

CS 1313 010: Programming for Non-Majors, Fall 2007
Programming Project #5: Sports Scores
Due by 11:20am Wednesday November 7 2007

This fifth project will give you experience writing programs that involve `for` loops and arrays. This project will use the same development process as in Programming Projects #2, #3 #4 and #3 $\frac{1}{2}$, and will be subject to the same rules and grading criteria, along with a few new criteria. **You WON'T need to submit a flow chart for this project.**

This specification will be less detailed than for previous programming projects. **You are expected to know how to perform basic tasks without having to be told explicitly,** based on your experience with previous programming projects.

To get full credit on this programming project, you **MUST** use `for` loops and dynamically allocated arrays appropriately.

I. PROJECT DESCRIPTION

The Big 4 collegiate athletic conference has (surprise!) four teams: Okay University (OU), Misery University (MU), Babbler University (BU) and Okay State University (OSU).

You've been hired as the statistician for the Big 4. Your job is to calculate, for each team:

- the number of wins;
- the number of losses;
- the percentage of wins;
- the mean points that the team scored per game;
- the mean points that were scored against them per game.

(NOTE: Here, “mean” refers to the **arithmetic mean**, described below.

In addition, your job is to determine, for the Big 4 conference as a whole:

- which team has the best offense (i.e., has scored the highest mean points per game);
- which team has the best defense (i.e., has had the lowest mean points scored against them per game);
- which team is the “winningest” (i.e., has the highest winning percentage);
- which team is the “losingest” (i.e., has the lowest winning percentage).

Note that there are no tie scores. Also, you may assume that no two teams will have exactly the same record.

II. PROGRAM DESCRIPTION

Write a program that calculates the statistics described above. The body of the program **MUST** be broken into **THREE SUBSECTIONS**, rather than the usual four subsections (that is, there will be no greeting subsection):

1. an input subsection;
2. a calculation subsection;
3. an output subsection.

Because of how data will be input (see below), **THERE WON'T BE A GREETING SUBSECTION.**

You are **ABSOLUTELY FORBIDDEN** to have:

- **ANY** calculations in the input subsection (and the only outputs should be idiotproofing error messages);
- **ANY** inputs or outputs in the calculation subsection;
- **ANY** inputs or calculations in the output subsection (for this project, `if` statements will not be considered calculations as such).

The three subsections **MUST BE COMPLETELY SEPARATE**, and **MUST BE CLEARLY LABELED**.

Note that you **WON'T** have a greeting subsection in this program.

A. ARRAY DECLARATIONS

You **MUST** use **DYNAMIC** memory allocation and deallocation for **ALL** arrays. Therefore, all arrays **MUST** be declared as **POINTERS**. For example:

```
float* OU_score          = (float*)NULL;
float* OU_opponent_score = (float*)NULL;
```

B. INPUT SUBSECTION

The program will take its input from a data file rather than from a user typing live at the keyboard (see below).

The input data will be in the following format:

- the year that the games were played and the number of games that year, **on a single line**;
- for each team (ordered OU, MU, BU, OSU):
 - for each game:
 - * the team's score and the opponent's score for that game, **on a single line**.

Several such data files will be provided, each representing an individual run. You **MUST** determine how to input the data **BY EXAMINING THE INPUT DATA FILES** (see **HOW TO FIND AND EXAMINE THE INPUT DATA FILES**, below).

Because of how the data will be input, **YOU WON'T PROMPT THE USER FOR THE INPUTS** (see **HOW THE DATA WILL BE INPUT**, below).

You **MUST** store the input data in appropriate one-dimensional arrays.

C. ALLOCATING ARRAYS

You **MUST** use **DYNAMIC** memory allocation and deallocation for **ALL** arrays. Therefore, **ALL** arrays **MUST** be declared as **POINTERS**. Note that **ALL** of the arrays **MUST** be allocated, at run-time, in the execution section, **IMMEDIATELY AFTER INPUTTING AND IDIOTPROOFING THE LENGTH OF THE ARRAYS**. In other words, once you have input and idiotproofed the length of the arrays, you **MUST IMMEDIATELY** allocate the arrays. After allocating each array, the program **MUST** check whether the array allocated successfully, and if not, it **MUST** output a suitable error message, and then it **MUST EXIT**.

For details, see the lecture slide packet "Array Lesson 2," slides 25-31.

D. IDIOTPROOFING

YOU MUST IDIOTPROOF ANY input that needs idiotproofing, to make sure that it has an appropriate value. **YOU** are responsible for figuring out all of the possible cases of idiocy that could come up. **ALL IDIOTPROOFING MUST BE COMPLETED BEFORE ANY CALCULATIONS ARE PERFORMED**; that is, idiotproofing belongs in the input subsection.

Note that, for this programming project, you are **ABSOLUTELY FORBIDDEN** to use `while` loops for your idiotproofing; that is, upon detecting idiocy, the program **MUST EXIT**.

Idiotproof error messages **MUST UNAMBIGUOUSLY** state the nature of the error.

Idiotproofing error messages **MUST** be clear, complete English sentences that **COMPLETELY AND UNAMBIGUOUSLY** state the nature of the error. Thus, no two error messages should be the same. For example, a good error message might be:

```
ERROR: Invalid opponent score -23 for OU game #2.
```

E. CALCULATION SUBSECTION

In the calculation subsection, the program **MUST** calculate the following quantities:

- For each team:
 - the number of games that the team won;
 - the number of games that the team lost;
 - the percentage of games that the team won;
 - the mean number of points that the team scored per game;
 - the mean number of points that the team had scored against them per game.

In any `for` loop in the calculation subsection, you **MUST** calculate **EXACTLY ONE** kind of result; that is, you are **ABSOLUTELY FORBIDDEN** to calculate multiple kinds of results in a single `for` loop. For each team, each quantity that is calculated in a `for` loop **MUST** have **ITS OWN** separate `for` loop; that is, you are **ABSOLUTELY FORBIDDEN** to calculate two (or more) different quantities, or the same quantity for two (or more) different teams, inside the same `for` loop. For example, the `for` loop that calculates the number of games that OU won **MUSTN'T** also calculate the number of games that OU lost, nor the number of games that OSU won.

G. MEAN

Given a list of n real numbers

$$x_1, x_2, \dots, x_n$$

the *mean* of the values in the list is a real number that is an *average*, i.e., a value that is typical of the values in the list. The mean, here denoted \bar{x} (pronounced “x-bar”), is calculated as the sum of all the values in the list, divided by the number of values in the list:

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n} = \frac{x_1 + x_2 + \dots + x_n}{n}$$

Note: $\sum_{i=1}^n x_i$ is known as *summation notation*: $\sum_{i=1}^n x_i = x_1 + x_2 + \dots + x_n$

H. OUTPUT SUBSECTION

In the output subsection, you **MUST** output the following:

- the year;
- the number of games;
- for each team:
 - the win-loss record (i.e., the number of wins and the number of losses);
 - the percentage of wins;
 - the mean points that the team scored per game;
 - the mean number of points that the team had scored against them per game;
- for the conference as a whole:
 - the team with the best offense (i.e., highest mean points scored per game);
 - the team with the best defense (i.e., **lowest** mean points scored against them per game);
 - the “winningest” team (i.e., highest winning percentage);
 - the “losingest” team (i.e., lowest winning percentage).

You may output these quantities in any format that you like, as long as the meaning of the quantities is **CLEARLY EXPLAINED** in the outputs. You are welcome to use format descriptors on your placeholders (for example, "%10.5f"), but you **AREN'T** required to use them.

J. DEALLOCATING ARRAYS

At the end of the program, after the output subsection, you **MUST** deallocate each of the arrays that were allocated in the input subsection, using a `free` statement, and then nullify the pointer, like so:

```
free(OU_opponent_score);
OU_opponent_score = (float*)NULL;
```

For details, see the lecture slide packet “Array Lesson 2,” slides 25-31.

The deallocations **MUST** occur in the **OPPOSITE ORDER** from the allocations; that is, whichever array was allocated first **MUST** be deallocated last, and so on.

III. INPUT DATA FILES

A. HOW TO FIND AND EXAMINE THE INPUT DATA FILES

The input files for your runs can be found on `ssh.coe.ou.edu` in the directory `~neem2345/CS1313pp5`. You can find the names of all of the data files using the Unix `ls` command:

```
% ls ~neem2345/CS1313pp5
```

The directory contains several data files; some are actual data and some are idiotproofing test files. You **MUST** perform the runs in alphabetical order.

Actual (non-idiotproofing) test files have file names beginning with the prefix

```
actual_
```

Idiotproofing test files have file names beginning with the prefix

```
idiot_
```

You **SHOULD EXAMINE** (but not change) the contents of each of the data files using `nano`:

```
% nano ~neem2345/CS1313pp5/actual_football_2002
```

B. HOW THE DATA WILL BE INPUT

For this programming project, **YOU WON'T PROMPT THE USER FOR THE INPUTS**, because there will be no user as such. Instead, the inputs will come from a file. To get the inputs from the file, you'll use a command like this at the Unix prompt:

```
% sports_scores < ~neem2345/CS1313pp5/actual_football_2002
```

This use of a file is referred to as *redirecting input*. The less than symbol `<` indicates that the input will come from the file named `actual_football_2002`. In other words, as far as the program is concerned, the file will appear to be a user typing at the keyboard, and the program will accept input from the file exactly as if that input were being typed at the keyboard by a real live user. Thus, you **MUST** write your `scanf` statements exactly as if a user were going to be typing the data at the keyboard, but without needing to be prompted.

However, because there isn't actually a real live user, it is not necessary to greet the user nor to prompt for inputs; the data file won't understand the prompts anyway, so to speak.

Your run commands **MUST** look like this example:

```
% sports_scores < ~neem2345/CS1313pp5/idiot_football_2002_01
```

This means, "run the executable named `sports_scores`, redirecting input from the file named `idiot_football_2002_01` that's in the directory named `~neem2345/CS1313pp5`."

IV. RUNS

Run this program several times, using the several different input files that are available (see below). The runs **MUST** be in the alphabetical order according to the input file names.

The order of the runs in your script file **MUST** be:

- all `actual_` files, in alphabetical order, followed by
- all `idiot_` files, in alphabetical order.

V. ADDITIONAL GRADING CRITERIA

All grading criteria for Programming Projects #2, #3, #3 $\frac{1}{2}$ & #4 apply, except as stated below.

1. SUBJECTIVE GRADING OF COMMENTS IN THE PROGRAM BODY

In previous programming projects, one of the grading criteria for comments in the program body has been that **EVERY** statement in the program body had to be preceded by a clear, helpful explanatory comment.

- You may choose to write fewer comments than this, in which case **YOU AGREE TO ACCEPT WITHOUT ARGUMENT** the graders' **SUBJECTIVE** opinion on whether the amount and nature of your comments is sufficient.
- Alternatively, you may choose to continue to comply with the old criterion, preceding **EVERY** statement in the program body with a clear, helpful explanatory comment, in which case you are guaranteed to get full credit for this aspect of documentation in the program body.

2. NEW GRADING CRITERIA

1. In the declaration section, the order of declarations **MUST** be:

- (a) named constants: `float` scalars followed by `int` scalars;
- (b) variables, in the following order:
 - i. arrays: `float` arrays followed by `int` arrays;
 - ii. scalars: `float` scalars followed by `int` scalars.

2. **ALL** block closes associated with `for` statements **MUST** be followed, on the same line, by a space, a comment open, a space, the keyword `for`, a space, the counter variable, a space, and a comment close. For example:

```
for (game = first_game; game < number_of_games; game++) {
    scanf("%f %f",
        &OU_score[game], &OU_opponent_score[game]);
} /* for game */
```

3. **Indenting** `for` statements and their associated block closes:

For a given `for` loop, the `for` statement and its associated block close **MUST** be indented identically, and this indentation amount **MUST** be appropriate with respect to their position within the program.

4. **Indenting** inside for loops:

For a given for loop, **ALL** statements **INSIDE** the for loop **MUST** be indented **FOUR SPACES** farther than the for statement and its associated block close. For example:

```
sum = initial_sum;
for (game = first_game; game < number_of_games; game++) {
    sum += OU_score[game];
} /* for game */
OU_score_mean = sum / number_of_games;
```

5. **Commenting** for loops:

Each for loop **MUST** be preceded by a comment that describes what the loop as a whole does. For example:

```
/*
 * Calculate the sum of all of the OU opponents' scores.
 */
for (game = first_game; game < number_of_games; game++) {
    sum += OU_opponent_score[game];
} /* for game */
```

6. **Commenting inside** for loops:

A statement inside a for loop **MUST** be preceded by a comment that describes what the statement does. The comment **MUST** be properly indented, so that the asterisk of the comment lines up with the statement. For example:

```
for (game = first_game; game < number_of_games; game++) {
    /*
     * Increase the OU score sum by the value of the current OU score.
     */
    sum += OU_score[game];
} /* for game */
```

REFERENCES

<http://soonersports.ocsn.com/sports/m-footbl/sched/okla-m-footbl-sched.html>
<http://mutigers.ocsn.com/sports/m-footbl/stats/teamstat.html>
<http://baylorbears.ocsn.com/sports/m-footbl/stats/teamstat.html>
<http://okstate.ocsn.com/sports/m-footbl/okst-m-footbl-sched.html>
<http://soonersports.ocsn.com/sports/w-baskbl/okla-w-baskbl-body.html>
<http://mutigers.ocsn.com/sports/w-baskbl/stats/teamstat.html>
<http://baylorbears.ocsn.com/sports/w-baskbl/sched/bay-w-baskbl-sched.html>
<http://okstate.ocsn.com/sports/w-baskbl/okst-w-baskbl-sched.html>

VI. EXTRA CREDIT

You can receive an extra credit bonus of as much as 5% of the total value of Programming Project #5 by doing the following:

1. Attend at least one CS1313 help session for at least 30 minutes, through Tue Nov 6.
2. During the help session that you attend, work on CS1313 assignments (ideally PP#5, but any CS1313 assignment is acceptable). **YOU CANNOT GET EXTRA CREDIT IF YOU DON'T WORK ON CS1313 ASSIGNMENTS DURING THE HELP SESSION.**
3. Before you leave the help session, fill out **BOTH** halves of the form on the last page of this project specification and have the help session leader (instructor or TA) sign **BOTH** halves. **THE FORM CANNOT BE SIGNED UNTIL IT IS COMPLETELY FILLED OUT.**
4. Attach the bottom half of the form to your PP#5 script printout, **AFTER** the script itself, and keep the top half for your own records.

BONUS VALUE NOTICE: Up through Tue Oct 30, the extra credit bonus will be worth **5%** of the total value of PP#5, but from Wed Oct 31 through Tue Nov 6, the extra credit bonus will be worth **only 2.5%** of the total value of PP#5. That is, **YOU'LL GET TWICE AS MUCH EXTRA CREDIT DURING THE FIRST WEEK AS DURING THE SECOND WEEK.**

NOTE: This extra credit bonus **WON'T** be available on any other programming project unless explicitly stated so in the project's specification.

CS1313 PROGRAMMING PROJECT #5 BONUS REQUEST FORM

Name _____ Lab _____

Help Session Date _____

Help Session Time _____

Instructor Signature _____

Keep this copy for your records.

CS1313 PROGRAMMING PROJECT #5 BONUS REQUEST FORM

Name _____ Lab _____

Help Session Date _____

Help Session Time _____

Instructor Signature _____

Submit this copy.

In your submission, attach this copy **AFTER** your script file printout.

If you put this in the wrong place in your submission, then you **WON'T** get the extra credit.