**Strings**

* Strings are containers that hold characters in it.**Initialization:**
* Assigning values directly to the string.

Syntax 1:

| string s; s = "ACM"; |
| --- |

Syntax 2 (while declaration)  
 string s ( "ACM" );

* Assigning single value to specific size of the string:  
  Takes two parameters first one is the size (number of indexes you want to assign the value to, the second one is the value itself.

Having string like “Hambozo1” we deal with all of them as characters all in place even integer 1 as long as it is used in a string we deal with it as a char not an int.

string s ( 5 , '\*' );  
 Or you can replace the character \*value\* with its corresponding integer value

In ASCII table.

| string s ( 5 , 42 ); |
| --- |

Both syntaxes will give the output as follows:  
 **\*\*\*\*\***

## **Input:** Strings are like containers that holds multiple characters. So we can either enter them as a whole:

| String s; cin>>s; |
| --- |

Or given the number of characters (which represents the size of the string) we can enter them one by one:

| int n ; *// size of the string*  cin >> n ;  string s( n , ' ' ) ;  for (int i=0 ; i<n ; i++)   cin >> s[i] ; |
| --- |

**Output:**

As a whole:

| string s ;  s = "ACM" ;  cout << s ; |
| --- |

Or using the string size:

| string s ;  s = "acmASCIS" ;  for (int i=0 ; i<s.size() ; i++) {  cout << s[i];  } |
| --- |

**String concatenation:**

* We can merge two or more strings using the plus (+) sign.

| string s = "MAD" , m = "MAX";  string x = s + " " + m;  cout << x; |
| --- |

The output for this is MAD MAX .

**Some other functions:**

* **Size**  
  Returns the number of characters in the string.

| string s = "Hanafy" ;  cout << s.size() ; |
| --- |

The output will be 6.

* **Find**  
   We use it to find some character or a substring in a string.

Returns integer value represents the index where this character or substring  
 was found first.

**Syntax:**

| string s = "F.R.I.E.N.D.S" ;  int x = s.find( '.' )  cout << x ; |
| --- |

The output will be 1 (as the strings like the arrays are zero based).  
 If the character/substring you are searching for wasn’t found it returns -1.

* **Insert**

We can use it to insert some string or some character to the original string.  
 It has many syntaxes with many uses.  
 Syntax 1:

(we give the function \*as parameters\*the position to be inserted in   
 and the string to be inserted).

| string s = "Al3abLeeh" ;  s.insert ( 5 , "Ba" ) ;  cout << s ; |
| --- |

The output will be Al3abBaLeeh.

* There are also more built in functions that makes strings more flexible,  
  You can find them all in the link below  
  <http://www.cplusplus.com/reference/string/string/>

**Vectors**

vectors are just like arrays but can change in size and have more functionalities.

**Initialization:**

* Declaring an empty vector.

| vector <int> vec; |
| --- |

* Declaring a vector with an initial size.

| vector <int> vec2(5) ; |
| --- |

**Input:**

* To insert an element at the end of the vector we use push\_back function.

| *vec.push\_back(1);  //vec elements now are : 1  vec.push\_back(2);  //vec elements now are : 1 2* |
| --- |

* Also we can insert elements in vectors like arrays using the index , but the

Vector must have an initial size bigger than or equal needed elements in insert.

| for(int i=0; i<5; i++)  cin>>vec2[i]; |
| --- |

**Output :**

* We can print all vector elements like any other normal array.

| for(int i=0; i<vec2.size(); i++)  cout<<vec2[i]<<" "; |
| --- |

**pop\_back/back:**

* If we want to know what is the last current value of our vector we can use back function.

| *// vec:1 2*  cout<<vec.back(); *// this will print 2* |
| --- |

* To remove the last element in a vector we use pop\_back function.

| *//vec: 1 2*  vec.pop\_back();  *// vec: 1* |
| --- |

**Clear:**

* Clear function is used to erase all elements in a vector we return its size to 0.

| vec.clear(); *//erase all elements* |
| --- |

**Stack**

**What’s the stack ?**

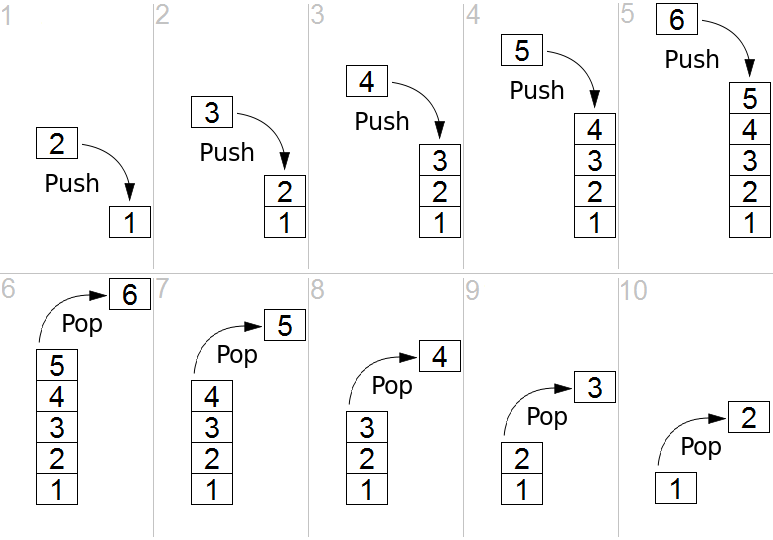
Stack is a data structure designed to operate in LIFO (Last in First out) context. In stack elements are inserted as well as get removed from only one end.

**Definition**

| stack< DATA TYPE > name ; |
| --- |

**LIFO**

In stack the first element inserted is the last element deleted.



**Stack Functions**

name.top(); It's the only way to access the stack elements, it gets a reference to the top element of the stack, does not pop it.

name.push(); Push an element onto the stack.

name.pop(); Pop off the top element of the stack.

name.size(); Get number of elements.**How to iterate over stack**

We can iterate over stack using the functions top() and pop().

| while (!name.empty()){ DATA TYPE Top = name.top(); name.pop(); } |
| --- |

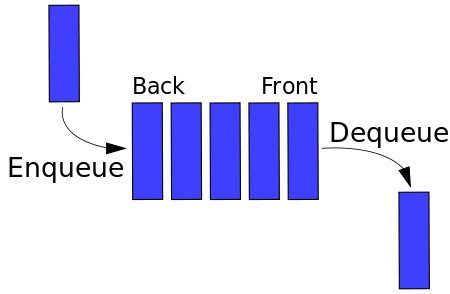
**Example**

| stack < int > stk ;  stk.push(5); *// insert an element with value 5 to the top of the stack*  stk.push(3); *// insert an element with value 3 to the top of the stack*  stk.push(2); *// insert an element with value 2 to the top of the stack*   cout << stk.top() << endl; *// print the top of stack ( 2 at this moment )*   stk.push(18); *// insert an element with value 18 to the top of the stack*  while (!stk.empty()){ // print the top of the stack and delete it int Top = stk.top();  cout << Top << endl;  stk.pop(); }  // the output of the previous loop is : 18  2  // 3  // 5 |
| --- |

**Queue**

**FIFO:**

* A queue is a linear data structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO).
* Elements are inserted into one end of the container and extracted from the other.



**Push (Enqueue):**

Inserts a new element at the end of the queue, after its current last element.

**Syntax:**

| queue<int> myque; myque.push(1); myque.push(2); *//queue: 1, 2* |
| --- |

**Front / Back:**

Front: returns a reference to the next element in the queue( the oldest element).

Back: returns a reference to the last element in the queue( the newest element).

**Syntax:**

| queue<int> myque; myque.push(1);  myque.push(2); myque.push(3); *//queue: 1, 2, 3* cout<< myque.front() << ' ' << myque.back()<< endl; *//output: 1 3* myque.front() = 0; myque.back() = 4; *//queue: 0, 2, 4* cout<< myque.front() << ' ' << myque.back()<< endl; *//output: 0 4* |
| --- |

**Pop (Dequeue):**

Removes the next element in the queue( the oldest element).

**Syntax:**

| queue<int> myque; myque.push(1); myque.push(2); *//queue: 1, 2* cout << myque.front() << endl; *//output: 1* myque.pop(); *//queue: 2* cout << myque.front() << endl; *//output: 2* |
| --- |

**Iterating over a Queue:**

We only have access to the front and back elements of the queue. To iterate over the queue elements, we can access the front element then pop it from the queue to access the next element.

**Syntax:**

| queue<int> myque; for(int i = 1; i <= 3; i++)  myque.push(i);  *//queue: 1, 2, 3* |
| --- |
| while(myque.size() != 0) *//loop until container is empty* {  cout << myque.front() << endl;  myque.pop(); } *//output: 1 2 3* *//after 1st iteration queue: 2, 3* *//after 2nd iteration queue: 3* *//after 3rd iteration queue:* |
|  |

**| Deque :**

**What’s the deque?**

**The deque is a container where we can add and remove elements from the front or the back but we can’t access an element in between**

**Declaration :**

You have to include the library of deque

#include <deque>

deque<int>dq; *//You can also replace int with any data type*

**.push\_front()**

To add elements to the front of the deque

| dq.push\_front(1); dq.push\_front(2); |
| --- |

**.front()**

| Cout << dq.front() << endl; *//the output is : 2* |
| --- |

**.push\_back()**

To add elements to the back of the deque

| dq.push\_back(3); dq.push\_back(4); |
| --- |

**.back()**

| Cout << dq.back() << endl; *//the output is : 4* |
| --- |

**.pop\_front()**

Removes the element at the front

| dq.pop\_front(); Cout << dq.front() << endl; *//the output is*  *: 1* |
| --- |

**.pop\_back()**

Removes the element at the end

| dq.pop\_back(); cout << dq.back() << endl; *//the output is : 3* |
| --- |

**.clear() & .size()**

Removes all elements ( clears the deque)



| dq.clear(); Cout << dq.size() << endl; *//the output is : 0* |
| --- |

**| Priority queue**

* **Declaration:**

You should mention the header of the priority queue class:

| #include <queue> *//use queue to include priority queue class in the cpp file* |
| --- |

Then you declare it as follows:

| priority\_queue<int> pq; *//any data type could be used* |
| --- |

* **Why priority queue:**

It’s used as a container for values but what makes it unique that it stores these values in a manner that it’ll always make the top of it be the biggest value among all the values you’ve pushed.

* **How to use priority queue:**

1. ***Push:***

| pq.push(1), pq.push(3), pq.push(2); |
| --- |

1. ***Top:***

| cout<<pq.top(); *//Output: 3* |
| --- |

1. ***Pop:***

| pq.pop(), cout<<pq.top(); *//Output: 2* |
| --- |

1. ***Empty:***

| if(pq.empty()) *//This returns true if it is empty or false if it is not* |
| --- |

* **How efficient is it:**

Due to the way it has been built the push and pop functions work in O(log(N)) while the top function works in O(1).

* **Can i make its top always has the smallest value:**

Yes, you can use greater<datatype> :

| priority\_queue<int, vector<int>, greater<int>> pq; *//change int with the datatype you want to use* |
| --- |

**| Map**

**What is a map ?!**

Maps are containers that store elements formed by a combination of a ***key*** and a ***value***.

Imagine the map as an array…. The ***key*** of the map is like the ***index*** of the array the only difference is that the index is always an integer number while the key can be of any data type (int, char, string, bool).

**Intro to binary search trees :**

A Binary search tree is a tree that consists of a number of nodes each node has two children ***at most***... left child and right child.

the value of the ***left*** child is ***always smaller*** than the value of its parent node while the value of the ***right*** child is ***always greater*** than that of its parent node.

**Declaration :**

Map < datatype, datatype > mapname ;

| map<int,int> mp; *// a map of key int and value int.* map<string,int> mp; *// a map of key string and value int.* |
| --- |

**Input / output :**

| *map<int,int> mp; mp[1]=7; // assigns the value 7 to the key 1 mp[4]++; // increments the value of the key 4*  *cout<<mp[1]<<endl; // outputs 7 cout<<mp[4]<<endl; // outputs 1 cout<<mp[2]<<endl; // outputs 0*  *map<string,int>mp2; mp2["mohamed"]=123456; mp2["ahmed"]++; cout<<mp2["mohamed"]<<endl; // outputs 123456 cout<<mp2["ahmed"]<<endl; // outputs 1 cout<<mp2["jack"]<<endl; // outputs 0* |
| --- |

**Clear:**

| map<int,int> mp; mp.clear(); *// deletes all the data in the map* |
| --- |

**| Set**

Sets are containers that store unique elements following an increasing order.

**Declaration :**

set < datatype > set\_name ;

| set<int>st; |
| --- |

**Insert :**

set<int>st;  
st.insert(5);

**Functions associated with Set:**

begin() – Returns an iterator to the first element in the set

end() – Returns an iterator to the theoretical element that follows the last element in the set

size() – Returns the number of elements in the set

max\_size() – Returns the maximum number of elements that the set can hold

empty() – Returns whether the set is empty

**Intro to Iterators :**

An *iterator* is any object that, pointing to some element in a range of elements (such as an array or a container), has the ability to iterate through the elements of that range using a set of operators.

**How to Iterate in Set :**

| **set<int>st ;**  **st.insert(5); st.insert(3); st.insert(2); st.insert(1); *// note that set is auto sort the elements*  set<int>::iterator it; *// this is the iterator*  for(it=st.begin() ; it!=st.end() ; it++) *// loop from start to end set*   cout<<\*it<<endl; *// \* before our iterator to show his value*** |
| --- |

**| Prime factorization**

**Idea:** Every number consists of prime numbers multiplied by each other.   
e.g: 28 = 2 x 2 x 7  
 9 = 3 x 3 iterate over all numbers from 2 to N, if we found a number i that divides N and is a prime, we consider i to be a prime factor of N.

| for(int i = 2; i < N; i++)   {   bool isPrime = true;   for(int j = 2; j\*j <= i; j++)   {   if(i%j == 0)   {   isPrime = false;   break;   }   }   while(N%i == 0 && isPrime)   {   st.insert(i);   N /= i;   }   } |
| --- |

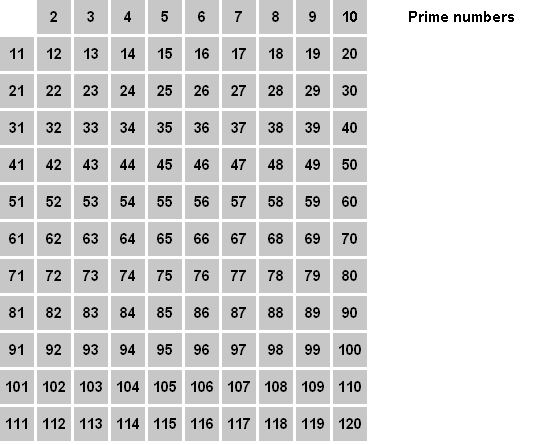
* What if we want to get the prime factors of multiple numbers?

= For every query, iterate with i over all the numbers from 2 to N, if it divides N and **is a prime**, insert it in the set of prime factors.

We can do better.

**| Sieve of Eratosthenes**Precalculate the prime numbers once at the beginning of a program.

**Idea:** if a number X is prime, then all multiples of X are not prime.  
e.g: 3 is prime, so 6, 9, 12, 15, 18, 21.. are not.



| bool primes[1000005]; int n = 1000000;  memset(primes, true, sizeof primes); for(int i = 2; i\*i <= n; i++) {  if(primes[i]) {  for(int j = i\*i; j <= n; j += i)  primes[j] = false;  } } |
| --- |
|  |