Maps

July 16, 2021

1 Maps

https://en.cppreference.com/w/cpp/container/map

1.1 Topics

- Map definition
- Declare map
- Access elements
- Map Modifiers
- Aggregate operations
- Iterators
- Lookup operations
- Applications

1.2 Map

- the containers such as **array** and **vector** are linear and the keys are fixed integer indices
- at times problems may require a dictionary like data-structure where you need to select your own key that is associated with some value
- **map** is such a data structure where you store key-value pairs of your choosen types
- **map** is also called associative container that contains key-value pairs with unique keys
  - map is automatically sorted based on keys
  - all keys are of the same type and all values are of the same type
  - key and value can be of the same type or can be different types
- the following figure depicts a map data structure that maps English numbers (string) to Spanish numbers (string)

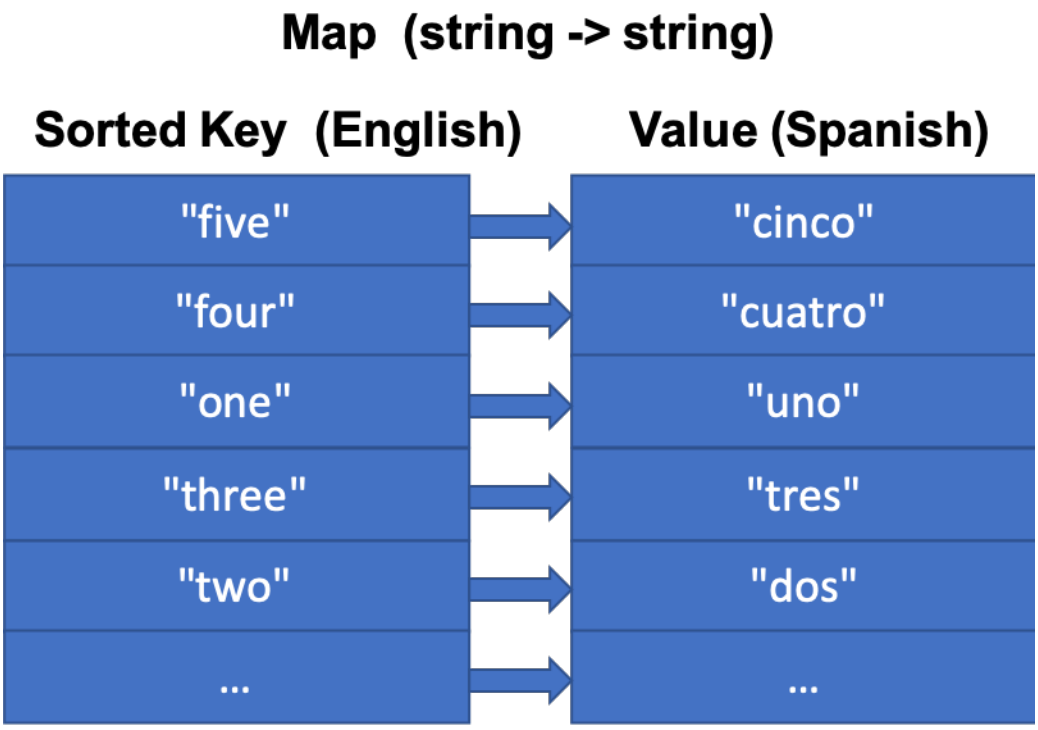


Fig. C++ Map Container

- keys are mapped to values (one-way)
  - values are not mapped to the keys
- under the hood **map** is implemented as red-black trees
- the complexity (efficiency) of common operations such as search, removal, and insertion operations is $O(log_2 n)$
  - simply put, if there are about 4 billion key-value pairs in a map, these common tasks can be completed in about 32 iterations (operations)
  - oder of operations is something discussed in more details in Data structures and Algorithm courses

1.3 Map objects

- must include header file **<map>** and use namespace std
- a template class designed to store key of any data type that can be compared
  - value can be of any type
- map objects must be declared before they can be used
- syntax

```
map<type, type> object;
```

[1]:
```
// include header files
#include <iostream>
#include <string>
```

```
#include <map>

using namespace std;
```

[4]:
```
// declare map containers without initialization
map<string, string> eng2Span;
map<char, int> charToNum;
map<int, char> numToChar;
```

[ ]:
```
// declare and initialize
map<string, int> words = {
        {"i", 10},
        {"love", 20},
        {"C++", 30},
        {"!", 40},
    };
map<string, float> prices = {{"apple", 1.99}, {"orange", 1.99}, {"banana", 2.
 ↪99}, {"lobster", 20.85}};
map<string, float> dupPrices = prices;
```

[16]:
```
// contents of words
words
```

[16]:
```
{ "!" => 40, "C++" => 30, "i" => 10, "love" => 20 }
```

[17]:
```
// prices contents:
prices
```

[17]:
```
{ "apple" => 1.99000f, "ball" => 0.00000f, "banana" => 2.99000f, "lobster" =>
20.8500f, "orange" => 1.99000f }
```

1.3.1 values can be user-defined type

[8]:
```
// define Rectangle type
// Note - the word Type is redundant! Rectangle by itself would mean a type
struct RectangleType {
    float length, width;
};
```

[9]:
```
// create a map that maps ints to RectangleType
map<int, RectangleType> myRects;
```

[10]:
```
// declare and initialize
map<char, RectangleType> rectMap = {{'A', {20, 10}}, {'x', {3.5, 2.1}}};
```

1.4 Accessing existing elements

- elements are accessed using keys and NOT the values
  - must know the key to get the corresponding values
  - can't get key from its value
- **at(key)** : access specified element with bounds checking
- **operator[key]** : access or insert specified element based on key
- similar to vector, but use actual key not index

[11]:
```
// accessing elements using [] bracket operator
cout << "love = " << words["love"] << endl;
cout << "apple = " << prices["apple"] << endl;
cout << "ball = " << prices["ball"] << endl; // "ball doesn't exist; returns 0"
```

```
love = 20
apple = 1.99
ball = 0
```

[11]: @0x105af2ec0

[20]:
```
// key must exist; value is unpredictable if key doesn't exist
cout << "cost of kite = " << prices["kite"];
```

```
cost of kite = 0
```

[12]:
```
// accessing elements using at() member function
cout << "love = " << words.at("love") << endl;
cout << "apple = " << prices.at("apple") << endl;
cout << "ball = " << prices.at("ball") << endl; // "ball doesn't exist; returns␣
↪0"
```

```
love = 20
apple = 1.99
ball = 0
```

[12]: @0x105af2ec0

[21]:
```
// key must exist; value is unpredictable if key doesn't exist
cout << "cost of kite = " << prices.at("kite");
```

```
cost of kite = 0
```

[14]:
```
// declared above, but should be empty map
eng2Span
```

[14]: {}

[24]:
```
// accessing user-defined type as value
rectMap['x'].length
```

[24]:
```
3.50000f
```

[25]:
```
cout << "area of rectangle x = " << rectMap['x'].length * rectMap['x'].width;
```

```
area of rectangle x = 7.35
```

1.4.1 inserting key->value pairs

- new key value pairs can be inserted to a map container
- if the key exists, existing value will be replaced with the new value
- if the key doesn't exist, new key-value pair will be inserted in the right location making sure keys are always sorted

[26]:
```
// add new elements
eng2Span["one"] = "uno";
eng2Span["two"] = "dos";
eng2Span["three"] = "tres";
eng2Span["four"] = "cuatro";
eng2Span["five"] = "sinco";
```

[27]:
```
eng2Span // sorted based on key
```

[27]:
```
{ "five" => "sinco", "four" => "cuatro", "one" => "uno", "three" => "tres",
"two" => "dos" }
```

[28]:
```
// sinco is misspelled; let's correct its spelling
eng2Span["five"] = "cinco";
```

[29]:
```
cout << " five in English is " << eng2Span["five"] << " in Spanish.";
```

```
 five in English is cinco in Spanish.
```

1.5 Capacity

- similar to vecotr, map provides member functions to find the capacity of map containers
- **empty()** : checks whethere the container is empty
- **size()** : returns the number of elements
  - recall, each element of map is key->value pair
- **max_size()** : returns the maximum possible number of elements

[31]:
```
cout << boolalpha; // convert boolean to text true/false
cout << "is eng2Span empty? " << eng2Span.empty() << endl;
cout << "is prices map empty? " << prices.empty() << endl;
cout << "total number of key->value pairs in prices: " << prices.size() << endl;
cout << "max_size of prices: " << prices.max_size() << endl;
```

```
is eng2Span empty? false
is prices map empty? false
```

```
total number of key->value pairs in prices: 6
max_size of prices: 288230376151711743
```

1.6 Modifying maps

- map objects also provide some member functions to modify the contents of the containers
- **clear()** : clears the contents
- **[key]** : modifies value at the specified key

[34]:
```
map<string, int> adultsage = {{"John",21}, {"Maya",74}, {"Jenny", 46},␣
↪{"Jordan", 48}, {"Mike", 46}};
```

[36]:
```
cout << adultsage << endl;
adultsage.clear(); // delete all the elements
```

```
{Jenny:46, John:21, Jordan:48, Maya:74, Mike:46}
```

[37]:
```
// should be empty
adultsage
```

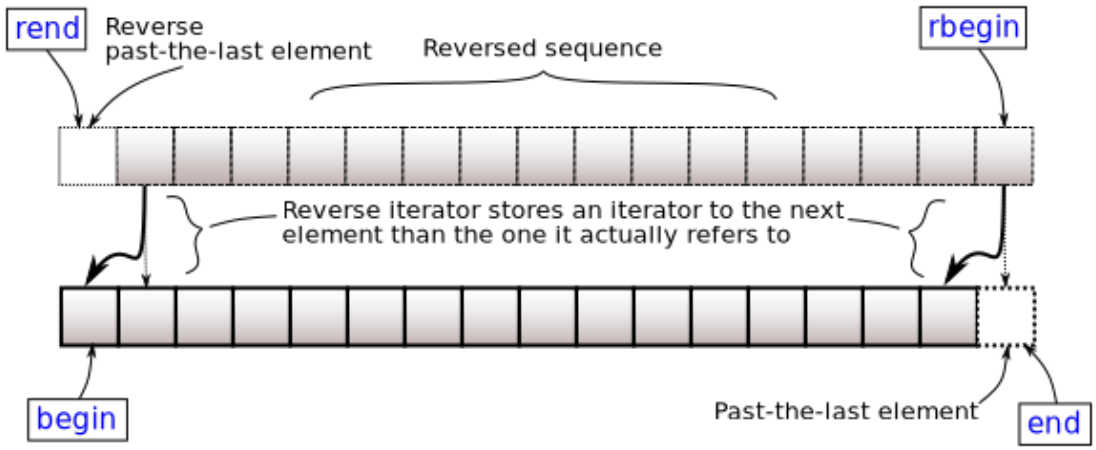
[37]: {}

1.7 Aggregate comparisons

- comparison operators ==, !=, <, >, <=, and >= are overloaded and works between two maps
- elements are compared lexicographically

1.8 Traversing maps

- map containers can be traveresed from the first element to the last (similar to array, string and vector)
- map provides iterators similar to iterators in string or vector
  - let's you iterate over all the elements
- iterator of map is a special pointer that has two elements **first** and **second**
  - first is the key and second is the value
- **begin** - returns an iterator to the beginning (first element)
- **end** - returns an iterator to the end (past the last element)
- **rbegin** - returns a reverse iterator to the beginning (past the last element)
- **rend** - returns a reverse iterator to the end (past the first element)



[39]:
```
map<int, string> amap = {{10, "val1"}, {15, "val2"}, {20, "val3"}, {30, "val4"},␣
↪{35, "val5"}};
```

[40]:
```
for(auto iterator = amap.begin(); iterator != amap.end(); iterator++)
    cout << (*iterator).first << " => " << iterator->second << endl;
```

```
10 => val1
15 => val2
20 => val3
30 => val4
35 => val5
```

[41]:
```
// iterate using range-based loop
for (auto e : amap)
    cout << e.first << " -> " << e.second << endl;
```

```
10 -> val1
15 -> val2
20 -> val3
30 -> val4
35 -> val5
```

[27]:
```
// type alias
using mii = map<int, int>;
```

[28]:
```
mii map1 = {{1,10}, {2,20}, {3,30}, {4,40}, {5,50}};
cout << map1 << endl;
```

```
{1:10, 2:20, 3:30, 4:40, 5:50}
```

1.9 Lookup lements

- map containers provide member functions to search for element with given key in a map container

  - is typically used if you're not sure if a given key exists or not
- **count(key)** : returns the number of elements matching specific key (always 1 if exists, 0 otherwise)
- **find(key)** : finds elements with specific key, returns iterator

[2]:
```
// map char to its ASCII value
map<char, int> mapci = {{'a', 'a'}, {'b', 'b'}, {'c', 'c'}, {'A', 'A'}, {'B',␣
 ↪'B'}, {'1', '1'}};
```

[3]:
```
mapci
```

[3]:
```
{ '1' => 49, 'A' => 65, 'B' => 66, 'a' => 97, 'b' => 98, 'c' => 99 }
```

[4]:
```
cout << mapci.count('a') << endl;
```

```
1
```

[5]:
```
cout << mapci.count('z') << endl;
```

```
0
```

[6]:
```
if (mapci.count('a') == 1)
    cout << "Found!";
else
    cout << "Not found!";
```

```
Found!
```

[7]:
```
// find method; returns iterator
auto it = mapci.find('c');
if (it != mapci.end())
    cout << "found " << it->first << " => " << it->second << endl;
else
    cout << "NOT found!";
```

```
found c => 99
```

[8]:
```
// erase using iterator
it = mapci.erase(it);
```

[9]:
```
// it points to key 'c', so it must be erased
mapci
```

[9]:
```
{ '1' => 49, 'A' => 65, 'B' => 66, 'a' => 97, 'b' => 98 }
```

1.10 Passing map objects to functions

- map objects can be passed by value and by reference

– by reference is preferred to prevent copying all the elements unless it's necessary

[18]:
```
// linear search function that returns true if key is found in someMap
bool searchMap(const map<char, int> & someMap, char key) {
    auto it = someMap.find(key);
    return it != someMap.end();
}
```

[20]:
```
cout << boolalpha << "A exists as key? " << searchMap(mapci, 'A');
```

```
A exists as key? true
```

[21]:
```
cout << boolalpha << "$ exists as key? " << searchMap(mapci, '$');
```

```
$ exists as key? false
```

1.11 Returning map objects from functions

- map objects can be returned from functions
- however, pass by reference is preferred to get the data out of function instead of explictly returning a map

[22]:
```
// function updates the map using pass-by-reference
void createMap(map<int, string> & m) {
    m[1] = "one";
    m[2] = "two";
    m[3] = "three";
    m[4] = "four";
    // ...etc.
}
```

[23]:
```
// create an empty map
map<int, string> numbers;
```

[24]:
```
// let's create the map using function
createMap(numbers);
```

[26]:
```
// check the contents if the function inserted elements into map
numbers
```

[26]:
```
{ 1 => "one", 2 => "two", 3 => "three", 4 => "four" }
```

1.12 Applications

- map can be applied to many problems #### keep track of menu items and the customers who ordered those items
- https://open.kattis.com/problems/baconeggsandspam

[10]:
```
#include <map>
#include <vector>
#include <algorithm>
#include <string>

using namespace std;
```

[11]:
```
map<string, vector<string> > items;
```

[12]:
```
// bacon is ordered by John
items["bacon"].push_back("John");
```

[13]:
```
// bacon is ordered by Jim
items["bacon"].push_back("Jim");
```

[14]:
```
// see all the custumers who ordered bacon
items["bacon"]
```

[14]:
```
{ "John", "Jim" }
```

[15]:
```
for (auto menu : items) {
    cout << menu.first; // print key  (menu item)
    // sort value (vector of customers)
    sort(menu.second.begin(), menu.second.end());
    // print each value in the vector which is the second element of p
    for (auto customer: menu.second)
        cout << " " << customer;
}
```

```
bacon Jim John
```

[16]:
```
// sort the vector with the key 'bacon' in descending (non-increasing) order
sort(items["bacon"].begin(), items["bacon"].end(),  greater<string>());
```

[17]:
```
// see the sorted vector
items["bacon"]
```

[17]:
```
{ "John", "Jim" }
```

1.13 Exercises

1. Write a function that finds and returns the letter frequency in a given word.
   - write 3 automated test cases

[1]:
```
// Sample solution for Exercise 1
#include <cctype>
#include <string>
```

```
#include <map>
#include <vector>
#include <iostream>
#include <cassert>

using namespace std;
```

[2]:
```
// linear search function that searches given key in given map
// returns true if key is found; false otherwise
bool searchMap(const map<char, int> m, char key) {
    auto find = m.find(key);
    return (find != m.end());
}
```

[3]:
```
void test_searchMap() {
    assert(searchMap({{'a', 1}, {'b', 5}, {'!', 1}}, 'a') == true);
    assert(searchMap({{'q', 2}, {'Z', 1}}, 'm') == false);
    cerr << "all test cases passed for searchMap\n";
}
```

[4]:
```
test_searchMap();
```

```
all test cases passed for searchMap
```

[5]:
```
// function finds and returns frequency of each character
void letterFrequency(string text, map<char, int> & freq) {
    for (char ch: text) {
        ch = char(tolower(ch)); // make case insensitive
        // find each c in freq map
        if (searchMap(freq, ch)) // found
            freq[ch] += 1; // update frequency by 1
        else
            freq[ch] = 1; // add new element
    }
}
```

[6]:
```
void test_letterFrequency() {
    map<char, int> ans;
    letterFrequency("Hi!", ans);
    map<char, int> expected = {{'!', 1}, {'h', 1}, {'i', 1}};
    assert(ans == expected);
    ans.clear();
    letterFrequency("Yo yO", ans);
    map<char, int> expected1 = {{' ', 1}, {'o', 2}, {'y', 2}};
    assert(ans == expected1);
    ans.clear();
    letterFrequency("Mississippi", ans);
```

```
    map<char, int> expected2 = {{'i', 4}, {'m', 1}, {'p', 2}, {'s', 4}};
    assert(ans == expected2);
    cerr << "all test cases passed for letterFrequency()\n";
}
```

[7]:
```
test_letterFrequency();
```

```
all test cases passed for letterFrequency()
```

[8]:
```
string input;
```

[10]:
```
cout << "Enter some text:";
getline(cin, input);
```

```
Enter some text:This is some text!
```

[11]:
```
input
```

[11]:
```
"This is some text!"
```

[12]:
```
map<char, int> answer;
```

[13]:
```
letterFrequency(input, answer);
```

[22]:
```
answer
```

[22]:
```
{ ' ' => 3, '!' => 1, 'e' => 2, 'h' => 2, 'i' => 3, 'm' => 1, 'o' => 1, 's' =>
4, 't' => 4, 'x' => 1 }
```

1.13.1 complete sample solution for Exercise 1 is at demos/maps/letter_frequency/

2. Write a function that finds and returns the frequency of vowels in a given word.
   - write 3 automated test cases
3. Write a program that reads some text data and prints a frequency table of the letters in alphabetical order. Case and punctionals should be ignored. A sample output of the program when the user enters the data "ThiS is String with Upper and lower case Letters", would look this:
   - design your program in such a way that you write automated test cases
   - prompt user to enter some text
   - use as many functions as possible
   - write at least 3 test cases for each function that computes some results

1.14 Kattis problems

- several problems in Kattis can be solved easier if map is used
- here are some of the programs that use map data structure

1. I've Been Everywhere, Man - https://open.kattis.com/problems/everywhere

2. Seven Wonders - https://open.kattis.com/problems/sevenwonders
3. ACM Contest Scoring - https://open.kattis.com/problems/acm
4. Stacking Cups - https://open.kattis.com/problems/cups
5. A New Alphabet - https://open.kattis.com/problems/anewalphabet
6. Words for Numbers - https://open.kattis.com/problems/wordsfornumbers
7. Babelfish - https://open.kattis.com/problems/babelfish
8. Popular Vote - https://open.kattis.com/problems/vote
9. Adding Words - https://open.kattis.com/problems/addingwords
10. Grandpa Bernie - https://open.kattis.com/problems/grandpabernie
11. Judging Troubles - https://open.kattis.com/problems/judging
12. Not Amused - https://open.kattis.com/problems/notamused
13. Engineering English - https://open.kattis.com/problems/engineeringenglish
14. Hardwood Species - https://open.kattis.com/problems/hardwoodspecies
15. Conformity - https://open.kattis.com/problems/conformity
16. Galactic Collegiate Programming Contest - https://open.kattis.com/problems/gcpc
17. Simplicity - https://open.kattis.com/problems/simplicity
18. Accounting - https://open.kattis.com/problems/bokforing

1.15 Summary

- learned a very useful associative data structure called map
- each element of map is a key-value pair
- elements of map are sorted based on key
- went through various member functions of map objects and their applications
- learned that maps can be passed to functions and can be returned from them as well
- exercises and sample solutions

```
[ ]:
```