STL Wiki - NSUPS

Vector

Vector is basically like an array. But we don't predefine the size of a vector at the beginning like what we do in array. We can insert the elements always without bothering about the size limit.

```
vector<int> vec; // Declaring a vector
vec.push_back(5); // Inserting element at the vector
vec.pop_back(); // Erasing the last element of the vector
cout << vec.size() // Printing the current size of the vector
for(int i = 0; i < vec.size(); i++){
     cout << vec[i] << endl; ///Printing the i'th element in
the vector (i is 0 based)
}
sort(vec.begin(), vec.end()); // Sort the elements of the vector
```

Pair

It couples together a pair of values, which may be of same or different data types. The individual values can be accessed through its public members first and second.

```
pair<int, double> pr;
pr = make_pair(22, 3.9); // Creates a pair and store in pr.
cout << pr.first; // Print the first element which is 22
cout << pr.second; // Print the second element which is 3.9
```

Sorting

There are some built in functions which can be used to sort some data structures.

How to sort an array?

```
int A[105];
sort(A, A+10); // Sort the elements from index 0 to 9 in
ascending order
sort(A+3, A+10); // Sort the elements from index 3 to 9 in
ascending order. All the remaining numbers stay same as before.
```

How to sort a vector?

```
vector<int> vec;
sort(vec.begin(), vec.end()); // Sort all the elements in the
vector in ascending order.
```

String

```
string str = ""; // Declaring an empty string
cout << str.size() << endl; /// printing the size of string

string a = "North"; // Declaring a string
string b = "South";
string c = a + b; // Concatenating two string, c = a + b =
"NorthSouth";

str.pop_back(); // Erasing the last character of string.
str.push_back( 'a' ); // Adding a character at the back of
string.

sort( str.begin() , str.end() ); // Sorting the whole string in
alphabetical order.
// So if str = "cab", after sorting str will become "abc".

reverse( str.begin() , str.end() ); // reversing the whole
string.
// So if str = "axy", after sorting str will become "yxa".
```

Set

Set always keeps the elements in sorted order. The elements in a set cannot be accessed by their index number. Set always keeps single occurrence of elements no matter how many times you insert them.
Let's say we want to insert the following numbers in a set: 2 , 4 , 3 , 1 , 5 , 2 , 4 , 1.
After all the insert operations are done our set will look like this: 1 , 2 , 3 , 4 , 5.

```
set<int> myset; // Declaring a set
myset.insert(5); // Inserting element at set
myset.erase(5); // Erasing an element from the set
cout << myset.size() << endl; /// printing the size of the set
myset.clear(); // making the set empty

// checking if a value is present in set or not.
if( myset.find( val ) != myset.end() ) {
     cout << "set contains the val" << endl;
} else {
     // set doesn't contain the element val
}

set <int> :: iterator it; // iterator to iterate over the set
for( it = myset.begin(); it != myset.end(); it++ ) {
     // printing all the elements in set
     cout << *it << endl;
}
```

The iterator is basically a pointer that points to an element in the set. `*it` gives you the value in that pointing location.

When `it = myset.end()` it is pointing to the end of the set. There's no element in that end location.
When `it = myset.begin()` it is pointing to the first element of the set.

Stringstream

Let's say you are given a string str where `str = "1 2 3 4 5"`. As you can see, str is basically a string that contains some space separated integers. Now you want to separate those numbers and store them in a vector named vec. We can do this using stringstream.

```
stringstream iss( str ); // Declaring stringstream. ( iss is
just a name, use any )
int num;
vector<int> vec;
while( iss >> num ) {
     cout << num << endl;
     vec.push_back(num);
}
```

The iss is basically selecting the space separated numbers one by one from the string str. So the while loop runs until iss finish selecting all the numbers. Whenever iss is selecting a number, we are saving it to the variable num and storing it in the vector vect inside the while loop.

If numbers are separated by comma(,) full stop(.) etc then just add conditions accordingly.

```
string str = "1,2,3,4,5";
while( iss >> num ) {
     if( iss.peek() == ',' || iss.peek() == '.' ) {
          iss.ignore();  // add conditions
     }
     cout << num << endl;
}
```

String to Integer conversion using stringstream. We'll convert the string s to an integer and store it to num.

```
string s = "789";
stringstream iss( s );
int num;
iss >> num;
printf("%d\n",num);
```

Integer to String conversion using stringstream. We'll convert the integer num to a string and store it to s.

```
int num = 456;
string s;
stringstream iss;
iss << num;
s = iss.str();
```

Note: You can use the same technique to convert between any pair of data types double/char/int/string.

Map

Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value. No two mapped values can have same key values.

Declaration :

```
map < typename1 , typename2 > M;
```

Here , typename1 is the Key value & typename2 is the Mapped value.

Example : Let's declare of map , where both key & mapped value are of int type.

```
map<int,int> MyMap;
MyMap[1] = 10;
MyMap[3] = 30;
MyMap[2] = 20;
```

```
Let's see how to print the key & mapped value using iterator.

map<int,int> :: iterator it; // declaring iterator
for( it = MyMap.begin(); it != MyMap.end(); it++ ) {
     cout << it->first << " , " <<it->second << endl; // key , mapped value
}
```

Map keeps its keys in sorted order. So , output will be like this :

1 , 10
2 , 20
3 , 30

Some Functions associated with Map:

begin() – Returns an iterator to the first element in the map
end() – Returns an iterator to the theoretical element that follows last element in the map
size() – Returns the number of elements in the map
empty() – Returns whether the map is empty
erase(iterator position) – Removes the element at the position pointed by the iterator
erase(const g)- Removes the key value 'g' from the map
clear() – Removes all the elements from the map

Stack

A *stack* is an abstract data type that serves as a collection of elements, with two principal operations:

- push, which adds an element to the top of the collection and
- pop, which removes the most recently added element in the collection that was not yet removed.

Stack follows LIFO - Last In First Out

```
stack <int> st;  // declaring a stack of integer type

st.push(10);  // Inserting an element on the top of the stack
st.push(15);
st.push(20);
// Now the stack has three element s: 20,15,10. (top -> bottom)

cout<<st.top()<<endl; // Get the top element from the stack
st.pop(); // Remove the top element from the stack
Cout << st.size() << endl; // Get the size of the stack;
```

Note: Before calling the pop() or top() you need to check the stack is empty or not, cause if the stack is empty and you call the top() or pop() you will get a runtime error (RE).

Queue

A queue is a particular kind of abstract data type or collection in which the entities in the collection are kept in order and the principle (or only) operations on the collection are the addition of entities to the rear terminal position, known as enqueue (push), and removal of entities from the front terminal position, known as dequeue (pop).

- enqueue, which adds an element to the end of the collection, and
- dequeue, which removes the front element in the collection that was not yet removed.

Queue follows *First-In-First-Out (FIFO)*

```
queue <int> q; // Declaring a queue of integer type

q.push(1); // Inserting element in the queue
q.push(2);
q.push(3);
```

```
Now the queue has three element : 1,2,3. (front -> rear)
cout << q.front() << endl; // Get the front element of the queue
cout <<  q.back() << endl; // Get the rear element of the queue

q.pop(); // Remove the front element from the queue
cout << q.size() << endl; // Get the size of the queue
```

Priority Queue

Priority Queue keeps the values in decreasing(non-increasing) order. Elements cannot be accessed by index number. We can always get the maximum value from a priority queue in log(N).

```
priority_queue < int > PQ; // Declaring a priority queue of int
type
PQ.push( 1 ); // Pushing elements into priority queue.
PQ.push( 2 );
PQ.push( 3 );
// So , priority queue now contains elements in order : 3 2 1

cout << PQ.size() << endl; // Printing size of priority queue

if( PQ.empty() ) cout << " Empty " << endl; // checking it's
empty or not
else cout << "Not Empty" << endl;

cout << PQ.top() << endl; // Printing the top (largest) value
but not popping it

while( !PQ.empty() ) { // Printing all the elements.
     cout << PQ.top() << endl;
     PQ.pop(); // Removing the top value out of the priority
queue
}
```

Unordered Map and Unordered Set

Unordered map is almost same as map and unordered set is almost same as set except they don’t keep the elements in sorted order.