Handout on Map Class

Maps are associative containers that store elements in the form of a key value pair combination in a sorted order based on the key.

Header file

To use this class, we need to include the following header file in our program

#include<map>

All the member functions of the map class and the map class itself is in the namespace std. Map objects can be constructed based on the contructor used.

In general, every **map<key_type, value_type>** object is a collection of (**key**,**value**) pairs.

Various ways in which a map object can be created are as follows:

Usage	**Explanation**
map<char, int> m1;	This creates an empty map object '**m1**' with key of type char and the value of type int
m1['a']=1;	This creates a new key value pair by storing the value 1 of type int along with key '**a**' of type char in the map named '**m1**'.
m1['b']=2; **m1['c']=3;**	Can be explained in the similar fashion as above
map<char,int> m2(m1.begin(),m1.end());	This creates a map object m2 of 3 (key,value) pairs by iterating through the map object m1.
map<char,int> m3 (m2);	This copies the map object m2 to map object m3.
map<char,int> m4 {{'a',1}, {'b',2},{'c',3}}	Map objects can also be initialized in this fashion

If we want to access the element indexed by the key value, we can write the following

cout << m1['A'];

This will print 49.

The '**operator=**' can be used to copy a map opject to another map object of the same type

For example , '**m4=m3;**' will copy map object '**m3**' to map object '**m4**'

Iterators:

Iterators are public member functions defined for iterating through the element of type defined for the containers. Following are the list of iterators available for the '**map**' class.

begin()

This returns an iterator referring to the first element in the map. The following statement will initialize the iterator pointing to the begining of the map container

map<char, int> m1;

map<char, int>::iterator it = m1.begin();

In order to access the key of the element in the map, we need to use '**it->first**', whereas to access the value of the element, we need to use '**it->second**'.

E.g.

```
map<char,int> m1;
m1['A'] = 49;
m1['C'] = 51;
m1['B'] = 50;
map<char,int>::iterator it = m1.begin();
cout << "Key = "<< it->first <<endl;
cout << "Value = " << it -> second <<endl;
```

Output

```
Key = A
Value = 49
```

end()

This returns an iterator pointing to the past-the-end element in the map. The following statement will initialize the iterator pointing to the past-the-end element of the map container

```
map<char, int> m1;
map<char, int>::iterator it = m1.end();
```

In order to access the key of the element in the map, we need to use '**it->first**', whereas to access the value of the element, we need to use '**it->second**'. We cannot use this immediately after executing this '**m1.end()**' as it does not point to any element of the map container and should no be used to dereference. We need to decrement this iterator and then use it.

E.g.

```
map<char,int> m1;
m1['A'] = 49;
m1['C'] = 51;
m1['B'] = 50;
map<char,int>::iterator it = m1.end();
it--;
cout << "Key = "<< it->first <<endl;
cout << "Value = " << it -> second <<endl;
```

Output

```
Key = C
Value = 51
```

'**begin**' in conjunction with '**end**' can be used to iterate through the map elements as follows:

```
for (map<char,int>::iterator it=m1.begin(); it!=m1.end(); ++it) {
  cout << it->first << " => " << it->second << '\n';
}
```

rbegin()

'**rbegin**' is a backward iterator and it returns a reverse iterator pointing to the last element in the map container. Incrementing this iterator, moves to the begining of the container. The following statement will initialize the reverse iterator pointing to the last element of the map container

```
map<char, int> m1;
```

map<char, int>::reverse_iterator rit = m1.rbegin();

In order to access the key of the element in the map, we need to use '**rit->first**', whereas to access the value of the element, we need to use '**rit->second**'.

E.g.

```
map<char,int> m1;
m1['A'] = 49;
m1['C'] = 51;
m1['B'] = 50;
map<char,int>::reverse_iterator rit = m1.rbegin();
cout << "Key = "<< rit->first <<endl;
cout << "Value = " << rit -> second <<endl;
```

Output Key = C
Value = 51

rend()

rend is a backward iterator and it returns a reverse iterator pointing to the element before the first element in the map container. The following statement will initialize the reverse iterator pointing to the element one before the first element

map<char, int> m1;

map<char, int>::reverse_iterator rit = m1.rend();

In order to access the key of the element in the map, we need to use '**rit->first**', whereas to access the value of the element, we need to use '**rit->second**'. Just after executing '**m1.rend()**', we should not use '**rit->first**' or '**rit->second**', to access the element of the map container, as it will point to the element before the first element of the map container.

E.g.

```
map<char,int> m1;
m1['A'] = 49;
m1['C'] = 51;
m1['B'] = 50;
map<char,int>::reverse_iterator rit = m1.rend();
rit--;
cout << "Key = "<< rit->first <<endl;
cout << "Value = " << rit -> second <<endl;
```

Output

Key = A
Value = 49

rbegin in conjunction with rend can be used to reverse iterate through the map elements as follows:

```
for (rit=m1.rbegin(); rit!=m1.rend(); ++rit)
   cout << rit->first << " => " << rit->second << '\n';
```

Capacity:

When maps are initialized, they typically consist of a pointer to a dynamically allocated memory. The allocated size(capacity) may be larger than the actual size used in the program. When new elements are added, the size of the map object is automatically set. Below we mention some of the useful member functions for checking size of the map object.

size()

size is a member function which returns the size(number of elements) of the map object. The following statements can be used to calculate the size of the map object. In this example, the size returned is 3.

map<char,int> m4 {{'a',1}, {'b',2},{'c',3}};

cout << m5.size();

If we had used **map<char,int> m4;** instead of the above statement, the size returned would have been **0**.

max_size()

This returns the maximum number of elements the vector can hold. It is system dependent. On some systems, you would get the value as high as **1073741823**.

empty()

This can be used to test if the map object is empty(no elements). The function returns a boolean result.

Usage:	Explanation
if(!m3.empty()){	'**m3**' is a map container checked to see if it is not empty
//do this	
}	
else{	
//do something	
}	

Element Access:

Elements of a map container can be accessed using operator[] and at() functions as described below:

operator[]

If the key value matches with a key in the map container, it returns a reference to the mapped value in the container. The member operator[] does not check for bounds and has undefined behavior if access is made using a position value which is out of bound.

The following statements of code will illustrate the usage of an operator[]

Usage	**Explanation**
map<char,int> m1;	This creates a map container '**m1**'
m1['A']=49;	This inserts the key value pair
cout<< m1['A'];	This prints the value at key '**A**' i.e. 49

at( )

If the key value matches with the key in the map container, it returns a reference to the mapped value of the element in the map container.

The following statements of code will illustrate the usage of an at( ) member function

Usage	**Explanation**
map<char,int> m1;	This creates a map container '**m1**'
m1['A']=49;	This inserts the key value pair

m1.at('A');	This prints the value at key '**A**' i.e. 49

Modifiers:

These are member functions which help in inserting, updating, and deleting elements of map containers. The following member functions are used to perform such operations:

insert( )

The map container can also be extended by inserting new elements. The size of the map is automatically increased. This operation is very inefficient as all the elements after the position specified need to be reallocated to make space for the new elements.

The below example illustrates the insert() function.

Usage	**Explanation**
map<char,int> m1;	This creates a map '**m1**'
m1['A'] = 49;	This initializes the map
m1['C'] = 51;	
m1['B'] = 50;	
m1.insert (pair<char,int>('D',100));	This inserts the key pair value '**D, 100**' after all the elements in the map

erase( )

The element(s) from the map container can also be removed by using this function. It can either remove a single element or a range of elements specified in the arguments. In using the range from first and last as an argument, it will remove all the elements positioned between the first and the last, including the element pointed by first(but not last). The size of the vector container will be automatically decreased

The below example illustrates the **erase()** function:

Usage	**Explanation**
map<char,int> m1;	This creates a map container '**m1**'
m1['A']=49;	This inserts the key value pairs
m1['B']=50;	
m1['C']=51;	
m1['D']=52;	
map<char,int>::iterator it = m1.begin();	initializing iterator '**it**'
m1.erase(it);	Erases the first element from the map
m1.erase(it, m1.end());	Erases the elements from where the iterator is pointing till the end

swap( )

This member function exchanges the content of the container by the content of another container of the same type specified in the argument. The iterators reamain valid even after swapping map contents.

E.g.

```
map<char,int>::iterator it;
```

```
 map<char,int> m1;
 m1['A'] = 49;
 m1['C'] = 51;
 m1['B'] = 50;
 map<char,int> m2;
 m2['D'] = 52;
 m2['E'] = 53;
 cout<<"Before Swapping: "<<endl;
 cout<<"Map 1: \n";
 for (it=m1.begin(); it!=m1.end(); ++it)
   cout << it->first << " => " << it->second << '\n';
 cout<<"\nMap 2: \n";
 for (it=m2.begin(); it!=m2.end(); ++it)
   cout << it->first << " => " << it->second << '\n';
 m1.swap(m2);
 cout<<"\nAfter Swapping: "<<endl;
 cout<<"Map 1: \n";
 for (it=m1.begin(); it!=m1.end(); ++it)
   cout << it->first << " => " << it->second << '\n';
 cout<<"\nMap 2: \n";
 for (it=m2.begin(); it!=m2.end(); ++it)
   cout << it->first << " => " << it->second << '\n';
```

Output

```
Before Swapping:
Map 1:
A => 49
B => 50
C => 51
Map 2:
D => 52
E => 53
After Swapping:
Map 1:
D => 52
E => 53
Map 2:
A => 49
B => 50
C => 51
```

Operations:

These are public member functions which help in locating a key in the map container.

find()

This searches the map container for an element with a key equivalent to the value given in the function argument. If it finds, then it returns the iterator to it, else it returns an iterator to '**map::end**'.

In the following, the iterator '**it**' is made to point to the eleemnt indexed by the key '**B**'

```
map<char, int>::iterator it = m1.find('B');
```

we can use the iterator to perform other operations like '**erase**'

```
m1.erase(it);                    // This will erase the element pair containing key value 'B',50
```

count()

This public memeber function searches the map container for an element with a key equivalent to the value given in the function argument. If it matches(all emenets are unique) it returns 1, else it returns 0.

E.g. Assume that the map 'm1' is already created and initialized.

```
 if(m1.count('B') > 0)
   cout<<"Element found";
 else
  cout<<"Element not found";
```

For more details, please refer to the following reference links:

http://www.cplusplus.com/reference

http://en.wikipedia.org/wiki/C++_Standard_Library