CSCI-1200 Data Structures — Spring 2023
Homework 2 — Hockey Classes

In this assignment you will parse and organize the game results from NCAA Division I Eastern College Athletic Conference men's ice hockey. Many of you are already familiar with the basics of this sport, as it is one of Rensselaer's most popular athletics events. We also summarize everything you need to know to complete this assignment. *Please read the entire handout before starting to code the assignment.*

Ice Hockey

Here's a crash course on the rules of ice hockey: Each game between two teams consists of three periods of twenty minutes each. Six players from each team are allowed on the ice at any one time. The players work together to score a goal by hitting the *puck* into the opponents' net. Usually one of the six players on each team is designated as the *goalie* to guard the net. When a goal is scored, one member of the team is credited with the score, and up to two other players are credited with assisting in the score. There are many rules and restrictions on the details of play. If these rules are violated the offending player/team is given a penalty of a two, five, or ten minute duration. During the penalty time period, the offending player must sit out (in the *penalty box*) and that team plays short-handed (with 5 players instead of 6). The other team has an advantage during this time – the so-called *power play*.

If the score is tied at the end of the 3 regular periods, the teams will play an additional *overtime* period. Play ends with the first goal scored in overtime (*sudden death*), and that team wins. If neither team scores during overtime, we will consider the result of this game a tie. Note: Normally the game proceeds to a *shootout* to resolve the tie. For more information see: http://en.wikipedia.org/wiki/Ice_hockey and http://en.wikipedia.org/wiki/ECAC_Hockey.

You will parse an input file which contains the details for one or more hockey games. The data was originally obtained from: http://collegehockeystats.net/1213/schedules/ecachm. The data has been pre-processed to make your work a bit simpler. Here is a partial input file for your program:

```
Saturday, March 2, 2013
Cornell at Yale
PERIOD 1
6:22 Yale goal Kenny_Agostino ( Andrew_Miller Jesse_Root )
7:42 Cornell goal Joel_Lowry ( Greg_Miller  )
PERIOD 2
PERIOD 3
2:01 Cornell penalty Erik_Axell  2:00 Tripping
2:07 Yale goal Stu_Wilson ( Kenny_Agostino Andrew_Miller )
8:45 Cornell penalty Rodger_Craig  2:00 Interference
20:00 Cornell penalty Joel_Lowry  5:00 Contact_to_the_Head
20:00 Yale penalty Gus_Young  5:00 Contact_to_the_Head
FINAL
Cornell 1
Yale 2
```

Each game begins with the date and the names of the two teams – the *away* team is listed first, followed by the *home* team. The details of the three periods (and overtime if necessary) begin with a label for the period followed by information about each goal scored and penalty assessed in that period.

For a *goal*, we are given the *time*, *team*, the keyword "goal", the name of the *player* who scored the goal, and then in parentheses the names of zero, one, or two *teammates who assisted* in scoring the goal. The format of time is mm:ss (minutes and seconds) since the start of that period (regular periods are 20 minutes long).

For a *penalty*, we are given the *time*, *team*, the keyword "penalty", the name of the *player* who violated the rule, the *length of time* the team will have to play with one fewer players on the ice, and a description of the *violation*. At the end of the details the final total number of goals for each team is listed. In this game, Yale scored more goals and is the winner.

Note that for ease of parsing we have replaced the spaces between players' first and last names with underscores ("_"). Similarly, all multi-word team names and multi-word penalty violation descriptions use underscores or dashes instead of spaces. We strongly recommend that you do all of your parsing for this homework using the STL stream operator `>>` and do not use `getline`. Your code should not need to rely on the exact placement or quantity of spaces or tabs or newlines in the input.

Hint on reading the file: We suggest you structure your code to read in one string at a time. Depending on the string, you'll know what to read next. If you see the word "`PERIOD`", you know to read in an integer for the next period number. If you see a time (formatted with a ':' in the middle), you'll know it is either a goal or penalty. When you are reading in the names of the players who assisted in the goal, you'll be looking for a closed parenthesis ')' to know when the list of assisting players is complete.

File I/O and Command Line Arguments

Your program will run with three command-line arguments. The first argument will be the name of the input file containing information for one or more ice hockey games. The second argument will be the name of the output file where you will write the computed statistics. The third argument will indicate which data table should be printed. Valid options for the third argument are: `--team_stats`, `--player_stats`, or `--custom_stats`. For example, here is a valid command line to your program:

```
hockey_statistics.exe 2012_small.txt 2012_small_output.txt --team_stats
```

Statistics Collected and Output

When `--team_stats` is specified, your program should create a table with the teams sorted by win percentage, with most wins at the top. We define win percentage as the (# of wins + 0.5 * # of ties) / total # of games played. If two teams are tied in win percentage, the team with more total goals is listed first. The teams are sorted alphabetically if they are tied in win percentage and total goals. Each row of the table includes the team's wins, losses, ties, win percentage, goals, and penalties. For example, here is the first part of the output for the `2012_small.txt` dataset:

```
Team Name     W   L   T   Win%  Goals  Penalties
Rensselaer    5   1   0   0.83     25         26
Yale          3   3   0   0.50     10         31
Cornell       2   4   0   0.33     15         33
Clarkson      2   4   0   0.33     11         37
```

The columns should be lined up and neatly justified. Your output spacing may be slightly different as long as it is well-formatted and easy to read. When the `--player_stats` option is specified, your program should instead create a table where the players are sorted first by their total number of goals plus assists (higher total first), and secondarily by their number of penalties (lower total first), and if tied in both aspects, then alphabetically by the name. Your table should look like the excerpt below from the complete table of all players from the `2012_small.txt` dataset:

```
Player Name         Team         Goals  Assists  Penalties
Ryan_Haggerty       Rensselaer       3        6          3
Nick_Bailen         Rensselaer       4        3          1
Greg_Miller         Cornell          2        4          0
Jacob_Laliberte     Rensselaer       2        4          0
```

```
Matt_Neal           Rensselaer      2       4       0
Mike_Zalewski       Rensselaer      4       2       0
Milos_Bubela        Rensselaer      2       4       2
Dustin_Mowrey       Cornell         0       6       4
Andrew_Miller       Yale            1       4       0
```

When the `--custom_stats` option is specified your program should print an additional, interesting and creative statistic that can be calculated from this data. Try to leverage some of the input data that was not utilized in preparing the first two tables. For example, can we show that the trailing team becomes more desperate and aggressive by examining the relative number of penalties in the final period? What are the more common penalties? Are certain teams more guilty of certain violations? How significant is the power play advantage? How many goals are scored during time intervals when the opposing team is playing with only five players (or only four players on the ice if two players from the same team are in the penalty box at the same time)? How significant is the "home field advantage"? Do some teams play relatively better at the beginning or end of the season?

Write a concise description (< 200 words) of your new statistic. What question you are trying to answer with your statistic? What data did you need to organize? What was interesting or challenging about the implementation of this statistic? Put this description in your *plaintext* `README.txt` file along with any other notes for the grader. Be sure to tell the grader which dataset best demonstrates your new statistic and include that output.txt file with your submission. Feel free to create your own dataset and include it and the corresponding output with your submission.

There are many possible data structure class designs that are similarly effective and elegant and appropriate for solving this problem. Your plans for your 3rd statistic will likely influence your overall design. Note that you do not need to store *all* of the information and relationships from the input file. Store the information that is necessary to elegantly organize the data that you need for the three parts of the output. Extra credit will be awarded to particularly interesting statistics that require clever programming.

Useful Code

To control the formatting of your tables, you'll want to read up on the various I/O manipulators: `std::setw(int)`, `std::setprecision(int)`, `std::fixed`, `std::left`, etc. To use those formatting controls you'll need to `#include <iomanip>`. And don't forget about the `sort` function that can be used to order the contents of a `vector`, and that you can create custom comparison functions for sorting.

Program Requirements & Submission Details

Your program should involve the definition of *at least two classes* that have their own `.h` and `.cpp` files, named appropriately. Initially you should focus on the smaller datasets. Once the basics are working you can extend your solution to handle the bigger test cases. Do all of your work in a folder named `hw2` inside of your Data Structures homeworks directory.

Use good coding style when you design and implement your program. Be sure to make up new test cases and don't forget to comment your code! Review the "Good Programming Practices" section on the course webpage to be sure that the TAs will be able give you credit for your hard work. A large portion of the TA grade for this assignment will be awarded for good class design and organization. Make sure to use const and pass-by-reference as appropriate. All class member variables should be private and the public accessor and modifier functions should be thoughtfully designed.

Please use the provided template `README.txt` file for any notes you want the grader to read. **You must do this assignment on your own, as described in the "Academic Integrity for Homework" handout. If you did discuss the problem or error messages, etc. with anyone, please list their names in your `README.txt` file.** When you've finished writing, testing, debugging, and commenting your code, prepare and submit your assignment as instructed on the course webpage. Please ask a TA if you need help preparing your assignment for submission or if you have difficulty writing portable code.

5 points on Test 3 + Test 4, plus passing the hidden simple test case by Wednesday night will earn you a 1 day extension for HW2. Extensions will be posted Thursday morning and show up under "My Late Days/Extensions" on Submitty. Even with the extension and your own late days, you cannot submit later than the end of Saturday. The hidden simple test case uses different (but similar) input to Test 3 to prevent hard coding (not solving the assignment). Hard coding misrepresents your work, is apparent when TAs do their grading, and will result in a 0 on an assignment. If you don't hard code your output, the hidden simple test case score should be the same as your Test 3 score.