"Advanced C++: Templates and Generic Programming"

STL Algorithms

Hüseyin Akcan
huseyin.akcan@ieu.edu.tr

To use the algorithms of the C++ standard library you must include the header file `<algorithm>`

We can analyze the algorithms in the following categories:

- Non-modifying algorithms
- Modifying algorithms
- Removing algorithms
- Mutating algorithms
- Sorting algorithms
- Sorted range algorithms
- Numeric algorithms
Non-modifying algorithms: for_each

- The for_each() algorithm is very flexible because it allows you to access, process, and modify each element in many different ways.

UnaryProc

for_each (InputIterator beg, InputIterator end, UnaryProc op)
- Calls op (elem) for each element in the range [beg,end).
- Returns a copy of the (internally modified) op.
- op might modify the elements.
- Any return value of op is ignored.
- Complexity: linear (numberOfElements calls of op()).

See for_each.cpp for example code...
Non-modifying algorithms: count

- difference _type count (InputIterator beg, InputIterator end, const T& value)
- difference _type count_if (InputIterator beg, InputIterator end, UnaryPredicate op)

- The first form counts the elements in the range [beg,end) that are equal to value.
- The second form counts the elements in the range [beg,end) for which the unary predicate op (elem) yields true.
- The type of the return value, difference_type, is the difference type of the iterator:
  - typename iterator_traits<lnputIterator>::difference_type
- op should not modify the passed arguments.
- Complexity: linear (numberofElements comparisons or calls of op() respectively).

See count.cpp for example code...
Modifying algorithms: fill

- void fill (ForwardIterator beg, ForwardIterator end, const T& newValue)
- void fill_n (OutputIterator beg, Size num, const T& newValue)

fill() assigns newValue to each element in the range [beg,end).
fill_n() assigns newValue to the first num elements in the range starting with beg.

The caller must ensure that the destination range is big enough or that insert iterators are used.

Complexity: linear (numberOfElements or num assignments respectively).

See fill.cpp for example code...
Modifying algorithms: generate

- void generate (ForwardIterator beg, ForwardIterator end, Func op)
- void generate_n (OutputIterator beg, Size num, Func op)
- `generate()` assigns the values that are generated by a call of `op()` to each element in the range `[beg,end)`.
- `generate_n()` assigns the values that are generated by a call of `op()` to the first num elements in the range starting with `beg`.
- The caller must ensure that the destination range is big enough or that insert iterators are used.
- Complexity: linear (numberOfElements or num calls of `op()` and assignments).

See generate.cpp for example code...
Removing algorithms

- The following algorithms remove elements from a range according to their value or to a criterion.
- These algorithms, however, cannot change the number of elements.
- They only move logically by overwriting "removed" elements with the following elements that were not removed.
- They return the new logical end of the range (the position after the last element not removed).

See remove.cpp for example code...
**Mutating algorithms: reverse**

- Mutating algorithms change the order of elements (but not their values). Because elements of associative containers have a fixed order, you can't use them as a destination for mutating algorithms.

- void reverse (BidirectionalIterator beg, BidirectionalIterator end)

- OutputIterator reverse_copy (BidirectionalIterator sourceBeg, BidirectionalIterator sourceEnd, Output Iterator destBeg)

- reverse() reverses the order of the elements inside the range [beg,end).

- reverse_copy() reverses the order of the elements while copying them from the source range [sourceBeg,sourceEnd) to the destination range starting with destBeg.

- reverse_copy() returns the position after the last copied element in the destination range (the first element that is not overwritten).

- The caller must ensure that the destination range is big enough or that insert iterators are used.

See reverse.cpp for example code...
Mutating algorithms: shuffle

- void random_shuffle (RandomAccessIterator beg, RandomAccessIterator end)

- void random_shuffle (RandomAccessIterator beg, RandomAccessIterator end, RandomFunc& op)

  The first form shuffles the order of the elements in the range [beg,end) using a uniform distribution random number generator.

  The second form shuffles the order of the elements in the range [beg,end) using op. op is called with an integral value of difference_type of the iterator: op (max)

- It should return a random number greater than or equal to zero and less than max. Thus, it should not return max itself.

- Note that op is a nonconstant reference, so you can't pass a temporary value or an ordinary function.

- Complexity: linear (numberOfElements-1 swaps).
Sorting algorithms: sort

- void sort (RandomAccessIterator beg, RandomAccessIterator end)
- void sort (RandomAccessIterator beg, RandomAccessIterator end, BinaryPredicate op)
- void stable_sort (RandomAccessIterator beg, RandomAccessIterator end)
- void stable_sort (RandomAccessIterator beg, RandomAccessIterator end, BinaryPredicate op)

The first forms of sort() and stable_sort() sort all elements in the range [beg,end) with operator <.

The second forms of sort() and stable_sort() sort all elements by using the binary predicate op(elem1,elem2) as the sorting criterion.

Note that op should not change its state during a function call.

The difference between sort() and stable_sort() is that stable_sort() guarantees that the order of equal elements remains stable.

Works on random access containers
Sorting algorithms: nth_element

- void nth_element (RandomAccessIterator beg, RandomAccessIterator nth, RandomAccessIterator end)
- void nth_element (RandomAccessIterator beg, RandomAccessIterator nth, RandomAccessIterator end, BinaryPredicate op)
- Both forms sort the elements in the range [beg,end) so that the correct element is at the nth position and all elements in front are less than or equal to this element, and all elements that follow are greater than or equal to it. Thus, you get two subsequences separated by the element at position n, whereby each element of the first subsequence is less than or equal to each element of the second subsequence. This is helpful if you need only the set of the n highest or lowest elements without having all the elements sorted.
- The first form uses operator < as the sorting criterion.
- The second form uses the binary predicate op(elem1,elem2) as the sorting criterion.
- Complexity: linear on average.
Sorted range algorithms

- binary_search: return whether the sorted range [beg,end) contains an element equal to value
- includes: return whether the sorted range [beg,end) contains all elements in the sorted range [searchBeg,searchEnd).
- lower_bound: returns the position of the first element that has a value less than or equal to value.
- upper_bound: returns the position of the first element that has a value greater than value.
- equal_range: return the range of elements that is equal to value
- merge: merge the elements of the sorted source ranges [source1Beg,source1End) and [source2Beg,source2End) so that the destination range starting with destBeg contains all elements that are in the first source range plus those that are in the second source range.
- set_union, set_intersection, set_difference, set_symmetric_difference