16.4 — STL algorithms overview

BY ALEX ON SEPTEMBER 11TH, 2011 | LAST MODIFIED BY ALEX ON JULY 22ND, 2017

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 global 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 min_element and max_element algorithms find the min and max element in a container class:

```
#include <iostream>
#include <list>
#include <algorithm>
int main()
{
    using namespace std;

    list<int> li;
    for (int nCount=0; nCount < 6; nCount++)
        li.push_back(nCount);

    list<int>::const_iterator it; // declare an iterator
    it = min_element(li.begin(), li.end());
        cout << *it << " ";
    it = max_element(li.begin(), li.end());
        cout << *it << " ";

    cout << endl;
}
```

Prints:

0 5

find (and list::insert)

In this example, we'll use the 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 <iostream>
#include <list>
#include <algorithm>
int main()
{
    using namespace std;

    list<int> li;
    for (int nCount=0; nCount < 6; nCount++)
        li.push_back(nCount);

    list<int>::iterator it; // declare an iterator
    it = find(li.begin(), li.end(), 3); // find the value 3 in the list
```

```
    li.insert(it, 8); // use list::insert to insert the value 8 before it

    for (it = li.begin(); it != li.end(); it++) // for loop with iterators
        cout << *it << " ";

    cout << endl;
}
```

This prints the value

0 1 2 8 3 4 5

sort and reverse

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

```
#include <iostream>
#include <vector>
#include <algorithm>
int main()
{
    using namespace std;

    vector<int> vect;
    vect.push_back(7);
    vect.push_back(-3);
    vect.push_back(6);
    vect.push_back(2);
    vect.push_back(-5);
    vect.push_back(0);
    vect.push_back(4);

    sort(vect.begin(), vect.end()); // sort the list

    vector<int>::const_iterator it; // declare an iterator
    for (it = vect.begin(); it != vect.end(); it++) // for loop with iterators
        cout << *it << " ";

    cout << endl;

    reverse(vect.begin(), vect.end()); // reverse the list

    for (it = vect.begin(); it != vect.end(); it++) // for loop with iterators
        cout << *it << " ";

    cout << endl;
}
```

This produces the result:

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

Note that 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**