operation for those elements not matching.

**Data races**

The objects in the range [first,last) are accessed.  
The objects in the range between *result* and the returned value are modified.

**Exceptions**

Throws if any of *pred*, the element comparisons, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::reverse**

template <class BidirectionalIterator>

void reverse (BidirectionalIterator first, BidirectionalIterator last);

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.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | *template* <*class* BidirectionalIterator>  *void* reverse (BidirectionalIterator first, BidirectionalIterator last)  {  *while* ((first!=last)&&(first!=--last)) {  std::iter\_swap (first,last);  ++first;  }  } |

**Parameters**

first, last

[Bidirectional iterators](http://www.cplusplus.com/BidirectionalIterator) to the initial and final positions of the sequence to be reversed. 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.

**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 | *// 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;  } |

Output:

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

**Complexity**

Linear in half the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Swaps elements.

**Data races**

The objects in the range [first,last) are modified.

**Exceptions**

Throws if either an element swap or an operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::reverse\_copy**

template <class BidirectionalIterator, class OutputIterator>

OutputIterator reverse\_copy (BidirectionalIterator first,

BidirectionalIterator last, OutputIterator result);

Copy range reversed

Copies the elements in the range [first,last) to the range beginning at *result*, but in reverse order.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | *template* <*class* BidirectionalIterator, *class* OutputIterator>  OutputIterator reverse\_copy (BidirectionalIterator first,  BidirectionalIterator last, OutputIterator result)  {  *while* (first!=last) {  --last;  \*result = \*last;  ++result;  }  *return* result;  } |

**Parameters**

first, last

[Bidirectional iterators](http://www.cplusplus.com/BidirectionalIterator) to the initial and final positions of the 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 of the range where the reversed range is stored.  
The pointed type shall support being assigned the value of an element in the range [first,last).

The ranges shall not overlap.

**Return value**

An output iterator pointing to the end of the copied range, which contains the same elements in reverse order.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | *// reverse\_copy example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::reverse\_copy*  *#include <vector> // std::vector*  *int* main () {  *int* myints[] ={1,2,3,4,5,6,7,8,9};  std::vector<*int*> myvector;  myvector.resize(9); *// allocate space*  std::reverse\_copy (myints, myints+9, myvector.begin());  *// 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;  } |

Output:

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

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Performs an assignment for each element.

**Data races**

The objects in the range [first,last) are accessed.  
The objects in the range between *result* and the returned value are modified.  
Each object is accessed exactly once.

**Exceptions**

Throws if either an element assignment or an operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::rotate**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

template <class ForwardIterator>

void rotate (ForwardIterator first, ForwardIterator middle,

ForwardIterator last);

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.  
  
The behavior of this function template (C++98) is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 | *template* <*class* ForwardIterator>  *void* rotate (ForwardIterator first, ForwardIterator middle,  ForwardIterator last)  {  ForwardIterator next = middle;  *while* (first!=next)  {  swap (\*first++,\*next++);  *if* (next==last) next=middle;  *else* *if* (first==middle) middle=next;  }  } |

**Parameters**

first, last

[Forward iterators](http://www.cplusplus.com/ForwardIterator) to the initial and final positions of the sequence to be rotated left. 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*.  
Notice that in this function these are not consecutive parameters, but the first and the **third**.

middle

[Forward iterator](http://www.cplusplus.com/ForwardIterator) pointing to the element within the range [first,last) that is moved to the first position in the range.

ForwardIterator 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).

**Return value**

* [C++98](javascript:switch2.select(1))
* [C++11](javascript:switch2.select(2))

none

**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;  } |

Output:

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

**Complexity**

Up to linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Swaps (or moves) elements until all elements have been relocated.

**Data races**

The objects in the range [first,last) are modified.

**Exceptions**

Throws if any element swap (or move) throws or if any operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::rotate\_copy**

template <class ForwardIterator, class OutputIterator>

OutputIterator rotate\_copy (ForwardIterator first, ForwardIterator middle,

ForwardIterator last, OutputIterator result);

Copy range rotated left

Copies the elements in the range [first,last) to the range beginning at *result*, but rotating the order of the elements in such a way that the element pointed by *middle* becomes the first element in the resulting range.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 | *template* <*class* ForwardIterator, *class* OutputIterator>  OutputIterator rotate\_copy (ForwardIterator first, ForwardIterator middle,  ForwardIterator last, OutputIterator result)  {  result=std::copy (middle,last,result);  *return* std::copy (first,middle,result);  } |

**Parameters**

first, last

[Forward iterators](http://www.cplusplus.com/ForwardIterator) to the initial and final positions of the range to be copy-rotated. 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*.  
Notice that in this function, these are not consecutive parameters, but the first and **third** ones.

middle

[Forward iterator](http://www.cplusplus.com/ForwardIterator) pointing to the element within the range [first,last) that is copied as the first element in the resulting range.

result

[Output iterator](http://www.cplusplus.com/OutputIterator) to the initial position of the range where the reversed range is stored.  
The pointed type shall support being assigned the value of an element in the range [first,last).

The ranges shall not overlap.

**Return value**

An output iterator pointing to the end of the copied range.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | *// rotate\_copy algorithm example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::rotate\_copy*  *#include <vector> // std::vector*  *int* main () {  *int* myints[] = {10,20,30,40,50,60,70};  std::vector<*int*> myvector (7);  std::rotate\_copy(myints,myints+3,myints+7,myvector.begin());  *// 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;  } |

Output:

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

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Performs an assignment for each element.

**Data races**

The objects in the range [first,last) are accessed.  
The objects in the range between *result* and the returned value are modified.  
Each object is accessed exactly once.

**Exceptions**

Throws if either an element assignment or an operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::random\_shuffle**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

|  |  |
| --- | --- |
| **generator by default (1)** | template <class RandomAccessIterator>  void random\_shuffle (RandomAccessIterator first, RandomAccessIterator last); |
| **specific generator (2)** | template <class RandomAccessIterator, class RandomNumberGenerator>  void random\_shuffle (RandomAccessIterator first, RandomAccessIterator last,  RandomNumberGenerator& gen); |

Randomly rearrange elements in range

Rearranges the elements in the range [first,last) randomly.  
  
The function swaps the value of each element with that of some other randomly picked element. When provided, the function *gen* determines which element is picked in every case. Otherwise, the function uses some unspecified source of randomness.  
  
To specify a *uniform random generator* as those defined in [<random>](http://www.cplusplus.com/%3Crandom%3E), see [shuffle](http://www.cplusplus.com/shuffle).  
  
The behavior of this function template *(2)* is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 | *template* <*class* RandomAccessIterator, *class* RandomNumberGenerator>  *void* random\_shuffle (RandomAccessIterator first, RandomAccessIterator last,  RandomNumberGenerator& gen)  {  iterator\_traits<RandomAccessIterator>::difference\_type i, n;  n = (last-first);  *for* (i=n-1; i>0; --i) {  swap (first[i],first[gen(i+1)]);  }  } |

**Parameters**

first, last

[Random-access iterators](http://www.cplusplus.com/RandomAccessIterator) to the initial and final positions of the sequence to be shuffled. 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*.

gen

Unary function taking one argument and returning a value, both convertible to/from the appropriate [difference type](http://www.cplusplus.com/iterator_traits::difference_type) used by the iterators. The function shall return a non-negative value less than its argument.  
This can either be a function pointer or a function object.

RandomAccessIterator shall point to a type for which [swap](http://www.cplusplus.com/swap) is defined and swaps the value of its arguments.

**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 | *// random\_shuffle example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::random\_shuffle*  *#include <vector> // std::vector*  *#include <ctime> // std::time*  *#include <cstdlib> // std::rand, std::srand*  *// random generator function:*  *int* myrandom (*int* i) { *return* std::rand()%i;}  *int* main () {  std::srand ( *unsigned* ( std::time(0) ) );  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*  *// using built-in random generator:*  std::random\_shuffle ( myvector.begin(), myvector.end() );  *// using myrandom:*  std::random\_shuffle ( myvector.begin(), myvector.end(), myrandom);  *// 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;  } |

Possible output:

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

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last* minus one: Obtains random values and swaps elements.

**Data races**

The objects in the range [first,last) are modified.

**Exceptions**

Throws if any of the random number generations, the element swaps or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::shuffle**

template <class RandomAccessIterator, class URNG>

void shuffle (RandomAccessIterator first, RandomAccessIterator last, URNG&& g);

Randomly rearrange elements in range using generator

Rearranges the elements in the range [first,last) randomly, using *g* as *uniform random number generator*.  
  
The function swaps the value of each element with that of some other randomly picked element. The function determines the element picked by calling g().  
  
This function works with standard generators as those defined in [<random>](http://www.cplusplus.com/%3Crandom%3E). To shuffle the elements of the range without such a generator, see [random\_shuffle](http://www.cplusplus.com/random_shuffle) instead.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | *template* <*class* RandomAccessIterator, *class* URNG>  *void* shuffle (RandomAccessIterator first, RandomAccessIterator last, URNG&& g)  {  *for* (*auto* i=(last-first)-1; i>0; --i) {  std::uniform\_int\_distribution<*decltype*(i)> d(0,i);  swap (first[i], first[d(g)]);  }  } |

**Parameters**

first, last

[Forward iterators](http://www.cplusplus.com/ForwardIterator) to the initial and final positions of the sequence to be shuffled. 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*.  
ForwardIterator shall point to a type for which [swap](http://www.cplusplus.com/swap) is defined and swaps the value of its arguments.

g

A uniform random number generator, used as the source of randomness.  
URNG shall be a *uniform random number generator*, such as one of the standard generator classes (see [<random>](http://www.cplusplus.com/%3Crandom%3E) for more information).

**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 | *// shuffle algorithm example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::move\_backward*  *#include <array> // std::array*  *#include <random> // std::default\_random\_engine*  *#include <chrono> // std::chrono::system\_clock*  *int* main () {  std::array<*int*,5> foo {1,2,3,4,5};  *// obtain a time-based seed:*  *unsigned* seed = std::chrono::system\_clock::now().time\_since\_epoch().count();  shuffle (foo.begin(), foo.end(), std::default\_random\_engine(seed));  std::cout << "shuffled elements:";  *for* (*int*& x: foo) std::cout << ' ' << x;  std::cout << '\n';  *return* 0;  } |

Possible output:

|  |
| --- |
| shuffled elements: 3 1 4 2 5 |

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last* minus one: Obtains random values and swaps elements.

**Data races**

The objects in the range [first,last) are modified.

**Exceptions**

Throws if any of the random number generations, the element swaps or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::is\_partitioned**

template <class InputIterator, class UnaryPredicate>

bool is\_partitioned (InputIterator first, InputIterator last, UnaryPredicate pred);

Test whether range is partitioned

Returns true if all the elements in the range [first,last) for which *pred* returns true precede those for which it returns false.  
  
If the range is empty, the function returns true.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 | *template* <*class* InputIterator1, *class* InputIterator2>  *bool* is\_partitioned (InputIterator1 first, InputIterator1 last, UnaryPredicate pred)  {  *while* (first!=last && pred(\*first)) {  ++first;  }  *while* (first!=last) {  *if* (pred(\*first)) *return* *false*;  ++first;  }  *return* *true*;  } |

**Parameters**

first, last

[Input iterators](http://www.cplusplus.com/InputIterator) 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*.

pred

Unary function that accepts an element in the range as argument, and returns a value convertible to bool. The value returned indicates whether the element belongs to the first group (if true, the element is expected before all the elements for which it returns false).  
The function shall not modify its argument.  
This can either be a function pointer or a function object.

**Return value**

true if all the elements in the range [first,last) for which *pred* returns true precede those for which it returns false.  
Otherwise it returns false.  
  
If the range is empty, the function 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 24 25 26 27 28 29 | *// is\_partitioned example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::is\_partitioned*  *#include <array> // std::array*  *bool* IsOdd (*int* i) { *return* (i%2)==1; }  *int* main () {  std::array<*int*,7> foo {1,2,3,4,5,6,7};  *// print contents:*  std::cout << "foo:"; *for* (*int*& x:foo) std::cout << ' ' << x;  *if* ( std::is\_partitioned(foo.begin(),foo.end(),IsOdd) )  std::cout << " (partitioned)\n";  *else*  std::cout << " (not partitioned)\n";  *// partition array:*  std::partition (foo.begin(),foo.end(),IsOdd);  *// print contents again:*  std::cout << "foo:"; *for* (*int*& x:foo) std::cout << ' ' << x;  *if* ( std::is\_partitioned(foo.begin(),foo.end(),IsOdd) )  std::cout << " (partitioned)\n";  *else*  std::cout << " (not partitioned)\n";  *return* 0;  } |

Possible output:

|  |
| --- |
| foo: 1 2 3 4 5 6 7 (not partitioned)  foo: 1 7 3 5 4 6 2 (partitioned) |

**Complexity**

Up to linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Calls *pred* for each element until a mismatch is found.

**Data races**

Some (or all) of the objects in the range [first,last) are accessed (once at most).

**Exceptions**

Throws if either *pred* or an operation on an iterator throws.  
Note that invalid parameters cause *undefined behavior*.

function template

<algorithm>

**std::partition**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

template <class BidirectionalIterator, class UnaryPredicate>

BidirectionalIterator partition (BidirectionalIterator first,

BidirectionalIterator last, UnaryPredicate pred);

Partition range in two

Rearranges the elements from the range [first,last), in such a way that all the elements for which *pred* returns true precede 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.  
  
The behavior of this function template (C++98) is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | *template* <*class* BidirectionalIterator, *class* UnaryPredicate>  BidirectionalIterator partition (BidirectionalIterator first,  BidirectionalIterator last, UnaryPredicate pred)  {  *while* (first!=last) {  *while* (pred(\*first)) {  ++first;  *if* (first==last) *return* first;  }  *do* {  --last;  *if* (first==last) *return* first;  } *while* (!pred(\*last));  swap (\*first,\*last);  ++first;  }  *return* first;  } |

**Parameters**

first, last

* [C++98](javascript:switch2.select(1))
* [C++11](javascript:switch2.select(2))

[Bidirectional iterators](http://www.cplusplus.com/BidirectionalIterator) to the initial and final positions of the sequence to partition. The range used is [first,last), which contains all the elements between *first* and *last1*, including the element pointed by *first* but not the element pointed by *last*.

pred

Unary function that accepts an element in the range as argument, and returns a value convertible to bool. The value returned indicates whether the element is to be placed before (if true, it is placed before all the elements for which it returns false).  
The function shall not modify its argument.  
This can either be a function pointer or a function object.

**Return value**

An iterator that points to the first element of the second group of elements (those for which *pred* returns false), or *last* if this group is empty.

**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;  } |

Possible output:

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

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Applies *pred* to each element, and possibly swaps some of them (if the iterator type is a [bidirectional](http://www.cplusplus.com/BidirectionalIterator), at most half that many swaps, otherwise at most that many).

**Data races**

The objects in the range [first,last) are modified.

**Exceptions**

Throws if either an element swap or an operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::stable\_partition**

template <class BidirectionalIterator, class UnaryPredicate>

BidirectionalIterator stable\_partition (BidirectionalIterator first,

BidirectionalIterator last,

UnaryPredicate pred);

Partition range in two - stable ordering

Rearranges the elements in the range [first,last), in such a way that all the elements for which *pred* returns true precede all those for which it returns false, and, unlike function [partition](http://www.cplusplus.com/partition), the relative order of elements within each group is preserved.  
  
This is generally implemented using an internal temporary buffer.

**Parameters**

first, last

[Bidirectional iterators](http://www.cplusplus.com/BidirectionalIterator) to the initial and final positions of the sequence to partition. 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 defined (and swaps the value of its arguments) and which is both [*move-constructible*](http://www.cplusplus.com/is_move_constructible) and [*move-assignable*](http://www.cplusplus.com/is_move_assignable).

pred

Unary function that accepts an element in the range as argument, and returns a value convertible to bool. The value returned indicates whether the element is to be placed before (if true, it is placed before all the elements for which it returns false).  
The function shall not modify its argument.  
This can either be a function pointer or a function object.

**Return value**

An iterator that points to the first element of the second group of elements (those for which *pred* returns false), or *last* if this group is empty.

**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 | *// stable\_partition example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::stable\_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::stable\_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;  } |

Output:

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

**Complexity**

If enough extra memory is available, linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Applies *pred* exactly once to each element, and performs up to that many element moves.  
Otherwise, up to linearithmic: Performs up to N\*log(N) element swaps (where *N* is the distance above). It also applies *pred* exactly once to each element.

**Data races**

The objects in the range [first,last) are modified.

**Exceptions**

Throws if any of the element comparisons, the element swaps (or moves) or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::partition\_copy**

template <class InputIterator, class OutputIterator1,

class OutputIterator2, class UnaryPredicate pred>

pair<OutputIterator1,OutputIterator2>

partition\_copy (InputIterator first, InputIterator last,

OutputIterator1 result\_true, OutputIterator2 result\_false,

UnaryPredicate pred);

Partition range into two

Copies the elements in the range [first,last) for which *pred* returns true into the range pointed by *result\_true*, and those for which it does not into the range pointed by *result\_false*.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | *template* <*class* InputIterator, *class* OutputIterator1,  *class* OutputIterator2, *class* UnaryPredicate pred>  pair<OutputIterator1,OutputIterator2>  partition\_copy (InputIterator first, InputIterator last,  OutputIterator1 result\_true, OutputIterator2 result\_false,  UnaryPredicate pred)  {  *while* (first!=last) {  *if* (pred(\*first)) {  \*result\_true = \*first;  ++result\_true;  }  *else* {  \*result\_false = \*first;  ++result\_false;  }  ++first;  }  *return* std::make\_pair (result\_true,result\_false);  } |

**Parameters**

first, last

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the range to be copy-partitioned. 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\_true

[Output iterator](http://www.cplusplus.com/OutputIterator) to the initial position of the range where the elements for which *pred* returns true are stored.

result\_false

[Output iterator](http://www.cplusplus.com/OutputIterator) to the initial position of the range where the elements for which *pred* returns false are stored.

pred

Unary function that accepts an element pointed by InputIterator as argument, and returns a value convertible to bool. The value returned indicates on which result range the element is copied.  
The function shall not modify its argument.  
This can either be a function pointer or a function object.

The ranges shall not overlap.  
The type pointed by the *output iterator* types shall support being assigned the value of an element in the range [first,last).

**Return value**

A [pair](http://www.cplusplus.com/pair) of iterators with the end of the generated sequences pointed by *result\_true* and *result\_false*, respectivelly.  
  
Its member first points to the element that follows the last element copied to the sequence of elements for which *pred* returned true.  
Its member second points to the element that follows the last element copied to the sequence of elements for which *pred* returned false.

**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 | *// partition\_copy example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::partition\_copy, std::count\_if*  *#include <vector> // std::vector*  *bool* IsOdd (*int* i) { *return* (i%2)==1; }  *int* main () {  std::vector<*int*> foo {1,2,3,4,5,6,7,8,9};  std::vector<*int*> odd, even;  *// resize vectors to proper size:*  *unsigned* n = std::count\_if (foo.begin(), foo.end(), IsOdd);  odd.resize(n); even.resize(foo.size()-n);  *// partition:*  std::partition\_copy (foo.begin(), foo.end(), odd.begin(), even.begin(), IsOdd);  *// print contents:*  std::cout << "odd: "; *for* (*int*& x:odd) std::cout << ' ' << x; std::cout << '\n';  std::cout << "even: "; *for* (*int*& x:even) std::cout << ' ' << x; std::cout << '\n';  *return* 0;  } |

Output:

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

**Complexity**

Linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Calls *pred* and performs an assignment once for each element.

**Data races**

The objects in the range [first,last) are accessed (each exactly once).  
The objects in the ranges pointed by *result\_true* and *result\_false* up to the elements pointed by the iterators returned are modified (each exactly once).

**Exceptions**

Throws if any of *pred*, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::partition\_point**

template <class ForwardIterator, class UnaryPredicate>

ForwardIterator partition\_point (ForwardIterator first, ForwardIterator last,

UnaryPredicate pred);

Get partition point

Returns an iterator to the first element in the partitioned range [first,last) for which *pred* is not true, indicating its partition point.  
  
The elements in the range shall already be partitioned, as if [partition](http://www.cplusplus.com/partition) had been called with the same arguments.  
  
The function optimizes the number of comparisons performed by comparing non-consecutive elements of the sorted range, which is specially efficient for [random-access iterators](http://www.cplusplus.com/RandomAccessIterator).  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | *template* <*class* ForwardIterator, *class* UnaryPredicate>  ForwardIterator partition\_point (ForwardIterator first, ForwardIterator last,  UnaryPredicate pred)  {  *auto* n = distance(first,last);  *while* (n>0)  {  ForwardIterator it = first;  *auto* step = n/2;  std::advance (it,step);  *if* (pred(\*it)) { first=++it; n-=step+1; }  *else* n=step;  }  *return* first;  } |

**Parameters**

first, last

[Forward iterators](http://www.cplusplus.com/ForwardIterator) to the initial and final positions of the partitioned 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*.

pred

Unary function that accepts an element in the range as argument, and returns a value convertible to bool. The value returned indicates whether the element goes before the partition point (if true, it goes before; if false goes at or after it).  
The function shall not modify its argument.  
This can either be a function pointer or a function object.

**Return value**

An iterator to the first element in the partitioned range [first,last) for which *pred* is not true, or *last* if it is not true for any element.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 | *// partition\_point example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::partition, std::partition\_point*  *#include <vector> // std::vector*  *bool* IsOdd (*int* i) { *return* (i%2)==1; }  *int* main () {  std::vector<*int*> foo {1,2,3,4,5,6,7,8,9};  std::vector<*int*> odd;  std::partition (foo.begin(),foo.end(),IsOdd);  *auto* it = std::partition\_point(foo.begin(),foo.end(),IsOdd);  odd.assign (foo.begin(),it);  *// print contents of odd:*  std::cout << "odd:";  *for* (*int*& x:odd) std::cout << ' ' << x;  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| odd: 1 3 5 7 9 |

**Complexity**

On average, logarithmic in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Performs approximately log2(N)+2 element comparisons (where *N* is this distance).  
On *non-*[*random-access*](http://www.cplusplus.com/RandomAccessIterator) *iterators*, the iterator [advances](http://www.cplusplus.com/advance) produce themselves an additional linear complexity in *N* on average.

**Data races**

Some of the objects in the range [first,last) are accessed.

**Exceptions**

Throws if either an element comparison or an operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::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); |

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)).

**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;  } |

Output:

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

**Complexity**

On average, linearithmic in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Performs approximately N\*log2(N) (where *N* is this distance) comparisons of elements, and up to that many element swaps (or moves).

**Data races**

The objects in the range [first,last) are modified.

**Exceptions**

Throws if any of the element comparisons, the element swaps (or moves) or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::stable\_sort**

template <class RandomAccessIterator>

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

template <class RandomAccessIterator, class Compare>

void stable\_sort ( RandomAccessIterator first, RandomAccessIterator last,

Compare comp );

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.

**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;  } |

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 |

**Complexity**

If enough extra memory is available, linearithmic in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Performs up to N\*log2(N) element comparisons (where *N* is this distance), and up to that many element moves.  
Otherwise, polyloglinear in that distance: Performs up to N\*log22(N) element comparisons, and up to that many element swaps.

**Data races**

The objects in the range [first,last) are modified.

**Exceptions**

Throws if any of the element comparisons, the element swaps (or moves) or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::partial\_sort**

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

Partially sort elements in range

Rearranges the elements in the range [first,last), in such a way that the elements before *middle* are the smallest elements in the entire range and are sorted in ascending order, while the remaining elements are left without any specific order.  
  
The elements are compared using operator< for the first version, and *comp* for the second.

**Parameters**

first, last

[Random-access iterators](http://www.cplusplus.com/RandomAccessIterator) to the initial and final positions of the sequence to be partially 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*.  
Notice that in this function these are not consecutive parameters, but the first and the **third**.

middle

[Random-access iterator](http://www.cplusplus.com/RandomAccessIterator) pointing to the element within the range [first,last) that is used as the upper boundary of the elements that are fully sorted.

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.

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).

**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 | *// partial\_sort example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::partial\_sort*  *#include <vector> // std::vector*  *bool* myfunction (*int* i,*int* j) { *return* (i<j); }  *int* main () {  *int* myints[] = {9,8,7,6,5,4,3,2,1};  std::vector<*int*> myvector (myints, myints+9);  *// using default comparison (operator <):*  std::partial\_sort (myvector.begin(), myvector.begin()+5, myvector.end());  *// using function as comp*  std::partial\_sort (myvector.begin(), myvector.begin()+5, myvector.end(),myfunction);  *// 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;  } |

Possible output:

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

**Complexity**

On average, less than linearithmic in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Performs approximately N\*log(M) comparisons of elements (where *N* is this distance, and *M* is the [distance](http://www.cplusplus.com/distance) between *first* and *middle*). It also performs up to that many element swaps (or moves).

**Data races**

The objects in the range [first,last) are modified.

**Exceptions**

Throws if any of the element comparisons, the element swaps (or moves) or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::partial\_sort\_copy**

|  |  |
| --- | --- |
| **default (1)** | template <class InputIterator, class RandomAccessIterator>  RandomAccessIterator  partial\_sort\_copy (InputIterator first,InputIterator last,  RandomAccessIterator result\_first,  RandomAccessIterator result\_last); |
| **custom (2)** | template <class InputIterator, class RandomAccessIterator, class Compare>  RandomAccessIterator  partial\_sort\_copy (InputIterator first,InputIterator last,  RandomAccessIterator result\_first,  RandomAccessIterator result\_last, Compare comp); |

Copy and partially sort range

Copies the smallest elements in the range [first,last) to [result\_first,result\_last), sorting the elements copied. The number of elements copied is the same as the [distance](http://www.cplusplus.com/distance) between *result\_first* and *result\_last* (unless this is more than the amount of elements in [first,last)).  
  
The range [first,last) is not modified.  
  
The elements are compared using operator< for the first version, and *comp* for the second.

**Parameters**

first, last

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the sequence to copy from. 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*.  
InputIterator shall point to a type [assignable](http://www.cplusplus.com/is_assignable) to the elements pointed by RandomAccessIterator.

result\_first, result\_last

[Random-access iterators](http://www.cplusplus.com/RandomAccessIterator) to the initial and final positions of the destination sequence. The range used is [result\_first,result\_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 result 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**

An iterator pointing to the element that follows the last element written in the result sequence.

**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 | *// partial\_sort\_copy example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::partial\_sort\_copy*  *#include <vector> // std::vector*  *bool* myfunction (*int* i,*int* j) { *return* (i<j); }  *int* main () {  *int* myints[] = {9,8,7,6,5,4,3,2,1};  std::vector<*int*> myvector (5);  *// using default comparison (operator <):*  std::partial\_sort\_copy (myints, myints+9, myvector.begin(), myvector.end());  *// using function as comp*  std::partial\_sort\_copy (myints, myints+9, myvector.begin(), myvector.end(), myfunction);  *// 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;  } |

Output:

|  |
| --- |
| myvector contains: 1 2 3 4 5 |

**Complexity**

On average, less than linearithmic in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Performs approximately N\*log(min(N,M)) comparisons of elements (where *N* is this distance, and *M* is the [distance](http://www.cplusplus.com/distance) between *result\_first* and *result\_last*). It also performs up to that many element swaps (or moves) and min(N,M) assignments between ranges.

**Data races**

The objects in the range [first,last) are modified.

**Exceptions**

Throws if any of the element comparisons, the element assignments, the element swaps (or moves) or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::is\_sorted**

|  |  |
| --- | --- |
| **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); |

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.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 | *template* <*class* ForwardIterator>  *bool* is\_sorted (ForwardIterator first, ForwardIterator last)  {  *if* (first==last) *return* *true*;  ForwardIterator next = first;  *while* (++next!=last) {  *if* (\*next<\*first) *// or, if (comp(\*next,\*first)) for version (2)*  *return* *false*;  ++first;  }  *return* *true*;  } |

**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;  } |

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! |

**Complexity**

Up to linear in one less than the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Compares pairs of elements until a mismatch is found.

**Data races**

The objects in the range [first,last) are accessed.

**Exceptions**

Throws if either an element comparison or an operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::is\_sorted\_until**

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

Find first unsorted element in range

Returns an iterator to the first element in the range [first,last) which does not follow an ascending order.  
  
The range between *first* and the iterator returned [is sorted](http://www.cplusplus.com/is_sorted).  
  
If the entire range is sorted, the function returns *last*.  
  
The elements are compared using operator< for the first version, and *comp* for the second.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | *template* <*class* ForwardIterator>  ForwardIterator is\_sorted\_until (ForwardIterator first, ForwardIterator last)  {  *if* (first==last) *return* first;  ForwardIterator next = first;  *while* (++next!=last) {  *if* (\*next<\*first) *return* next;  ++first;  }  *return* last;  } |

**Parameters**

first, last

[Forward iterators](http://www.cplusplus.com/ForwardIterator) to the initial and final positions in a 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**

An iterator to the first element in the range which does not follow an ascending order, or *last* if all elements are sorted or if the range contains less than two 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 | *// is\_sorted\_until example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::is\_sorted\_until, std::prev\_permutation*  *#include <array> // std::array*  *int* main () {  std::array<*int*,4> foo {2,4,1,3};  std::array<*int*,4>::iterator it;  *do* {  *// try a new permutation:*  std::prev\_permutation(foo.begin(),foo.end());  *// print range:*  std::cout << "foo:";  *for* (*int*& x:foo) std::cout << ' ' << x;  it=std::is\_sorted\_until(foo.begin(),foo.end());  std::cout << " (" << (it-foo.begin()) << " elements sorted)\n";  } *while* (it!=foo.end());  std::cout << "the range is sorted!\n";  *return* 0;  } |

Output:

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

**Complexity**

Up to linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Calls *comp* for each element until a mismatch is found.

**Data races**

Some (or all) of the objects in the range [first,last) are accessed (once at most).

**Exceptions**

Throws if either *comp* or an operation on an iterator throws.  
Note that invalid parameters cause *undefined behavior*.

function template

<algorithm>

**std::nth\_element**

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

Sort element in range

Rearranges the elements in the range [first,last), in such a way that the element at the *nth* position is the element that would be in that position in a sorted sequence.  
  
The other elements are left without any specific order, except that none of the elements preceding *nth* are greater than it, and none of the elements following it are less.  
  
The elements are compared using operator< for the first version, and *comp* for the second.

**Parameters**

first, last

[Random-access iterators](http://www.cplusplus.com/RandomAccessIterator) to the initial and final positions of the sequence to be used. 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*.  
Notice that in this function, these are not consecutive parameters, but the first and the **third**.

nth

[Random-access iterator](http://www.cplusplus.com/RandomAccessIterator) pointing to the location within the range [first,last) that will contain the sorted element.  
Notice that the value of the element pointed by *nth* before the call is not relevant.

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 | *// nth\_element example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::nth\_element, std::random\_shuffle*  *#include <vector> // std::vector*  *bool* myfunction (*int* i,*int* j) { *return* (i<j); }  *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::random\_shuffle (myvector.begin(), myvector.end());  *// using default comparison (operator <):*  std::nth\_element (myvector.begin(), myvector.begin()+5, myvector.end());  *// using function as comp*  std::nth\_element (myvector.begin(), myvector.begin()+5, myvector.end(),myfunction);  *// 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;  } |

Possible output:

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

**Complexity**

On average, linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Compares elements, and possibly swaps (or moves) them, until the elements are properly rearranged.

**Data races**

The objects in the range [first,last) are modified.

**Exceptions**

Throws if any of the element comparisons, the element swaps (or moves) or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::lower\_bound**

|  |  |
| --- | --- |
| **default (1)** | template <class ForwardIterator, class T>  ForwardIterator lower\_bound (ForwardIterator first, ForwardIterator last,  const T& val); |
| **custom (2)** | template <class ForwardIterator, class T, class Compare>  ForwardIterator lower\_bound (ForwardIterator first, ForwardIterator last,  const T& val, Compare comp); |

Return iterator to lower bound

Returns an iterator pointing to the first element in the range [first,last) which does not compare less than *val*.  
  
The elements are compared using operator< for the first version, and *comp* for the second. The elements in the range shall already be [sorted](http://www.cplusplus.com/is_sorted) according to this same criterion (operator< or *comp*), or at least [partitioned](http://www.cplusplus.com/is_partitioned) with respect to *val*.  
  
The function optimizes the number of comparisons performed by comparing non-consecutive elements of the sorted range, which is specially efficient for [random-access iterators](http://www.cplusplus.com/RandomAccessIterator).  
  
Unlike [upper\_bound](http://www.cplusplus.com/upper_bound), the value pointed by the iterator returned by this function may also be equivalent to *val*, and not only greater.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 | *template* <*class* ForwardIterator, *class* T>  ForwardIterator lower\_bound (ForwardIterator first, ForwardIterator last, *const* T& val)  {  ForwardIterator it;  iterator\_traits<ForwardIterator>::difference\_type count, step;  count = distance(first,last);  *while* (count>0)  {  it = first; step=count/2; advance (it,step);  *if* (\*it<val) { *// or: if (comp(\*it,val)), for version (2)*  first=++it;  count-=step+1;  }  *else* count=step;  }  *return* first;  } |

**Parameters**

first, last

[Forward iterators](http://www.cplusplus.com/ForwardIterator) to the initial and final positions of a [sorted](http://www.cplusplus.com/is_sorted) (or properly [partitioned](http://www.cplusplus.com/is_partitioned)) 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*.

val

Value of the lower bound to search for in the range.  
For *(1)*, T shall be a type supporting being compared with elements of the range [first,last) as the right-hand side operand of operator<.

comp

Binary function that accepts two arguments (the first of the type pointed by ForwardIterator, and the second, always *val*), and returns a value convertible to bool. The value returned indicates whether the first argument is considered to go before the second.  
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 lower bound of *val* in the range.  
If all the element in the range compare less than *val*, 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 | *// lower\_bound/upper\_bound example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::lower\_bound, std::upper\_bound, std::sort*  *#include <vector> // std::vector*  *int* main () {  *int* myints[] = {10,20,30,30,20,10,10,20};  std::vector<*int*> v(myints,myints+8); *// 10 20 30 30 20 10 10 20*  std::sort (v.begin(), v.end()); *// 10 10 10 20 20 20 30 30*  std::vector<*int*>::iterator low,up;  low=std::lower\_bound (v.begin(), v.end(), 20); *// ^*  up= std::upper\_bound (v.begin(), v.end(), 20); *// ^*  std::cout << "lower\_bound at position " << (low- v.begin()) << '\n';  std::cout << "upper\_bound at position " << (up - v.begin()) << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| lower\_bound at position 3  upper\_bound at position 6 |

**Complexity**

On average, logarithmic in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Performs approximately log2(N)+1 element comparisons (where *N* is this distance).  
On *non-*[*random-access*](http://www.cplusplus.com/RandomAccessIterator) *iterators*, the iterator [advances](http://www.cplusplus.com/advance) produce themselves an additional linear complexity in *N* on average.

**Data races**

The objects in the range [first,last) are accessed.

**Exceptions**

Throws if either an element comparison or an operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::upper\_bound**

|  |  |
| --- | --- |
| **default (1)** | template <class ForwardIterator, class T>  ForwardIterator upper\_bound (ForwardIterator first, ForwardIterator last,  const T& val); |
| **custom (2)** | template <class ForwardIterator, class T, class Compare>  ForwardIterator upper\_bound (ForwardIterator first, ForwardIterator last,  const T& val, Compare comp); |

Return iterator to upper bound

Returns an iterator pointing to the first element in the range [first,last) which compares greater than *val*.  
  
The elements are compared using operator< for the first version, and *comp* for the second. The elements in the range shall already be [sorted](http://www.cplusplus.com/is_sorted) according to this same criterion (operator< or *comp*), or at least [partitioned](http://www.cplusplus.com/is_partitioned) with respect to *val*.  
  
The function optimizes the number of comparisons performed by comparing non-consecutive elements of the sorted range, which is specially efficient for [random-access iterators](http://www.cplusplus.com/RandomAccessIterator).  
  
Unlike [lower\_bound](http://www.cplusplus.com/lower_bound), the value pointed by the iterator returned by this function cannot be equivalent to *val*, only greater.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | *template* <*class* ForwardIterator, *class* T>  ForwardIterator upper\_bound (ForwardIterator first, ForwardIterator last, *const* T& val)  {  ForwardIterator it;  iterator\_traits<ForwardIterator>::difference\_type count, step;  count = std::distance(first,last);  *while* (count>0)  {  it = first; step=count/2; std::advance (it,step);  *if* (!(val<\*it)) *// or: if (!comp(val,\*it)), for version (2)*  { first=++it; count-=step+1; }  *else* count=step;  }  *return* first;  } |

**Parameters**

first, last

[Forward iterators](http://www.cplusplus.com/ForwardIterator) to the initial and final positions of a [sorted](http://www.cplusplus.com/is_sorted) (or properly [partitioned](http://www.cplusplus.com/is_partitioned)) 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*.

val

Value of the upper bound to search for in the range.  
For *(1)*, T shall be a type supporting being compared with elements of the range [first,last) as the left-hand side operand of operator<.

comp

Binary function that accepts two arguments (the first is always *val*, and the second of the type pointed by ForwardIterator), and returns a value convertible to bool. The value returned indicates whether the first argument is considered to go before the second.  
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 upper bound position for *val* in the range.  
If no element in the range compares greater than *val*, 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 | *// lower\_bound/upper\_bound example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::lower\_bound, std::upper\_bound, std::sort*  *#include <vector> // std::vector*  *int* main () {  *int* myints[] = {10,20,30,30,20,10,10,20};  std::vector<*int*> v(myints,myints+8); *// 10 20 30 30 20 10 10 20*  std::sort (v.begin(), v.end()); *// 10 10 10 20 20 20 30 30*  std::vector<*int*>::iterator low,up;  low=std::lower\_bound (v.begin(), v.end(), 20); *// ^*  up= std::upper\_bound (v.begin(), v.end(), 20); *// ^*  std::cout << "lower\_bound at position " << (low- v.begin()) << '\n';  std::cout << "upper\_bound at position " << (up - v.begin()) << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| lower\_bound at position 3  upper\_bound at position 6 |

**Complexity**

On average, logarithmic in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Performs approximately log2(N)+1 element comparisons (where *N* is this distance).  
On *non-*[*random-access*](http://www.cplusplus.com/RandomAccessIterator) *iterators*, the iterator [advances](http://www.cplusplus.com/advance) produce themselves an additional linear complexity in *N* on average.

**Data races**

The objects in the range [first,last) are accessed.

**Exceptions**

Throws if either an element comparison or an operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::equal\_range**

|  |  |
| --- | --- |
| **default (1)** | template <class ForwardIterator, class T>  pair<ForwardIterator,ForwardIterator>  equal\_range (ForwardIterator first, ForwardIterator last, const T& val); |
| **custom (2)** | template <class ForwardIterator, class T, class Compare>  pair<ForwardIterator,ForwardIterator>  equal\_range (ForwardIterator first, ForwardIterator last, const T& val,  Compare comp); |

Get subrange of equal elements

Returns the bounds of the subrange that includes all the elements of the range [first,last) with values equivalent to *val*.  
  
The elements are compared using operator< for the first version, and *comp* for the second. Two elements, a and b are considered equivalent if (!(a<b) && !(b<a)) or if (!comp(a,b) && !comp(b,a)).  
  
The elements in the range shall already be [sorted](http://www.cplusplus.com/is_sorted) according to this same criterion (operator< or *comp*), or at least [partitioned](http://www.cplusplus.com/is_partitioned) with respect to *val*.  
  
If *val* is not equivalent to any value in the range, the subrange returned has a length of zero, with both iterators pointing to the nearest value greater than *val*, if any, or to *last*, if *val* compares greater than all the elements in the range.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 | *template* <*class* ForwardIterator, *class* T>  pair<ForwardIterator,ForwardIterator>  equal\_range (ForwardIterator first, ForwardIterator last, *const* T& val)  {  ForwardIterator it = std::lower\_bound (first,last,val);  *return* std::make\_pair ( it, std::upper\_bound(it,last,val) );  } |

**Parameters**

first, last

[Forward iterators](http://www.cplusplus.com/ForwardIterator) to the initial and final positions of a [sorted](http://www.cplusplus.com/is_sorted) (or properly [partitioned](http://www.cplusplus.com/is_partitioned)) 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*.

val

Value of the subrange to search for in the range.  
For *(1)*, T shall be a type supporting being compared with elements of the range [first,last) as either operand of operator<.

comp

Binary function that accepts two arguments of the type pointed by ForwardIterator (and of type T), and returns a value convertible to bool. The value returned indicates whether the first argument is considered to go before the second.  
The function shall not modify any of its arguments.  
This can either be a function pointer or a function object.

**Return value**

A [pair](http://www.cplusplus.com/pair) object, whose member pair::first is an iterator to the lower bound of the subrange of equivalent values, and pair::second its upper bound.  
The values are the same as those that would be returned by functions [lower\_bound](http://www.cplusplus.com/lower_bound) and [upper\_bound](http://www.cplusplus.com/upper_bound) respectively.

**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 | *// equal\_range example*  *// equal\_range example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::equal\_range, std::sort*  *#include <vector> // std::vector*  *bool* mygreater (*int* i,*int* j) { *return* (i>j); }  *int* main () {  *int* myints[] = {10,20,30,30,20,10,10,20};  std::vector<*int*> v(myints,myints+8); *// 10 20 30 30 20 10 10 20*  std::pair<std::vector<*int*>::iterator,std::vector<*int*>::iterator> bounds;  *// using default comparison:*  std::sort (v.begin(), v.end()); *// 10 10 10 20 20 20 30 30*  bounds=std::equal\_range (v.begin(), v.end(), 20); *// ^ ^*  *// using "mygreater" as comp:*  std::sort (v.begin(), v.end(), mygreater); *// 30 30 20 20 20 10 10 10*  bounds=std::equal\_range (v.begin(), v.end(), 20, mygreater); *// ^ ^*  std::cout << "bounds at positions " << (bounds.first - v.begin());  std::cout << " and " << (bounds.second - v.begin()) << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| bounds at positions 2 and 5 |

**Complexity**

On average, up to twice logarithmic in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Performs approximately 2\*log2(N)+1 element comparisons (where *N* is this distance).  
On *non-*[*random-access*](http://www.cplusplus.com/RandomAccessIterator) *iterators*, the iterator [advances](http://www.cplusplus.com/advance) produce themselves an additional up to twice linear complexity in *N* on average.

**Data races**

The objects in the range [first,last) are accessed.

**Exceptions**

Throws if either an element comparison or an operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.  
function template

<algorithm>

**std::binary\_search**

|  |  |
| --- | --- |
| **default (1)** | template <class ForwardIterator, class T>  bool binary\_search (ForwardIterator first, ForwardIterator last,  const T& val); |
| **custom (2)** | template <class ForwardIterator, class T, class Compare>  bool binary\_search (ForwardIterator first, ForwardIterator last,  const T& val, Compare comp); |

Test if value exists in sorted sequence

Returns true if any element in the range [first,last) is equivalent to *val*, and false otherwise.  
  
The elements are compared using operator< for the first version, and *comp* for the second. Two elements, a and b are considered equivalent if (!(a<b) && !(b<a)) or if (!comp(a,b) && !comp(b,a)).  
  
The elements in the range shall already be [sorted](http://www.cplusplus.com/is_sorted) according to this same criterion (operator< or *comp*), or at least [partitioned](http://www.cplusplus.com/is_partitioned) with respect to *val*.  
  
The function optimizes the number of comparisons performed by comparing non-consecutive elements of the sorted range, which is specially efficient for [random-access iterators](http://www.cplusplus.com/RandomAccessIterator).  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 | *template* <*class* ForwardIterator, *class* T>  *bool* binary\_search (ForwardIterator first, ForwardIterator last, *const* T& val)  {  first = std::lower\_bound(first,last,val);  *return* (first!=last && !(val<\*first));  } |

**Parameters**

first, last

[Forward iterators](http://www.cplusplus.com/ForwardIterator) to the initial and final positions of a [sorted](http://www.cplusplus.com/is_sorted) (or properly [partitioned](http://www.cplusplus.com/is_partitioned)) 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*.

val

Value to search for in the range.  
For *(1)*, T shall be a type supporting being compared with elements of the range [first,last) as either operand of operator<.

comp

Binary function that accepts two arguments of the type pointed by ForwardIterator (and of type T), and returns a value convertible to bool. The value returned indicates whether the first argument is considered to go before the second.  
The function shall not modify any of its arguments.  
This can either be a function pointer or a function object.

**Return value**

true if an element equivalent to *val* is found, and false otherwise.

**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 | *// binary\_search example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::binary\_search, std::sort*  *#include <vector> // std::vector*  *bool* myfunction (*int* i,*int* j) { *return* (i<j); }  *int* main () {  *int* myints[] = {1,2,3,4,5,4,3,2,1};  std::vector<*int*> v(myints,myints+9); *// 1 2 3 4 5 4 3 2 1*  *// using default comparison:*  std::sort (v.begin(), v.end());  std::cout << "looking for a 3... ";  *if* (std::binary\_search (v.begin(), v.end(), 3))  std::cout << "found!\n"; *else* std::cout << "not found.\n";  *// using myfunction as comp:*  std::sort (v.begin(), v.end(), myfunction);  std::cout << "looking for a 6... ";  *if* (std::binary\_search (v.begin(), v.end(), 6, myfunction))  std::cout << "found!\n"; *else* std::cout << "not found.\n";  *return* 0;  } |

Output:

|  |
| --- |
| looking for a 3... found!  looking for a 6... not found. |

**Complexity**

On average, logarithmic in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Performs approximately log2(N)+2 element comparisons (where *N* is this distance).  
On *non-*[*random-access*](http://www.cplusplus.com/RandomAccessIterator) *iterators*, the iterator [advances](http://www.cplusplus.com/advance) produce themselves an additional linear complexity in *N* on average.

**Data races**

The objects in the range [first,last) are accessed.

**Exceptions**

Throws if either an element comparison or an operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::merge**

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

Merge sorted ranges

Combines the elements in the sorted ranges [first1,last1) and [first2,last2), into a new range beginning at *result* with all its elements sorted.  
  
The elements are compared using operator< for the first version, and *comp* for the second. The elements in both ranges shall already be ordered according to this same criterion (operator< or *comp*). The resulting range is also sorted according to this.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | *template* <*class* InputIterator1, *class* InputIterator2, *class* OutputIterator>  OutputIterator merge (InputIterator1 first1, InputIterator1 last1,  InputIterator2 first2, InputIterator2 last2,  OutputIterator result)  {  *while* (*true*) {  *if* (first1==last1) *return* std::copy(first2,last2,result);  *if* (first2==last2) *return* std::copy(first1,last1,result);  \*result++ = (\*first2<\*first1)? \*first2++ : \*first1++;  }  } |

**Parameters**

first1, last1

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the first sorted 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

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the second sorted sequence. The range used is [first2,last2).

result

[Output iterator](http://www.cplusplus.com/OutputIterator) to the initial position of the range where the resulting combined range is stored. Its size is equal to the sum of both ranges above.

comp

Binary function that accepts two arguments of the types pointed by the iterators, 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.

The ranges shall not overlap.  
The elements in both input ranges should be [assignable](http://www.cplusplus.com/is_assignable) to the elements in the range pointed by *result*. They should also be comparable (with operator< for *(1)*, and with *comp* for *(2)*).

**Return value**

An iterator pointing to the *past-the-end* element in the resulting sequence.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 | *// merge algorithm example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::merge, 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);  std::sort (first,first+5);  std::sort (second,second+5);  std::merge (first,first+5,second,second+5,v.begin());  std::cout << "The resulting vector contains:";  *for* (std::vector<*int*>::iterator it=v.begin(); it!=v.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| The resulting vector contains: 5 10 10 15 20 20 25 30 40 50 |

**Complexity**

Up to linear in (1+count1-count2), where *countX* is the [distance](http://www.cplusplus.com/distance) between *firstX* and *lastX*: Compares and assigns all elements.

**Data races**

The objects in the ranges [first1,last1) and [first2,last2)are accessed.  
The objects in the range between *result* and the returned value are modified.

**Exceptions**

Throws if any of element comparisons, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::inplace\_merge**

|  |  |
| --- | --- |
| **default (1)** | template <class BidirectionalIterator>  void inplace\_merge (BidirectionalIterator first, BidirectionalIterator middle,  BidirectionalIterator last); |
| **custom (2)** | template <class BidirectionalIterator, class Compare>  void inplace\_merge (BidirectionalIterator first, BidirectionalIterator middle,  BidirectionalIterator last, Compare comp); |

Merge consecutive sorted ranges

Merges two consecutive sorted ranges: [first,middle) and [middle,last), putting the result into the combined sorted range [first,last).  
  
The elements are compared using operator< for the first version, and *comp* for the second. The elements in both ranges shall already be ordered according to this same criterion (operator< or *comp*). The resulting range is also sorted according to this.  
  
The function preserves the relative order of elements with equivalent values, with the elements in the first range preceding those equivalent in the second.

**Parameters**

first

[Bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) to the initial position in the first sorted sequence to merge. This is also the initial position where the resulting merged range is stored.

middle

[Bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) to the initial position of the second sorted sequence, which because both sequences must be consecutive, matches the *past-the-end* position of the first sequence.

last

[Bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) to the *past-the-end* position of the second sorted sequence. This is also the *past-the-end* position of the range where the resulting merged range is stored.

comp

Binary function that accepts two arguments of the types pointed by the iterators, 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.

BidirectionalIterator 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).

**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 | *// inplace\_merge example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::inplace\_merge, std::sort, std::copy*  *#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);  std::vector<*int*>::iterator it;  std::sort (first,first+5);  std::sort (second,second+5);  it=std::copy (first, first+5, v.begin());  std::copy (second,second+5,it);  std::inplace\_merge (v.begin(),v.begin()+5,v.end());  std::cout << "The resulting vector contains:";  *for* (it=v.begin(); it!=v.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| The resulting vector contains: 5 10 10 15 20 20 25 30 40 50 |

**Complexity**

If enough extra memory is available, linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Performs N-1 comparisons and up to twice that many element moves.  
Otherwise, up to linearithmic: Performs up to N\*log(N) element comparisons (where *N* is the distance above), and up to that many element swaps.

**Data races**

The objects in the range [first,last) are modified.

**Exceptions**

Throws if any of the element comparisons, the element swaps (or moves) or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::includes**

template <class InputIterator1, class InputIterator2>

bool includes ( InputIterator1 first1, InputIterator1 last1,

InputIterator2 first2, InputIterator2 last2 );

template <class InputIterator1, class InputIterator2, class Compare>

bool includes ( InputIterator1 first1, InputIterator1 last1,

InputIterator2 first2, InputIterator2 last2, Compare comp );

Test whether sorted range includes another sorted range

Returns true if the sorted range [first1,last1) contains all the elements in the sorted range [first2,last2).  
  
The elements are compared using operator< for the first version, and *comp* for the second. Two elements, a and b are considered equivalent if (!(a<b) && !(b<a)) or if (!comp(a,b) && !comp(b,a)).  
  
The elements in the range shall already be ordered according to this same criterion (operator< or *comp*).  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | *template* <*class* InputIterator1, *class* InputIterator2>  *bool* includes (InputIterator1 first1, InputIterator1 last1,  InputIterator2 first2, InputIterator2 last2)  {  *while* (first2!=last2) {  *if* ( (first1==last1) || (\*first2<\*first1) ) *return* *false*;  *if* (!(\*first1<\*first2)) ++first2;  ++first1;  }  *return* *true*;  } |

**Parameters**

first1, last1

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the first sorted sequence (which is tested on whether it contains the second 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

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the second sorted sequence (which is tested on whether it is contained in the first sequence). The range used is [first2,last2).

comp

Binary function that accepts two elements as arguments (one from each of the two sequences, in the same order), 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 every element in the range [first2,last2) is contained in the range [first1,last1), false otherwise.

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

If [first2,last2) is an empty range, the result is unspecified.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 | *// includes algorithm example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::includes, std::sort*  *bool* myfunction (*int* i, *int* j) { *return* i<j; }  *int* main () {  *int* container[] = {5,10,15,20,25,30,35,40,45,50};  *int* continent[] = {40,30,20,10};  std::sort (container,container+10);  std::sort (continent,continent+4);  *// using default comparison:*  *if* ( std::includes(container,container+10,continent,continent+4) )  std::cout << "container includes continent!\n";  *// using myfunction as comp:*  *if* ( std::includes(container,container+10,continent,continent+4, myfunction) )  std::cout << "container includes continent!\n";  *return* 0;  } |

Output:

|  |
| --- |
| container includes continent!  container includes continent! |

**Complexity**

Up to linear in twice the [distances](http://www.cplusplus.com/distances) in both ranges: Performs up to 2\*(count1+count2)-1 comparisons (where *countX* is the [distance](http://www.cplusplus.com/distance) between *firstX* and *lastX*).

**Data races**

Some (or all) of the objects in both ranges are accessed (twice each at most).

**Exceptions**

Throws if any element comparison (or call to *comp*) throws or if any of the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::set\_union**

|  |  |
| --- | --- |
| **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); |

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.  
  
The elements are compared using operator< for the first version, and *comp* for the second. Two elements, a and b are considered equivalent if (!(a<b) && !(b<a)) or if (!comp(a,b) && !comp(b,a)).  
  
The elements in the ranges shall already be ordered according to this same criterion (operator< or *comp*). The resulting range is also sorted according to this.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | *template* <*class* InputIterator1, *class* InputIterator2, *class* OutputIterator>  OutputIterator set\_union (InputIterator1 first1, InputIterator1 last1,  InputIterator2 first2, InputIterator2 last2,  OutputIterator result)  {  *while* (*true*)  {  *if* (first1==last1) *return* std::copy(first2,last2,result);  *if* (first2==last2) *return* std::copy(first1,last1,result);  *if* (\*first1<\*first2) { \*result = \*first1; ++first1; }  *else* *if* (\*first2<\*first1) { \*result = \*first2; ++first2; }  *else* { \*result = \*first1; ++first1; ++first2; }  ++result;  }  } |

**Parameters**

first1, last1

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the first sorted 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

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the second sorted sequence. The range used is [first2,last2).

result

[Output iterator](http://www.cplusplus.com/OutputIterator) to the initial position of the range where the resulting sequence is stored.  
The pointed type shall support being assigned the value of an element from the other ranges.

comp

Binary function that accepts two arguments of the types pointed by the input iterators, 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.

The ranges shall not overlap.

**Return value**

An iterator to the end of the constructed range.

**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;  } |

Output:

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

**Complexity**

Up to linear in 2\*(count1+count2)-1 (where *countX* is the [distance](http://www.cplusplus.com/distance) between *firstX* and *lastX*): Compares and assigns elements.

**Data races**

The objects in the ranges [first1,last1) and [first2,last2)are accessed.  
The objects in the range between *result* and the returned value are modified.

**Exceptions**

Throws if any of the element comparisons, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::set\_intersection**

|  |  |
| --- | --- |
| **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); |

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.  
  
The elements are compared using operator< for the first version, and *comp* for the second. Two elements, a and b are considered equivalent if (!(a<b) && !(b<a)) or if (!comp(a,b) && !comp(b,a)).  
  
The elements in the ranges shall already be ordered according to this same criterion (operator< or *comp*). The resulting range is also sorted according to this.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | *template* <*class* InputIterator1, *class* InputIterator2, *class* OutputIterator>  OutputIterator set\_intersection (InputIterator1 first1, InputIterator1 last1,  InputIterator2 first2, InputIterator2 last2,  OutputIterator result)  {  *while* (first1!=last1 && first2!=last2)  {  *if* (\*first1<\*first2) ++first1;  *else* *if* (\*first2<\*first1) ++first2;  *else* {  \*result = \*first1;  ++result; ++first1; ++first2;  }  }  *return* result;  } |

**Parameters**

first1, last1

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the first sorted 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

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the second sorted sequence. The range used is [first2,last2).

result

[Output iterator](http://www.cplusplus.com/OutputIterator) to the initial position of the range where the resulting sequence is stored.  
The pointed type shall support being assigned the value of an element from the first range.

comp

Binary function that accepts two arguments of the types pointed by the input iterators, 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.

The ranges shall not overlap.

**Return value**

An iterator to the end of the constructed range.

**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;  } |

Output:

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

**Complexity**

Up to linear in 2\*(count1+count2)-1 (where *countX* is the [distance](http://www.cplusplus.com/distance) between *firstX* and *lastX*): Compares and assigns elements.

**Data races**

The objects in the ranges [first1,last1) and [first2,last2)are accessed.  
The objects in the range between *result* and the returned value are modified.

**Exceptions**

Throws if any of the element comparisons, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::set\_difference**

|  |  |
| --- | --- |
| **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); |

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.  
  
Notice that this is a directional operation - for a symmetrical equivalent, see [set\_symmetric\_difference](http://www.cplusplus.com/set_symmetric_difference).  
  
The elements are compared using operator< for the first version, and *comp* for the second. Two elements, a and b are considered equivalent if (!(a<b) && !(b<a)) or if (!comp(a,b) && !comp(b,a)).  
  
The elements in the ranges shall already be ordered according to this same criterion (operator< or *comp*). The resulting range is also sorted according to this.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 | *template* <*class* InputIterator1, *class* InputIterator2, *class* OutputIterator>  OutputIterator set\_difference (InputIterator1 first1, InputIterator1 last1,  InputIterator2 first2, InputIterator2 last2,  OutputIterator result)  {  *while* (first1!=last1 && first2!=last2)  {  *if* (\*first1<\*first2) { \*result = \*first1; ++result; ++first1; }  *else* *if* (\*first2<\*first1) ++first2;  *else* { ++first1; ++first2; }  }  *return* std::copy(first1,last1,result);  } |

**Parameters**

first1, last1

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the first sorted 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

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the second sorted sequence. The range used is [first2,last2).

result

[Output iterator](http://www.cplusplus.com/OutputIterator) to the initial position of the range where the resulting sequence is stored.  
The pointed type shall support being assigned the value of an element from the first range.

comp

Binary function that accepts two arguments of the types pointed by the input iterators, 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.

The ranges shall not overlap.

**Return value**

An iterator to the end of the constructed range.

**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;  } |

Output:

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

**Complexity**

Up to linear in 2\*(count1+count2)-1 (where *countX* is the [distance](http://www.cplusplus.com/distance) between *firstX* and *lastX*): Compares and assigns elements.

**Data races**

The objects in the ranges [first1,last1) and [first2,last2)are accessed.  
The objects in the range between *result* and the returned value are modified.

**Exceptions**

Throws if any of the element comparisons, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::set\_symmetric\_difference**

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

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.  
  
The elements are compared using operator< for the first version, and *comp* for the second. Two elements, a and b are considered equivalent if (!(a<b) && !(b<a)) or if (!comp(a,b) && !comp(b,a)).  
  
The elements in the ranges shall already be ordered according to this same criterion (operator< or *comp*). The resulting range is also sorted according to this.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | *template* <*class* InputIterator1, *class* InputIterator2, *class* OutputIterator>  OutputIterator set\_symmetric\_difference (InputIterator1 first1, InputIterator1 last1,  InputIterator2 first2, InputIterator2 last2,  OutputIterator result)  {  *while* (*true*)  {  *if* (first1==last1) *return* std::copy(first2,last2,result);  *if* (first2==last2) *return* std::copy(first1,last1,result);  *if* (\*first1<\*first2) { \*result=\*first1; ++result; ++first1; }  *else* *if* (\*first2<\*first1) { \*result = \*first2; ++result; ++first2; }  *else* { ++first1; ++first2; }  }  } |

**Parameters**

first1, last1

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the first sorted 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

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the second sorted sequence. The range used is [first2,last2).

result

[Output iterator](http://www.cplusplus.com/OutputIterator) to the initial position of the range where the resulting sequence is stored.  
The pointed type shall support being assigned the value of an element from the other ranges.

comp

Binary function that accepts two arguments of the types pointed by the input iterators, 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.

The ranges shall not overlap.

**Return value**

An iterator to the end of the constructed range.

**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;  } |

Output:

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

**Complexity**

Up to linear in 2\*(count1+count2)-1 (where *countX* is the [distance](http://www.cplusplus.com/distance) between *firstX* and *lastX*): Compares and assigns elements.

**Data races**

The objects in the ranges [first1,last1) and [first2,last2)are accessed.  
The objects in the range between *result* and the returned value are modified.

**Exceptions**

Throws if any of the element comparisons, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::push\_heap**

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

Push element into heap range

Given a heap in the range [first,last-1), this function extends the range considered a heap to [first,last) by placing the value in (last-1) into its corresponding location within it.  
  
A range can be organized into a heap by calling [make\_heap](http://www.cplusplus.com/make_heap). After that, its heap properties are preserved if elements are added and removed from it using push\_heap and [pop\_heap](http://www.cplusplus.com/pop_heap), respectively.

**Parameters**

first, last

[Random-access iterators](http://www.cplusplus.com/RandomAccessIterator) to the initial and final positions of the new heap range, including the pushed element. 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*.

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 be less than the second in the specific *strict weak ordering* it defines.  
Unless [first,last) is an empty or one-element heap, this argument shall be the same as used to construct the heap.  
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 | *// range heap example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::make\_heap, std::pop\_heap, std::push\_heap, std::sort\_heap*  *#include <vector> // std::vector*  *int* main () {  *int* myints[] = {10,20,30,5,15};  std::vector<*int*> v(myints,myints+5);  std::make\_heap (v.begin(),v.end());  std::cout << "initial max heap : " << v.front() << '\n';  std::pop\_heap (v.begin(),v.end()); v.pop\_back();  std::cout << "max heap after pop : " << v.front() << '\n';  v.push\_back(99); std::push\_heap (v.begin(),v.end());  std::cout << "max heap after push: " << v.front() << '\n';  std::sort\_heap (v.begin(),v.end());  std::cout << "final sorted range :";  *for* (*unsigned* i=0; i<v.size(); i++)  std::cout << ' ' << v[i];  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| initial max heap : 30  max heap after pop : 20  max heap after push: 99  final sorted range : 5 10 15 20 99 |

**Complexity**

Up to logarithmic in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Compares elements and potentially swaps (or moves) them until rearranged as a longer heap.

**Data races**

Some (or all) of the objects in the range [first,last) are modified.

**Exceptions**

Throws if any of the element comparisons, the element swaps (or moves) or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::pop\_heap**

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

Pop element from heap range

Rearranges the elements in the heap range [first,last) in such a way that the part considered a heap is shortened by one: The element with the highest value is moved to (last-1).  
  
While the element with the highest value is moved from *first* to (last-1) (which now is out of the heap), the other elements are reorganized in such a way that the range [first,last-1) preserves the properties of a heap.  
  
A range can be organized into a heap by calling [make\_heap](http://www.cplusplus.com/make_heap). After that, its heap properties are preserved if elements are added and removed from it using push\_heap and [pop\_heap](http://www.cplusplus.com/pop_heap), respectively.

**Parameters**

first, last

[Random-access iterators](http://www.cplusplus.com/RandomAccessIterator) to the initial and final positions of the heap to be shrank by one. 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*.  
This shall not be an empty range.

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 be less than the second in the specific *strict weak ordering* it defines.  
Unless [first,last) is a one-element heap, this argument shall be the same as used to construct the heap.  
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 | *// range heap example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::make\_heap, std::pop\_heap, std::push\_heap, std::sort\_heap*  *#include <vector> // std::vector*  *int* main () {  *int* myints[] = {10,20,30,5,15};  std::vector<*int*> v(myints,myints+5);  std::make\_heap (v.begin(),v.end());  std::cout << "initial max heap : " << v.front() << '\n';  std::pop\_heap (v.begin(),v.end()); v.pop\_back();  std::cout << "max heap after pop : " << v.front() << '\n';  v.push\_back(99); std::push\_heap (v.begin(),v.end());  std::cout << "max heap after push: " << v.front() << '\n';  std::sort\_heap (v.begin(),v.end());  std::cout << "final sorted range :";  *for* (*unsigned* i=0; i<v.size(); i++)  std::cout << ' ' << v[i];  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| initial max heap : 30  max heap after pop : 20  max heap after push: 99  final sorted range : 5 10 15 20 99 |

**Complexity**

Up to twice logarithmic in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Compares elements and potentially swaps (or moves) them until rearranged as a shorter heap.

**Data races**

Some (or all) of the objects in the range [first,last) are modified.

**Exceptions**

Throws if any of the element comparisons, the element swaps (or moves) or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::make\_heap**

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

Make heap from range

Rearranges the elements in the range [first,last) in such a way that they form a *heap*.  
  
A *heap* is a way to organize the elements of a range that allows for fast retrieval of the element with the highest value at any moment (with [pop\_heap](http://www.cplusplus.com/pop_heap)), even repeatedly, while allowing for fast insertion of new elements (with [push\_heap](http://www.cplusplus.com/push_heap)).  
  
The element with the highest value is always pointed by *first*. The order of the other elements depends on the particular implementation, but it is consistent throughout all heap-related functions of this header.  
  
The elements are compared using operator< (for the first version), or *comp* (for the second): The element with the highest value is an element for which this would return false when compared to every other element in the range.  
  
The standard container adaptor [priority\_queue](http://www.cplusplus.com/priority_queue) calls make\_heap, [push\_heap](http://www.cplusplus.com/push_heap) and [pop\_heap](http://www.cplusplus.com/pop_heap) automatically to maintain *heap properties* for a container.

**Parameters**

first, last

[Random-access iterators](http://www.cplusplus.com/RandomAccessIterator) to the initial and final positions of the sequence to be transformed into a heap. 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 be less than 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 | *// range heap example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::make\_heap, std::pop\_heap, std::push\_heap, std::sort\_heap*  *#include <vector> // std::vector*  *int* main () {  *int* myints[] = {10,20,30,5,15};  std::vector<*int*> v(myints,myints+5);  std::make\_heap (v.begin(),v.end());  std::cout << "initial max heap : " << v.front() << '\n';  std::pop\_heap (v.begin(),v.end()); v.pop\_back();  std::cout << "max heap after pop : " << v.front() << '\n';  v.push\_back(99); std::push\_heap (v.begin(),v.end());  std::cout << "max heap after push: " << v.front() << '\n';  std::sort\_heap (v.begin(),v.end());  std::cout << "final sorted range :";  *for* (*unsigned* i=0; i<v.size(); i++)  std::cout << ' ' << v[i];  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| initial max heap : 30  max heap after pop : 20  max heap after push: 99  final sorted range : 5 10 15 20 99 |

**Complexity**

Up to linear in three times the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Compares elements and potentially swaps (or moves) them until rearranged as a heap.

**Data races**

The objects in the range [first,last) are modified.

**Exceptions**

Throws if any of the element comparisons, the element swaps (or moves) or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::sort\_heap**

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

Sort elements of heap

Sorts the elements in the heap range [first,last) into ascending order.  
  
The elements are compared using operator< for the first version, and *comp* for the second, which shall be the same as used to construct the heap.  
  
The range loses its properties as a heap.

**Parameters**

first, last

[Random-access iterators](http://www.cplusplus.com/RandomAccessIterator) to the initial and final positions of the heap range 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*.

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.  
Unless [first,last) is a one-element heap, this argument shall be the same as used to construct the heap.  
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 | *// range heap example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::make\_heap, std::pop\_heap, std::push\_heap, std::sort\_heap*  *#include <vector> // std::vector*  *int* main () {  *int* myints[] = {10,20,30,5,15};  std::vector<*int*> v(myints,myints+5);  std::make\_heap (v.begin(),v.end());  std::cout << "initial max heap : " << v.front() << '\n';  std::pop\_heap (v.begin(),v.end()); v.pop\_back();  std::cout << "max heap after pop : " << v.front() << '\n';  v.push\_back(99); std::push\_heap (v.begin(),v.end());  std::cout << "max heap after push: " << v.front() << '\n';  std::sort\_heap (v.begin(),v.end());  std::cout << "final sorted range :";  *for* (*unsigned* i=0; i<v.size(); i++)  std::cout << ' ' << v[i];  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| initial max heap : 30  max heap after pop : 20  max heap after push: 99  final sorted range : 5 10 15 20 99 |

**Complexity**

Up to linearithmic in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Performs at most N\*log(N) (where *N* is this distance) comparisons of elements, and up to that many element swaps (or moves).

**Data races**

The objects in the range [first,last) are modified.

**Exceptions**

Throws if any of the element comparisons, the element swaps (or moves) or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::is\_heap**

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

Test if range is heap

Returns true if the range [first,last) forms a *heap*, as if constructed with [make\_heap](http://www.cplusplus.com/make_heap).  
  
The elements are compared using operator< for the first version, and *comp* for the second.

**Parameters**

first, last

[RandomAccess iterators](http://www.cplusplus.com/RandomAccessIterator) 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 a *heap* (as if constructed with [make\_heap](http://www.cplusplus.com/make_heap)), 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 | *// is\_heap example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::is\_heap, std::make\_heap, std::pop\_heap*  *#include <vector> // std::vector*  *int* main () {  std::vector<*int*> foo {9,5,2,6,4,1,3,8,7};  *if* (!std::is\_heap(foo.begin(),foo.end()))  std::make\_heap(foo.begin(),foo.end());  std::cout << "Popping out elements:";  *while* (!foo.empty()) {  std::pop\_heap(foo.begin(),foo.end()); *// moves largest element to back*  std::cout << ' ' << foo.back(); *// prints back*  foo.pop\_back(); *// pops element out of container*  }  std::cout << '\n';  *return* 0;  } |

Output:

|  |
| --- |
| Popping out elements: 9 8 7 6 5 4 3 2 1 |

**Complexity**

Up to linear in one less than the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Compares pairs of elements until a mismatch is found.

**Data races**

The objects in the range [first,last) are accessed.

**Exceptions**

Throws if either an element comparison or an operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::is\_heap\_until**

|  |  |
| --- | --- |
| **default (1)** | template <class RandomAccessIterator>  RandomAccessIterator is\_heap\_until (RandomAccessIterator first,  RandomAccessIterator last); |
| **custom (2)** | template <class RandomAccessIterator, class Compare>  RandomAccessIterator is\_heap\_until (RandomAccessIterator first,  RandomAccessIterator last  Compare comp); |

Find first element not in heap order

Returns an iterator to the first element in the range [first,last) which is not in a valid position if the range is considered a heap (as if constructed with [make\_heap](http://www.cplusplus.com/make_heap)).  
  
The range between *first* and the iterator returned [is a heap](http://www.cplusplus.com/is_heap).  
  
If the entire range is a valid heap, the function returns *last*.  
  
The elements are compared using operator< for the first version, and *comp* for the second.

**Parameters**

first, last

[Random-access iterators](http://www.cplusplus.com/RandomAccessIterator) to the initial and final positions in a 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**

An iterator to the first element in the range which is not in a valid position for the range to be a heap, or *last* if all elements are validly positioned or if the range contains less than two elements.

**Example**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | *// is\_heap example*  *#include <iostream> // std::cout*  *#include <algorithm> // std::is\_heap\_until, std::sort, std::reverse*  *#include <vector> // std::vector*  *int* main () {  std::vector<*int*> foo {2,6,9,3,8,4,5,1,7};  std::sort(foo.begin(),foo.end());  std::reverse(foo.begin(),foo.end());  *auto* last = std::is\_heap\_until (foo.begin(),foo.end());  std::cout << "The " << (last-foo.begin()) << " first elements are a valid heap:";  *for* (*auto* it=foo.begin(); it!=last; ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  *return* 0;  } |

Most implementations consider a range sorted in reverse order a valid heap:  
Possible output:

|  |
| --- |
| The 9 first elements are a valid heap: 9 8 7 6 5 4 3 2 1 |

**Complexity**

Up to linear in the [distance](http://www.cplusplus.com/distance) between *first* and *last*: Compares elements until a mismatch is found.

**Data races**

The objects in the range [first,last) are accessed.

**Exceptions**

Throws if either *comp* or an operation on an iterator throws.  
Note that invalid parameters cause *undefined behavior*.

function template

<algorithm>

**std::min**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

|  |  |
| --- | --- |
| **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); |

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.  
  
The behavior of this function template (C++98) is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 | *template* <*class* T> *const* T& min (*const* T& a, *const* T& b) {  *return* !(b<a)?a:b; *// or: return !comp(b,a)?a:b; for version (2)*  } |

**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<.

* [C++98](javascript:switch2.select(1))
* [C++11](javascript:switch2.select(2))

T shall be [*copy constructible*](http://www.cplusplus.com/is_copy_constructible).

**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;  } |

Output:

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

**Complexity**

Linear in one less than the number of elements compared (constant for *(1)* and *(2)*).

**Exceptions**

Throws if any comparison throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::max**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

|  |  |
| --- | --- |
| **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); |

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.  
  
The behavior of this function template (C++98) is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 | *template* <*class* T> *const* T& max (*const* T& a, *const* T& b) {  *return* (a<b)?b:a; *// or: return comp(a,b)?b:a; for version (2)*  } |

**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<.

* [C++98](javascript:switch2.select(1))
* [C++11](javascript:switch2.select(2))

T shall be [*copy constructible*](http://www.cplusplus.com/is_copy_constructible).

**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;  } |

Output:

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

**Complexity**

Linear in one less than the number of elements compared (constant for *(1)* and *(2)*).

**Exceptions**

Throws if any comparison throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::minmax**

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

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).  
  
The versions for *initializer lists (3)* return a [pair](http://www.cplusplus.com/pair) with the smallest of all the elements in the list as first element (the first of them, if there are more than one), and the largest as second (the last of them, if there are more than one).  
  
The function uses operator< (or *comp*, if provided) to compare the values.  
  
The behavior of this function template (version *(1)*) is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 | *template* <*class* T> pair <*const* T&,*const* T&> minmax (*const* T& a, *const* T& b) {  *return* (b<a) ? std::make\_pair(b,a) : std::make\_pair(a,b);  } |

**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<.  
For *(3)*, T shall be [*copy constructible*](http://www.cplusplus.com/is_copy_constructible).

**Return value**

The lesser of the values passed as arguments.

**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;  } |

Output:

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

**Complexity**

Up to linear in one and half times the number of elements compared (constant for *(1)* and *(2)*).

**Exceptions**

Throws if any comparison throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::min\_element**

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

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.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | *template* <*class* ForwardIterator>  ForwardIterator min\_element ( ForwardIterator first, ForwardIterator last )  {  *if* (first==last) *return* last;  ForwardIterator smallest = first;  *while* (++first!=last)  *if* (\*first<\*smallest) *// or: if (comp(\*first,\*smallest)) for version (2)*  smallest=first;  *return* smallest;  } |

**Parameters**

first, last

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the sequence to compare. 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*.

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 less than the second.  
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 smallest value in the range, or *last* if the range is empty.

**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;  } |

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 |

**Complexity**

Linear in one less than the number of elements compared.

**Data races**

The objects in the range [first,last) are accessed.

**Exceptions**

Throws if any comparison throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::max\_element**

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

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.  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | *template* <*class* ForwardIterator>  ForwardIterator max\_element ( ForwardIterator first, ForwardIterator last )  {  *if* (first==last) *return* last;  ForwardIterator largest = first;  *while* (++first!=last)  *if* (\*largest<\*first) *// or: if (comp(\*largest,\*first)) for version (2)*  largest=first;  *return* largest;  } |

**Parameters**

first, last

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the sequence to compare. 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*.

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 less than the second.  
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 largest value in the range, or *last* if the range is empty.

**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;  } |

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 |

**Complexity**

Linear in one less than the number of elements compared.

**Data races**

The objects in the range [first,last) are accessed.

**Exceptions**

Throws if any comparison throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::minmax\_element**

|  |  |
| --- | --- |
| **default (1)** | template <class ForwardIterator>  pair<ForwardIterator,ForwardIterator>  minmax\_element (ForwardIterator first, ForwardIterator last); |
| **custom (2)** | template <class ForwardIterator, class Compare>  pair<ForwardIterator,ForwardIterator>  minmax\_element (ForwardIterator first, ForwardIterator last, Compare comp); |

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.

**Parameters**

first, last

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the sequence to compare. 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*.

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 less than the second.  
The function shall not modify any of its arguments.  
This can either be a function pointer or a function object.

**Return value**

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.  
  
[pair](http://www.cplusplus.com/pair) is a class template defined in [<utility>](http://www.cplusplus.com/%3Cutility%3E).

**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;  } |

Output:

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

**Complexity**

Up to linear in 1.5 times one less than the number of elements compared.

**Data races**

The objects in the range [first,last) are accessed.

**Exceptions**

Throws if any comparison throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::lexicographical\_compare**

|  |  |
| --- | --- |
| **default (1)** | template <class InputIterator1, class InputIterator2>  bool lexicographical\_compare (InputIterator1 first1, InputIterator1 last1,  InputIterator2 first2, InputIterator2 last2); |
| **custom (2)** | template <class InputIterator1, class InputIterator2, class Compare>  bool lexicographical\_compare (InputIterator1 first1, InputIterator1 last1,  InputIterator2 first2, InputIterator2 last2,  Compare comp); |

Lexicographical less-than comparison

Returns true if the range [first1,last1) compares *lexicographically less* than the range [first2,last2).  
  
A *lexicographical comparison* is the kind of comparison generally used to sort words alphabetically in dictionaries; It involves comparing sequentially the elements that have the same position in both ranges against each other until one element is not equivalent to the other. The result of comparing these first non-matching elements is the result of the lexicographical comparison.  
  
If both sequences compare equal until one of them ends, the shorter sequence is *lexicographically less* than the longer one.  
  
The elements are compared using operator< for the first version, and *comp* for the second. Two elements, a and b are considered equivalent if (!(a<b) && !(b<a)) or if (!comp(a,b) && !comp(b,a)).  
  
The behavior of this function template is equivalent to:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 | *template* <*class* InputIterator1, *class* InputIterator2>  *bool* lexicographical\_compare (InputIterator1 first1, InputIterator1 last1,  InputIterator2 first2, InputIterator2 last2)  {  *while* (first1!=last1)  {  *if* (first2==last2 || \*first2<\*first1) *return* *false*;  *else* *if* (\*first1<\*first2) *return* *true*;  ++first1; ++first2;  }  *return* (first2!=last2);  } |

**Parameters**

first1, last1

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the first 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

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions of the second sequence. The range used is [first2,last2).

comp

Binary function that accepts two arguments of the types pointed by the iterators, 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 first range compares *lexicographically less* than than the second.  
false otherwise (including when all the elements of both ranges are equivalent).

**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 | *// lexicographical\_compare example*  *#include <iostream> // std::cout, std::boolalpha*  *#include <algorithm> // std::lexicographical\_compare*  *#include <cctype> // std::tolower*  *// a case-insensitive comparison function:*  *bool* mycomp (*char* c1, *char* c2)  { *return* std::tolower(c1)<std::tolower(c2); }  *int* main () {  *char* foo[]="Apple";  *char* bar[]="apartment";  std::cout << std::boolalpha;  std::cout << "Comparing foo and bar lexicographically (foo<bar):\n";  std::cout << "Using default comparison (operator<): ";  std::cout << std::lexicographical\_compare(foo,foo+5,bar,bar+9);  std::cout << '\n';  std::cout << "Using mycomp as comparison object: ";  std::cout << std::lexicographical\_compare(foo,foo+5,bar,bar+9,mycomp);  std::cout << '\n';  *return* 0;  } |

The default comparison compares plain ASCII character codes, where 'A' (65) compares less than 'a' (97).  
Our mycomp function transforms the letters to lowercase before comparing them, so here the first letter not matching is the third ('a' vs 'p').  
  
Output:

|  |
| --- |
| Comparing foo and bar lexicographically (foo<bar):  Using default comparison (operator<): true  Using mycomp as comparison object: false |

**Complexity**

At most, performs 2\*min(count1,count2) comparisons or applications of *comp* (where *countX* is the distance between *firstX* and *lastX*).

**Complexity**

Up to linear in 2\*min(count1,count2) (where *countX* is the distance between *firstX* and *lastX*): Compares elements symmetrically until a mismatch is found.

**Data races**

The objects in the ranges [first1,last1) and [first2,last2) are accessed.

**Exceptions**

Throws if either an element comparison or an operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::next\_permutation**

|  |  |
| --- | --- |
| **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); |

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.  
  
A *permutation* is each one of the N! possible arrangements the elements can take (where *N* is the number of elements in the range). Different permutations can be ordered according to how they compare [*lexicographicaly*](http://www.cplusplus.com/lexicographical_compare) to each other; The first such-sorted possible permutation (the one that would compare *lexicographically smaller* to all other permutations) is the one which has all its elements sorted in ascending order, and the largest has all its elements sorted in descending order.  
  
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.

**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;  } |

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 |

**Complexity**

Up to linear in half the [distance](http://www.cplusplus.com/distance) between *first* and *last* (in terms of actual swaps).

**Data races**

The objects in the range [first,last) are modified.

**Exceptions**

Throws if any element swap throws or if any operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.

function template

<algorithm>

**std::prev\_permutation**

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

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.  
  
A *permutation* is each one of the N! possible arrangements the elements can take (where *N* is the number of elements in the range). Different permutations can be ordered according to how they compare [*lexicographicaly*](http://www.cplusplus.com/lexicographical_compare) to each other; The first such-sorted possible permutation (the one that would compare *lexicographically smaller* to all other permutations) is the one which has all its elements sorted in ascending order, and the largest has all its elements sorted in descending order.  
  
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.

**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 smaller permutation.  
Otherwise, the function returns false to indicate that the arrangement is not less than the previous, but the largest possible (sorted in descending order).

**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;  } |

Output:

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

**Complexity**

Up to linear in half the [distance](http://www.cplusplus.com/distance) between *first* and *last* (in terms of actual swaps).

**Data races**

The objects in the range [first,last) are modified.

**Exceptions**

Throws if any element swap throws or if any operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.