CS 1313 010: Programming for Non-majors in C, Spring 2025
Programming Project #2: Census
Due by Wednesday February 12 2025 9:50am Central Time

This 2nd assignment will help you learn how to design, develop, test and debug your own C program, as well as declaring variables, inputting and outputting. You will also learn to add new projects to your makefile. You **MUST** follow all the instructions you see in this document.

I. PROBLEM DESCRIPTION

You are a software developer for the United States Census Bureau, working on the 2025 Census.

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

1. the average number of cell phone calls that the subject makes per week;
2. the average number of photos that the subject posts to social media per month;
3. the subject's 9-digit ZIP code (also known as ZIP+4).

Notice that the average number of cell phone calls that the subject makes per week **MIGHT NOT BE AN INTEGER.** For example, a person might average 12.25 cell phone calls per week. Likewise, a person might average 8.75 photos posted to social media per month.

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 integers, 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) is:

12345-6789

So, this program will have a user input two real numbers (average number of cell phone calls made per week, average number of photos posted to social media per month). and two integers (the parts of their 9-digit ZIP code), and then output those numbers in a specific format.

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 numbers, and then
3. output the four numbers.

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 (PP) specification contains many small code examples. Most of these code examples will be very useful in your actual PP#2. **WE URGE YOU TO USE THEM.**

NOTE: Except where and as explicitly permitted in writing (for example, in a PP specification, as above), you are **ABSOLUTELY FORBIDDEN** to **COPY EVEN A SINGLE CHARACTER** from, or to have **ANY** shared code with, **ANY** other entity, whether a human being, a text resource,

[1] `https://en.wikipedia.org/wiki/ZIP_Code#ZIP+4`

a computing resource or anything else, whether in person, on a local computer, online or anywhere else. It's **INCREDIBLY EASY** for us to detect copying, so **DON'T EVEN THINK ABOUT IT!**

II. WHAT TO DO FIRST: Insert an Entry for the New Program into Your Makefile

AS THE VERY FIRST STEP, in your makefile, insert a makefile entry for the new program, 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`

Here's how: Using your preferred text editor (for example, `nano`), edit your makefile (which is named `makefile`) to include the following lines **at the TOP of the makefile**, **ABOVE** the make entry for 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!!!

- In the `gcc` command, the filename after `-o` should be the **executable** (**NO** .c extension), and the filename at the end should be the **C source file** (**WITH** the .c extension).
- **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, update the `clean` entry (the bottommost entry in the makefile) by putting in another `rm` command, as the **LAST** `rm` command in the `clean` entry, like this:

```
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** (for PP#2, `census`) and **NEVER NEVER NEVER** a source file (for example, it **SHOULDN'T** be `census.c`).

III. WHAT TO DO SECOND: EXAMPLE PROGRAM

DO ALL OF THE FOLLOWING BEFORE DOING ANY OF SECTION IV.

The following is worth 10% of the total value of PP#2 (a full letter grade):

1. In the lecture slide packet titled "Standard I/O Lesson," on slide #21, find the example C source file named:
   `read_list.c`
2. If you aren't already logged in to `ssh.ou.edu`, log in.
3. If you aren't already in your `CS1313` subdirectory, change directory to there.
   NOTE: For **EVERY** Programming Project, you **MUST** do **ALL** of your work in your `CS1313` directory, **WITHOUT EXCEPTION**.
4. Using your preferred text editor (for example, `nano`), edit your `makefile` to add an entry for `read_list`, as described in this PP#2 specification, page 2, section II. Put the new `make` entry **BETWEEN** the `makefile` entry for `census` and the `makefile` entry for `my_number`, with a blank line above this new `makefile` entry and a blank line below this new `makefile` entry, like this:

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

read_list:       read_list.c
                 gcc  -o  read_list  read_list.c

my_number:       my_number.c
                 gcc  -o  my_number  my_number.c
```

5. While still in your `makefile`, update the `clean` entry to add an `rm` command for `read_list`, as described in section II, between the `rm` command for `census` and the `rm` command for `my_number`, like this:

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

6. Save and exit from editing your `makefile`.
7. Using your preferred text editor (for example, `nano`), edit a new C source file named:
   `read_list.c`
   NOTE: You can find out how to edit a file that doesn't exist yet, in this PP#2 specification, at the top of page 5, the beginning of section IV.
8. Type in, **BY HAND**, the C source code for
   `read_list.c`
   that appears in "Standard I/O Lesson," slide #21.
   NOTE: Copy-and-paste **WON'T WORK PROPERLY** for this!
9. In `read_list.c`, make sure that the indenting is correct, in compliance with PP#2 grading criterion #11, on page 11 of this PP#2 specification.
10. In `read_list.c`, make sure to comply with PP#2 grading criterion #21, on page 12 of this PP#2 specification.
11. When you're done editing `read_list.c`, save and exit.

12. Compile the program, using the following command:
    `make read_list`
13. If the program doesn't compile, then fix the C source file `read_list.c`, and return to step III.12, above.
14. Run the executable, using the input values in "Standard I/O Lesson," slide #22.
    DON'T do the `gcc` command!
15. If the executable produces the wrong output, then fix the C source file `read_list.c`, and return to step III.12, above.
16. Once the program runs correctly, using your preferred text editor (for example, `nano`), edit the C source file `read_list.c`.
17. Comment the example program's C source file `read_list.c` throughout the entire C source file, using the PP#2 grading criteria #1(a)-(g) on page 9, based on the comments in `my_number.c` from PP#1. See, for example, "Software Lesson 1," slides #20-22.
18. When you're done adding in all the comments to `read_list.c`, save and exit.
19. Compile the program, again using the following command:
    `make read_list`
20. If the program doesn't compile, then fix the C source file `read_list.c`, and return to step III.19, above.
21. Run the executable, again using the input values on "Standard I/O Lesson," slide #22.
    DON'T do the `gcc` command!
22. If the executable produces the wrong output, then fix the C source file `read_list.c`, and return to step III.19, above.
23. Once the commented version of the program works correctly, create a script file named `pp2_example.txt`, using the method described in the PP#1 specification, section VIII, pages 18-20, but using the script file named
    `pp2_example.txt`
    and the source file named
    `read_list.c`
    and the executable named
    `read_list`
    And do the **ONE RUN** described in step 21, above.

IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!
This example program here in section III is **SO IMPORTANT** that it's worth 10% of the total value of PP#2 (a **FULL LETTER GRADE**).

It **MUST BE COMPLETED BEFORE CONTINUING ON TO SECTION IV AND BEYOND** (writing `census.c` and so on).

If you do **ANY** of section IV or beyond (writing `census.c` and so on) before completing **ALL** of section III, then you might **LOSE ALL OF THE VALUE OF THIS SECTION** — that is, you'd **LOSE A FULL LETTER GRADE** on PP#2.

And it'll be straightforward for us to tell whether you've violated this rule.

Therefore, **COMPLETE** section III **BEFORE** going on to anything else.

IV. DETAILED DESCRIPTION OF THE NEW PROGRAM

WARNING: If you haven't already completed section III, go back and complete it **BEFORE** doing this section, or you could lose a **FULL LETTER GRADE** on PP#2.

HOW TO EDIT A FILE THAT DOESN'T EXIST YET

As noted in section II, 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

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.

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), the program **MUST** then have the main function header, followed, on the next line, by the main function block open on a line by itself, 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, on the same line, a blank space, then the comment open delimiter, then a blank space, then the word `main` in all lower case, then a blank space, then the comment close delimiter.

So the basic structure of the program, **OTHER THAN COMMENTS**, will look like this:

```
#include <stdio.h>

int main ()
{ /* main */

} /* main */
```

WE URGE YOU TO USE THE CODE JUST ABOVE IN YOUR PP#2!

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 some of each.

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) Prompt for and input the first real (floating point) quantity:
      i. **Prompt** the user to input the subject's average number of cell phone calls made per week.
      ii. **Input** the subject's average number of cell phone calls made per week.
   (b) Prompt for and input the second real (floating point) quantity:
      i. **Prompt** the user to input the subject's average number of photos posted to social media per month.
      ii. **Input** the subject's average number of photos posted to social media per month.
   (c) Prompt for and 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 between the pair of integers (see section VI); for example,
      ```
      printf("What is the subject's 9-digit ZIP code, in two parts,\n");
      printf(" separated by a blank space?\n");
      ```
      WE URGE YOU TO USE THE CODE JUST ABOVE IN YOUR PP#2!
      ii. **Input** the two integers in the above order, **using a single `scanf` statement** to input both of the `int` variables from a single line of input text.
3. **Output Subsection:**
   (a) Output the subject's average number of cell phone calls made per week, including helpful explanatory text; for example, the **output text** might look like:
   ```
   The subject makes an average of 12.25 cell phone calls per week.
   ```
   (b) Output the subject's average number of photos posted to social media per month, 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. 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) numbers that you output may come out with a weird format, like this:

```
The subject makes an average of 12.250000 cell phone calls per week.
```

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

```
The subject makes an average of 13.599999 cell phone calls per week.
```

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

V. 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 should **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.

VI. 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 cell phone calls made per week: 12.25;
- average number of photos posted to social media per month: 8.75;
- 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, separate the pieces of the 9-digit ZIP code (ZIP+4) with spaces 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.

NOTE: You are **ABSOLUTELY FORBIDDEN** to use any ZIP+4 that starts with a zero in the leftmost digit of either of the two pieces.

VII. WHAT TO SUBMIT

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

Once you are satisfied with your program, create your script file, which **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`, and doing three runs using the input values that you've tested (section VI, above).

IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!
CAREFULLY PROOFREAD 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.

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

Download to PC: You **MUST** download your example script file `pp2_example.txt`, your C source file `census.c` and your script file `pp2.txt` to the PC that you want to upload to Canvas from, as you did for PP#1 (see the PP#1 specification, pages 21-22, section IX, but using the filenames just listed).

Upload to Canvas: You **MUST** upload your summary essay, your example script file `pp2_example.txt`, your C source file `census.c` and your script file `pp2.txt` to Canvas, as you did for PP#1, but into this project's Canvas dropbox (see the PP#1 specification, pages 24-25, section X.2, using the filenames listed just above).

VIII. GRADING CRITERIA

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

Grading Criteria for Summary Essay, Script Files and Upload to Canvas:

The rules and grading criteria for the summary essay, the script file and uploading to Canvas, as described in the Programming Project #1 specification, also apply to the summary essay, the script files and uploading 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% OF 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 in that comment **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 cell phone calls made per week.
 */
printf("The subject averages %f cell phone calls made per week.\n",
    average_weekly_cell_phone_calls);
```

   WE URGE YOU TO USE THE CODE JUST ABOVE IN YOUR PP#2!

   (g) C++-style comments that start with `//` are **ABSOLUTELY FORBIDDEN**.

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` in all lower case, 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 execution 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 (Variables Lesson, slide #35). 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_cell_phone_calls;
    float average_monthly_photos_posted_to_social_media;
    int   zip_code_basic_part, zip_code_addon_part;
} /* main */
```

WE URGE YOU TO USE THE CODE JUST ABOVE IN YOUR PP#2!

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 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_cell_phone_calls;

    printf("The subject averages %f cell phone calls per week.\n",
        average_weekly_cell_phone_calls);
} /* main */
```

WE URGE YOU TO USE THE CODE JUST ABOVE IN YOUR PP#2!

NOTES:
(i) In CS1313, in any C source file, indenting with tabs instead of spaces is **FORBIDDEN**.
(ii) 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)! (And likewise on future PPs.)

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; in the input subsection, the only outputs that are allowed 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");
printf("separated by spaces)?\n");
```

WE URGE YOU TO USE THE CODE JUST ABOVE IN YOUR PP#2!

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 averages %f cell phone calls made per week.\n",
    average_weekly_cell_phone_calls);
```

WE URGE YOU TO USE THE CODE JUST ABOVE IN YOUR PP#2!

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 that ends that string literal. 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 for the 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_cell_phone_calls);
```

WE URGE YOU TO USE THE CODE JUST ABOVE IN YOUR PP#2!

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. For PP#2, you are allowed to use **ONLY** the following kinds of programming constructs in your C source file, but **NO OTHER KINDS OF PROGRAMMING CONSTRUCTS**:
    - comments;
    - preprocessor directives;
    - main function header;
    - block open and block close of the main function;
    - variable declarations;
    - `printf` statements;
    - `scanf` statements.

    Use of any other kind of programming constructs in your C source file might result in **SEVERE PENALTIES, UP TO 50% OFF BEFORE ANY OTHER DEDUCTIONS ARE APPLIED,** at our sole discretion.
23. 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

HELP SESSION BONUS EXTRA CREDIT

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

1. Attend at least one regularly scheduled CS1313 help session for at least 30 minutes, through Wed Feb 12.
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.**

VALUE OF THE HELP SESSION EXTRA CREDIT BONUS:

- for attending a regularly scheduled help session Mon Feb 3 - Tue Feb 4: 5% of the total value of PP#2;
- for attending a regularly scheduled help session Mon Feb 10 - Wed Feb 12: 2.5% of the total value of PP#2.

PP#2 CHECKLIST

- ☐ SSH window size: In the window that I use to access `ssh.ou.edu` (for example, PuTTY in Microsoft Windows or the MacOS terminal window in MacOS), I always verify that my window size is **EXACTLY** 80 columns wide by **EXACTLY** 40 rows high and that I've set that window to forbid resizing (as described in the PP#1 specification, page 3, item I.1.a.xi-xii).
- ☐ CS1313 subdirectory use: **ALL** of my PP#2 work is in my `CS1313` subdirectory, and this will be true for **ALL** of my future CS1313 work (as described in the PP#1 specification, page 10, item IV.4).
- ☐ Edit `makefile`: I edited my `makefile`, using an editor in Linux, for example `nano` (as described in the PP#2 specification, page 2, section II).
- ☐ PP#2 entry in `makefile`: In my `makefile`, I have added an entry for PP#2, for the executable named `census` and its source file `census.c` (as described in the PP#2 specification, page 2, section II, before the first bullet list).
- ☐ PP#2 entry in `makefile` is the first entry: In my `makefile`, the entry for PP#2 (`census` and `census.c`) is the **FIRST** entry in my `makefile` (as described in the PP#2 specification, page 2, section II).
- ☐ PP#2 entry in `makefile` followed by blank line: In my `makefile`, the entry for PP#2 (`census` and `census.c`) has a blank line between this new entry and the entry for PP#1 (as described in the PP#2 specification, page 2, section II).
- ☐ PP#2 entry in `makefile` first line has tab(s) instead of blank spaces: In my `makefile`, in the entry for PP#2, for the executable named `census` and its source file `census.c`, on the first line of this `makefile` entry, between the colon and the name of the source file `census.c`, I used one or more tabs and no blank spaces (as described in the PP#2 specification, page 2, section II, first bullet list, 3rd bullet).
- ☐ PP#2 entry in `makefile`: second line starts with tab(s) instead of blank spaces: In my `makefile`, in the entry for PP#2, for the executable named `census` and its source file `census.c`, on the second line of this `makefile` entry, before the `gcc` command, I used one or more tabs and no blank spaces (as described in the PP#2 specification, page 2, section II, first bullet list, 4th bullet).
- ☐ Changed `clean` entry in `makefile`: In my `makefile`, I have changed the `clean` entry, by adding the following for PP#2:
  `rm -f census`
  as the **LAST** `rm` command in the `clean` entry (as described in the PP#2 specification, page 2, section II, between the bullet lists).
- ☐ Tab(s) instead of blank spaces in `makefile clean` entry: In my `makefile`, in the `clean` entry, the `rm` command is preceded by one or more tabs but no blank spaces, and has the executable name `census` (**WITHOUT** the `.c`) immediately after the `-f` (as described in the PP#2 specification, page 2, section II, second bullet list, second bullet).
- ☐ Nothing after the colon on first line of `makefile clean` entry: In my `makefile`, in the `clean` entry, the `clean:` line has **NOTHING ELSE** on the same line (as described in the PP#2 specification, page 2, section II, second bullet list, last bullet).

CHECKLIST ITEMS FOR TYPING IN, COMPILING AND RUNNING THE EXAMPLE PROGRAM FROM THE LECTURE SLIDES

- ☐ Edit in Unix/Linux, NOT in Windows nor in MacOS: When editing my C source file, I edited my C source file directly on `ssh.ou.edu` using a Unix text editor such as `nano`, **NOT** in Microsoft Windows using a Microsoft Windows editor, **NOR** in MacOS using a MacOS editor (as described in the PP#1 specification, page 14, item VI.2).
- ☐ Add example program to your **`makefile`**: I added an entry for the example program to my `makefile` (as described in the PP#2 specification, page 3, item III.4).
- ☐ Add example program to your `clean` entry: I added the example program's executable to the `clean` entry in my `makefile` (as described in the PP#2 specification, page 3, item III.5).
- ☐ Type in by hand, **NOT** copy-and-paste: I typed in the example program from the lecture slides, intead of copy-and-paste (as described in the PP#2 specification, page 3, item III.8).
- ☐ Indenting in original (uncommented) example C source file: I made sure that the original (uncommented) example program from the lecture slides was correctly indented, in compliance with PP#2 grading criterion #11 (as described in the PP#2 specification, page 3, item III.9).
- ☐ No carriage returns in string literals in original (uncommented) example C source file: I made sure that the original (uncommented) example program from the lecture slides had no carriage returns inside string literals, in compliance with PP#2 grading criterion #21 (as described in the PP#2 specification, page 3, item III.10).
- ☐ Compile original (uncommented) C source file: I successfully compiled the original (uncommented) example program from the lecture slides (as described in the PP#2 specification, page 4, item III.12).
- ☐ Run original executable: I successfully ran the original (uncommented) example program's executable (as described in the PP#2 specification, page 4, item III.14).
- ☐ Add comments to the example C source file: I added comments throughout the entire example C source file, using the PP#2 grading criteria #1(a)-(g), based on the comments in `my_number.c` from PP#1 (as described in the PP#2 specification, page 4, item III.17).
- ☐ Compile commented C source file: I successfully compiled the commented example program from the lecture slides (as described in the PP#2 specification, page 4, item III.19).
- ☐ Run commented executable: I successfully ran the commented example program's executable (as described in the PP#2 specification, page 4, item III.21).
- ☐ Create script file for example program: I successfully create a script file for the commented example program (as described in the PP#2 specification, page 4, item III.23).

CHECKLIST ITEMS FOR CREATING AND TESTING YOUR OWN `census.c`

- ☐ Edit in Unix/Linux, NOT in Windows nor in MacOS: When editing my C source file, I edited my C source file directly on `ssh.ou.edu` using a Unix text editor such as `nano`, **NOT** in Microsoft Windows using a Microsoft Windows editor, **NOR** in MacOS using a MacOS editor (as described in the PP#1 specification, page 14, item VI.2).
- ☐ Saving regularly while editing: When editing my C source file `census.c` (or any other file), I saved my work regularly and repeatedly, every few minutes (as described in the PP#1 specification, page 15, item VI.5).
- ☐ Code writing: frequent saving, compiling and running to test: As I was writing my C source code, I regularly saved my work, then I compiled using `make`, and, if the compile was successful, then I ran the program to test it, again and again and again, after each small change to the code (as described in the PP#1 specification, page 15, items VI.5 and VI.10, and page 17, items VII.2-3 and VII.6, **EXCEPT** ignore the last sentence).
- ☐ Line lengths in output of runs: In my runs of my C source file `census.c`, I verified that every line of output text is less than 80 characters long, the width of my terminal window (as described in this PP#2 specification, page 12, grading criterion #15).
- ☐ Compile (make) your `census`: I successfully compiled my C source file named `census.c`, using the `make census` command (as described in the PP#1 specification, page 17, item VII.2).

CHECKLIST ITEMS FOR SOURCE FILE CONTENTS

- ☐ Comment block: At the very beginning of my C source file, I have a comment block, like the one in `my_number.c` (as described in the PP#2 specification, page 9, grading criterion #1a).
- ☐ Preprocessor directive: Other than comments, my C source file starts with the preprocessor directive
  `#include <stdio.h>`
  (as described in the PP#2 specification, page 5, section IV.A).
- ☐ Main function header: Immediately after the preprocessor directive, my C source file has the main function header
  `int main ()`
  (as described in the PP#2 specification, page 5, section IV.A).
- ☐ Main function block open delimiter: Immediately after the main function header, my C source file has the main function block open delimiter
  {
  (as described in the PP#2 specification, page 5, section IV.A).
- ☐ Main function block close delimiter: At the end of the main function, my C source file has the main function block close delimiter
  }
  (as described in the PP#2 specification, page 5, section IV.A).
- ☐ Main function block delimiter comments: For my main function block delimiters, each is followed, on the same line, by a comment with the name of the function, with a blank space between the function name and each comment delimiter (as described in the PP#2 specification, page 10, grading criterion #2).

- ☐ Declaration section followed by execution section: Inside my main function is the declaration section, followed by the execution section, in that order (as described in the PP#2 specification, page 5, section IV.A, and page 10, grading criterion #3).
- ☐ Declaration section comment: Immediately before my declaration section is a comment labeling it as the declaration section, as in `my_number.c` (as described in the PP#2 specification, page 9, grading criterion #1b).
- ☐ Variable comments: My variable declarations are preceded by comments that describe the nature and purpose of each variable, as in `my_number.c` (as described in the PP#2 specification, page 9, grading criterion #1c).
- ☐ Identifier structure: In my declaration section, my identifiers (in this case, variable names) have the following structure: adjacent words are separated by underscores, and all letters are lower case, except that proper nouns, if any, are capitalized (as described in the PP#2 specification, page 10, grading criterion #4).
- ☐ Favorite professor rule: In my declaration section, my identifiers (in this case, variable names) all comply with the "favorite professor" rule (as described in the PP#2 specification, page 10, grading criterion #5).
- ☐ Variable data types: In my declaration section, my variable names have appropriate data types (as described in the PP#2 specification, page 10, grading criterion #6).
- ☐ Declaration order: In my declaration section, my declaration statements have all `float` declarations followed by all `int` declarations (as described in the PP#2 specification, page 5, section IV.B, and page 10, grading criterion #7).
- ☐ Declaration structure: In my declaration section, my declaration statements follow the structure rules (as described in the PP#2 specification, page 10, grading criterion #8).
- ☐ Declaration spacing: In my declaration section, my declaration statements follow the spacing rules (as described in the PP#2 specification, page 10, grading criterion #9).
- ☐ Declaration statements with multiple variables: In my declaration section, any of my declaration statements that declare multiple variables follow the multiple variable declaration statement rules (as described in the PP#2 specification, page 11, grading criterion #10).
- ☐ Execution order: In my execution section, I have the following subsections, in the following order: greeting subsection, input subsection, output subsection, with no mixing of subsections (as described in the PP#2 specification, page 6, section IV.C, and page 11, grading criterion #12).
- ☐ Subsection comments: In my execution section, my subsections (greeting, input, output) are preceded by comments, as in `my_number.c` (as described in the PP#2 specification, page 9, grading criterion #1d).
- ☐ Statement comments: In my execution section, **EVERY SINGLE EXECUTABLE STATEMENT** is preceded by a helpful comment that describes what that statement does, except that, if I have multiple `printf` statements in a row that output a single message, then only the first of those `printf` statements in a row needs to be preceded by a comment (as described in the PP#2 specification, page 9, grading criterion #1e).
- ☐ Statement comment format: In my execution section, every comment preceding **EVERY SINGLE EXECUTABLE STATEMENT** has the appropriate format (as described in the PP#2 specification, page 9, grading criterion #1f).

- ☐ No mixing of subsections: In my execution section, my greeting subsection has no inputs, my input subsection has no outputs except prompts for inputs, and my output subsection has no inputs — that is, my execution section does output, prompt input, prompt input, prompt input, output output output, **NOT** output, prompt input output, prompt input output, prompt input output, which would be **WRONG** (as described in the PP#2 specification, page 6, section IV.C, and page 11, grading criterion #13).
- ☐ Indentation: In both my declaration section and my execution section, indentation follows the rules, specifically four spaces but **NO TABS** (as described in the PP#2 specification, page 6, section IV.C, and page 9, grading criterion #11).
- ☐ Line lengths in C source code: In both my declaration section and my execution section, **EVERY** line of C source code text is less than 80 characters long — that is, less than the width of the PuTTY or terminal window (as described in the PP#2 specification, page 12, grading criterion #14).
- ☐ Line lengths in output: In my execution section, **EVERY** line of C output text is less than 80 characters long — that is, less than the width of the PuTTY or terminal window (as described in the PP#2 specification, page 12, grading criterion #15).
- ☐ `printf` statements without placeholders: In my execution section, **EVERY** `printf` statement that doesn't have placeholders has the correct format (as described in the PP#2 specification, page 12, grading criterion #16).
- ☐ `printf` statements with placeholders: In my execution section, **EVERY** `printf` statement that does have placeholders has the correct format (as described in the PP#2 specification, page 12, grading criterion #17).
- ☐ Newlines in `printf` statements: In my execution section, **EVERY** `printf` statement that does have a newline `\n` has that newline at the **END** of the string literal (as described in the PP#2 specification, page 12, grading criterion #18).
- ☐ `scanf` statements: In my execution section, **EVERY** `scanf` statement has the correct format (as described in the PP#2 specification, page 12, grading criterion #19).
- ☐ `scanf` statement ampersands: In my execution section, **EVERY** `scanf` statement has an ampersand `&` before **EVERY** variable after the `scanf` statement's string literal (as described in the PP#2 specification, page 12, grading criterion #19).
- ☐ **NO** newlines in `scanf` statements: In my execution section, **NO** `scanf` statement has a newline `\n` anywhere in its string literal (as described in the PP#2 specification, page 12, grading criterion #20).
- ☐ String literals on a single line: In my execution section, **EVERY** `printf` statement has its string literal on a single line, with **NO** carriage returns inside the string literals, but may have the newline token `\n` inside the string literal (as described in the PP#2 specification, page 13, grading criterion #21).
- ☐ Outputs are complete sentences: In my execution section, all outputs, including the greeting, all prompts, and all outputs of variable values, are complete English sentences (as described in the PP#2 specification, page 6, section IV.C).

CHECKLIST ITEMS FOR SCRIPTING

- ☐ Runs before scripting: Before starting my scripting session, I thoroughly tested and debugged my code by running it at least three times, with at least three different sets of appropriate inputs, the first run using the required first set of inputs, 12.25 cell phone calls, 8.25 photos and 12345-6789 (as described in the PP#2 specification, page 8, section VI).
- ☐ Start script session: I successfully started my scripting session, using the correct `script` command, with the correct filename, which for PP#2 is `pp2.txt` (*small-P small-P two period small-T small-X small-T*),
  `script pp2.txt`
  (as described in the PP#2 specification, page 8, section VII, paragraph 3, and the PP#1 specification, page 18, item VIII.2).
- ☐ Script session `pwd`: In my scripting session, I properly did the
  `pwd`
  command (as described in the PP#1 specification, page 18, item VIII.3).
- ☐ Script session `ls -l`: In my scripting session, I properly did the
  `ls -l`
  command (*small-L small-S space hyphen small-L*, **NOT** *small-L small-S space hyphen one*, which would be **WRONG**) (as described in the PP#1 specification, page 18, item VIII.4).
- ☐ Script session `cat makefile`: In my scripting session, I properly did the
  `cat makefile`
  command (as described in the PP#1 specification, page 18, item VIII.5).
- ☐ Script session `cat census.c`: In my scripting session, I properly did the
  `cat census.c`
  command (as described in the PP#1 specification, page 18, item VIII.6, except with the C source file name for PP#2).
- ☐ Script session `make clean`: In my scripting session, I properly did the
  `make clean`
  command (as described in the PP#1 specification, page 18, item VIII.7).
- ☐ Script session `make census`: In my scripting session, I properly did the
  `make census`
  command (as described in the PP#1 specification, page 19, item VIII.8, except with the C source file name for PP#2).
- ☐ Script runs: In my scripting session, I did the correct number of runs, in the correct order, with appropriate values (as described in the PP#2 specification, page 8, section VI).
- ☐ Script session termination: In my scripting session, after completing the appropriate commands, I terminated the scripting session using
  Ctrl-**D**
  (as described in the PP#1 specification, page 19, item VIII.10).
- ☐ Script file cleanup with `dos2unix`: After my scripting session, I cleaned up my script file `pp2.txt` using the
  `dos2unix pp2.txt`
  command (as described in the PP#1 specification, page 19, item VIII.12).

- ☐ Script file unedited: After cleaning up my script file `pp2.txt` using the `dos2unix` command, I **NEVER** edited or altered my script file `pp2.txt` (as described in the PP#1 specification, page 19, item VIII.13), though I did replace it with a new one if needed.
- ☐ Script file proofread: I carefully proofread my script file `pp2.txt` (as described in the PP#1 specification, page 20, item VIII.15).

CHECKLIST ITEMS FOR SUBMISSION

- ☐ Summary essay: In my summary essay, I verified that I have included all of the required sections and information, in the correct order (as described in the PP#1 specification, page 24, item X.1.b).
- ☐ Summary essay references section: In my summary essay, I verified that I have included a references section, even if I have no references (as described in the PP#1 specification, page 24, item X.1.b.vi).
- ☐ Summary essay long enough: My summary essay is at least a half page single spaced/full page double spaced (as described in the PP#1 specification, page 24, item X.1.b).
- ☐ Proofreading: Before submitting, I thoroughly **PROOFREAD** every part of my submission: my summary essay, my `read_list` script, and my `census` script.
- ☐ Download to PC: I downloaded, to the PC that I wanted to upload to Canvas from, **ALL OF** my **EXAMPLE SCRIPT** file `pp2_example.txt`, my **C SOURCE** file `census.c` **AND** my **SCRIPT** file `pp2.txt` **BUT NOT** my executable file nor any other file (as described in the PP#2 specification, page 8, section VII, next-to-final paragraph).
- ☐ Upload to Canvas: I uploaded, to the Canvas dropbox for PP#2, **ALL OF** my **EXAMPLE SCRIPT** file `pp2_example.txt`, my **C SOURCE** file `census.c` **AND** my **SCRIPT** file `pp2.txt` **BUT NOT** my executable file nor any other file (as described in the PP#2 specification, page 8, section VII, final paragraph).
- ☐ Upload verification: I verified that I uploaded the correct files — and only the correct files — to the Canvas dropbox for PP#2.
- ☐ Files **NEVER** deleted: For the entire semester, I **NEVER** will delete my C source files nor my script files, even after this programming project is graded.