

C++ STL Map

September 1, 2019

```
[1]: #include <iostream>

using namespace std;
```

0.1 Why Map in C++?

To understand, we'll go through a problem.

Problem: Given an array $N(a_1, a_2, \dots)$ positive integers. Find frequency of each number in the array.

0.1.1 Way 1: Brute Force Solution

```
[2]: int len;
cin >> len;
int* N = new int[len];
```

10

```
[3]: for(int i=0; i<len; i++) {
    cin >> N[i];
}
```

3
4
3
3
2
4
55
2
3
3

```
[4]: // Drawback: Returns frequency of all the elements even being repetitive
for(int i=0; i<len; i++) {
    int count = 0;
    for(int j=0; j<len; j++) {
```

```

        if(N[i] == N[j])
            count++;
    }
    cout << "Count of " << N[i] << " : " << count << endl;
}

```

```

Count of 3 : 5
Count of 4 : 2
Count of 3 : 5
Count of 3 : 5
Count of 2 : 2
Count of 4 : 2
Count of 55 : 1
Count of 2 : 2
Count of 3 : 5
Count of 3 : 5

```

0.1.2 Way 2: Using Hashing

```

[5]: // Create Hashing Array
     int hash_array[1000001];

```

```

[6]: for(int i=0; i<len; i++) {
     hash_array[N[i]]++;
}

```

```

[7]: cout << "Size of the array: " << sizeof(hash_array)/sizeof(hash_array[0]) << "\n";

```

Size of the array: 1000001

```

[8]: for(int i=0; i<sizeof(hash_array)/sizeof(hash_array[0]); i++) {
     if(hash_array[i] != 0)
         cout << "Count of " << N[i] << " : " << hash_array[i] << endl;
}

```

```

Count of 3 : 2
Count of 3 : 5
Count of 2 : 2
Count of 873858069 : 1

```

Disadvantages:

1. Memory wastage: The `hash_array` array is allocated with 1000001 locations. While each location will take 4 bytes, and thus $4 * 1000001 = 4000004$ bytes will be allocated.
2. What if the array length is larger than 1000001? In that case, this method won't work.

0.1.3 Way 3: Using Map

```
[9]: #include <map>

[10]: map <int, int> map_arr;

[11]: // Originally, all elements in map_arr are initialized to 0
for(int i=0; i<len; i++) {
    map_arr[N[i]]++;
}

[12]: for(map <int, int>::iterator iter = map_arr.begin(); iter != map_arr.end();
    ↪iter++) {
    cout << "Count of " << iter->first << " : " << iter->second << endl;
}
```

```
Count of 2 : 2
Count of 3 : 5
Count of 4 : 2
Count of 55 : 1
```

1. In case we have array length of array $> 10^7$, then we can use `long int` instead of `int`. It's better because it saves the memory space.
2. To access any element from the map, time complexity is $O(\log N)$ where N is the size of map.
3. map stores the elements in sorted order. (Logic: we do `map_arr[number]++` so, lesser the number is, earlier the location is.

```
[13]: // Size of map
cout << map_arr.size();
```

```
4
```

0.2 Find in Map

```
if(given key present in the map) {
    return iterator pointing to that key value pair
} else {
    return pointer to the end of the map
}
```

```
[14]: auto iter = map_arr.find(3);
// if key not found, then find function will return map_arr.end()
if(iter != map_arr.end()) {
    cout << "Key " << iter->first << " found at: " << iter->second << endl;
}
```

```
Key 3 found at: 5
```

Time complexity of `find()` is $O(\log N)$. N is size of the map.

0.3 Erase from the Map

```
[15]: cout << "Map before erasing\n";  
for(iter = map_arr.begin(); iter != map_arr.end(); iter++) {  
    cout << iter->first << ", " << iter->second << endl;  
}
```

Map before erasing

2, 2
3, 5
4, 2
55, 1

```
[16]: // Let's erase 55 from the array  
// It doesn't fit well with the other small numbers ;)  
iter = map_arr.find(55);  
if(iter != map_arr.end()) {  
    cout << "Key found, and erasing." << endl;  
    map_arr.erase(iter);  
} else {  
    cout << "Key not found. Can't erase" << endl;  
}
```

Key found, and erasing.

```
[17]: cout << "Map after erasing\n";  
for(iter = map_arr.begin(); iter != map_arr.end(); iter++) {  
    cout << iter->first << ", " << iter->second << endl;  
}
```

Map after erasing

2, 2
3, 5
4, 2

Time complexity to erase:

1. if iterator is passed to `map_arr.erase()` then it takes $O(1)$ time.
2. if number is passed to `map_arr.erase()` then it takes $O(\log N)$ time.

0.4 Clear in Map

```
[18]: // Clear all the key value pairs and make it's size to 0  
cout << "Size before clearing: " << map_arr.size() << endl;  
map_arr.clear();  
cout << "Size after clearing: " << map_arr.size() << endl;
```

Size before clearing: 3

Size after clearing: 0

0.5 Examples

map in C++ is a little similar to dict in Python. Let's experiment with writing a snippet similar to following in C++:

```
dict_ = {1:'one', 2:'two', 3:'three', 4:'four', 5:'five', 6:'six', 7:'seven', 8:'eight', 9:'nine'}
for i in dict_.keys():
    print(i, dict_[i])
num = int(input("Enter number: "))
print("You entered: {}".format(dict_[num]))
```

```
[19]: map<int, string> dict_;
```

```
[20]: dict_[1] = "one";
dict_[2] = "two";
dict_[3] = "three";
dict_[4] = "four";
dict_[5] = "five";
dict_[6] = "six";
dict_[7] = "seven";
dict_[8] = "eight";
dict_[9] = "nine";
dict_[10] = "ten";
dict_[11] = "eleven";
dict_[12] = "twelve";
dict_[13] = "thirteen";
dict_[14] = "fourteen";
dict_[15] = "fifteen";
dict_[16] = "sixteen";
dict_[17] = "seventeen";
dict_[18] = "eighteen";
dict_[19] = "nineteen";
dict_[20] = "twenty";
dict_[30] = "thirty";
dict_[40] = "forty";
dict_[50] = "fifty";
dict_[60] = "sixty";
dict_[70] = "seventy";
dict_[80] = "eighty";
dict_[90] = "ninty";
```

```
[21]: map<int, string> suffix;
suffix[100] = "hundred";
suffix[1000] = "thousand";
```

```
[22]: int num;
cout << "Enter number: ";
cin >> num;
```

Enter number: 4

```
[23]: cout << "You entered: " << dict_[num] << endl;
```

You entered: four

```
[24]: int _pow(int x, int y) {  
    if(y == 0) {  
        return 1;  
    }  
    while(y > 1) {  
        x = x * x;  
        y -= 1;  
    }  
    return x;  
}
```

```
[25]: void print(int num) {  
    // Find unit, tenth, hundredth and so on digits  
    int count = 0;  
    int suf = 1;  
    vector<string> output;  
    while(true) {  
        int rem = num % 10;  
        num = int(num/10);  
        if(count > 0)  
            suf = suf * 10;  
  
        auto it = suffix.find(suf);  
        if(it != suffix.end() && rem != 0) {  
            output.push_back(suffix[suf]);  
            output.push_back(dict_[rem]);  
        } else {  
            output.push_back(dict_[rem * suf]);  
        }  
        count++;  
        if(num == 0)  
            break;  
    }  
    reverse(begin(output), end(output));  
  
    for(auto it = output.begin(); it != output.end(); it++) {  
        cout << *it << " ";  
    }  
}
```

```
[26]: print(1334);
```

one thousand three hundred thirty four

```
[27]: print(309);
```

three hundred nine

```
[28]: print(300);
```

three hundred

```
[29]: print(9000);
```

nine thousand

```
[30]: print(9999);
```

nine thousand nine hundred ninty nine