Project #04: Analyzing DIVVY data (v1.1)

Assignment: C++ program to input and analyze DIVVY bike data
Evaluation: Manual review by the TAs
Policy: Individual work only
Complete By: Friday 3/19 @ 5pm (no early bonus, no late period)

Overview

Your program is going to input 2 types of data: stations in the DIVVY system, and bike trips that riders have taken on a DIVVY bike. Your program starts by prompting and inputting the name of the stations file from the keyboard, and then prompting and inputting the name of the bike trips file from the keyboard. If either file cannot be opened, your program should output an error message (including the filename) and stop.

Assuming both files are opened successfully, you'll input this data into two dynamically-allocated arrays of struct. Your program will then respond to commands input by the user until the user inputs "#", at which point your program should free memory and stop. The set of commands are discussed in a latter section. You'll need to develop a testing strategy, as testing is considered part of the project; Gradescope will not offer testing.

TAs, office hours, and Piazza

To address the long lines at office hours, the TAs will now restrict themselves to 5 minutes per student. To enable this, the TAs will focus on answering conceptual questions, suggesting high-level approaches to problem solving, and helping with output on the screen --- e.g. explaining an error message or helping interpret the possible causes of erroneous output. The TAs will not write code, nor debug code. Debugging is a skill you can learn; Panopto recordings #17 and #18 discuss strategies and tools for debugging.

The same approach will hold for Piazza: no writing of code, no debugging of code. You can post a snippet of code if you don't understand an error message, or post screen output for suggestions for help debugging. The value of Piazza is answering questions, not trying to use the platform to debug or write code. We will attempt to re-focus office hours, and Piazza, to questions that can be answered in 5 minutes. Your job will be to frame questions in a way that can be understood and answered within 5 minutes; unfocused questions will be skipped so the next student can be helped in a timely manner.

Input Files

The first filename input by the user defines the set of DIVVY **stations**, in no particular order. Here's the start of one possible stations file:

```
1243
S42329 19 41.791478 -87.599861 University & 57th
341X2 35 41.866095 -87.607267 Adler
YY5900 15 41.93533728 -87.71688929 Central Park & Elbridge
X60550 15 42.052939 -87.673447 University Library
178y4 15 41.856594 -87.627542 State & 19th
.
.
.
```

The first line contains a single integer **S** that defines the number of stations in the file; you may assume S > 0. The data starts on line 2, and each data line represents one station in the DIVVY system. Each line contains exactly 5 values, separated by exactly one space or one tab:

- Station ID — a single word (string) denoting the station, every station has a different station ID
- Capacity: — capacity (integer) that denotes the maximum number of bikes at that station
- Latitude: — latitude (double) position of the station
- Longitude: — longitude (double) position of the station
- Name: — the name of the station; note this string can be multiple words and punctuation

Do not assume the station ID or Name have a particular format, though assume the station ID is a single word. However, since the Name can be multiple words, you'll need to input that value using the **getline()** function discussed in Panopto recording #4. The other values can be input using the extraction operator >>. Example:

```
infile >> id;
infile >> capacity;
infile >> latitude;
infile >> longitude;
getline(infile, name);
```

Note the use of getline() will also capture the space / tab before the name; use the string class's **erase()** function to delete this leading whitespace?

The second filename input by the user defines a set of DIVVY **bike trips**, in no particular order. With DIVVY, a "bike trip" 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 the start of one possible bike trips file:

```
22140
T221843 B5229 S42329 YY5900 1620 23:35
T10091 B223 341X2 341X2 3001 8:22
T98237 B5229 YY5900 S42329 2311 0:08
T08123 B148 X60550 S42329 1200 7:32
```

```
T4318 B9231 341X2 X60550 1863 9:01
.
.
.
```

The first line contains a single integer **T** that defines the number of bike trips in the file; you may assume T > 0. The data starts on line 2, and each data line represents one DIVVY bike trip. Each line contains exactly 6 values, separated by exactly one space or one tab:

- Trip ID: a single word (string) denoting the trip, every trip has a different trip ID
- Bike ID: a single word (string) denoting the bike, every bike has a different bike ID
- Start station ID: the station ID denoting where the bike was checked out (start of trip)
- End station ID: the station ID denoting where the bike was returned (end of trip)
- Trip duration: how long the bike was checked out (integer), in seconds
- Start time: the hour:minute (string) denoting when the bike was checked out; the hour is a 1 or 2-digit number in the range 0..23, the minute is a two-digit number 00..59

Do not assume the trip ID or bike ID have a particular format. You may assume the input files are properly formatted, no error checking is required. If a bike trip refers to a station ID, you may assume the station ID exists. In case you're curious, actual DIVVY data is available for browsing on Chicago's data portal; the raw data is from DIVVY @ https://www.divvybikes.com/data.

User commands

After inputting the data from the two input files, your program should loop and input commands from the user. This is repeated until the user enters "#" to stop. Here's a short summary of the commands:

1. Quick statistics
2. Summary of bike durations
3. Histogram of starting times
4. Stations near me
5. List all stations
6. Find stations
7. Find trips within timespan

Each command is discussed in more detail in the following sub-sections. We are showing examples of what we want the output to look like, but your output does not have to match this format exactly --- we will accept any reasonably-close format, as long as the answers are correct. In other words, an extra space here or there doesn't matter. However, your output must contain all the required elements, in a similar format, readable with proper spelling. If one of your answers is wrong, or you omit any of the required elements, the command is considered wrong; you can't give us half the answer and expect half-credit.

<< continued on next page >>

1. Quick statistics:

```
Enter command (# to stop)> stats
  stations: 1243
  trips: 22140
  total bike capacity: 15002
```

Outputs the total # of stations, the total # of trips, and the total capacity across all the stations.

2. Summary of bike trip durations:

```
Enter command (# to stop)> durations
  trips <= 30 mins: 233
  trips 30..60 mins: 5
  trips 1-2 hrs: 3
  trips 2-5 hrs: 0
  trips > 5 hrs: 2
```

Analyzes the bike trips, and then categorizes the trips into 1 of 5 categories based on the duration. The 5 categories are defined more precisely as follows:

1. Number of trips where duration <= 30 minutes (1800 seconds)
2. Number of trips where 30 minutes < duration <= 60 minutes
3. Number of trips where 60 minutes (1 hour) < duration <= 2 hours
4. Number of trips where 2 hours < duration <= 5 hours
5. Number of trips where duration > 5 hours

3. Histogram of starting times:

```
Enter command (# to stop)> starting
  0: 7
  1: 5
  2: 3
  .
  .
  .
  22: 11
  23: 4
```

Analyzes the bike trips, and then categorizes the trips into 1 of 24 categories based on the starting hour. The

total number of trips falling into each category is displayed. For example, "23" represents the hour of 11pm, so the above screenshot denotes that a total of 4 trips started during the hour of 11pm.

A trip with the start time 0:08 represents 12:08am, i.e. 8 minutes after midnight; this trip falls into category "0" because the starting hour is 0. The trip 8:22 represents 8:22am, and falls into category "8". The last category is 23, representing trips that start during the 11pm hour. The values 0..23 as hours avoid the need for AM vs. PM, and are often referred to as *military time.*

4. Stations near me:

```
Enter command (# to stop)> nearme 41.86 -87.62 0.1
  The following stations are within 0.1 miles of (41.86, -87.62):
  none found

Enter command (# to stop)> nearme 41.86 -87.62 0.5
  The following stations are within 0.5 miles of (41.86, -87.62):
  station SS338c (Calumet & 18th): 0.168045 miles
  station W27331 (Michigan & 18th): 0.278975 miles
  station 72a (Wabash & 16th): 0.300635 miles
  .
  .
  .
  station 178y4 (State & 19th): 0.454386 miles
```

Finds all stations near a given position, expressed as **latitude** (Y coordinate) and **longitude** (X coordinate). The format of the command: `nearme lat1 long1 D`.

To be more precise, the command searches for all stations that are <= D miles away from (lat1, long1). The stations are output in order by distance, nearest to farthest; if 2 stations are the same distance away, they may be listed in either order. If no stations are found, "none found" should be output.

A function **distBetween2Points(lat1, long1, lat2, long2)** is available to compute the distance between two points (lat1, long1) and (lat2, long2). If you wish to use this function, copy the function and paste into your "main.cpp". It turns out that you may get slightly different results based on the order in which you pass the coordinates to this function. To better match our answers, the first coordinate (lat1, long1) passed to the function should be the values entered as part of the command, and the second coordinate (lat2, long2) passed to the function should be the station's position.

<< continued on next page >>

5. List all stations:

```
Enter command (# to stop)> stations
  Adler (341X2) @ (41.8661, -87.6073), 35 capacity, 31 trips
  .
  .
  .
  Montrose Harbor (2gs4922) @ (41.964, -87.6382), 31 capacity, 0 trips
  .
  .
  .
```

Lists all stations in alphabetical order by name. For each station, the output includes the name, station ID, position (latitude, longitude), capacity, and the total # of bike trips where this station was the starting station or the ending station. [*NOTE: if a trip starts and ends at the same station, this counts as 1 trip, not 2.*]

6. Find stations:

```
Enter command (# to stop)> find park
  none found

Enter command (# to stop)> find Park
  Avondale & Irving Park (483p9) @ (41.9534, -87.732), 19 capacity, 7 trips
  Central Park & Elbridge (YY5900) @ (41.9353, -87.7169), 15 capacity, 1 trips
  .
  .
  .

Enter command (# to stop)> find Calif
  California & Division (216541v) @ (41.903, -87.6975), 15 capacity, 41 trips
```

Performs a case-sensitive search of the stations whose name includes the word input by the user; you may assume it's one word, and not a string of words. The matching stations are output in alphabetical order by name. For each station, the output includes the name, station ID, position (latitude, longitude), capacity, and the total # of bike trips where this station was the starting station or the ending station. [*NOTE: if a trip starts and ends at the same station, this counts as 1 trip, not 2.*]

<< continued on next page >>

7. Find trips within timespan:

```
Enter command (# to stop)> trips 2:00 3:00
  none found

Enter command (# to stop)> trips 7:30 9:30
  3 trips found
  avg duration: 33 minutes
  stations where trip started: Adler, University Library

Enter command (# to stop)> trips 23:30 0:30
  1 trips found
  avg duration: 38 minutes
  stations where trip started: Central Park & Elbridge
```

Searches the bike trips for all trips with a start time that falls within the given timespan. For example, given the command "trips 2:00 3:00", a search is made for all trips where the start time falls within 2:00 to 3:00, inclusive --- a start time of 2:00am, 2:01am, 2:02am, …, or 3:00am. If no such trips are found, "none found" is output. Otherwise, the number of trips is output along with the average duration of those trips, in minutes. When computing the average duration, round down to the nearest minute. For example, if the average duration is 2021.33 seconds, that is an average time of 33 minutes. Note that the timespan can fall across midnight as shown in the last example ("trips 23:30 0:30").

Finally, output the names of all stations where these trips started. Output the names in alphabetical order, no duplicates. For example, if the bike trips input file consisted of the 5 trips shown earlier on pages 2-3, the command "trips 7:30 9:30" would yield the output shown above; note that "Adler" appears only once.

Requirements

How you solve the problem is just as important as developing a correct solution. In this assignment we are going to impose the following restrictions, which you must adhere to. Breaking a restriction typically leads to a score of 0.

1. No global variables. A global variable is a variable declared in "main.cpp" but outside of any functions. Use local variables and parameters instead of global variables. Global struct definitions are fine.
2. The only data structures permitted are dynamically-allocated arrays, created using the **new** operator. It is not acceptable to allocate larger arrays than required and assume those will be big enough; you must dynamically allocate all arrays using the new operator.
3. This assignment is about arrays, not files. You are allowed to open each input file exactly once. Exception: you may open an input file twice if the first time is simply to open and close --- e.g. to check that the file can be re-opened successfully later when you plan to input the data.
4. No use of the built-in algorithms for searching, sorting, min, max, etc. Write your own functions. The only built-in functions you can safely use are those provided by the **string** class, and <cmath>.

<< continued on next page >>

5. You must write at least one sorting function, and one search function.
6. All functions must be less than 100 lines of code (not counting comments); this includes main().
7. Your program must represent a general solution, the ability to input & process any properly-formatted input files.

Getting Started

A Codio environment is provided if you want to use Codio, login and open "cs141-project04". One input file is provided to help you get started with testing: stations.txt. We strongly recommend building the program one step / command at a time. Be sure to run valgrind locally to make sure there are no memory errors in your program, and no memory leaks. You'll need two data structures, one for storing data about the stations, and one for storing data about the trips. You'll need two struct definitions as well.

Suggestion: some of the commands require the output of data that is not explicitly available, but that has to be computed ("aggregated"). You can do this ahead of time, e.g. as you input the data from the files, and store this computed / aggregated data in your struct.

Grading

Submission: submit your program as a single "main.cpp" file via Gradescope "project 04: main.cpp"

Unlike previous projects, Gradescope will be used as a collection mechanism only; no test cases will be run, and no score will be reported. You will simply be acknowledged as to whether your "main.cpp" file was successfully submitted. Testing your program is part of the project.

After the due date, the TAs will manually run, test, and review your program. The final score will be posted on Gradescope once this is completed for the entire class. Pay close attention to the project restrictions mentioned earlier, they will be strictly enforced. In terms of grading:

1. We expect good commenting, readability, whitespace. If you don't know what this means, see this example of what we consider good commenting, readability, and whitespace. This counts for 10/100.
2. -20 deduction for each function that contains more than 100 lines of code (not counting comments); this includes main().
3. -20 deduction if you don't have at least one sort function.
4. -20 deduction if you don't have at least one search function.
5. -10 deduction if you do not properly free memory in all cases.
6. If a command triggers a valgrind error, that command will be marked as incorrect (no credit).
7. If the program doesn't compile, no credit (0/100).
8. If the program breaks a project restriction, no credit (0/100).
9. If the program runs but cannot input from properly-formatted files, very little credit (max 10/100)

Policy

In this assignment, all work submitted for grading *must* be done individually. While we encourage you to talk to your peers and learn from them, 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 available here:

https://dos.uic.edu/conductforstudents.shtml

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, screen sharing, 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 a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at https://dos.uic.edu/conductforstudents.shtml .