CUET COMPUTER CLUB
A doc for C++ STL

List of content:

Vector:

What is a vector?

Vector is a container like array but it can change its size. We don't need to initialize its size. Just like array it use continuous location on memory.

What Library function we need to include to use vector?

#include<vector>

What we can keep in a vector?

We can keep all basic data type like character, integer, long long integer,float,double etc. We can also store string, pair, structure etc.

How to declare a vector?

vector<data type>name;

let we will declare a vector names vp that will store integer type data, then we will write

```
vetor<int>vp;
```

vector of string:

```
vector<string>vp;
```

vector of structure:

if our structure name is st then

```
vector<st>vp;
```

Can we declare multi dimentional vector?

Yes,we can. A two dimentional vector declaration is given below.

```
vector<int>vp[100];
```

This will create 100 vector names vp[0], v[1],vp[2],……..vp[99].

Or you can declare like this:

```
vector< vector<double> > matrix;
```

it will declare a 2d vector of size 0*0.

We can simply access or use them as like 2d array. i.e.

```
printf("%d ",vp[i][j]):
```

Can we initialize vector as we memset a array?

Yes, we can.

```
vector<int>vp(50,100);
```

This will create a vector of size 50 filling each index with 100.

Another way to initialize 2d vector:

```
#include<bits/stdc++.h>
using namespace std:
int main()
{
    //      freopen("output.txt","w",stdout);
    //      freopen("input.txt","r",stdin);
   //ios_base::sync_with_stdio(false);
   int a,b,c,d,h,m,n,p,x,y,i,j,k,l,q,r,t,cnt,sm,tmp;
```

```
  int num_of_row = 6;
int num_of_col = 9;
double init_value = 3.14;
vector< vector<double> > matrix;
//now we have an empty 2D-matrix of size (0,0). Resizing it with one single command:
matrix.resize( num_of_row , vector<double>( num_of_col , init_value ) );
// and we are good to go ...
for(i=0;i<num_of_row;i++)
{
    for(j=0;j<num_of_col;j++)
    pf("%lf ",matrix[i][j]);
pf("\n");
}
    return 0;
}
```

What operator we can use in vector?

We can use = operator to copy one vector to another and == to check either two vector are equals or not. i.e.

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
vector<int>v1,v2,v3,v4;
v1.push_back(1);
v1.push_back(2);

v2=v1;  //copying vector v1 into v2
for(i=0;i<v2.size();i++)
    printf("%d ",v2[i]);
```

```
printf("\n");
if(v1==v2)     //checking either two vector are equals or not.
    printf("Yes\n");
else
    printf("no\n");

    v2.push_back(5);
if(v1==v2)
    printf("Yes\n");
else
    printf("no\n");
    return 0;
}
```

Some Member function of vector:

Modifier functions:

assign()

Assigns new contents to the vector, replacing its current contents, and modifying its size accordingly.

```
// vector assign
#include <iostream>
#include <vector>
using namespace std:
int main ()
{
  vector<int> first;
  vector<int> second;
  vector<int> third;

  first.assign (7,100);             // 7 ints with a value of 100

  vector<int>::iterator it;
  it=first.begin()+1;

  second.assign (it,first.end()-1); // the 5 central values of first

  int myints[] = {1776,7,4};
  third.assign (myints,myints+3);   // assigning from array.
```

```
  cout << "Size of first: " << int (first.size()) << '\n';
  cout << "Size of second: " << int (second.size()) << '\n';
  cout << "Size of third: " << int (third.size()) << '\n';
  return 0;
}
```

push_back()

Adds a new element at the end of the vector, after its current last element. The content of *val* is copied (or moved) to the new element.

This effectively increases the container size by one, which causes an automatic reallocation of the allocated storage space if -and only if- the new vector size surpasses the current vector capacity.

```
// vector::push_back
#include <iostream>
#include <vector>
using namespace std:
int main ()
{
  vector<int> myvector;
  int myint;

  cout << "Please enter some integers (enter 0 to end):\n";

  do {
    cin >> myint;
    myvector.push_back (myint);
  } while (myint);//this loop will break when you will enter 0.

  cout << "myvector stores " << int(myvector.size()) << " numbers.\n";

  return 0;
}
```

pop_back()

Removes the last element in the vector, effectively reducing the container size by one. This destroys the removed element.

```
// vector::pop_back
#include <iostream>
#include <vector>
using namespace std;
int main ()
{
  vector<int> myvector;
  int sum (0);
  myvector.push_back (100);
  myvector.push_back (200);
  myvector.push_back (300);

  while (!myvector.empty())
```

```
  {
    sum+=myvector.back();
    myvector.pop_back();
  }
  cout << "The elements of myvector add up to " << sum << '\n';
  return 0;
}
```

insert()

Insert elements

The vector is extended by inserting new elements before the element at the specified position, effectively increasing the container size by the number of elements inserted.

This causes an automatic reallocation of the allocated storage space if -and only if- the new vector size surpasses the current vector capacity.

```
// inserting into a vector
#include <iostream>
#include <vector>
using namespace std:
int main ()
{
  vector<int> myvector (3,100);
  vector<int>::iterator it;

  it = myvector.begin();
  it = myvector.insert ( it , 200 );

  myvector.insert (it,2,300);

  // "it" no longer valid, get a new one:
  it = myvector.begin();

  vector<int> anothervector (2,400);
  myvector.insert (it+2,anothervector.begin(),anothervector.end());

  int myarray [] = { 501,502,503 };
  myvector.insert (myvector.begin(), myarray, myarray+3);

  cout << "myvector contains:";
  for (it=myvector.begin(); it<myvector.end(); it++)
    cout << ' ' << *it;
  cout << '\n';

  return 0;
}
```

erase()

Erase elements
Removes from the vector either a single element (position) or a range of elements ([first,last)).

This effectively reduces the container size by the number of elements removed, which are destroyed.

```
// erasing from vector
#include <iostream>
#include <vector>

using namespace std:
int main ()
{
  vector<int> myvector;

  // set some values (from 1 to 10)
  for (int i=1; i<=10; i++) myvector.push_back(i);

  // erase the 6th element
  myvector.erase (myvector.begin()+5);

  // erase the first 3 elements:
  myvector.erase (myvector.begin(),myvector.begin()+3);

  cout << "myvector contains:";
  for (unsigned i=0; i<myvector.size(); ++i)
    cout << ' ' << myvector[i];
  cout << '\n';

  return 0;
}
```

swap()

Swap content
Exchanges the content of the container by the content of x, which is another vector object of the same type. Sizes may differ.

After the call to this member function, the elements in this container are those which were in x before the call, and the elements of x are those which were in this. All iterators, references and pointers remain valid for the swapped objects.

Notice that a non-member function exists with the same name, swap, overloading that algorithm with an optimization that behaves like this member function.

```
// swap vectors
#include <iostream>
#include <vector>

using namespace std:
int main ()
{
  vector<int> foo (3,100);   // three ints with a value of 100
  vector<int> bar (5,200);   // five ints with a value of 200

  foo.swap(bar);
```

```
  cout << "foo contains:";
  for (unsigned i=0; i<foo.size(); i++)
    cout << ' ' << foo[i];
  cout << '\n';

  cout << "bar contains:";
  for (unsigned i=0; i<bar.size(); i++)
    cout << ' ' << bar[i];
    cout << '\n';

  return 0;
}
```

clear()

Clear content
Removes all elements from the vector (which are destroyed), leaving the container with a size of 0.

```
// clearing vectors
#include <iostream>
#include <vector>
using namespace std:
int main ()
{
  std::vector<int> myvector;
  myvector.push_back (100);
  myvector.push_back (200);
  myvector.push_back (300);

  cout << "myvector contains:";
  for (unsigned i=0; i<myvector.size(); i++)
    cout << ' ' << myvector[i];
  cout << '\n';

  myvector.clear();
  myvector.push_back (1101);
  myvector.push_back (2202);

  cout << "myvector contains:";
  for (unsigned i=0; i<myvector.size(); i++)
    cout << ' ' << myvector[i];
  cout << '\n';

  return 0;
}
```

Element acess:

Those functions will acess element from a vector.

Vectorname[ ]

Returns a reference to the element at position n in the vector container.

```
// vector::operator[]
#include <iostream>
#include <vector>
using namspace std:
int main ()
{
  vector<int> myvector (10);   // 10 zero-initialized elements

  vector<int>::size_type sz = myvector.size();

  // assign some values:
  for (unsigned i=0; i<sz; i++) myvector[i]=i;

  // reverse vector using operator[]:
  for (unsigned i=0; i<sz/2; i++)
  {
    int temp;
    temp = myvector[sz-1-i];
    myvector[sz-1-i]=myvector[i];
    myvector[i]=temp;
  }

  cout << "myvector contains:";
  for (unsigned i=0; i<sz; i++)
   cout << ' ' << myvector[i];
  cout << '\n';

  return 0;
}
```

at()

Access element

Returns a reference to the element at position n in the vector.

```
// vector::at
#include <iostream>
#include <vector>
using namspace std:
int main ()
{
  vector<int> myvector (10);   // 10 zero-initialized ints

  // assign some values:
  for (unsigned i=0; i<myvector.size(); i++)
    myvector.at(i)=i;

  cout << "myvector contains:";
  for (unsigned i=0; i<myvector.size(); i++)
    cout << ' ' << myvector.at(i);
  cout << '\n';

  return 0;
```

```
}
```

front()

Access first element
Returns a reference to the first element in the vector.

```
// vector::front
#include <iostream>
#include <vector>
using namespace std:
int main ()
{
  vector<int> myvector;

  myvector.push back(78);
  myvector.push_back(16);

  // now front equals 78, and back 16

  myvector.front() -= myvector.back();

  cout << "myvector.front() is now " << myvector.front() << '\n';

  return 0;
}
```

back()

Access last element
Returns a reference to the last element in the vector.

```
// vector::back
#include <iostream>
#include <vector>

using namespace sdt:

int main ()
{
  vector<int> myvector;

  myvector.push_back(10);

  while (myvector.back() != 0)
  {
    myvector.push_back ( myvector.back() -1 );
  }

  cout << "myvector contains:";
  for (unsigned i=0; i<myvector.size() ; i++)
    cout << ' ' << myvector[i];
```

```
  cout << '\n';

  return 0;
}
```

Size related:

size()

Return size
Returns the number of elements in the vector.

```
// vector::size
#include <iostream>
#include <vector>

using namespace std:

int main ()
{
  vector<int> myints;
  cout << "0. size: " << myints.size() << '\n';

  for (int i=0; i<10; i++) myints.push_back(i);
   cout << "1. size: " << myints.size() << '\n';

  myints.insert (myints.end(),10,100);
  cout << "2. size: " << myints.size() << '\n';

  myints.pop_back();
  cout << "3. size: " << myints.size() << '\n';

  return 0;
}
```

max_size()

Return maximum size
Returns the maximum number of elements that the vector can hold.

```
// comparing size, capacity and max_size
#include <iostream>
#include <vector>

using namspace std:

int main ()
{
  vector<int> myvector;

  // set some content in the vector:
  for (int i=0; i<100; i++)
        myvector.push_back(i);
```

```
cout << "size: " << myvector.size() << "\n";
  cout << "capacity: " << myvector.capacity() << "\n";
  cout << "max_size: " << myvector.max_size() << "\n";
  return 0;
}
```

resize()

Change size

Resizes the container so that it contains n elements.

If n is smaller than the current container size, the content is reduced to its first n elements, removing those beyond (and destroying them).

If n is greater than the current container size, the content is expanded by inserting at the end as many elements as needed to reach a size of n. If val is specified, the new elements are initialized as copies of val, otherwise, they are value-initialized.

f n is also greater than the current container capacity, an automatic reallocation of the allocated storage space takes place.

```
// resizing vector
#include <iostream>
#include <vector>

using namespace std:

int main ()
{
  vector<int> myvector;

  // set some initial content:
  for (int i=1;i<10;i++) myvector.push_back(i);

  myvector.resize(5);
  myvector.resize(8,100);
  myvector.resize(12);

  cout << "myvector contains:";
  for (int i=0;i<myvector.size();i++)
    cout << ' ' << myvector[i];
  cout << '\n';

  return 0;
}
```

capacity()

Return size of allocated storage capacity
Returns the size of the storage space currently allocated for the vector, expressed in terms of elements.

```
// comparing size, capacity and max_size
#include <iostream>
#include <vector>
using namespace std:

int main ()
{
  vector<int> myvector;

  // set some content in the vector:
  for (int i=0; i<100; i++) myvector.push_back(i);

  cout << "size: " << (int) myvector.size() << '\n';
  cout << "capacity: " << (int) myvector.capacity() << '\n';
  cout << "max_size: " << (int) myvector.max_size() << '\n';
  return 0;
}
```

empty()

Test whether vector is empty

Returns whether the vector is empty (i.e. whether its size is 0).

It returns true if the container size is 0, false otherwise.

```
// vector::empty
#include <iostream>
#include <vector>

using namespace std:

int main ()
{
  vector<int> myvector;
  int sum (0);

  for (int i=1;i<=10;i++) myvector.push_back(i);

  while (!myvector.empty())
  {
     sum += myvector.back();
     myvector.pop_back();
  }

  cout << "total: " << sum << '\n';

  return 0;
}
```

Itterator functions:

begin()

Return iterator to beginning
Returns an iterator pointing to the first element in the vector.

Notice that, unlike member vector::front, which returns a reference to the first element, this function returns a random access iterator pointing to it.

If the container is empty, the returned iterator value shall not be dereferenced.

```
// vector::begin/end
#include <iostream>
#include <vector>

using namespace std:

int main ()
{
  vector<int> myvector;
  for (int i=1; i<=5; i++) myvector.push_back(i);

  cout << "myvector contains:";
  for (std::vector<int>::iterator it = myvector.begin() ; it != myvector.end(); ++it)
    cout << ' ' << *it;
  cout << '\n';

  return 0;
}
```

end()

Return iterator to end
Returns an iterator referring to the past-the-end element in the vector container.

The past-the-end element is the theoretical element that would follow the last element in the vector. It does not point to any element, and thus shall not be dereferenced.

```
// vector::begin/end
#include <iostream>
#include <vector>

using namespace std:

int main ()
{
  vector<int> myvector;
  for (int i=1; i<=5; i++) myvector.push_back(i);

  cout << "myvector contains:";
  for (vector<int>iterator it = myvector.begin() ; it != myvector.end(); ++it)
    cout << ' ' << *it;
  cout << '\n';
```

```
    return 0;
}
```

Special function for vector.

If vp is a vector then number of elements equals d in the range k-1.

```
equal_range(vp.begin(),vp.begin()+k,d);
```

⟶ Map:

What is map?

Maps are associative containers that store elements formed by a combination of a *key value* and a *mapped value*, following a specific order.

map containers are generally slower than unordered_map containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.

The mapped values in a map can be accessed directly by their corresponding key using the *bracket operator* ((operator[HYPERLINK "http://www.cplusplus.com/map::operator%5b%5d"]).

Maps are typically implemented as *binary search trees*.

What a map can contain?

A map can contain all basic types of data along and also string,pair ,structure, etc. The main advantage of map we can use any type of variable as index. That means we can use string, structure,double ,character etc as index.

What library function need to include to use map?

#include<map>

How to declare a map?

map<types_of_index,types_of_stored_value >map_name;

i.e.
To store integer in integer index,

map<int,int>mp;

to store in string index.

map<string,int>mp;

to store string in string index,
map<string,string>mp;

etc.

How to declare multi_dimentional map?

For 2d map,
map<int,map<int,int> >mp;
this can be use like 2d array.

For 3d map,
map<int,map<int,map<int,int> > >mp;

don't miss space between > > in multidimentional map.

How to copy one map to another?

Just use = operator.

mp1=mp2; //copying mp2 into mp1

Some member function of Map:

1.Modifier:

mp[]

This is not a member function it is operator. We can keep elements in a map using this like array.

map[21]=324;

map['d]=34123;

map[""abcd""]=""English_alphabet"";

clear()

Clear content

Removes all elements from the map container (which are destroyed), leaving the container with a size of 0.

```
// map::clear
#include <iostream>
#include <map>

using namespace std;

int main ()
{
  map<char,int> mymap;
```

```
  mymap['x']=100;
  mymap['y']=200;
  mymap['z']=300;

  cout << "mymap contains:\n";
  for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
    cout << it->first << " => " << it->second << '\n';

  mymap.clear();
  mymap['a']=1101;
  mymap['b']=2202;

  cout << "mymap contains:\n";
  for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
    cout << it->first << " => " << it->second << '\n';

  return 0;
}
```

2.Elements Access:

operator[]

Access element

If *k* matches the key of an element in the container, the function returns a reference to its mapped value.

```
// accessing mapped values
#include <iostream>
#include <map>
#include <string>

using namespace std;

int main ()
{
  map<char,std::string> mymap;

  mymap['a']="an element";
  mymap['b']="another element";
  mymap['c']=mymap['b'];

  cout << "mymap['a'] is " << mymap['a'] << '\n';
  cout << "mymap['b'] is " << mymap['b'] << '\n';
  cout << "mymap['c'] is " << mymap['c'] << '\n';
  cout << "mymap['d'] is " << mymap['d'] << '\n';

  cout << "mymap now contains " << mymap.size() << " elements.\n";

  return 0;
}
```

at()

Access element

Returns a reference to the mapped value of the element identified with key k.

If k does not match the key of any element in the container, the function throws an out_of_range exception.

```
// map::at
#include <iostream>
#include <string>
#include <map>
using namespace std:
int main ()
{
  map<string,int> mymap = {
                { "alpha", 0 },
                { "beta", 0 },
                { "gamma", 0 } };

  mymap.at("alpha") = 10;
  mymap.at("beta") = 20;
  mymap.at("gamma") = 30;

  for (auto& x: mymap) {
    cout << x.first << ": " << x.second << '\n';
  }

  return 0;
}
```

3. Capacity:

size():

Return container size

Returns the number of elements in the map container.

```
// map::size
#include <iostream>
#include <map>
int main ()
{
  std::map<char,int> mymap;
  mymap['a']=101;
  mymap['b']=202;
  mymap['c']=302;

  std::cout << "mymap.size() is " << mymap.size() << '\n';

  return 0;
}
```

empty()

Returns whether the map container is empty (i.e. whether its size is 0).

This function does not modify the container in any way. To clear the content of a map container, see map::clear.

Returns true if the container size is 0, false otherwise.

```
// map::empty
#include <iostream>
#include <map>
int main ()
{
```

```
  std::map<char,int> mymap;
  mymap['a']=10;
  mymap['b']=20;
  mymap['c']=30;
  while (!mymap.empty())
  {
    std::cout << mymap.begin()->first << " => " << mymap.begin()->second << '\n';
    mymap.erase(mymap.begin());
  }
  return 0;
}
```

max_size()

Return maximum size

Returns the maximum number of elements that the map container can hold.

```
// map::max_size
#include <iostream>
#include <map>
int main ()
{
  int i;
  std::map<int,int> mymap;

  if (mymap.max_size()>1000)
  {
    for (i=0; i<1000; i++) mymap[i]=0;
    std::cout << "The map contains 1000 elements.\n";
  }
  else std::cout << "The map could not hold 1000 elements.\n";
  return 0;
}
```

4. Operations:

find()

Get iterator to element

Searches the container for an element with a key equivalent to k and returns an iterator to it if found, otherwise it returns an iterator to map::end.

```
// map::find
#include <iostream>
#include <map>
int main ()
{
  std::map<char,int> mymap;
  std::map<char,int>::iterator it;
  mymap['a']=50;
  mymap['b']=100;
  mymap['c']=150;
  mymap['d']=200;
  it=mymap.find('b');
  mymap.erase (it);
  mymap.erase (mymap.find('d'));
  // print content:
  std::cout << "elements in mymap:" << '\n';
  std::cout << "a => " << mymap.find('a')->second << '\n';
  std::cout << "c => " << mymap.find('c')->second << '\n';
  return 0;
}
```

count()

Count elements with a specific key

Searches the container for elements with a key equivalent to k and returns the number of matches.

Because all elements in a map container are unique, the function can only return 1 (if the element is found) or zero (otherwise).

```
// map::count
#include <iostream>
#include <map>
int main ()
{
  std::map<char,int> mymap;
  char c;
  mymap ['a']=101;
  mymap ['c']=202;
  mymap ['f']=303;
  for (c='a'; c<'h'; c++)
  {
    std::cout << c;
    if (mymap.count(c)>0)
      std::cout << " is an element of mymap.\n";
    else
      std::cout << " is not an element of mymap.\n";
  }
  return 0;
}
```

lower_bound()

Return iterator to lower bound

Returns an iterator pointing to the first element in the container whose key is not considered to go before k (i.e., either it is equivalent or goes after).

```
// map::lower_bound/upper_bound
#include <iostream>
#include <map>
int main ()
```

```
{
  std::map<char,int> mymap;
  std::map<char,int>::iterator itlow,itup;
  mymap['a']=20;
  mymap['b']=40;
  mymap['c']=60;
  mymap['d']=80;
  mymap['e']=100;
  itlow=mymap.lower_bound ('b');  // itlow points to b
  itup=mymap.upper_bound ('d');   // itup points to e (not d!)
  mymap.erase(itlow,itup);        // erases [itlow,itup)
  // print content:
  for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';
  return 0;
}
```

upper_bound()

Return iterator to upper bound

Returns an iterator pointing to the first element in the container whose key is considered to go after k.

```
// map::lower_bound/upper_bound
#include <iostream>
#include <map>
int main ()
{
  std::map<char,int> mymap;
  std::map<char,int>::iterator itlow,itup;
  mymap['a']=20;
  mymap['b']=40;
  mymap['c']=60;
  mymap['d']=80;
  mymap['e']=100;
```

```
  itlow=mymap.lower_bound ('b');  // itlow points to b
  itup=mymap.upper_bound ('d');   // itup points to e (not d!)
  mymap.erase(itlow,itup);        // erases [itlow,itup)
  // print content:
  for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';
  return 0;
}
```

Pair:

what is pair?

his class couples together a pair of values, which may be of different types (T1 and T2). The individual values can be accessed through its public members first and second.

what types of data can be coupled in pair?

We can couple any kind of basic data type and string,vetor,structure etc in pair. We can also couple two differnt types data together.

How to declare a pair?

pair<first_element_type,second_element_type>pair_name;

We can constract pair in flowing way,

```
// pair::pair example
#include <utility>       // std::pair, std::make_pair
#include <string>        // std::string
#include <iostream>      // std::cout
int main () {
  std::pair <std::string,double> product1;                     // default constructor
  std::pair <std::string,double> product2 ("tomatoes",2.30);   // value init
  std::pair <std::string,double> product3 (product2);          // copy constructor
```

```
  product1 = std::make_pair(std::string("lightbulbs"),0.99);   // using make_pair (move)
  product2.first = "shoes";                     // the type of first is string
  product2.second = 39.90;                      // the type of second is double
  std::cout << "The price of " << product1.first << " is $" << product1.second << '\n';
  std::cout << "The price of " << product2.first << " is $" << product2.second << '\n';
  std::cout << "The price of " << product3.first << " is $" << product3.second << '\n';
  return 0;
}
```

What happen when we sort pair?

It first sort according to first elements then second element.

Some member function of pair:

operator =

Assign contents

Assigns pr as the new content for the pair object.

Member first is assigned pr.first, and member second is assigned pr.second.

```
// pair::operator= example
#include <utility>       // std::pair, std::make_pair
#include <string>        // std::string
#include <iostream>      // std::cout
int main () {
  std::pair <std::string,int> planet, homeplanet;
  planet = std::make_pair("Earth",6371);
  homeplanet = planet;
  std::cout << "Home planet: " << homeplanet.first << '\n';
  std::cout << "Planet size: " << homeplanet.second << '\n';
  return 0;
}
```

swap()

Swap contents

Exchanges the contents of the pair object with the contents of pr.

```
// pair::swap example
#include <utility>       // std::pair
#include <iostream>      // std::cout
int main () {
  std::pair<int,char> foo (10,'a');
  std::pair<int,char> bar (90,'z');
  foo.swap(bar);
  std::cout << "foo contains: " << foo.first;
  std::cout << " and " << foo.second << '\n';
  return 0;
}
```

make_pair()

Construct pair object

Constructs a pair object with its first element set to x and its second element set to y.

```
// make_pair example
#include <utility>       // std::pair
#include <iostream>      // std::cout
int main () {
  std::pair <int,int> foo;
  std::pair <int,int> bar;
  foo = std::make_pair (10,20);
  bar = std::make_pair (10.5,'A'); // ok: implicit conversion from pair<double,char>
  std::cout << "foo: " << foo.first << ", " << foo.second << '\n';
  std::cout << "bar: " << bar.first << ", " << bar.second << '\n';
  return 0;
```

}

Set:

what is set?

Sets are containers that store unique elements following a specific order.

In a set, the value of an element also identifies it (the value is itself the key, of type T), and each value must be unique. The value of the elements in a set cannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container.

set containers are generally slower than unordered set containers to access individual elements by their key, but they allow the direct iteration on subsets based on their order.

Sets are typically implemented as binary search trees.

what we have to include to use set?

#include<set>

How to declare a set?

set<data_type>set_name;

More way to declare a set:

```
// constructing sets
#include <iostream>
#include <set>
bool fncomp (int lhs, int rhs) {return lhs<rhs;}
struct classcomp {
  bool operator() (const int& lhs, const int& rhs) const
  {return lhs<rhs;}
};
int main ()
{
  std::set<int> first;                           // empty set of ints
```

```
  int myints[]= {10,20,30,40,50};
  std::set<int> second (myints,myints+5);          // range
std::set<int> third (second);                      // a copy of second

  std::set<int> fourth (second.begin(), second.end());  // iterator ctor.
  std::set<int,classcomp> fifth;                     // class as Compare
  bool(*fn_pt)(int,int) = fncomp;
  std::set<int,bool(*)(int,int)> sixth (fn_pt);  // function pointer as Compare
  return 0;
}
```

Copy a set to another set:

operator =

Copy container content

Assigns new contents to the container, replacing its current content.

```
// assignment operator with sets
#include <iostream>
#include <set>
int main ()
{
  int myints[]={ 12,82,37,64,15 };
  std::set<int> first (myints,myints+5);    // set with 5 ints
  std::set<int> second;                      // empty set
  second = first;                            // now second contains the 5 ints
  first = std::set<int>();                   // and first is empty
  std::cout << "Size of first: " << int (first.size()) << '\n';
  std::cout << "Size of second: " << int (second.size()) << '\n';
  return 0;
}
```

How to print elements of a set?

As set does not sequential memory as array. So to print elements of a set we have to use iterator;

we can do it in flowing way,

```
#include <iostream>
#include <set>
int main ()
{
  std::set<int> myset;
  std::set<int>::iterator it; //iterator to iterate over the set.
  // insert some values:
  for (int i=1; i<10; i++) myset.insert(i*10);  // 10 20 30 40 50 60 70 80 90
  std::cout << "myset contains:";
  for (it=myset.begin(); it!=myset.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```

Some member function of Set:

1.Modifier:

insert()

Insert element.

Extends the container by inserting new elements, effectively increasing the container size by the number of elements inserted.

Because elements in a set are unique, the insertion operation checks whether each inserted element is equivalent to an element already in the container, and if so, the element is not inserted, returning an iterator to this existing element (if the function returns a value).

```
// set::insert (C++98)
```

```
#include <iostream>
#include <set>

int main ()
{
  std::set<int> myset;
  std::set<int>::iterator it;
  std::pair<std::set<int>::iterator,bool> ret;
  // set some initial values:
  for (int i=1; i<=5; ++i) myset.insert(i*10);    // set: 10 20 30 40 50
  ret = myset.insert(20);               // no new element inserted
  if (ret.second==false) it=ret.first;  // "it" now points to element 20
  myset.insert (it,25);                 // max efficiency inserting
  myset.insert (it,24);                 // max efficiency inserting
  myset.insert (it,26);                 // no max efficiency inserting
  int myints[]= {5,10,15};              // 10 already in set, not inserted
  myset.insert (myints,myints+3);
  std::cout << "myset contains:";
  for (it=myset.begin(); it!=myset.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```

erase()

Erase elements

Removes from the set container either a single element or a range of elements ([first,last)).

This effectively reduces the container size by the number of elements removed, which are destroyed.

```
// erasing from set
#include <iostream>
```

```
#include <set>
int main ()
{
  std::set<int> myset;
  std::set<int>::iterator it;
  // insert some values:
  for (int i=1; i<10; i++) myset.insert(i*10);  // 10 20 30 40 50 60 70 80 90
  it = myset.begin();
  ++it;                                         // "it" points now to 20
  myset.erase (it);
  myset.erase (40);
  it = myset.find (60);
  myset.erase (it, myset.end());
  std::cout << "myset contains:";
  for (it=myset.begin(); it!=myset.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```

clear()

Clear content

Removes all elements from the set container (which are destroyed), leaving the container with a size of 0.

```
// set::clear
#include <iostream>
#include <set>
int main ()
{
  std::set<int> myset;
```

```
  myset.insert (100);
  myset.insert (200);
  myset.insert (300);
  std::cout << "myset contains:";
  for (std::set<int>::iterator it=myset.begin(); it!=myset.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  myset.clear();
  myset.insert (1101);
  myset.insert (2202);
  std::cout << "myset contains:";
  for (std::set<int>::iterator it=myset.begin(); it!=myset.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```

2. Capacity:

size()

Return container size

Returns the number of elements in the set container.

```
// set::size
#include <iostream>
#include <set>
int main ()
{
  std::set<int> myints;
  std::cout << "0. size: " << myints.size() << '\n';
  for (int i=0; i<10; ++i) myints.insert(i);
  std::cout << "1. size: " << myints.size() << '\n';
```

```
  myints.insert (100);
  std::cout << "2. size: " << myints.size() << '\n';
  myints.erase(5);
  std::cout << "3. size: " << myints.size() << '\n';
  return 0;
}
```

empty()

Test whether container is empty

Returns whether the set container is empty (i.e. whether its size is 0).

This function does not modify the container in any way. To clear the content of a set container, see set::clear.

```
// set::empty
#include <iostream>
#include <set>
int main ()
{
  std::set<int> myset;
  myset.insert(20);
  myset.insert(30);
  myset.insert(10);
  std::cout << "myset contains:";
  while (!myset.empty())
  {
     std::cout << ' ' << *myset.begin();
     myset.erase(myset.begin());
  }
  std::cout << '\n';
  return 0;
}
```

max_size()

Return maximum size

Returns the maximum number of elements that the set container can hold.

```
// set::max_size
#include <iostream>
#include <set>
int main ()
{
  int i;
  std::set<int> myset;
  if (myset.max_size()>1000)
  {
    for (i=0; i<1000; i++) myset.insert(i);
    std::cout << "The set contains 1000 elements.\n";
  }
  else std::cout << "The set could not hold 1000 elements.\n";
  return 0;
}
```

3. Operations:

find()

Get iterator to element

Searches the container for an element equivalent to val and returns an iterator to it if found, otherwise it returns an iterator to set::end.

```
// set::find
#include <iostream>
#include <set>
int main ()
{
```

```
  std::set<int> myset;
  std::set<int>::iterator it;

  // set some initial values:
  for (int i=1; i<=5; i++) myset.insert(i*10);    // set: 10 20 30 40 50
  it=myset.find(20);
  myset.erase (it);
  myset.erase (myset.find(40));
  std::cout << "myset contains:";
  for (it=myset.begin(); it!=myset.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```

4. Itterator:

begin()

Returns an iterator referring to the first element in the set container.

Because set containers keep their elements ordered at all times, begin points to the element that goes first following the container's sorting criterion.

```
// set::begin/end
#include <iostream>
#include <set>
int main ()
{
  int myints[] = {75,23,65,42,13};
  std::set<int> myset (myints,myints+5);
  std::cout << "myset contains:";
  for (std::set<int>::iterator it=myset.begin(); it!=myset.end(); ++it)
    std::cout << ' ' << *it;
```

```
  std::cout << '\n';
  return 0;
}
```

end()

Return iterator to end

Returns an iterator referring to the past-the-end element in the set container.

```
// set::begin/end
#include <iostream>
#include <set>
int main ()
{
  int myints[] = {75,23,65,42,13};
  std::set<int> myset (myints,myints+5);
  std::cout << "myset contains:";
  for (std::set<int>::iterator it=myset.begin(); it!=myset.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```

Stack:

What is stack?

LIFO stack

Stacks are a type of container adaptor, specifically designed to operate in a LIFO context (last-in first-out), where elements are inserted and extracted only from one end of the container.

What we need to include to use stack?

```
#include<stack>
```

How to declare Stack?

stack<data_type>stack_name;

i.e.

stack<int>stk;

Some member function of Stack:

push()

Insert element

Inserts a new element at the top of the stack, above its current top element. The content of this new element is initialized to a copy of val.

```
// stack::push/pop
#include <iostream>       // std::cout
#include <stack>          // std::stack
int main ()
{
  std::stack<int> mystack;
  for (int i=0; i<5; ++i) mystack.push(i);
  std::cout << "Popping out elements...";
  while (!mystack.empty())
  {
     std::cout << ' ' << mystack.top();
     mystack.pop();
  }
  std::cout << '\n';
  return 0;
}
```

pop()

Remove top element

Removes the element on top of the stack, effectively reducing its size by one.

The element removed is the latest element inserted into the stack, whose value can be retrieved by calling member stack::top.

This calls the removed element's destructor.

```
// stack::push/pop
#include <iostream>       // std::cout
#include <stack>          // std::stack
int main ()
{
  std::stack<int> mystack;
  for (int i=0; i<5; ++i) mystack.push(i);
  std::cout << "Popping out elements...";
  while (!mystack.empty())
  {
     std::cout << ' ' << mystack.top();
     mystack.pop();
  }
  std::cout << '\n';
  return 0;
}
```

top()

Access next element

Returns a reference to the top element in the stack.

Since stacks are last-in first-out containers, the top element is the last element inserted into the stack.

```
// stack::top
#include <iostream>       // std::cout
#include <stack>          // std::stack
int main ()
```

```
{
  std::stack<int> mystack;
  mystack.push(10);
  mystack.push(20);
  mystack.top() -= 5;
  std::cout << "mystack.top() is now " << mystack.top() << '\n';
  return 0;
}
```

size()

Return size

Returns the number of elements in the stack.

```
// stack::size
#include <iostream>       // std::cout
#include <stack>          // std::stack
int main ()
{
  std::stack<int> myints;
  std::cout << "0. size: " << myints.size() << '\n';
  for (int i=0; i<5; i++) myints.push(i);
  std::cout << "1. size: " << myints.size() << '\n';
  myints.pop();
  std::cout << "2. size: " << myints.size() << '\n';
  return 0;
}
```

empty()

Test whether container is empty

Returns whether the stack is empty: i.e. whether its size is zero.

Returns true if the underlying container's size is 0, false otherwise.

```
// stack::empty
#include <iostream>       // std::cout
#include <stack>          // std::stack

int main ()
{
  std::stack<int> mystack;
  int sum (0);
  for (int i=1;i<=10;i++) mystack.push(i);
  while (!mystack.empty())
  {
     sum += mystack.top();
     mystack.pop();
  }
  std::cout << "total: " << sum << '\n';
  return 0;
}
```

Queue:

What is queue?

FIFO queue

queues are a type of container adaptor, specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other.

What is need to include to use queue?

```
#include<queue>
```

How to declare queue?

queue<int>qt;

Some Member Function of Queue:

1. push()

Insert element

Inserts a new element at the end of the queue, after its current last element. The content of this new element is initialized to val.

```
// queue::push/pop
#include <iostream>       // std::cin, std::cout
#include <queue>          // std::queue
int main ()
{
  std::queue<int> myqueue;
  int myint;
  std::cout << "Please enter some integers (enter 0 to end):\n";
  do {
    std::cin >> myint;
    myqueue.push (myint);
  } while (myint);
  std::cout << "myqueue contains: ";
  while (!myqueue.empty())
  {
    std::cout << ' ' << myqueue.front();
    myqueue.pop();
  }
  std::cout << '\n';
  return 0;
}
```

2. pop()

Remove next element

Removes the next element in the queue, effectively reducing its size by one.

The element removed is the "oldest" element in the queue whose value can be retrieved by calling member queue::front.

```
// queue::push/pop
#include <iostream>       // std::cin, std::cout
#include <queue>          // std::queue
int main ()
{
  std::queue<int> myqueue;
  int myint;
  std::cout << "Please enter some integers (enter 0 to end):\n";
  do {
    std::cin >> myint;
    myqueue.push (myint);
  } while (myint);
  std::cout << "myqueue contains: ";
  while (!myqueue.empty())
  {
    std::cout << ' ' << myqueue.front();
    myqueue.pop();
  }
  std::cout << '\n';
  return 0;
}
```

front()

Access next element

Returns a reference to the next element in the queue.

The next element is the "oldest" element in the queue and the same element that is popped out from the queue when queue::pop is called.

```
// queue::front
#include <iostream>       // std::cout
#include <queue>          // std::queue
int main ()
{
  std::queue<int> myqueue;
  myqueue.push(77);
  myqueue.push(16);
  myqueue.front() -= myqueue.back();     // 77-16=61
  std::cout << "myqueue.front() is now " << myqueue.front() << '\n';
  return 0;
}
```

4. back()

Access last element

Returns a reference to the last element in the queue. This is the "newest" element in the queue (i.e. the last element pushed into the queue).

```
// queue::back
#include <iostream>       // std::cout
#include <queue>          // std::queue
int main ()
{
  std::queue<int> myqueue;
  myqueue.push(12);
  myqueue.push(75);   // this is now the back
  myqueue.back() -= myqueue.front();
  std::cout << "myqueue.back() is now " << myqueue.back() << '\n';
  return 0;
```

```
}
```

5.size()

Return size

Returns the number of elements in the queue.

```
// queue::size
#include <iostream>       // std::cout
#include <queue>          // std::queue
int main ()
{
  std::queue<int> myints;
  std::cout << "0. size: " << myints.size() << '\n';
  for (int i=0; i<5; i++) myints.push(i);
  std::cout << "1. size: " << myints.size() << '\n';
  myints.pop();
  std::cout << "2. size: " << myints.size() << '\n';
  return 0;
}
```

6. empty()

Test whether container is empty

Returns whether the queue is empty: i.e. whether its size is zero.

Returns true if the underlying container's size is 0, false otherwise.

```
// queue::empty
#include <iostream>       // std::cout
#include <queue>          // std::queue
int main ()
{
```

```
  std::queue<int> myqueue;
  int sum (0);
  for (int i=1;i<=10;i++) myqueue.push(i);
  while (!myqueue.empty())
  {
     sum += myqueue.front();
     myqueue.pop();
  }
  std::cout << "total: " << sum << '\n';
  return 0;
}
```

Deque:

What is a deque?

Double ended queue

deque (usually pronounced like "deck") is an irregular acronym of double-ended queue. Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends (either its front or its back).

What needs to include to use deque?

#include<deque>

How to declare deque?

deque<data_type>dq;

i.e.

deque<int>dq;

more way to declare deque:

// constructing deques

```
#include <iostream>
#include <deque>
int main ()
{
  unsigned int i;
  // constructors used in the same order as described above:
  std::deque<int> first;                                // empty deque of ints
  std::deque<int> second (4,100);                       // four ints with value 100
  std::deque<int> third (second.begin(),second.end());  // iterating through second
  std::deque<int> fourth (third);                       // a copy of third
  // the iterator constructor can be used to copy arrays:
  int myints[] = {16,2,77,29};
  std::deque<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );
  std::cout << "The contents of fifth are:";
  for (std::deque<int>::iterator it = fifth.begin(); it!=fifth.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```

How to copy a deque to another deque?

Using = operator.

It assigns new contents to the container, replacing its current contents, and modifying its size accordingly.

```
// assignment operator with deques
#include <iostream>
#include <deque>
int main ()
{
  std::deque<int> first (3);      // deque with 3 zero-initialized ints
```

```
  std::deque<int> second (5);    // deque with 5 zero-initialized ints
  second = first;
  first = std::deque<int>();
  std::cout << "Size of first: " << int (first.size()) << '\n';
  std::cout << "Size of second: " << int (second.size()) << '\n';
  return 0;
}
```

Some Member function of Dequeu:

1.Modifiers:

a. assign() //Assign container content (public member function)

b. push_back() //Add element at the end (public member function)

c. push_front() //Insert element at beginning (public member function)

d. pop_back() //Delete last element (public member function)

e. pop_front() //Delete first element (public member function)

f. insert() //Insert elements (public member function)

g. erase() //Erase elements (public member function)

h. clear() // Clear content (public member function)

2. Element access:

a. operator[] //Access element (public member function)

b. at() //Access element (public member function)

c. front() //Access first element (public member function)

d. back() //Access last element (public member function)

3.Capacity:

a. size() //Return size (public member function)

b. max_size() //Return maximum size (public member function)

c. resize() //Change size (public member function)

d. empty() //Test whether container is empty (public member function)

Priority Queue:

What is priority queue?

Priority queue

Priority queues are a type of container adaptors, specifically designed such that its first element is always the greatest of the elements it contains, according to some strict weak ordering criterion.

This context is similar to a heap, where elements can be inserted at any moment, and only the max heap element can be retrieved (the one at the top in the priority queue).

What needs to include to use priority queue?

#include<queue>

How to declare a priority queue?

priority_queue<data_type> name;

i.e.

priority_queue<int> first;

Example:

```
// constructing priority queues
#include <iostream>          // std::cout
#include <queue>             // std::priority_queue
#include <vector>            // std::vector
#include <functional>        // std::greater
class mycomparison
{
  bool reverse;
public:
  mycomparison(const bool& revparam=false)
    {reverse=revparam;}
```

```
  bool operator() (const int& lhs, const int&rhs) const
  {
    if (reverse) return (lhs>rhs);
    else return (lhs<rhs);
  }
};

int main ()
{
  int myints[]= {10,60,50,20};

  std::priority_queue<int> first;
  std::priority_queue<int> second (myints,myints+4);
  std::priority_queue<int, std::vector<int>, std::greater<int> >
                              third (myints,myints+4);
  // using mycomparison:
  typedef std::priority_queue<int,std::vector<int>,mycomparison> mypq_type;
  mypq_type fourth;                         // less-than comparison
  mypq_type fifth (mycomparison(true));    // greater-than comparison
  return 0;
}
```

Member Function of priority queue:

1.push()

Insert element

Inserts a new element in the priority_queue. The content of this new element is initialized to val.

```
// priority_queue::push/pop
#include <iostream>         // std::cout
#include <queue>            // std::priority_queue
int main ()
```

```
{
  std::priority_queue<int> mypq;
  mypq.push(30);
  mypq.push(100);
  mypq.push(25);
  mypq.push(40);
  std::cout << "Popping out elements...";
  while (!mypq.empty())
  {
     std::cout << ' ' << mypq.top();
     mypq.pop();
  }
  std::cout << '\n';
  return 0;
}
```

2. pop()

Remove top element

Removes the element on top of the priority_queue, effectively reducing its size by one. The element removed is the one with the highest value.

The value of this element can be retrieved before being popped by calling member priority_queue::top.

```
// priority_queue::push/pop
#include <iostream>       // std::cout
#include <queue>          // std::priority_queue
int main ()
{
  std::priority_queue<int> mypq;
  mypq.push(30);
  mypq.push(100);
```

```
  mypq.push(25);
  mypq.push(40);
  std::cout << "Popping out elements...";
  while (!mypq.empty())
  {
     std::cout << ' ' << mypq.top();
     mypq.pop();
  }
  std::cout << '\n';
  return 0;
}
```

3. top()

Access top element

Returns a constant reference to the top element in the priority_queue.

The top element is the element that compares higher in the priority_queue, and the next that is removed from the container when priority_queue::pop is called.

```
// priority_queue::top
#include <iostream>       // std::cout
#include <queue>          // std::priority_queue
int main ()
{
  std::priority_queue<int> mypq;
  mypq.push(10);
  mypq.push(20);
  mypq.push(15);
  std::cout << "mypq.top() is now " << mypq.top() << '\n';
  return 0;
}
```

4. empty()

Test whether container is empty

Returns whether the priority_queue is empty: i.e. whether its size is zero.

```
// priority_queue::empty
#include <iostream>       // std::cout
#include <queue>          // std::priority_queue
int main ()
{
  std::priority_queue<int> mypq;
  int sum (0);

  for (int i=1;i<=10;i++) mypq.push(i);
  while (!mypq.empty())
  {
     sum += mypq.top();
     mypq.pop();
  }
  std::cout << "total: " << sum << '\n';
  return 0;
}
```

MD: EMRUZ HOSSAIN

Chittagong University of Engineering & Technology

Dept: CSE

ID: 1204084

Most of it contents of this doc is taken form

http://www.cplusplus.com/