16.4 — STL algorithms overview

BY ALEX ON SEPTEMBER 11TH, 2011 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

In addition to container classes and iterators, STL also provides a number of generic algorithms for working with the elements of the container classes. These allow you to do things like search, sort, insert, reorder, remove, and copy elements of the container class.

Note that algorithms are implemented as functions that operate using iterators. This means that each algorithm only needs to be implemented once, and it will generally automatically work for all containers that provides a set of iterators (including your custom container classes). While this is very powerful and can lead to the ability to write complex code very quickly, it's also got a dark side: some combination of algorithms and container types may not work, may cause infinite loops, or may work but be extremely poor performing. So use these at your risk.

STL provides quite a few algorithms -- we will only touch on some of the more common and easy to use ones here. The rest (and the full details) will be saved for a chapter on STL algorithms.

To use any of the STL algorithms, simply include the algorithm header file.

min_element and max_element

The `std::min_element` and `std::max_element` algorithms find the min and max element in a container class. `std::iota` generates a contiguous series of values.

```
#include <algorithm> // std::min_element and std::max_element
#include <iostream>
#include <list>
#include <numeric> // std::iota

int main()
{
    std::list<int> li(6);
    // Fill li with numbers starting at 0.
    std::iota(li.begin(), li.end(), 0);

    std::cout << *std::min_element(li.begin(), li.end()) << ' '
              << *std::max_element(li.begin(), li.end()) << '\n';

    return 0;
}
```

Prints:

0 5

find (and list::insert)

In this example, we'll use the `std::find()` algorithm to find a value in the list class, and then use the list::insert() function to add a new value into the list at that point.

```
#include <algorithm>
#include <iostream>
#include <list>
#include <numeric>

int main()
{
    std::list<int> li(6);
    std::iota(li.begin(), li.end(), 0);
```

```
10
11      // Find the value 3 in the list
12      auto it{ find(li.begin(), li.end(), 3) };
13
14      // Insert 8 right before 3.
15      li.insert(it, 8);
16
17      for (int i : li) // for loop with iterators
18          std::cout << i << ' ';
19
20      std::cout << '\n';
21
22      return 0;
23  }
```

This prints the values

```
0 1 2 8 3 4 5
```

When a searching algorithm doesn't find what it was looking for, it returns the end iterator.
If we didn't know for sure that 3 is an element of `li`, we'd have to check if `std::find` found it before we use the returned iterator for anything else.

```
if (it == li.end())
{
  std::cout << "3 was not found\n";
}
else
{
  // ...
}
```

sort and reverse

In this example, we'll sort a vector and then reverse it.

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

int main()
{
    std::vector<int> vect{ 7, -3, 6, 2, -5, 0, 4 };

    // sort the vector
    std::sort(vect.begin(), vect.end());

    for (int i : vect)
    {
        std::cout << i << ' ';
    }

    std::cout << '\n';

    // reverse the vector
    std::reverse(vect.begin(), vect.end());

    for (int i : vect)
    {
        std::cout << i << ' ';
    }
```

```
26
27      std::cout << '\n';
28
29      return 0;
30  }
```

This produces the result:

```
-5 -3 0 2 4 6 7
7 6 4 2 0 -3 -5
```

Alternatively, we could pass a custom comparison function as the third argument to `std::sort`. There are several comparison functions in the <functional> header which we can use so we don't have to write our own. We can pass `std::greater` to `std::sort` and remove the call to `std::reverse`. The vector will be sorted from high to low right away.

Note that `std::sort()` doesn't work on list container classes -- the list class provides its own `sort()` member function, which is much more efficient than the generic version would be.

Conclusion

Although this is just a taste of the algorithms that STL provides, it should suffice to show how easy these are to use in conjunction with iterators and the basic container classes. There are enough other algorithms to fill up a whole chapter!

 **17.1 -- std::string and std::wstring**

 **Index**

 **16.3 -- STL iterators overview**