CS 1313 010: Programming for Non-Majors, Spring 2025
Programming Project #3: Two Little Calculations
Due by Wednesday February 26 2025 9:50am Central Time

This third assignment will give you experience writing programs that involve arithmetic expressions. You will write two short programs. Each program will greet the user, then prompt for and input data from the user, then perform one or more calculations, and then output the result(s) to the user. Therefore, each program body will have a greeting subsection, an input subsection, a calculation subsection, and an output subsection. This project will use the same development process as in Programming Project #2, and will be subject to the same rules and grading criteria, plus some additional criteria. **YOU ARE EXPECTED TO KNOW HOW TO DO MANY OF THESE TASKS WITHOUT HAVING THEM DESCRIBED IN DETAIL.**

The two programs will involve: converting measurements from English to metric units; calculating statistics. Put each of the two programs in a separate C source file; you **MUST** name them:

`conversions.c`   `statistics.c`

NOTE: Except where and as explicitly permitted in writing (for example, in a PP specification), 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, 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 such copying, so **DON'T EVEN THINK ABOUT IT!**

I. WHAT TO DO FIRST

Using the same method as in the PP#2 specification, page 2, section II:

At the top of your makefile, add entries that look like these:

```
conversions:    conversions.c
          gcc  -o  conversions  conversions.c  -lm

statistics:     statistics.c
          gcc  -o  statistics  statistics.c  -lm

circlecalc:     circlecalc.c
          gcc -o circlecalc circlecalc.c -lm
```

(Note the `-lm` which is to say *hyphen small-L small-M*, at the end of each `gcc` command.)
DON'T DELETE PREVIOUS makefile ENTRIES!

ALSO, you **MUST** put new `rm` commands in the `clean` entry at the bottom of your makefile.

II. WHAT TO DO SECOND

Using the same method as in the PP#2 specification, pages 3-4, section III, using the example program `circlecalc.c` in "Arithmetic Expressions Lesson 1," slide #8:

1. Type in, compile and run that program, using the input value on that slide.
2. Comment that program.
3. Compile and run the commented version of the program, using the input value on that slide.
4. Create a script file named
   `pp3_example.txt`
   in the usual way, from the commented version of the program, using the input value on that slide.

IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!
This programming project specification contains some small code examples. Some of these code examples will be extremely useful in your actual PP#3. **WE URGE YOU TO USE THEM.**

NOTE: Except where and as explicitly permitted in writing (for example, in a Programming Project 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, 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 such copying, so **DON'T EVEN THINK ABOUT IT!**

III. CODE DEVELOPMENT PROCESS

The process for developing these programs will be the same as described in the PP#2 specification, on page 7 in Section V, titled "Advice on How to Write a Program," except that (a) you will do calculations, and (b) you will output the values of different variables than you input into.

Pay close attention to the last numbered list on that page. The only difference between the task list for PP#2 and the process that you will use for PP#3 will be that the two programs in PP#3 will have calculations (the program in PP#2 didn't), and also which of the variables will be output.

IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT IMPORTANT!!!
For each program in PP#3, you should follow the directions in the PP#2 specification section V **EXACTLY**, ignoring the calculation subsection until you have completed the rest of the program. (At this stage, some of the outputs in the output subsection will be garbage.) Once everything except the calculation subsection is written and seems to be working properly, you should then write the calculation subsection. **NOTE THAT YOU WILL DEVELOP EACH PROGRAM OUT OF ORDER, CREATING THE CALCULATION SUBSECTION LAST, EVEN THOUGH IT IS LOCATED IN THE MIDDLE OF THE PROGRAM BODY.**

On the following pages are the specifications of the two programs that you will write.

IV.A. CONVERSIONS

According to the *Mars Climate Orbiter Mishap Investigation Board Phase I Report*, Executive Summary, page 6* (Nov 10 1999),

> ... The MCO ... was lost sometime following the spacecraft's entry into Mars occultation .... [T]he root cause for the loss ... was the failure to use metric units in the coding of ... software ... used in trajectory models. Specifically, thruster performance data in English units instead of metric units was used in the software application code titled SM_FORCES (small forces). A file called Angular Momentum Desaturation (AMD) contained the output data from the SM_FORCES software. The data in the AMD file was required to be in metric units ... and the trajectory modelers assumed the data was provided in metric units per the requirements. ...

Write a program to convert from English units to metric units,† specifically:

- to convert volume from US fluid ounces to milliliters,
  AND
- to convert energy from US gallons of gasoline to kilowatt hours.

For your conversions, you **MUST** use the following constant values **AND NO OTHERS**, declaring and initializing appropriate named constants (you are **ABSOLUTELY FORBIDDEN** to combine these in initializations):

- There are 8 US fluid ounces per US cup.
- There are 4 US cups per US quart.
- There are 1.05669 US quarts per liter.
- There are 1000 milliliters per liter.
- There are 114,000 British Thermal Units (BTUs) per US gallon of gasoline.
- There are 1055.056 joules per BTU.
- There is 1 watt-second per joule.
- There are 1000 watts per kilowatt.
- There are 60 seconds per minute.
- There are 60 minutes per hour.

Figuring out these calculations will require *dimensional analysis*.‡

*https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf

†http://www.calculateme.com/, https://www.unitconverters.net/

‡https://www.albert.io/blog/how-to-perform-dimensional-analysis/
https://www.chem.tamu.edu/class/fyp/mathrev/mr-da.html
https://www.math.net/dimensional-analysis

The program body **MUST** incorporate the following subsections, **IN THE FOLLOWING ORDER**:

1. **Greeting Subsection:** Greet the user with useful information about the program.
2. **Input Subsection**
   (a) Prompt the user for a volume in US fluid ounces.
   (b) Input the volume in US fluid ounces.
   (c) Prompt the user for an energy in US gallons of gasoline.
   (d) Input the energy in US gallons of gasoline.
3. **Calculation Subsection**
   (a) Calculate the volume in milliliters.
   (b) Calculate the energy in kilowatt hours.
4. **Output Subsection**
   (a) Output the volume in both US fluid ounces and milliliters.
   (b) Output the energy in both US gallons of gasoline and kilowatt hours.

IMPORTANT: Volumes and energies **AREN'T** constrained to be integers.

RUNS: Run this program three times using three different sets of input values. The first run **MUST** use 32 US fluid ounces and 10 US gallons of gasoline as input values. For the other two runs, you may choose **APPROPRIATE** values to your liking, but each such input value **MUST** differ from the same input in the other runs and from the other input in the same run.

IV.B. STATISTICS

Consider a list of n real numbers:

$$x_1, x_2, \ldots, x_n$$

The *power mean*[§] of the values in the list, here denoted M_p for some real number p, is a real number such that

$$M_p(x_1,\ x_2,\ \ldots,\ x_n) \ = \ \left(\frac{\sum_{i=1}^{n} x_i^p}{n}\right)^{1/p} \ = \ \left(\frac{x_1^p \ + \ x_2^p \ + \ \ldots \ + \ x_n^p}{n}\right)^{1/p}$$

Note: $\sum_{i=1}^{n} z_i$ is known as *summation notation*: $\sum_{i=1}^{n} z_i \ = \ z_1 + z_2 + \ldots + z_n$

Example #1: The *arithmetic mean*,[¶] also known simply as the *mean*, which is the power mean with p of 1, is a real number that is an *average*; that is, a value that is typical of the values in the list. The arithmetic mean, here denoted $\overline{x}$ (pronounced "x-bar"), is calculated as the sum of all the values in the list, divided by the number of values in the list:

$$\overline{x} \ = \ M_1(x_1,\ x_2,\ \ldots,\ x_n) \ = \ \frac{\sum_{i=1}^{n} x_i}{n} \ = \ \frac{x_1 \ + \ x_2 \ + \ \ldots \ + \ x_n}{n}$$

Example #2: The *geometric mean*,[‖], which is the power mean with p of 0, is calculated as the product of all the values in the list, taken to the power of one over the number of values in the list:

$$G(x_1,\ x_2,\ \ldots,\ x_n) \ = \ M_0(x_1,\ x_2,\ \ldots,\ x_n) \ = \ \left(\prod_{i=1}^{n} x_i\right)^{1/n} \ = \ (x_1 \ \cdot \ x_2 \ \cdot \ \ldots \ \cdot \ x_n)^{1/n}$$

Note: $\prod_{i=1}^{n} z_i$ is known as *product notation*: $\prod_{i=1}^{n} z_i \ = \ z_1 \cdot z_2 \cdot \ldots \cdot z_n$

Example #3: The *root mean square*[**], denoted $R(x)$, is the power mean with p of 2:

$$R(x_1,\ x_2,\ \ldots,\ x_n) \ = \ M_2(x_1,\ x_2,\ \ldots,\ x_n) \ = \ \left(\frac{\sum_{i=1}^{n} x_i^2}{n}\right)^{\frac{1}{2}} \ = \ \sqrt{\frac{x_1^2 \ + \ x_2^2 \ + \ \ldots \ + \ x_n^2}{n}}$$

Example #4: The *harmonic mean*[††], denoted $H(x)$, is the power mean with p of -1:

$$H(x_1,\ x_2,\ \ldots,\ x_n) \ = \ M_{-1}(x_1,\ x_2,\ \ldots,\ x_n) \ = \ \left(\frac{\sum_{i=1}^{n} x_i^{-1}}{n}\right)^{-1} \ = \ \frac{n}{\frac{1}{x_1} \ + \ \frac{1}{x_2} \ + \ \ldots \ + \ \frac{1}{x_n}}$$

[§] http://mathworld.wolfram.com/PowerMean.html
[¶] http://mathworld.wolfram.com/Mean.html
[‖] http://mathworld.wolfram.com/GeometricMean.html
[**] http://mathworld.wolfram.com/Root-Mean-Square.html
[††] http://mathworld.wolfram.com/HarmonicMean.html

Write a program to calculate the statistics described above, for a list of 4 values.
The program **MUST** incorporate the following subsections, in the following order:

1. **Greeting Subsection:** Greet the user with useful information about the program.
2. **Input Subsection**
   (a) Prompt the user to input 4 values.
   (b) Input the 4 values, **using a single `scanf` statement.**
3. **Calculation Subsection**
   (a) Calculate the arithmetic mean of the 4 values.
   (b) Calculate the geometric mean of the 4 values.
   (c) Calculate the root mean square of the 4 values.
   (d) Calculate the harmonic mean of the 4 values.
4. **Output Subsection**
   (a) Output the 4 values.
   (b) Output their arithmetic mean.
   (c) Output their geometric mean.
   (d) Output their root mean square.
   (e) Output their harmonic mean.

You may use the C math library functions `sqrt` and `pow` for square root and raising to a power, respectively.

To use either or both of them, you **MUST first** do this: immediately after the usual preprocessor directive

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

you **MUST** have another preprocessor directive:

```
#include <math.h>
```

Then, to use the math library function `sqrt`, do this:

```
y = sqrt(x);
```

for some variables `x` and `y`.

Of course, in your program, the variables will have different names than these; in fact, instead of a variable inside `sqrt`, there might be an expression.

Note that the equivalent in mathematics is

$$y = \sqrt{x}$$

Similarly, to use the math library function `pow`, do this:

```
z = pow(x, y);
```

for some variables `x`, `y` and `z`.

Of course, in your program, the variables will have different names than these, and one or both of the terms inside the `pow` function might be named constants rather than variables, or might be expressions.

Note that the equivalent in mathematics is

$$z = x^y$$

HINT for calculating the square (**NOT** the square **root**):

What is the definition of the square of a number? You can calculate the square that way.

Finally, the compile command in your makefile entry for the program **MUST** end with

`-lm`

(that is, *hyphen small-L small-M*, **NOT** *hyphen one small-M*), as shown in the makefile entries at the beginning of this document.

NOTE: You may find it helpful to have extra variables for partial results (for example, for the individual reciprocals in the harmonic mean).

NOTE: You may **NOT** use `x`, `x_bar`, `G`, `R`, `H`, etc., as variable names, because they would violate the "favorite professor" rule.

IMPORTANT: Statistics are almost always non-integers.

RUNS: Run this program three times using three different sets of input values.
The first run **MUST** use the following input values:
123.75 456.50 789.25 101.00
For the other two runs, you may choose **APPROPRIATE** values to your liking.

V. ADDITIONAL GRADING CRITERIA

NOTE: **ALL** grading criteria introduced in any Programming Project apply to **ALL** future PPs, unless explicitly stated otherwise.

A. Additional Grading Criteria for C Source Code

1. **Declaration subsections:** Within the declaration section, there **MUST** be a subsection of named constant declarations, followed by a subsection of variable declarations. These two declaration subsections **MUST** be clearly labeled by comments, as shown in `my_number.c`.
2. **Declaration subsection order:** The named constant declaration subsection **MUST** appear **BEFORE** the variable declaration subsection, and therefore **ALL** named constant declarations **MUST** appear before **ANY** variable declarations, as shown in `my_number.c`.
3. **Named constant and variable declaration order:** **ALL** `float` named constants **MUST** be declared before **ANY** `int` named constants. Likewise, **ALL** `float` variables **MUST** be declared before **ANY** `int` variables.
4. **Declaration comments:** Named constant and variable declarations **MUST** be preceded by comments clearly explaining the nature and purpose of each declared name, as shown in `my_number.c`.
5. **No mixing of sections and subsections:** You are **ABSOLUTELY FORBIDDEN** to have:
   (a) **ANY** executable statements in your declaration section;
   (b) **ANY** declaration statements in your program body;
   (c) **ANY** inputs or calculations in your greeting subsection;
   (d) **ANY** calculations, or outputs other than prompts, in your input subsection;
   (e) **ANY** inputs or outputs in your calculation subsection;
   (f) **ANY** inputs or calculations in your output subsection.
6. **Numeric literal constants** are **ABSOLUTELY FORBIDDEN** in a program's execution section (body). (They are permitted in the declaration section when initializing variables and named constants.) All numeric constants used in the program body **MUST** be named constants. **There are NO EXCEPTIONS to this rule.**
7. **Numeric literal constants embedded inside string literals** are also **ABSOLUTELY FORBIDDEN** in the program body; for example, the statement below is **NOT** acceptable:
   ```
   printf("This is the year 2025.\n"); /* <-- BAD BAD BAD! */
   ```
   The only exception to this rule is the use of numeric literal constants in placeholder format descriptors.
8. **Constant names**, like variable names, **MUST** be meaningful, and **MUST** satisfy the "favorite professor" rule.
9. **Constant names that reflect the value of the constant**, rather than its purpose, are **ABSOLUTELY FORBIDDEN** (for example, `one` and `six` are **NOT ACCEPTABLE** as constant names).
10. **Assignment statements** **MUST** have the following format: indentation, followed by the name of the variable whose value is being assigned, followed by one or more blank spaces (usually just one), followed by a single equals sign, followed by one or more blank spaces (usually just one), followed by the expression to calculate the variable's value, followed by the statement terminator.

11. **Expressions in assignment statements** **MUST** have the following format:
    (a) Each operator (for example, `+ - * /`) **MUST** be surrounded on each side by one or more blank spaces.
    (b) An open parenthesis **MUSTN'T** have any blank spaces to its right.
    (c) A close parenthesis **MUSTN'T** have any blank spaces to its left.
    (d) If an expression requires multiple lines of source code text, then each line (other than the last) **MUST** end with an operator (or the single equals sign), and corresponding parts of the expression **MUST** line up. For example:

```
volume_in_milliliters =
    volume_in_US_fluid_ounces   /
    (US_fluid_ounces_per_US_cup *
     US_cups_per_US_quart       *
     US_quarts_per_liter)       *
    milliliters_per_liter;
```

WE URGE YOU TO USE THE CODE JUST ABOVE IN YOUR PP#3!

12. For PP#3, you are allowed to use **ONLY** the following kinds of programming constructs in your C source file, but **NO OTHER KINDS OF PROGRAMMING CONSTRUCTS**:
    - all programming constructs allowed in PP#2;
    - named constant declarations;
    - assignment statements that have arithmetic expressions on the right hand side of the single equals sign.

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

B. Additional Grading Criteria for Summary Essays

You will need to write **TWO SUMMARY ESSAYS,** one for **EACH** of the two new programs. Together, they will be worth at least 10% of the project's total value, and each **MUST** cover the points listed in the specification for Programming Project #1, page 24, section X, item 1(a)i-vi. For this project, each of the two summary essays **MUST** be at least half a page single spaced or a full page double spaced, in a 10 to 12 point font, with margins of 1 inch on each side.

VI. SCRIPTS

Before creating either of your two script files, thoroughly test and debug both of your programs. Be sure to test them with the input values that you will be required to use in your script files.

To ensure that your programs are producing the correct results, calculate the correct results by hand, and compare your hand-calculated values to the associated program output.

As you develop your programs, you will compile, run, test and then script each of these programs separately, using the scripting process described in Programming Project #1. You will create two separate script files, one for each of the two programs. **You are ABSOLUTELY FORBIDDEN to use a single script file for both programs.** The script files **MUST** be named:

`pp3_conversions.txt` `pp3_statistics.txt`

VII. WHAT TO SUBMIT

You will need to **UPLOAD** the example script file, both summary essays (for conversions and statistics), both C source files and both script files to the Canvas dropbox for PP#3.

For this project, you are not required to include idiotproofing checks on the input, because we have not yet learned `if` statements. Future programming projects will include idiotproofing.

It is **YOUR** responsibility to read and comply with all of the grading criteria listed for Programming Projects #1 and #2, as well as the additional criteria for this project.

PP#3 CHECKLIST

- [ ] Edit `makefile`: I edited my `makefile` to add make entries for `conversions`, `statistics` and the example program, and to update the `clean` entry for all of those programs (as described in the PP#3 specification, page 1, section I).
- [ ] Example program: I typed in, compiled and ran the example program (as described in the PP#3 specification, page 2, section II).
- [ ] Example program commented: I commented, then recompiled and reran the example program (as described in the PP#3 specification, page 2, section II).
- [ ] Example program scripted: I scripted the example program (as described in the PP#3 specification, page 2, section II).
- [ ] Code development process: I used the recommended code development process (as described in the PP#3 specification, page 2, section III).

FOR EACH OF MY NEW PROGRAMS:

- [ ] Declaration subsections: My declaration section has two subsections, one for named constants, the other for variables (as described in the PP#3 specification, page 8, grading criterion #1).
- [ ] Declaration subsection labeling comments: Before my named constant subsection and before my variable subsection, there are comments labeling those subsections (as described in the PP#3 specification, page 8, grading criterion #1).
- [ ] Declaration subsection order: My named constant declaration subsection comes before my variable declaration subsection, and therefore all my named constant declarations come before any of my variable declarations (as described in the PP#3 specification, page 8, grading criterion #2).
- [ ] Declaration order within each subsection: In my named constant subsection and in my variable subsection, all `float` declarations come before any `int` declarations (as described in the PP#3 specification, page 8, grading criterion #3).
- [ ] Declaraction comments: In my named constant subsection and in my variable subsection, there are comments clearly explaining each named constant or variable (as described in the PP#3 specification, page 8, grading criterion #4).
- [ ] No mixing of sections or subsections: In my program, there are no declarations in my execution section (body), no inputs or calculations in my greeting subsection, no calculations in my input subsection (and the only outputs are prompts for inputs), no inputs or outputs in my calculation subsection, and no inputs or calculations in my output subsection (as described in the PP#3 specification, page 8, grading criterion #5).
- [ ] No numeric literal constants in execution section (body): In the execution section (body) of my program, there are no numeric literal constants, only named constants (though I do have numeric literal constants in my declaration section) (as described in the PP#3 specification, page 8, grading criterion #6).
- [ ] No numeric literal constants inside string literals: In the execution section (body) of my program, there are no numeric literal constants embedded inside string literals (as described in the PP#3 specification, page 8, grading criterion #7).
- [ ] Favorite professor rule for named constants: The names of my named constants (and of my variables) conform to the favorite professor rule (as described in the PP#3 specification, page 8, grading criterion #8).

- ☐ No named constants named for their own values: The name of each of my named constants reflects the purpose of that named constant, not the value of that named constant (as described in the PP#3 specification, page 8, grading criterion #9).
- ☐ Assignment statement format: My assignment statements have the correct format (as described in the PP#3 specification, page 8, grading criterion #10).
- ☐ Expression format: My expressions (for example, arithmetic expressions on the right hand side of assignment statements) have the correct format (as described in the PP#3 specification, page 9, grading criterion #11).
- ☐ Types of constructs: My statements and other constructs are **ONLY** of the allowed types, specifically those allowed in PP#2, plus named constant declarations and assignments statements that have arithmetic expressions on the right hand side of the single equals sign (as described in the PP#3 specification, page 9, grading criterion #12).
- ☐ Two summary essays: I have two summary essays, one for each program (as described in the PP#3 specification, page 9, section V.B).
- ☐ Three script files: I have three script files, created in three separate script sessions, one for each of the programs I've written plus one for the example program I've commented (as described in the PP#3 specification, page 9, section VI).
- ☐ Script file names: The script files for my two programs are `pp3_conversions.txt` and `pp3_statistics.txt` (as described in the PP#3 specification, page 9, section VI).
- ☐ Uploads: I have uploaded, to the Canvas PP#3 dropbox, the script file for the example program, as well as, for both of my new programs, the summary essays, the C source files, and the script files (as described in the PP#3 specification, page 9, section VII).

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#3 as follows:

1. Attend at least one regularly scheduled CS1313 help session for at least 30 minutes, through Wed Feb 26.
2. During the regularly scheduled help session that you attend, work on CS1313 assignments (ideally PP#3, 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 17 - Tue Feb 18: 5% of the total value of PP#3;
- for attending a regularly scheduled help session Mon Feb 24 - Wed Feb 26: 2.5% of the total value of PP#3.