

### The top 10 user and movie ratings :

```
"D:\Work\School\3rd Semester\C++\term-project\hw\cmake-build-debug\hw.exe"
-----
userID | #Ratings
-----
190:    1633
1475:    1436
591:     1388
19:      1330
52:      1087
24:       964
703:      935
1758:      906
1814:      882
942:      836
-----
movieID | #Ratings
-----
118:     16000
198:     13145
11:      11993
3:       11848
135:     11249
49:      10533
27:      10009
8:        9724
72:      8779
30:      8174
```


In order to achieve this conclusion briefly I wrote **csv\_reader** class has a method called **read\_csv** that reads a CSV file with user, item, and rating data, and creates two maps: **userMap** and **itemMap**. The **userMap** maps a user to the count of their ratings, and the **itemMap** maps an item to the count of its ratings and then our function reads each line of the file and uses **std::getline** to extract the user, item, and rating from the line, separated by commas. It then increments the count of ratings for the user and item in their respective maps. If the user or item does not yet exist in the map, it is added to the map with a count of 1, eventually the

function sorts the **userMap** and **itemMap** by their values in descending order and prints out the top 10 users and items by rating count.


The code is getting executed around 2-3 seconds in my computer

As I have explained it in the video, the data structures and algorithms that I used for the code are as follows:

```
std::map<std::string, int> userMap;  
std::map<std::string, int> itemMap;
```

 Copy code

**std::map:** The **std::map** class is a template class that implements a sorted associative container that contains key-value pairs, where the keys are unique and are used to access the values. The **std::map** class is implemented as a red-black tree, which is a self-balancing binary search tree. This means that the **std::map** class maintains the tree structure such that the height of the tree is always  $O(\log n)$ , where  $n$  is the number of elements in the tree. This allows the **std::map** class to provide logarithmic time complexity for inserting, deleting, and searching elements, as well as for iterating over the elements in order. In this code, The **userMap** and **itemMap** variables in this code are both maps that map a string (representing a user or item name) to an integer (representing the count of ratings). These maps are initialized as empty maps, and are populated by reading the CSV file line by line.


 Copy code

```
while (std::getline(file, line))
{
    // Read the first, second, and third columns from the line
    std::string user, ratingStr, item;
    std::getline(lineStream, user, ',');
    std::getline(lineStream, item, ',');
    std::getline(lineStream, ratingStr);

    // Increment the count of ratings for the user
    userMap[user]++;


    // Increment the count of ratings for the item
    itemMap[item]++;
}
```

For each line, the user and item name are extracted from the line using **std::getline**, and the count of ratings for the user and item are incremented in the **userMap** and **itemMap** maps, respectively. If the user or item does not yet exist in the map, it is added to the map with a count of 1. **std::vector**: The **std::vector** class is a template class that implements a dynamic array that can grow and shrink as needed. The **std::vector** class provides constant time complexity for accessing elements using the array subscript operator (e.g. **vec[i]**), and amortized constant time complexity for inserting and deleting elements at the end of the vector.

 Copy code


```
std::vector<std::pair<std::string, int>> sortedUserMap;
std::vector<std::pair<std::string, int>> sortedItemMap;
```

The **sortedUserMap** and **sortedItemMap** variables in this code are both vectors that are used to store the sorted **userMap** and **itemMap** maps, respectively. These vectors are initialized as empty vectors, and are populated by iterating over the elements in the **userMap** and **itemMap** maps and inserting them into the vectors using the **emplace\_back** member function.

 Copy code

```
for (const auto& pair : userMap) {  
    sortedUserMap.emplace_back(pair);  
}  
for (const auto& pair : itemMap) {  
    sortedItemMap.emplace_back(pair);  
}
```

The **emplace\_back** function constructs a new element in-place at the end of the vector, which can be more efficient than copying an element from another container. **std::sort**: The **std::sort** function is a function template that sorts the elements in a range in ascending order using the **<** operator by default. The **std::sort** function uses an implementation of the **quicksort algorithm**, which has an average time complexity of **O(n log n)** for sorting a range of elements. The quicksort algorithm works by selecting a pivot element from the range, partitioning the range into two subranges based on whether the elements are less than or greater than the pivot, and recursively sorting the subranges.

 Copy code

```
std::sort(sortedUserMap.begin(), sortedUserMap.end(),  
    [](const auto& a, const auto& b) { return a.second > b.second; });  
std::sort(sortedItemMap.begin(), sortedItemMap.end(),  
    [](const auto& a, const auto& b) { return a.second > b.second; });
```

In this code, the **std::sort** function is used to sort the **sortedUserMap** and **sortedItemMap** vectors by their values in descending order. To do this, the **std::sort** function is passed a comparison function as its third argument, which compares the second elements (i.e. the count of ratings) of the pairs in the vectors. The comparison function returns true if the first element is greater than the second element, and false otherwise. This causes the **std::sort** function to sort the vectors in descending order by the count of ratings.