# Learning C++: Sorting Algorithms of the STL

The Standard Template Library (STL) has a small set of sorting algorithms you can use to sort container elements. The type of sorting algorithm used is implementation-dependent but the STL guarantees n-log-n (good) performance. In this article I'll demonstrate how to use the various sorting algorithms in the STL.

## The sort Function

The fundamental sorting function of the STL is sort. This function takes a range of container elements and sorts them. In the first version of the function, it sorts them using the < operator and in a second version, you can provide a binary predicate function to sort the elements.

Here are the syntax templates for the two versions of sort:

*void sort(range-start, range-end);*

*void sort(range-start, range-end, binary-predicate-function);*

Here is a program that demonstrates how the first version of sort works:

#include <iostream>

#include <vector>

#include <algorithm>

#include <cstdlib>

#include <ctime>

using namespace std;

void buildVec(vector<int> &vec, int n) {

for (int i = 1; i <= n; i++) {

vec.push\_back(rand() % 100 + 1);

}

}

void printVec(vector<int> &vec) {

for (const int n : vec) {

cout << n << " ";

}

}

int main () {

srand(time(0));

vector<int> numbers;

buildVec(numbers, 20);

printVec(numbers);

sort(numbers.begin(), numbers.end());

cout << endl << endl;

printVec(numbers);

return 0;

}

Here is the output from one run of this program:

66 19 34 94 8 53 2 46 61 30 14 65 8 19 23 95 45 42 41 34

2 8 8 14 19 19 23 30 34 34 41 42 45 46 53 61 65 66 94 95

As you can see, the elements of the vector are sorted into ascending order, as specified by the default action of the < operator.

If we want to sort this data into descending order, we can pass a third parameter to the function to change the order. For this example, we'll use the greater<>() function object. Here's the program:

int main () {

srand(time(0));

vector<int> numbers;

buildVec(numbers, 20);

printVec(numbers);

sort(numbers.begin(), numbers.end(), greater<int>());

cout << endl << endl;

printVec(numbers);

return 0;

}

Here is the output from one run of this program:

10 49 36 75 32 66 96 32 82 10 38 94 15 79 30 99 97 48 30 88

99 97 96 94 88 82 79 75 66 49 48 38 36 32 32 30 30 15 10 10

There are some constraints on using the sort function. The main constraint is that you cannot use the function on an associative container, such as a map, or on a list-based container such as the list or forward\_list since these container types don't provide the type of iterators needed to sort the elements in these types of containers.

## The stable\_sort Function

The next type of sort I want to discuss is the stable sort. A stable sort is one where the relative position of equal elements is kept the same from the original container to the sorted container. The function that achieves a stable sort is the stable\_sort function. This function can also take a binary predicate function as an optional argument. Here are the syntax templates for this function:

*void stable\_sort(range-begin, range-end);*

*void stable\_sort(range-begin, range-end, binary-predicate-function);*

The next program is an example of how the stable\_sort function retains the position of equal elements. The vector contains strings that are of different lengths and there is a function that compares the length of the strings, enforcing a less than ordering. First, we call the sort function:

bool length(const string &s1, const string &s2) {

return s1.length() < s2.length();

}

int main () {

vector<string> words = {"1xxx", "2x", "3x", "4x", "5xx",

"6xxxx", "7xx", "8xxx", "9xx",

"10xxx", "11", "12", "13", "14xx",

"15", "16", "17"};

vector<string> copied(words);

sort(words.begin(), words.end(), length);

cout << "Order with sort function: " << endl;

printVec(words);

return 0;

}

The output from this program is:

Order with sort function:

2x 17 16 15 13 12 11 4x 3x 9xx 7xx 5xx 8xxx 14xx 1xxx 10xxx 6xxxx

And here is the stable\_sort program:

bool length(const string &s1, const string &s2) {

return s1.length() < s2.length();

}

int main () {

vector<string> words = {"1xxx", "2x", "3x", "4x", "5xx",

"6xxxx","7xx", "8xxx", "9xx", "10xxx",

"11", "12","13", "14xx",

"15","16","17"};

vector<string> copied(words);

stable\_sort(words.begin(), words.end(), length);

cout << "Order with stable\_sort function: " << endl;

printVec(words);

return 0;

}

Here is the output from this program:

Order with stable\_sort function:

2x 3x 4x 11 12 13 15 16 17 5xx 7xx 9xx 1xxx 8xxx 14xx 6xxxx 10xxx

As you can see, the original order of the strings with the numbers first is kept with the stable\_sort function but is not kept with the sort function.

## Partial Sorting Algorithms

The algorithms I discussed above are used primarily to sort a full range of elements, such as a full container. This next function is used to perform a partial sort where only a specified subrange of a range is sorted.

The function that performs partial sorting is partial\_sort. Its arguments are the range start, the position to stop the partial sorting, and the full range end. A second version of partial\_sort allows a binary predicate function as a final argument.

Here are the syntax templates for partial\_sort:

*void partial\_sort(range-start, partial-range-end, range-end);*

*void partial\_sort(range-start, partial-range-end, range-end, binary-predicate-function);*

The following program demonstrates the first version of partial\_sort, with sorting stopping after the 10th element in the container. Here's the program:

int main () {

srand(time(0));

vector<int> numbers;

buildVec(numbers, 20);

printVec(numbers);

partial\_sort(numbers.begin(), numbers.begin() + 10,

numbers.end());

cout << endl << endl;

printVec(numbers);

return 0;

}

Here is the output from this program:

73 39 68 1 32 4 78 74 7 39 39 45 10 22 33 78 53 50 20 68

1 4 7 10 20 22 32 33 39 39 78 74 73 68 45 78 53 50 39 68

If you look closely, you will see that only the first ten elements of the vector were sorted, as specified in the function call.

Here is a program that uses the second version of partial\_sort to sort the first half of the vector into descending order:

int main () {

srand(time(0));

vector<int> numbers;

buildVec(numbers, 20);

printVec(numbers);

partial\_sort(numbers.begin(), numbers.begin() + 10,

numbers.end(), greater<int>());

cout << endl << endl;

printVec(numbers);

return 0;

}

Here is the output from this program:

33 61 50 27 24 71 1 88 12 13 30 75 47 44 34 97 100 93 77 91

100 97 93 91 88 77 75 71 61 50 1 12 13 24 27 30 33 34 44 47

There is also a function that does a partial sort and copies the sorted elements to another container. This function is called partial\_sort\_copy. It also has an overloaded version that accepts a binary predicate function as a last argument. Here are the syntax templates for this function:

*void partial\_sort\_copy(range-start, range-end, destination-range-begin, destination-range-end);*

*void partial\_sort\_copy(range-start, range-end, destination-range-begin, destination-range-end,*

*binary-predicate-function);*

The following program demonstrates how to use the first version of partial\_sort\_copy:

int main () {

srand(time(0));

vector<int> numbers;

vector<int> partSort(20);

buildVec(numbers, 20);

printVec(numbers);

partial\_sort\_copy(numbers.begin(), numbers.begin() + 10,

partSort.begin(), partSort.end());

cout << endl << endl;

cout << "Showing copied elements: " << endl;

printVec(partSort);

return 0;

}

The output from this program is:

21 46 29 92 10 52 24 29 32 10 29 70 18 33 74 18 46 9 62 94

Showing copied elements:

10 10 21 24 29 29 32 46 52 92

Only the sorted elements are copied into the new vector as shown by the output.

## Sorting to the *n*th Element

Sometimes you only need to see a certain range of the highest or lowest values in a container. The STL has a function to provide this view: nth\_element. The syntax templates for this function is:

*void nth\_element(range-start, iterator-nth-position, range-end);*

*void nth\_element(range-start, iterator-nth-position, range-end, binary-predicate-function);*

In this first program, I will display the four lowest values in a vector:

int main () {

srand(time(0));

vector<int> numbers;

buildVec(numbers, 20);

printVec(numbers);

cout << endl << endl;

nth\_element(numbers.begin(), numbers.begin()+3,

numbers.end());

cout << "The four lowest values are: " << endl;

copy(numbers.begin(), numbers.begin()+4,

ostream\_iterator<int>(cout, " "));

cout << endl;

return 0;

}

The output from this program is:

71 43 55 81 27 56 21 42 96 98 3 29 35 23 44 82 30 27 87 21

The four lowest values are:

21 3 21 23

## The Heap Algorithms

The last set of algorithms I will discuss in this article are the heap algorithms. A heap is a data structure that has several properties that make it useful for storing sorted data. Covering the details of the heap is beyond the scope of this article, but you can learn more about heaps [here](https://en.wikipedia.org/wiki/Heap_(data_structure)).

There are four functions that are part of the heap algorithms:

* make\_heap; converts a range of elements to a heap.
* push\_heap: add one element to a heap.
* pop\_heap: remove the next element from a heap.
* sort\_heap: convert a heap to a sorted collection.

Rather than cover the syntax templates for each of these functions, I will just provide a program that demonstrates how the primary versions of the functions work. Here's the program:

int main () {

srand(time(0));

vector<int> numbers = {4,1,2,18,9,6,11};

cout << "The vector before make\_heap: " << endl;

printVec(numbers);

make\_heap(numbers.begin(), numbers.end());

cout << endl << endl << "After make\_heap: " << endl;

printVec(numbers);

numbers.push\_back(15);

push\_heap(numbers.begin(), numbers.end());

cout << endl << endl << "After push\_heap: " << endl;

printVec(numbers);

pop\_heap(numbers.begin(), numbers.end());

cout << endl << endl << "After pop\_heap: " << endl;

printVec(numbers);

sort\_heap(numbers.begin(), numbers.end());

cout << endl << endl << "After sort\_heap: " << endl;

printVec(numbers);

return 0;

}

The output from this program is:

The vector before make\_heap:

4 1 2 18 9 6 11

After make\_heap:

18 9 11 1 4 6 2

After push\_heap:

18 15 11 9 4 6 2 1

After pop\_heap:

15 9 11 1 4 6 2 18

After sort\_heap:

1 2 4 6 9 11 18 15

You will notice that the data is not perfectly sorted after the sort\_heap function is called. I would not use these functions for your sorting purposes but I covered them to be complete.

## Next Up – Sorted-Range Algorithms

In my next article, I'll present a set of algorithms you can use on data that is already sorted. The most famous of these algorithms is binary search, but there are others as well.

Thanks for reading this article and please email me with comments and suggestions.