



Project #02: Divvy Station and Ridership Analysis

Complete By: Saturday, February 4th @ 11:59pm

Assignment: C++ program to perform movie review analysis

Policy: Individual work only, late work **is** accepted (see “Policy” section on last page for more details)

Submission: online via zyLabs (“Project02 Divvy Analysis”, section 1.11)

Assignment

The assignment is to input Divvy station and ridership data, and then interact with the user to support the following four commands:

1. Find stations nearby
2. Info about a station
3. Top N stations by ridership
4. Hourly ridership totals

The data will come from 2 input files, both in CSV format (Comma-Separated Values). Your job is to write a modern C++ program to input this data, organize it into one or more data structures, and perform the requested analyses / output.

Here’s a screenshot for the input files “stations.csv” and “rides.csv” --- the user input is circled in red. The user inputs the filenames (stations file first), and then supplies one or more commands to the program. Your program must match this output exactly, since the first level of grading will use the zyLabs system.

```
C:\Windows\system32\cmd.exe
stations.csv
rides.csv
**Num stations: 581
**Num rides: 1689
Find 41.86 -87.62 0.5
Station 338: 0.1680 miles
Station 273: 0.2790 miles
Station 72: 0.3006 miles
Station 168: 0.3401 miles
Station 97: 0.3835 miles
Station 370: 0.4046 miles
Station 4: 0.4290 miles
Station 178: 0.4544 miles
info 338
Name: 'Calumet Ave & 18th St'
Position: (41.8576, -87.6194)
Capacity: 15
Total rides to/from: 8
Num rides originating:
 6-9am: 1
 noon-1pm: 0
 3-7pm: 1
top 3
1. Station 35: 63 rides @ 'Streeter Dr & Grand Ave'
2. Station 268: 60 rides @ 'Lake Shore Dr & North Blvd'
3. Station 85: 52 rides @ 'Michigan Ave & Oak St'
exit
Press any key to continue . . .
```

Input Files

The input files are text files in **CSV** format — i.e. comma-separated values. The first file defines the set of **stations**, in no particular order. An example of the format is as follows:

```
id,name,latitude,longitude,dpcapacity,online_date
456,2112 W Peterson Ave,41.991178,-87.683593,15,5/12/2015
101,63rd St Beach,41.78101637,-87.57611976,23,4/20/2015
109,900 W Harrison St,41.874675,-87.650019,19,8/6/2013
.
.
.
```

The first line contains column headers, and should be ignored. The data starts on line 2, and each data line represents one Divvy station for docking/undocking bicycles. Each station contains 6 values: a unique station ID (integer), the stations name (string), the position of the station in (longitude, latitude) format, the capacity of the station (integer, # of bikes you can dock there), and the date the station came online (string). Do not assume the station IDs are contiguous 1..N, they are not.

The second file defines a set of **rides**, in no particular order. With Divvy, a “bike ride” means a rider checks out a bike from one station, rides it around the city, and then checks it back into the same station, or a different station. The main purpose of Divvy is commuting, so most bike rides are short — less than 30 minutes. Here’s an example of the file format (you may need to open one of the data files to see this better):

```
trip_id,starttime,stoptime,bikeid,tripduration,from_station_id,from_station_name,to_station_id,to_station_name,usertype,gender,birthyear
10426648,6/30/2016 23:57,7/1/2016 0:22,4050,1466,259,California Ave & ...,123,California Ave & ...,Subscriber,Female,1986
10426638,6/30/2016 23:55,7/1/2016 0:40,4579,2713,177,Theater on the Lake,340,Clark St & Wrightwood Ave,Customer,,
.
.
.
```

The first line contains column headers, and should be ignored. The data starts on line 2, and each data line represents one Divvy ride. Each data line consists of 12 values:

- Trip id: integer
- Start time: date and time of trip start
- Stop time: data and time of trip end
- Bike id: integer
- Trip duration: integer, in seconds
- From station id: integer, where bike was checked out
- From station name: string, where bike was checked out
- To station id: integer, where bike was returned
- To station name: string, where bike was returned
- User type: a registered “subscriber” or a new “customer”
- Gender: optional string, may be missing
- Birth year: optional integer, may be missing

You will only be dealing with a subset of the ridership values, feel free to ignore the rest. In case you’re curious, the Divvy data is available for browsing on Chicago’s data [portal](#); the raw data was downloaded from

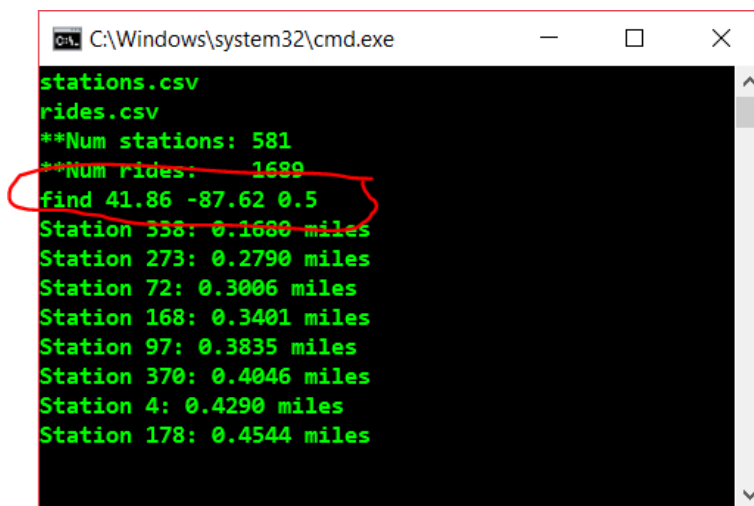
A set of input files will be provided as part of the Visual Studio solution we are releasing. As usual, the files we provide are just one possible set of input files. We will be using different, and much larger files, when grading.

User commands

The user can input any number of commands; for simplicity assume the input is valid in the sense that if the user needs to input an integer or a real number of a string, he/she will do so. The four commands are “find”, “info”, “top”, and “hourly”. The user enters “exit” when he/she wishes to stop.

1. find latitude longitude distance

The “find” command finds Divvy stations that are nearby to the position input by the user. The user inputs their latitude, longitude, and distance they are willing to ride (in miles), and the program outputs the Divvy stations whose position is \leq that distance away. Example:



```
C:\Windows\system32\cmd.exe
stations.csv
rides.csv
**Num stations: 581
**NUM rides: 1689
find 41.86 -87.62 0.5
Station 338: 0.1680 miles
Station 273: 0.2790 miles
Station 72: 0.3006 miles
Station 168: 0.3401 miles
Station 97: 0.3835 miles
Station 370: 0.4046 miles
Station 4: 0.4290 miles
Station 178: 0.4544 miles
```

Notice the stations are sorted and output by their distance from the entered position. [A function is provided in the given code to compute the distance between 2 points in (lat, long) format.]

2. info stationID

The “info” command outputs information about a Divvy station, including some basic analysis of the ridership data. The user inputs a station id, and the program outputs information about that station. Assume the user will input an integer, but do not assume the station id exists. Example:

```
C:\Windows\system32\cmd.exe

stations.csv
rides.csv
**Num stations: 581
**Num rides: 1689
info 268
Name: 'Lake Shore Dr & North Blvd'
Position: (41.9117, -87.6268)
Capacity: 39
Total rides to/from: 60
Num rides originating:
  6-9am: 1
  noon-1pm: 0
  3-7pm: 4
info 12345
**Not found...
```

In terms of “Number of rides originating”, this means the # of rides that started from this station, where the starting hour H of the ride began during those hours start $\leq H < \text{end}$. For example, for the time period “3-7pm”, that means count all rides where the start hour H is $15 \leq H < 19$. [*Hours are in military time.*]

3. top N

The “top” command outputs the top N stations in terms of total ridership to/from each station, in descending order. If 2 stations have the same # of rides, then output in order by station name. For example:

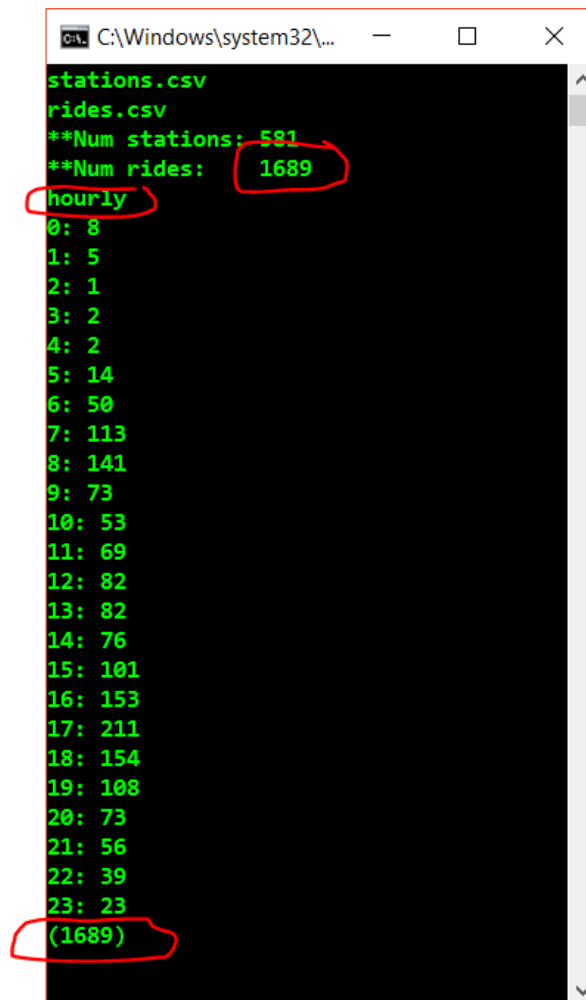
```
C:\Windows\system32\cmd.exe

stations.csv
rides.csv
**Num stations: 581
**Num rides: 1689
top 12
1. Station 35: 63 rides @ 'Streeter Dr & Grand Ave'
2. Station 268: 60 rides @ 'Lake Shore Dr & North Blvd'
3. Station 85: 52 rides @ 'Michigan Ave & Oak St'
4. Station 77: 45 rides @ 'Clinton St & Madison St'
5. Station 91: 43 rides @ 'Clinton St & Washington Blvd'
6. Station 90: 42 rides @ 'Millennium Park'
7. Station 177: 42 rides @ 'Theater on the Lake'
8. Station 192: 33 rides @ 'Canal St & Adams St'
9. Station 81: 33 rides @ 'Daley Center Plaza'
10. Station 75: 32 rides @ 'Canal St & Jackson Blvd'
11. Station 287: 32 rides @ 'Franklin St & Monroe St'
12. Station 174: 31 rides @ 'Canal St & Madison St'
```

You may assume $N > 0$, but do not assume $N \leq \# \text{ of stations}$. The input file may only contain 8 stations, and the user might ask for the top 10; it’s simple enough to handle this scenario.

4. hourly

The “hourly” command analyses and output the data in terms of when each ride started. For each hour 0-23, count the # of rides whose starting hour H began during this hour. For example, the “rides.csv” file contains data about 1,689 rides. These rides were distributed across the hours as follows:



```
C:\Windows\system32\...
stations.csv
rides.csv
**Num stations: 581
**Num rides: 1689
hourly
0: 8
1: 5
2: 1
3: 2
4: 2
5: 14
6: 50
7: 113
8: 141
9: 73
10: 53
11: 69
12: 82
13: 82
14: 76
15: 101
16: 153
17: 211
18: 154
19: 108
20: 73
21: 56
22: 39
23: 23
(1689)
```

Notice that all 1,689 rides are accounted for (your program should sum the rides from each hour and output at the end to confirm).

Requirements

As will be the case for all homework assignments in CS 341, how you solve the problem is just as important as simply getting the correct answers. You are required to solve the problem the “proper” way, which in this case means using modern C++ techniques: classes, objects, built-in generic algorithms and data structures, foreach-based iteration, etc. It’s too easy to fall back on existing habits to just “get the homework done.”

In this assignment, your solution is required to do the following:

- Use ifstream to open / close the input files.
- Use one or more classes to store the data — at a minimum you must have a **Station** class that

contains data about each Divvy station. The design of that class is up to you, but at the very least (1) all data members must be private, (2) the class contains a constructor to initialize the data members, and (3) getter/setter functions are used to access/update the data members.

- Use one of the built-in containers to store your objects: `std::vector`, `std::map`, etc.
- Sort using the built-in `std::sort` algorithm.
- Your solution must be efficient when faced with large input files. For example, given the input file “rides-2016-q1.csv”, your program input the data within 30 seconds, and answer any user command within 30 seconds. [*Note: this assumes Visual Studio is in release mode (i.e. generating optimized code).*]
- No explicit pointers of any kind — no `char *`, no `NULL`, no C-style pointers. You can use the “*” operator with respect to iterators, but that’s the only exception.
- No global variables whatsoever; use functions and parameter passing.

We are serious about these requirements, and may impose large deductions for not adhering to these requirements.

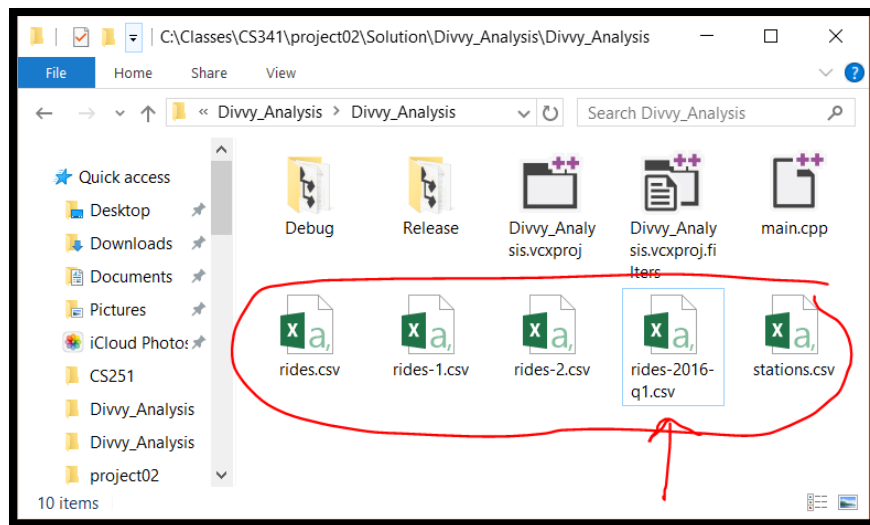
Getting Started

We are providing a Visual Studio solution, with initial C++ code and input files, to help you get started. Here are the steps to download and get started:

1. Download “[Divvy_Analysis.zip](#)” from course web site: “Projects”, “Project01-files”.
2. Open the .zip and extract the folder “Divvy_Analysis” to your desktop.
3. Close the .zip and delete — this way you don’t accidentally try to work from the .zip.
4. Leave the “Divvy_Analysis” folder on your desktop, or move elsewhere — please do not move this folder to a location that is automatically backed up by dropbox or other cloud services. Keep the folder on your local, native storage.
5. Open the folder and double-click on the “Divvy_Analysis.sln” file (VS Solution File). This opens the program in Visual Studio.
6. The “Solution Explorer” should appear in the top-right corner of the screen; if not, use the View menu to display.
7. Build via Build menu.
8. Start without debugging (Ctrl+F5). This leaves the window open when the program finishes. If you want to run and use the debugger, Start with debugging (F5). The program is expecting you to enter two filenames: “stations.csv” and “rides.csv” will work.
9. Exit the program

We are providing a set of input files to test against. Minimize / exit Visual Studio, and drill down one more level in the project folders by opening the “Divvy_Analysis” sub-folder:

<< see screenshot on next page >>



You can open these input files in Excel, or with an ordinary text editor (I used Notepad2). The highlighted file is a bigger input file when you are ready to test the efficiency of your solution.

Correctness and Efficiency

We'll be evaluating your program from 3 perspectives: design, correctness, and efficiency. Worry about the first two first, and use the Visual Studio debugger to help you find and correct errors. Copy-paste your program into zyLabs to help you with testing.

After you have the program working correctly, then check for efficiency. The goal is that your program should never take more than 30 seconds to load, or answer a user command. The first step is to switch to "Release" mode so that Visual Studio can optimize the program (this is equivalent to the "-O" option in gcc):



Optimization makes a huge difference in C++, often reducing execution time from minutes to seconds. Notice the other drop-down shown above is set to "x64" — this configures Visual Studio to generate a 64-bit program. You may need to switch from "x86" (32-bit) to "x64" (64-bit) mode in order to process really large files. Test against the "rides-2016-q1.csv" input file to see what happens against a larger input file. If designed properly, this input file should only take 1-2 seconds to process and analyze.

```
C:\Windows\system32\cmd.exe
stations.csv
rides.csv
**Num stations: 581
**Num rides: 1689
top 12
1. Station 35: 63 rides @ 'Streeter Dr & Grand Ave'
2. Station 268: 60 rides @ 'Lake Shore Dr & North Blvd'
3. Station 85: 52 rides @ 'Michigan Ave & Oak St'
4. Station 77: 45 rides @ 'Clinton St & Madison St'
5. Station 91: 43 rides @ 'Clinton St & Washington Blvd'
6. Station 90: 42 rides @ 'Millennium Park'
7. Station 177: 42 rides @ 'Theater on the Lake'
8. Station 192: 33 rides @ 'Canal St & Adams St'
9. Station 81: 33 rides @ 'Daley Center Plaza'
10. Station 75: 32 rides @ 'Canal St & Jackson Blvd'
11. Station 287: 32 rides @ 'Franklin St & Monroe St'
12. Station 174: 31 rides @ 'Canal St & Madison St'
```

Submission

The program is to be submitted via zyLabs: see “**Project02 Divvy Analysis**”, section 1.11. You can submit only a single file, so your program must be written as a single file (this is unfortunate, but a limitation we have to live with right now). When you’re ready to submit, copy-paste your program into the zyLabs editor pane, switch to “Submit” mode, and click “SUBMIT FOR GRADING”.

Keep in mind this is only a first level grading mechanism to ensure you have (a) submitted the correct file, and (b) are following the basic requirements for I/O. The grade you receive from your zyLabs submission is only a preliminary grade. After the deadline, the TAs will then manually review each program for the following:

- design and adherence to requirements
- efficiency against much larger input files
- commenting, indentation, whitespace and readability

Policy

Late work **is** accepted. You may submit as late as 24 hours after the deadline for a penalty of 25%. After 24 hours, no submissions will be accepted.

All work is to be done individually — group work is not allowed. While I encourage you to talk to your peers and learn from them (e.g. via Piazza), this interaction must be superficial with regards to all work submitted for grading. This means you **cannot** work in teams, you cannot work side-by-side, you cannot submit someone else’s work (partial or complete) as your own. The University’s policy is described here:

<http://www.uic.edu/depts/dos/docs/Student%20Disciplinary%20Policy.pdf> .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. Academic dishonesty is unacceptable, and penalties range from failure to expulsion from the university; cases are handled via the official student conduct process described at <http://www.uic.edu/depts/dos/studentconductprocess.shtml> .