
11.2. Algorithm Overview

This section presents an overview of all the C++ standard library algorithms to give you an idea of their abilities and to be better able to find the best algorithm to solve a certain problem.

11.2.1. A Brief Introduction

Algorithms were introduced in Chapter 6 along with the STL. In particular, Section 6.4, page 199, and Section 6.7, page 217, discuss the role of algorithms and some important constraints about their use. All STL algorithms process one or more iterator ranges. The first range is usually specified by its beginning and its end. For additional ranges, you generally need to pass only the beginning because the end follows from the number of elements of the first range. The caller must ensure that the ranges are valid. That is, the beginning must refer to a previous or the same element of the same container as the end. Additional ranges must have enough elements.

Algorithms work in overwrite mode rather than in insert mode. Thus, the caller must ensure that destination ranges have enough elements. You can use special insert iterators (see Section 9.4.2, page 454) to switch from overwrite to insert mode.

To increase their flexibility and power, several algorithms allow the user to pass user-defined operations, which they call internally. These operations might be ordinary functions or function objects. If these functions return a Boolean value, they are called *predicates*. You can use predicates for the following tasks:

- You can pass a function, a function object, or a lambda that specifies a unary predicate as the search criterion for a search algorithm. The unary predicate is used to check whether an element fits the criterion. For example, you could search the first element that is less than 50.
- You can pass a function, a function object, or a lambda that specifies a binary predicate as the sorting criterion for a sort algorithm. The binary predicate is used to compare two elements. For example, you could pass a criterion that lets objects that represent a person sort according to the person's last name (see Section 10.1.1, page 476, for an example).
- You can pass a unary predicate as the criterion that specifies for which elements an operation should apply. For example, you could specify that only elements with an odd value should be removed.
- You can specify the numeric operation of numeric algorithms. For example, you could use `accumulate()`, which normally processes the sum of elements, to process the product of all elements.

Note that predicates should not modify their state due to a function call (see Section 10.1.4, page 483).

See Section 6.8, page 224, Section 6.9, page 229, Section 6.10, page 233, and Chapter 10 for examples and details about functions, function objects, and lambdas that are used as algorithm parameters.

11.2.2. Classification of Algorithms

Different algorithms meet different needs and so can be classified by their main purposes. For example, some algorithms operate as read only, some modify elements, and some change the order of elements. This subsection gives you a brief idea of the functionality of each algorithm and in which aspect it differs from similar algorithms.

The name of an algorithm gives you a first impression of its purpose. The designers of the STL introduced two special suffixes:

1. **The `_if` suffix** is used when you can call two forms of an algorithm that have the same number of parameters either by passing a value or by passing a function or function object. In this case, the version without the suffix is used for values, and the version with the `_if` suffix is used for functions and function objects. For example, `find()` searches for an element that has a certain value, whereas `find_if()` searches for an element that meets the criterion passed as a function, a function object, or a lambda.

   However, not all algorithms that have a parameter for functions and function objects have the `_if` suffix. When the function or function-object version of an algorithm has an additional argument, it has the same name. For example, `min_element()` called with two arguments returns the minimum element in the range according to a comparison with operator `<`. If you pass a third element, it is used as the comparison criterion.

2. **The `_copy` suffix** is used as an indication that elements are not only manipulated but also copied into a destination range. For example, `reverse()` reverses the order of elements inside a range, whereas `reverse_copy()` copies the elements into another range in reverse order.

The following subsections and sections describe the algorithms according to the following classification:

- Nonmodifying algorithms
- Modifying algorithms
- Removing algorithms
- Mutating algorithms
- Sorting algorithms
- Sorted-range algorithms
- Numeric algorithms

Algorithms belonging to more than one category are described in the category I consider to be the most important.

Nonmodifying Algorithms

Nonmodifying algorithms change neither the order nor the value of the elements they process. These algorithms operate with input and forward iterators; therefore, you can call them for all standard containers. Table 11.1 lists the nonmodifying algorithms of the C++ standard library. See page 515 for nonmodifying algorithms that are provided especially for sorted input ranges.

Table 11.1. Nonmodifying Algorithms

Name	Effect	Page
`for_each()`	Performs an operation for each element	519
`count()`	Returns the number of elements	524
`count_if()`	Returns the number of elements that match a criterion	524
`min_element()`	Returns the element with the smallest value	526
`max_element()`	Returns the element with the largest value	526
`minmax_element()`	Returns the elements with the smallest and largest value (since C++11)	526
`find()`	Searches for the first element with the passed value	528
`find_if()`	Searches for the first element that matches a criterion	528
`find_if_not()`	Searches for the first element that matches a criterion not (since C++11)	528
`search_n()`	Searches for the first *n* consecutive elements with certain properties	531
`search()`	Searches for the first occurrence of a subrange	534
`find_end()`	Searches for the last occurrence of a subrange	537
`find_first_of()`	Searches the first of several possible elements	539
`adjacent_find()`	Searches for two adjacent elements that are equal (by some criterion)	540
`equal()`	Returns whether two ranges are equal	542
`is_permutation()`	Returns whether two unordered ranges contain equal elements (since C++11)	544
`mismatch()`	Returns the first elements of two sequences that differ	546
`lexicographical_compare()`	Returns whether a range is lexicographically less than another range	548
`is_sorted()`	Returns whether the elements in a range are sorted (since C++11)	550
`is_sorted_until()`	Returns the first unsorted element in a range (since C++11)	550
`is_partitioned()`	Returns whether the elements in a range are partitioned in two groups according to a criterion (since C++11)	552
`partition_point()`	Returns the partitioning element for a range partitioned into elements fulfilling and elements not fulfilling a predicate (since C++11)	552
`is_heap()`	Returns whether the elements in a range are sorted as a heap (since C++11)	554
`is_heap_until()`	Returns the first element in a range not sorted as a heap (since C++11)	554
`all_of()`	Returns whether all elements match a criterion (since C++11)	555
`any_of()`	Returns whether at least one element matches a criterion (since C++11)	555
`none_of()`	Returns whether none of the elements matches a criterion (since C++11)	555

Historically, one of the most important algorithms was `for_each()`. `for_each()` calls an operation provided by the caller for each element. That operation is usually used to process each element of the range individually. For example, you can pass `for_each()` a function that prints each element or calls a modifying operation for each element. Note, however, that since C++11, the range-based `for` loop provides this behavior more conveniently and more naturally (see Section 3.1.4, page 17, and Section 6.2.1,

page 174). Thus, `for_each()` might lose its importance over time.

Several of the nonmodifying algorithms perform searching. Unfortunately, the naming scheme of searching algorithms is a mess. In addition, the naming schemes of searching algorithms and searching string functions differ (Table 11.2). As is often the case, there are historical reasons for this. First, the STL and string classes were designed independently. Second, the `find_end()`, `find_first_of()`, and `search_n()` algorithms were not part of the original STL. So, for example, the name `find_end()` instead of `search_end()` was chosen by accident (it is easy to forget aspects of the whole picture, such as consistency, when you are caught up in the details). Also by accident, a form of `search_n()` breaks the general concept of the original STL. See Section 11.5.3, page 532, for a description of this problem.

Table 11.2. Comparison of Searching String Operations and Algorithms

Search for	String Function	STL Algorithm
First occurrence of one element	`find()`	`find()`
Last occurrence of one element	`rfind()`	`find()` with reverse iterators
First occurrence of a subrange	`find()`	`search()`
Last occurrence of a subrange	`rfind()`	`find_end()`
First occurrence of several elements	`find_first_of()`	`find_first_of()`
Last occurrence of several elements	`find_last_of()`	`find_first_of()` with reverse iterators
First occurrence of n consecutive elements		`search_n()`

Modifying Algorithms

Modifying algorithms change the value of elements. Such algorithms might modify the elements of a range directly or modify them while they are being copied into another range. If elements are copied into a destination range, the source range is not changed. Table 11.3 lists the modifying algorithms of the C++ standard library.

Table 11.3. Modifying Algorithms

Name	Effect	Page
for_each()	Performs an operation for each element	519
copy()	Copies a range starting with the first element	557
copy_if()	Copies elements that match a criterion (since C++11)	557
copy_n()	Copies *n* elements (since C++11)	557
copy_backward()	Copies a range starting with the last element	561
move()	Moves elements of a range starting with the first element (since C++11)	557
move_backward()	Moves elements of a range starting with the last element (since C++11)	561
transform()	Modifies (and copies) elements; combines elements of two ranges	563 564
merge()	Merges two ranges	614
swap_ranges()	Swaps elements of two ranges	566
fill()	Replaces each element with a given value	568
fill_n()	Replaces *n* elements with a given value	568
generate()	Replaces each element with the result of an operation	569
generate_n()	Replaces *n* elements with the result of an operation	569
iota()	Replaces each element with a sequence of incremented values (since C++11)	571
replace()	Replaces elements that have a special value with another value	571
replace_if()	Replaces elements that match a criterion with another value	571
replace_copy()	Replaces elements that have a special value while copying the whole range	573
replace_copy_if()	Replaces elements that match a criterion while copying the whole range	573

The fundamental modifying algorithms are `for_each()` (again) and `transform()`. You can use both to modify elements of a sequence. However, their behavior differs as follows:

- **`for_each()`** accepts an operation that modifies its argument. Thus, the argument has to be passed by reference. For example:

Click here to view code image

```
void square (int& elem)    // call-by-reference
{
    elem = elem * elem;    // assign processed value directly
}
...
for_each(coll.begin(),coll.end(),    // range
         square);                    // operation
```

- **`transform()`** uses an operation that returns the modified argument. The trick is that it can be used to assign the result to the original element. For example:

Click here to view code image

```
int square (int elem)                    // call-by-value
{
    return elem * elem;                  // return processed value
}
...
transform (coll.cbegin(), coll.cend(),   // source range
           coll.begin(),                 // destination range
           square);                      // operation
```

Using `transform()` is a bit slower because it returns and assigns the result instead of modifying the element directly. However, it is more flexible because it can also be used to modify elements while they are being copied into a different destination sequence. Another version of `transform()` can process and combine elements of two source ranges.

Strictly speaking, `merge()` does not necessarily have to be part of the list of modifying algorithms, because it requires that its input ranges be sorted. So, it should be part of the algorithms for sorted ranges (see page 515). In practice, however, `merge()` also merges the elements of unsorted ranges. Of course, then the result is unsorted. Nevertheless, to be safe, you should call `merge()` only for sorted ranges.

Note that elements of associative and unordered containers are constant to ensure that you can't compromise the sorted order of the elements due to an element modification. Therefore, you can't use these containers as a destination for modifying algorithms.

In addition to these modifying algorithms, the C++ standard library provides modifying algorithms for sorted ranges. See page 515 for details.

Removing Algorithms

Removing algorithms are a special form of modifying algorithms. They can remove the elements either in a single range or while these elements are being copied into another range. As with modifying algorithms, you can't use an associative or unordered container as a destination, because the elements of these containers are considered to be constant. Table 11.4 lists the removing algorithms of the C++ standard library.

Table 11.4. Removing Algorithms

Name	Effect	Page
`remove()`	Removes elements with a given value	575
`remove_if()`	Removes elements that match a given criterion	575
`remove_copy()`	Copies elements that do not match a given value	577
`remove_copy_if()`	Copies elements that do not match a given criterion	577
`unique()`	Removes adjacent duplicates (elements that are equal to their predecessor)	578
`unique_copy()`	Copies elements while removing adjacent duplicates	580

Note that these algorithms remove elements logically only by overwriting them with the following elements that were not removed. Thus, removing algorithms do not change the number of elements in the ranges on which they operate. Instead, they return the position of the new "end" of the range. It's up to the caller to use that new end, such as to remove the elements physically. See Section 6.7.1, page 218, for a detailed discussion of this behavior.

Mutating Algorithms

Mutating algorithms are algorithms that change the order of elements (and not their values) by assigning and swapping their values. Table 11.5 lists the mutating algorithms of the C++ standard library. As with modifying algorithms, you can't use an associative or unordered container as a destination, because the elements of these containers are considered to be constant.

Table 11.5. Mutating Algorithms

Name	Effect	Page
`reverse()`	Reverses the order of the elements	583
`reverse_copy()`	Copies the elements while reversing their order	583
`rotate()`	Rotates the order of the elements	584
`rotate_copy()`	Copies the elements while rotating their order	585
`next_permutation()`	Permutates the order of the elements	587
`prev_permutation()`	Permutates the order of the elements	587
`shuffle()`	Brings the elements into a random order (since C++11)	589
`random_shuffle()`	Brings the elements into a random order	589
`partition()`	Changes the order of the elements so that elements that match a criterion are at the front	592
`stable_partition()`	Same as `partition()` but preserves the relative order of matching and nonmatching elements	592
`partition_copy()`	Copies the elements while changing the order so that elements that match a criterion are at the front	594

Sorting Algorithms

Sorting algorithms are a special kind of mutating algorithm because they also change the order of the elements. However, sorting is more complicated and therefore usually takes more time than simple mutating operations. In fact, these algorithms usually have worse than linear complexity[1] and require random-access iterators (for the destination). Table 11.6 lists the sorting algorithms. Table 11.7 lists the corresponding algorithms that allow checking whether a sequence is (partially) sorted.

[1] See Section 2.2, page 10, for an introduction to and a discussion of complexity.

Table 11.6. Sorting Algorithms

Name	Effect	Page
`sort()`	Sorts all elements	596
`stable_sort()`	Sorts while preserving order of equal elements	596
`partial_sort()`	Sorts until the first *n* elements are correct	599
`partial_sort_copy()`	Copies elements in sorted order	600
`nth_element()`	Sorts according to the *n*th position	602
`partition()`	Changes the order of the elements so that elements that match a criterion are at the beginning	592
`stable_partition()`	Same as `partition()` but preserves the relative order of matching and nonmatching elements	592
`partition_copy()`	Copies the elements while changing the order so that elements that match a criterion are at the beginning	594
`make_heap()`	Converts a range into a heap	604
`push_heap()`	Adds an element to a heap	605
`pop_heap()`	Removes an element from a heap	605
`sort_heap()`	Sorts the heap (it is no longer a heap after the call)	605

Table 11.7. Algorithms Checking for Sortings

Name	Effect	Page
`is_sorted()`	Returns whether the elements in a range are sorted (since C++11)	550
`is_sorted_until()`	Returns the first unsorted element in a range (since C++11)	550
`is_partitioned()`	Returns whether the elements in a range are partitioned in two groups according to a criterion (since C++11)	552
`partition_point()`	Returns the partitioning element for a range partitioned into elements fulfilling and elements not fulfilling a predicate (since C++11)	552
`is_heap()`	Returns whether the elements in a range are sorted as a heap (since C++11)	554
`is_heap_until()`	Returns the first element in a range not sorted as a heap (since C++11)	554

Time often is critical for sorting algorithms. Therefore, the C++ standard library provides more than one sorting algorithm. The algorithms use different ways of sorting, and some algorithms don't sort all elements. For example, `nth_element()` stops when the *n*th element of the sequence is correct according to the sorting criterion. For the other elements, it guarantees only that the previous elements have a lesser or equal value and that the following elements have a greater or equal value. To sort all elements of a sequence, you should consider the following algorithms:

- **`sort()`**, based historically on *quicksort*. However, current implementations usually use *introsort*. Introsort is a new algorithm that, by default, operates like quicksort but switches to heapsort when it is going to have quadratic complexity. Thus, this algorithm guarantees a good runtime (*n* * *log*(*n*) complexity):

Click here to view code image

```
// sort all elements (best n*log(n) complexity)
sort (coll.begin(), coll.end());
```

Before C++11, `sort()` did only guarantee to have *n* * *log*(*n*) complexity on average, but quadratic complexity in the worst case. To ensure to avoid the worst case scenario, you had to use another algorithm, such as `partial_sort()` or `stable_sort()`.

- **`partial_sort()`**, based historically on *heapsort*. Thus, it guarantees *n* * *log*(*n*) complexity in any case. However, in most circumstances, heapsort is slower than quicksort by a factor of two to five. So, if `sort()` is implemented as quicksort and `partial_sort()` is implemented as heapsort, `partial_sort()` has the better complexity, but `sort()` has the better runtime in most cases. The advantage of `partial_sort()` is that it guarantees *n* * *log*(*n*) complexity in any case, so it never reaches quadratic complexity.

  In addition, `partial_sort()` has the special ability to stop sorting when only the first *n* elements need to be sorted. To sort all the elements, you have to pass the end of the sequence as second and last argument:

Click here to view code image

```
// sort all elements
// - always n*log(n) complexity
// - but usually twice as long as sort()
partial_sort (coll.begin(), coll.end(), coll.end());
```

- **stable_sort()**, based historically on *mergesort*. It sorts all the elements:

Click here to view code image

```
// sort all elements
// - n*log(n) or n*log(n)*log(n) complexity
stable_sort (coll.begin(), coll.end());
```

However, it needs enough additional memory to have *n* * *log*(*n*) complexity. Otherwise, it has *n* * *log*(*n*) * *log*(*n*) complexity. The advantage of `stable_sort()` is that it preserves the order of equal elements.

Now you have a brief idea of which sorting algorithm might best meet your needs. But note that the standard guarantees complexity but not how it is implemented. This is an advantage in that an implementation could benefit from algorithm innovations and use a better way of sorting without breaking the standard. For example, the `sort()` algorithm initially was implemented by using *quicksort*. For this reason C++98 specified "*n* * *log*(*n*) on average" complexity, because quicksort can become quadratic in the worst case. However, even implementations of C++98 and C++03 switched to *introsort* to benefit from this new sorting algorithm, so that with C++11 the "on average" restriction was removed. The disadvantage of the fact that the standard does not guarantee exact complexity is that an implementation could use a standard-conforming, but very bad, algorithm. For example, using heapsort to implement `sort()` would be standard conforming. Of course, you simply could test, which algorithm fits best, but be aware that measurements might not be portable.

There are even more algorithms to sort elements. For example, the heap algorithms are provided to call the functions that implement a heap directly (a heap can be considered as a binary tree implemented as sequential collection). The heap algorithms are provided and used as the base for efficient implementations of priority queues (see Section 12.3, page 641). You can use them to sort all elements of a collection by calling them as follows:

```
// sort all elements
// - n+n*log(n) complexity
make_heap (coll.begin(), coll.end());
sort_heap (coll.begin(), coll.end());
```

See Section 11.9.4, page 604, for details about heaps and heap algorithms.

The `nth_element()` algorithms are provided if you need only the *n*th sorted element or the set of the *n* highest or *n* lowest elements (not sorted). Thus, `nth_element()` is a way to split elements into two subsets according to a sorting criterion. However, you could also use `partition()` or `stable_partition()` to do this. The differences are as follows:

- For **nth_element()**, you pass the number of elements you want to have in the first part (and therefore also in the second part). For example:

Click here to view code image

```
// move the four lowest elements to the front
nth_element (coll.begin(),      // beginning of range
             coll.begin()+3,    // position between first and second part
             coll.end());       // end of range
```

However, after the call, you don't know the exact criterion that is the difference between the first and the second parts. Both parts may, in fact, have elements with the same value as the *n*th element.

- For **partition()**, you pass the exact sorting criterion that serves as the difference between the first and the second parts. For example:

Click here to view code image

```
// move all elements less than seven to the front
vector<int>::iterator pos;
pos = partition (coll1.begin(), coll1.end(),    // range
                 [](int elem){                  // criterion
                     return elem<7;
                 });
```

Here, after the call, you don't know how many elements are owned by the first and the second parts. The return value `pos` refers to the first element of the second part that contains all elements that don't match the criterion, if any.

- **stable_partition()** behaves similarly to `partition()` but has an additional ability. It guarantees that the order of the elements in both parts remains stable according to their relative positions to the other elements in the same part.

You can always pass the sorting criterion to all sorting algorithms as an optional argument. The default sorting argument is the function object `less<>`, so that elements are sorted in ascending order according to their values. Note that the sorting criterion has to define a *strict weak ordering* on the values. A criterion, where values are compared as equal or less, such as operator `<=`, does not fit this requirement. See Section 7.7, page 314, for details.

As with modifying algorithms, you can't use an associative container as a destination, because the elements of the associative containers are considered to be constant.

Lists and forward lists do not provide random-access iterators, so you can't call sorting algorithms for them either. However, both provide a member function `sort()` to sort their elements; see Section 8.8.1, page 422.

Sorted-Range Algorithms

Sorted-range algorithms require that the ranges on which they operate be sorted according to their sorting criterion. Table 11.8 lists all C++ standard library algorithms that are especially written for sorted ranges. As for associative containers, these algorithms have the advantage of a better complexity.

Table 11.8. Algorithms for Sorted Ranges

Name	Effect	Page
`binary_search()`	Returns whether the range contains an element	608
`includes()`	Returns whether each element of a range is also an element of another range	609
`lower_bound()`	Finds the first element greater than or equal to a given value	611
`upper_bound()`	Finds the first element greater than a given value	611
`equal_range()`	Returns the range of elements equal to a given value	613
`merge()`	Merges the elements of two ranges	614
`set_union()`	Processes the sorted union of two ranges	616
`set_intersection()`	Processes the sorted intersection of two ranges	617
`set_difference()`	Processes a sorted range that contains all elements of a range that are not part of another range	618
`set_symmetric_difference()`	Processes a sorted range that contains all elements that are in exactly one of two ranges	619
`inplace_merge()`	Merges two consecutive sorted ranges	622
`partition_point()`	Returns the partitioning element for a range partitioned into elements fulfilling and elements not fulfilling a predicate (since C++11)	552

The first five sorted-range algorithms in Table 11.8 are nonmodifying, searching only according to their purpose. The other algorithms combine two sorted input ranges and write the result to a destination range. In general, the result of these algorithms is also sorted.

Numeric Algorithms

These algorithms combine numeric elements in different ways. Table 11.9 lists the numeric algorithms of the C++ standard library. If you understand the names, you get an idea of the purpose of the algorithms. However, these algorithms are more flexible and more powerful than they may seem at first. For example, by default, `accumulate()` processes the sum of all elements. When you use strings as elements, you concatenate them by using this algorithm. When you switch from operator `+` to operator `*`, you get the product of all elements. As another example, you should know that `adjacent_difference()` and `partial_sum()` transfer a range of absolute values into a range of relative values and vice versa.

Table 11.9. Numeric Algorithms

Name	Effect	Page
`accumulate()`	Combines all element values (processes sum, product, and so forth)	623
`inner_product()`	Combines all elements of two ranges	625
`adjacent_difference()`	Combines each element with its predecessor	628
`partial_sum()`	Combines each element with all its predecessors	627

Both `accumulate()` and `inner_product()` process and return a single value without modifying the ranges. The other algorithms write the results to a destination range that has the same number of elements as the source range.