STL Algorithms

Sanjeev Qazi

STL Algorithms

- STL provides a lot of algorithms that you should get familiar with so that when the need arises, you know its available in the STL library.

- This does not necessarily mean that you need to know all of them very well, you may know some of them well and some others not so well, but if you know they exist, then you can always look them up when the need arises.

- We are going to talk about some of them, but this is by no means an exhaustive list.
- As you go through this information, you will notice that the algorithms are based on iterators into the STL containers… but what you should also realize is that these don't have to be STL containers. You could write your own container class and provide the iterators on it, and then call any of the STL algorithm functions on your container. You would have to provide your own iterators, because if it is your container class, only you would know the implementation details for that container.

Function objects

- These are nothing but objects of a class that has overloaded the function call operator.
- We looked at these in the Operator overloading class.
- These are also known as *functors*. For example:

Function objects

```
class My2DCollection {
...
        Element &              operator() ( int row, int col );
        const Element &        operator() ( int row, int col ) const;
};

int main()
{
    My2DCollection m2;
    // add elements to m2.
    Element & e1 = m2( 0, 0 );          // get a reference to element at 0,0.
    // Or I could say :
    Element & e1 = m2.operator()( 0, 0 );

    // Remember, operators are just like member methods.
    // Think of above operator as if it were a member method called GetElement
    //  Element & e1 = m2.GetElement( 0, 0 );
}
```

STL Algorithms

Unary and binary functions

Unary functions are functions that take one argument, and binary functions take two arguments. The return type of these can be anything.

unary_predicate

Unary predicate is a type of unary function that takes one argument and returns a boolean if the passed in argument satisfies the criteria that predicate is checking for.

A simple example here would be a predicate that checks for the passed in argument being odd or even.

STL Algorithms

binary_predicate

Binary predicate is a type of binary function that takes two arguments and returns a boolean if the passed in arguments match the criteria that the binary predicate is checking for.

A simple example of a binary predicate is a lessThan comparison, or a greaterThan comparison.

STL Algorithms

generator_n

This function assigns the values by calling a user supplied function or functor into the iterator passed into generator_n.

See code example.

fill_n

Similar to generator_n , except it takes a specific value that you pass in, whereas for generator_n you have to supply a function or functor.

See code example.

equal

equal

This function simply returns true if the elements specified in the two ranges are "equal". There are two flavors of this function, one of them uses the == operator and the other one uses a binary predicate that is passed in to the equal function.

bool equal (startIter1, endIter1, startIter2);

bool equal (startIter1, endIter1, startIter2, binary_pred);

binary_pred is a functor (or can be a function pointer) that takes two elements and returns true or false, based on the code written inside the binary_pred(that you write inside the operator()).

See code example.

mismatch

mismatch

- Let v1 and v2 be two collections, say
- vector< int > v1, v2;

- mismatch is a function that will return the first position where elements of v1 and v2 are different. This is returned as a pair of iterators, first is an iterator into v1, second is an iterator into v2.

- bool areEqual = mismatch(v1.begin(), v1.end(), v2.begin()).first == v1.end();
- typedef pair< vector<int>::iterator, vector<int>::iterator > IterPair;

- IterPair iterPair = mismatch(v1.begin(), v1.end(), v2.begin());
- bool areEqual = iterPair.first == v1.end();

min, max

min and max

- These compare two elements (not ranges) and return the min and max respectively.
- There are two versions of min and max:
  - one version compares using the operator <
    - so operator < should be overloaded if you are using a class object.
  - the other version takes in a function object (i.e., an instance of the binary predicate).

min_element, max_element

min_element, max_element

- These functions return the minimum (maximum) element in the half open range [first, last) that is passed in.
- An iterator to that element is returned, and no other iterator in that range points to a smaller (larger) element.
- If all elements are equal, then an iterator to the first element is returned.
- Iterator to the last element is returned if the range is empty.

sort

- This algorithm sorts the elements in [first, last) into a non-descending order.

- This algorithm is not guaranteed to be a stable sorting algorithm… what that means is that if there are two elements x and y, where neither of them is less than the other, in other words, they are equivalent (depending on the comparison criteria), then the final sorted result is not guaranteed to have these two elements in the same relative order as they were in the input.

sort

Here's an example to clarify this:

- Let's say you have a vector of Student pointers, and you want to sort them based on their age.
- This means that two students having the same age are equivalent(based on our sorting criteria), but are not identical.
- So now, you would supply a sort comparison function or functor that would do the age comparison.
- Lets say that two students in this collection, studentA and studentB have the same age.
- And lets say in the input, studentA happens to be *before* studentB .
- The final sorted vector could have the studentB before studentA, because this sort algorithm is not a stable sort algrorithm.

- The complexity of this algorithm is O(n log n) on average.

stable_sort

- This is similar to sort algorithm, except:
  - sort is faster,
  - but stable_sort maintains the relative ordering of equivalent elements (if that is important in your situation).
  - An example where you may want to use this is this scenario:
    - you sort a collection of names based on the first name,
    - and then you do a stable_sort based on last name,
    - and that will maintain the first name ordering that you got in the first sort (for folks with the same last name).
    - However, as I write this, I am thinking that you could just use sort to sort on last name, and then use sort again to sort on first name. That way you would not need to use stable_sort in this scenario ☺.

is_sorted

is_sorted

This takes a range [first, last) and

- returns true if it is sorted,
- returns false otherwise.

binary_search

binary_search

These are algorithms that operate on sorted ranges.

// needs operator < to be defined for element's class.

binary_search (first, last, element);

// comp is the function/functor

binary_search (first, last, element , comp);

binary_search will return true if the element being searched for is present in the given sorted range.

lower_bound, upper_bound

// needs operator < to be defined for element's class.

lower_bound(first, last, element);

lower_bound(first, last, comp);

lower_bound returns an iterator to the element being searched for if it is present, if not present, the iterator points to the element that is not less than the search element. (Also look up the lower_bound method on map/multimap from the earlier lesson).

// needs operator < to be defined for element's class.

upper_bound(first, last, element);

lower_bound(first, last, comp);

upper_bound returns an iterator to the element that is not less than the element passed in to upper_bound.

find

- This algorithm does a linear search in the range [first, last)
- So, complexity is O (N).

- It returns:
  - iterator to the element if found,
  - else it returns the *last* iterator that was passed in to find.

find_if

- Similar to find in that it is linear complexity.

- It returns the first element in the range [first, last) which satisfies the unary predicate passed in to find_if.
- This is a generalization of the find algorithm.
- You could use this to do the same thing as a find by passing in a function/functor that will do an equality comparison against the element you are searching for.

- eqComparator is a functor whose constructor takes the element being searched for and compares it against each element from the range that will be passed to it by find_if.

- find_if(vec.begin(), vec.end(), eqComparator(valueBeingSearchedFor);

adjacent_find

- template <class FwdIt> FwdIt adjacent_find (FwdIt first, FwdIt last);
- template <class FwdIt> FwdIt adjacent_find (FwdIt first, FwdIt last, BinaryPredicate pred);

- This algorithm looks for two adjacent elements with the "same value".
- The first flavor will determine equality by using operator ==.
- Second flavor will determine equality using the predicate passed in.
- It returna an iterator to the first element of the two it finds to be equal (or satisfy the binary predicate passed in).

adjacent_find

- Think of this something like:

```
template <typename FwdIt>
FwdIt   adjacent_find (FwdIt   start, FwdIt   last)
{
        if (start != last)
        {
                FwdIt next = start;
                ++ next;
                while ( next != last)
                {
                        if ( * start == *next)
                                return start;
                        ++ next;  ++ start;
                }
        }
        return last;
}
```

copy

- The copy algorithm copies data from the half open range [first, last) to the destination provided.

- Example:

```
vector<int> vi1(10);
vector<int> vi2(10);

// Add elements to vi1. Then copy to vi2

copy ( vi1.begin(),  vi1.end(),  vi2.begin()  );
```

OR …

copy

```
vector<int> vi1(10);        // pre allocate 10 elements
vector<int> vi2;            // Notice, no pre allocation of space in this one

// Fill up vi1;
// Now, we didn't pre-allocate space in vi2,
// so we have to use back_inserter function, which
// will return a back_insert_iterator to let us append to vi2.

copy ( vi1.begin(), vi1.end(), back_inserter( vi2) ) ;
```

copy

```
list <int>  iList;
vec<int> iVec;

// fill up iVec
// use a front inserter to insert at beginning of the list

copy(  iVec.begin(),  iVec.end(),  front_inserter( iList )  );
```

- You can use front_inserter on a list or a deque.
- The container that you call front_inserter on has to have the member method push_front defined.
- So you cannot call this on a vector (in a vector, you can do push_back, but not a push_front, as that would be very inefficient since all elements would need to be shifted to the right).

accumulate

- See code example