Sabanci University

Faculty of engineering and Natural Sciences

CS 300 Data Structures

Standard Library

This document will provide a brief introduction to the unoredred_map class in the C++ standard library, which is a templated implementation of **hashtables**. In particular, it will provide some explanations of the data structure and how it is used, a list of important functions and their time complexities, and some coding examples.

`std::unordered_map` (hashtable)

This class is a templated implementation of the hashtable data structure. The easiest way to understand it, is to think of it like a dictionary in Python. It allows the user to access elements using keys in constant time (i.e., O(1)). It is also templated, which means the user can use any data type for keys and for values.

It has two templated types, the type of the key, and the type of the value. The programmer can mix and match those types depending on their need. For example, the following declaration will create an unordered_map where the key used to access the elements is a string, and the value that each element will contain is an integer:

```
unordered_map<string, int> grades;
grades["Science"] = 98; // inserts an element with key "Science"
                        // and value 98
grades["Math"] = 95;    // inserts an element with key "Math"
                        // and value 95
grades["Science"] = 92; // Changes the value of the element with
                        // key "Science" to be 92
```

And here is an example of an unoredered_map in which the key is an integer and the value that each element will contain is a string:

```
unordered_map<int, string> street_number_to_name;
street_number_to_name[4] = "Fatih";
street_number_to_name[99] = "İstiklal";
```

Important Functions

The following are some of the most useful functions when working with unordered_maps. You can find more here.

For all the upcoming definitions, assume that we have an unordered_map with type K for its key, and type D for its elements:

```
unordered_map<K, D> example;
```

`operator[](K key_value);` **Complexity = O(1)**

If an element with the key *key_value* exists in *example*, this function will return a reference to the element. If no element with this key exists, a new element will be added to the map whose key will be equal to *key_value* and a reference to the element will be returned. Please note that the element will be initialized using the default constructor of its type.

Example:

```
#include <unordered_map>
using namespace std;

int main(){
      unordered_map<char, int> grade_boundaries;
      grade_boundaries['F']; // adds an element with
                             // key 'F' but its value will be random
      grade_boundaries['A'] = 99.98;  // adds an element with key 'A'
                                      // and directly sets that
                                      // elements value to 99.98
      grade_boundaries['F'] = 89; // an element with
                                  // the key 'F' already exists,
                                  // so no new elements are added
```

```
                                    // to the unorederd_map and a
                                    // reference to the element
                                    // with key 'F' is returned
}
```

`example.find(K to_search)` **Complexity = O(1)**

Will search the unordered_map *example* for an element whose key is equal to the variable *to_search*. If an element with that key exists in the map, returns an iterator that points at it, if it doesn't exist, returns `example.end()`.

Example:

```
#include <unordered_map>
#include <iostream>
using namespace std;

int main(){
      unordered_map<short, short> a2b;
      a2b[1];
      // since an element with key 1 exists in
      // the map, find() will return
      // an iterator != a2b.end()
      if (a2b.find(1) == a2b.end()){
         cout << "1 is not in the map\n";
      }
      // No element with key 2 exists in the
      // map. find() will return a2b.end()
      if (a2b.find(2) == a2b.end()){
         cout << "2 is not in the map\n";
      }
      return 0;
}
```

Output:

```
2 is not in the map
```

`example.erase(K to_search)` **Complexity = O(1)**

Will remove from `example` the element with the key `to_search`.

Example

```
#include <unordered_map>
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main(){
      unordered_map<string, vector<string>> siblings;
      // Add an element with the key "Myself".
      // The element's value is an empty vector
siblings["Myself"];
     // Add the string "my brother" to the empty vector that
      // the key "Myself" maps to
      siblings["Myself"].push_back("my brother");
      if (siblings.find("Myself") == siblings.end()){
         cout << "1 - Myself is not in the map\n";
      }
      // Remove the element with key "Myself"
      siblings.erase("Myself");
      if (siblings.find("Myself") == siblings.end()){
         cout << "2 - Myself is not in the map\n";
      }
      return 0;
}
```

Output:

```
2 - Myself is not in the map
```

Additional Coding Examples

Virtually any type in C++ can be used as a value in an unordered map. You can use built-in types (int, double, float, etc.), your own defined classes and structs, or even other standard library classes like vectors or even unordered maps!

Two important notes

1. If you are creating an unordered_map with the value type as your own custom class, **you need to give that class/struct a default constructor.**
2. If you are creating an unordered_map with the key type as string, you need to include the <string> header in your code. Otherwise C++ will not be able to create hashes for string objects.

`Example:`

```
#include <unordered_map>
#include <iostream>
#include <string>
using namespace std;

struct item{
        int weight;
        string description;
        item(int w, string d):weight(w), description(d){}
        item(){}
};

int main(){
   unordered_map <string, unordered_map<string, item>> inventory;
   inventory["potions"];
   inventory["potions"]["Health Potion"];
   inventory["potions"]["Health Potion"] = item(0.5, "Restores 20 HP");
   inventory["potions"]["Mana Potion"]= item(0.4, "Restores 5 mana");

   inventory["weapons"]["Long Sword"] = item(800, "A sword that is long");
   inventory["weapons"]["Masters Sword"] = item(730, "A sword with two degrees");

   inventory["armor"]["Plot Armor"] = item(200, "Takes the fun out of everything");
   inventory["armor"]["Cloth Armor"] = item(191, "Not really armor");

   if (inventory.find("potions") != inventory.end()){
        if (inventory["potions"].find("Health Potion") != inventory["potions"].end()){
        cout << "Used health potion!" << endl;
        inventory["potions"].erase("Health Potion");
        }
   }
```

```
    if (inventory["weapons"].find("Long Sword") != inventory["weapons"].end()){
        cout << "Equipped the long sword!\n";
    }

    if (inventory["armor"].find("Plot Armor") != inventory["armor"].end()){
        cout << "You cheated! Way to ruin the game.\n";
    }
    return 0;
}
```

Output:

```
Used health potion!
Equipped the long sword!
You cheated! Way to ruin the game.
```

Bonus: Tuple

A tuple is a templated container that has a fixed size (cannot be made bigger or smaller) and that can contain any combination of types. It is similar to a tuple in python but its syntax is a bit different. Here is a tuple that will contain two integers and a string:

```
tuple<int, int, string> bridge_height_width_name;
get<0>(bridge_height_width_name) = 20;// sets first element
get<1>(bridge_height_width_name) = 300;// sets second element
get<2>(bridge_height_width_name) = "Bridgton";// sets third element
```

Here is a tuple that will contain two integers, followed by a character, followed by a float, and followed by an integer:

```
tuple<int, int, char, float, int> why;
```

Unlike python, tuples are not immutable; you can change the values of elements within a tuple at any time. However, the way you access tuples is a bit different than Python as shown in the example above. Here are the most important functions you'll need with tuples:

Given the tuple:

```
tuple<K, F, G> ex;
```

```
ex = make_tuple(K k_elm, F f_elm, G g_elm);
```

This is the method used to construct a tuple. Without using this function, the programmer must set every element separately using the function `get<i>(ex)` mentioned next.

```
get<i>(ex);
```

Will return a reference to the element in `ex` at the `ith` position. This can be used to retrieve values or to change values.