CS 1313 010: Programming for Non-majors, Fall 2017
Programming Project #2: Census
Due by 10:20am Wednesday September 20 2017

This second assignment will introduce you to designing, developing, testing and debugging your own C program, as well as to declaring variables, inputting and outputting. You will also learn to add new projects to your makefile.

I. PROBLEM DESCRIPTION

You are a software developer for the United States Census Bureau, working on software for the 2020 Census.

The particular program that you're developing will ask three questions about a census subject:

1. the average number of servings of vegetables that the subject eats per week;
2. the average number of text messages that the subject receives per day;
3. the subject's 9-digit ZIP code (also known as ZIP+4);

Notice that the average number of servings of vegetables that the subject eats per week **MIGHT NOT BE AN INTEGER;** for example, a person might average 24.5 servings of vegetables per week. Likewise, the person might average 14.25 text messages received per day.

Note that a number that doesn't have to be an integer is known in mathematics as a *real* number, and is also known in computing as a *floating point* number.

On the other hand, notice that a person's 9-digit ZIP code (ZIP+4) can be expressed as two integer quantities, separated by a hyphen: the basic part and the add-on part[1] — for example, Schenectady NY has a 5-digit ZIP code of 12345, so with an add-on of 6789, the 9-digit ZIP code (ZIP+4) can be expressed as:

12345-6789

So, this program will have a user input two real quantities (average number of servings of vegetables eaten per week, average number of text messages received per day). and two integer quantities (the parts of their 9-digit ZIP code).

Write a program to perform the above task. Note that your program **MUST** begin with a declaration section in which you declare all necessary variables. This will be followed by the execution section (body), which will:

1. greet the user and explain what the program does, and then
2. prompt the user and input the four quantities, and then
3. output the four quantities.

Details follow. Please read all of this specification **CAREFULLY.**

Remember, every word Dr. Neeman writes down is **PURE GOLD.**

IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!
This programming project specification contains many small code examples. In most cases, these code examples will be extremely useful in your actual PP#2. We urge you to use them.

[1] `https://en.wikipedia.org/wiki/ZIP_Code#ZIP.2B4`

II. WHAT TO DO FIRST: Insert the New Project Into Your Makefile

AS THE VERY FIRST STEP, insert the new program into your makefile, so that, when you're ready to compile your new program, you can use `make` instead of having to use the `gcc` command directly (which would risk disaster).

Your C source file **MUST** be named **`census.c`**, and your executable **MUST** be named **`census`**.

Using your preferred text editor (for example, `nano`), edit your makefile to include the following lines **at the TOP of the makefile**, **ABOVE** the make entry for Programming Project #1 (PP#1), with a **blank line** between the entries for PP#2 and PP#1:

```
census:  census.c
         gcc  -o  census  census.c
```

IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!

- **DON'T DELETE THE MAKE ENTRY FOR PROGRAMMING PROJECT #1,** nor any other make entry, **EVER.**
- On the first line, above, between the colon and the name of the C source file, there are one or more tabs (on most keyboards, it's in the upper left, to the left of the Q key). There are **NO SPACES** between the colon and the filename.
- On the second line, immediately before the `gcc`, there are one or more tabs. There are **NO SPACES** immediately before the `gcc`.

IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!
Also in the makefile, alter the `clean` entry (the bottommost entry in the makefile) by putting in another `rm` command, like so:

```
clean:
        rm  -f  my_number
        rm  -f  census
```

NOTES:

- **DON'T DELETE THE `rm` COMMAND FOR PROGRAMMING PROJECT #1,** nor any other `rm` command, **EVER.**
- In the new `rm` command, above, immediately before the `rm`, there are one or more tabs. There are **NO SPACES** immediately before the `rm`.
- **NEVER** put **ANYTHING** on the same line as `clean:` regardless of what it may be that you want to put there. **LEAVE THAT LINE COMPLETELY ALONE!**

IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!
In the `clean` entry, the file to be removed with the `rm` should **ALWAYS ALWAYS ALWAYS** be the **EXECUTABLE** and should **NEVER NEVER NEVER** be a source file.

NOTE: You **MUST** use the lecture slide packets titled "C Introduction," "Variables" and "Standard I/O" to complete this project. You should study every single slide **CAREFULLY.** You can also look at the "Software" and "Constants" packets, but the bulk of the crucial information will be in the "C Introduction," "Variables" and "Standard I/O" packets.

A detailed description of the program follows.

III. DETAILED DESCRIPTION OF THE PROGRAM

HOW TO EDIT A FILE THAT DOESN'T EXIST YET

As noted above, your C source file for PP#2 **MUST** be named **`census.c`**, and your executable **MUST** be named **`census`**.

But when you start working on PP#2, the C source file named **`census.c`** doesn't exist yet.

Question: If a file doesn't exist yet, how can you edit it?

Answer: Pretend that the file already exists, and edit it just as if that were true. The first time you save what you're editing, the file will come to exist.

IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!
When you're editing a file, remember to save your work **OFTEN**, preferably every few minutes.

A. BASIC STRUCTURE OF THE PROGRAM

OTHER THAN COMMENTS (see Grading Criteria, below), the program **MUST** begin with the following preprocessor directive:

```
#include <stdio.h>
```

OTHER THAN COMMENTS (see Grading Criteria, below), the program **MUST** then have the main function header, followed on the next line by the main function block open, and, **AT THE END OF THE PROGRAM**, the main function block close on a line by itself.

The main function block open and the main function block close will each have, following it on its same line, a blank space, followed by the comment open delimiter, followed by a blank space, followed by the word `main` in all lower case, followed by a blank space, followed by the comment close delimiter.

So the basic structure of the program, **OTHER THAN COMMENTS** (see Grading Criteria, below), will look like this:

```
#include <stdio.h>

int main ()
{ /* main */

} /* main */
```

INSIDE the main function — that is, between the block open and the block close of the main function — **FIRST** should be the declaration section, **FOLLOWED BY** the execution section (body) of the program, **IN THAT ORDER.**

B. STRUCTURE OF THE DECLARATION SECTION

In the declaration section, **OTHER THAN COMMENTS** (see Grading Criteria, below), **FIRST** should be **ALL** `float` variable declarations, **FOLLOWED BY ALL** `int` variable declarations.

If you wish, you may put multiple variables of the **SAME DATA TYPE** in the same declaration statement, or you may use an individual declaration statement for each variable, or you may do some of each.

A detailed description of the execution section (body) follows.

C. STRUCTURE OF THE EXECUTION SECTION (BODY)

The **EXECUTION SECTION (BODY)** of the program **MUST** have the following structure and **MUST** be in the following order — interleaving these pieces is **ABSOLUTELY FORBIDDEN:**

1. **Greeting Subsection:** Your program **MUST** begin by outputting a helpful message telling the user what the program does. This message may be a single line of output text, or multiple lines of output text. **ALL OUTPUTS, THROUGHOUT THE ENTIRE PROGRAM, MUST BE MEANINGFUL, COMPLETE ENGLISH SENTENCES.**
2. **Input Subsection**
   (a) Input the first real (floating point) quantity:
      i. **Prompt** the user to input the subject's average number of servings of vegetables eaten per week.
      ii. **Input** the subject's average number of servings of vegetables eaten per week.
   (b) Input the second real (floating point) quantity:
      i. **Prompt** the user to input the subject's average number of text messages received per day.
      ii. **Input** the subject's average number of text messages received per day.
   (c) Input the subject's 9-digit ZIP code (ZIP+4):
      i. **Prompt** the user to input the subject's 9-digit ZIP code (ZIP+4) as two integers, separated by a blank space (see section V on page 7); for example,
      ```
      printf("What is the subject's 9-digit ZIP code, in two parts,\n");
      printf(" separated by a blank space?\n");
      ```
      ii. **Input** the two integer quantities in the above order, **using a single `scanf` statement** to read both of the `int` variables from a single line of input text.
3. **Output Subsection:**
   (a) Output the subject's average number of servings of vegetables eaten per week. including helpful explanatory text; for example, the **output text** might look like:
   ```
   The subject eats an average of 24.5 servings of vegetables per week.
   ```
   (b) Output the subject's average number of text messages received per day, including helpful explanatory text.
   (c) Output the subject's 9-digit ZIP code (ZIP+4), including helpful explanatory text. This output **MUST** use the 2-part hyphenated notation shown on page 1. The output may contain extra spaces between the numbers and the hyphens. For example, the **output text** might look like:
   ```
   The subject's 9-digit ZIP code was 12345-6789.
   ```

We encourage you to make your comments and outputs entertaining, but not profane or offensive.

The real (floating point) quantities that you output may come out with a weird format, like so:

```
The subject eats an average of 24.500000 servings of vegetables per week.
```

For runs #2 and #3, which will use values that you've chosen, you may see something like this:

```
The subject eats an average of 31.299999 servings of vegetables per week.
```

If either of these happens, **DON'T PANIC! THESE ARE NORMAL**, so don't worry about them.

IV. ADVICE ON HOW TO WRITE A PROGRAM

When you're writing a program:

1. write a little bit of the source code;
2. make;
3. if the make fails, then debug the source code;
4. when the make succeeds, then run;
5. if the run fails, then debug the source code;
6. when the run succeeds, then go on and write a little more, and so on.

For example, in the case of this program:

1. Start by writing the skeleton of the source code: the `#include` directive, the `main` function header, the `main` function block open and block close. and appropriate comments for these items. Then make, then run. (This run won't be very interesting, unless the program crashes, in which case debug it.)
2. Then, write the variable declarations, with appropriate comments. Then make, then run. (This run won't be very interesting, unless the program crashes, in which case debug it.)
3. Then, write the greeting subsection, with appropriate comments. Then make, then run.
4. Then, write the input subsection, with appropriate comments. Then make, then run.
5. Then, write the output subsection, with appropriate comments. Then make, then run.

Also, in your preferred text editor (for example, `nano`), **FREQUENTLY SAVE YOUR WORK.** Specifically, we recommend that, in your preferred text editor, you **SAVE YOUR WORK EVERY FEW MINUTES.** (For example, in `nano`, press `Ctrl-O` to save your work, and do this every few minutes.)

NOTE: When you write a comment open delimiter (slash asterisk), you should **IMMEDIATELY** write the comment close delimiter (asterisk slash) so that you don't end up forgetting it later — and then you can put the actual comment text in between.

Likewise, when you write a block open delimiter (open curly brace), you should **IMMEDIATELY** write the block close delimiter (close curly brace) so that you don't end up forgetting it later — and then you can put the actual source code text of the `main` function in between.

IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!
When you're editing a file, remember to save your work **OFTEN**, preferably every few minutes.

V. RUNS

In the script session that produces your script file (described below), you **MUST** run your program **three times**. For the first run, use the following inputs:

- average number of servings of vegetables eaten per week: 24.5;
- average number of text messages received per day: 14.25;
- 9-digit ZIP code (ZIP+4) in two parts: 12345 6789 — note that this should be **input** as
  `12345 6789`
  but it should be **output** as
  `12345-6789`
  (along with helpful explanatory text).

IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!
When you input the subject's 9-digit ZIP code (ZIP+4) at runtime, **YOU DON'T INPUT A HYPHEN.** Instead, you separate the pieces of the subject's 9-digit ZIP code (ZIP+4) with spaces (preferred) or carriage returns.

For the second and third runs, choose any **VALID** answers to these questions that you want, but all three runs **MUST** have different inputs for all questions; that is, every question **MUST** have different answers for each of the three runs, and all inputs within a run must differ from each other.

VI. WHAT TO SUBMIT

Before creating your script file, **THOROUGHLY TEST AND DEBUG YOUR PROGRAM.**

Once you are satisfied with your program, you are ready to create your script file. Your script file **MUST** be named `pp2.txt`. Use the procedure described in the Programming Project #1 specification to create your script file, except replacing `census` for `my_number` and `census.c` for `my_number.c`, doing three runs using the input values that you've tested (section V, above).

IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!
CAREFULLY PROOFREAD THE PRINTOUT OF YOUR SCRIPT FILE. Frequently, students lose **SIGNIFICANT CREDIT** because of failure to proofread. Especially, **CHECK YOUR MAKE COMMANDS** to be sure (a) that you did them and (b) that they worked properly.

Cover and Summary: You **MUST** create a cover page and a summary essay following the same rules as in Programming Project #1.

Binding Order: You **MUST** bind, **IN THIS ORDER,** the cover page, then the summary, then the script, then the bottom half of the extra credit form (at the very end), if applicable (see below). Thus, for binding PP#2, follow the same rules as in PP#1.

Upload to Canvas: You also **MUST** upload your source file and script file to Canvas, as you did for PP#1, but into the place for this project.

VII. GRADING CRITERIA

The following grading criteria will apply to **ALL** CS1313 programming projects, unless explicitly stated otherwise.

Grading Criteria for Cover Page, Summary Essay, Script File and Upload to Canvas:

The rules and grading criteria for the cover page, summary essay, script file and upload to Canvas, as described in the Programming Project #1 specification, also apply to the cover page, summary essay, script file and upload for Programming Project #2, and will also apply to all future Programming Projects unless explicitly stated otherwise. Failure to upload the correct files to the correct place in Canvas by the PP#2 deadline may cost you up to 5% of the total value of PP#2, right off the top before any other deductions are applied.

IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!

You **MUST** properly do the `make clean` and `make census` steps in your script. **FAILING TO PROPERLY DO THE `make clean` AND/OR `make census` STEPS, OR HAVING A COMPILE FAIL DUE TO ERRORS, WILL COST YOU AT LEAST 50% THE POINTS FOR THIS PROGRAMMING PROJECT,** right off the top before any other deductions are applied. **COMPILER WARNINGS in response to the `make census` step — other than the "clock skew" warning — WILL COST YOU AT LEAST 25% OF THE POINTS FOR THIS PROGRAMMING PROJECT,** right off the top before any other deductions are applied.

Grading Criteria for C Source Code

1. **Documentation MUST** be similar to that in `my_number.c`, and will count for **at least** 10% of the total value of this project.
   (a) The program **MUST** be preceded by a comment block, as shown in `my_number.c`.
   (b) The declaration section and the execution section (body) **MUST** be clearly labeled, as shown in `my_number.c`.
   (c) Variable declarations **MUST** be preceded by comments explaining the nature and purpose of each declared name, as shown in `my_number.c`.
   (d) Each subsection of the execution section (body) of the program — greeting, input, output — **MUST** be clearly labeled, as shown in `my_number.c`.
   (e) **EVERY executable statement MUST** be preceded by a comment that **clearly** explains what the statement does, well enough so that even a non-programmer could understand. **Exception:** Multiple `printf` statements in a row that together output a single message need a comment only before the first of them.
   (f) **ALL comments MUST** use the format shown below. Specifically, the first line of the comment **MUST** simply be the comment open delimiter (slash asterisk), and the last line **MUST** simply be the comment close delimiter (asterisk slash). All other lines **MUST** have, as their first non-blank character, an asterisk, followed by a blank space, followed by the text of the comment. **ALL** of the asterisks **MUST** line up with the text of the program statement that the comment describes. For example:

```
/*
 * Output to the terminal screen the subject's
 * average number of servings of vegetables eaten per week.
 */
printf("The subject eats an average of %f servings",
    average_weekly_vegetable_servings_eaten);
printf("  of vegetables per week.\n",
```

2. **Block open/block close comments:** The block open and block close for the `main` function **MUST** each be followed, on the same line, by a comment indicating that the block that they begin and end is the `main` function. Specifically, the line with the block open or the block close **MUST** have the following structure: the block open or block close, followed by a single blank space, followed by the comment open delimiter, followed by a single blank space, followed by the keyword `main`, followed by a single blank space, followed by the comment close delimiter. For example:

```
{ /* main */

} /* main */
```

3. **Section order:** The section order **MUST** be as follows: the declaration section, followed by the executable section (body), as shown in `my_number.c`. Therefore, **ALL** declarations **MUST** appear **BEFORE ANY** executable statements.
4. **Identifier structure:** Identifiers such as variable names **MUST** be in **ALL LOWER CASE,** except where upper case is appropriate as part of a proper name (for example, `population_of_Oklahoma`). Adjacent words in an identifier **MUST** be separated by an underscore.
5. **Favorite professor rule for identifiers:** Identifiers such as variable names **MUST** strictly observe the "favorite professor" rule, as described in the lecture slides. Meaningless, obscure or cryptic names will be penalized, as will abbreviations that aren't in common use in non-programming contexts.
6. **Data types: EVERY** variable **MUST** have an **appropriate data type.** Inappropriate data types will be penalized.
7. **Variable declaration grouping:** Variable declarations **MUST** be grouped by data type; that is, you **MUST** first declare **ALL** `float` variables, followed by **ALL** `int` variables.
8. **Variable declaration statement structure MUST** be as follows: the indentation, followed by the data type, followed by one or more blank spaces, followed by the name of the variable, followed by the statement terminator (or you may declare multiple variables in the same declaration statement, separated by commas and with a statement terminator at the end, as shown in the lecture slides).
9. **Variable declaration spacing MUST** have the following property: The first character of the first variable name of **ALL** declaration statements, regardless of data type, should be in the same column of source code text. In the case of PP#2, this means that, in a `float` variable declaration, there should be **EXACTLY ONE** blank space after the keyword `float`, and in an `int` variable declaration, there should be **EXACTLY THREE** blank spaces after the keyword `int`. (In other Programming Projects, the blank space counts may differ, but the principle will be the same.) For example:

```
int main ()
{ /* main */
    float average_weekly_vegetable_servings_eaten;
    float average_daily_text_messages_received;
    int   zip_code_basic_part, zip_code_addon_part;
} /* main */
```

10. **Multiple variables in the same declaration statement:** An individual declaration statement may declare multiple variables, but it is **STRONGLY RECOMMENDED** that this be done only when the variables are very closely related to one another. If an individual declaration statement declares multiple variables, then in its comma-separated list of variable names, each comma **MUST** be followed by a single blank space, as shown in the example just above. If the multiple variables would exceed the proper length of a line of source code text, then the declaration statement may continue on to the next line, in which case, in the subsequent line(s) of the declaration statement, the first variable name of each line should line up with the first variable name of the first line of the declaration statement.
11. **Indentation MUST** be used properly and consistently. The `#include` directive and the `main` function header **MUST NOT BE INDENTED AT ALL** (that is, they **MUST** begin in the leftmost column). Likewise, the `main` function's block open (open curly brace {) and the block close (close curly brace }) **MUST NOT BE INDENTED AT ALL. ALL OTHER STATEMENTS,** both declarations and executable statements, **MUST** be indented an additional **FOUR SPACES** beyond the function header. For example:

```
#include <stdio.h>

int main ()
{ /* main */
    float average_weekly_vegetable_servings_eaten;

    printf("The subject eats an average of %f servings",
        average_weekly_vegetable_servings_eaten);
    printf("  of vegetables per week.\n",
} /* main */
```

NOTE: If a statement uses more than one line of source code text, then the second line (and beyond) of source code text of that statement **MUST** be indented farther, preferably 4 spaces farther than the first line of the statement, as shown in the example just above.

IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!
Indenting is SO INCREDIBLY IMPORTANT that it's worth at least 10% of your overall score on PP#2 (A FULL LETTER GRADE)!

12. **Subsection order in the execution section:** In the execution section, the subsection order **MUST** be as follows: the greeting subsection, followed by the input subsection, followed by the output subsection.
13. **Execution subsection contents:** In the execution section: the greeting subsection is **ABSOLUTELY FORBIDDEN** to contain any inputs; the only outputs that may be in the input subsection are prompts for inputs; the output subsection is **ABSOLUTELY FORBIDDEN** to contain any inputs.
14. **The length of each line of C source code text MUST** be less than 80 characters (the width of a typical PuTTY window); 72 characters or less is preferred.
15. **The length of each line of output text MUST** be less than 80 characters; 72 characters or less is preferred.

16. **`printf` WITHOUT placeholders:** **EVERY** `printf` statement that **DOESN'T** contain any placeholders **MUST** have the following structure: indentation, followed by the word `printf`, followed by an open parenthesis, followed by a double quote, followed by the text of the string literal (probably but not necessarily ending with a newline), followed by a double quote, followed by a close parenthesis, followed by the statement terminator. For example:
```
printf("What is the subject's 9-digit ZIP code, in two parts?\n");
```
17. **`printf` WITH placeholders:** **EVERY** `printf` statement that **DOES** contains one or more placeholder(s) **MUST** have the following structure: indentation, followed by the word `printf`, followed by an open parenthesis, followed by a double quote, followed by the text of the string literal including placeholder(s) (probably but not necessarily ending with a newline), followed by a double quote, followed by a comma, followed by a blank space, followed by the comma-separated list of variables whose values are replacing the placeholder(s), with a blank space after each comma, followed by a close parenthesis, followed by the statement terminator. For example:
```
printf("The subject eats an average of %f servings",
    average_weekly_vegetable_servings_eaten);
printf("  of vegetables per week.\n",
```
18. **Newlines in `printf` statements.** Every line of output text **MUST** end with a newline. The last (or only) `printf` statement for a particular line of output text **MUST** have a newline `\n` as the **LAST** characters in its string literal, immediately before the double quote. See above for examples.
19. **`scanf`:** **EVERY** `scanf` statement **MUST** have the following structure: indentation, followed by the word `scanf`, followed by an open parenthesis, followed by a double quote, followed by the text of the string literal including placeholder(s), followed by a double quote, followed by a comma, followed by a blank space, followed by the comma-separated list of variables whose values are being input — each preceded by an ampersand `&` with no blank space after the ampersand — with a blank space after each comma, followed by a close parenthesis, followed by the statement terminator. For example:
```
scanf("%f", &average_weekly_vegetable_servings_eaten);
```
20. **Newlines in `scanf` statements are FORBIDDEN.** A `scanf` statement **CANNOT** have a newline `\n` anywhere in its string literal. See above for an example.
21. **String literals MUST NOT** have carriage returns embedded inside them. So, the following statement is **BAD BAD BAD:**
```
printf("This is a very long sentence so it needs
        to be broken into pieces.\n");
```
The output text above **MUST** be broken into multiple `printf` statements, so the following statements are **GOOD:**
```
printf("This is a very long sentence so it needs");
printf(" to be broken into pieces.\n");
```
Note that the resulting line of output text **MUST** be less than 80 characters long, preferably no more than 72.
22. Once you've created your script file, you are **ABSOLUTELY FORBIDDEN** to alter your script file **IN ANY WAY, EVER**. (But, you may replace it with a completely new script file.)

EXTRA CREDIT

You can receive an extra credit bonus of as much as 5% of the total value of PP#2 by doing the following:

1. Attend at least one regularly scheduled CS1313 help session for at least 30 minutes, through Wed Sep 20.
2. During the regularly scheduled help session that you attend, work on CS1313 assignments (ideally PP#2, 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 regularly scheduled 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 IN INK.** Use of pencil on these forms is **ABSOLUTELY FORBIDDEN.**
4. Attach the bottom half of the form to your PP#2 script printout, **AFTER** the script itself, and keep the top half for your own records.

VALUE OF THE EXTRA CREDIT BONUS:

- for attending a regularly scheduled help session Mon Sep 11 - Wed Sep 13: 5% of the total value of PP#2;
- for attending a regularly scheduled help session Mon Sep 18 - Wed Sep 20: 2.5% of the total value of PP#2.

NOTES:

- You can only get the extra credit bonus **ONCE** per programming project that offers it.
- This extra credit bonus **WON'T** be available on any other programming project unless explicitly stated so in that project's specification.

THIS PAGE INTENTIONALLY LEFT BLANK.

CS1313 PROGRAMMING PROJECT #2 BONUS REQUEST FORM

Name _____ Lab _____
Help Session Date _____
Help Session Time (Arrive) _____ Help Session Time (Depart) _____

Instructor Signature _____

Keep this copy for your records.

CS1313 PROGRAMMING PROJECT #2 BONUS REQUEST FORM

Name _____ Lab _____
Help Session Date _____
Help Session Time (Arrive) _____ Help Session Time (Depart) _____

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.