Algorithms library

The algorithms library defines functions for a variety of purposes (e.g. searching, sorting, counting, manipulating) that operate on ranges of elements. Note that a range is defined as `[first, last)` where `last` refers to the element *past* the last element to inspect or modify.

Non-modifying sequence operations

Defined in header `<algorithm>`

`all_of` (C++11) **`any_of`** (C++11) **`none_of`** (C++11)	checks if a predicate is `true` for all, any or none of the elements in a range (function template)
`for_each`	applies a function to a range of elements (function template)
`count` **`count_if`**	returns the number of elements satisfying specific criteria (function template)
`mismatch`	finds the first position where two ranges differ (function template)
`equal`	determines if two sets of elements are the same (function template)
`find` **`find_if`** **`find_if_not`** (C++11)	finds the first element satisfying specific criteria (function template)
`find_end`	finds the last sequence of elements in a certain range (function template)
`find_first_of`	searches for any one of a set of elements (function template)
`adjacent_find`	finds the first two adjacent items that are equal (or satisfy a given predicate) (function template)
`search`	searches for a range of elements (function template)
`search_n`	searches for a number consecutive copies of an element in a range (function template)

Modifying sequence operations

Defined in header `<algorithm>`

`copy` **`copy_if`** (C++11)	copies a range of elements to a new location (function template)
`copy_n` (C++11)	copies a number of elements to a new location (function template)
`copy_backward`	copies a range of elements in backwards order (function template)
`move` (C++11)	moves a range of elements to a new location (function template)
`move_backward` (C++11)	moves a range of elements to a new location in backwards order (function template)
`fill`	assigns a range of elements a certain value (function template)
`fill_n`	assigns a value to a number of elements (function template)
`transform`	applies a function to a range of elements (function template)
`generate`	saves the result of a function in a range (function template)
`generate_n`	saves the result of N applications of a function (function template)
`remove` **`remove_if`**	removes elements satisfying specific criteria (function template)
	copies a range of elements omitting those that satisfy specific criteria

remove_copy **remove_copy_if**	(function template)
replace **replace_if**	replaces all values satisfying specific criteria with another value (function template)
replace_copy **replace_copy_if**	copies a range, replacing elements satisfying specific criteria with another value (function template)
swap	swaps the values of two objects (function template)
swap_ranges	swaps two ranges of elements (function template)
iter_swap	swaps the elements pointed to by two iterators (function template)
reverse	reverses the order of elements in a range (function template)
reverse_copy	creates a copy of a range that is reversed (function template)
rotate	rotates the order of elements in a range (function template)
rotate_copy	copies and rotate a range of elements (function template)
random_shuffle (until C++17) **shuffle** (C++11)	randomly re-orders elements in a range (function template)
unique	removes consecutive duplicate elements in a range (function template)
unique_copy	creates a copy of some range of elements that contains no consecutive duplicates (function template)

Partitioning operations

Defined in header `<algorithm>`	
is_partitioned (C++11)	determines if the range is partitioned by the given predicate (function template)
partition	divides a range of elements into two groups (function template)
partition_copy (C++11)	copies a range dividing the elements into two groups (function template)
stable_partition	divides elements into two groups while preserving their relative order (function template)
partition_point (C++11)	locates the partition point of a partitioned range (function template)

Sorting operations

Defined in header `<algorithm>`	
is_sorted (C++11)	checks whether a range is sorted into ascending order (function template)
is_sorted_until (C++11)	finds the largest sorted subrange (function template)
sort	sorts a range into ascending order (function template)
partial_sort	sorts the first N elements of a range (function template)
partial_sort_copy	copies and partially sorts a range of elements (function template)
stable_sort	sorts a range of elements while preserving order between equal elements (function template)
nth_element	partially sorts the given range making sure that it is partitioned by the given element (function template)

Binary search operations (on sorted ranges)

Defined in header `<algorithm>`

`lower_bound`	returns an iterator to the first element *not less* than the given value (function template)
`upper_bound`	returns an iterator to the first element *greater* than a certain value (function template)
`binary_search`	determines if an element exists in a certain range (function template)
`equal_range`	returns range of elements matching a specific key (function template)

Set operations (on sorted ranges)

Defined in header `<algorithm>`

`merge`	merges two sorted ranges (function template)
`inplace_merge`	merges two ordered ranges in-place (function template)
`includes`	returns true if one set is a subset of another (function template)
`set_difference`	computes the difference between two sets (function template)
`set_intersection`	computes the intersection of two sets (function template)
`set_symmetric_difference`	computes the symmetric difference between two sets (function template)
`set_union`	computes the union of two sets (function template)

Heap operations

Defined in header `<algorithm>`

`is_heap` (C++11)	checks if the given range is a max heap (function template)
`is_heap_until` (C++11)	finds the largest subrange that is a max heap (function template)
`make_heap`	creates a max heap out of a range of elements (function template)
`push_heap`	adds an element to a max heap (function template)
`pop_heap`	removes the largest element from a max heap (function template)
`sort_heap`	turns a max heap into a range of elements sorted in ascending order (function template)

Minimum/maximum operations

Defined in header `<algorithm>`

`max`	returns the larger of two elements (function template)
`max_element`	returns the largest element in a range (function template)
`min`	returns the smaller of two elements (function template)
`min_element`	returns the smallest element in a range (function template)
`minmax` (C++11)	returns the larger and the smaller of two elements (function template)
`minmax_element` (C++11)	returns the smallest and the largest element in a range (function template)
`lexicographical_compare`	returns true if one range is lexicographically less than another (function template)
`is_permutation` (C++11)	determines if a sequence is a permutation of another sequence (function template)
`next_permutation`	generates the next greater lexicographic permutation of a range of elements (function template)

prev_permutation	generates the next smaller lexicographic permutation of a range of elements (function template)

Numeric operations

Defined in header `<numeric>`	
iota (C++11)	fills a range with successive increments of the starting value (function template)
accumulate	sums up a range of elements (function template)
inner_product	computes the inner product of two ranges of elements (function template)
adjacent_difference	computes the differences between adjacent elements in a range (function template)
partial_sum	computes the partial sum of a range of elements (function template)

C library

Defined in header `<cstdlib>`	
qsort	sorts a range of elements with unspecified type (function)
bsearch	searches an array for an element of unspecified type (function)

See also

C documentation for **`Algorithms`**

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm&oldid=70996"

std::all_of, std::any_of, std::none_of

Defined in header `<algorithm>`		

`template< class InputIt, class UnaryPredicate >`
`bool all_of( InputIt first, InputIt last, UnaryPredicate p );`
	(1)	(since C++11)

`template< class InputIt, class UnaryPredicate >`
`bool any_of( InputIt first, InputIt last, UnaryPredicate p );`
	(2)	(since C++11)

`template< class InputIt, class UnaryPredicate >`
`bool none_of( InputIt first, InputIt last, UnaryPredicate p );`
	(3)	(since C++11)

1) Checks if unary predicate `p` returns `true` for all elements in the range `[first, last)`.

2) Checks if unary predicate `p` returns `true` for at least one element in the range `[first, last)`.

3) Checks if unary predicate `p` returns `true` for no elements in the range `[first, last)`.

Parameters

first, last - the range of elements to examine

p - unary predicate .

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type &a);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it.
The type `Type` must be such that an object of type `InputIt` can be dereferenced and then implicitly converted to `Type`.

Type requirements

- `InputIt` must meet the requirements of `InputIterator`.
- `UnaryPredicate` must meet the requirements of `Predicate`.

Return value

1) `true` if unary predicate returns `true` for all elements in the range, `false` otherwise. Returns `true` if the range is empty.

2) `true` if unary predicate returns `true` for at least one element in the range, `false` otherwise. Returns `false` if the range is empty.

3) `true` if unary predicate returns `true` for no elements in the range, `false` otherwise. Returns `true` if the range is empty.

Complexity

At most `last` - `first` applications of the predicate

Possible implementation

First version

```
template< class InputIt, class UnaryPredicate >
bool all_of(InputIt first, InputIt last, UnaryPredicate p)
{
    return std::find_if_not(first, last, p) == last;
}
```

Second version

```
template< class InputIt, class UnaryPredicate >
bool any_of(InputIt first, InputIt last, UnaryPredicate p)
{
    return std::find_if(first, last, p) != last;
}
```

Third version

```
template< class InputIt, class UnaryPredicate >
bool none_of(InputIt first, InputIt last, UnaryPredicate p)
{
    return std::find_if(first, last, p) == last;
}
```

Example

Run this code

```
#include <vector>
#include <numeric>
#include <algorithm>
#include <iterator>
#include <iostream>
#include <functional>

int main()
{
    std::vector<int> v(10, 2);
    std::partial_sum(v.cbegin(), v.cend(), v.begin());
    std::cout << "Among the numbers: ";
    std::copy(v.cbegin(), v.cend(), std::ostream_iterator<int>(std::cout, " "));
    std::cout << '\n';

    if (std::all_of(v.cbegin(), v.cend(), [](int i){ return i % 2 == 0; })) {
        std::cout << "All numbers are even\n";
    }
    if (std::none_of(v.cbegin(), v.cend(), std::bind(std::modulus<int>(),
                                                     std::placeholders::_1, 2))) {
        std::cout << "None of them are odd\n";
    }
    struct DivisibleBy
    {
        const int d;
        DivisibleBy(int n) : d(n) {}
        bool operator()(int n) const { return n % d == 0; }
    };

    if (std::any_of(v.cbegin(), v.cend(), DivisibleBy(7))) {
        std::cout << "At least one number is divisible by 7\n";
    }
}
```

Output:

```
Among the numbers: 2 4 6 8 10 12 14 16 18 20
All numbers are even
None of them are odd
At least one number is divisible by 7
```

See also

`std::experimental::parallel::all_of` (parallelism TS)	parallelized version of `std::all_of` (function template)
`std::experimental::parallel::any_of` (parallelism TS)	parallelized version of `std::any_of` (function template)
`std::experimental::parallel::none_of` (parallelism TS)	parallelized version of `std::none_of` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/all_any_none_of&oldid=79868"

std::for_each

Defined in header `<algorithm>`

```
template< class InputIt, class UnaryFunction >
UnaryFunction for_each( InputIt first, InputIt last, UnaryFunction f );
```

Applies the given function object `f` to the result of dereferencing every iterator in the range `[first, last)`, in order.

If `InputIt` is a mutable iterator, `f` may modify the elements of the range through the dereferenced iterator. If `f` returns a result, the result is ignored.

Parameters

first, last - the range to apply the function to

f - function object, to be applied to the result of dereferencing every iterator in the range `[first, last)`

The signature of the function should be equivalent to the following:

```
void fun(const Type &a);
```

The signature does not need to have `const &`.
The type `Type` must be such that an object of type `InputIt` can be dereferenced and then implicitly converted to `Type`.

Type requirements

- `InputIt` must meet the requirements of `InputIterator`.
- `UnaryFunction` must meet the requirements of `MoveConstructible`. Does not have to be `CopyConstructible`

Return value

`f`	(until C++11)
`std::move(f)`	(since C++11)

Complexity

Exactly `last` - `first` applications of `f`

Possible implementation

```
template<class InputIt, class UnaryFunction>
UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction f)
{
    for (; first != last; ++first) {
        f(*first);
    }
    return f;
}
```

Example

The following example uses a lambda function to increment all of the elements of a vector and then uses an overloaded `operator()` in a functor to compute their sum:

Run this code

```
#include <vector>
#include <algorithm>
#include <iostream>

struct Sum {
    Sum() { sum = 0; }
    void operator()(int n) { sum += n; }

    int sum;
};

int main()
{
    std::vector<int> nums{3, 4, 2, 9, 15, 267};

    std::cout << "before:";
    for (auto n : nums) {
        std::cout << ' ' << n;
    }
    std::cout << '\n';

    std::for_each(nums.begin(), nums.end(), [](int &n){ n++; });

    // Calls Sum::operator() for each number
    Sum s = std::for_each(nums.begin(), nums.end(), Sum());

    std::cout << "after: ";
    for (auto n : nums) {
        std::cout << ' ' << n;
    }
    std::cout << '\n';
    std::cout << "sum: " << s.sum << '\n';
}
```

Output:

```
before: 3 4 2 9 15 267
after:  4 5 3 10 16 268
sum: 306
```

See also

transform	applies a function to a range of elements (function template)
range-for loop	executes loop over range (since C++11)
for_each (parallelism TS)	similar to **std::for_each** except returns void (function template)
for_each_n (parallelism TS)	applies a function object to the first n elements of a sequence (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/for_each&oldid=79856"

std::count, std::count_if

Defined in header `<algorithm>`

```
template< class InputIt, class T >
typename iterator_traits<InputIt>::difference_type                    (1)
    count( InputIt first, InputIt last, const T &value );
```

```
template< class InputIt, class UnaryPredicate >
typename iterator_traits<InputIt>::difference_type                    (2)
    count_if( InputIt first, InputIt last, UnaryPredicate p );
```

Returns the number of elements in the range `[first, last)` satisfying specific criteria. The first version counts the elements that are equal to `value`, the second version counts elements for which predicate `p` returns `true`.

Parameters

first, last - the range of elements to examine

value - the value to search for

p - unary predicate which returns `true` for the required elements.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type &a);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it.
The type `Type` must be such that an object of type `InputIt` can be dereferenced and then implicitly converted to `Type`.

Type requirements

- `InputIt` must meet the requirements of `InputIterator`.

Return value

number of elements satisfying the condition.

Complexity

exactly `last` - `first` comparisons / applications of the predicate

Notes

For the number of elements in the range `[first, last)` without any additional criteria, see `std::distance`.

Possible implementation

First version

```
template<class InputIt, class T>
typename iterator_traits<InputIt>::difference_type
    count(InputIt first, InputIt last, const T& value)
{
    typename iterator_traits<InputIt>::difference_type ret = 0;
    for (; first != last; ++first) {
        if (*first == value) {
            ret++;
        }
    }
    return ret;
}
```

Second version

```
template<class InputIt, class UnaryPredicate>
typename iterator_traits<InputIt>::difference_type
    count_if(InputIt first, InputIt last, UnaryPredicate p)
{
    typename iterator_traits<InputIt>::difference_type ret = 0;
    for (; first != last; ++first) {
        if (p(*first)) {
            ret++;
        }
    }
    return ret;
}
```

Example

The following code uses `count` to determine how many integers in a `std::vector` match a target value.

Run this code

```
#include <algorithm>
#include <iostream>
#include <vector>

int main()
{
    int data[] = { 1, 2, 3, 4, 4, 3, 7, 8, 9, 10 };
    std::vector<int> v(data, data+10);

    int target1 = 3;
    int target2 = 5;
    int num_items1 = std::count(v.begin(), v.end(), target1);
    int num_items2 = std::count(v.begin(), v.end(), target2);

    std::cout << "number: " << target1 << " count: " << num_items1 << '\n';
    std::cout << "number: " << target2 << " count: " << num_items2 << '\n';
}
```

Output:

```
number: 3 count: 2
number: 5 count: 0
```

This example uses a lambda expression to count elements divisible by 3.

Run this code

```
#include <algorithm>
#include <iostream>
#include <vector>

int main()
{
    int data[] = { 1, 2, 3, 4, 4, 3, 7, 8, 9, 10 };
    std::vector<int> v(data, data+10);

    int num_items1 = std::count_if(v.begin(), v.end(), [](int i) {return i % 3 == 0;});

    std::cout << "number divisible by three: " << num_items1 << '\n';
}
```

Output:

```
number divisible by three: 3
```

See also

std::experimental::parallel::count (parallelism TS)	parallelized version of `std::count` (function template)
std::experimental::parallel::count_if (parallelism TS)	parallelized version of `std::count_if` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/count&oldid=79872"

std::mismatch

Defined in header <algorithm>

```
template< class InputIt1, class InputIt2 >
std::pair<InputIt1,InputIt2>
    mismatch( InputIt1 first1, InputIt1 last1,
              InputIt2 first2 );
```

(1)

```
template< class InputIt1, class InputIt2, class BinaryPredicate >
std::pair<InputIt1,InputIt2>
    mismatch( InputIt1 first1, InputIt1 last1,
              InputIt2 first2,
              BinaryPredicate p );
```

(2)

```
template< class InputIt1, class InputIt2 >
std::pair<InputIt1,InputIt2>
    mismatch( InputIt1 first1, InputIt1 last1,
              InputIt2 first2, InputIt2 last2 );
```

(3) (since C++14)

```
template< class InputIt1, class InputIt2, class BinaryPredicate >
std::pair<InputIt1,InputIt2>
    mismatch( InputIt1 first1, InputIt1 last1,
              InputIt2 first2, InputIt2 last2,
              BinaryPredicate p );
```

(4) (since C++14)

Returns the first mismatching pair of elements from two ranges: one defined by `[first1, last1)` and another defined by `[first2,last2)`. If `last2` is not provided (overloads (1,2)), it denotes `first2 + (last1 - first1)`.

Overloads (1,3) use `operator==` to compare the elements, overloads (2,4) use the given binary predicate `p`.

Parameters

first1, last1 - the first range of the elements

first2, last2 - the second range of the elements

p - binary predicate which returns `true` if the elements should be treated as equal.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that objects of types `InputIt1` and `InputIt2` can be dereferenced and then implicitly converted to `Type1` and `Type2` respectively.

Type requirements

- `InputIt1` must meet the requirements of `InputIterator`.
- `InputIt2` must meet the requirements of `InputIterator`.
- `BinaryPredicate` must meet the requirements of `BinaryPredicate`.

Return value

`std::pair` with iterators to the first two non-equivalent elements.

If no mismatches are found when the comparison reaches `last1`, the pair holds `last1` and the corresponding iterator from the second range. The behavior is undefined if the second range is shorter than the first range.	(until C++14)
If no mismatches are found when the comparison reaches `last1` or `last2`, whichever happens first, the pair holds the end iterator and the corresponding iterator from the other range.	(since C++14)

Complexity

1,2) At most `last1 - first1` applications of `operator==` or the predicate `p`

3,4) At most min(`last1 - first1`, `last2 - first2`) applications of `operator==` or the predicate `p`.

Possible implementation

First version

```
template<class InputIt1, class InputIt2>
std::pair<InputIt1, InputIt2>
    mismatch(InputIt1 first1, InputIt1 last1, InputIt2 first2)
{
    while (first1 != last1 && *first1 == *first2) {
        ++first1, ++first2;
    }
    return std::make_pair(first1, first2);
}
```

Second version

```
template<class InputIt1, class InputIt2, class BinaryPredicate>
std::pair<InputIt1, InputIt2>
    mismatch(InputIt1 first1, InputIt1 last1, InputIt2 first2, BinaryPredicate p)
{
    while (first1 != last1 && p(*first1, *first2)) {
        ++first1, ++first2;
    }
    return std::make_pair(first1, first2);
}
```

Third version

```
template<class InputIt1, class InputIt2>
std::pair<InputIt1, InputIt2>
    mismatch(InputIt1 first1, InputIt1 last1, InputIt2 first2, InputIt2 last2)
{
    while (first1 != last1 && first2 != last2 && *first1 == *first2) {
        ++first1, ++first2;
    }
    return std::make_pair(first1, first2);
}
```

Fourth version

```
template<class InputIt1, class InputIt2, class BinaryPredicate>
std::pair<InputIt1, InputIt2>
    mismatch(InputIt1 first1, InputIt1 last1, InputIt2 first2, InputIt2 last2, BinaryPredica
{
    while (first1 != last1 && first2 != last2 && p(*first1, *first2)) {
        ++first1, ++first2;
    }
    return std::make_pair(first1, first2);
}
```

Example

This program determines the longest substring that is simultaneously found at the very beginning of the given string and at the very end of it, in reverse order (possibly overlapping)

Run this code

```
#include <iostream>
#include <string>
#include <algorithm>

std::string mirror_ends(const std::string& in)
{
    return std::string(in.begin(),
                       std::mismatch(in.begin(), in.end(), in.rbegin()).first);
}

int main()
{
    std::cout << mirror_ends("abXYZba") << '\n'
              << mirror_ends("abca") << '\n'
              << mirror_ends("aba") << '\n';
}
```

Output:

```
ab
a
aba
```

See also

equal	determines if two sets of elements are the same (function template)
find **find_if** **find_if_not** (C++11)	finds the first element satisfying specific criteria (function template)
lexicographical_compare	returns true if one range is lexicographically less than another (function template)
search	searches for a range of elements (function template)
std::experimental::parallel::mismatch (parallelism TS)	parallelized version of `std::mismatch` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/mismatch&oldid=79896"

std::equal

Defined in header `<algorithm>`

```
template< class InputIt1, class InputIt2 >
bool equal( InputIt1 first1, InputIt1 last1,
            InputIt2 first2 );
```
(1)

```
template< class InputIt1, class InputIt2, class BinaryPredicate >
bool equal( InputIt1 first1, InputIt1 last1,
            InputIt2 first2, BinaryPredicate p );
```
(2)

```
template< class InputIt1, class InputIt2 >
bool equal( InputIt1 first1, InputIt1 last1,
            InputIt2 first2, InputIt2 last2 );
```
(3) (since C++14)

```
template< class InputIt1, class InputIt2, class BinaryPredicate >
bool equal( InputIt1 first1, InputIt1 last1,
            InputIt2 first2, InputIt2 last2,
            BinaryPredicate p );
```
(4) (since C++14)

1,2) Returns `true` if the range `[first1, last1)` is equal to the range `[first2, first2 + (last1 - first1))`, and `false` otherwise

3,4) Returns `true` if the range `[first1, last1)` is equal to the range `[first2, last2)`, and `false` otherwise.

The two ranges are considered equal if, for every iterator `i` in the range `[first1,last1)`, `*i` equals `*(first2 + (i - first1))`. The overloads (1,3) use `operator==` to determine if two elements are equal, whereas overloads (2,4) use the given binary predicate `p`.

Parameters

first1, last1 - the first range of the elements to compare

first2, last2 - the second range of the elements to compare

p - binary predicate which returns `true` if the elements should be treated as equal.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that objects of types `InputIt1` and `InputIt2` can be dereferenced and then implicitly converted to `Type1` and `Type2` respectively.

Type requirements

- `InputIt1, InputIt2` must meet the requirements of `InputIterator`.

Return value

3,4) If the length of the range `[first1, last1)` does not equal the length of the range `[first2, last2)`, returns `false`

If the elements in the two ranges are equal, returns `true`.

Otherwise returns `false`.

Notes

`std::equal` may not be used to compare the ranges formed by the iterators from `std::unordered_set`, `std::unordered_multiset`, `std::unordered_map`, or `std::unordered_multimap` because the order in which the elements are stored in those containers may be different even if the two containers store the same elements.

When comparing entire containers for equality, `operator==` for the corresponding container are usually preferred.

Complexity

1,2) At most `last1` - `first1` applications of the predicate `p`

3,4) At most min(`last1` - `first1`, `last2` - `first2`) applications of the predicate `p`.
However, if `InputIt1` and `InputIt2` meet the requirements of `RandomAccessIterator` and `last1 - first1 != last2 - first2` then no applications of the predicate `p` are made.

Possible implementation

First version

```
template<class InputIt1, class InputIt2>
bool equal(InputIt1 first1, InputIt1 last1,
           InputIt2 first2)
{
    for (; first1 != last1; ++first1, ++first2) {
        if (!(*first1 == *first2)) {
            return false;
        }
    }
    return true;
}
```

Second version

```
template<class InputIt1, class InputIt2, class BinaryPredicate>
bool equal(InputIt1 first1, InputIt1 last1,
           InputIt2 first2, BinaryPredicate p)
{
    for (; first1 != last1; ++first1, ++first2) {
        if (!p(*first1, *first2)) {
            return false;
        }
    }
    return true;
}
```

Example

The following code uses `equal()` to test if a string is a palindrome

Run this code

```
#include <algorithm>
#include <iostream>
#include <string>

bool is_palindrome(const std::string& s)
{
    return std::equal(s.begin(), s.begin() + s.size()/2, s.rbegin());
}

void test(const std::string& s)
{
    std::cout << "\"" << s << "\" "
        << (is_palindrome(s) ? "is" : "is not")
        << " a palindrome\n";
}

int main()
{
    test("radar");
    test("hello");
}
```

Output:

```
"radar" is a palindrome
"hello" is not a palindrome
```

find **find_if** **find_if_not** (C++11)	finds the first element satisfying specific criteria (function template)
lexicographical_compare	returns true if one range is lexicographically less than another (function template)
mismatch	finds the first position where two ranges differ (function template)
search	searches for a range of elements (function template)
std::experimental::parallel::equal (parallelism TS)	parallelized version of `std::equal` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/equal&oldid=79873"

std::find, std::find_if, std::find_if_not

Defined in header `<algorithm>`

`template< class InputIt, class T >`
`InputIt find( InputIt first, InputIt last, const T& value );`
	(1)	

`template< class InputIt, class UnaryPredicate >`
`InputIt find_if( InputIt first, InputIt last,`
`                 UnaryPredicate p );`
	(2)	

`template< class InputIt, class UnaryPredicate >`
`InputIt find_if_not( InputIt first, InputIt last,`
`                     UnaryPredicate q );`
	(3)	(since C++11)

Returns the first element in the range `[first, last)` that satisfies specific criteria:

1) `find` searches for an element equal to `value`

2) `find_if` searches for an element for which predicate `p` returns `true`

3) `find_if_not` searches for an element for which predicate `q` returns `false`

Parameters

first, last - the range of elements to examine

value - value to compare the elements to

p - unary predicate which returns `true` for the required element.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type &a);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it.
The type `Type` must be such that an object of type `InputIt` can be dereferenced and then implicitly converted to `Type`.

q - unary predicate which returns `false` for the required element.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type &a);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it.
The type `Type` must be such that an object of type `InputIt` can be dereferenced and then implicitly converted to `Type`.

Type requirements

- `InputIt` must meet the requirements of `InputIterator`.
- `UnaryPredicate` must meet the requirements of `Predicate`.

Return value

Iterator to the first element satisfying the condition or `last` if no such element is found.

Complexity

At most `last` - `first` applications of the predicate

Possible implementation

First version

```
template<class InputIt, class T>
InputIt find(InputIt first, InputIt last, const T& value)
{
    for (; first != last; ++first) {
        if (*first == value) {
            return first;
        }
    }
    return last;
}
```

Second version

```
template<class InputIt, class UnaryPredicate>
InputIt find_if(InputIt first, InputIt last, UnaryPredicate p)
{
    for (; first != last; ++first) {
        if (p(*first)) {
            return first;
        }
    }
    return last;
}
```

Third version

```
template<class InputIt, class UnaryPredicate>
InputIt find_if_not(InputIt first, InputIt last, UnaryPredicate q)
{
    for (; first != last; ++first) {
        if (!q(*first)) {
            return first;
        }
    }
    return last;
}
```

If you do not have C++11, an equivalent to **`std::find_if_not`** is to use **`std::find_if`** with the negated predicate.

```
template<class InputIt, class UnaryPredicate>
InputIt find_if_not(InputIt first, InputIt last, UnaryPredicate q)
{
    return std::find_if(first, last, std::not1(q));
}
```

Example

The following example finds an integer in a vector of integers.

Run this code

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>

int main()
{
    int n1 = 3;
    int n2 = 5;

    std::vector<int> v{0, 1, 2, 3, 4};

    auto result1 = std::find(std::begin(v), std::end(v), n1);
    auto result2 = std::find(std::begin(v), std::end(v), n2);

    if (result1 != std::end(v)) {
        std::cout << "v contains: " << n1 << '\n';
    } else {
        std::cout << "v does not contain: " << n1 << '\n';
    }

    if (result2 != std::end(v)) {
        std::cout << "v contains: " << n2 << '\n';
    } else {
        std::cout << "v does not contain: " << n2 << '\n';
    }
}
```

Output:

```
v contains: 3
v does not contain: 5
```

See also

adjacent_find	finds the first two adjacent items that are equal (or satisfy a given predicate) (function template)
find_end	finds the last sequence of elements in a certain range (function template)
find_first_of	searches for any one of a set of elements (function template)
mismatch	finds the first position where two ranges differ (function template)
search	searches for a range of elements (function template)
std::experimental::parallel::find (parallelism TS)	parallelized version of `std::find` (function template)
std::experimental::parallel::find_if (parallelism TS)	parallelized version of `std::find_if` (function template)
std::experimental::parallel::find_if_not (parallelism TS)	parallelized version of `std::find_if_not` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/find&oldid=79876"

std::find_end

Defined in header `<algorithm>`

```
template< class ForwardIt1, class ForwardIt2 >
ForwardIt1 find_end( ForwardIt1 first, ForwardIt1 last,                         (1)
                     ForwardIt2 s_first, ForwardIt2 s_last );
```

```
template< class ForwardIt1, class ForwardIt2, class BinaryPredicate >
ForwardIt1 find_end( ForwardIt1 first, ForwardIt1 last,                         (2)
                     ForwardIt2 s_first, ForwardIt2 s_last, BinaryPredicate p );
```

Searches for the last subsequence of elements `[s_first, s_last)` in the range `[first, last)`. The first version uses `operator==` to compare the elements, the second version uses the given binary predicate `p`.

Parameters

first, last - the range of elements to examine

s_first, s_last - the range of elements to search for

p - binary predicate which returns `true` if the elements should be treated as equal.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that objects of types `ForwardIt1` and `ForwardIt2` can be dereferenced and then implicitly converted to `Type1` and `Type2` respectively.

Type requirements

- `ForwardIt1` must meet the requirements of `ForwardIterator`.
- `ForwardIt2` must meet the requirements of `ForwardIterator`.

Return value

Iterator to the beginning of last subsequence `[s_first, s_last)` in range `[first, last)`.

If no such subsequence is found, `last` is returned.	(until C++11)
If `[s_first, s_last)` is empty or if no such subsequence is found, `last` is returned.	(since C++11)

Complexity

Does at most `S*(N-S+1)` comparisons where `S = distance(s_first, s_last)` and `N = distance(first, last)`.

Possible implementation

First version

```
template<class ForwardIt1, class ForwardIt2>
ForwardIt1 find_end(ForwardIt1 first, ForwardIt1 last,
                    ForwardIt2 s_first, ForwardIt2 s_last)
```

```
{
    if (s_first == s_last)
        return last;
    ForwardIt1 result = last;
    while (1) {
        ForwardIt1 new_result = std::search(first, last, s_first, s_last);
        if (new_result == last) {
            return result;
        } else {
            result = new_result;
            first = result;
            ++first;
        }
    }
    return result;
}
```

Second version

```
template<class ForwardIt1, class ForwardIt2, class BinaryPredicate>
ForwardIt1 find_end(ForwardIt1 first, ForwardIt1 last,
                    ForwardIt2 s_first, ForwardIt2 s_last,
                    BinaryPredicate p)
{
    if (s_first == s_last)
        return last;
    ForwardIt1 result = last;
    while (1) {
        ForwardIt1 new_result = std::search(first, last, s_first, s_last, p);
        if (new_result == last) {
            return result;
        } else {
            result = new_result;
            first = result;
            ++first;
        }
    }
    return result;
}
```

Example

The following code uses `find_end()` to search for two different sequences of numbers.

Run this code

```
#include <algorithm>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v{1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4};
    std::vector<int>::iterator result;

    std::vector<int> t1{1, 2, 3};

    result = std::find_end(v.begin(), v.end(), t1.begin(), t1.end());
    if (result == v.end()) {
        std::cout << "subsequence not found\n";
    } else {
        std::cout << "last subsequence is at: "
                  << std::distance(v.begin(), result) << "\n";
    }
    std::vector<int> t2{4, 5, 6};
    result = std::find_end(v.begin(), v.end(), t2.begin(), t2.end());
    if (result == v.end()) {
        std::cout << "subsequence not found\n";
    } else {
        std::cout << "last subsequence is at: "
                  << std::distance(v.begin(), result) << "\n";
    }
}
```

Output:

```
last subsequence is at: 8
subsequence not found
```

See also

search	searches for a range of elements (function template)
includes	returns true if one set is a subset of another (function template)
adjacent_find	finds the first two adjacent items that are equal (or satisfy a given predicate) (function template)
find **find_if** **find_if_not** (C++11)	finds the first element satisfying specific criteria (function template)
find_first_of	searches for any one of a set of elements (function template)
search_n	searches for a number consecutive copies of an element in a range (function template)
std::experimental::parallel::find_end (parallelism TS)	parallelized version of `std::find_end` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/find_end&oldid=79877"

std::find_first_of

Defined in header `<algorithm>`

```
template< class ForwardIt1, class ForwardIt2 >
ForwardIt1 find_first_of( ForwardIt1 first, ForwardIt1 last,
                          ForwardIt2 s_first, ForwardIt2 s_last );
```
(until C++11)

(1)

```
template< class InputIt, class ForwardIt >
InputIt find_first_of( InputIt first, InputIt last,
                       ForwardIt s_first, ForwardIt s_last );
```
(since C++11)

```
template< class ForwardIt1, class ForwardIt2, class BinaryPredicate >
ForwardIt1 find_first_of( ForwardIt1 first, ForwardIt1 last,
                          ForwardIt2 s_first, ForwardIt2 s_last, BinaryPredicate p );
```
(until C++11)

(2)

```
template< class InputIt, class ForwardIt, class BinaryPredicate >
InputIt find_first_of( InputIt first, InputIt last,
                       ForwardIt s_first, ForwardIt s_last, BinaryPredicate p );
```
(since C++11)

Searches the range `[first, last)` for any of the elements in the range `[s_first, s_last)`. The first version uses `operator==` to compare the elements, the second version uses the given binary predicate `p`.

Parameters

`first, last` - the range of elements to examine

`s_first, s_last` - the range of elements to search for

`p` - binary predicate which returns `true` if the elements should be treated as equal.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that objects of types `ForwardIt1` and `ForwardIt2` can be dereferenced and then implicitly converted to `Type1` and `Type2` respectively.

Type requirements

- `InputIt` must meet the requirements of `InputIterator`.
- `ForwardIt1` must meet the requirements of `ForwardIterator`.
- `ForwardIt2` must meet the requirements of `ForwardIterator`.

Return value

Iterator to the first element in the range `[first, last)` that is equal to an element from the range `[s_first; s_last)`. If no such element is found, `last` is returned.

Complexity

Does at most `(S*N)` comparisons where `S = distance(s_first, s_last)` and `N = distance(first, last)`.

Possible implementation

First version

```
template<class InputIt, class ForwardIt>
InputIt find_first_of(InputIt first, InputIt last,
                      ForwardIt s_first, ForwardIt s_last)
{
    for (; first != last; ++first) {
        for (ForwardIt it = s_first; it != s_last; ++it) {
            if (*first == *it) {
                return first;
            }
        }
    }
    return last;
}
```

Second version

```
template<class InputIt, class ForwardIt, class BinaryPredicate>
InputIt find_first_of(InputIt first, InputIt last,
                      ForwardIt s_first, ForwardIt s_last,
                      BinaryPredicate p)
{
    for (; first != last; ++first) {
        for (ForwardIt it = s_first; it != s_last; ++it) {
            if (p(*first, *it)) {
                return first;
            }
        }
    }
    return last;
}
```

Example

The following code searches for any of specified integers in a vector of integers:

Run this code

```
#include <algorithm>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v{0, 2, 3, 25, 5};
    std::vector<int> t{3, 19, 10, 2};

    auto result = std::find_first_of(v.begin(), v.end(), t.begin(), t.end());

    if (result == v.end()) {
        std::cout << "no elements of v were equal to 3, 19, 10 or 2\n";
    } else {
        std::cout << "found a match at "
                  << std::distance(v.begin(), result) << "\n";
    }
 }
```

Output:

```
found a match at 1
```

See also

find **find_if** **find_if_not** (C++11)	finds the first element satisfying specific criteria (function template)
std::experimental::parallel::find_first_of (parallelism TS)	parallelized version of `std::find_first_of` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/find_first_of&oldid=79878"

std::adjacent_find

Defined in header `<algorithm>`

`template< class ForwardIt >`
`ForwardIt adjacent_find( ForwardIt first, ForwardIt last );`
	(1)

`template< class ForwardIt, class BinaryPredicate>`
`ForwardIt adjacent_find( ForwardIt first, ForwardIt last, BinaryPredicate p );`
	(2)

Searches the range `[first, last)` for two consecutive identical elements. The first version uses `operator==` to compare the elements, the second version uses the given binary predicate `p`.

Parameters

first, last - the range of elements to examine

p - binary predicate which returns `true` if the elements should be treated as equal.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `ForwardIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `ForwardIt` must meet the requirements of `ForwardIterator`.

Return value

an iterator to the first of the first pair of identical elements, that is, the first iterator `it` such that `*it == *(it+1)` for the first version or `p(*it, *(it + 1)) != false` for the second version.

If no such elements are found, `last` is returned

Complexity

Exactly the smaller of `std::distance(first, result) + 1` and `std::distance(first, last) - 1` applications of the predicate where `result` is the return value.

Possible implementation

First version

```
template<class ForwardIt>
ForwardIt adjacent_find(ForwardIt first, ForwardIt last)
{
    if (first == last) {
        return last;
    }
    ForwardIt next = first;
    ++next;
    for (; next != last; ++next, ++first) {
        if (*first == *next) {
            return first;
        }
    }
    return last;
}
```

Second version

```
template<class ForwardIt, class BinaryPredicate>
ForwardIt adjacent_find(ForwardIt first, ForwardIt last,
                        BinaryPredicate p)
{
    if (first == last) {
        return last;
    }
    ForwardIt next = first;
    ++next;
    for (; next != last; ++next, ++first) {
        if (p(*first, *next)) {
            return first;
        }
    }
    return last;
}
```

Example

Run this code

```
#include <algorithm>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v1{0, 1, 2, 3, 40, 40, 41, 41, 5};

    auto i1 = std::adjacent_find(v1.begin(), v1.end());

    if (i1 == v1.end()) {
        std::cout << "no matching adjacent elements\n";
    } else {
        std::cout << "the first adjacent pair of equal elements at: "
                  << std::distance(v1.begin(), i1) << '\n';
    }

    auto i2 = std::adjacent_find(v1.begin(), v1.end(), std::greater<int>());
    if (i2 == v1.end()) {
        std::cout << "The entire vector is sorted in ascending order\n";
    } else {
        std::cout << "The last element in the non-decreasing subsequence is at: "
                  << std::distance(v1.begin(), i2) << '\n';
    }
}
```

Output:

```
The first adjacent pair of equal elements at: 4
The last element in the non-decreasing subsequence is at: 7
```

See also

unique	removes consecutive duplicate elements in a range (function template)
std::experimental::parallel::ajacent_find (parallelism TS)	parallelized version of `std::ajacent_find` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/adjacent_find&oldid=79864"

std::search

Defined in header `<algorithm>`

```
template< class ForwardIt1, class ForwardIt2 >
ForwardIt1 search( ForwardIt1 first, ForwardIt1 last,
                   ForwardIt2 s_first, ForwardIt2 s_last );
```
(1)

```
template< class ForwardIt1, class ForwardIt2, class BinaryPredicate >
ForwardIt1 search( ForwardIt1 first, ForwardIt1 last,
                   ForwardIt2 s_first, ForwardIt2 s_last, BinaryPredicate p );
```
(2)

Searches for the first occurrence of the subsequence of elements `[s_first, s_last)` in the range `[first, last - (s_last - s_first))`. The first version uses `operator==` to compare the elements, the second version uses the given binary predicate `p`.

Parameters

first, last - the range of elements to examine

s_first, s_last - the range of elements to search for

p - binary predicate which returns `true` if the elements should be treated as equal.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that objects of types `ForwardIt1` and `ForwardIt2` can be dereferenced and then implicitly converted to `Type1` and `Type2` respectively.

Type requirements

- `ForwardIt1, ForwardIt2` must meet the requirements of `ForwardIterator`.

Return value

Iterator to the beginning of first subsequence `[s_first, s_last)` in the range `[first, last - (s_last - s_first))`. If no such subsequence is found, `last` is returned.
If `[s_first, s_last)` is empty, `first` is returned. (since C++11)

Complexity

At most `S*N` comparisons where `S = std::distance(s_first, s_last)` and `N = std::distance(first, last)`.

Possible implementation

First version

```
template<class ForwardIt1, class ForwardIt2>
ForwardIt1 search(ForwardIt1 first, ForwardIt1 last,
                        ForwardIt2 s_first, ForwardIt2 s_last)
{
    for (; ; ++first) {
        ForwardIt1 it = first;
        for (ForwardIt2 s_it = s_first; ; ++it, ++s_it) {
            if (s_it == s_last) {
                return first;
            }
            if (it == last) {
                return last;
            }
            if (!(*it == *s_it)) {
                break;
            }
        }
    }
}
```

Second version

```
template<class ForwardIt1, class ForwardIt2, class BinaryPredicate>
ForwardIt1 search(ForwardIt1 first, ForwardIt1 last,
                        ForwardIt2 s_first, ForwardIt2 s_last,
                        BinaryPredicate p)
{
    for (; ; ++first) {
        ForwardIt1 it = first;
        for (ForwardIt2 s_it = s_first; ; ++it, ++s_it) {
            if (s_it == s_last) {
                return first;
            }
            if (it == last) {
                return last;
            }
            if (!p(*it, *s_it)) {
                break;
            }
        }
    }
}
```

Example

Run this code

```
#include <string>
#include <algorithm>
#include <iostream>
#include <vector>

template<typename Container>
bool in_quote(const Container& cont, const std::string& s)
{
    return std::search(cont.begin(), cont.end(), s.begin(), s.end()) != cont.end();
}

int main()
{
    std::string str = "why waste time learning, when ignorance is instantaneous?";
    // str.find() can be used as well
    std::cout << std::boolalpha << in_quote(str, "learning") << '\n'
                                << in_quote(str, "lemming")  << '\n';

    std::vector<char> vec(str.begin(), str.end());
    std::cout << std::boolalpha << in_quote(vec, "learning") << '\n'
                                << in_quote(vec, "lemming")  << '\n';
}
```

Output:

```
true
false
true
false
```

See also

find_end	finds the last sequence of elements in a certain range (function template)
includes	returns true if one set is a subset of another (function template)
equal	determines if two sets of elements are the same (function template)
find **find_if** **find_if_not** (C++11)	finds the first element satisfying specific criteria (function template)
lexicographical_compare	returns true if one range is lexicographically less than another (function template)
mismatch	finds the first position where two ranges differ (function template)
search_n	searches for a number consecutive copies of an element in a range (function template)
std::experimental::search (library fundamentals TS)	applies a searcher to a sequence (function template)
std::experimental::parallel::search (parallelism TS)	parallelized version of `std::search` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/search&oldid=79912"

std::search_n

Defined in header `<algorithm>`

```
template< class ForwardIt, class Size, class T >
ForwardIt search_n( ForwardIt first, ForwardIt last, Size count, const T& value );    (1)
```

```
template< class ForwardIt, class Size, class T, class BinaryPredicate >
ForwardIt search_n( ForwardIt first, ForwardIt last, Size count, const T& value,      (2)
                    BinaryPredicate p );
```

Searches the range `[first, last)` for the first sequence of count identical elements, each equal to the given value value. The first version uses `operator==` to compare the elements, the second version uses the given binary predicate `p`.

Parameters

first, last - the range of elements to examine

count - the length of the sequence to search for

value - the value of the elements to search for

p - binary predicate which returns `true` if the elements should be treated as equal.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it.
The type `Type1` must be such that an object of type `ForwardIt` can be dereferenced and then implicitly converted to `Type1`. The type `Type2` must be such that an object of type `T` can be implicitly converted to `Type2`.

Type requirements

- `ForwardIt` must meet the requirements of `ForwardIterator`.

Return value

Iterator to the beginning of the found sequence in the range `[first, last)`. If no such sequence is found, `last` is returned.

Complexity

At most `last - first` applications of the predicate.

Possible implementation

First version

```
template<class ForwardIt, class Size, class T>
ForwardIt search_n(ForwardIt first, ForwardIt last,
                    Size count, const T& value)
{
    for(; first != last; ++first) {
        if (!(*first == value)) {
            continue;
        }
        ForwardIt candidate = first;
        Size cur_count = 0;

        while (true) {
            ++cur_count;
            if (cur_count == count) {
                // success
                return candidate;
            }
            ++first;
            if (first == last) {
                // exhausted the list
                return last;
            }
            if (!(*first == value)) {
                // too few in a row
                break;
            }
        }
    }
    return last;
}
```

Second version

```
template<class ForwardIt, class Size, class T, class BinaryPredicate>
ForwardIt search_n(ForwardIt first, ForwardIt last,
                    Size count, const T& value, BinaryPredicate p)
{
    for(; first != last; ++first) {
        if (!p(*first, value)) {
            continue;
        }

        ForwardIt candidate = first;
        Size cur_count = 0;

        while (true) {
            ++cur_count;
            if (cur_count == count) {
                // success
                return candidate;
            }
            ++first;
            if (first == last) {
                // exhausted the list
                return last;
            }
            if (!p(*first, value)) {
                // too few in a row
                break;
            }
        }
    }
    return last;
}
```

Example

Run this code

```
#include <iostream>
#include <algorithm>
```

```
#include <iterator>

template <class Container, class Size, class T>
bool consecutive_values(const Container& c, Size count, const T& v)
{
  return std::search_n(std::begin(c),std::end(c),count,v) != std::end(c);
}

int main()
{
   const char sequence[] = "1001010100010101001010101";

   std::cout << std::boolalpha;
   std::cout << "Has 4 consecutive zeros: "
             << consecutive_values(sequence,4,'0') << '\n';
   std::cout << "Has 3 consecutive zeros: "
             << consecutive_values(sequence,3,'0') << '\n';
}
```

Output:

```
Has 4 consecutive zeros: false
Has 3 consecutive zeros: true
```

See also

find_end	finds the last sequence of elements in a certain range (function template)
find **find_if** **find_if_not** (C++11)	finds the first element satisfying specific criteria (function template)
search	searches for a range of elements (function template)
std::experimental::parallel::search_n (parallelism TS)	parallelized version of `std::search_n` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/search_n&oldid=79913"

std::copy, std::copy_if

Defined in header `<algorithm>`

```
template< class InputIt, class OutputIt >
OutputIt copy( InputIt first, InputIt last, OutputIt d_first );
```

(1)

```
template< class InputIt, class OutputIt, class UnaryPredicate >
OutputIt copy_if( InputIt first, InputIt last,
                  OutputIt d_first,
                  UnaryPredicate pred );
```

(2) (since C++11)

Copies the elements in the range, defined by `[first, last)`, to another range beginning at `d_first`. The second function only copies the elements for which the predicate `pred` returns `true`. The order of the elements that are not removed is preserved.

For `std::copy`, the behavior is undefined if `d_first` is within the range `[first, last)`. In this case, `std::copy_backward` may be used instead.

For `std::copy_if`, the behavior is undefined if the source and the destination ranges overlap.

Parameters

`first, last` - the range of elements to copy

`d_first` - the beginning of the destination range.

`pred` - unary predicate which returns `true` for the required elements.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type &a);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it.
The type `Type` must be such that an object of type `InputIt` can be dereferenced and then implicitly converted to `Type`.

Type requirements

- `InputIt` must meet the requirements of `InputIterator`.
- `OutputIt` must meet the requirements of `OutputIterator`.
- `UnaryPredicate` must meet the requirements of `Predicate`.

Return value

Output iterator to the element in the destination range, one past the last element copied.

Complexity

1) Exactly `last - first` assignments

2) Exactly `last - first` applications of the predicate

Notes

In practice, implementations of `std::copy` avoid multiple assignments and use bulk copy functions such as `std::memmove` if the value type is `TriviallyCopyable`

When copying overlapping ranges, `std::copy` is appropriate when copying to the left (beginning of the destination range is outside the source range) while `std::copy_backward` is appropriate when copying to the right (end of the destination range is outside the source range).

Possible implementation

First version

```
template<class InputIt, class OutputIt>
OutputIt copy(InputIt first, InputIt last,
              OutputIt d_first)
{
    while (first != last) {
        *d_first++ = *first++;
    }
    return d_first;
}
```

Second version

```
template<class InputIt, class OutputIt, class UnaryPredicate>
OutputIt copy_if(InputIt first, InputIt last,
                 OutputIt d_first, UnaryPredicate pred)
{
    while (first != last) {
        if (pred(*first))
            *d_first++ = *first;
        first++;
    }
    return d_first;
}
```

Example

The following code uses copy to both copy the contents of one vector to another and to display the resulting vector:

Run this code

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <iterator>
#include <numeric>

int main()
{
    std::vector<int> from_vector(10);
    std::iota(from_vector.begin(), from_vector.end(), 0);

    std::vector<int> to_vector;
    std::copy(from_vector.begin(), from_vector.end(),
              std::back_inserter(to_vector));
// or, alternatively,
//  std::vector<int> to_vector(from_vector.size());
//  std::copy(from_vector.begin(), from_vector.end(), to_vector.begin());
// either way is equivalent to
//  std::vector<int> to_vector = from_vector;

    std::cout << "to_vector contains: ";

    std::copy(to_vector.begin(), to_vector.end(),
              std::ostream_iterator<int>(std::cout, " "));
    std::cout << '\n';
}
```

Output:

```
to_vector contains: 0 1 2 3 4 5 6 7 8 9
```

See also

copy_backward	copies a range of elements in backwards order (function template)
copy_n (C++11)	copies a number of elements to a new location (function template)
fill	assigns a range of elements a certain value (function template)
remove_copy **remove_copy_if**	copies a range of elements omitting those that satisfy specific criteria (function template)
std::experimental::parallel::copy (parallelism TS)	parallelized version of `std::copy` (function template)
std::experimental::parallel::copy_if (parallelism TS)	parallelized version of `std::copy_if` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/copy&oldid=79870"

std::copy_n

Defined in header `<algorithm>`

```
template< class InputIt, class Size, class OutputIt >
OutputIt copy_n( InputIt first, Size count, OutputIt result );
```
(since C++11)

Copies exactly `count` values from the range beginning at `first` to the range beginning at `result`, if `count>0`. Does nothing otherwise.

Parameters

first	-	the beginning of the range of elements to copy from
count	-	number of the elements to copy
result	-	the beginning of the destination range

Type requirements

- `InputIt` must meet the requirements of `InputIterator`.
- `OutputIt` must meet the requirements of `OutputIterator`.

Return value

Iterator in the destination range, pointing past the last element copied if `count>0` or `result` otherwise.

Complexity

Exactly `count` assignments, if `count>0`.

Possible implementation

```
template< class InputIt, class Size, class OutputIt>
OutputIt copy_n(InputIt first, Size count, OutputIt result)
{
    if (count > 0) {
        *result++ = *first;
        for (Size i = 1; i < count; ++i) {
            *result++ = *++first;
        }
    }
    return result;
}
```

Example

Run this code

```
#include <iostream>
#include <string>
#include <algorithm>
#include <iterator>

int main()
{
    std::string in = "1234567890";
    std::string out;
    std::copy_n(in.begin(), 4, std::back_inserter(out));
    std::cout << out << '\n';
}
```

Output:

```
1234
```

See also

copy **copy_if** (C++11)	copies a range of elements to a new location (function template)
std::experimental::parallel::copy_n (parallelism TS)	parallelized version of `std::copy_n` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/copy_n&oldid=79871"

std::copy_backward

Defined in header <algorithm>

```
template< class BidirIt1, class BidirIt2 >
BidirIt2 copy_backward( BidirIt1 first, BidirIt1 last, BidirIt2 d_last );
```

Copies the elements from the range, defined by `[first, last)`, to another range ending at `d_last`. The elements are copied in reverse order (the last element is copied first), but their relative order is preserved.

The behavior is undefined if `d_last` is within `(first, last]`. `std::copy` must be used instead of `std::copy_backward` in that case.

Parameters

first, last - the range of the elements to copy

d_last - end of the destination range..

Type requirements

- `BidirIt` must meet the requirements of `BidirectionalIterator`.

Return value

iterator to the last element copied.

Complexity

Exactly `last - first` assignments.

Notes

When copying overlapping ranges, `std::copy` is appropriate when copying to the left (beginning of the destination range is outside the source range) while `std::copy_backward` is appropriate when copying to the right (end of the destination range is outside the source range).

Possible implementation

```
template< class BidirIt1, class BidirIt2 >
BidirIt2 copy_backward(BidirIt1 first, BidirIt1 last, BidirIt2 d_last)
{
    while (first != last) {
        *(--d_last) = *(--last);
    }
    return d_last;
}
```

Example

Run this code

```
#include <algorithm>
#include <iostream>
#include <vector>
```

```
int main()
{
    std::vector<int> from_vector;
    for (int i = 0; i < 10; i++) {
        from_vector.push_back(i);
    }

    std::vector<int> to_vector(15);

    std::copy_backward(from_vector.begin(), from_vector.end(), to_vector.end());

    std::cout << "to_vector contains: ";
    for (unsigned int i = 0; i < to_vector.size(); i++) {
        std::cout << to_vector[i] << " ";
    }
 }
```

Output:

```
to_vector contains: 0 0 0 0 0 0 1 2 3 4 5 6 7 8 9
```

See also

copy **copy_if** (C++11)	copies a range of elements to a new location (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/copy_backward&oldid=69123"

std::move

Defined in header `<algorithm>`

```
template< class InputIt, class OutputIt >
OutputIt move( InputIt first, InputIt last, OutputIt d_first );
```
(since C++11)

Moves the elements in the range `[first, last)`, to another range beginning at `d_first`. After this operation the elements in the moved-from range will still contain valid values of the appropriate type, but not necessarily the same values as before the move.

Parameters

first, last	-	the range of elements to move
d_first	-	the beginning of the destination range. If `d_first` is within `[first, last)`, `std::move_backward` must be used instead of `std::move`.

Type requirements

- `InputIt` must meet the requirements of `InputIterator`.
- `OutputIt` must meet the requirements of `OutputIterator`.

Return value

Output iterator to the element past the last element moved (`d_first + (last - first)`)

Complexity

Exactly `last - first` move assignments.

Possible implementation

```
template<class InputIt, class OutputIt>
OutputIt move(InputIt first, InputIt last, OutputIt d_first)
{
    while (first != last) {
        *d_first++ = std::move(*first++);
    }
    return d_first;
}
```

Notes

When moving overlapping ranges, `std::move` is appropriate when moving to the left (beginning of the destination range is outside the source range) while `std::move_backward` is appropriate when moving to the right (end of the destination range is outside the source range).

Example

The following code moves thread objects (which themselves are not copyable) from one container to another.

Run this code

```
#include <iostream>
#include <vector>
```

```
#include <list>
#include <iterator>
#include <thread>
#include <chrono>

void f(int n)
{
    std::this_thread::sleep_for(std::chrono::seconds(n));
    std::cout << "thread " << n << " ended" << '\n';
}

int main()
{
    std::vector<std::thread> v;
    v.emplace_back(f, 1);
    v.emplace_back(f, 2);
    v.emplace_back(f, 3);
    std::list<std::thread> l;
    // copy() would not compile, because std::thread is noncopyable

    std::move(v.begin(), v.end(), std::back_inserter(l));
    for (auto& t : l) t.join();
}
```

Output:

```
thread 1 ended
thread 2 ended
thread 3 ended
```

See also

move_backward (C++11)	moves a range of elements to a new location in backwards order (function template)
move (C++11)	obtains an rvalue reference (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/move&oldid=66372"

std::move_backward

Defined in header `<algorithm>`

```
template< class BidirIt1, class BidirIt2 >
BidirIt2 move_backward( BidirIt1 first, BidirIt1 last, BidirIt2 d_last );
```

(since C++11)

Moves the elements from the range `[first, last)`, to another range ending at `d_last`. The elements are moved in reverse order (the last element is moved first), but their relative order is preserved.

The behavior is undefined if `d_last` is within `(first, last]`. `std::move` must be used instead of `std::move_backward` in that case.

Parameters

first, last	-	the range of the elements to move
d_last	-	end of the destination range

Type requirements

- `BidirIt1, BidirIt2` must meet the requirements of `BidirectionalIterator`.

Return value

Iterator in the destination range, pointing at the last element moved.

Complexity

Exactly `last - first` move assignments.

Possible implementation

```
template< class BidirIt1, class BidirIt2 >
BidirIt2 move_backward(BidirIt1 first,
                                     BidirIt1 last,
                                     BidirIt2 d_last)
{
    while (first != last) {
        *(--d_last) = std::move(*(--last));
    }
    return d_last;
}
```

Notes

When moving overlapping ranges, `std::move` is appropriate when moving to the left (beginning of the destination range is outside the source range) while `std::move_backward` is appropriate when moving to the right (end of the destination range is outside the source range).

Example

Run this code

```
#include <algorithm>
#include <vector>
```

```
#include <string>
#include <iostream>

int main()
{
    std::vector<std::string> src{"foo", "bar", "baz"};
    std::vector<std::string> dest(src.size());

    std::cout << "src: ";
    for (const auto &s : src)
    {
        std::cout << s << ' ';
    }
    std::cout << "\ndest: ";
    for (const auto &s : dest)
    {
        std::cout << s << ' ';
    }
    std::cout << '\n';

    std::move_backward(src.begin(), src.end(), dest.end());

    std::cout << "src: ";
    for (const auto &s : src)
    {
        std::cout << s << ' ';
    }
    std::cout << "\ndest: ";
    for (const auto &s : dest)
    {
        std::cout << s << ' ';
    }
    std::cout << '\n';
}
```

Output:

```
src: foo bar baz
dest:
src:
dest: foo bar baz
```

See also

move (C++11)	moves a range of elements to a new location (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/move_backward&oldid=65287"

std::fill

Defined in header `<algorithm>`

```
template< class ForwardIt, class T >
void fill( ForwardIt first, ForwardIt last, const T& value );
```

Assigns the given `value` to the elements in the range `[first, last)`.

Parameters

`first, last`	-	the range of elements to modify
`value`	-	the value to be assigned

Type requirements

- `ForwardIt` must meet the requirements of `ForwardIterator`.

Return value

(none)

Complexity

Exactly `last - first` assignments.

Possible implementation

```
template< class ForwardIt, class T >
void fill(ForwardIt first, ForwardIt last, const T& value)
{
    for (; first != last; ++first) {
        *first = value;
    }
}
```

Example

The following code uses `fill()` to set all of the elements of a vector of integers to -1:

Run this code

```
#include <algorithm>
#include <vector>
#include <iostream>

int main()
{
    std::vector<int> v{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    std::fill(v.begin(), v.end(), -1);

    for (auto elem : v) {
        std::cout << elem << " ";
    }
    std::cout << "\n";
}
```

Output:

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

See also

fill_n	assigns a value to a number of elements (function template)
copy **copy_if** (C++11)	copies a range of elements to a new location (function template)
generate	saves the result of a function in a range (function template)
transform	applies a function to a range of elements (function template)
std::experimental::parallel::fill (parallelism TS)	parallelized version of `std::fill` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/fill&oldid=79874"

std::fill_n

Defined in header `<algorithm>`

`template< class OutputIt, class Size, class T >`
`void fill_n( OutputIt first, Size count, const T& value );`
	(until C++11)

`template< class OutputIt, class Size, class T >`
`OutputIt fill_n( OutputIt first, Size count, const T& value );`
	(since C++11)

Assigns the given value `value` to the first `count` elements in the range beginning at `first` if `count > 0`. Does nothing otherwise.

Parameters

`first` - the beginning of the range of elements to modify

`count` - number of elements to modify

`value` - the value to be assigned

Type requirements

- `OutputIt` must meet the requirements of `OutputIterator`.

Return value

(none)	(until C++11)
Iterator one past the last element assigned if `count > 0`, `first` otherwise.	(since C++11)

Complexity

Exactly `count` assignments, for `count > 0`.

Possible implementation

```
template<class OutputIt, class Size, class T>
OutputIt fill_n(OutputIt first, Size count, const T& value)
{
    for (Size i = 0; i < count; i++) {
        *first++ = value;
    }
    return first;
}
```

Example

The following code uses `fill_n()` to assign -1 to the first half of a vector of integers:

Run this code

```
#include <algorithm>
#include <vector>
#include <iostream>
#include <iterator>

int main()
{
    std::vector<int> v1{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    std::fill_n(v1.begin(), 5, -1);

    std::copy(begin(v1), end(v1), std::ostream_iterator<int>(std::cout, " "));
    std::cout << "\n";
}
```

Output:

```
-1 -1 -1 -1 -1 5 6 7 8 9
```

See also

fill	assigns a range of elements a certain value (function template)
std::experimental::parallel::fill_n (parallelism TS)	parallelized version of `std::fill_n` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/fill_n&oldid=79875"

std::transform

Defined in header `<algorithm>`

```
template< class InputIt, class OutputIt, class UnaryOperation >
OutputIt transform( InputIt first1, InputIt last1, OutputIt d_first,
                    UnaryOperation unary_op );
```
(1)

```
template< class InputIt1, class InputIt2, class OutputIt, class BinaryOperation >
OutputIt transform( InputIt1 first1, InputIt1 last1, InputIt2 first2,
                    OutputIt d_first, BinaryOperation binary_op );
```
(2)

`std::transform` applies the given function to a range and stores the result in another range, beginning at `d_first`.

In the first version unary operation `unary_op` is applied to the range defined by `[first1, last1)`. In the second version the binary operation `binary_op` is applied to pairs of elements from two ranges: one defined by `[first1, last1)` and the other beginning at `first2`.

`unary_op` and `binary_op` must not have side effects.	(until C++11)
`unary_op` and `binary_op` must not invalidate any iterators, including the end iterators, or modify any elements of the ranges involved.	(since C++11)

Parameters

`first1, last1` - the first range of elements to transform

`first2` - the beginning of the second range of elements to transform

`d_first` - the beginning of the destination range, may be equal to `first1` or `first2`

`unary_op` - unary operation function object that will be applied.

The signature of the function should be equivalent to the following:

```
Ret fun(const Type &a);
```

The signature does not need to have `const &`.
The type `Type` must be such that an object of type `InputIt` can be dereferenced and then implicitly converted to `Type`. The type `Ret` must be such that an object of type `OutputIt` can be dereferenced and assigned a value of type `Ret`.

`binary_op` - binary operation function object that will be applied.

The signature of the function should be equivalent to the following:

```
Ret fun(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`.
The types `Type1` and `Type2` must be such that objects of types `InputIt1` and `InputIt2` can be dereferenced and then implicitly converted to `Type1` and `Type2` respectively. The type `Ret` must be such that an object of type `OutputIt` can be dereferenced and assigned a value of type `Ret`.

Type requirements

- `InputIt` must meet the requirements of `InputIterator`.
- `InputIt1` must meet the requirements of `InputIterator`.
- `InputIt2` must meet the requirements of `InputIterator`.
- `OutputIt` must meet the requirements of `OutputIterator`.

Return value

Output iterator to the element past the last element transformed.

Complexity

1) Exactly `std::distance(first1, last1)` applications of `unary_op`

2) Exactly `std::distance(first1, last1)` applications of `binary_op`

Possible implementation

First version

```
template<class InputIt, class OutputIt, class UnaryOperation>
OutputIt transform(InputIt first1, InputIt last1, OutputIt d_first,
                   UnaryOperation unary_op)
{
    while (first1 != last1) {
        *d_first++ = unary_op(*first1++);
    }
    return d_first;
}
```

Second version

```
template<class InputIt1, class InputIt2,
         class OutputIt, class BinaryOperation>
OutputIt transform(InputIt1 first1, InputIt1 last1, InputIt2 first2,
                   OutputIt d_first, BinaryOperation binary_op)
{
    while (first1 != last1) {
        *d_first++ = binary_op(*first1++, *first2++);
    }
    return d_first;
}
```

Notes

`std::transform` does not guarantee in-order application of `unary_op` or `binary_op`. To apply a function to a sequence in-order or to apply a function that modifies the elements of a sequence, use `std::for_each`

Example

The following code uses transform to convert a string to uppercase using the toupper function:

Run this code

```
#include <string>
#include <ctype.h>
#include <algorithm>
#include <functional>
#include <iostream>

int main()
{
    std::string s("hello");
    std::transform(s.begin(), s.end(), s.begin(), ::toupper);
    std::cout << s;
}
```

Output:

```
HELLO
```

See also

`for_each`	applies a function to a range of elements (function template)
`std::experimental::parallel::transform` (parallelism TS)	parallelized version of `std::transform` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/transform&oldid=79922"

std::generate

Defined in header `<algorithm>`

```
template< class ForwardIt, class Generator >
void generate( ForwardIt first, ForwardIt last, Generator g );
```

Assigns each element in range `[first, last)` a value generated by the given function object `g`.

Parameters

`first, last` - the range of elements to generate

`g` - generator function object that will be called.

The signature of the function should be equivalent to the following:

```
Ret fun();
```

The type `Ret` must be such that an object of type `ForwardIt` can be dereferenced and assigned a value of type `Ret` .

Type requirements

- `ForwardIt` must meet the requirements of `ForwardIterator`.

Return value

(none)

Complexity

Exactly `std::distance(first, last)` invocations of `g()` and assignments.

Possible implementation

```
template<class ForwardIt, class Generator>
void generate(ForwardIt first, ForwardIt last, Generator g)
{
    while (first != last) {
        *first++ = g();
    }
}
```

Example

The following code fills a vector with random numbers:

Run this code

```
#include <algorithm>
#include <iostream>
#include <cstdlib>

int main()
```

```
{
    std::vector<int> v(5);
    std::generate(v.begin(), v.end(), std::rand); // Using the C function rand()

    std::cout << "v: ";
    for (auto iv: v) {
        std::cout << iv << " ";
    }
    std::cout << "\n";

    // Initialize with default values 0,1,2,3,4 from a lambda function
    // Equivalent to std::iota(v.begin(), v.end(), 0);
    int n(0);
    std::generate(v.begin(), v.end(), [&n]{ return n++; });

    std::cout << "v: ";
    for (auto iv: v) {
        std::cout << iv << " ";
    }
    std::cout << "\n";
}
```

Possible output:

```
v: 52894 15984720 41513563 41346135 51451456
v: 0 1 2 3 4
```

See also

fill	assigns a range of elements a certain value (function template)
generate_n	saves the result of N applications of a function (function template)
std::experimental::parallel::generate (parallelism TS)	parallelized version of `std::generate` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/generate&oldid=80294"

std::generate_n

Defined in header `<algorithm>`

```
template< class OutputIt, class Size, class Generator >
void generate_n( OutputIt first, Size count, Generator g );
```

(until C++11)

```
template< class OutputIt, class Size, class Generator >
OutputIt generate_n( OutputIt first, Size count, Generator g );
```

(since C++11)

Assigns values, generated by given function object `g`, to the first `count` elements in the range beginning at `first`, if `count>0`. Does nothing otherwise.

Parameters

first - the beginning of the range of elements to generate

count - number of the elements to generate

g - generator function object that will be called.

The signature of the function should be equivalent to the following:

```
Ret fun();
```

The type `Ret` must be such that an object of type `OutputIt` can be dereferenced and assigned a value of type `Ret`.

Type requirements

- `OutputIt` must meet the requirements of `OutputIterator`.

Return value

(none)	(until C++11)
Iterator one past the last element assigned if `count>0`, `first` otherwise.	(since C++11)

Complexity

Exactly `count` invocations of `g()` and assignments, for `count>0`.

Possible implementation

```
template< class OutputIt, class Size, class Generator >
OutputIt generate_n( OutputIt first, Size count, Generator g )
{
    for( Size i = 0; i < count; i++ ) {
        *first++ = g();
    }
    return first;
}
```

Example

The following code fills an array of integers with random numbers.

Run this code

```
#include <cstddef>
#include <cstdlib>
#include <iostream>
#include <iterator>
#include <algorithm>

int main()
{
    const std::size_t N = 5;
    int ar[N];
    std::generate_n(ar, N, std::rand); // Using the C function rand()

    std::cout << "ar: ";
    std::copy(ar, ar+N, std::ostream_iterator<int>(std::cout, " "));
    std::cout << "\n";
}
```

Possible output:

```
ar: 52894 15984720 41513563 41346135 51451456
```

See also

`fill_n`	assigns a value to a number of elements (function template)
`generate`	saves the result of a function in a range (function template)
`std::experimental::parallel::generate_n` (parallelism TS)	parallelized version of `std::generate_n` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/generate_n&oldid=79880"

std::remove, std::remove_if

Defined in header `<algorithm>`

```
template< class ForwardIt, class T >
ForwardIt remove( ForwardIt first, ForwardIt last, const T& value );
```
(1)

```
template< class ForwardIt, class UnaryPredicate >
ForwardIt remove_if( ForwardIt first, ForwardIt last, UnaryPredicate p );
```
(2)

Removes all elements satisfying specific criteria from the range `[first, last)` and returns a past-the-end iterator for the new end of the range.

The first version removes all elements that are equal to `value`, the second version removes all elements for which predicate `p` returns `true`.

Removing is done by shifting (by means of move assignment) the elements in the range in such a way that the elements that are not to be removed appear in the beginning of the range. Relative order of the elements that remain is preserved and the *physical* size of the container is unchanged. Iterators pointing to an element between the new *logical* end and the *physical* end of the range are still dereferenceable, but the elements themselves have unspecified values (as per `MoveAssignable` post-condition). A call to `remove` is typically followed by a call to a container's `erase` method, which erases the unspecified values and reduces the *physical* size of the container to match its new *logical* size.

Parameters

`first, last` - the range of elements to process

`value` - the value of elements to remove

`p` - unary predicate which returns `true` if the element should be removed.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type &a);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it.
The type `Type` must be such that an object of type `ForwardIt` can be dereferenced and then implicitly converted to `Type`.

Type requirements

- `ForwardIt` must meet the requirements of `ForwardIterator`.
- The type of dereferenced `ForwardIt` must meet the requirements of `MoveAssignable`.
- `UnaryPredicate` must meet the requirements of `Predicate`.

Return value

Iterator to the new end of the range

Complexity

Exactly `std::distance(first, last)` applications of the predicate.

Notes

The similarly-named container member functions `list::remove`, `list::remove_if`, `forward_list::remove`, and `forward_list::remove_if` erase the removed elements.

These algorithms cannot be used with associative containers such as `std::set` and `std::map` because ForwardIt does not dereference to a MoveAssignable type (the keys in these containers are not modifiable)

Possible implementation

First version

```
template< class ForwardIt, class T >
ForwardIt remove(ForwardIt first, ForwardIt last, const T& value)
{
    first = std::find(first, last, value);
    if (first != last)
        for(ForwardIt i = first; ++i != last; )
            if (!(*i == value))
                *first++ = std::move(*i);
    return first;
}
```

Second version

```
template<class ForwardIt, class UnaryPredicate>
ForwardIt remove_if(ForwardIt first, ForwardIt last, UnaryPredicate p)
{
    first = std::find_if(first, last, p);
    if (first != last)
        for(ForwardIt i = first; ++i != last; )
            if (!p(*i))
                *first++ = std::move(*i);
    return first;
}
```

Examples

The following code removes all spaces from a string by shifting all non-space characters to the left and then erasing the extra. This is an example of erase-remove idiom .

Run this code

```
#include <algorithm>
#include <string>
#include <iostream>
#include <cctype>

int main()
{
    std::string str1 = "Text with some   spaces";
    str1.erase(std::remove(str1.begin(), str1.end(), ' '),
               str1.end());
    std::cout << str1 << '\n';

    std::string str2 = "Text\n with\tsome \t  whitespaces\n\n";
    str2.erase(std::remove_if(str2.begin(),
                              str2.end(),
                              [](char x){return std::isspace(x);}),
               str2.end());
    std::cout << str2 << '\n';
}
```

Output:

```
Textwithsomespaces
Textwithsomewhitespaces
```

See also

`remove_copy` **`remove_copy_if`**	copies a range of elements omitting those that satisfy specific criteria (function template)
`unique`	removes consecutive duplicate elements in a range (function template)
`std::experimental::parallel::remove` (parallelism TS)	parallelized version of `std::remove` (function template)
`std::experimental::parallel::remove_if` (parallelism TS)	parallelized version of `std::remove_if` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/remove&oldid=79904"

std::remove_copy, std::remove_copy_if

Defined in header `<algorithm>`

```
template< class InputIt, class OutputIt, class T >
OutputIt remove_copy( InputIt first, InputIt last, OutputIt d_first,
                      const T& value );
```
(1)

```
template< class InputIt, class OutputIt, class UnaryPredicate >
OutputIt remove_copy_if( InputIt first, InputIt last, OutputIt d_first,
                         UnaryPredicate p );
```
(2)

Copies elements from the range `[first, last)`, to another range beginning at `d_first`, omitting the elements which satisfy specific criteria. The first version ignores the elements that are equal to `value`, the second version ignores the elements for which predicate `p` returns `true`. Source and destination ranges cannot overlap.

Parameters

first, last	-	the range of elements to copy
d_first	-	the beginning of the destination range.
value	-	the value of the elements not to copy

Type requirements

- `InputIt` must meet the requirements of `InputIterator`.
- `OutputIt` must meet the requirements of `OutputIterator`.
- `UnaryPredicate` must meet the requirements of `Predicate`.

Return value

Iterator to the element past the last element copied.

Complexity

Exactly `last - first` applications of the predicate.

Possible implementation

First version

```
template<class InputIt, class OutputIt, class T>
OutputIt remove_copy(InputIt first, InputIt last,
                     OutputIt d_first, const T& value)
{
    for (; first != last; ++first) {
        if (!(*first == value)) {
            *d_first++ = *first;
        }
    }
    return d_first;
}
```

Second version

```
template<class InputIt, class OutputIt, class UnaryPredicate>
OutputIt remove_copy_if(InputIt first, InputIt last,
                        OutputIt d_first, UnaryPredicate p)
{
    for (; first != last; ++first) {
        if (!p(*first)) {
            *d_first++ = *first;
        }
    }
    return d_first;
}
```

Example

The following code outputs a string while erasing the spaces on the fly.

Run this code

```
#include <algorithm>
#include <iterator>
#include <string>
#include <iostream>
int main()
{
    std::string str = "Text with some   spaces";
    std::cout << "before: " << str << "\n";

    std::cout << "after:  ";
    std::remove_copy(str.begin(), str.end(),
                     std::ostream_iterator<char>(std::cout), ' ');
    std::cout << '\n';
}
```

Output:

```
before: Text with some   spaces
after:  Textwithsomespaces
```

See also

remove **remove_if**	removes elements satisfying specific criteria (function template)
copy **copy_if** (C++11)	copies a range of elements to a new location (function template)
std::experimental::parallel::remove_copy (parallelism TS)	parallelized version of `std::remove_copy` (function template)
std::experimental::parallel::remove_copy_if (parallelism TS)	parallelized version of `std::remove_copy_if` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/remove_copy&oldid=79907"

std::replace, std::replace_if

Defined in header <algorithm>

```
template< class ForwardIt, class T >
void replace( ForwardIt first, ForwardIt last,                          (1)
              const T& old_value, const T& new_value );
```

```
template< class ForwardIt, class UnaryPredicate, class T >
void replace_if( ForwardIt first, ForwardIt last,                       (2)
                 UnaryPredicate p, const T& new_value );
```

Replaces all elements satisfying specific criteria with `new_value` in the range `[first, last)`. The first version replaces the elements that are equal to `old_value`, the second version replaces elements for which predicate `p` returns `true` .

Parameters

first, last - the range of elements to process

old_value - the value of elements to replace

p - unary predicate which returns `true` if the element value should be replaced.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type &a);
```

The signature does not need to have `const &` , but the function must not modify the objects passed to it.
The type `Type` must be such that an object of type `ForwardIt` can be dereferenced and then implicitly converted to `Type` .

new_value - the value to use as replacement

Type requirements

- `ForwardIt` must meet the requirements of `ForwardIterator`.
- `UnaryPredicate` must meet the requirements of `Predicate`.

Return value

(none)

Complexity

Exactly `last - first` applications of the predicate.

Possible implementation

First version

```
template<class ForwardIt, class T>
void replace(ForwardIt first, ForwardIt last,
             const T& old_value, const T& new_value)
{
    for (; first != last; ++first) {
        if (*first == old_value) {
            *first = new_value;
        }
    }
}
```

Second version

```
template<class ForwardIt, class UnaryPredicate, class T>
void replace_if(ForwardIt first, ForwardIt last,
                UnaryPredicate p, const T& new_value)
{
    for (; first != last; ++first) {
        if(p(*first)) {
            *first = new_value;
        }
    }
}
```

Example

The following code at first replaces all occurrences of `8` with `88` in a vector of integers. Then it replaces all values less than `5` with 55.

Run this code

```
#include <algorithm>
#include <array>
#include <iostream>
#include <functional>

int main()
{
    std::array<int, 10> s{5, 7, 4, 2, 8, 6, 1, 9, 0, 3};

    std::replace(s.begin(), s.end(), 8, 88);

    for (int a : s) {
        std::cout << a << " ";
    }
    std::cout << '\n';

    std::replace_if(s.begin(), s.end(),
                    std::bind(std::less<int>(), std::placeholders::_1, 5), 55);
    for (int a : s) {
        std::cout << a << " ";
    }
    std::cout << '\n';
}
```

Output:

```
5 7 4 2 88 6 1 9 0 3
5 7 55 55 88 6 55 9 55 55
```

See also

replace_copy **replace_copy_if**	copies a range, replacing elements satisfying specific criteria with another value (function template)
std::experimental::parallel::replace (parallelism TS)	parallelized version of `std::replace` (function template)
std::experimental::parallel::replace_if (parallelism TS)	parallelized version of `std::replace_if`

(function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/replace&oldid=79906"

std::replace_copy, std::replace_copy_if

Defined in header `<algorithm>`

```
template< class InputIt, class OutputIt, class T >
OutputIt replace_copy( InputIt first, InputIt last, OutputIt d_first,       (1)
                       const T& old_value, const T& new_value );
```

```
template< class InputIt, class OutputIt, class UnaryPredicate, class T >
OutputIt replace_copy_if( InputIt first, InputIt last, OutputIt d_first,    (2)
                          UnaryPredicate p, const T& new_value );
```

Copies the all elements from the range `[first, last)` to another range beginning at `d_first` replacing all elements satisfying specific criteria with `new_value`. The first version replaces the elements that are equal to `old_value`, the second version replaces elements for which predicate `p` returns `true` . The source and destination ranges cannot overlap.

Parameters

first, last - the range of elements to copy

d_first - the beginning of the destination range

old_value - the value of elements to replace

p - unary predicate which returns `true` if the element value should be replaced.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type &a);
```

The signature does not need to have `const &` , but the function must not modify the objects passed to it.
The type `Type` must be such that an object of type `InputIt` can be dereferenced and then implicitly converted to `Type` .

new_value - the value to use as replacement

Type requirements

- `InputIt` must meet the requirements of `InputIterator`.
- `OutputIt` must meet the requirements of `OutputIterator`.

Return value

Iterator to the element past the last element copied.

Complexity

Exactly `last - first` applications of the predicate.

Possible implementation

First version

```
template<class InputIt, class OutputIt, class T>
OutputIt replace_copy(InputIt first, InputIt last, OutputIt d_first,
                      const T& old_value, const T& new_value)
{
    for (; first != last; ++first) {
        *d_first++ = (*first == old_value) ? new_value : *first;
    }
    return d_first;
}
```

Second version

```
template<class InputIt, class OutputIt,
         class UnaryPredicate, class T>
OutputIt replace_copy_if(InputIt first, InputIt last, OutputIt d_first,
                         UnaryPredicate p, const T& new_value)
{
    for (; first != last; ++first) {
        *d_first++ = p( *first ) ? new_value : *first;
    }
    return d_first;
}
```

Example

The following copy prints a vector, replacing all values over 5 with 99 on the fly.

Run this code

```
#include <algorithm>
#include <vector>
#include <iostream>
#include <iterator>
#include <functional>

int main()
{
    std::vector<int> v{5, 7, 4, 2, 8, 6, 1, 9, 0, 3};
    std::replace_copy_if(v.begin(), v.end(),
                         std::ostream_iterator<int>(std::cout, " "),
                         [](int n){return n > 5;}, 99);
    std::cout << '\n';
}
```

Output:

```
5 99 4 2 99 99 1 99 0 3
```

See also

remove **remove_if**	removes elements satisfying specific criteria (function template)
std::experimental::parallel::replace_copy (parallelism TS)	parallelized version of `std::replace_copy` (function template)
std::experimental::parallel::replace_copy_if (parallelism TS)	parallelized version of `std::replace_copy_if` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/replace_copy&oldid=79905"

std::swap

Defined in header `<algorithm>` (until C++11)
Defined in header `<utility>` (since C++11)

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

(1)

```
template< class T2, size_t N >
void swap( T2 (&a)[N], T2 (&b)[N]);
```

(2) (since C++11)

Exchanges the given values.

1) Swaps the values `a` and `b`.

2) Swaps the arrays `a` and `b`. In effect calls `std::swap_ranges(a, a+N, b)`.

Parameters

a, b - the values to be swapped

Type requirements

- `T` must meet the requirements of `MoveAssignable` and `MoveConstructible`.
- `T2` must meet the requirements of `Swappable`.

Return value

(none)

Exceptions

1)

(none) (until C++11)

`noexcept` specification: (since C++11)

```
noexcept(

    std::is_nothrow_move_constructible<T>::value &&
    std::is_nothrow_move_assignable<T>::value

)
```

2) `noexcept` specification:

```
noexcept(noexcept(swap(*a, *b)))
```

Complexity

1) Constant

2) Linear in N

Specializations

`std::swap` may be specialized in namespace std for user-defined types, but such specializations are not found by ADL (the namespace std is not the associated namespace for the user-defined type). The expected way to make a user-defined type swappable is to provide a non-member function swap in the same namespace as the type: see `Swappable` for details.

The following overloads are already provided by the standard library:

std::swap(std::pair) (C++11)	specializes the **std::swap** algorithm (function template)
std::swap(std::tuple) (C++11)	specializes the **std::swap** algorithm (function template)
std::swap(std::shared_ptr) (C++11)	specializes the **std::swap** algorithm (function template)
std::swap(std::weak_ptr) (C++11)	specializes the **std::swap** algorithm (function template)
std::swap(std::unique_ptr) (C++11)	specializes the **std::swap** algorithm (function template)
std::swap(std::function) (C++11)	specializes the **std::swap** algorithm (function template)
std::swap(std::basic_string)	specializes the **std::swap** algorithm (function template)
std::swap(std::array) (C++11)	specializes the **std::swap** algorithm (function template)
std::swap(std::deque)	specializes the **std::swap** algorithm (function template)
std::swap(std::forward_list) (C++11)	specializes the **std::swap** algorithm (function template)
std::swap(std::list)	specializes the **std::swap** algorithm (function template)
std::swap(std::vector)	specializes the **std::swap** algorithm (function template)
std::swap(std::map)	specializes the **std::swap** algorithm (function template)
std::swap(std::multimap)	specializes the **std::swap** algorithm (function template)
std::swap(std::set)	specializes the **std::swap** algorithm (function template)
std::swap(std::multiset)	specializes the **std::swap** algorithm (function template)
std::swap(std::unordered_map) (C++11)	specializes the **std::swap** algorithm (function template)
std::swap(std::unordered_multimap) (C++11)	specializes the **std::swap** algorithm (function template)
std::swap(std::unordered_set) (C++11)	specializes the **std::swap** algorithm (function template)
std::swap(std::unordered_multiset) (C++11)	specializes the **std::swap** algorithm (function template)
std::swap(std::queue)	specializes the **std::swap** algorithm (function template)
std::swap(std::priority_queue)	specializes the **std::swap** algorithm (function template)
std::swap(std::stack)	specializes the **std::swap** algorithm (function template)
std::swap(std::valarray) (C++11)	specializes the **std::swap()** algorithm (function template)
std::swap(std::basic_stringbuf) (C++11)	specializes the **std::swap** algorithm (function template)
std::swap(std::basic_istringstream) (C++11)	specializes the **std::swap** algorithm (function template)
std::swap(std::basic_ostringstream) (C++11)	specializes the **std::swap** algorithm (function template)
std::swap(std::basic_stringstream) (C++11)	specializes the **std::swap** algorithm (function template)
std::swap(std::basic_filebuf) (C++11)	specializes the **std::swap** algorithm (function template)
std::swap(std::basic_ifstream) (C++11)	specializes the **std::swap** algorithm (function template)
std::swap(std::basic_ofstream) (C++11)	specializes the **std::swap** algorithm (function template)

std::swap(std::basic_fstream) (C++11)	specializes the **std::swap** algorithm (function template)
std::swap(std::basic_regex) (C++11)	specializes the **std::swap** algorithm (function template)
std::swap(std::match_results) (C++11)	specializes the **std::swap()** algorithm (function template)
std::swap(std::thread) (C++11)	specializes the **std::swap** algorithm (function template)
std::swap(std::unique_lock) (C++11)	specialization of **std::swap** for unique_lock (function template)
std::swap(std::promise) (C++11)	specializes the **std::swap** algorithm (function template)
std::swap(std::packaged_task) (C++11)	specializes the **std::swap** algorithm (function template)

Example

Run this code

```
#include <algorithm>
#include <iostream>

int main()
{
   int a = 5, b = 3;

   // before
   std::cout << a << ' ' << b << '\n';

   std::swap(a,b);

   // after
   std::cout << a << ' ' << b << '\n';
}
```

Output:

```
5 3
3 5
```

See also

iter_swap	swaps the elements pointed to by two iterators (function template)
swap_ranges	swaps two ranges of elements (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/swap&oldid=72222"

std::swap_ranges

Defined in header `<algorithm>`

```
template< class ForwardIt1, class ForwardIt2 >
ForwardIt2 swap_ranges( ForwardIt1 first1, ForwardIt1 last1, ForwardIt2 first2 );
```

Exchanges elements between range `[first1, last1)` and another range starting at `first2`.

Parameters

`first1, last1` - the first range of elements to swap

`first2` - beginning of the second range of elements to swap

Type requirements

- `ForwardIt1, ForwardIt2` must meet the requirements of `ForwardIterator`.
- The types of dereferenced `ForwardIt1` and `ForwardIt2` must meet the requirements of `Swappable`

Return value

Iterator to the element past the last element exchanged in the range beginning with `first2`.

Possible implementation

```
template<class ForwardIt1, class ForwardIt2>
ForwardIt2 swap_ranges(ForwardIt1 first1,
                             ForwardIt1 last1,
                             ForwardIt2 first2)
{
    while (first1 != last1) {
        std::iter_swap(first1++, first2++);
    }
    return first2;
}
```

Example

Demonstrates swapping of subranges from different containers

Run this code

```
#include <algorithm>
#include <list>
#include <vector>
#include <iostream>
int main()
{
    std::vector<int> v = {1, 2, 3, 4, 5};
    std::list<int> l = {-1, -2, -3, -4, -5};

    std::swap_ranges(v.begin(), v.begin()+3, l.begin());

    for(int n : v)
       std::cout << n << ' ';
    std::cout << '\n';
    for(int n : l)
       std::cout << n << ' ';
    std::cout << '\n';
}
```

Output:

```
-1 -2 -3 4 5
1 2 3 -4 -5
```

Complexity

linear in the distance between `first` and `last`

See also

iter_swap	swaps the elements pointed to by two iterators (function template)
swap	swaps the values of two objects (function template)
std::experimental::parallel::swap_ranges (parallelism TS)	parallelized version of `std::swap_ranges` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/swap_ranges&oldid=79921"

std::iter_swap

Defined in header `<algorithm>`

```
template< class ForwardIt1, class ForwardIt2 >
void iter_swap( ForwardIt1 a, ForwardIt2 b );
```

Swaps the values of the elements the given iterators are pointing to.

Parameters

a, b - iterators to the elements to swap

Type requirements

- `ForwardIt1, ForwardIt2` must meet the requirements of `ForwardIterator`.
- `*a, *b` must meet the requirements of `Swappable`.

Return value

(none)

Complexity

constant

Possible implementation

```
template<class ForwardIt1, class ForwardIt2>
void iter_swap(ForwardIt1 a, ForwardIt2 b)
{
   using std::swap;
   swap(*a, *b);
}
```

Example

The following is an implementation of selection sort in C++

Run this code

```
#include <random>
#include <vector>
#include <iostream>
#include <algorithm>
#include <functional>
#include <iterator>

template<class ForwardIt>
void selection_sort(ForwardIt begin, ForwardIt end)
{
    for (ForwardIt i = begin; i != end; ++i)
        std::iter_swap(i, std::min_element(i, end));
}

int main()
{
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dist(-10, 10);
    std::vector<int> v;
    generate_n(back_inserter(v), 20, bind(dist, gen));

    std::cout << "Before sort: ";
    copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "));

    selection_sort(v.begin(), v.end());

    std::cout << "\nAfter sort: ";
    copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "));
    std::cout << '\n';
}
```

Output:

```
Before sort: -7 6 2 4 -1 6 -9 -1 2 -5 10 -9 -5 -3 -5 -3 6 6 1 8
After sort: -9 -9 -7 -5 -5 -5 -3 -3 -1 -1 1 2 2 4 6 6 6 6 8 10
```

See also

swap	swaps the values of two objects (function template)
swap_ranges	swaps two ranges of elements (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/iter_swap&oldid=56985"

std::reverse

Defined in header `<algorithm>`

```
template< class BidirIt >
void reverse( BidirIt first, BidirIt last );
```

Reverses the order of the elements in the range `[first, last)`

Behaves as if applying `std::iter_swap` to every pair of iterators `first+i, (last-i) - 1` for each non-negative `i < (last-first)/2`

Parameters

`first, last` - the range of elements to reverse

Type requirements

- `BidirIt` must meet the requirements of `BidirectionalIterator`.
- The type of dereferenced `BidirIt` must meet the requirements of `Swappable`.

Return value

(none)

Possible implementation

```
template<class BidirIt>
void reverse(BidirIt first, BidirIt last)
{
    while ((first != last) && (first != --last)) {
        std::iter_swap(first++, last);
    }
}
```

Example

Run this code

```
#include <vector>
#include <iostream>
#include <iterator>
#include <algorithm>

int main()
{
    std::vector<int> v({1,2,3});
    std::reverse(std::begin(v), std::end(v));
    std::cout << v[0] << v[1] << v[2] << '\n';

    int a[] = {4, 5, 6, 7};
    std::reverse(std::begin(a), std::end(a));
    std::cout << a[0] << a[1] << a[2] << a[3] << '\n';
}
```

Output:

```
321
7654
```

Complexity

linear in the distance between `first` and `last`

See also

`reverse_copy`	creates a copy of a range that is reversed (function template)
`std::experimental::parallel::reverse` (parallelism TS)	parallelized version of `std::reverse` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/reverse&oldid=79908"

std::reverse_copy

Defined in header `<algorithm>`

```
template< class BidirIt, class OutputIt >
OutputIt reverse_copy( BidirIt first, BidirIt last, OutputIt d_first );
```

Copies the elements from the range `[first, last)` to another range beginning at `d_first` in such a way that the elements in the new range are in reverse order.

Behaves as if by executing the assignment `*(d_first + (last - first) - 1 - i) = *(first + i)` once for each non-negative `i < (last - first)`

If the source and destination ranges (that is, `[first, last)` and `[d_first, d_first+(last-first))` respectively) overlap, the behavior is undefined.

Parameters

first, last - the range of elements to copy

d_first - the beginning of the destination range

Type requirements

- `BidirIt` must meet the requirements of `BidirectionalIterator`.
- `OutputIt` must meet the requirements of `OutputIterator`.

Return value

Output iterator to the element past the last element copied.

Possible implementation

```
template<class BidirIt, class OutputIt>
OutputIt reverse_copy(BidirIt first, BidirIt last, OutputIt d_first)
{
    while (first != last) {
        *(d_first++) = *(--last);
    }
    return d_first;
}
```

Example

Run this code

```
#include <vector>
#include <iostream>
#include <algorithm>

int main()
{
    std::vector<int> v({1,2,3});
    for (const auto& value : v) {
        std::cout << value << " ";
    }
    std::cout << '\n';
    std::vector<int> destiny(3);
    std::reverse_copy(std::begin(v), std::end(v), std::begin(destiny));
    for (const auto& value : destiny) {
        std::cout << value << " ";
    }
    std::cout << '\n';
}
```

Output:

```
1 2 3
3 2 1
```

Complexity

Linear in the distance between `first` and `last`

See also

`reverse`	reverses the order of elements in a range (function template)
`std::experimental::parallel::reverse_copy` (parallelism TS)	parallelized version of `std::reverse_copy` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/reverse_copy&oldid=79909"

std::rotate

Defined in header `<algorithm>`

`template< class ForwardIt >`
`void rotate( ForwardIt first, ForwardIt n_first, ForwardIt last );`
	(until C++11)

`template< class ForwardIt >`
`ForwardIt rotate( ForwardIt first, ForwardIt n_first, ForwardIt last );`
	(since C++11)

Performs a left rotation on a range of elements.

Specifically, `std::rotate` swaps the elements in the range `[first, last)` in such a way that the element `n_first` becomes the first element of the new range and `n_first - 1` becomes the last element.

A precondition of this function is that `[first, n_first)` and `[n_first, last)` are valid ranges.

Parameters

first	-	the beginning of the original range
n_first	-	the element that should appear at the beginning of the rotated range
last	-	the end of the original range

Type requirements

- `ForwardIt` must meet the requirements of `ValueSwappable` and `ForwardIterator`.
- The type of dereferenced `ForwardIt` must meet the requirements of `MoveAssignable` and `MoveConstructible`.

Return value

(none)	(until C++11)
The iterator equal to `first + (last - n_first)`	(since C++11)

Complexity

Linear in the distance between `first` and `last`

Possible implementation

```
template <class ForwardIt>
ForwardIt rotate(ForwardIt first, ForwardIt n_first, ForwardIt last)
{
    ForwardIt next = n_first;
    while (first != next) {
        std::iter_swap(first++, next++);
        if (next == last) {
            next = n_first;
        } else if (first == n_first) {
            n_first = next;
        }
    }
    return next;
}
```

Example

`std::rotate` is a common building block in many algorithms. This example demonstrates insertion sort:

Run this code

```
#include <vector>
#include <iostream>
#include <algorithm>

int main()
{
    std::vector<int> v{2, 4, 2, 0, 5, 10, 7, 3, 7, 1};

    std::cout << "before sort:      ";
    for (int n: v)
        std::cout << n << ' ';
    std::cout << '\n';

    // insertion sort
    for (auto i = v.begin(); i != v.end(); ++i) {
        std::rotate(std::upper_bound(v.begin(), i, *i), i, i+1);
    }

    std::cout << "after sort:       ";
    for (int n: v)
        std::cout << n << ' ';
    std::cout << '\n';

    // simple rotation to the left
    std::rotate(v.begin(), v.begin() + 1, v.end());

    std::cout << "simple rotate left  : ";
    for (int n: v)
        std::cout << n << ' ';
    std::cout << '\n';

    // simple rotation to the right
    std::rotate(v.rbegin(), v.rbegin() + 1, v.rend());

    std::cout << "simple rotate right : ";
    for (int n: v)
        std::cout << n << ' ';
    std::cout << '\n';

}
```

Output:

```
before sort:      2 4 2 0 5 10 7 3 7 1
after sort:       0 1 2 2 3 4 5 7 7 10
simple rotate left : 1 2 2 3 4 5 7 7 10 0
simple rotate right: 0 1 2 2 3 4 5 7 7 10
```

See also

rotate_copy	copies and rotate a range of elements (function template)
std::experimental::parallel::rotate (parallelism TS)	parallelized version of `std::rotate` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/rotate&oldid=79910"

std::rotate_copy

Defined in header `<algorithm>`

```
template< class ForwardIt, class OutputIt >
OutputIt rotate_copy( ForwardIt first, ForwardIt n_first,
                      ForwardIt last, OutputIt d_first );
```

Copies the elements from the range `[first, last)`, to another range beginning at `d_first` in such a way, that the element `n_first` becomes the first element of the new range and `n_first - 1` becomes the last element.

Parameters

first, last - the range of elements to copy

n_first - an iterator to an element in `[first, last)` that should appear at the beginning of the new range

d_first - beginning of the destination range

Type requirements

- `ForwardIt` must meet the requirements of `ForwardIterator`.
- `OutputIt` must meet the requirements of `OutputIterator`.

Return value

Output iterator to the element past the last element copied.

Possible implementation

```
template<class ForwardIt, class OutputIt>
OutputIt rotate_copy(ForwardIt first, ForwardIt n_first,
                           ForwardIt last, OutputIt d_first)
{
    d_first = std::copy(n_first, last, d_first);
    return std::copy(first, n_first, d_first);
}
```

Example

Run this code

```
#include <algorithm>
#include <vector>
#include <iostream>

int main()
{
    std::vector<int> src = {1, 2, 3, 4, 5};
    auto pivot = std::find(src.begin(), src.end(), 3);
    std::vector<int> dest(src.size());

    std::rotate_copy(src.begin(), pivot, src.end(), dest.begin());

    for (const auto &i : dest) {
        std::cout << i << ' ';
    }
    std::cout << '\n';
}
```

Output:

```
3 4 5 1 2
```

Complexity

linear in the distance between `first` and `last`

See also

`rotate`	rotates the order of elements in a range (function template)
`std::experimental::parallel::rotate_copy` (parallelism TS)	parallelized version of `std::rotate_copy` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/rotate_copy&oldid=79911"

std::random_shuffle, std::shuffle

Defined in header `<algorithm>`

`template< class RandomIt >`
`void random_shuffle( RandomIt first, RandomIt last );`
	(1)	(until C++17)
		(deprecated in C++14)

`template< class RandomIt, class RandomFunc >`
`void random_shuffle( RandomIt first, RandomIt last, RandomFunc& r );`
	(2)	(until C++11)

`template< class RandomIt, class RandomFunc >`
`void random_shuffle( RandomIt first, RandomIt last, RandomFunc&& r );`
		(since C++11)
		(until C++17)
		(deprecated in C++14)

`template< class RandomIt, class URNG >`
`void shuffle( RandomIt first, RandomIt last, URNG&& g );`
	(3)	(since C++11)

Reorders the elements in the given range `[first, last)` such that each possible permutation of those elements has equal probability of appearance.

1) The random number generator is implementation-defined, but the function `std::rand` is often used.

2) The random number generator is the function object `r`.

3) The random number generator is the function object `g`.

Parameters

first, last	-	the range of elements to shuffle randomly
r	-	function object returning a randomly chosen value of type convertible to `std::iterator_traits<RandomIt>::difference_type` in the interval [0,n) if invoked as `r(n)`
g	-	a `UniformRandomNumberGenerator` whose result type is convertible to `std::iterator_traits<RandomIt>::difference_type`

Type requirements

- `RandomIt` must meet the requirements of `ValueSwappable` and `RandomAccessIterator`.
- `URNG` must meet the requirements of `UniformRandomNumberGenerator`.

Return value

(none)

Complexity

Linear in the distance between `first` and `last`

Possible implementation

First version

```
template< class RandomIt >
void random_shuffle( RandomIt first, RandomIt last )
{
    typename std::iterator_traits<RandomIt>::difference_type i, n;
    n = last - first;
    for (i = n-1; i > 0; --i) {
        using std::swap;
        swap(first[i], first[std::rand() % (i+1)]);
    }
}
```

Second version

```
template<class RandomIt, class RandomFunc>
void random_shuffle(RandomIt first, RandomIt last, RandomFunc&& r)
{
    typename std::iterator_traits<RandomIt>::difference_type i, n;
    n = last - first;
    for (i = n-1; i > 0; --i) {
        using std::swap;
        swap(first[i], first[r(i+1)]);
    }
}
```

Third version

```
template<class RandomIt, class UniformRandomNumberGenerator>
void shuffle(RandomIt first, RandomIt last,
             UniformRandomNumberGenerator&& g)
{
    typedef typename std::iterator_traits<RandomIt>::difference_type diff_t;
    typedef typename std::make_unsigned<diff_t>::type udiff_t;
    typedef typename std::uniform_int_distribution<udiff_t> distr_t;
    typedef typename distr_t::param_type param_t;

    distr_t D;
    diff_t n = last - first;
    for (diff_t i = n-1; i > 0; --i) {
        using std::swap;
        swap(first[i], first[D(g, param_t(0, i))]);
    }
}
```

Example

The following code randomly shuffles the integers 1..10:

Run this code

```
#include <random>
#include <algorithm>
#include <iterator>
#include <iostream>

int main()
{
    std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    std::random_device rd;
    std::mt19937 g(rd());

    std::shuffle(v.begin(), v.end(), g);

    std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "));
    std::cout << "\n";
}
```

Possible output:

```
8 6 10 4 2 3 7 1 9 5
```

See also

`next_permutation`	generates the next greater lexicographic permutation of a range of elements (function template)
`prev_permutation`	generates the next smaller lexicographic permutation of a range of elements (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/random_shuffle&oldid=73955"

std::unique

Defined in header `<algorithm>`

```
template< class ForwardIt >
ForwardIt unique( ForwardIt first, ForwardIt last );
```
(1)

```
template< class ForwardIt, class BinaryPredicate >
ForwardIt unique( ForwardIt first, ForwardIt last, BinaryPredicate p );
```
(2)

Removes all consecutive duplicate elements from the range `[first, last)` and returns a past-the-end iterator for the new *logical* end of the range. The first version uses `operator==` to compare the elements, the second version uses the given binary predicate `p`.

Removing is done by shifting the elements in the range in such a way that elements to be erased are overwritten. Relative order of the elements that remain is preserved and the *physical* size of the container is unchanged. Iterators pointing to an element between the new *logical* end and the *physical* end of the range are still dereferenceable, but the elements themselves have unspecified values. A call to `unique` is typically followed by a call to a container's `erase` method, which erases the unspecified values and reduces the *physical* size of the container to match its new *logical* size.

Parameters

first, last - the range of elements to process

p - binary predicate which returns `true` if the elements should be treated as equal.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `ForwardIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `ForwardIt` must meet the requirements of `ForwardIterator`.
- The type of dereferenced `ForwardIt` must meet the requirements of `MoveAssignable`.

Return value

Forward iterator to the new end of the range

Complexity

For nonempty ranges, exactly `std::distance(first,last) -1` applications of the corresponding predicate.

Possible implementation

First version

```
template<class ForwardIt>
ForwardIt unique(ForwardIt first, ForwardIt last)
{
    if (first == last)
        return last;

    ForwardIt result = first;
    while (++first != last) {
        if (!(*result == *first) && ++result != first) {
            *result = std::move(*first);
        }
    }
    return ++result;
}
```

Second version

```
template<class ForwardIt, class BinaryPredicate>
ForwardIt unique(ForwardIt first, ForwardIt last, BinaryPredicate p)
{
    if (first == last)
        return last;

    ForwardIt result = first;
    while (++first != last) {
        if (!p(*result, *first) && ++result != first) {
            *result = std::move(*first);
        }
    }
    return ++result;
}
```

Examples

The following code removes all consecutive equivalent elements from a vector of integers.

Run this code

```
#include <iostream>
#include <algorithm>
#include <vector>

int main()
{
    std::vector<int> v{1, 2, 2, 2, 3, 3, 2, 2, 1};
    std::vector<int>::iterator last;

    last = std::unique(v.begin(), v.end()); // 1 2 3 2 1 3 2 2 1
                                            //           ^
    for (std::vector<int>::iterator it = v.begin(); it != last; ++it) {
        std::cout << *it << " ";
    }
    std::cout << "\n";
}
```

Output:

```
1 2 3 2 1
```

The following code removes all duplicate elements from a vector of integers.

Run this code

```
#include <iostream>
```

```
#include <algorithm>
#include <vector>

int main()
{
    std::vector<int> v{1,2,3,1,2,3,3,4,5,4,5,6,7};
    std::sort(v.begin(), v.end());
    auto last = std::unique(v.begin(), v.end());
    v.erase(last, v.end());
    for (const auto& i : v)
      std::cout << i << " ";
    std::cout << "\n";
}
```

Output:

```
1 2 3 4 5 6 7
```

See also

adjacent_find	finds the first two adjacent items that are equal (or satisfy a given predicate) (function template)
unique_copy	creates a copy of some range of elements that contains no consecutive duplicates (function template)
remove **remove_if**	removes elements satisfying specific criteria (function template)
unique	removes consecutive duplicate elements (public member function of std::list)
std::experimental::parallel::unique (parallelism TS)	parallelized version of std::unique (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/unique&oldid=79927"

std::unique_copy

Defined in header `<algorithm>`

```
template< class InputIt, class OutputIt >
OutputIt unique_copy( InputIt first, InputIt last,
                      OutputIt d_first );
```

(1)

```
template< class InputIt, class OutputIt, class BinaryPredicate >
OutputIt unique_copy( InputIt first, InputIt last,
                      OutputIt d_first, BinaryPredicate p );
```

(2)

Copies the elements from the range `[first, last)`, to another range beginning at `d_first` in such a way that there are no consecutive equal elements. Only the first element of each group of equal elements is copied. The first version uses `operator==` to compare the elements, the second version uses the given binary predicate `p`.

Parameters

first, last - the range of elements to process

d_first - the beginning of the destination range

p - binary predicate which returns `true` if the elements should be treated as equal.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `InputIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `InputIt` must meet the requirements of `InputIterator`.
- `OutputIt` must meet the requirements of `OutputIterator`.
- The type of dereferenced `InputIt` must meet the requirements of `CopyAssignable`. if `InputIt` does not satisfy `ForwardIterator`
- The type of dereferenced `InputIt` must meet the requirements of `CopyConstructible`. if neither `InputIt` nor `OutputIt` satisfies `ForwardIterator`, or if `InputIt` does not satisfy `ForwardIterator` and the value type of `InputIt` differs from that of `OutputIt`

Return value

Output iterator to the element past the last written element

Complexity

For nonempty ranges, exactly `std::distance(first, last) - 1` applications of the corresponding comparator.

Notes

If `InputIt` satisfies `ForwardIterator`, this function rereads the input in order to detect duplicates.

Otherwise, if `OutputIt` satisfies `ForwardIterator`, and the value type of `InputIt` is the same as that of `OutputIt`, this function compare `*d_first` to `*first`.

Otherwise, this function compares `*first` to a local element copy.

Example

Run this code

```
#include <string>
#include <iostream>
#include <algorithm>
#include <iterator>

int main()
{
    std::string s1 = "The      string    with many       spaces!";
    std::cout << "before: " << s1 << '\n';

    std::string s2;
    std::unique_copy(s1.begin(), s1.end(), std::back_inserter(s2),
                     [](char c1, char c2){ return c1 == ' ' && c2 == ' '; });

    std::cout << "after:  " << s2 << '\n';
}
```

Output:

```
before: The      string    with many       spaces!
after:  The string with many spaces!
```

See also

adjacent_find	finds the first two adjacent items that are equal (or satisfy a given predicate) (function template)
unique	removes consecutive duplicate elements in a range (function template)
std::experimental::parallel::unique_copy (parallelism TS)	parallelized version of `std::unique_copy` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/unique_copy&oldid=79928"

std::is_partitioned

Defined in header `<algorithm>`

```
template< class InputIt, class UnaryPredicate >
bool is_partitioned( InputIt first, InputIt last, UnaryPredicate p );
```
(since C++11)

Returns `true` if all elements in the range `[first, last)` that satisfy the predicate `p` appear before all elements that don't. Also returns `true` if `[first, last)` is empty.

Parameters

first, last - the range of elements to check

p - unary predicate which returns `true` for the elements expected to be found in the beginning of the range.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type &a);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it.
The type `Type` must be such that an object of type `InputIt` can be dereferenced and then implicitly converted to `Type`.

Type requirements

- `InputIt` must meet the requirements of `InputIterator`.
- `UnaryPredicate` must meet the requirements of `Predicate`.

Return value

`true` if the range `[first, last)` is empty or is partitioned by `p`. `false` otherwise.

Complexity

At most `std::distance(first, last)` applications of `p`.

Possible implementation

```
template< class InputIt, class UnaryPredicate >
bool is_partitioned(InputIt first, InputIt last, UnaryPredicate p)
{
    for (; first != last; ++first)
        if (!p(*first))
            break;
    for (; first != last; ++first)
        if (p(*first))
            return false;
    return true;
}
```

Example

Run this code

```
#include <algorithm>
#include <array>
#include <iostream>

int main()
{
    std::array<int, 9> v = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    auto is_even = [](int i){ return i % 2 == 0; };
    std::cout.setf(std::ios_base::boolalpha);
    std::cout << std::is_partitioned(v.begin(), v.end(), is_even) << ' ';

    std::partition(v.begin(), v.end(), is_even);
    std::cout << std::is_partitioned(v.begin(), v.end(), is_even) << ' ';

    std::reverse(v.begin(), v.end());
    std::cout << std::is_partitioned(v.begin(), v.end(), is_even);
}
```

Output:

```
false true false
```

See also

partition	divides a range of elements into two groups (function template)
partition_point (C++11)	locates the partition point of a partitioned range (function template)
std::experimental::parallel::is_partitioned (parallelism TS)	parallelized version of `std::is_partitioned` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/is_partitioned&oldid=79887"

std::partition

Defined in header <algorithm>

`template< class BidirIt, class UnaryPredicate >`
`BidirIt partition( BidirIt first, BidirIt last, UnaryPredicate p );`
	(until C++11)

`template< class ForwardIt, class UnaryPredicate >`
`ForwardIt partition( ForwardIt first, ForwardIt last, UnaryPredicate p );`
	(since C++11)

Reorders the elements in the range `[first, last)` in such a way that all elements for which the predicate `p` returns `true` precede the elements for which predicate `p` returns `false`. Relative order of the elements is not preserved.

Parameters

first, last - the range of elements to reorder

p - unary predicate which returns `true` if the element should be ordered before other elements.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type &a);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it.
The type `Type` must be such that an object of type `ForwardIt` can be dereferenced and then implicitly converted to `Type`.

Type requirements

- `BidirIt` must meet the requirements of `BidirectionalIterator`.
- `ForwardIt` must meet the requirements of `ValueSwappable` and `ForwardIterator`. However, the operation is more efficient if `ForwardIt` also satisfies the requirements of `BidirectionalIterator`
- `UnaryPredicate` must meet the requirements of `Predicate`.

Return value

Iterator to the first element of the second group.

Complexity

Exactly `std::distance(first,last)` applications of the predicate and at most `std::distance(first,last)` swaps. If `ForwardIt` meets the requirements of `BidirectionalIterator` at most `std::distance(first,last)/2` swaps are done.

Possible implementation

```
template<class BidirIt, class UnaryPredicate>
BidirIt partition(BidirIt first, BidirIt last, UnaryPredicate p)
{
    while (1) {
        while ((first != last) && p(*first)) {
            ++first;
        }
        if (first == last--) break;
        while ((first != last) && !p(*last)) {
            --last;
        }
        if (first == last) break;
        std::iter_swap(first++, last);
    }
    return first;
}
```

Example

Run this code

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>
#include <forward_list>

template <class ForwardIt>
 void quicksort(ForwardIt first, ForwardIt last)
 {
    if(first == last) return;
    auto pivot = *std::next(first, std::distance(first,last)/2);
    ForwardIt middle1 = std::partition(first, last,
                         [pivot](const auto& em){ return em < pivot; });
    ForwardIt middle2 = std::partition(middle1, last,
                         [pivot](const auto& em){ return !(pivot < em); });
    quicksort(first, middle1);
    quicksort(middle2, last);
 }

int main()
{
    std::vector<int> v = {0,1,2,3,4,5,6,7,8,9};
    std::cout << "Original vector:\n    ";
    for (int elem : v) std::cout << elem << ' ';

    auto it = std::partition(v.begin(), v.end(), [](int i){return i % 2 == 0;});

    std::cout << "\nPartitioned vector:\n    ";
    std::copy(std::begin(v), it, std::ostream_iterator<int>(std::cout, " "));
    std::cout << " * ";
    std::copy(it, std::end(v), std::ostream_iterator<int>(std::cout, " "));

    std::forward_list<int> fl = {1, 30, -4, 3, 5, -4, 1, 6, -8, 2, -5, 64, 1, 92};
    std::cout << "\nUnsorted list:\n    ";
    for(int n : fl) std::cout << n << ' ';
    std::cout << '\n';

    quicksort(std::begin(fl), std::end(fl));
    std::cout << "Sorted using quicksort:\n    ";
    for(int fi : fl) std::cout << fi << ' ';
    std::cout << '\n';
}
```

Output:

```
Original vector:
    0 1 2 3 4 5 6 7 8 9
Partitioned vector:
    0 8 2 6 4  *  5 3 7 1 9
Unsorted list:
    1 30 -4 3 5 -4 1 6 -8 2 -5 64 1 92
Sorted using quicksort:
    -8 -5 -4 -4 1 1 1 2 3 5 6 30 64 92
```

See also

`is_partitioned` (C++11)	determines if the range is partitioned by the given predicate (function template)
`stable_partition`	divides elements into two groups while preserving their relative order (function template)
`std::experimental::parallel::partition` (parallelism TS)	parallelized version of `std::partition` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/partition&oldid=79901"

std::partition_copy

Defined in header `<algorithm>`

```
template< class InputIt, class OutputIt1,
          class OutputIt2, class UnaryPredicate >
std::pair<OutputIt1, OutputIt2>
     partition_copy( InputIt first, InputIt last,
                     OutputIt1 d_first_true, OutputIt2 d_first_false,
                     UnaryPredicate p );
```

(since C++11)

Copies the elements from the range `[first, last)` to two different ranges depending on the value returned by the predicate `p`. The elements, that satisfy the predicate `p`, are copied to the range beginning at `d_first_true`. The rest of the elements are copied to the range beginning at `d_first_false`.

The behavior is undefined if the input range overlaps either of the output ranges.

Parameters

first, last - the range of elements to sort

d_first_true - the beginning of the output range for the elements that satisfy p

d_first_false - the beginning of the output range for the elements that do not satisfy p

p - unary predicate which returns `true` if the element should be placed in d_first_true.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type &a);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it.
The type `Type` must be such that an object of type `InputIt` can be dereferenced and then implicitly converted to `Type`.

Type requirements

- `InputIt` must meet the requirements of `InputIterator`.
- The type of dereferenced `InputIt` must meet the requirements of `CopyAssignable`.
- `OutputIt1` must meet the requirements of `OutputIterator`.
- `OutputIt2` must meet the requirements of `OutputIterator`.
- `UnaryPredicate` must meet the requirements of `Predicate`.

Return value

A `pair` constructed from the iterator to the end of the `d_first_true` range and the iterator to the end of the `d_first_false` range.

Complexity

Exactly `distance(first, last)` applications of `p`.

Possible implementation

```
template<class InputIt, class OutputIt1,
         class OutputIt2, class UnaryPredicate>
std::pair<OutputIt1, OutputIt2>
    partition_copy(InputIt first, InputIt last,
                   OutputIt1 d_first_true, OutputIt2 d_first_false,
                   UnaryPredicate p)
```

```
{
    while (first != last) {
        if (p(*first)) {
            *d_first_true = *first;
            ++d_first_true;
        } else {
            *d_first_false = *first;
            ++d_first_false;
        }
        ++first;
    }
    return std::pair<OutputIt1, OutputIt2>(d_first_true, d_first_false);
}
```

Example

Run this code

```
#include <iostream>
#include <algorithm>
#include <utility>

int main()
{
    int arr [10] = {1,2,3,4,5,6,7,8,9,10};
    int true_arr [5] = {0};
    int false_arr [5] = {0};

    std::partition_copy(std::begin(arr), std::end(arr), std::begin(true_arr),std::begin(fals
                        [] (int i) {return i > 5;});

    std::cout << "true_arr: ";
    for (auto it = std::begin(true_arr); it != std::end(true_arr); ++it) {
        std::cout << *it << ' ';
    }
    std::cout << '\n';

    std::cout << "false_arr: ";
    for (auto it = std::begin(false_arr); it != std::end(false_arr); ++it) {
        std::cout << *it << ' ';
    }
    std::cout << '\n';

    return 0;

}
```

Output:

```
true_arr: 6 7 8 9 10
false_arr: 1 2 3 4 5
```

See also

partition	divides a range of elements into two groups (function template)
stable_partition	divides elements into two groups while preserving their relative order (function template)
	parallelized version of

`std::experimental::parallel::partition_copy` (parallelism TS)	`std::partition_copy` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/partition_copy&oldid=79902"

std::stable_partition

Defined in header `<algorithm>`

```
template< class BidirIt, class UnaryPredicate >
BidirIt stable_partition( BidirIt first, BidirIt last, UnaryPredicate p );
```

Reorders the elements in the range `[first, last)` in such a way that all elements for which the predicate `p` returns `true` precede the elements for which predicate `p` returns `false`. Relative order of the elements is preserved.

Parameters

first, last - the range of elements to reorder

p - unary predicate which returns `true` if the element should be ordered before other elements.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type &a);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it.
The type `Type` must be such that an object of type `BidirIt` can be dereferenced and then implicitly converted to `Type`.

Type requirements

- `BidirIt` must meet the requirements of `ValueSwappable` and `BidirectionalIterator`.
- The type of dereferenced `BidirIt` must meet the requirements of `MoveAssignable` and `MoveConstructible`.
- `UnaryPredicate` must meet the requirements of `Predicate`.

Return value

Iterator to the first element of the second group

Complexity

Exactly `last-first` applications of the predicate and at most `(last-first)*log(last-first)` swaps if there is insufficient memory or linear number of swaps if sufficient memory is available.

Notes

This function attempts to allocate a temporary buffer, typically by calling `std::get_temporary_buffer`. If the allocation fails, the less efficient algorithm is chosen.

Example

Run this code

```
#include <iostream>
#include <algorithm>
#include <vector>

int main()
{
    std::vector<int> v{0, 0, 3, 0, 2, 4, 5, 0, 7};
    std::stable_partition(v.begin(), v.end(), [](int n){return n>0;});
    for (int n : v) {
        std::cout << n << ' ';
    }
    std::cout << '\n';
}
```

Output:

```
3 2 4 5 7 0 0 0 0
```

See also

partition	divides a range of elements into two groups (function template)
std::experimental::parallel::stable_partition (parallelism TS)	parallelized version of `std::stable_partition` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/stable_partition&oldid=79919"

std::partition_point

Defined in header `<algorithm>`

`template< class ForwardIt, class UnaryPredicate >`
`ForwardIt partition_point( ForwardIt first, ForwardIt last, UnaryPredicate p );`
	(1)	(since C++11)

Examines the partitioned (as if by `std::partition`) range `[first, last)` and locates the end of the first partition, that is, the first element that does not satisfy `p` or `last` if all elements satisfy `p`.

Parameters

`first, last` - the partitioned range of elements to examine

`p` - unary predicate which returns `true` for the elements found in the beginning of the range.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type &a);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it.
The type `Type` must be such that an object of type `ForwardIt` can be dereferenced and then implicitly converted to `Type`.

Type requirements

- `ForwardIt` must meet the requirements of `ForwardIterator`.
- `UnaryPredicate` must meet the requirements of `Predicate`.

Return value

The iterator past the end of the first partition within `[first, last)` or `last` if all elements satisfy `p`.

Complexity

The number of applications of the predicate p is logarithmic in the distance between `first` and `last`

Example

Run this code

```
#include <algorithm>
#include <array>
#include <iostream>
#include <iterator>

int main()
{
    std::array<int, 9> v = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    auto is_even = [](int i){ return i % 2 == 0; };
    std::partition(v.begin(), v.end(), is_even);

    auto p = std::partition_point(v.begin(), v.end(), is_even);

    std::cout << "Before partition:\n    ";
    std::copy(v.begin(), p, std::ostream_iterator<int>(std::cout, " "));
    std::cout << "\nAfter partition:\n    ";
    std::copy(p, v.end(), std::ostream_iterator<int>(std::cout, " "));
}
```

Output:

```
Before partition:
    8 2 6 4
After partition:
    5 3 7 1 9
```

See also

is_sorted (C++11)	checks whether a range is sorted into ascending order (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/partition_point&oldid=67253"

std::is_sorted

Defined in header `<algorithm>`

`template< class ForwardIt >`
`bool is_sorted( ForwardIt first, ForwardIt last );`
	(1)	(since C++11)

`template< class ForwardIt, class Compare >`
`bool is_sorted( ForwardIt first, ForwardIt last, Compare comp );`
	(2)	(since C++11)

Checks if the elements in range `[first, last)` are sorted in ascending order. The first version of the function uses `operator<` to compare the elements, the second uses the given comparison function `comp`.

Parameters

first, last - the range of elements to examine

comp - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than (i.e. is ordered *before*) the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `ForwardIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `ForwardIt` must meet the requirements of `ForwardIterator`.

Return value

`true` if the elements in the range are sorted in ascending order

Complexity

linear in the distance between `first` and `last`

Possible implementation

First version

```
template<class ForwardIt>
bool is_sorted(ForwardIt first, ForwardIt last)
{
    return std::is_sorted_until(first, last) == last;
}
```

Second version

```
template<class ForwardIt, class Compare>
bool is_sorted(ForwardIt first, ForwardIt last, Compare comp)
{
    return std::is_sorted_until(first, last, comp) == last;
}
```

Example

Run this code

```
#include <iostream>
#include <algorithm>

int main()
{
    const int N = 5;
    int digits[N] = {3, 1, 4, 1, 5};

    for (auto i : digits) std::cout << i << ' ';
    std::cout << ": is_sorted: " << std::is_sorted(digits, digits+N) << '\n';

    std::sort(digits, digits+N);

    for (auto i : digits) std::cout << i << ' ';
    std::cout << ": is_sorted: " << std::is_sorted(digits, digits+N) << '\n';
}
```

Output:

```
3 1 4 1 5 : is_sorted: 0
1 1 3 4 5 : is sorted: 1
```

See also

is_sorted_until (C++11)	finds the largest sorted subrange (function template)
std::experimental::parallel::is_sorted (parallelism TS)	parallelized version of std::is_sorted (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/is_sorted&oldid=79889"

std::is_sorted_until

Defined in header `<algorithm>`

```
template< class ForwardIt >
ForwardIt is_sorted_until( ForwardIt first, ForwardIt last );
```

(1) (since C++11)

```
template< class ForwardIt, class Compare >
ForwardIt is_sorted_until( ForwardIt first, ForwardIt last,
                           Compare comp );
```

(2) (since C++11)

Examines the range `[first, last)` and finds the largest range beginning at `first` in which the elements are sorted in ascending order. The first version of the function uses `operator<` to compare the elements, the second uses the given comparison function `comp`.

Parameters

first, last	-	the range of elements to examine
comp	-	comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than (i.e. is ordered *before*) the second. The signature of the comparison function should be equivalent to the following: `bool cmp(const Type1 &a, const Type2 &b);` The signature does not need to have `const &`, but the function object must not modify the objects passed to it. The types `Type1` and `Type2` must be such that an object of type `ForwardIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `ForwardIt` must meet the requirements of `ForwardIterator`.

Return value

The upper bound of the largest range beginning at `first` in which the elements are sorted in ascending order. That is, the last iterator `it` for which range `[first, it)` is sorted.

Complexity

linear in the distance between `first` and `last`

Possible implementation

First version

```
template<class ForwardIt>
ForwardIt is_sorted_until(ForwardIt first, ForwardIt last)
{
    if (first != last) {
        ForwardIt next = first;
        while (++next != last) {
            if (*next < *first)
                return next;
            first = next;
        }
    }
    return last;
}
```

Second version

```
template <class ForwardIt, class Compare>
ForwardIt is_sorted_until(ForwardIt first, ForwardIt last, Compare comp)
{
    using namespace std::placeholders;
    ForwardIt it = std::adjacent_find(first, last, std::bind(comp, _2, _1));
    return it == last ? last : std::next(it);
}
```

Example

Run this code

```
#include <iostream>
#include <algorithm>
#include <iterator>
#include <random>

int main()
{
    std::random_device rd;
    std::mt19937 g(rd());
    const int N = 6;
    int nums[N] = {3, 1, 4, 1, 5, 9};

    const int min_sorted_size = 4;
    int sorted_size = 0;
    do {
        std::random_shuffle(nums, nums + N, g);
        int *sorted_end = std::is_sorted_until(nums, nums + N);
        sorted_size = std::distance(nums, sorted_end);

        for (auto i : nums) std::cout << i << ' ';
        std::cout << " : " << sorted_size << " initial sorted elements\n";
    } while (sorted_size < min_sorted_size);
}
```

Possible output:

```
4 1 9 5 1 3  : 1 initial sorted elements
4 5 9 3 1 1  : 3 initial sorted elements
9 3 1 4 5 1  : 1 initial sorted elements
1 3 5 4 1 9  : 3 initial sorted elements
5 9 1 1 3 4  : 2 initial sorted elements
4 9 1 5 1 3  : 2 initial sorted elements
1 1 4 9 5 3  : 4 initial sorted elements
```

See also

is_sorted (C++11)	checks whether a range is sorted into ascending order (function template)
std::experimental::parallel::is_sorted_until (parallelism TS)	parallelized version of `std::is_sorted_until` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/is_sorted_until&oldid=79890"

std::sort

Defined in header `<algorithm>`

`template< class RandomIt >`
`void sort( RandomIt first, RandomIt last );`
	(1)

`template< class RandomIt, class Compare >`
`void sort( RandomIt first, RandomIt last, Compare comp );`
	(2)

Sorts the elements in the range `[first, last)` in ascending order. The order of equal elements is not guaranteed to be preserved. The first version uses `operator<` to compare the elements, the second version uses the given comparison function object `comp`.

Parameters

first, last - the range of elements to sort

comp - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than (i.e. is ordered *before*) the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `RandomIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `RandomIt` must meet the requirements of `ValueSwappable` and `RandomAccessIterator`.
- The type of dereferenced `RandomIt` must meet the requirements of `MoveAssignable` and `MoveConstructible`.
- `Compare` must meet the requirements of `Compare`.

Return value

(none)

Complexity

O(N·log(N)), where `N = std::distance(first, last)` comparisons on average.	(until C++11)
O(N·log(N)), where `N = std::distance(first, last)` comparisons.	(since C++11)

Example

Run this code

```
#include <algorithm>
#include <functional>
#include <array>
#include <iostream>

int main()
{
    std::array<int, 10> s = {5, 7, 4, 2, 8, 6, 1, 9, 0, 3};
    // sort using the default operator<
    std::sort(s.begin(), s.end());
    for (int a : s) {
        std::cout << a << " ";
    }
    std::cout << '\n';

    // sort using a standard library compare function object
    std::sort(s.begin(), s.end(), std::greater<int>());
    for (int a : s) {
        std::cout << a << " ";
    }
    std::cout << '\n';

    // sort using a custom function object
    struct {
        bool operator()(int a, int b)
        {
            return a < b;
        }
    } customLess;
    std::sort(s.begin(), s.end(), customLess);
    for (int a : s) {
        std::cout << a << " ";
    }
    std::cout << '\n';

    // sort using a lambda expression
    std::sort(s.begin(), s.end(), [](int a, int b) {
        return b < a;
    });
    for (int a : s) {
        std::cout << a << " ";
    }
    std::cout << '\n';
}
```

Output:

```
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
```

See also

partial_sort	sorts the first N elements of a range (function template)
stable_sort	sorts a range of elements while preserving order between equal elements (function template)
std::experimental::parallel::sort (parallelism TS)	parallelized version of `std::sort` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/sort&oldid=79918"

std::partial_sort

Defined in header `<algorithm>`

```
template< class RandomIt >
void partial_sort( RandomIt first, RandomIt middle, RandomIt last );
```
(1)

```
template< class RandomIt, class Compare >
void partial_sort( RandomIt first, RandomIt middle, RandomIt last, Compare comp );
```
(2)

Rearranges elements such that the range `[first, middle)` contains the sorted `middle - first` smallest elements in the range `[first, last)`.

The order of equal elements is not guaranteed to be preserved. The order of the remaining elements in the range `[middle, last)` is unspecified. The first version uses `operator<` to compare the elements, the second version uses the given comparison function `comp`.

Parameters

first, last - the range of elements to sort

comp - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than (i.e. is ordered *before*) the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `RandomIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `RandomIt` must meet the requirements of `ValueSwappable` and `RandomAccessIterator`.
- The type of dereferenced `RandomIt` must meet the requirements of `MoveAssignable` and `MoveConstructible`.

Return value

(none)

Complexity

Approximately *(last-first)log(middle-first))* applications of `cmp`

Example

Run this code

```
#include <algorithm>
#include <functional>
#include <array>
#include <iostream>

int main()
{
    std::array<int, 10> s{5, 7, 4, 2, 8, 6, 1, 9, 0, 3};

    std::partial_sort(s.begin(), s.begin() + 3, s.end());
    for (int a : s) {
        std::cout << a << " ";
    }
}
```

Possible output:

```
0 1 2 7 8 6 5 9 4 3
```

See also

`nth_element`	partially sorts the given range making sure that it is partitioned by the given element (function template)
`partial_sort_copy`	copies and partially sorts a range of elements (function template)
`stable_sort`	sorts a range of elements while preserving order between equal elements (function template)
`sort`	sorts a range into ascending order (function template)
`std::experimental::parallel::partial_sort` (parallelism TS)	parallelized version of `std::partial_sort` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/partial_sort&oldid=79899"

std::partial_sort_copy

Defined in header `<algorithm>`

```
template< class InputIt, class RandomIt >
RandomIt partial_sort_copy( InputIt first, InputIt last,
                            RandomIt d_first, RandomIt d_last );
```
(1)

```
template< class InputIt, class RandomIt, class Compare >
RandomIt partial_sort_copy( InputIt first, InputIt last,
                            RandomIt d_first, RandomIt d_last,
                            Compare comp );
```
(2)

Sorts some of the elements in the range `[first, last)` in ascending order, storing the result in the range `[d_first, d_last)`.

At most `d_last - d_first` of the elements are moved to the range `[d_first, d_first + n)` and then sorted. `n` is the number of elements to sort (`n = min(last - first, d_last - d_first)`). The order of equal elements is not guaranteed to be preserved. The first version uses `operator<` to compare the elements, the second version uses the given comparison function `comp`.

Parameters

first, last - the range of elements to sort

d_first, d_last - random access iterators defining the destination range

comp - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than (i.e. is ordered *before*) the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `RandomIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `InputIt` must meet the requirements of `InputIterator`.
- `RandomIt` must meet the requirements of `ValueSwappable` and `RandomAccessIterator`.
- The type of dereferenced `RandomIt` must meet the requirements of `MoveAssignable` and `MoveConstructible`.

Return value

an iterator to the element defining the upper boundary of the sorted range, i.e. `d_first + min(last - first, d_last - d_first)`.

Complexity

O(N·log(min(D,N)), where `N = std::distance(first, last)`, `D = std::distance(d_first, d_last)` applications of `cmp`.

Example

The following code sorts an vector of integers and copies them into a smaller and a larger vector.

Run this code

```
#include <algorithm>
#include <vector>
#include <functional>
#include <iostream>

int main()
{
    std::vector<int> v0{4, 2, 5, 1, 3};
    std::vector<int> v1{10, 11, 12};
    std::vector<int> v2{10, 11, 12, 13, 14, 15, 16};
    std::vector<int>::iterator it;

    it = std::partial_sort_copy(v0.begin(), v0.end(), v1.begin(), v1.end());

    std::cout << "Writing to the smaller vector in ascending order gives: ";
    for (int a : v1) {
        std::cout << a << " ";
    }
    std::cout << '\n';
    if(it == v1.end())
        std::cout << "The return value is the end iterator\n";

    it = std::partial_sort_copy(v0.begin(), v0.end(), v2.begin(), v2.end(),
                                std::greater<int>());

    std::cout << "Writing to the larger vector in descending order gives: ";
    for (int a : v2) {
        std::cout << a << " ";
    }
    std::cout << '\n' << "The return value is the iterator to " << *it << '\n';
}
```

Output:

```
Writing to the smaller vector in ascending order gives: 1 2 3
The return value is the end iterator
Writing to the larger vector in descending order gives: 5 4 3 2 1 15 16
The return value is the iterator to 15
```

See also

partial_sort	sorts the first N elements of a range (function template)
sort	sorts a range into ascending order (function template)
stable_sort	sorts a range of elements while preserving order between equal elements (function template)
std::experimental::parallel::partial_sort_copy (parallelism TS)	parallelized version of `std::partial_sort_copy` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/partial_sort_copy&oldid=79900"

std::stable_sort

Defined in header `<algorithm>`

```
template< class RandomIt >
void stable_sort( RandomIt first, RandomIt last );
```
(1)

```
template< class RandomIt, class Compare >
void stable_sort( RandomIt first, RandomIt last, Compare comp );
```
(2)

Sorts the elements in the range `[first, last)` in ascending order. The order of equal elements is guaranteed to be preserved. The first version uses `operator<` to compare the elements, the second version uses the given comparison function `comp`.

Parameters

`first, last` - the range of elements to sort

`comp` - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than (i.e. is ordered *before*) the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `RandomIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `RandomIt` must meet the requirements of `ValueSwappable` and `RandomAccessIterator`.
- The type of dereferenced `RandomIt` must meet the requirements of `MoveAssignable` and `MoveConstructible`.

Return value

(none)

Complexity

$O(N\cdot log^2(N))$, where `N = std::distance(first, last)` applications of `cmp`. If additional memory is available, then the complexity is $O(N\cdot log(N))$.

Notes

This function attempts to allocate a temporary buffer equal in size to the sequence to be sorted, typically by calling `std::get_temporary_buffer`. If the allocation fails, the less efficient algorithm is chosen.

Example

Run this code

```
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
```

```
struct Employee {
    Employee(int age, std::string name) : age(age), name(name) { }
    int age;
    std::string name;  // Does not particpate in comparisons
};

bool operator<(const Employee &lhs, const Employee &rhs) {
    return lhs.age < rhs.age;
}

int main()
{
    std::vector<Employee> v = {
        Employee(108, "Zaphod"),
        Employee(32, "Arthur"),
        Employee(108, "Ford"),
    };

    std::stable_sort(v.begin(), v.end());

    for (const Employee &e : v) {
        std::cout << e.age << ", " << e.name << '\n';
    }
}
```

Output:

```
32, Arthur
108, Zaphod
108, Ford
```

See also

partial_sort	sorts the first N elements of a range (function template)
sort	sorts a range into ascending order (function template)
std::experimental::parallel::stable_sort (parallelism TS)	parallelized version of `std::stable_sort` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/stable_sort&oldid=79920"

std::nth_element

Defined in header <algorithm>

```
template< class RandomIt >
void nth_element( RandomIt first, RandomIt nth, RandomIt last );
```
(1)

```
template< class RandomIt, class Compare >
void nth_element( RandomIt first, RandomIt nth, RandomIt last, Compare comp );
```
(2)

`nth_element` is a partial sorting algorithm that rearranges elements in `[first, last)` such that:

- The element pointed at by `nth` is changed to whatever element would occur in that position if `[first, last)` was sorted.
- All of the elements before this new `nth` element are less than or equal to the elements after the new `nth` element.

More formally, `nth_element` partially sorts the range `[first, last)` in ascending order so that the condition `!(*j < *i)` (for the first version, or `comp(*j, *i) == false` for the second version) is met for any `i` in the range `[first, nth)` and for any `j` in the range `[nth, last)`. The element placed in the `nth` position is exactly the element that would occur in this position if the range was fully sorted.

`nth` may be the end iterator, in this case the function has no effect.

Parameters

`first, last` - random access iterators defining the range sort

`nth` - random access iterator defining the sort partition point

`comp` - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than (i.e. is ordered *before*) the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `RandomIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `RandomIt` must meet the requirements of `ValueSwappable` and `RandomAccessIterator`.
- The type of dereferenced `RandomIt` must meet the requirements of `MoveAssignable` and `MoveConstructible`.

Return value

(none)

Complexity

Linear in `std::distance(first, last)` on average.

Notes

The algorithm used is typically introselect although other selection algorithms with suitable average-case complexity are allowed.

Example

Run this code

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

int main()
{
    std::vector<int> v{5, 6, 4, 3, 2, 6, 7, 9, 3};

    std::nth_element(v.begin(), v.begin() + v.size()/2, v.end());
    std::cout << "The median is " << v[v.size()/2] << '\n';

    std::nth_element(v.begin(), v.begin()+1, v.end(), std::greater<int>());
    std::cout << "The second largest element is " << v[1] << '\n';
}
```

Output:

```
The median is 5
The second largest element is 7
```

See also

partial_sort_copy	copies and partially sorts a range of elements (function template)
stable_sort	sorts a range of elements while preserving order between equal elements (function template)
sort	sorts a range into ascending order (function template)
std::experimental::parallel::nth_element (parallelism TS)	parallelized version of `std::nth_element` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/nth_element&oldid=79898"

std::lower_bound

Defined in header `<algorithm>`

```
template< class ForwardIt, class T >
ForwardIt lower_bound( ForwardIt first, ForwardIt last, const T& value );                    (1)
```

```
template< class ForwardIt, class T, class Compare >
ForwardIt lower_bound( ForwardIt first, ForwardIt last, const T& value, Compare comp );      (2)
```

Returns an iterator pointing to the first element in the range `[first, last)` that is *not less* than (i.e. greater or equal to) `value`.

The range `[first, last)` must be at least partially ordered, i.e. partitioned with respect to the expression `element < value` or `comp(element, value)`. A fully-sorted range meets this criterion, as does a range resulting from a call to `std::partition`.

The first version uses `operator<` to compare the elements, the second version uses the given comparison function `comp`.

Parameters

first, last - iterators defining the partially-ordered range to examine

value - value to compare the elements to

comp - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The type `Type1` must be such that an object of type `ForwardIt` can be dereferenced and then implicitly converted to `Type1`. The type `Type2` must be such that an object of type `T` can be implicitly converted to `Type2`.

Type requirements

- `ForwardIt` must meet the requirements of `ForwardIterator`.

Return value

Iterator pointing to the first element that is not *less* than `value`, or `last` if no such element is found.

Complexity

The number of comparisons performed is logarithmic in the distance between `first` and `last` (At most $\log_2(last - first) + O(1)$ comparisons). However, for non-`RandomAccessIterator`s, the number of iterator increments is linear.

Possible implementation

First version

```
template<class ForwardIt, class T>
ForwardIt lower_bound(ForwardIt first, ForwardIt last, const T& value)
{
    ForwardIt it;
    typename std::iterator_traits<ForwardIt>::difference_type count, step;
    count = std::distance(first, last);

    while (count > 0) {
        it = first;
        step = count / 2;
        std::advance(it, step);
        if (*it < value) {
            first = ++it;
            count -= step + 1;
        }
        else
            count = step;
    }
    return first;
}
```

Second version

```
template<class ForwardIt, class T, class Compare>
ForwardIt lower_bound(ForwardIt first, ForwardIt last, const T& value, Compare comp)
{
    ForwardIt it;
    typename std::iterator_traits<ForwardIt>::difference_type count, step;
    count = std::distance(first,last);

    while (count > 0) {
        it = first;
        step = count / 2;
        std::advance(it, step);
        if (comp(*it, value)) {
            first = ++it;
            count -= step + 1;
        }
        else
            count = step;
    }
    return first;
}
```

Example

Run this code

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

int main()
{
    std::vector<int> data = { 1, 1, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 6 };

    auto lower = std::lower_bound(data.begin(), data.end(), 4);
    auto upper = std::upper_bound(data.begin(), data.end(), 4);

    std::copy(lower, upper, std::ostream_iterator<int>(std::cout, " "));
}
```

Output:

```
4 4 4
```

See also

`equal_range`	returns range of elements matching a specific key (function template)
`partition`	divides a range of elements into two groups (function template)
`upper_bound`	returns an iterator to the first element *greater* than a certain value (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/lower_bound&oldid=78134"

std::upper_bound

Defined in header `<algorithm>`

```
template< class ForwardIt, class T >
ForwardIt upper_bound( ForwardIt first, ForwardIt last, const T& value );
```
(1)

```
template< class ForwardIt, class T, class Compare >
ForwardIt upper_bound( ForwardIt first, ForwardIt last, const T& value, Compare comp );
```
(2)

Returns an iterator pointing to the first element in the range `[first, last)` that is *greater* than `value`.

The range `[first, last)` must be at least partially ordered, i.e. partitioned with respect to the expression `!(value < element)` or `!comp(value, element)`. A fully-sorted range meets this criterion, as does a range resulting from a call to `std::partition`.

The first version uses `operator<` to compare the elements, the second version uses the given comparison function `comp`.

Parameters

first, last - the range of elements to examine

value - value to compare the elements to

comp - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `T` can be implicitly converted to both `Type1` and `Type2`, and an object of type `ForwardIt` can be dereferenced and then implicitly converted to both `Type1` and `Type2`.

Type requirements

- `ForwardIt` must meet the requirements of `ForwardIterator`.

Return value

iterator pointing to the first element that is *greater* than `value`, or `last` if no such element is found.

Complexity

The number of comparisons performed is logarithmic in the distance between `first` and `last` (At most $log_2(last - first) + O(1)$ comparisons). However, for non-`RandomAccessIterator`s, the number of iterator increments is linear.

Possible implementation

First version

```
template<class ForwardIt, class T>
ForwardIt upper_bound(ForwardIt first, ForwardIt last, const T& value)
{
    ForwardIt it;
    typename std::iterator_traits<ForwardIt>::difference_type count, step;
    count = std::distance(first,last);

    while (count > 0) {
        it = first;
        step = count / 2;
        std::advance(it, step);
        if (!(value < *it)) {
            first = ++it;
            count -= step + 1;
        } else count = step;
    }
    return first;
}
```

Second version

```
template<class ForwardIt, class T, class Compare>
ForwardIt upper_bound(ForwardIt first, ForwardIt last, const T& value, Compare comp)
{
    ForwardIt it;
    typename std::iterator_traits<ForwardIt>::difference_type count, step;
    count = std::distance(first,last);

    while (count > 0) {
        it = first;
        step = count / 2;
        std::advance(it, step);
        if (!comp(value, *it)) {
            first = ++it;
            count -= step + 1;
        } else count = step;
    }
    return first;
}
```

Example

Run this code

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

int main()
{
    std::vector<int> data = { 1, 1, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 6 };

    auto lower = std::lower_bound(data.begin(), data.end(), 4);
    auto upper = std::upper_bound(data.begin(), data.end(), 4);

    std::copy(lower, upper, std::ostream_iterator<int>(std::cout, " "));
}
```

Output:

```
4 4 4
```

See also

`equal_range`	returns range of elements matching a specific key (function template)
`lower_bound`	returns an iterator to the first element *not less* than the given value (function template)
`partition`	divides a range of elements into two groups (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/upper_bound&oldid=77487"

std::binary_search

Defined in header `<algorithm>`

```
template< class ForwardIt, class T >
bool binary_search( ForwardIt first, ForwardIt last, const T& value );
```
(1)

```
template< class ForwardIt, class T, class Compare >
bool binary_search( ForwardIt first, ForwardIt last, const T& value, Compare comp );
```
(2)

Checks if an element equivalent to `value` appears within the range `[first, last)`.

For `std::binary_search` to succeed, the range `[first, last)` must be at least partially ordered, i.e. it must satisfy all of the following requirements:

- partitioned with respect to `element < value` or `comp(element, value)`
- partitioned with respect to `!(value < element)` or `!comp(value, element)`
- for all elements, if `element < value` or `comp(element, value)` is `true` then `!(value < element)` or `!comp(value, element)` is also `true`

A fully-sorted range meets these criteria, as does a range resulting from a call to `std::partition`.

The first version uses `operator<` to compare the elements, the second version uses the given comparison function `comp`.

Parameters

`first, last` - the range of elements to examine

`value` - value to compare the elements to

`comp` - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than (i.e. is ordered *before*) the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `T` can be implicitly converted to both `Type1` and `Type2`, and an object of type `ForwardIt` can be dereferenced and then implicitly converted to both `Type1` and `Type2`.

Type requirements

- `ForwardIt` must meet the requirements of `ForwardIterator`.

Return value

`true` if an element equal to `value` is found, `false` otherwise.

Complexity

The number of comparisons performed is logarithmic in the distance between `first` and `last` (At most $log_2(last - first) + O(1)$ comparisons). However, for non-`RandomAccessIterator`s, number of iterator increments is linear.

Possible implementation

First version

```
template<class ForwardIt, class T>
bool binary_search(ForwardIt first, ForwardIt last, const T& value)
{
    first = std::lower_bound(first, last, value);
    return (!(first == last) && !(value < *first));
}
```

Second version

```
template<class ForwardIt, class T, class Compare>
bool binary_search(ForwardIt first, ForwardIt last, const T& value, Compare comp)
{
    first = std::lower_bound(first, last, value, comp);
    return (!(first == last) && !(comp(value, *first)));
}
```

Example

Run this code

```
#include <iostream>
#include <algorithm>
#include <vector>

int main()
{
    std::vector<int> haystack {1, 3, 4, 5, 9};
    std::vector<int> needles {1, 2, 3};

    for (auto needle : needles) {
        std::cout << "Searching for " << needle << '\n';
        if (std::binary_search(haystack.begin(), haystack.end(), needle)) {
            std::cout << "Found " << needle << '\n';
        } else {
            std::cout << "no dice!\n";
        }
    }
}
```

Output:

```
Searching for 1
Found 1
Searching for 2
no dice!
Searching for 3
Found 3
```

See also

equal_range	returns range of elements matching a specific key (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/binary_search&oldid=78428"

std::equal_range

Defined in header `<algorithm>`

```
template< class ForwardIt, class T >
std::pair<ForwardIt,ForwardIt>
    equal_range( ForwardIt first, ForwardIt last,
                 const T& value );
```
(1)

```
template< class ForwardIt, class T, class Compare >
std::pair<ForwardIt,ForwardIt>
    equal_range( ForwardIt first, ForwardIt last,
                 const T& value, Compare comp );
```
(2)

Returns a range containing all elements equivalent to `value` in the range `[first, last)`.

The range `[first, last)` must be partitioned with respect to comparison with `value`, i.e. it must satisfy all of the following requirements:

- partitioned with respect to `element < value` or `comp(element, value)`
- partitioned with respect to `!(value < element)` or `!comp(value, element)`
- for all elements, if `element < value` or `comp(element, value)` is `true` then `!(value < element)` or `!comp(value, element)` is also `true`

A fully-sorted range meets these criteria, as does a range resulting from a call to `std::partition`.

The returned range is defined by two iterators, one pointing to the first element that is *not less* than `value` and another pointing to the first element *greater* than `value`. The first iterator may be alternatively obtained with `std::lower_bound()`, the second - with `std::upper_bound()`.

The first version uses `operator<` to compare the elements, the second version uses the given comparison function `comp`.

Parameters

`first, last` - the range of elements to examine

`value` - value to compare the elements to

`comp` - comparison function which returns `true` if the first argument is *less* than the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it.
`cmp` will be called as both `cmp(value, *iterator)` and `cmp(*iterator, value)`.

Type requirements

- `ForwardIt` must meet the requirements of `ForwardIterator`.

Return value

`std::pair` containing a pair of iterators defining the wanted range, the first pointing to the first element that is *not less* than `value` and the second pointing to the first element *greater* than `value`.

If there are no elements *not less* than `value`, `last` is returned as the first element. Similarly if there are no elements *greater* than `value`, `last` is returned as the second element

Complexity

The number of comparisons performed is logarithmic in the distance between `first` and `last` (At most $2 * \log_2(last - first) + O(1)$ comparisons). However, for non-`RandomAccessIterator`s, the number of iterator increments is linear.

Possible implementation

First version

```
template<class ForwardIt, class T>
std::pair<ForwardIt,ForwardIt>
    equal_range(ForwardIt first, ForwardIt last,
                const T& value)
{
    return std::make_pair(std::lower_bound(first, last, value),
                          std::upper_bound(first, last, value));
}
```

Second version

```
template<class ForwardIt, class T, class Compare>
std::pair<ForwardIt,ForwardIt>
    equal_range(ForwardIt first, ForwardIt last,
                const T& value, Compare comp);
{
    return std::make_pair(std::lower_bound(first, last, value, comp),
                          std::upper_bound(first, last, value, comp));
}
```

Example

Run this code

```
#include <algorithm>
#include <vector>
#include <iostream>

struct S
{
    int number;
    char name;

    S ( int number, char name  )
        : number ( number ), name ( name )
    {}

    // only the number is relevant with this comparison
    bool operator< ( const S& s ) const
    {
        return number < s.number;
    }
};


int main()
{
    // note: not ordered, only partitioned w.r.t. S defined below
    std::vector<S> vec = { {1,'A'}, {2,'B'}, {2,'C'}, {2,'D'}, {4,'G'}, {3,'F'} };
    S value ( 2, '?' );

    auto p = std::equal_range(vec.begin(),vec.end(),value);

    for ( auto i = p.first; i != p.second; ++i )
        std::cout << i->name << ' ';
}
```

Output:

```
B C D
```

Example With Comparator

Run this code

```
#include <algorithm>
#include <vector>
#include <iostream>

struct S
{
    int number;
    char name;

    S ( int number, char name  )
        : number ( number ), name ( name )
    {}

    // only the number is relevant with this comparison
    bool operator< ( const S& s ) const
    {
        return number < s.number;
    }
};

struct Comp
{
    bool operator() ( const S& s, int i )
    {
        return s.number < i;
    }

    bool operator() ( int i, const S& s )
    {
        return i < s.number;
    }
};

int main()
{
    // note: not ordered, only partitioned w.r.t. S defined below
    std::vector<S> vec = { {1,'A'}, {2,'B'}, {2,'C'}, {2,'D'}, {4,'G'}, {3,'F'} };

    auto p = std::equal_range(vec.begin(),vec.end(),2,Comp());

    for ( auto i = p.first; i != p.second; ++i )
        std::cout << i->name << ' ';
}
```

Output:

```
B C D
```

See also

`lower_bound`	returns an iterator to the first element *not less* than the given value (function template)
`upper_bound`	returns an iterator to the first element *greater* than a certain value (function template)
`binary_search`	determines if an element exists in a certain range (function template)
`partition`	divides a range of elements into two groups (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/equal_range&oldid=77489"

std::merge

Defined in header `<algorithm>`

```
template< class InputIt1, class InputIt2, class OutputIt >
OutputIt merge( InputIt1 first1, InputIt1 last1,
                InputIt2 first2, InputIt2 last2,
                OutputIt d_first );
```
(1)

```
template< class InputIt1, class InputIt2, class OutputIt, class Compare>
OutputIt merge( InputIt1 first1, InputIt1 last1,
                InputIt2 first2, InputIt2 last2,
                OutputIt d_first, Compare comp );
```
(2)

Merges two sorted ranges `[first1, last1)` and `[first2, last2)` into one sorted range beginning at `d_first`. The first version uses `operator<` to compare the elements, the second version uses the given comparison function `comp`. The relative order of equivalent elements is preserved.

The behavior is undefined if the destination range overlaps either of the input ranges (the input ranges may overlap each other).

Parameters

first1, last1 - the first range of elements to merge

first2, last2 - the second range of elements to merge

d_first - the beginning of the destination range

comp - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than (i.e. is ordered *before*) the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that objects of types `InputIt1` and `InputIt2` can be dereferenced and then implicitly converted to both `Type1` and `Type2`.

Type requirements

- `InputIt1` must meet the requirements of `InputIterator`.
- `InputIt2` must meet the requirements of `InputIterator`.
- `OutputIt` must meet the requirements of `OutputIterator`.

Return value

An output iterator to element past the last element copied.

Complexity

At most `std::distance(first1, last1) + std::distance(first2, last2) - 1` comparisons.

Notes

This algorithm performs a similar task as `std::set_union` does. Both consume two sorted input ranges and produce a sorted output with elements from both inputs. The difference bewteen these two algorithms is with handling values from both input ranges which compare equivalent (see notes on `LessThanComparable`). If any equivalent values appeared `n` times in the first range and `m` times in the second, `std::merge` would output all `n+m` occurrences whereas `std::set_union` would output `std::max(n, m)` ones only. So `std::merge` outputs exactly `std::distance(first1, last1) + std::distance(first2, last2)` values and `std::set_union` may produce less.

Possible implementation

First version

```
template<class InputIt1, class InputIt2, class OutputIt>
OutputIt merge(InputIt1 first1, InputIt1 last1,
               InputIt2 first2, InputIt2 last2,
               OutputIt d_first)
{
    for (; first1 != last1; ++d_first) {
        if (first2 == last2) {
            return std::copy(first1, last1, d_first);
        }
        if (*first2 < *first1) {
            *d_first = *first2;
            ++first2;
        } else {
            *d_first = *first1;
            ++first1;
        }
    }
    return std::copy(first2, last2, d_first);
}
```

Second version

```
template<class InputIt1, class InputIt2,
         class OutputIt, class Compare>
OutputIt merge(InputIt1 first1, InputIt1 last1,
               InputIt2 first2, InputIt2 last2,
               OutputIt d_first, Compare comp)
{
    for (; first1 != last1; ++d_first) {
        if (first2 == last2) {
            return std::copy(first1, last1, d_first);
        }
        if (comp(*first2, *first1)) {
            *d_first = *first2;
            ++first2;
        } else {
            *d_first = *first1;
            ++first1;
        }
    }
    return std::copy(first2, last2, d_first);
}
```

Example

Run this code

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include <vector>
#include <random>
```

```
#include <functional>

int main()
{
    // fill the vectors with random numbers
    std::random_device rd;
    std::mt19937 mt(rd());
    std::uniform_int_distribution<> dis(0, 9);

    std::vector<int> v1(10), v2(10);
    std::generate(v1.begin(), v1.end(), std::bind(dis, std::ref(mt)));
    std::generate(v2.begin(), v2.end(), std::bind(dis, std::ref(mt)));

    // sort
    std::sort(v1.begin(), v1.end());
    std::sort(v2.begin(), v2.end());

    // output v1
    std::cout << "v1 : ";
    std::copy(v1.begin(), v1.end(), std::ostream_iterator<int>(std::cout, " "));
    std::cout << '\n';

    // output v2
    std::cout << "v2 : ";
    std::copy(v2.begin(), v2.end(), std::ostream_iterator<int>(std::cout, " "));
    std::cout << '\n';

    // merge
    std::vector<int> dst;
    std::merge(v1.begin(), v1.end(), v2.begin(), v2.end(), std::back_inserter(dst));

    // output
    std::cout << "dst: ";
    std::copy(dst.begin(), dst.end(), std::ostream_iterator<int>(std::cout, " "));
    std::cout << '\n';
}
```

Possible output:

```
v1 : 0 1 3 4 4 5 5 8 8 9
v2 : 0 2 2 3 6 6 8 8 8 9
dst: 0 0 1 2 2 3 3 4 4 5 5 6 6 8 8 8 8 8 9 9
```

See also

inplace_merge	merges two ordered ranges in-place (function template)
set_union	computes the union of two sets (function template)
sort	sorts a range into ascending order (function template)
stable_sort	sorts a range of elements while preserving order between equal elements (function template)
std::experimental::parallel::merge (parallelism TS)	parallelized version of `std::merge` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/merge&oldid=79893"

std::inplace_merge

Defined in header `<algorithm>`

```
template< class BidirIt >
void inplace_merge( BidirIt first, BidirIt middle, BidirIt last );
```
(1)

```
template< class BidirIt, class Compare>
void inplace_merge( BidirIt first, BidirIt middle, BidirIt last, Compare comp );
```
(2)

Merges two consecutive sorted ranges `[first, middle)` and `[middle, last)` into one sorted range `[first, last)`. The order of equal elements is guaranteed to be preserved. The first version uses `operator<` to compare the elements, the second version uses the given comparison function `comp`.

Parameters

`first` - the beginning of the first sorted range

`middle` - the end of the first sorted range and the beginning of the second

`last` - the end of the second sorted range

`comp` - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than (i.e. is ordered *before*) the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `BidirIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `BidirIt` must meet the requirements of `ValueSwappable` and `BidirectionalIterator`.
- The type of dereferenced `BidirIt` must meet the requirements of `MoveAssignable` and `MoveConstructible`.

Return value

(none)

Complexity

Exactly *N-1* comparisons if enough additional memory is available, otherwise *N·log(N)* where `N = std::distance(first, last)`.

Notes

This function attempts to allocate a temporary buffer, typically by calling `std::get_temporary_buffer`. If the allocation fails, the less efficient algorithm is chosen.

Example

The following code is an implementation of merge sort.

Run this code

```
#include <vector>
#include <iostream>
```

```
#include <algorithm>

template<class Iter>
void merge_sort(Iter first, Iter last)
{
    if (last - first > 1) {
        Iter middle = first + (last - first) / 2;
        merge_sort(first, middle);
        merge_sort(middle, last);
        std::inplace_merge(first, middle, last);
    }
}

int main()
{
    std::vector<int> v{8, 2, -2, 0, 11, 11, 1, 7, 3};
    merge_sort(v.begin(), v.end());
    for(auto n : v) {
        std::cout << n << ' ';
    }
    std::cout << '\n';
}
```

Output:

```
-2 0 1 2 3 7 8 11 11
```

See also

merge	merges two sorted ranges (function template)
sort	sorts a range into ascending order (function template)
stable_sort	sorts a range of elements while preserving order between equal elements (function template)
std::experimental::parallel::inplace_merge (parallelism TS)	parallelized version of `std::inplace_merge` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/inplace_merge&oldid=79883"

std::includes

Defined in header `<algorithm>`

```
template< class InputIt1, class InputIt2 >
bool includes( InputIt1 first1, InputIt1 last1,                    (1)
               InputIt2 first2, InputIt2 last2 );
```

```
template< class InputIt1, class InputIt2, class Compare >
bool includes( InputIt1 first1, InputIt1 last1,                    (2)
               InputIt2 first2, InputIt2 last2, Compare comp );
```

Returns `true` if every element from the sorted range `[first2, last2)` is found within the sorted range `[first1, last1)`. Also returns `true` if `[first2, last2)` is empty.

The first version expects both ranges to be sorted with `operator<`, the second version expects them to be sorted with the given comparison function `comp`.

Parameters

first1, last1 - the sorted range of elements to examine

first2, last2 - the sorted range of elements to search for

comp - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than (i.e. is ordered *before*) the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `InputIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `InputIt` must meet the requirements of `InputIterator`.

Return value

`true` if every element from `[first2, last2)` is a member of `[first, last)`.

Complexity

At most $2\cdot(N_1+N_2-1)$ comparisons, where N_1 `= std::distance(first1, last1)` and N_2 `= std::distance(first2, last2)`.

Possible implementation

First version

```
template<class InputIt1, class InputIt2>
bool includes(InputIt1 first1, InputIt1 last1,
              InputIt2 first2, InputIt2 last2)
{
    for (; first2 != last2; ++first1)
    {
        if (first1 == last1 || *first2 < *first1)
```

```
            return false;
        if ( !(*first1 < *first2) )
            ++first2;
    }
    return true;
}
```

Second version

```
template<class InputIt1, class InputIt2>
bool includes(InputIt1 first1, InputIt1 last1,
              InputIt2 first2, InputIt2 last2, Compare comp)
{
    for (; first2 != last2; ++first1)
    {
        if (first1 == last1 || comp(*first2, *first1))
            return false;
        if (!comp(*first1, *first2))
            ++first2;
    }
    return true;
}
```

Example

Run this code

```
#include <iostream>
#include <algorithm>
#include <cctype>
#include <vector>

int main()
{
  std::vector<char> v1 {'a', 'b', 'c', 'f', 'h', 'x'};
  std::vector<char> v2 {'a', 'b', 'c'};
  std::vector<char> v3 {'a', 'c'};
  std::vector<char> v4 {'g'};
  std::vector<char> v5 {'a', 'c', 'g'};

  for (auto i : v1) std::cout << i << ' ';
  std::cout << "\nincludes:\n" << std::boolalpha;

  for (auto i : v2) std::cout << i << ' ';
  std::cout << ": " << std::includes(v1.begin(), v1.end(), v2.begin(), v2.end()) << '\n';
  for (auto i : v3) std::cout << i << ' ';
  std::cout << ": " << std::includes(v1.begin(), v1.end(), v3.begin(), v3.end()) << '\n';
  for (auto i : v4) std::cout << i << ' ';
  std::cout << ": " << std::includes(v1.begin(), v1.end(), v4.begin(), v4.end()) << '\n';
  for (auto i : v5) std::cout << i << ' ';
  std::cout << ": " << std::includes(v1.begin(), v1.end(), v5.begin(), v5.end()) << '\n';

  auto cmp_nocase = [](char a, char b) {
    return std::tolower(a) < std::tolower(b);
  };

  std::vector<char> v6 {'A', 'B', 'C'};
  for (auto i : v6) std::cout << i << ' ';
  std::cout << ": (case-insensitive) "
            << std::includes(v1.begin(), v1.end(), v6.begin(), v6.end(), cmp_nocase)
            << '\n';
}
```

Output:

```
a b c f h x
includes:
a b c : true
a c : true
g : false
a c g : false
A B C : (case-insensitive) true
```

See also

`set_difference`	computes the difference between two sets (function template)
`search`	searches for a range of elements (function template)
`std::experimental::parallel::includes` (parallelism TS)	parallelized version of `std::includes` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/includes&oldid=79881"

std::set_difference

Defined in header `<algorithm>`

```
template< class InputIt1, class InputIt2, class OutputIt >
OutputIt set_difference( InputIt1 first1, InputIt1 last1,
                         InputIt2 first2, InputIt2 last2,
                         OutputIt d_first );
```

(1)

```
template< class InputIt1, class InputIt2,
          class OutputIt, class Compare >
OutputIt set_difference( InputIt1 first1, InputIt1 last1,
                         InputIt2 first2, InputIt2 last2,
                         OutputIt d_first, Compare comp );
```

(2)

Copies the elements from the sorted range `[first1, last1)` which are not found in the sorted range `[first2, last2)` to the range beginning at `d_first`.

The resulting range is also sorted. The first version expects both input ranges to be sorted with `operator<`, the second version expects them to be sorted with the given comparison function `comp`. Equivalent elements are treated individually, that is, if some element is found `m` times in `[first1, last1)` and `n` times in `[first2, last2)`, it will be copied to `d_first` exactly `std::max(m-n, 0)` times. The resulting range cannot overlap with either of the input ranges.

Parameters

`first1, last1` - the range of elements to examine

`first2, last2` - the range of elements to search for

`comp` - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than (i.e. is ordered *before*) the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that objects of types `InputIt1` and `InputIt2` can be dereferenced and then implicitly converted to both `Type1` and `Type2`.

Type requirements

- `InputIt1` must meet the requirements of `InputIterator`.
- `InputIt2` must meet the requirements of `InputIterator`.
- `OutputIt` must meet the requirements of `OutputIterator`.

Return value

Iterator past the end of the constructed range.

Complexity

At most $2\cdot(N_1+N_2-1)$ comparisons, where N_1 `= std::distance(first1, last1)` and N_2 `= std::distance(first2, last2)`.

Possible implementation

First version

```
template<class InputIt1, class InputIt2, class OutputIt>
OutputIt set_difference(InputIt1 first1, InputIt1 last1,
                        InputIt2 first2, InputIt2 last2,
                        OutputIt d_first)
{
    while (first1 != last1) {
        if (first2 == last2) return std::copy(first1, last1, d_first);

        if (*first1 < *first2) {
            *d_first++ = *first1++;
        } else {
            if (! (*first2 < *first1)) {
                ++first1;
            }
            ++first2;
        }
    }
    return d_first;
}
```

Second version

```
template<class InputIt1, class InputIt2,
         class OutputIt, class Compare>
OutputIt set_difference( InputIt1 first1, InputIt1 last1,
                         InputIt2 first2, InputIt2 last2,
                         OutputIt d_first, Compare comp)
{
    while (first1 != last1) {
        if (first2 == last2) return std::copy(first1, last1, d_first);

        if (comp(*first1, *first2)) {
            *d_first++ = *first1++;
        } else {
            if (!comp(*first2, *first1)) {
                ++first1;
            }
            ++first2;
        }
    }
    return d_first;
}
```

Example

Run this code

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>

int main() {
    std::vector<int> v1 {1, 2, 5, 5, 5, 9};
    std::vector<int> v2 {2, 5, 7};
    std::vector<int> diff;

    std::set_difference(v1.begin(), v1.end(), v2.begin(), v2.end(),
                        std::inserter(diff, diff.begin()));

    for (auto i : v1) std::cout << i << ' ';
    std::cout << "minus ";
    for (auto i : v2) std::cout << i << ' ';
    std::cout << "is: ";

    for (auto i : diff) std::cout << i << ' ';
    std::cout << '\n';
}
```

Output:

```
1 2 5 5 5 9 minus 2 5 7 is: 1 5 5 9
```

See also

includes	returns true if one set is a subset of another (function template)
set_symmetric_difference	computes the symmetric difference between two sets (function template)
std::experimental::parallel::set_difference (parallelism TS)	parallelized version of `std::set_difference` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/set_difference&oldid=79914"

std::set_intersection

Defined in header `<algorithm>`

```
template< class InputIt1, class InputIt2, class OutputIt >
OutputIt set_intersection( InputIt1 first1, InputIt1 last1,
                           InputIt2 first2, InputIt2 last2,          (1)
                           OutputIt d_first );
```

```
template< class InputIt1, class InputIt2,
          class OutputIt, class Compare >
OutputIt set_intersection( InputIt1 first1, InputIt1 last1,          (2)
                           InputIt2 first2, InputIt2 last2,
                           OutputIt d_first, Compare comp );
```

Constructs a sorted range beginning at `d_first` consisting of elements that are found in both sorted ranges `[first1, last1)` and `[first2, last2)`. The first version expects both input ranges to be sorted with `operator<`, the second version expects them to be sorted with the given comparison function `comp`. If some element is found `m` times in `[first1, last1)` and `n` times in `[first2, last2)`, the first `std::min(m, n)` elements will be copied from the first range to the destination range. The order of equivalent elements is preserved. The resulting range cannot overlap with either of the input ranges.

Parameters

`first1, last1` - the first range of elements to examine

`first2, last2` - the second range of elements to examine

`comp` - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than (i.e. is ordered *before*) the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that objects of types `InputIt1` and `InputIt2` can be dereferenced and then implicitly converted to both `Type1` and `Type2`.

Type requirements

- `InputIt1` must meet the requirements of `InputIterator`.
- `InputIt2` must meet the requirements of `InputIterator`.
- `OutputIt` must meet the requirements of `OutputIterator`.

Return value

Iterator past the end of the constructed range.

Complexity

At most $2\cdot(N_1+N_2-1)$ comparisons, where N_1 `= std::distance(first1, last1)` and N_2 `= std::distance(first2, last2)`.

Possible implementation

First version

```
template<class InputIt1, class InputIt2, class OutputIt>
OutputIt set_intersection(InputIt1 first1, InputIt1 last1,
                          InputIt2 first2, InputIt2 last2,
                          OutputIt d_first)
{
    while (first1 != last1 && first2 != last2) {
        if (*first1 < *first2) {
            ++first1;
        } else  {
            if (!(*first2 < *first1)) {
                *d_first++ = *first1++;
            }
            ++first2;
        }
    }
    return d_first;
}
```

Second version

```
template<class InputIt1, class InputIt2,
         class OutputIt, class Compare>
OutputIt set_intersection(InputIt1 first1, InputIt1 last1,
                          InputIt2 first2, InputIt2 last2,
                          OutputIt d_first, Compare comp)
{
    while (first1 != last1 && first2 != last2) {
        if (comp(*first1, *first2)) {
            ++first1;
        } else {
            if (!comp(*first2, *first1)) {
                *d_first++ = *first1++;
            }
            ++first2;
        }
    }
    return d_first;
}
```

Example

Run this code

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
int main()
{
    std::vector<int> v1{1,2,3,4,5,6,7,8};
    std::vector<int> v2{        5,  7,  9,10};
    std::sort(v1.begin(), v1.end());
    std::sort(v2.begin(), v2.end());

    std::vector<int> v_intersection;

    std::set_intersection(v1.begin(), v1.end(),
                          v2.begin(), v2.end(),
                          std::back_inserter(v_intersection));
    for(int n : v_intersection)
        std::cout << n << ' ';
}
```

Output:

```
5 7
```

See also

set_union	computes the union of two sets (function template)
std::experimental::parallel::set_intersection (parallelism TS)	parallelized version of `std::set_intersection` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/set_intersection&oldid=79915"

std::set_symmetric_difference

Defined in header `<algorithm>`

```
template< class InputIt1, class InputIt2, class OutputIt >
OutputIt set_symmetric_difference( InputIt1 first1, InputIt1 last1,
                                   InputIt2 first2, InputIt2 last2,
                                   OutputIt d_first );
```
(1)

```
template< class InputIt1, class InputIt2,
          class OutputIt, class Compare >
OutputIt set_symmetric_difference( InputIt1 first1, InputIt1 last1,
                                   InputIt2 first2, InputIt2 last2,
                                   OutputIt d_first, Compare comp );
```
(2)

Computes symmetric difference of two sorted ranges: the elements that are found in either of the ranges, but not in both of them are copied to the range beginning at `d_first`. The resulting range is also sorted.

The first version expects both input ranges to be sorted with `operator<`, the second version expects them to be sorted with the given comparison function `comp`. If some element is found `m` times in `[first1, last1)` and `n` times in `[first2, last2)`, it will be copied to `d_first` exactly `std::abs(m-n)` times. If `m>n`, then the last `m-n` of those elements are copied from `[first1,last1)`, otherwise the last `n-m` elements are copied from `[first2,last2)`. The resulting range cannot overlap with either of the input ranges.

Parameters

`first1, last1` - the first sorted range of elements

`first2, last2` - the second sorted range of elements

`comp` - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than (i.e. is ordered *before*) the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that objects of types `InputIt1` and `InputIt2` can be dereferenced and then implicitly converted to both `Type1` and `Type2`.

Type requirements

- `InputIt1` must meet the requirements of `InputIterator`.
- `InputIt2` must meet the requirements of `InputIterator`.
- `OutputIt` must meet the requirements of `OutputIterator`.

Return value

Iterator past the end of the constructed range.

Complexity

At most $2\cdot(N_1+N_2-1)$ comparisons, where N_1 `= std::distance(first1, last1)` and N_2 `= std::distance(first2, last2)`.

Possible implementation

First version

```
template<class InputIt1, class InputIt2, class OutputIt>
OutputIt set_symmetric_difference(InputIt1 first1, InputIt1 last1,
                                  InputIt2 first2, InputIt2 last2,
                                  OutputIt d_first)
{
    while (first1 != last1) {
        if (first2 == last2) return std::copy(first1, last1, d_first);

        if (*first1 < *first2) {
            *d_first++ = *first1++;
        } else {
            if (*first2 < *first1) {
                *d_first++ = *first2;
            } else {
                ++first1;
            }
            ++first2;
        }
    }
    return std::copy(first2, last2, d_first);
}
```

Second version

```
template<class InputIt1, class InputIt2,
         class OutputIt, class Compare>
OutputIt set_symmetric_difference(InputIt1 first1, InputIt1 last1,
                                  InputIt2 first2, InputIt2 last2,
                                  OutputIt d_first, Compare comp)
{
    while (first1 != last1) {
        if (first2 == last2) return std::copy(first1, last1, d_first);

        if (comp(*first1, *first2)) {
            *d_first++ = *first1++;
        } else {
            if (comp(*first2, *first1)) {
                *d_first++ = *first2;
            } else {
                ++first1;
            }
            ++first2;
        }
    }
    return std::copy(first2, last2, d_first);
}
```

Example

Run this code

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
int main()
{
    std::vector<int> v1{1,2,3,4,5,6,7,8     };
    std::vector<int> v2{        5,  7,  9,10};
    std::sort(v1.begin(), v1.end());
    std::sort(v2.begin(), v2.end());

    std::vector<int> v_symDifference;
    std::set_symmetric_difference(
        v1.begin(), v1.end(),
        v2.begin(), v2.end(),
        std::back_inserter(v_symDifference));

    for(int n : v_symDifference)
        std::cout << n << ' ';
}
```

Output:

```
1 2 3 4 6 8 9 10
```

See also

includes	returns true if one set is a subset of another (function template)
set_difference	computes the difference between two sets (function template)
set_union	computes the union of two sets (function template)
set_intersection	computes the intersection of two sets (function template)
std::experimental::parallel::set_symmetric_difference (parallelism TS)	parallelized version of `std::set_symmetric_difference` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/set_symmetric_difference&oldid=79916"

std::set_union

Defined in header `<algorithm>`

```
template< class InputIt1, class InputIt2, class OutputIt >
OutputIt set_union( InputIt1 first1, InputIt1 last1,
                    InputIt2 first2, InputIt2 last2,
                    OutputIt d_first );                          (1)
```

```
template< class InputIt1, class InputIt2,
          class OutputIt, class Compare >
OutputIt set_union( InputIt1 first1, InputIt1 last1,             (2)
                    InputIt2 first2, InputIt2 last2,
                    OutputIt d_first, Compare comp );
```

Constructs a sorted range beginning at `d_first` consisting of all elements present in one or both sorted ranges `[first1, last1)` and `[first2, last2)`.

1) Expects both input ranges to be sorted with `operator<`

2) Expects them to be sorted with the given comparison function `comp`

If some element is found `m` times in `[first1, last1)` and `n` times in `[first2, last2)`, then all `m` elements will be copied from `[first1, last1)` to `d_first`, preserving order, and then exactly `std::max(n-m, 0)` elements will be copied from `[first2, last2)` to `d_first`, also preserving order.

The resulting range cannot overlap with either of the input ranges.

Parameters

first1, last1 - the first input sorted range

first2, last2 - the second input sorted range

comp - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than (i.e. is ordered *before*) the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that objects of types `InputIt1` and `InputIt2` can be dereferenced and then implicitly converted to both `Type1` and `Type2`.

Type requirements

- `InputIt1` must meet the requirements of `InputIterator`.
- `InputIt2` must meet the requirements of `InputIterator`.
- `OutputIt` must meet the requirements of `OutputIterator`.

Return value

Iterator past the end of the constructed range.

Complexity

At most $2\cdot(N_1+N_2-1)$ comparisons, where N_1 `= std::distance(first1, last1)` and N_2 `= std::distance(first2, last2)`.

Notes

This algorithm performs a similar task as `std::merge` does. Both consume two sorted input ranges and produce a sorted output with elements from both inputs. The difference bewteen these two algorithms is with handling values from both input ranges which compare equivalent (see notes on `LessThanComparable`). If any equivalent values appeared `n` times in the first range and `m` times in the second, `std::merge` would output all `n+m` occurrences whereas `std::set_union` would output `std::max(n, m)` ones only. So `std::merge` outputs exactly `std::distance(first1, last1) + std::distance(first2, last2)` values and `std::set_union` may produce less.

Possible implementation

First version

```
template<class InputIt1, class InputIt2, class OutputIt>
OutputIt set_union(InputIt1 first1, InputIt1 last1,
                   InputIt2 first2, InputIt2 last2,
                   OutputIt d_first)
{
    for (; first1 != last1; ++d_first) {
        if (first2 == last2)
            return std::copy(first1, last1, d_first);
        if (*first2 < *first1) {
            *d_first = *first2++;
        } else {
            *d_first = *first1;
            if (!(*first1 < *first2))
                ++first2;
            ++first1;
        }
    }
    return std::copy(first2, last2, d_first);
}
```

Second version

```
template<class InputIt1, class InputIt2,
         class OutputIt, class Compare>
OutputIt set_union(InputIt1 first1, InputIt1 last1,
                   InputIt2 first2, InputIt2 last2,
                   OutputIt d_first, Compare comp)
{
    for (; first1 != last1; ++d_first) {
        if (first2 == last2)
            return std::copy(first1, last1, d_first);
        if (comp(*first2, *first1)) {
            *d_first = *first2++;
        } else {
            *d_first = *first1;
            if (!comp(*first1, *first2))
                ++first2;
            ++first1;
        }
    }
    return std::copy(first2, last2, d_first);
}
```

Example

Example with vectors :

Run this code

```
#include <vector>
#include <set>
#include <iostream>
#include <algorithm>
#include <iterator>

int main()
{
    std::vector<int> v1 = {1, 2, 3, 4, 5};
    std::vector<int> v2 = {      3, 4, 5, 6, 7};
    std::vector<int> dest1;

    std::set_union(v1.begin(), v1.end(),
                   v2.begin(), v2.end(),
                   std::back_inserter(dest1));

    for (const auto &i : dest1) {
        std::cout << i << ' ';
    }
    std::cout << '\n';
}
```

Output:

```
1 2 3 4 5 6 7
```

See also

includes	returns true if one set is a subset of another (function template)
merge	merges two sorted ranges (function template)
set_difference	computes the difference between two sets (function template)
set_intersection	computes the intersection of two sets (function template)
set_symmetric_difference	computes the symmetric difference between two sets (function template)
std::experimental::parallel::set_union (parallelism TS)	parallelized version of `std::set_union` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/set_union&oldid=79917"

std::is_heap

Defined in header `<algorithm>`

```
template< class RandomIt >
bool is_heap( RandomIt first, RandomIt last );
```

(1) (since C++11)

```
template< class RandomIt, class Compare >
bool is_heap( RandomIt first, RandomIt last, Compare comp );
```

(2) (since C++11)

Checks if the elements in range `[first, last)` are a *max heap*.

The first version of `is_heap` uses `operator<` to compare elements, whereas the second uses the given comparison function `comp`.

Parameters

first, last - the range of elements to examine

comp - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `RandomIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `RandomIt` must meet the requirements of `RandomAccessIterator`.

Return value

`true` if the range is *max heap*, `false` otherwise.

Complexity

Linear in the distance between `first` and `last`

Notes

A *max heap* is a range of elements `[f,l)` that has the following properties:

- `*f` is the largest element in the range
- a new element can be added using `std::push_heap()`
- the first element can be removed using `std::pop_heap()`

The actual arrangement of the elements is implementation defined.

Example

Run this code

```
#include <iostream>
#include <algorithm>
#include <vector>
```

```
int main()
{
    std::vector<int> v { 3, 1, 4, 1, 5, 9 };

    std::cout << "initially, v: ";
    for (auto i : v) std::cout << i << ' ';
    std::cout << '\n';

    if (!std::is_heap(v.begin(), v.end())) {
        std::cout << "making heap...\n";
        std::make_heap(v.begin(), v.end());
    }

    std::cout << "after make_heap, v: ";
    for (auto i : v) std::cout << i << ' ';
    std::cout << '\n';
}
```

Output:

```
initially, v: 3 1 4 1 5 9
making heap...
after make_heap, v: 9 5 4 1 1 3
```

See also

is_heap_until (C++11)	finds the largest subrange that is a max heap (function template)
std::experimental::parallel::is_heap (parallelism TS)	parallelized version of `std::is_heap` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/is_heap&oldid=79884"

std::is_heap_until

Defined in header `<algorithm>`

`template< class RandomIt >`
`RandomIt is_heap_until( RandomIt first, RandomIt last );`
	(1)	(since C++11)

`template< class RandomIt, class Compare >`
`RandomIt is_heap_until( RandomIt first, RandomIt last, Compare comp );`
	(2)	(since C++11)

Examines the range `[first, last)` and finds the largest range beginning at `first` which is a *max heap*. The first version of the function uses `operator<` to compare the elements, the second uses the given comparison function `comp`.

Parameters

`first, last` - the range of elements to examine

`comp` - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `RandomIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `RandomIt` must meet the requirements of `RandomAccessIterator`.

Return value

The upper bound of the largest range beginning at `first` which is a *max heap*. That is, the last iterator `it` for which range `[first, it)` is a *max heap*.

Complexity

Linear in the distance between `first` and `last`

Notes

A *max heap* is a range of elements `[f,l)` that has the following properties:

- `*f` is the largest element in the range
- a new element can be added using `std::push_heap()`
- the first element can be removed using `std::pop_heap()`

The actual arrangement of the elements is implementation defined.

Example

Run this code

```
#include <iostream>
#include <algorithm>
#include <vector>
```

```
int main()
{
    std::vector<int> v { 3, 1, 4, 1, 5, 9 };

    std::make_heap(v.begin(), v.end());

    // probably mess up the heap
    v.push_back(2);
    v.push_back(6);

    auto heap_end = std::is_heap_until(v.begin(), v.end());

    std::cout << "all of v: ";
    for (auto i : v) std::cout << i << ' ';
    std::cout << '\n';

    std::cout << "only heap: ";
    for (auto i = v.begin(); i != heap_end; ++i) std::cout << *i << ' ';
    std::cout << '\n';
}
```

Output:

```
all of v:  9 5 4 1 1 3 2 6
only heap: 9 5 4 1 1 3 2
```

See also

`is_heap` (C++11)	checks if the given range is a max heap (function template)
`std::experimental::parallel::is_heap_until` (parallelism TS)	parallelized version of `std::is_heap_until` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/is_heap_until&oldid=79888"

std::make_heap

Defined in header `<algorithm>`

```
template< class RandomIt >
void make_heap( RandomIt first, RandomIt last );                    (1)
```

```
template< class RandomIt, class Compare >
void make_heap( RandomIt first, RandomIt last,                      (2)
                Compare comp );
```

Constructs a *max heap* in the range `[first, last)`. The first version of the function uses `operator<` to compare the elements, the second uses the given comparison function `comp`.

Parameters

`first, last` - the range of elements to make the heap from

`comp` - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `RandomIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `RandomIt` must meet the requirements of `RandomAccessIterator`.
- The type of dereferenced `RandomIt` must meet the requirements of `MoveAssignable` and `MoveConstructible`.

Return value

(none)

Complexity

At most `3*std::distance(first, last)` comparisons.

Notes

A *max heap* is a range of elements `[f,l)` that has the following properties:

- `*f` is the largest element in the range
- a new element can be added using `std::push_heap()`
- the first element can be removed using `std::pop_heap()`

The actual arrangement of the elements is implementation defined.

Example

Run this code

```
#include <iostream>
#include <algorithm>
```

```
#include <vector>

int main()
{
    std::vector<int> v { 3, 1, 4, 1, 5, 9 };

    std::cout << "initially, v: ";
    for (auto i : v) std::cout << i << ' ';
    std::cout << '\n';

    std::make_heap(v.begin(), v.end());

    std::cout << "after make_heap, v: ";
    for (auto i : v) std::cout << i << ' ';
    std::cout << '\n';

    std::pop_heap(v.begin(), v.end());
    auto largest = v.back();
    v.pop_back();
    std::cout << "largest element: " << largest << '\n';

    std::cout << "after removing the largest element, v: ";
    for (auto i : v) std::cout << i << ' ';
    std::cout << '\n';
}
```

Output:

```
initially, v: 3 1 4 1 5 9
after make_heap, v: 9 5 4 1 1 3
largest element: 9
after removing the largest element, v: 5 3 4 1 1
```

See also

sort_heap	turns a max heap into a range of elements sorted in ascending order (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/make_heap&oldid=69143"

std::push_heap

Defined in header `<algorithm>`

```
template< class RandomIt >
void push_heap( RandomIt first, RandomIt last );
```
(1)

```
template< class RandomIt, class Compare >
void push_heap( RandomIt first, RandomIt last,
                Compare comp );
```
(2)

Inserts the element at the position `last-1` into the *max heap* defined by the range `[first, last-1)`. The first version of the function uses `operator<` to compare the elements, the second uses the given comparison function `comp`.

Parameters

first, last - the range of elements defining the heap to modify

comp - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `RandomIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `RandomIt` must meet the requirements of `RandomAccessIterator`.
- The type of dereferenced `RandomIt` must meet the requirements of `MoveAssignable` and `MoveConstructible`.

Return value

(none)

Complexity

At most *2×log(N)* comparisons where `N=std::distance(first, last)`.

Notes

A *max heap* is a range of elements `[f,l)` that has the following properties:

- `*f` is the largest element in the range
- a new element can be added using `std::push_heap()`
- the first element can be removed using `std::pop_heap()`

The actual arrangement of the elements is implementation defined.

Example

Run this code

```
#include <iostream>
```

```
#include <algorithm>
#include <vector>

int main()
{
    std::vector<int> v { 3, 1, 4, 1, 5, 9 };

    std::make_heap(v.begin(), v.end());

    std::cout << "v: ";
    for (auto i : v) std::cout << i << ' ';
    std::cout << '\n';

    v.push_back(6);

    std::cout << "before push_heap: ";
    for (auto i : v) std::cout << i << ' ';
    std::cout << '\n';

    std::push_heap(v.begin(), v.end());

    std::cout << "after push_heap: ";
    for (auto i : v) std::cout << i << ' ';
    std::cout << '\n';
}
```

Output:

```
v: 9 5 4 1 1 3
before push_heap: 9 5 4 1 1 3 6
after push heap:  9 5 6 1 1 3 4
```

See also

pop_heap	removes the largest element from a max heap (function template)
make_heap	creates a max heap out of a range of elements (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/push_heap&oldid=77343"

std::pop_heap

Defined in header `<algorithm>`

```
template< class RandomIt >
void pop_heap( RandomIt first, RandomIt last );
```
(1)

```
template< class RandomIt, class Compare >
void pop_heap( RandomIt first, RandomIt last, Compare comp );
```
(2)

Swaps the value in the position `first` and the value in the position `last-1` and makes the subrange `[first, last-1)` into a *max heap*. This has the effect of removing the first (largest) element from the heap defined by the range `[first, last)`.

The first version of the function uses `operator<` to compare the elements, the second uses the given comparison function `comp`.

Parameters

`first, last` - the range of elements defining the valid nonempty heap to modify

`comp` - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `RandomIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `RandomIt` must meet the requirements of `ValueSwappable` and `RandomAccessIterator`.
- The type of dereferenced `RandomIt` must meet the requirements of `MoveAssignable` and `MoveConstructible`.

Return value

(none)

Complexity

At most *2×log(N)* comparisons where `N=std::distance(first, last)`.

Notes

A *max heap* is a range of elements `[f,l)` that has the following properties:

- `*f` is the largest element in the range
- a new element can be added using `std::push_heap()`
- the first element can be removed using `std::pop_heap()`

The actual arrangement of the elements is implementation defined.

Example

Run this code

```
#include <iostream>
#include <algorithm>
#include <vector>

int main()
{
    std::vector<int> v { 3, 1, 4, 1, 5, 9 };

    std::make_heap(v.begin(), v.end());

    std::cout << "v: ";
    for (auto i : v) std::cout << i << ' ';
    std::cout << '\n';

    std::pop_heap(v.begin(), v.end()); // moves the largest to the end

    std::cout << "after pop_heap: ";
    for (auto i : v) std::cout << i << ' ';
    std::cout << '\n';

    int largest = v.back();
    v.pop_back();  // actually removes the largest element
    std::cout << "largest element: " << largest << '\n';

    std::cout << "heap without largest: ";
    for (auto i : v) std::cout << i << ' ';
    std::cout << '\n';
}
```

Output:

```
v: 9 5 4 1 1 3
after pop_heap: 5 3 4 1 1 9
largest element: 9
heap without largest: 5 3 4 1 1
```

See also

push_heap	adds an element to a max heap (function template)
make_heap	creates a max heap out of a range of elements (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/pop_heap&oldid=77342"

std::sort_heap

Defined in header `<algorithm>`

`template< class RandomIt >`
`void sort_heap( RandomIt first, RandomIt last );`
	(1)

`template< class RandomIt, class Compare >`
`void sort_heap( RandomIt first, RandomIt last, Compare comp );`
	(2)

Converts the *max heap* `[first, last)` into a sorted range in ascending order. The resulting range no longer has the heap property.

The first version of the function uses `operator<` to compare the elements, the second uses the given comparison function `comp`.

Parameters

`first, last` - the range of elements to sort

`comp` - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `RandomIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `RandomIt` must meet the requirements of `ValueSwappable` and `RandomAccessIterator`.
- The type of dereferenced `RandomIt` must meet the requirements of `MoveAssignable` and `MoveConstructible`.

Return value

(none)

Complexity

At most *N×log(N)* comparisons where `N=std::distance(first, last)`.

Notes

A *max heap* is a range of elements `[f,l)` that has the following properties:

- `*f` is the largest element in the range
- a new element can be added using `std::push_heap()`
- the first element can be removed using `std::pop_heap()`

The actual arrangement of the elements is implementation defined.

Possible implementation

First version

```
template< class RandomIt >
void sort_heap( RandomIt first, RandomIt last );
{
    while (first != last)
        std::pop_heap(first, last--);
}
```

Second version

```
template< class RandomIt, class Compare >
void sort_heap( RandomIt first, RandomIt last, Compare comp );
{
    while (first != last)
        std::pop_heap(first, last--, comp);
}
```

Example

Run this code

```
#include <algorithm>
#include <vector>
#include <iostream>

int main()
{
    std::vector<int> v = {3, 1, 4, 1, 5, 9};

    std::make_heap(v.begin(), v.end());

    std::cout << "heap:\t";
    for (const auto &i : v) {
        std::cout << i << ' ';
    }

    std::sort_heap(v.begin(), v.end());

    std::cout << "\nsorted:\t";
    for (const auto &i : v) {
        std::cout << i << ' ';
    }
    std::cout << '\n';
}
```

Output:

```
heap:   9 4 5 1 1 3
sorted: 1 1 3 4 5 9
```

See also

make_heap	creates a max heap out of a range of elements (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/sort_heap&oldid=65850"

std::max

Defined in header `<algorithm>`

`template< class T >`
`const T& max( const T& a, const T& b );`
	(1)	(until C++14)

`template< class T >`
`constexpr const T& max( const T& a, const T& b );`
		(since C++14)

`template< class T, class Compare >`
`const T& max( const T& a, const T& b, Compare comp );`
	(2)	(until C++14)

`template< class T, class Compare >`
`constexpr const T& max( const T& a, const T& b, Compare comp );`
		(since C++14)

`template< class T >`
`T max( std::initializer_list<T> ilist );`
	(3)	(since C++11)
		(until C++14)

`template< class T >`
`constexpr T max( std::initializer_list<T> ilist );`
		(since C++14)

`template< class T, class Compare >`
`T max( std::initializer_list<T> ilist, Compare comp );`
	(4)	(since C++11)
		(until C++14)

`template< class T, class Compare >`
`constexpr T max( std::initializer_list<T> ilist, Compare comp );`
		(since C++14)

Returns the greater of the given values.

1-2) Returns the greater of `a` and `b`.

3-4) Returns the greatest of the values in initializer list `ilist`.

The (1,3) versions use `operator<` to compare the values, the (2,4) versions use the given comparison function `comp`.

Parameters

a, b	-	the values to compare
ilist	-	initializer list with the values to compare
comp	-	comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if if `a` is *less* than `b`.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `T` can be implicitly converted to both of them.

Type requirements

- `T` must meet the requirements of `LessThanComparable` in order to use overloads (1,3).
- `T` must meet the requirements of `CopyConstructible` in order to use overloads (3,4).

Return value

1-2) The greater of `a` and `b`. If they are equivalent, returns `a`.

3-4) The greatest value in `ilist`. If several values are equivalent to the greatest, returns the leftmost one.

Complexity

1-2) Exactly one comparison

3-4) Exactly `ilist.size() - 1` comparisons

Possible implementation

First version

```
template<class T>
const T& max(const T& a, const T& b)
{
    return (a < b) ? b : a;
}
```

Second version

```
template<class T, class Compare>
const T& max(const T& a, const T& b, Compare comp)
{
    return (comp(a, b)) ? b : a;
}
```

Third version

```
template< class T >
T max( std::initializer_list<T> ilist)
{
    return *std::max_element(ilist.begin(), ilist.end());
}
```

Fourth version

```
template< class T, class Compare >
T max( std::initializer_list<T> ilist, Compare comp )
{
    return *std::max_element(ilist.begin(), ilist.end(), comp);
}
```

Example

Run this code

```
#include <algorithm>
#include <iostream>
#include <string>

int main()
{
    std::cout << "larger of 1 and 9999: " << std::max(1, 9999) << '\n'
              << "larger of 'a', and 'b': " << std::max('a', 'b') << '\n'
              << "longest of \"foo\", \"bar\", and \"hello\": " <<
                  std::max( { "foo", "bar", "hello" },
                            [](const std::string& s1, const std::string& s2) {
                                 return s1.size() < s2.size();
                             }) << '\n';
}
```

Output:

```
larger of 1 and 9999: 9999
larger of 'a', and 'b': b
longest of "foo", "bar", and "hello": hello
```

See also

min	returns the smaller of two elements (function template)
minmax (C++11)	returns the larger and the smaller of two elements (function template)
max_element	returns the largest element in a range (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/max&oldid=78448"

std::max_element

Defined in header `<algorithm>`

`template< class ForwardIt >`
`ForwardIt max_element(ForwardIt first, ForwardIt last);`
	(1)	(until C++17)

`template< class ForwardIt >`
`constexpr ForwardIt max_element(ForwardIt first, ForwardIt last);`
		(since C++17)

`template< class ForwardIt, class Compare >`
`ForwardIt max_element(ForwardIt first, ForwardIt last, Compare cmp);`
	(2)	(until C++17)

`template< class ForwardIt, class Compare >`
`constexpr ForwardIt max_element(ForwardIt first, ForwardIt last, Compare cmp);`
		(since C++17)

Finds the greatest element in the range `[first, last)`. The first version uses `operator<` to compare the values, the second version uses the given comparison function `cmp`.

Parameters

first, last - forward iterators defining the range to examine

cmp - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `ForwardIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `ForwardIt` must meet the requirements of `ForwardIterator`.

Return value

Iterator to the greatest element in the range `[first, last)`. If several elements in the range are equivalent to the greatest element, returns the iterator to the first such element. Returns `last` if the range is empty.

Complexity

Exactly *max(N-1,0)* comparisons, where `N = std::distance(first, last)`.

Possible implementation

First version

```
template<class ForwardIt>
ForwardIt max_element(ForwardIt first, ForwardIt last)
{
    if (first == last) {
        return last;
    }
    ForwardIt largest = first;
    ++first;
    for (; first != last; ++first) {
        if (*largest < *first) {
            largest = first;
        }
    }
    return largest;
}
```

Second version

```
template<class ForwardIt, class Compare>
ForwardIt max_element(ForwardIt first, ForwardIt last,
                      Compare cmp)
{
    if (first == last) {
        return last;
    }
    ForwardIt largest = first;
    ++first;
    for (; first != last; ++first) {
        if (cmp(*largest, *first)) {
            largest = first;
        }
    }
    return largest;
}
```

Example

Run this code

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <cmath>

static bool abs_compare(int a, int b)
{
    return (std::abs(a) < std::abs(b));
}

int main()
{
    std::vector<int> v{ 3, 1, -14, 1, 5, 9 };
    std::vector<int>::iterator result;

    result = std::max_element(v.begin(), v.end());
    std::cout << "max element at: " << std::distance(v.begin(), result) << '\n';

    result = std::max_element(v.begin(), v.end(), abs_compare);
    std::cout << "max element (absolute) at: " << std::distance(v.begin(), result);
}
```

Output:

```
max element at: 5
max element (absolute) at: 2
```

See also

min_element	returns the smallest element in a range (function template)

`minmax_element` (C++11)	returns the smallest and the largest element in a range (function template)
`max`	returns the larger of two elements (function template)
`std::experimental::parallel::max_element` (parallelism TS)	parallelized version of `std::max_element` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/max_element&oldid=79892"

std::min

Defined in header `<algorithm>`

`template< class T >`
`const T& min( const T& a, const T& b );`
	(1)	(until C++14)

`template< class T >`
`constexpr const T& min( const T& a, const T& b );`
		(since C++14)

`template< class T, class Compare >`
`const T& min( const T& a, const T& b, Compare comp );`
	(2)	(until C++14)

`template< class T, class Compare >`
`constexpr const T& min( const T& a, const T& b, Compare comp );`
		(since C++14)

`template< class T >`
`T min( std::initializer_list<T> ilist );`
	(3)	(since C++11)
		(until C++14)

`template< class T >`
`constexpr T min( std::initializer_list<T> ilist );`
		(since C++14)

`template< class T, class Compare >`
`T min( std::initializer_list<T> ilist, Compare comp );`
	(4)	(since C++11)
		(until C++14)

`template< class T, class Compare >`
`constexpr T min( std::initializer_list<T> ilist, Compare comp );`
		(since C++14)

Returns the smaller of the given values.

1-2) Returns the smaller of `a` and `b`.

3-4) Returns the smallest of the values in initializer list `ilist`.

The (1,3) versions use `operator<` to compare the values, the (2,4) versions use the given comparison function `comp`.

Parameters

a, b - the values to compare

ilist - initializer list with the values to compare

cmp - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if if `a` is *less* than `b`.

The signature of the comparison function should be equivalent to the following:

```
 bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `T` can be implicitly converted to both of them.

Type requirements

- `T` must meet the requirements of `LessThanComparable` in order to use overloads (1,3).
- `T` must meet the requirements of `CopyConstructible` in order to use overloads (3,4).

Return value

1-2) The smaller of `a` and `b`. If the values are equivalent, returns `a`.

3-4) The smallest value in `ilist`. If several values are equivalent to the smallest, returns the leftmost such value.

Complexity

1-2) Exactly one comparison

3-4) Exactly `ilist.size() - 1` comparisons

Possible implementation

First version

```
template<class T>
const T& min(const T& a, const T& b)
{
    return (b < a) ? b : a;
}
```

Second version

```
template<class T, class Compare>
const T& min(const T& a, const T& b, Compare comp)
{
    return (comp(b, a)) ? b : a;
}
```

Third version

```
template<class T>
T min( std::initializer_list<T> ilist)
{
    return *std::min_element(ilist.begin(), ilist.end());
}
```

Fourth version

```
template<class T, class Compare>
T min(std::initializer_list<T> ilist, Compare comp)
{
    return *std::min_element(ilist.begin(), ilist.end(), comp);
}
```

Example

Run this code

```
#include <algorithm>
#include <iostream>
#include <string>

int main()
{
    std::cout << "smaller of 1 and 9999: " << std::min(1, 9999) << '\n'
              << "smaller of 'a', and 'b': " << std::min('a', 'b') << '\n'
              << "shortest of \"foo\", \"bar\", and \"hello\": " <<
                  std::min( { "foo", "bar", "hello" },
                            [](const std::string& s1, const std::string& s2) {
                                 return s1.size() < s2.size();
                             }) << '\n';
}
```

Output:

```
smaller of 1 and 9999: 1
smaller of 'a', and 'b': a
shortest of "foo", "bar", and "hello": foo
```

See also

max	returns the larger of two elements (function template)
minmax (C++11)	returns the larger and the smaller of two elements (function template)
min_element	returns the smallest element in a range (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/min&oldid=78449"

std::min_element

Defined in header `<algorithm>`

`template< class ForwardIt >`
`ForwardIt min_element( ForwardIt first, ForwardIt last );`
	(1)	(until C++17)

`template< class ForwardIt >`
`constexpr ForwardIt min_element( ForwardIt first, ForwardIt last );`
		(since C++17)

`template< class ForwardIt, class Compare >`
`ForwardIt min_element( ForwardIt first, ForwardIt last, Compare comp );`
	(2)	(until C++17)

`template< class ForwardIt, class Compare >`
`constexpr ForwardIt min_element( ForwardIt first, ForwardIt last, Compare comp );`
		(since C++17)

Finds the smallest element in the range `[first, last)`. The first version uses `operator<` to compare the values, the second version uses the given comparison function `comp`.

Parameters

first, last - forward iterators defining the range to examine

cmp - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if `a` is *less* than `b`.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `ForwardIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `ForwardIt` must meet the requirements of `ForwardIterator`.

Return value

Iterator to the smallest element in the range `[first, last)`. If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns `last` if the range is empty.

Complexity

Exactly *max(N-1,0)* comparisons, where `N = std::distance(first, last)`.

Possible implementation

First version

```
template<class ForwardIt>
ForwardIt min_element(ForwardIt first, ForwardIt last)
{
    if (first == last) return last;

    ForwardIt smallest = first;
    ++first;
    for (; first != last; ++first) {
        if (*first < *smallest) {
            smallest = first;
        }
    }
    return smallest;
}
```

Second version

```
template<class ForwardIt, class Compare>
ForwardIt min_element(ForwardIt first, ForwardIt last,
                            Compare comp)
{
    if (first == last) return last;

    ForwardIt smallest = first;
    ++first;
    for (; first != last; ++first) {
        if (comp(*first, *smallest)) {
            smallest = first;
        }
    }
    return smallest;
}
```

Example

Run this code

```
#include <algorithm>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v{3, 1, 4, 1, 5, 9};

    std::vector<int>::iterator result = std::min_element(std::begin(v), std::end(v));
    std::cout << "min element at: " << std::distance(std::begin(v), result);
}
```

Output:

```
min element at: 1
```

See also

max_element	returns the largest element in a range (function template)
minmax_element (C++11)	returns the smallest and the largest element in a range (function template)
min	returns the smaller of two elements (function template)
std::experimental::parallel::min_element (parallelism TS)	parallelized version of `std::min_element` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/min_element&oldid=79894"

std::minmax

Defined in header <algorithm>

```
template< class T >
std::pair<const T&,const T&> minmax( const T& a, const T& b );
```

(1) (since C++11) (until C++14)

```
template< class T >
constexpr std::pair<const T&,const T&> minmax( const T& a, const T& b );
```

(since C++14)

```
template< class T, class Compare >
std::pair<const T&,const T&> minmax( const T& a, const T& b,
                                     Compare comp );
```

(2) (since C++11) (until C++14)

```
template< class T, class Compare >
constexpr std::pair<const T&,const T&> minmax( const T& a, const T& b,
                                               Compare comp );
```

(since C++14)

```
template< class T >
std::pair<T,T> minmax( std::initializer_list<T> ilist);
```

(3) (since C++11) (until C++14)

```
template< class T >
constexpr std::pair<T,T> minmax( std::initializer_list<T> ilist);
```

(since C++14)

```
template< class T, class Compare >
std::pair<T,T> minmax( std::initializer_list<T> ilist, Compare comp );
```

(4) (since C++11) (until C++14)

```
template< class T, class Compare >
constexpr std::pair<T,T> minmax( std::initializer_list<T> ilist, Compare comp );
```

(since C++14)

Returns the lowest and the greatest of the given values.

1-2) Returns the smaller and the greater of a and b.

3-4) Returns the smallest and the greatest of the values in initializer list ilist.

The (1,3) versions use `operator<` to compare the values, whereas the (2,4) versions use the given comparison function comp.

Parameters

a, b	-	the values to compare
ilist	-	initializer list with the values to compare
comp	-	comparison function object (i.e. an object that satisfies the requirements of Compare) which returns `true` if the first argument is *less* than the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `T` can be implicitly converted to both of them.

Type requirements

- T must meet the requirements of LessThanComparable in order to use overloads (1,3).
- T must meet the requirements of CopyConstructible in order to use overloads (3,4).

Return value

1-2) Returns the result of `std::pair<const T&, const T&>(a, b)` if a<b or if a is equivalent to b. Returns the result of `std::pair<const T&, const T&>(b, a)` if b<a.

3-4) A pair with the smallest value in `ilist` as the first element and the greatest as the second. If several elements are equivalent to the smallest, the leftmost such element is returned. If several elements are equivalent to the largest, the rightmost such element is returned.

Complexity

1-2) Exactly one comparison

3-4) At most `ilist.size() * 3 / 2` comparisons

Possible implementation

First version

```
template<class T>
std::pair<const T&, const T&> minmax( const T& a, const T& b )
{
    return (b < a) ? std::pair<const T&, const T&>(b, a)
                   : std::pair<const T&, const T&>(a, b);
}
```

Second version

```
template<class T, class Compare>
std::pair<const T&, const T&> minmax( const T& a, const T& b, Compare comp )
{
    return comp(b, a) ? std::pair<const T&, const T&>(b, a)
                      : std::pair<const T&, const T&>(a, b);
}
```

Third version

```
template< class T >
std::pair<T, T> minmax( std::initializer_list<T> ilist )
{
    auto p = std::minmax_element(ilist.begin(), ilist.end());
    return std::make_pair(*p.first, *p.second);
}
```

Fourth version

```
template< class T, class Compare >
std::pair<T, T> minmax( std::initializer_list<T> ilist, Compare comp )
{
    auto p = std::minmax_element(ilist.begin(), ilist.end(), comp);
    return std::make_pair(*p.first, *p.second);
}
```

Example

Run this code

```
#include <algorithm>
```

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>

int main()
{
    std::vector<int> v {3, 1, 4, 1, 5, 9, 2, 6};
    std::srand(std::time(0));
    std::pair<int, int> bounds = std::minmax(std::rand() % v.size(),
                                             std::rand() % v.size());

    std::cout << "v[" << bounds.first << "," << bounds.second << "]: ";
    for (int i = bounds.first; i < bounds.second; ++i) {
        std::cout << v[i] << ' ';
    }
    std::cout << '\n';
}
```

Possible output:

```
v[2,7]: 4 1 5 9 2
```

See also

min	returns the smaller of two elements (function template)
max	returns the larger of two elements (function template)
minmax_element (C++11)	returns the smallest and the largest element in a range (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/minmax&oldid=78450"

std::minmax_element

Defined in header `<algorithm>`

`template< class ForwardIt >`
`std::pair<ForwardIt,ForwardIt>`
`    minmax_element( ForwardIt first, ForwardIt last );`
	(1)	(since C++11)
		(until C++17)

`template< class ForwardIt >`
`constexpr std::pair<ForwardIt,ForwardIt>`
`    minmax_element( ForwardIt first, ForwardIt last );`
		(since C++17)

`template< class ForwardIt, class Compare >`
`std::pair<ForwardIt,ForwardIt>`
`    minmax_element( ForwardIt first, ForwardIt last, Compare comp );`
	(2)	(since C++11)
		(until C++17)

`template< class ForwardIt, class Compare >`
`constexpr std::pair<ForwardIt,ForwardIt>`
`    minmax_element( ForwardIt first, ForwardIt last, Compare comp );`
		(since C++17)

Finds the greatest and the smallest element in the range `[first, last)`. The first version uses `operator<` to compare the values, the second version uses the given comparison function `comp`.

Parameters

first, last - forward iterators defining the range to examine

cmp - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if if `*a` is *less* than `*b`.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `ForwardIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `ForwardIt` must meet the requirements of `ForwardIterator`.

Return value

a pair consisting of an iterator to the smallest element as the first element and an iterator to the greatest element as the second. Returns `std::make_pair(first, first)` if the range is empty. If several elements are equivalent to the smallest element, the iterator to the first such element is returned. If several elements are equivalent to the largest element, the iterator to the last such element is returned.

Complexity

At most $max(floor(3/2(N-1)), 0)$ applications of the predicate, where `N = std::distance(first, last)`
.

Notes

This algorithm is different from `std::make_pair(std::min_element(), std::max_element())`, not only in efficiency, but also in that this algorithm finds the *last* biggest element while `std::max_element` finds the *first* biggest element.

Possible implementation

First version

```
template<class ForwardIt>
std::pair<ForwardIt, ForwardIt>
    minmax_element(ForwardIt first, ForwardIt last)
{
    return std::minmax_element(first, last, std::less<>());
}
```

Second version

```
template<class ForwardIt, class Compare>
std::pair<ForwardIt, ForwardIt>
    minmax_element(ForwardIt first, ForwardIt last, Compare comp)
{
    std::pair<ForwardIt, ForwardIt> result(first, first);

    if (first == last) return result;
    if (++first == last) return result;

    if (comp(*first, *result.first)) {
        result.first = first;
    } else {
        result.second = first;
    }
    while (++first != last) {
        ForwardIt i = first;
        if (++first == last) {
            if (comp(*i, *result.first)) result.first = i;
            else if (!(comp(*i, *result.second))) result.second = i;
            break;
        } else {
            if (comp(*first, *i)) {
                if (comp(*first, *result.first)) result.first = first;
                if (!(comp(*i, *result.second))) result.second = i;
            } else {
                if (comp(*i, *result.first)) result.first = i;
                if (!(comp(*first, *result.second))) result.second = first;
            }
        }
    }
    return result;
}
```

Example

Run this code

```
#include <algorithm>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v = { 3, 9, 1, 4, 2, 5, 9 };

    auto result = std::minmax_element(v.begin(), v.end());
    std::cout << "min element at: " << (result.first - v.begin()) << '\n';
    std::cout << "max element at: " << (result.second - v.begin()) << '\n';
}
```

Output:

```
min element at: 2
max element at: 6
```

See also

`min_element`	returns the smallest element in a range (function template)
`max_element`	returns the largest element in a range (function template)
`std::experimental::parallel::minmax_element` (parallelism TS)	parallelized version of `std::minmax_element` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/minmax_element&oldid=79895"

std::lexicographical_compare

Defined in header `<algorithm>`

```
template< class InputIt1, class InputIt2 >
bool lexicographical_compare( InputIt1 first1, InputIt1 last1,         (1)
                              InputIt2 first2, InputIt2 last2 );
```

```
template< class InputIt1, class InputIt2, class Compare >
bool lexicographical_compare( InputIt1 first1, InputIt1 last1,         (2)
                              InputIt2 first2, InputIt2 last2,
                              Compare comp );
```

Checks if the first range `[first1, last1)` is lexicographically *less* than the second range `[first2, last2)`. The first version uses `operator<` to compare the elements, the second version uses the given comparison function `comp`.

Lexicographical comparison is a operation with the following properties:

- Two ranges are compared element by element.
- The first mismatching element defines which range is lexicographically *less* or *greater* than the other.
- If one range is a prefix of another, the shorter range is lexicographically *less* than the other.
- If two ranges have equivalent elements and are of the same length, then the ranges are lexicographically *equal*.
- An empty range is lexicographically *less* than any non-empty range.
- Two empty ranges are lexicographically *equal*.

Parameters

`first1, last1` - the first range of elements to examine

`first2, last2` - the second range of elements to examine

`comp` - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that objects of types `InputIt1` and `InputIt2` can be dereferenced and then implicitly converted to both `Type1` and `Type2`.

Type requirements

- `InputIt1, InputIt2` must meet the requirements of `InputIterator`.

Return value

`true` if the first range is lexicographically *less* than the second.

Complexity

At most $2 \cdot min(N1, N2)$ applications of the comparison operation, where `N1 = std::distance(first1, last1)` and `N2 = std::distance(first2, last2)`.

Possible implementation

First version

```
template<class InputIt1, class InputIt2>
bool lexicographical_compare(InputIt1 first1, InputIt1 last1,
                             InputIt2 first2, InputIt2 last2)
{
    for ( ; (first1 != last1) && (first2 != last2); first1++, first2++ ) {
        if (*first1 < *first2) return true;
        if (*first2 < *first1) return false;
    }
    return (first1 == last1) && (first2 != last2);
}
```

Second version

```
template<class InputIt1, class InputIt2, class Compare>
bool lexicographical_compare(InputIt1 first1, InputIt1 last1,
                             InputIt2 first2, InputIt2 last2,
                             Compare comp)
{
    for ( ; (first1 != last1) && (first2 != last2); first1++, first2++ ) {
        if (comp(*first1, *first2)) return true;
        if (comp(*first2, *first1)) return false;
    }
    return (first1 == last1) && (first2 != last2);
}
```

Example

Run this code

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>

int main()
{
    std::vector<char> v1 {'a', 'b', 'c', 'd'};
    std::vector<char> v2 {'a', 'b', 'c', 'd'};

    std::srand(std::time(0));
    while (!std::lexicographical_compare(v1.begin(), v1.end(),
                                         v2.begin(), v2.end())) {
        for (auto c : v1) std::cout << c << ' ';
        std::cout << ">= ";
        for (auto c : v2) std::cout << c << ' ';
        std::cout << '\n';

        std::random_shuffle(v1.begin(), v1.end());
        std::random_shuffle(v2.begin(), v2.end());
    }

    for (auto c : v1) std::cout << c << ' ';
    std::cout << "< ";
    for (auto c : v2) std::cout << c << ' ';
    std::cout << '\n';
}
```

Possible output:

```
a b c d >= a b c d
d a b c >= c b d a
b d a c >= a d c b
a c d b < c d a b
```

See also

equal	determines if two sets of elements are the same (function template)
std::experimental::parallel::lexicographical_compare (parallelism TS)	parallelized version of `std::lexicographical_compare` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/lexicographical_compare&oldid=79891"

std::is_permutation

Defined in header `<algorithm>`

```
template< class ForwardIt1, class ForwardIt2 >
bool is_permutation( ForwardIt1 first1, ForwardIt1 last1,
                     ForwardIt2 first2 );
```

(1) (since C++11)

```
template< class ForwardIt1, class ForwardIt2, class BinaryPredicate >
bool is_permutation( ForwardIt1 first1, ForwardIt1 last1,
                     ForwardIt2 first2, BinaryPredicate p );
```

(2) (since C++11)

```
template< class ForwardIt1, class ForwardIt2 >
bool is_permutation( ForwardIt1 first1, ForwardIt1 last1,
                     ForwardIt2 first2, ForwardIt2 last2 );
```

(3) (since C++14)

```
template< class ForwardIt1, class ForwardIt2, class BinaryPredicate >
bool is_permutation( ForwardIt1 first1, ForwardIt1 last1,
                     ForwardIt2 first2, ForwardIt2 last2,
                     BinaryPredicate p );
```

(4) (since C++14)

Returns `true` if there exists a permutation of the elements in the range `[first1, last1)` that makes that range equal to the range `[first2,last2)`, where `last2` denotes `first2 + (last1 - first1)` if it was not given.

The overloads (1) and (3) use `operator==` for equality, whereas the overloads (2) and (4) use the binary predicate `p`.

Parameters

first1, last1 - the range of elements to compare

first2, last2 - the second range to compare

p - binary predicate which returns `true` if the elements should be treated as equal.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type &a, const Type &b);
```

`Type` should be the value type of both `ForwardIt1` and `ForwardIt2`. The signature does not need to have `const &`, but the function must not modify the objects passed to it.

Type requirements

- `ForwardIt1, ForwardIt2` must meet the requirements of `ForwardIterator`.
- `ForwardIt1, ForwardIt2` must have the same value type.

Return value

`true` if the range `[first1, last1)` is a permutation of the range `[first2, last2)`.

Complexity

At most $O(N^2)$ applications of the predicate, or exactly N if the sequences are already equal, where `N=std::distance(first1, last1)`.

However if `ForwardIt1` and `ForwardIt2` meet the requirements of `RandomAccessIterator` and `std::distance(first1, last1) != std::distance(first2, last2)` no applications of the predicate are made.

Possible implementation

```
template<class ForwardIt1, class ForwardIt2>
bool is_permutation(ForwardIt1 first, ForwardIt1 last,
                    ForwardIt2 d_first)
{
   // skip common prefix
   std::tie(first, d_first) = std::mismatch(first, last, d_first);
   // iterate over the rest, counting how many times each element
   // from [first, last) appears in [d_first, d_last)
   if (first != last) {
       ForwardIt2 d_last = d_first;
       std::advance(d_last, std::distance(first, last));
       for (ForwardIt1 i = first; i != last; ++i) {
            if (i != std::find(first, i, *i)) continue; // already counted this *i

            auto m = std::count(d_first, d_last, *i);
            if (m==0 || std::count(i, last, *i) != m) {
                return false;
            }
        }
    }
    return true;
}
```

Example

Run this code

```
#include <algorithm>
#include <vector>
#include <iostream>
int main()
{
    std::vector<int> v1{1,2,3,4,5};
    std::vector<int> v2{3,5,4,1,2};
    std::cout << "3,5,4,1,2 is a permutation of 1,2,3,4,5? "
              << std::boolalpha
              << std::is_permutation(v1.begin(), v1.end(), v2.begin()) << '\n';

    std::vector<int> v3{3,5,4,1,1};
    std::cout << "3,5,4,1,1 is a permutation of 1,2,3,4,5? "
              << std::boolalpha
              << std::is_permutation(v1.begin(), v1.end(), v3.begin()) << '\n';
}
```

Output:

```
3,5,4,1,2 is a permutation of 1,2,3,4,5? true
3,5,4,1,1 is a permutation of 1,2,3,4,5? false
```

See also

next_permutation	generates the next greater lexicographic permutation of a range of elements (function template)
prev_permutation	generates the next smaller lexicographic permutation of a range of elements (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/is_permutation&oldid=72389"

std::next_permutation

Defined in header `<algorithm>`

```
template< class BidirIt >
bool next_permutation( BidirIt first, BidirIt last );
```
(1)

```
template< class BidirIt, class Compare >
bool next_permutation( BidirIt first, BidirIt last, Compare comp );
```
(2)

Transforms the range `[first, last)` into the next permutation from the set of all permutations that are lexicographically ordered with respect to `operator<` or `comp`. Returns `true` if such permutation exists, otherwise transforms the range into the first permutation (as if by `std::sort(first, last)`) and returns `false`.

Parameters

first, last - the range of elements to permute

comp - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `BidirIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `BidirIt` must meet the requirements of `ValueSwappable` and `BidirectionalIterator`.

Return value

`true` if the new permutation is lexicographically greater than the old. `false` if the last permutation was reached and the range was reset to the first permutation.

Exceptions

Any exceptions thrown from iterator operations or the element swap.

Complexity

At most *N*/*2* swaps, where `N = std::distance(first, last)`.

Possible implementation

```
template<class BidirIt>
bool next_permutation(BidirIt first, BidirIt last)
{
    if (first == last) return false;
    BidirIt i = last;
    if (first == --i) return false;

    while (1) {
        BidirIt i1, i2;
        i1 = i;
        if (*--i < *i1) {
            i2 = last;
            while (!(*i < *--i2))
                ;
            std::iter_swap(i, i2);
            std::reverse(i1, last);
            return true;
        }
        if (i == first) {
            std::reverse(first, last);
            return false;
        }
    }
}
```

Example

The following code prints all three permutations of the string "aba"

Run this code

```
#include <algorithm>
#include <string>
#include <iostream>

int main()
{
    std::string s = "aba";
    std::sort(s.begin(), s.end());
    do {
        std::cout << s << '\n';
    } while(std::next_permutation(s.begin(), s.end()));
}
```

Output:

```
aab
aba
baa
```

See also

is_permutation (C++11)	determines if a sequence is a permutation of another sequence (function template)
prev_permutation	generates the next smaller lexicographic permutation of a range of elements (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/next_permutation&oldid=57014"

std::prev_permutation

Defined in header `<algorithm>`

```
template< class BidirIt >
bool prev_permutation( BidirIt first, BidirIt last );
```
(1)

```
template< class BidirIt, class Compare >
bool prev_permutation( BidirIt first, BidirIt last, Compare comp );
```
(2)

Transforms the range `[first, last)` into the previous permutation from the set of all permutations that are lexicographically ordered with respect to `operator<` or `comp`. Returns `true` if such permutation exists, otherwise transforms the range into the last permutation (as if by `std::sort(first, last); std::reverse(first, last);`) and returns `false`.

Parameters

first, last - the range of elements to permute

comp - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `BidirIt` can be dereferenced and then implicitly converted to both of them.

Type requirements

- `BidirIt` must meet the requirements of `ValueSwappable` and `BidirectionalIterator`.

Return value

`true` if the new permutation precedes the old in lexicographical order. `false` if the first permutation was reached and the range was reset to the last permutation.

Exceptions

Any exceptions thrown from iterator operations or the element swap.

Complexity

At most `(last-first)/2` swaps.

Possible implementation

```
template<class BidirIt>
bool prev_permutation(BidirIt first, BidirIt last)
{
    if (first == last) return false;
    BidirIt i = last;
    if (first == --i) return false;

    while (1) {
        BidirIt i1, i2;
        i1 = i;
        if (*i1 < *--i) {
            i2 = last;
            while (!(*--i2 < *i))
                ;
            std::iter_swap(i, i2);
            std::reverse(i1, last);
            return true;
        }
        if (i == first) {
            std::reverse(first, last);
            return false;
        }
    }
}
```

Example

The following code prints all six permutations of the string "abc" in reverse order

Run this code

```
#include <algorithm>
#include <string>
#include <iostream>
#include <functional>
int main()
{
    std::string s="abc";
    std::sort(s.begin(), s.end(), std::greater<char>());
    do {
        std::cout << s << ' ';
    } while(std::prev_permutation(s.begin(), s.end()));
    std::cout << '\n';
}
```

Output:

```
cba cab bca bac acb abc
```

See also

is_permutation (C++11)	determines if a sequence is a permutation of another sequence (function template)
prev_permutation	generates the next smaller lexicographic permutation of a range of elements (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/prev_permutation&oldid=57033"

std::iota

Defined in header `<numeric>`

```
template< class ForwardIterator, class T >
void iota( ForwardIterator first, ForwardIterator last, T value );
```

(since C++11)

Fills the range `[first, last)` with sequentially increasing values, starting with `value` and repetitively evaluating `++value`.

Equivalent operation:

```
*(d_first)   = value;
*(d_first+1) = ++value;
*(d_first+2) = ++value;
*(d_first+3) = ++value;
...
```

Parameters

first, last	-	the range of elements to fill with sequentially increasing values starting with value
value	-	initial value to store, the expression ++value must be well-formed

Return value

(none)

Complexity

Exactly `last - first` increments and assignments.

Possible implementation

```
template<class ForwardIterator, class T>
void iota(ForwardIterator first, ForwardIterator last, T value)
{
    while(first != last) {
        *first++ = value;
        ++value;
    }
}
```

Notes

The function is named after the integer function ι from the programming language APL. It was one of the STL components (http%3A//www.sgi.com/tech/stl/iota.html) that were not included in C++98, but eventually made it into the standard library in C++11.

Example

The following example applies `std::shuffle` to a vector of `std::list` iterators since `std::shuffle` cannot be applied to a `std::list` directly. `std::iota` is used to populate both containers.

Run this code

```
#include <algorithm>
#include <iostream>
#include <list>
#include <numeric>
#include <random>
#include <vector>

int main()
{
    std::list<int> l(10);
    std::iota(l.begin(), l.end(), -4);

    std::vector<std::list<int>::iterator> v(l.size());
    std::iota(v.begin(), v.end(), l.begin());

    std::shuffle(v.begin(), v.end(), std::mt19937{std::random_device{}()});

    std::cout << "Contents of the list: ";
    for(auto n: l) std::cout << n << ' ';
    std::cout << '\n';

    std::cout << "Contents of the list, shuffled: ";
    for(auto i: v) std::cout << *i << ' ';
    std::cout << '\n';
}
```

Possible output:

```
Contents of the list: -4 -3 -2 -1 0 1 2 3 4 5
Contents of the list, shuffled: 0 -1 3 4 -4 1 -2 -3 2 5
```

See also

fill	assigns a range of elements a certain value (function template)
generate	saves the result of a function in a range (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/iota&oldid=79790"

std::accumulate

Defined in header `<numeric>`

```
template< class InputIt, class T >
T accumulate( InputIt first, InputIt last, T init );
```

(1)

```
template< class InputIt, class T, class BinaryOperation >
T accumulate( InputIt first, InputIt last, T init,
              BinaryOperation op );
```

(2)

Computes the sum of the given value `init` and the elements in the range `[first, last)`. The first version uses `operator+` to sum up the elements, the second version uses the given binary function `op`.

`op` must not have side effects.	(until C++11)
`op` must not invalidate any iterators, including the end iterators, or modify any elements of the range involved.	(since C++11)

Parameters

first, last	-	the range of elements to sum
init	-	initial value of the sum
op	-	binary operation function object that will be applied.

The signature of the function should be equivalent to the following:

```
Ret fun(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`.
The type `Type1` must be such that an object of type `T` can be implicitly converted to `Type1`. The type `Type2` must be such that an object of type `InputIt` can be dereferenced and then implicitly converted to `Type2`. The type `Ret` must be such that an object of type `T` can be assigned a value of type `Ret`.

Type requirements

- `InputIt` must meet the requirements of `InputIterator`.
- `T` must meet the requirements of `CopyAssignable` and `CopyConstructible`.

Return value

1) The sum of the given value and elements in the given range.

2) The result of left fold of the given range over `op`

Notes

Although `std::accumulate` performs left fold by default, right fold may be achieved by using reverse iterators, e.g. `std::accumulate(v.rbegin(), v.rend(), init, binop)`

Possible implementation

First version

```
template<class InputIt, class T>
T accumulate(InputIt first, InputIt last, T init)
{
    for (; first != last; ++first) {
        init = init + *first;
    }
    return init;
}
```

Second version

```
template<class InputIt, class T, class BinaryOperation>
T accumulate(InputIt first, InputIt last, T init,
             BinaryOperation op)
{
    for (; first != last; ++first) {
        init = op(init, *first);
    }
    return init;
}
```

Example

Run this code

```
#include <iostream>
#include <vector>
#include <numeric>
#include <string>
#include <functional>

int main()
{
    std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    int sum = std::accumulate(v.begin(), v.end(), 0);

    int product = std::accumulate(v.begin(), v.end(), 1, std::multiplies<int>());

    std::string s = std::accumulate(v.begin()+1, v.end(), std::to_string(v[0]),
                        [](const std::string& a, int b) {
                            return a + '-' + std::to_string(b);
                        });

    std::cout << "sum: " << sum << '\n'
              << "product: " << product << '\n'
              << "dash-separated string: " << s << '\n';
}
```

Output:

```
sum: 55
product: 3628800
dash-separated string: 1-2-3-4-5-6-7-8-9-10
```

See also

adjacent_difference	computes the differences between adjacent elements in a range (function template)
inner_product	computes the inner product of two ranges of elements (function template)
	computes the partial sum of a range of elements

`partial_sum`	(function template)
`reduce` (parallelism TS)	similar to **`std::accumulate`**, except out of order (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/accumulate&oldid=79857"

std::inner_product

Defined in header `<numeric>`

```
template< class InputIt1, class InputIt2, class T >
T inner_product( InputIt1 first1, InputIt1 last1,
                 InputIt2 first2, T value );
```
(1)

```
template<class InputIt1, class InputIt2, class T,
         class BinaryOperation1, class BinaryOperation2>
T inner_product( InputIt1 first1, InputIt1 last1,
                 InputIt2 first2, T value,
                 BinaryOperation1 op1,
                 BinaryOperation2 op2 );
```
(2)

Computes inner product (i.e. sum of products) of the range `[first1, last1)` and another range beginning at `first2`. The first version uses `operator*` to compute product of the element pairs and `operator+` to sum up the products, the second version uses `op2` and `op1` for these tasks respectively.

`op1` and `op2` must not have side effects.	(until C++11)
`op1` and `op2` must not invalidate any iterators, including the end iterators, or modify any elements of the ranges involved.	(since C++11)

Parameters

first1, last1 - the first range of elements

first2 - the beginning of the second range of elements

value - initial value of the sum of the products

op1 - binary operation function object that will be applied. This "sum" function takes a value returned by op2 and the current value of the accumulator and produces a new value to be stored in the accumulator.

The signature of the function should be equivalent to the following:

```
Ret fun(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`.
The types `Type1` and `Type2` must be such that objects of types `T` and `Type3` can be implicitly converted to `Type1` and `Type2` respectively. The type `Ret` must be such that an object of type `T` can be assigned a value of type `Ret`
.

op2 - binary operation function object that will be applied. This "product" function takes one value from each range and produces a new value.

The signature of the function should be equivalent to the following:

```
Ret fun(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`.
The types `Type1` and `Type2` must be such that objects of types `InputIt1` and `InputIt2` can be dereferenced and then implicitly converted to `Type1` and `Type2` respectively. The type `Ret` must be such that an object of type `Type3` can be assigned a value of type `Ret`.

Type requirements

- `InputIt1, InputIt2` must meet the requirements of `InputIterator`.
- `T` must meet the requirements of `CopyAssignable` and `CopyConstructible`.

Return value

The inner product of two ranges.

Possible implementation

First version

```
template<class InputIt1, class InputIt2, class T>
T inner_product(InputIt1 first1, InputIt1 last1,
                InputIt2 first2, T value)
{
    while (first1 != last1) {
         value = value + *first1 * *first2;
         ++first1;
         ++first2;
    }
    return value;
}
```

Second version

```
template<class InputIt1, class InputIt2,
         class T,
         class BinaryOperation1, class BinaryOperation2>
T inner_product(InputIt1 first1, InputIt1 last1,
                InputIt2 first2, T value,
                BinaryOperation1 op1
                BinaryOperation2 op2)
{
    while (first1 != last1) {
         value = op1(value, op2(*first1, *first2));
         ++first1;
         ++first2;
    }
    return value;
}
```

Example

Run this code

```
#include <numeric>
#include <iostream>
#include <vector>
#include <functional>
int main()
{
    std::vector<int> a{0, 1, 2, 3, 4};
    std::vector<int> b{5, 4, 2, 3, 1};

    int r1 = std::inner_product(a.begin(), a.end(), b.begin(), 0);
    std::cout << "Inner product of a and b: " << r1 << '\n';

    int r2 = std::inner_product(a.begin(), a.end(), b.begin(), 0,
                                std::plus<int>(), std::equal_to<int>());
    std::cout << "Number of pairwise matches between a and b: " <<  r2 << '\n';
}
```

Output:

```
Inner product of a and b: 21
```

```
Number of pairwise matches between a and b: 2
```

See also

accumulate	sums up a range of elements (function template)
partial_sum	computes the partial sum of a range of elements (function template)
std::experimental::parallel::inner_product (parallelism TS)	parallelized version of `std::inner_product` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/inner_product&oldid=79930"

std::adjacent_difference

Defined in header `<numeric>`

```
template< class InputIt, class OutputIt >
OutputIt adjacent_difference( InputIt first, InputIt last,
                              OutputIt d_first );
```
(1)

```
template< class InputIt, class OutputIt, class BinaryOperation >
OutputIt adjacent_difference( InputIt first, InputIt last,
                              OutputIt d_first, BinaryOperation op );
```
(2)

Computes the differences between the second and the first of each adjacent pair of elements of the range `[first, last)` and writes them to the range beginning at `d_first + 1`. Unmodified copy of `first` is written to `d_first`. The first version uses `operator-` to calculate the differences, the second version uses the given binary function `op`.

Equivalent operation:

```
*(d_first)   = *first;
*(d_first+1) = *(first+1) - *(first);
*(d_first+2) = *(first+2) - *(first+1);
*(d_first+3) = *(first+3) - *(first+2);
...
```

`op` must not have side effects.	(until C++11)
`op` must not invalidate any iterators, including the end iterators, or modify any elements of the ranges involved.	(since C++11)

Parameters

first, last - the range of elements

d_first - the beginning of the destination range

op - binary operation function object that will be applied.

The signature of the function should be equivalent to the following:

```
Ret fun(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`.
The types `Type1` and `Type2` must be such that an object of type `iterator_traits<InputIt>::value_type` can be implicitly converted to both of them. The type `Ret` must be such that an object of type `OutputIt` can be dereferenced and assigned a value of type `Ret`.

Type requirements

- `InputIt` must meet the requirements of `InputIterator`.
- `OutputIt` must meet the requirements of `OutputIterator`.

Return value

It to the element past the last element written.

Notes

If `first == last`, this function has no effect and will merely return `d_first`.

Complexity

Exactly `(last - first) - 1` applications of the binary operation

Possible implementation

First version

```
template<class InputIt, class OutputIt>
OutputIt adjacent_difference(InputIt first, InputIt last,
                             OutputIt d_first)
{
    if (first == last) return d_first;

    typedef typename std::iterator_traits<InputIt>::value_type value_t;
    value_t acc = *first;
    *d_first = acc;
    while (++first != last) {
        value_t val = *first;
        *++d_first = val - acc;
        acc = std::move(val);
    }
    return ++d_first;
}
```

Second version

```
template<class InputIt, class OutputIt, class BinaryOperation>
OutputIt adjacent_difference(InputIt first, InputIt last,
                             OutputIt d_first, BinaryOperation op)
{
    if (first == last) return d_first;

    typedef typename std::iterator_traits<InputIt>::value_type value_t;
    value_t acc = *first;
    *d_first = acc;
    while (++first != last) {
        value_t val = *first;
        *++d_first = op(val, acc);
        acc = std::move(val);
    }
    return ++d_first;
}
```

Example

The following code converts a sequence of even numbers to repetitions of the number 2 and converts a sequence of ones to a sequence of Fibonacci numbers.

Run this code

```
#include <numeric>
#include <vector>
#include <iostream>
#include <functional>

int main()
{
    std::vector<int> v{2, 4, 6, 8, 10, 12, 14, 16, 18, 20};
    std::adjacent_difference(v.begin(), v.end(), v.begin());
    for (auto n : v) {
        std::cout << n << ' ';
    }
    std::cout << '\n';

    v = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
    std::adjacent_difference(v.begin(), v.end() - 1, v.begin() + 1, std::plus<int>());

    for (auto n : v) {
        std::cout << n << ' ';
    }
    std::cout << '\n';
}
```

Output:

```
2 2 2 2 2 2 2 2 2 2
1 1 2 3 5 8 13 21 34 55
```

See also

partial_sum	computes the partial sum of a range of elements (function template)
accumulate	sums up a range of elements (function template)
std::experimental::parallel::ajacent_difference (parallelism TS)	parallelized version of `std::ajacent_difference` (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/adjacent_difference&oldid=80057"

std::partial_sum

Defined in header `<numeric>`

```
template< class InputIt, class OutputIt >
OutputIt partial_sum( InputIt first, InputIt last, OutputIt d_first );        (1)
```

```
template< class InputIt, class OutputIt, class BinaryOperation >
OutputIt partial_sum( InputIt first, InputIt last, OutputIt d_first,          (2)
                      BinaryOperation op );
```

Computes the partial sums of the elements in the subranges of the range `[first, last)` and writes them to the range beginning at `d_first`. The first version uses `operator+` to sum up the elements, the second version uses the given binary function `op`.

Equivalent operation:

```
*(d_first)   = *first;
*(d_first+1) = *first + *(first+1);
*(d_first+2) = *first + *(first+1) + *(first+2);
*(d_first+3) = *first + *(first+1) + *(first+2) + *(first+3);
...
```

`op` must not have side effects.	(until C++11)
`op` must not invalidate any iterators, including the end iterators, or modify any elements of the ranges involved.	(since C++11)

Parameters

first, last - the range of elements to sum

d_first - the beginning of the destination range

op - binary operation function object that will be applied.

The signature of the function should be equivalent to the following:

```
Ret fun(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`.
The type `Type1` must be such that an object of type `iterator_traits<InputIt>::value_type` can be implicitly converted to `Type1`. The type `Type2` must be such that an object of type `InputIt` can be dereferenced and then implicitly converted to `Type2`. The type `Ret` must be such that an object of type `iterator_traits<InputIt>::value_type` can be assigned a value of type `Ret`.

Type requirements

- `InputIt` must meet the requirements of `InputIterator`.
- `OutputIt` must meet the requirements of `OutputIterator`.

Return value

Iterator to the element past the last element written.

Complexity

Exactly `(last - first) - 1` applications of the binary operation

Possible implementation

First version

```
template<class InputIt, class OutputIt>
OutputIt partial_sum(InputIt first, InputIt last,
                     OutputIt d_first)
{
    if (first == last) return d_first;

    typename std::iterator_traits<InputIt>::value_type sum = *first;
    *d_first = sum;

    while (++first != last) {
       sum = sum + *first;
       *++d_first = sum;
    }
    return ++d_first;

    // or, since C++14:
    // return std::partial_sum(first, last, d_first, std::plus<>());
}
```

Second version

```
template<class InputIt, class OutputIt, class BinaryOperation>
OutputIt partial_sum(InputIt first, InputIt last,
                     OutputIt d_first, BinaryOperation op)
{
    if (first == last) return d_first;

    typename std::iterator_traits<InputIt>::value_type sum = *first;
    *d_first = sum;

    while (++first != last) {
       sum = op(sum, *first);
       *++d_first = sum;
    }
    return ++d_first;
}
```

Example

Run this code

```
#include <numeric>
#include <vector>
#include <iostream>
#include <iterator>
#include <functional>

int main()
{
    std::vector<int> v = {2, 2, 2, 2, 2, 2, 2, 2, 2, 2}; // or std::vector<int>v(10, 2);

    std::cout << "The first 10 even numbers are: ";
    std::partial_sum(v.begin(), v.end(),
                     std::ostream_iterator<int>(std::cout, " "));
    std::cout << '\n';

    std::partial_sum(v.begin(), v.end(), v.begin(), std::multiplies<int>());
    std::cout << "The first 10 powers of 2 are: ";
    for (auto n : v) {
        std::cout << n << " ";
    }
    std::cout << '\n';
}
```

Output:

```
The first 10 even numbers are: 2 4 6 8 10 12 14 16 18 20
The first 10 powers of 2 are: 2 4 8 16 32 64 128 256 512 1024
```

See also

adjacent_difference	computes the differences between adjacent elements in a range (function template)
accumulate	sums up a range of elements (function template)
inclusive_scan (parallelism TS)	similar to **std::partial_sum**, includes the ith input element in the ith sum (function template)
exclusive_scan (parallelism TS)	similar to **std::partial_sum**, excludes the ith input element from the ith sum (function template)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/partial_sum&oldid=79859"

std::qsort

Defined in header `<cstdlib>`

```
extern "C" void qsort( void *ptr, std::size_t count, std::size_t size,
           int (*comp)(const void *, const void *) );
```

```
extern "C++" void qsort( void *ptr, std::size_t count, std::size_t size,
           int (*comp)(const void *, const void *) );
```

Sorts the given array pointed to by `ptr` in ascending order. The array contains `count` elements of `size` bytes. Function pointed to by `comp` is used for object comparison.

If `comp` indicates two elements as equivalent, their order is undefined.

Parameters

ptr - pointer to the array to sort

count - number of element in the array

size - size of each element in the array in bytes

comp - comparison function which returns a negative integer value if the first argument is *less* than the second,

a positive integer value if the first argument is *greater* than the second and zero if the arguments are equal.
The signature of the comparison function should be equivalent to the following:

```
 int cmp(const void *a, const void *b);
```

The function must not modify the objects passed to it and must return consistent results when called for the same objects, regardless of their positions in the array.

Return value

(none)

Notes

Despite the name, C++, C, and POSIX standards do not require this function to be implemented using quicksort or make any complexity or stability guarantees.

The type of the elements of the array must be a `TrivialType`, otherwise the behavior is undefined.

The two overloads provided by the C++ standard library are distinct because the types of the parameter `comp` are distinct (language linkage is part of its type)

Example

The following code sorts an array of integers using `qsort()`.

Run this code

```
#include <iostream>
#include <cstdlib>
#include <climits>

int main() {
  int a[] = {-2, 99, 0, -743, 2, INT_MIN, 4};
  constexpr std::size_t size = sizeof a / sizeof *a;

  std::qsort(a, size, sizeof *a, [](const void* a, const void* b) {
    int arg1 = *static_cast<const int*>(a);
    int arg2 = *static_cast<const int*>(b);

    if(arg1 < arg2) return -1;
    if(arg1 > arg2) return 1;
    return 0;

    // return (arg1 > arg2) - (arg1 < arg2); // possible shortcut
    // return arg1 - arg2; // erroneous shortcut (fails if INT_MIN is present)
  });

  for (int ai : a) std::cout << ai << ' ';
}
```

Output:

```
-2147483648 -743 -2 0 2 4 99
```

See also

bsearch	searches an array for an element of unspecified type (function)
sort	sorts a range into ascending order (function template)
is_trivial (C++11)	checks if a type is trivial (class template)
C documentation for **qsort**	

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/qsort&oldid=80308"

std::bsearch

Defined in header `<cstdlib>`

```
extern "C" void* bsearch( const void* key, const void* ptr, std::size_t count,
              std::size_t size, int (*comp)(const void*, const void*) );
```

```
extern "C++" void* bsearch( const void* key, const void* ptr, std::size_t count,
              std::size_t size, int (*comp)(const void*, const void*) );
```

Finds an element equal to element pointed to by `key` in an array pointed to by `ptr`. The array contains `count` elements of `size` bytes each and must be partitioned with respect to the object pointed to by `key`, that is, all the elements that compare less than must appear before all the elements that compare equal to, and those must appear before all the elements that compare greater than the key object. A fully sorted array satisfies these requirements. The elements are compared using function pointed to by `comp`.

The behavior is undefined if the array is not already partitioned in ascending order with respect to key, according to the same criterion that `comp` uses.

If the array contains several elements that `comp` would indicate as equal to the element searched for, then it is unspecified which element the function will return as the result.

Parameters

key - pointer to the element to search for

ptr - pointer to the array to examine

count - number of element in the array

size - size of each element in the array in bytes

comp - comparison function which returns a negative integer value if the first argument is *less* than the second,

a positive integer value if the first argument is *greater* than the second and zero if the arguments are equal. `key` is passed as the first argument, an element from the array as the second.
The signature of the comparison function should be equivalent to the following:

```
int cmp(const void *a, const void *b);
```

The function must not modify the objects passed to it and must return consistent results when called for the same objects, regardless of their positions in the array.

Return value

Pointer to the found element or null pointer if the element has not been found.

Notes

The two overloads provided by the C++ standard library are distinct because the types of the parameter `comp` are distinct (language linkage is part of its type)

Example

Run this code

```
#include <cstdlib>
#include <iostream>

int compare(const void *ap, const void *bp)
```

```
{
    const int *a = (int *) ap;
    const int *b = (int *) bp;
    if(*a < *b)
        return -1;
    else if(*a > *b)
        return 1;
    else
        return 0;
}

int main(int argc, char **argv)
{
    const int ARR_SIZE = 8;
    int arr[ARR_SIZE] = { 1, 2, 3, 4, 5, 6, 7, 8 };

    int key1 = 4;
    int *p1 = (int *) std::bsearch(&key1, arr, ARR_SIZE, sizeof(arr[0]), compare);
    if(p1)
        std::cout << "value " << key1 << " found at position " << (p1 - arr) << '\n';
     else
        std::cout << "value " << key1 << " not found\n";

    int key2 = 9;
    int *p2 = (int *) std::bsearch(&key2, arr, ARR_SIZE, sizeof(arr[0]), compare);
    if(p2)
        std::cout << "value " << key2 << " found at position " << (p2 - arr) << '\n';
     else
        std::cout << "value " << key2 << " not found\n";
}
```

Output:

```
value 4 found at position 3
value 9 not found
```

See also

qsort	sorts a range of elements with unspecified type (function)
equal_range	returns range of elements matching a specific key (function template)
C documentation for **bsearch**	

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/algorithm/bsearch&oldid=76404"