map in C++ STL

The **map** container in C++ STL is an associative container that is used to store key-value pairs in an ordered fashion. By default, the order of elements in the map is in increasing order of the key values, however, we can change this order by providing a user-defined comparison function.

Internally map in C++ STL uses Red-Black trees for implementation and like a set, the map also does not allows duplicate key values.

Creating a map: A map in C++ can be created using below syntax.

```
map map_name;

Here, the key type and value type can be any valid
combination of data types available.
It can be:
int, int
string, string
int, string
string, int
or, anything else.
```

Inserting in a map in C++ STL: We can insert values in a map using the insert() function or the "[ ]" operator. The difference between these two is that the insert() function takes specific values for both the key and the value and inserts the new pair. However, if we do not provide value with the member access operator "[]", it inserts the default value of the data type provided for that particular key.

Below program illustrates this:

```

// C++ Program to illustrate map in C++ STL

#include <algorithm>
#include <iostream>
#include <iterator>
#include <map>

using namespace std;

int main()
{
    // empty map container
    map<int, int> m;

    // Insert elements using insert() function
    m.insert({ 10, 100 });
    m.insert({ 30, 300 });

    // Inserting using "[]" operator
    m[20] = 200;

    // Donot provide any value with key 40
    // The operator access the key and since
    // it doesnot exists, it will insert defaut
    // value of int
    m[40];

    // Traverseing map
    cout << "The map is : \n";
```

Run

Output:

```
The map is :
    KEY    ELEMENT
    10     100
    20     200
    30     300
    40     0


The map is :
    KEY    ELEMENT
    10     100
    20    200
    30    300
    40    400
```

find() and count() Functions

- The **map::find()** is a built-in function in C++ STL which returns an iterator or a constant iterator that refers to the position where the key is present in the map. If the key is not present in the map container, it returns an iterator or a constant iterator which refers to map.end().
- The **map::count()** is a built-in function in C++ STL which returns 1 if the element with key K is present in the map container. It returns 0 if the element with key K is not present in the container.

Below program illustrates the find() and count() functions:

```

// C++ program to illustrate map::find()
// map::count() functions
#include <algorithm>
#include <iostream>
#include <map>

using namespace std;

int main()
{

    // initialize container
    map<int, int> mp;

    // insert elements in random order
    mp.insert({ 2, 30 });
    mp.insert({ 1, 40 });
    mp.insert({ 3, 20 });
    mp.insert({ 4, 50 });

    // Check if the element 3 exists
    // in the map or not
    if (mp.find(3) != mp.end())
        cout << "3 Found!\n\n";
    else
        cout << "3 Not Found!\n";

    // Using find() function to print elements
    // starting from a given key
```

Run

Output:

```
3 Found!

The elements from position 3 in map are :
KEY    ELEMENT
3    20
4    50

4 Found!
```

lower_bound() function in map

The **map::lower_bound(k)** is a built-in function in C++ STL which returns an iterator pointing to the key in the container which is equivalent to k passed in the parameter.

Syntax:

```
map_name.lower_bound(key)
```

The function returns an iterator pointing to the key in the map container which is equivalent to k passed in the parameter. In case k is not present in the map container, the function returns an iterator pointing to the immediate next element which is just greater than k. If the key passed in the parameter exceeds the maximum key in the container, then the iterator returned points to the number of elements in the map as key and element=0.

Below program illustrate the lower_bound() function:

```cpp

// C++ function for illustration
// map::lower_bound() function
#include <bits/stdc++.h>
using namespace std;

int main()
{

    // initialize container
    map<int, int> mp;

    // insert elements in random order
    mp.insert({ 2, 30 });
    mp.insert({ 1, 10 });
    mp.insert({ 5, 50 });
    mp.insert({ 4, 40 });
    for (auto it = mp.begin(); it != mp.end(); it++) {
        cout << (*it).first << " " << (*it).second << endl;
    }

    // when 2 is present
    auto it = mp.lower_bound(2);
    cout << "The lower bound of key 2 is ";
    cout << (*it).first << " " << (*it).second << endl;

    // when 3 is not present
    // points to next greater after 3
    it = mp.lower_bound(3);
    cout << "The lower bound of key 3 is ";
```

Run

Output:

```
1 10
2 30
4 40
5 50
The lower bound of key 2 is 2 30
The lower bound of key 3 is 4 40
The lower bound of key 6 is 4 0
```

upper_bound() function in map

The **map::upper_bound()** is a built-in function in C++ STL which returns an iterator pointing to the immediate next element just greater than k. If the key passed in the parameter exceeds the maximum key in the container, then the iterator returned points to the number of elements in the map container as key and element=0.

Syntax:

```
map_name.upper_bound(key)
```

The function returns an iterator pointing to the immediate next element which is just greater than k. If the key passed in the parameter exceeds the maximum key in the container, then the iterator returned points to the number of elements in the map container as key and element=0.

Below is the implementation of the above approach:

```cpp

// C++ function for illustration
// map::upper_bound() function
#include <bits/stdc++.h>
using namespace std;

int main()
{
    // initialize container
    map<int, int> mp;

    // insert elements in random order
    mp.insert({ 12, 30 });
    mp.insert({ 11, 10 });
    mp.insert({ 15, 50 });
    mp.insert({ 14, 40 });

    // when 11 is present
    auto it = mp.upper_bound(11);
    cout << "The upper bound of key 11 is ";
    cout << (*it).first << " " << (*it).second << endl;

    // when 13 is not present
    it = mp.upper_bound(13);
    cout << "The upper bound of key 13 is ";
    cout << (*it).first << " " << (*it).second << endl;

    // when 17 is exceeds the maximum key, so size
    // of mp is returned as key and value as 0.
    it = mp.upper_bound(17);
```

Run

Output:

```
The upper bound of key 11 is 12 30
The upper bound of key 13 is 14 40
The upper bound of key 17 is 4 0
```

map erase() function

The **map::erase()** is a built-in function in C++ STL which is used to erase element from the container. It can be used to erase keys, elements at any specified position or a given range.

Below program illustrate the working of erase() function with Map in C++ STL:

```

// C++ program to illustrate
// map::erase() function

#include <algorithm>
#include <iostream>
#include <map>
using namespace std;

int main()
{

    // initialize container
    map<int, int> mp;

    // insert elements in random order
    mp.insert({ 2, 30 });
    mp.insert({ 1, 40 });
    mp.insert({ 3, 60 });
    mp.insert({ 5, 50 });

    // Initial size of map
    cout << "Initial size of map: " << mp.size() << "\n";

    // function to erase given position
    auto it = mp.find(2);

    // Passing iterator pointing to key 2
    // to erase it
    mp.erase(it);
```

Run

Output:

```
Initial size of map: 4
Size after erasing one element: 3
Size after erasing second element: 2
Final Size: 0
```

Time Complexity

A map contains key-value pairs and stores them in a sorted order by default. Map in C++ STL uses Red Black trees internally for implementation.

Below is the time complexities of some important functions of Map STL:

```
begin()     --------
end()       -------- \
rbegin()    --------  \
rend()      --------    ---> O(1) Time Complexity
size()      --------  /
empty()     -------- /


count()       --------
find()        --------  \
erase(key)    --------    ---> O(logN) Time Complexity
insert()      --------  /
[] operator   -------- /
```

Sample Problem : Print Elements of an Array according to order defined by another Array

Given two arrays a1[] and a2[], print elements of a1 in such a way that the relative order among the elements will be the same as those are in a2. That is, elements that come before in the array a2[], print those elements first from the array a1[]. For the elements not present in a2, print them at last in sorted order.

It is also given that the number of elements in a2[] is smaller than or equal to the number of elements in a1[], and a2[] has all distinct elements.

Example:

> **Input:** a1[] = {2, 1, 2, 5, 7, 1, 9, 3, 6, 8, 8}
> a2[] = {2, 1, 8, 3}
> **Output:** 2 2 1 1 8 8 3 5 6 7 9
>
> **Input:** a1[] = {2, 1, 2, 5, 7, 1, 9, 3, 6, 8, 8}
> a2[] = {1, 10, 11}
> **Output:** 1 1 2 2 3 5 6 7 8 8 9

Approach: We can print the elements of a1[] according to the order defined by a2[] using **map in c++** in O(mlog(n)) time. We traverse through a1[] and store the frequency of every number in a map. Then we traverse through a2[] and check if the number is present in the map. If the number is present, then print it that many times and erase the number from the map. Print the rest of the numbers present in the map sequentially as numbers are stored in the map in sorted order.

Below is the implementation of the above approach:

```

// A C++ program to print an array according
// to the order defined by another array
#include <bits/stdc++.h>
using namespace std;

// Function to print an array according
// to the order defined by another array
void print_in_order(int a1[], int a2[], int n, int m)
{
    // Declaring map and iterator
    map<int, int> mp;
    map<int, int>::iterator itr;

    // Store the frequncy of each
    // number of a1[] int the map
    for (int i = 0; i < n; i++)
        mp[a1[i]]++;

    // Traverse through a2[]
    for (int i = 0; i < m; i++) {
        // Check whether number
        // is present in map or not

        if (mp.find(a2[i]) != mp.end()) {
```

```
        if (mp.find(a2[i]) != mp.end()) {
            itr = mp.find(a2[i]);

            // Print that number that
            // many times of its frequncy
            for (int j = 0; j < itr->second; j++)
```

Run

Output:

```
2 2 1 1 8 8 3 5 6 7 9
```

multimap in C++ STL

The **multimap** container in C++ is similar to the map container with an addition that a multimap can have multiple key-value pairs with the same key. Rather than each element is unique, the key-value and mapped value pair have to be unique in this case.

Multimap is also implemented using Red-Black trees and hence the basic operations like search, insert, delete works in O(LogN) time for multimap as well.

Some Basic Functions associated with multimap:

- **begin()** – Returns an iterator to the first element in the multimap.
- **end()** – Returns an iterator to the theoretical element that follows the last element in the multimap.
- **size()** – Returns the number of elements in the multimap.
- **empty()** – Returns whether the multimap is empty.
- **insert(keyvalue,multimapvalue)** – Adds a new element to the multimap.

Note: We also have seen an operator **"[]"** which was used to access and also insert elements in the map container. However, multimap doesn't allow the use of member access operator "[]" as there can be multiple key-value pairs with the same key.

Below program illustrate the multimap in C++ STL:

```

#include<iostream>
#include<algorithm>
#include<map>

using namespace std;

int main()
{
    multimap<int, int> mp;

    mp.insert({10,20});
    mp.insert({5, 50});
    mp.insert({10,25});

    for(auto x:mp)
        cout<<x.first<<" "<<x.second<<endl;

    return 0;
}

```

Run

Output:

```
5 50
10 20
10 25
```

Another difference between map and multimap is that for a map container the count() function returns either 1 or 0 depending on whether a key exists in the map or not whereas in a multimap container, *the count() function returns the number of occurrences* of key in the multimap passed to it as a parameter.

Also, the erase() function in multimap is also similar to that of map container, the difference here is as there can exist multiple key-value pairs with same key, therefore, in multimap, the erase() function will erase all of the key-value pairs of the key provided.

Below program illustrate the above two methods:

```

#include<iostream>
#include<algorithm>
#include<map>

using namespace std;

int main()
{
    multimap<int, int> mp;

    mp.insert({10,20});
    mp.insert({5, 50});
    mp.insert({10,25});

    cout<<"Count of the key 10: "<<mp.count(10);

    // Erase the key 10
    mp.erase(10);

    cout<<"\nCount of the key 10: "<<mp.count(10);

    return 0;
}

```

Run

Output:

```
Count of the key 10: 2
Count of the key 10: 0
```

lower_bound() function of multimap

The **multimap::lower_bound(k)** is a built-in function in C++ STL which returns an iterator pointing to the key in the container which is equivalent to k passed in the parameter. In case k is not present in the multimap container, the function returns an iterator pointing to the immediate next element which is just greater than k. If the key passed in the parameter exceeds the maximum key in the container, then the iterator returned points to key+1 and element = 0.

```

// C++ program to illustrate
// multimap::lower_bound() function
#include <bits/stdc++.h>
using namespace std;

int main()
{

    // initialize container
    multimap<int, int> mp;

    // insert elements in random order
    mp.insert({ 2, 30 });
    mp.insert({ 1, 40 });
    mp.insert({ 2, 60 });
    mp.insert({ 2, 20 });
    mp.insert({ 1, 50 });
    mp.insert({ 4, 50 });

    // when 2 is present
    auto it = mp.lower_bound(2);
    cout << "The lower bound of key 2 is ";
    cout << (*it).first << " "
        << (*it).second << endl;

    // when 3 is not present
    it = mp.lower_bound(3);
    cout << "The lower bound of key 3 is ";
    cout << (*it).first << " "
```

Run

Output:

```
The lower bound of key 2 is 2 30
The lower bound of key 3 is 4 50
The lower bound of key 3 is 6 0
```

upper_bound() function of multimap

The **multimap::upper_bound(k)** is a built-in function in C++ STL which returns an iterator pointing to the immediate next element which is just greater than k. If the key passed in the parameter exceeds the maximum key in the container, then the iterator returned points to key+1 and element=0.

```cpp

// C++ program to illustrate
// multimap::upper_bound() function
#include <bits/stdc++.h>
using namespace std;

int main()
{
    // initialize container
    multimap<int, int> mp;

    // insert elements in random order
    mp.insert({ 2, 30 });
    mp.insert({ 1, 40 });
    mp.insert({ 2, 60 });
    mp.insert({ 2, 20 });
    mp.insert({ 1, 50 });
    mp.insert({ 4, 50 });

    // when 2 is present
    auto it = mp.upper_bound(2);
    cout << "The upper bound of key 2 is ";
    cout << (*it).first << " " << (*it).second << endl;

    // when 3 is not present
    it = mp.upper_bound(3);
    cout << "The upper bound of key 3 is ";
    cout << (*it).first << " " << (*it).second << endl;

    // when 5 is exceeds the maximum key
```

Run

Output:

```
The upper bound of key 2 is 4 50
The upper bound of key 3 is 4 50
The upper bound of key 5 is 6 0
```

equal_range() function of multimap

The **multimap::equal_range()** is a built-in function in C++ STL which returns an iterator of pairs. The pair refers to the bounds of a range that includes all the elements in the container which have a key equivalent to k. If there are no matches with key K, the range returned is of length 0 with both iterators pointing to the first element that has a key consideration to go after k according to the container's internal comparison object (key_comp).

```cpp

// C++ program to illustrate the
// equal_range() function
#include <bits/stdc++.h>
using namespace std;

int main()
{

    // initialize container
    multimap<int, int> mp;
    // insert elements in random order
    mp.insert({ 2, 30 });
    mp.insert({ 1, 40 });
    mp.insert({ 3, 60 });
    mp.insert({ 1, 20 });
    mp.insert({ 5, 50 });

    // Stores the range of key 1
    auto it = mp.equal_range(1);

    cout << "The multimap elements of key 1 is : \n";
    cout << "KEY\tELEMENT\n";

    // Prints all the elements of key 1
    for (auto itr = it.first; itr != it.second; ++itr) {
        cout << itr->first
             << '\t' << itr->second << '\n';
    }
    return 0;
```

Run

Output:

```
The multimap elements of key 1 is :
KEY     ELEMENT
1     40
1     20
```

Sample Problem: Implementing Dictionary using Multimap

Problem: Implement a dictionary using multimap by storing a few words having multiple meanings. For eg., "Apple" is both a fruit and the name of a company, "Kiwi" is both a fruit and a flightless bird.

Example:

```
Apple: A fruit

Apple: A company



Kiwi: A fruit

Kiwi: A flightless bird.
```

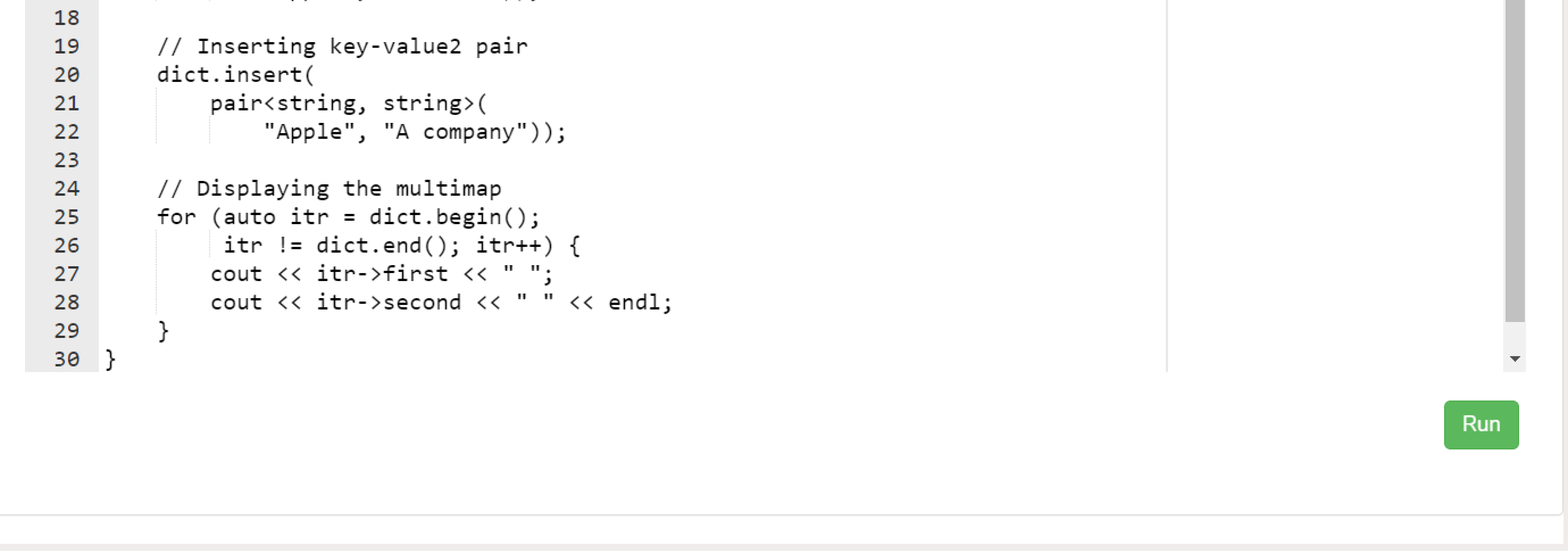
Approach: Although, using Multimap is not the best way of implementing a dictionary as it can be more optimized by using the Trie data structure, but we have used this to explain one of the many uses of a multimap. To do this, just create a multimap and store in it a key and its multiple values. There is a single key which can have more than one values. The insert function can be used to insert multiple pair of values into a multimap having the same key but different values. Then this multimap can be displayed using the begin and end operation implemented through a for loop.

```


// C++ code to implement dictionary
// using a multimap
#include <bits/stdc++.h>
using namespace std;

// Driver method
int main()
{
    // Creating a multimap
    multimap<string, string> dict;

    // Inserting key-value1 pair
    dict.insert(
        pair<string, string>(
            "Apple", "A fruit"));
```



```
    // Inserting key-value2 pair
    dict.insert(
        pair<string, string>(
            "Apple", "A company"));

    // Displaying the multimap
    for (auto itr = dict.begin();
         itr != dict.end(); itr++) {
        cout << itr->first << " ";
        cout << itr->second << " " << endl;
    }
}
```