

Sorting

```
#include <algorithm>
```

The algorithm file in C++ provide several common algorithms that are very useful. Most of these algorithms operate on a **range** of elements, which can be specified using iterators. The most common algorithm is perhaps **sort**. The following demonstrates how to sort a vector of integers.

Example 1:

```
vector<int> v;
v.push_back( 3 );
v.push_back( 1 );
v.push_back( 2 );
sort( v.begin(), v.end() ); // sort v in increasing order
```

The **sort** function takes two parameters, A and B, both iterators, and sorts the range [A, B) in $O(N \log N)$ time, in **increasing** order. The requirement is that A and B are both random access iterators, and so we cannot use **sort** to sort a list, for example. So, what if we want to sort in decreasing order, or to sort some special data structure? We can always define our own comparison function and use that.

Example 2:

```
struct object {
    int weight, cost;
};

struct ComparisonFunctor {
    bool operator() ( const object &a, const object &b ) const {
        if ( a.cost != b.cost ) return a.cost < b.cost;
        else return a.weight > b.weight;
    }
};

vector<object> vo;
// ... insert some objects to the vector

// A. sort the vector first by lowest cost, then by greatest weight
sort( vo.begin(), vo.end(), ComparisonFunctor() );

// B. we can also define a set using the functor.
// the following copies the vector to the set, which will be kept in
// sorted order (Binary Search Tree)
set< object, ComparisonFunctor > myset( vo.begin(), vo.end() );
```

We use a **function object**, or **functor** in the above example. This allows us to use it as a class in templated containers, like set, or simply as a comparison function. Note that in this case, we have to have a comparator because without it, the compiler will try to use the < operator to compare to **objects**, and since there is no such operator defined, the program will not compile. If you do not need the comparison function as a **functor**, then just create it as a normal function, like:

Example 3:

```
bool compare( const object &a, const object &b ) {
    if ( a.cost != b.cost ) return a.cost < b.cost;
    else return a.weight > b.weight;
}
```

```
// and to use it...
sort( vo.begin(), vo.end(), compare ); // no brackets this time
```

next_permutation()

```
#include <algorithm>
```

Another useful function in <algorithm> is next_permutation(). Once again, its two parameters are iterators defining a range [A, B). But first, we need a little background on permutations.

Given a finite set $S = \{1, 2, \dots, n\}$, there are $n!$ ways of listing the elements of the set on a line – $n!$ different orderings (or permutations) of elements. For example, for a set of 3 elements, there are $3! = 6$ different permutations: (1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2) and (3,2,1). Note that the 6 permutations are ordered lexicographically – from the smallest (a sorted list) to the largest (the list sorted backwards). We can extend the notion of a permutation to collections with duplicate elements. For example, there are 10 permutations of (1,1,2,2,2). They are (1,1,2,2,2), (1,2,1,2,2), (1,2,2,1,2), (1,2,2,2,1), (2,1,1,2,2), (2,1,2,1,2), (2,1,2,2,1), (2,2,1,1,2), (2,2,1,2,1) and (2,2,2,1,1). Permutations have many uses; especially, in brute-force algorithms where we need to enumerate all of the different orderings of a collection of elements.

The next_permutation() function takes a range that contains a permutation and modifies that range to contain the lexicographically next permutation.

Example 4:

```
vector< int > v;
v.push_back( 2 ); v.push_back( 2 ); v.push_back( 1 ); v.push_back( 2 );
next_permutation( v.begin(), v.end() );
```

The vector v starts out with (2,2,1,2). After executing the function, v will contain (2,2,2,1).

So what happens when the input range already contains the largest permutation? In this case, next_permutation() will wrap around to produce the smallest permutation again. It will also return false in this case. In all other cases, it returns true. Here is how we can print all the permutations of a collection.

Example 5:

```
vector< int > v;
// ...fill in v with some integers
sort( v.begin(), v.end() );
do {
    for( int i = 0; i < ( int )v.size(); i++ ) cout << " " << v[i];
    cout << endl;
} while( next_permutation( v.begin(), v.end() ) );
```

First, we need to sort v to get the smallest permutation. Then the do-while loop will repeat these two steps – print a permutation and produce the next one. When we reach the largest permutation, we will print it, then produce the smallest one again and exit the loop. Therefore, after the loop has finished, v 's elements are once again in sorted order.

There are other very useful functions in <algorithm>, like stable_sort() and lower_bound(). You can read about them here: <http://www.sgi.com/tech/stl>. They might come in handy.