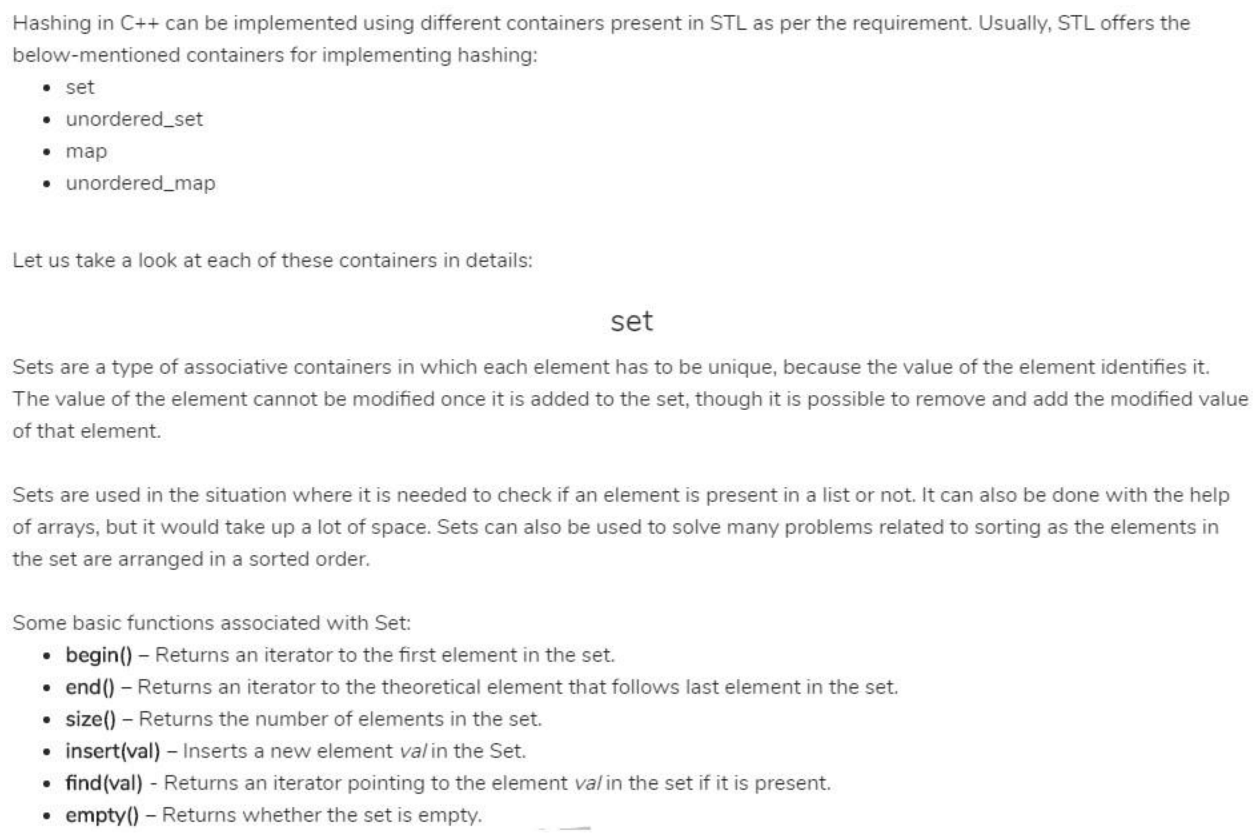
Hashing in C++ using STL



Hashing in C++ can be implemented using different containers present in STL as per the requirement. Usually, STL offers the below-mentioned containers for implementing hashing:

- set
- unordered_set
- map
- unordered_map

Let us take a look at each of these containers in details:

set

Sets are a type of associative containers in which each element has to be unique, because the value of the element identifies it. The value of the element cannot be modified once it is added to the set, though it is possible to remove and add the modified value of that element.

Sets are used in the situation where it is needed to check if an element is present in a list or not. It can also be done with the help of arrays, but it would take up a lot of space. Sets can also be used to solve many problems related to sorting as the elements in the set are arranged in a sorted order.

Some basic 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 last element in the set.
- **size()** – Returns the number of elements in the set.
- **insert(val)** – Inserts a new element *val* in the Set.
- **find(val)** - Returns an iterator pointing to the element *val* in the set if it is present.
- **empty()** – Returns whether the set is empty.

Implementation:

```cpp
// C++ program to illustrate hashing using set in CPP STL
#include <iostream>
#include <set>
#include <iterator>
using namespace std;
int main()
{
    // empty set container
    set <int> s;
    // List of elements
    int arr[] = {40, 20, 60, 30, 50, 50, 10};
    // Insert the elements of the List to the set
    for(int i = 0; i < 7; i++)
        s.insert(arr[i]);
    // Print the content of the set
    // The elements of the set will be sorted without any duplicates
    cout<<"The elements in the set are: \n";
    for(auto itr=s.begin(); itr!=s.end(); itr++)
    {
        cout<<*itr<<" ";
    }
    // Check if 50 is present in the set
    if(s.find(50)!=s.end())
    {
        cout<<"\n\n50 is present";
    }
    else
    {
        cout<<"\n\n50 is not present";
    }

    return 0;
}
```

Output:

```
The elements in the set are:
10 20 30 40 50 60

50 is present
```

unordered_set

The unordered_set container is implemented using a hash table where keys are hashed into indices of this hash table so it is not possible to maintain any order. All operation on unordered_set takes constant time O(1) on an average which can go up to linear time in the worst case which depends on the internally used hash function but practically they perform very well and generally provide constant time search operation.

The unordered-set can contain key of any type – predefined or user-defined data structure but when we define key of a user-defined type, we need to specify our comparison function according to which keys will be compared.

set vs unordered_set

- Set is an ordered sequence of unique keys whereas unordered_set is a set in which key can be stored in any order, so unordered.
- Set is implemented as a balanced tree structure that is why it is possible to maintain order between the elements (by specific tree traversal). The time complexity of set operations is O(Log n) while for unordered_set, it is O(1).

Note: Like set containers, the Unordered_set also allows only unique keys.

Implementation:

```cpp
// C++ program to illustrate hashing using unordered_set in CPP STL
#include <iostream>
#include <unordered_set>
#include <iterator>
using namespace std;
int main()
{
    // empty set container
    unordered_set <int> s;
    // List of elements
    int arr[] = {40, 20, 60, 30, 50, 50, 10};
    // Insert the elements of the List to the set
    for(int i = 0; i < 7; i++)
        s.insert(arr[i]);
    // Print the content of the unordered set
    // The elements of the set will not be sorted without any duplicates
    cout<<"The elements in the unordered_set are: \n";
    for(auto itr=s.begin(); itr!=s.end(); itr++)
    {
        cout<<*itr<<" ";
    }
    // Check if 50 is present in the set
    if(s.find(50)!=s.end())
    {
        cout<<"\n\n50 is present";
    }
    else
    {
        cout<<"\n\n50 is not present";
    }

    return 0;
}

```

```
The elements in the unordered_set are:
10 50 30 60 40 20

50 is present
```

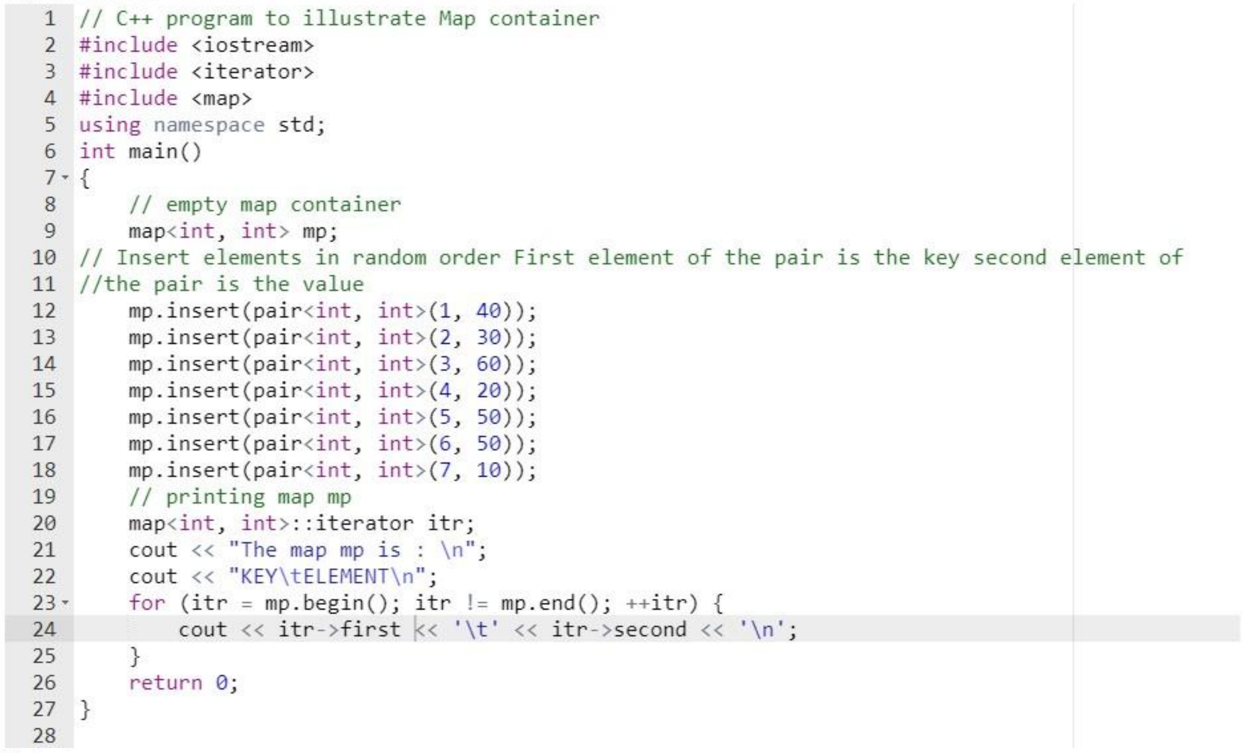
Map container

As a set, the Map container is also associative and stores elements in an ordered way but Maps store elements in a mapped fashion. Each element has a key value and a mapped value. No two mapped values can have the same key values.

Some basic functions associated with Map:

- **begin()** – Returns an iterator to the first element in the map.
- **end()** – Returns an iterator to the theoretical element that follows last element in the map.
- **size()** – Returns the number of elements in the map.
- **empty()** – Returns whether the map is empty.
- **insert({keyvalue, mapvalue})** – Adds a new element to the map.
- **erase(iterator position)** – Removes the element at the position pointed by the iterator
- **erase(const g)**– Removes the key value 'g' from the map.
- **clear()** – Removes all the elements from the map.

Implementation:



```cpp
// C++ program to illustrate Map container
#include <iostream>
#include <iterator>
#include <map>
using namespace std;
int main()
{
    // empty map container
    map<int, int> mp;
// Insert elements in random order First element of the pair is the key second element of
//the pair is the value
    mp.insert(pair<int, int>(1, 40));
    mp.insert(pair<int, int>(2, 30));
    mp.insert(pair<int, int>(3, 60));
    mp.insert(pair<int, int>(4, 20));
    mp.insert(pair<int, int>(5, 50));
    mp.insert(pair<int, int>(6, 50));
    mp.insert(pair<int, int>(7, 10));
    // printing map mp
    map<int, int>::iterator itr;
    cout << "The map mp is : \n";
    cout << "KEY\tELEMENT\n";
    for (itr = mp.begin(); itr != mp.end(); ++itr) {
        cout << itr->first << '\t' << itr->second << '\n';
    }
    return 0;
}

```

Output:

```
The map mp is :
KEY    ELEMENT
1    40
2    30
3    60
4    20
5    50
6    50
7    10
```

unordered_map Container

The unordered_map is an associated container that stores elements formed by a combination of key value and a mapped value. The key value is used to uniquely identify the element and mapped value is the content associated with the key. Both key and value can be of any type predefined or user-defined.
Internally unordered_map is implemented using Hash Table, the key provided to map are hashed into indices of a hash table that is why the performance of data structure depends on hash function a lot but on an average, the cost of search, insert and delete from hash table is O(1).

Implementation:

```cpp

// C++ program to demonstrate functionality of unordered_map
#include <iostream>
#include <unordered_map>
using namespace std;

int main()
{
    // Declaring umap to be of <string, int> type
    // key will be of string type and mapped value will
    // be of double type
    unordered_map<string, int> umap;

    // inserting values
    umap.insert({"GeeksforGeeks", 10});
    umap.insert({"Practice", 20});
    umap.insert({"Contribute", 30});

    // Traversing an unordered map
      cout << "The map umap is : \n";
    cout << "KEY\t\tELEMENT\n";
    for (auto itr = umap.begin(); itr!= umap.end(); itr++)
      cout << itr->first << '\t' << itr->second << '\n';
      return 0;
}

```

Output:

```
The map umap is :
KEY         ELEMENT
Contribute    30
GeeksforGeeks    10
Practice    20
```

unordered_map vs unordered_set

In unordered_set, we have only key, no value, these are mainly used to see presence/absence in a set. For example, consider the problem of counting frequencies of individual words. We can't use unordered_set (or set) as we can't store counts.

unordered_map vs map

map (like set) is an ordered sequence of unique keys whereas in the unordered_map key can be stored in any order, so unordered. A map is implemented as a balanced tree structure that is why it is possible to maintain order between the elements (by specific tree traversal). The time complexity of map operations is O(Log n) while for unordered_set, it is O(1) on average.