unordered\_map in STL and its applications

unordered\_map is an associated container that stores elements formed by 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 hash table that is why performance of data structure depends on hash function a lot but on an average the cost of look-up from hash table is O(1). In worst case unordered\_map may require O(n) time but practically it is much faster and outperforms tree based maps.

**unordered\_map vs**[**unordered\_set**](https://www.geeksforgeeks.org/unorderd_set-stl-uses/)**:**  
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**](http://geeksquiz.com/map-associative-containers-the-c-standard-template-library-stl/)**:**  
map (like [set](http://geeksquiz.com/set-associative-containers-the-c-standard-template-library-stl/)) is an ordered sequence of unique keys whereas in unordered\_map key can be stored in any order, so unordered.  
Map is implemented as balanced tree structure that is why it is possible to maintain an order between the elements (by specific tree traversal). Time complexity of map operations is O(Log n) while for unordered\_set, it is O(1) on average.

**Methods on unordered\_map**  
A lot of function are available which work on unordered\_map. most useful of them are – operator =, operator [], empty and size for capacity, begin and end for iterator, find and count for lookup, insert and erase for modification.  
The C++11 library also provides function to see internally used bucket count, bucket size and also used hash function and various hash policies but they are less useful in real application.  
We can iterate over all elements of unordered\_map using Iterator. Initialization, indexing and iteration is shown in below sample code :

|  |
| --- |
| // C++ program to demonstrate functionality of unordered\_map  #include <bits/stdc++.h>  using namespace std;    int main()  {      // Declaring umap to be of <string, double> type      // key will be of string type and mapped value will      // be of double type      unordered\_map<string, double> umap;        // inserting values by using [] operator      umap["PI"] = 3.14;      umap["root2"] = 1.414;      umap["root3"] = 1.732;      umap["log10"] = 2.302;      umap["loge"] = 1.0;        // inserting value by insert function      umap.insert(make\_pair("e", 2.718));        string key = "PI";        // If key not found in map iterator to end is returned      if (umap.find(key) == umap.end())          cout << key << " not found\n\n";        // If key found then iterator to that key is returned      else          cout << "Found " << key << "\n\n";        key = "lambda";      if (umap.find(key) == umap.end())          cout << key << " not found\n";      else          cout << "Found " << key << endl;        //    iterating over all value of umap      unordered\_map<string, double>:: iterator itr;      cout << "\nAll Elements : \n";      for (itr = umap.begin(); itr != umap.end(); itr++)      {          // itr works as a pointer to pair<string, double>          // type itr->first stores the key part  and          // itr->second stroes the value part          cout << itr->first << "  " << itr->second << endl;       }  } |

Run on IDE

Output :

Found PI

lambda not found

All Elements :

loge 1

log10 2.302

root3 1.732

e 2.718

root2 1.414

PI 3.14

**A practical problem based on unordered\_map** – given a string of words, find frequencies of individual words.

Input : str = "geeks for geeks geeks quiz practice qa for";

Output : Frequencies of individual words are

(practice, 1)

(for, 2)

(qa, 1)

(quiz, 1)

(geeks, 3)

Below is C++ solution using unordered\_map.

|  |
| --- |
| // C++ program to find freq of every word using  // unordered\_map  #include <bits/stdc++.h>  using namespace std;    // Prints frequencies of individual words in str  void printFrequencies(const string &str)  {      // declaring map of <string, int> type, each word      // is mapped to its frequency      unordered\_map<string, int> wordFreq;        // breaking input into word using string stream      stringstream ss(str);  // Used for breaking words      string word; // To store individual words      while (ss >> word)          wordFreq[word]++;        // now iterating over word, freq pair and printing      // them in <, > format      unordered\_map<string, int>:: iterator p;      for (p = wordFreq.begin(); p != wordFreq.end(); p++)          cout << "(" << p->first << ", " << p->second << ")\n";  }    // Driver code  int main()  {      string str = "geeks for geeks geeks quiz "                   "practice qa for";      printFrequencies(str);      return 0;  } |

Run on IDE

Output :

(practice, 1)

(for, 2)

(qa, 1)

(quiz, 1)

(geeks, 3)

#include <bits/stdc++.h>

using namespace std;

struct HASH{

size\_t operator()(const pair<int,int>&x)const{

return ((long long)x.first)^(((long long)x.second)<<32);

}

};

unordered\_map<pair<int,int>,int,HASH>m;

int main(){

auto T=clock();

for(int i=0;i<1000000;i++)

m[make\_pair (1, i)] = i;

cout<<double(clock()-T)/CLOCKS\_PER\_SEC<<'\n';

}

In this article we will discuss the different ways to initialize an unordered\_map.

unordered\_map provides different overloaded constructors. We will use some of them to initialize an unordered\_set in 3 different ways i.e.

* Initializing through an initializer\_list
* Initializing through a iterable range.
* Initializing through an another map.

Lets see them in detail,

## Initializing an unordered\_map using initializer\_list

unordered\_map provides an over loaded constructor that can accept an initializer\_list as an argument and can initialize  an unordered\_map.

Let’s create an unordered\_map of key type string and value type int. While creation it will initialized with 3 elements through an initializer\_list i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // Initialize an unordered\_map through initializer\_list    std::unordered\_map<std::string, int> wordMap({    { "First", 1 },    { "Second", 2 },    { "Third", 3 }                                      }); |

## Initializing an unordered\_map through a range

We can also initialize an unordered\_map through a range of std::pair elements i.e

|  |  |
| --- | --- |
| 1  2  3 | // Initialize an unordered\_map through another range of elements of type std::pair    std::unordered\_map<std::string, int> wordMap\_2(wordMap.begin(),        wordMap.end()); |

## Initializing an unordered\_map through other unordered\_map

We can also initialize an unordered\_map through an existing unordered\_map i.e.

|  |  |
| --- | --- |
| 1  2 | // Initialize an unordered\_map through other unordered\_map    std::unordered\_map<std::string, int> wordMap\_3(wordMap); |

Complete working example, is as follows,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | #include <iostream>  #include <unordered\_map>  #include <string>    int main()  {  // Initialize an unordered\_map through initializer\_list  std::unordered\_map<std::string, int> wordMap(  {  { "First", 1 },  { "Second", 2 },  { "Third", 3 } });    // Iterate over an unordered\_map and display elements  for (std::pair<std::string, int> element : wordMap)  std::cout << element.first << " :: " << element.second << std::endl;    std::cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << std::endl;    // Initialize an unordered\_map through another range of elements of type std::pair  std::unordered\_map<std::string, int> wordMap\_2(wordMap.begin(),  wordMap.end());    // Iterate over an unordered\_map and display elements  for (std::pair<std::string, int> element : wordMap\_2)  std::cout << element.first << " :: " << element.second << std::endl;    std::cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << std::endl;    // Initialize an unordered\_map through other unordered\_map  std::unordered\_map<std::string, int> wordMap\_3(wordMap);    // Iterate over an unordered\_map and display elements  for (std::pair<std::string, int> element : wordMap\_3)  std::cout << element.first << " :: " << element.second << std::endl;    return 0;  } |

**Ouput:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Third :: 3  First :: 1  Second :: 2  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Second :: 2  Third :: 3  First :: 1  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Third :: 3  First :: 1  Second :: 2 |

**How check if a given key exists in a Map | C++**

[Varun](https://thispointer.com/author/admin/) January 29, 2017 [How check if a given key exists in a Map | C++](https://thispointer.com/how-check-if-a-given-key-exists-in-a-map-c/)2017-01-29T16:06:52+00:00[C++](https://thispointer.com/category/c/), [count](https://thispointer.com/category/c/stl-c/stdmap/count/), [find](https://thispointer.com/category/c/stl-c/stdmap/find/), [std::map](https://thispointer.com/category/c/stl-c/stdmap/), [STL](https://thispointer.com/category/c/stl-c/) [1 Comment](https://thispointer.com/how-check-if-a-given-key-exists-in-a-map-c/#comments)

In this article we will discuss how to find if a given key exists in map or not.

Map internally store elements in Key-Value pair. It provides 2 member functions to check if a given key exists in map i.e.

* std::map::find
* std::map::count

**Check if map contains a key using std::map::count**

std::map provides a member function count() i.e.

|  |  |
| --- | --- |
| 1 | size\_type count (const key\_type& K) const; |

It finds & returns the count of number of elements in map with key K. As map contains elements with unique key only. So, it will return 1 if key exists else 0.

Suppose we have map of string & int i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | std::map<std::string, int> wordMap = {  { "is", 6 },  { "the", 5 },  { "hat", 9 },  { "at", 6 }  }; |

Let’s check if key ‘hat’ exists in the map or not i.e.

|  |  |
| --- | --- |
| 1  2  3  4 | if (wordMap.count("hat") > 0)  {  std::cout << "'hat' Found" << std::endl;  } |

Complete example as follows,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | #include <iostream>  #include <map>  #include <string>    int main() {    // Map of string & int i.e. words as key & there  // occurrence count as values  std::map<std::string, int> wordMap = {  { "is", 6 },  { "the", 5 },  { "hat", 9 },  { "at", 6 }  };      // Check if key 'hat' exists in the map  if (wordMap.count("hat") > 0)  {  std::cout << "'hat' Found" << std::endl;  }  else  {  std::cout << "'hat' Not Found" << std::endl;  }    // Check if key 'hello' exists in the map  if (wordMap.count("hello") > 0)  {  std::cout << "'hello' Found" << std::endl;  }  else  {  std::cout << "'hello' Not Found" << std::endl;  }    return 0;  } |

**Output**

|  |  |
| --- | --- |
| 1  2 | 'hat' Found  'hello' Not Found |

std::map::count just tells if the given key exists in map or not. But what if we also want to access the value  
associated with the given key. For that map provides another member function i.e. std::map::find

**Check if map contains a key using std::map::find**

std::map provides a member function find() i.e.

|  |  |
| --- | --- |
| 1 | iterator find (const key\_type& k); |

It checks if any element with given key ‘k’ exists in the map and if yes then it returns its iterator else  
it returns the end of map.

Suppose we have map of string & int i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | std::map<std::string, int> wordMap = {  { "is", 6 },  { "the", 5 },  { "hat", 9 },  { "at", 6 }  }; |

Let’s check if key ‘hat’ exists in the map or not i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | // Create an iterator of map  std::map<std::string, int>::iterator it;    // Find the element with key 'hat'  it = wordMap.find("hat");    // Check if element exists in map or not  if (it != wordMap.end())  {  // Element with key 'hat' found  std::cout << "'hat' Found" << std::endl;    // Access the Key from iterator  std::string key = it->first;  // Access the Value from iterator  int value = it->second;    std::cout << "key = " << key << " :: Value = " << key << std::endl;  } |

Complete example as follows,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59 | #include <iostream>  #include <map>  #include <string>  #include <iterator>  #include <algorithm>    int main() {    // Map of string & int i.e. words as key & there  // occurrence count as values  std::map<std::string, int> wordMap = {  { "is", 6 }, { "the", 5 },  { "hat", 9 }, { "at", 6 }  };    // Create an iterator of map  std::map<std::string, int>::iterator it;    // Find the element with key 'hat'  it = wordMap.find("hat");    // Check if element exists in map or not  if (it != wordMap.end()) {  // Element with key 'hat' found  std::cout << "'hat' Found" << std::endl;    // Access the Key from iterator  std::string key = it->first;  // Access the Value from iterator  int value = it->second;    std::cout << "key = " << key << " :: Value = " << value << std::endl;  } else {  // Element with key 'hat' Not Found  std::cout << "'hat' Not Found" << std::endl;  }    // Find the element with key 'hello'  it = wordMap.find("hello");    // Check if element exists in map or not  if (it != wordMap.end()) {  // Element with key 'hello' found  std::cout << "'hello' Found" << std::endl;    // Access the Key from iterator  std::string key = it->first;  // Access the Value from iterator  int value = it->second;    std::cout << "key = " << key << " :: Value = " << key << std::endl;    } else {  // Element with key 'hello' Not Found  std::cout << "'hello' Not Found" << std::endl;  }    return 0;  } |

**Output:**

|  |  |
| --- | --- |
| 1  2  3 | 'hat' Found  key = hat :: Value = 9  'hello' Not Found |

To Compile the above code use following command,

***g++ –std=c++11 example.cpp***

# C++ Map : Operator [] – Usage Details

[Varun](https://thispointer.com/author/admin/) February 17, 2017 [C++ Map : Operator [] – Usage Details](https://thispointer.com/c-map-operator-usage-details/)2017-02-17T23:03:33+00:00[operator[]](https://thispointer.com/category/c/stl-c/stdmap/operator/), [std::map](https://thispointer.com/category/c/stl-c/stdmap/), [STL](https://thispointer.com/category/c/stl-c/) [1 Comment](https://thispointer.com/c-map-operator-usage-details/#comments)

In this article we will discuss how to use operator [] with map in C++.

std::map has Operator[] i.e.

|  |  |
| --- | --- |
| 1 | mapped\_type& operator[] (const key\_type& k); |

operator [] works in **Find or Create Mode** i.e. when called with a key K, it will try to look for element with given key K and can go in below 2 directions i.e.

* If any element is found with key K, then it will return the reference of its value.
* But if there is no element in map with K, then it will create a new element in map with key K and assign default value of value\_type in its value field. Then it will return the value of newly created element as reference.

Let’s discuss it in detail,

## Element lookup with Operator[]

Suppose we have a map of String and int i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // Map of string & int i.e. words as key & there  // occurrence count as values  std::map<std::string, int> wordMap = {  { "is", 6 },  { "the", 5 }  }; |

Now suppose, we call the operator [] on above map with key value ‘Hello’.

|  |  |
| --- | --- |
| 1  2  3  4  5 | // As there is no element with key 'Hello'  // So, operator [] will create a new element in map  // with key as 'Hello' and default value, which in this case  // will be 0.  int & value1 = wordMap["Hello"]; |

As this key ‘Hello’ is not present in map. So operator [] will create a new entry for it with Key ‘Hello’ & Value as default value of int i.e. 0.

Also, will return its value by reference.

## Overriding values with operator []

operator [] returns value by reference, so assigning value by calling operator[] can change the value i.e

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // As [] returns the value by reference, therefore  // it will override the new value.  wordMap["is"] = 4;    // Override the value of element with key 'is'  wordMap["is"] = 4; |

Checkout the complete example as follows

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | #include <iostream>  #include <map>  #include <string>    int main() {    // Map of string & int i.e. words as key & there  // occurrence count as values  std::map<std::string, int> wordMap = {  { "is", 6 },  { "the", 5 }  };  // As there is no element with key 'Hello'  // So, operator [] will create a new element in map  // with key as 'Hello' and default value, which in this case  // will be 0.  int & value1 = wordMap["Hello"];    // Display the value  std::cout<<"value of 'Hello' = "<<value1<<std::endl;    // Fetch the value for key 'is'  // As key already exist therefore will not create new entry  // Will just return the reference of value  int & value2 = wordMap["is"];    // Display the value  std::cout<<"value of 'is' = "<<value2<<std::endl;      // As [] returns the value by reference, therefore  // it will override the new value.  wordMap["is"] = 4;    int & value3 = wordMap["is"];    // Display the value  std::cout<<"value of 'is' = "<<value3<<std::endl;      // Override the value of element with key 'is'  wordMap["is"] = 4;    // Will create a new wntry with Key 'Thanks' and override its  // default value 0 with 10.  wordMap["Thanks"] = 10;      std::cout<< "\*\*\*\*\*\*\*\*\*\*\*Map Entries\*\*\*\*\*\*\*\*\*\*\*" <<std::endl;  // Print the map elements  for (auto elem : wordMap)  std::cout << elem.first << " :: " << elem.second << std::endl;    return 0;  } |

**Output**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | value of 'Hello' = 0  value of 'is' = 6  value of 'is' = 4  \*\*\*\*\*\*\*\*\*\*\*Map Entries\*\*\*\*\*\*\*\*\*\*\*  Hello :: 0  Thanks :: 10  is :: 4  the :: 5 |

## Using operator[] with User Defined objects as Value

As operator[] can create new entry with default value. So, if we are using a User defined class or struct as value in a map, then its must to have default constructor in it.

Let’s see an example

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | #include <iostream>  #include <map>  #include <string>    struct Occurance  {  int count;    // Default Constructor  // Un Comment it to compile the code  /\*  Occurance()  {  this->count = 0;  }  \*/    // Parametrized constructor  Occurance(int count)  {  this->count = count;  }  };  int main() {    // Map of string & int i.e. words as key & there  // occurrence count as values  std::map<std::string, Occurance> wordMap = {  { "is", Occurance(6) },  { "the", Occurance(5) }  };      // As key is not in map, so operator[] will create new entry  // With default value of value field. Therefore, Will compile  // only if Occurance sruct has default constructor.  Occurance occur = wordMap["Hello"];    return 0;  } |

As key is not in map, so operator[] will create new entry with default value of value field. Therefore, above example will compile only if **Occurance** struct has default constructor. Un-comment it to compile succussfuly.

# Unordered\_map Usage Tutorial and Example

[Varun](https://thispointer.com/author/admin/) July 23, 2016 [Unordered\_map Usage Tutorial and Example](https://thispointer.com/unordered_map-usage-tutorial-and-example/)2016-07-23T20:45:54+00:00[C++ 11](https://thispointer.com/category/c/c-11/), [unordered\_map](https://thispointer.com/category/c/c-11/unordered_map/) [No Comment](https://thispointer.com/unordered_map-usage-tutorial-and-example/#respond)

In this article we will discuss what is unordered\_map and how it use it.

## What is an unordered\_map

Unordered\_map provides a functionality of map i.e. it store the elements in key value pair and with unique key only.

Unordered\_map internally uses the hashing to achieve this. It internally uses a hash table to implement this hashing feature. If you want to know more about hashing check following article,  
[What is Hashing and Hash Table](https://thispointer.com/what-is-hashing-and-hash-table/)

In an unordered\_map elements are stored in a key value pair combination. But elements are stored in arbitrary order unlike associative containers where elements were stored in sorted order of keys.

Let’s see an example by creating an unordered\_map of string and int, to maintain the repeated count of a word i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | #include <iostream>  #include <unordered\_map>  #include <string>    int main()  {  // Create an empty unordered\_map  std::unordered\_map<std::string, int> wordMap;    // Insert Few elements in map  wordMap.insert( { "First", 1 });  wordMap.insert( { "Second", 2 });  wordMap.insert( { "Third", 3 });    // Overwrite value of an element  wordMap["Third"] = 8;    // Iterate Over the unordered\_map and display elements  for (std::pair<std::string, int> element : wordMap)  std::cout << element.first << " :: " << element.second << std::endl;    return 0;  } |

**Output:**

|  |  |
| --- | --- |
| 1  2  3 | Third :: 8  Second :: 2  First :: 1 |

As we can see above, elements are stored in arbitrary order.

## How unordered\_map store elements?

The reason because unordered\_map does the hashing i.e. whenever we try to insert an element in a unordered\_map, it internally does the following steps,

* First hash of key is calculated using Hasher function and then on the basis of that hash an appropriate bucket is choose.
* Once bucket is identified then it compares the key with key of each element inside the bucket using Comparator function to identify if given element is a duplicate or not.
* If its not a duplicate then only it stores the element in that bucket

Therefore, there is no specific order in which elements are stored internally.

## Advantage of Unordered\_map

The basic advantage of using unordered\_map instead of associative map is the searching efficiency. In an unordered\_map complexity to search for an element is O(1) if hash code are chosen efficiently.