Programming Assignment #1: Function Mayhem

COP 3223H, Fall 2019

Due: Wednesday, September 18, *before* 11:59 PM

Table of Contents

Deliverables

assignment01.c

Note! The capitalization and spelling of your filename matter!

Note! Code must be tested on Eustis, but submitted via Webcourses.

1. Super Important: Initial Setup (*Read This Carefully!*)

At the very top of your *assignment01.c* file, write a comment with your name, the course number, the current semester, and your NID. Directly below that, you **must** include the following line of code:

```
#include "assignment01.h"
```

Yes, that needs to be in “double quotes” instead of <angled brackets>, because *assignment01.h* is a local header file that we’ve included with this assignment, rather than a standard system header file.

The *assignment01.h* file we have included with this assignment is a special header file that will enable us to grade your program. If you do not *#include* that file properly, your program will not compile on our end, and it will not receive credit.

In your code, writing *main()* is optional. If you do write *main()* (which is a good idea so you can call and test your other functions) it doesn’t matter too much what it does (as long as it’s not coded up to do anything naughty or malicious), because we won’t actually run your *main()* function. When we grade your program, we will write a script that will seek out your *main()* function, destroy it, and replace it with our own *main()* function for each test case we want to run. This is savage, but effective. If you don’t have that *#include "assignment01.h"* line in your code, our script will be unable to inject our own *main()* functions into your code, and all the test cases we use to grade your program will fail. SAD.

From this point forward, you will always have to have *assignment01.h* (provided in the ZIP file for this assignment) in the same folder as your *assignment01.c* file any time you want to compile your code.

2. Note: Test Case Files Might Look Wonky in Notepad

Included with this assignment are several test cases, along with output files showing exactly what your output should look like when you run those test cases. You will have to refer to those as the gold standard for how your output should be formatted.

Please note that if you open those files in older versions of Notepad, they will appear to contain one long line of text. That’s because Notepad used to handle end-of-line characters differently from Linux and Unix-based systems. One solution is to view those files in a text editor designed for coding, such as Atom, Sublime, or Notepad++. For those using Mac or Linux systems, the input files should look just fine.

3. Assignment Overview

In this assignment, you will write several functions (listed below). Some will be more straightforward than others. They are designed to ramp up in difficulty and challenge your understanding of the material covered in class. Before you start working on this assignment, your best bet is to read and comprehend *all* the notes posted in Webcourses. This assignment will also serve as an introduction to how assignments are submitted and graded in this course. As you know, much of your grade will be based on your program’s ability to produce output that is an *exact* match for the sample output files released with this assignment.

4. Function Requirements

You must implement the following functions in a file named *assignment01.c*. Please be sure the spelling, capitalization, return types, and function parameters match the ones given below. Even the most minor deviation could cause a huge loss of points.

The order in which you write these functions in your file does not matter, as long as it compiles *without any warnings* (or errors, for that matter). Your functions are allowed to call one another, and you can write additional functions ("helper functions") if you find that doing so will make it easier for you to write some of these required functions.

```
int return_five(void)
```

Description: This function should simply return 5.

Output: This function should not print anything to the screen.

Return Value: This function should return 5.

Related Test Case: *testcase01.c*

```
int pancake_printer(void)
```

Description: This function should print "Pancake Breakfast with Professor Patrick!" to the screen (*including* the double quotes!), followed by a newline character. The function should then return 52847.

Output: This function should print a string to the screen, as described above.

Return Value: This function should return 52847.

Related Test Case: *testcase02.c*

```
double get_pancake_count(double minutes, double pancakes_per_minute)
```

Description: This function takes two parameters: the number of minutes someone spends consuming pancakes (*double minutes*) and the average number of pancakes that person consumes per minute during that pancake-eating session (*double pancakes_per_minute*). Based on those inputs, you should return the number of pancakes this person consumed as a double. You may assume that both inputs will be positive (i.e., strictly greater than zero) real numbers. The return value is a double (not just an integer) because this person might consume only part of the last pancake.

Output: This function should not print anything to the screen.

Return Value: This function should return a real number (a double) as described above.

Related Test Case: *testcase03.c*

`double get_pancakes_per_minute(double minutes, double pancake_count)`

Description: This function takes two parameters: the number of minutes someone spends consuming pancakes (*double minutes*) and the number of pancakes they manage to consume (*double pancake_count*). Based on those inputs, you should return the average rate of pancake consumption for that person's pancake-eating session. You may assume that both inputs will be positive (i.e., strictly greater than zero) real numbers.

Output: This function should not print anything to the screen.

Return Value: This function should return a real number (a double) as described above.

Related Test Case: *testcase04.c*

`double get_minutes_spent_munching(double pancake_count, double pancakes_per_minute)`

Description: This function takes two parameters: the number of pancakes someone consumes (*double pancake_count*) and the average rate at which they consumed those pancakes (*double pancakes_per_minute*). Based on those inputs, you should return the number of minutes this person spent eating pancakes. You may assume that both inputs will be positive (i.e., strictly greater than zero) real numbers.

Output: This function should not print anything to the screen.

Return Value: This function should return a real number (a double) as described above.

Related Test Case: *testcase05.c*

`void print_pancake_count(double minutes, double pancakes_per_minute)`

Description: This function has all the same behaviors as *get_pancake_count()*, except it does not return a value; instead, it prints the result to the screen. Your output must include exactly three decimal places of precision and follow the format shown below (without any indentation, and followed by a newline character):

```
Pancakes consumed: 3.112
```

Output: See above for a description of this function's output. Also, see the test cases and sample output included with this assignment for further examples of this function's output. Your output must match our sample output files *exactly*.

Return Value: This is a *void* function and should therefore not return a value.

Related Test Case: *testcase06.c*

`void print_pancakes_per_minute(double minutes, double pancake_count)`

Description: This function has all the same behaviors as *get_pancakes_per_minute()*, except it does not return a value; instead, it prints the result to the screen. Your output must include exactly three decimal places of precision and follow the format shown below (without any indentation, and followed by a

newline character):

Pancakes per minute: 1.442

Output: See above for a description of this function's output. Also, see the test cases and sample output included with this assignment for further examples of this function's output. Your output must match our sample output files *exactly*.

Return Value: This is a *void* function and should therefore not return a value.

Related Test Case: *testcase07.c*

`void print_minutes_spent_munching(double pancake_count, double pancakes_per_minute)`

Description: This function has all the same behaviors as *get_minutes_spent_munching()*, except it does not return a value; instead, it prints the result to the screen. Your output must include exactly three decimal places of precision and follow the format shown below (without any indentation, and followed by a newline character):

Minutes spent munching: 12.000

Output: See above for a description of this function's output. Also, see the test cases and sample output included with this assignment for further examples of this function's output. Your output must match our sample output files *exactly*.

Return Value: This is a void function and should therefore not return a value.

Related Test Case: *testcase08.c*

`double get_pancake_data(double pancake_count, double pancakes_per_minute, double minutes)`

Description: This function takes three parameters. Exactly one of these parameters will be zero; the other two will be positive (i.e., strictly greater than zero) real numbers. From the two non-zero input parameters, calculate and return correct value for the input parameter that was equal to zero.

For example, if someone calls *get_pancake_data(5.0, 2.5, 0.0)*, that means you're given a *pancake_count* and *pancakes_per_minute* rate, and you need to return the number of minutes that person spent eating pancakes (in this case, 2.0).

This will require some math, as well as some if-else logic.

Output: This function should not print anything to the screen.

Return Value: This function should return a real number (a double) as described above.

Related Test Case: *testcase09.c*

`void print_pancake_data(double pancake_count, double pancakes_per_minute, double minutes)`

Description: This function has all the same behaviors as *get_pancake_data()*, except it does not return a value; instead, it prints the result to the screen. Your output must include exactly three decimal places of

precision and follow the format shown in the other functions that print pancake-related results.

For example, if someone calls *print_pancake_data(5.0, 2.5, 0.0)*, that means you're given a *pancake_count* and *pancakes_per_minute* rate, and you need to print the number of minutes that person spent eating pancakes (in this case, 2.000), using the format below (without any indentation, and followed by a newline character):

```
Minutes spent munching: 2.000
```

This will require some math, as well as some if-else logic.

Output: See above for a description of this function's output. Also, see the test cases and sample output included with this assignment for further examples of this function's output. Your output must match our sample output files *exactly*.

Return Value: This is a *void* function and should therefore not return a value.

Related Test Case: *testcase10.c*

`int round_up_or_down(double dub)`

Description: This function takes a non-negative real number as its sole input parameter (*double dub*) and returns the integer closest to that real number. If the decimal portion of *dub* is greater than or equal to 0.5, you should round the integer up. If the decimal portion of *dub* is less than 0.5, you should round down. For example, if we pass 2.0 or 2.1 to this function, it should return 2. If we pass 5.5 or 5.8 to this function, it should return 6.

Note: This function does not have to handle negative values. We will never pass a negative value to *round_up_or_down()* when grading your assignment.

This function will require some clever thought on your part. A solution to this function will likely involve some arithmetic, some integer truncation, and some if-else statements.

Output: This function should not print anything to the screen.

Return Value: This function returns an integer as described above.

Related Test Case: *testcase11.c*

`int get_median_of_three(int a, int b, int c)`

Description: This function takes three integers (*a*, *b*, and *c*) and returns the median of those three integers.

This will require if-else statements.

Output: This function should not print anything to the screen.

Return Value: This function returns an integer as described above.

Related Test Case: *testcase12.c*

`void print_median_of_three(int a, int b, int c)`

Description: This function takes three integers (*a*, *b*, and *c*) and prints the median of those three integers on a line all by itself, followed by a newline character.

This will require if-else statements.

Output: This function should print a single integer to the screen, as described above. Also, see the test cases and sample output included with this assignment for further examples of this function's output. Your output must match our sample output files *exactly*.

Return Value: This is a *void* function and should therefore not return a value.

Related Test Case: *testcase13.c*

`int get_ordinal_day(int month, int day)`

Description: This function takes two parameters that, taken together, represent a date: *month* (an integer from 1 through 12) and *day* (an integer 1 through 31). The function then returns an integer from 1 through 365 indicating which ordinal day of the year the given date corresponds to, assuming we are not dealing with a leap year (so, assuming February has only 28 days).

For example, January 31st is the 31st day of the year, and so if we pass 1 and 31 to the function, it should return 31. Similarly, February 1st is the 32nd day of the year, and so if we pass 2 and 1 to the function, it should return 32. December 31st is the 365th day of the year, and so if we pass 12 and 31 to the function, it should return 365.

This function will require some if-else logic and some math.

Special Restriction: This function cannot be longer than 75 lines of code. (That way, you cannot just hard-code return values for all 365 days of the year.)

Output: This function should not print anything to the screen.

Return Value: This function returns an integer as described above.

Related Test Case: *testcase14.c*

`int print_ordinal_day(int month, int day)`

Description: This function has all the same behaviors as the *get_ordinal_day()* function, except that it also prints a formatted string to the screen indicating what month and day were passed into the function, as well as which ordinal day of the year that corresponds to. The integers printed in this function must have "st," "nd," "rd," or "th" at the end, as appropriate. *Spelling, capitalization, and punctuation matter!* For example:

> January 3rd is the 3rd day of the year.
>
> February 1st is the 32nd day of the year.
>
> December 31st is the 365th day of the year.

As with the *get_ordinal_day()* function, this function assumes that we are ***not*** dealing with a leap year, and it returns an integer from 1 through 365 indicating which ordinal day of the year the given date corresponds to.

This function will require some if-else logic and some math.

Special Restriction: You cannot just hard-code results for all 365 days of the year.

Output: See above for a description of this function's output. Also, see the test cases and sample output included with this assignment for further examples of this function's output. Your output must match our sample output files *exactly*.

Return Value: This function returns an integer as described above for *get_ordinal_day()*.

Related Test Case: *testcase15.c*

```
int get_ordinal_day_extended(int month, int day, int leapyear)
```

Description: This function takes three parameters. The first two parameters, taken together, represent a date: *month* (an integer from 1 through 12) and *day* (an integer 1 through 31). The third parameter, *leapyear*, indicates whether or not we are computing the result for a leap year. If *leapyear* is 0, that means we are ***not*** working with a leap year (in which case February only has 28 days, and there are only 365 days in the year). If *leapyear* is 1, that means we ***are*** working with a leap year (in which case February has 29 days, and the year has 366 days in total).

As with the *get_ordinal_day()* function, this function then returns an integer (from 1 through 365 in the case of non-leap years, or 1 through 366 in the case of leap years) indicating which ordinal day of the year the given date corresponds to.

This function will require some if-else logic and some math.

Special Restriction: This function cannot be longer than 175 lines of code. (That way, you cannot just hard-code return values for all 365 (or 366) days of the year.)

Output: This function should not print anything to the screen.

Return Value: This function returns an integer as described above.

Related Test Case: *testcase16.c*

```
int get_ordinal_day_with_error_checking(int month, int day, int leapyear)
```

Description: This function has all the same behaviors as the *get_ordinal_day_extended()* function, except that it also checks whether the integers passed into the function correspond to a valid day of the year. If so, the function returns the ordinal integer that corresponds to that day. If the inputs are ***not*** valid, the function must return -1.

For example, if we pass 13 and 2 to the function for the *month* and *day*, respectively, the function should return -1, because 13 does not correspond to a valid month in our calendar year. Similarly, the function should return -1 if we pass -545 and 2 to the function for *month* and *day*, since -545 does not correspond

to a valid month in our calendar year.

Furthermore, if we pass 2 and 31 to the function for *month* and *day*, it should return -1 since there is no 31st day of February. If we pass 2 and 29 to the function, it should return -1 if *leapyear* is 0, but it should return 60 if *leapyear* is 1, since February 29th is not a real day in a non-leap year, but it is the 60th day of the year during a leap year.

This function will require some if-else logic and some math.

Special Restriction: This function cannot be longer than 250 lines of code. You cannot just hard-code results for all 365 (or 366) days of the year.

Output: This function should not print anything to the screen.

Return Value: This function returns an integer as described above.

Related Test Case: *testcase17.c*

`double difficulty_rating(void)`

Description: The body of this function should contain a single line of code that returns a value indicating how difficult you found this assignment on a scale from 1.0 (ridiculously easy) through 5.0 (insanely difficult).

Output: This function should not print anything to the screen.

Return Value: This function returns a real number (a double) as described above.

Related Test Case: *testcase18.c*

`double hours_invested(void)`

Description: The body of this function should contain a single line of code that returns a value (greater than zero) that is an estimate of the number of hours you invested in this assignment. Your return value must be a realistic and reasonable estimate. Unreasonably large values will result in loss of credit for this particular function.

Output: This function should not print anything to the screen.

Return Value: This function returns a real number (a double) as described above.

Related Test Case: *testcase19.c*

`double prior_experience(void)`

Description: The body of this function should contain a single line of code that returns a value indicating how much prior programming experience you had coming into this course on a scale from 1.0 (never programmed before) through 5.0 (seasoned veteran who has worked in industry as a programmer). Someone who has been programming for 1-2 years and has taken at least one formal programming course (whether in high school or at the university level) will probably want to return

somewhere around a 3.0 for this function.

Output: This function should not print anything to the screen.

Return Value: This function returns a real number (a double) as described above.

Related Test Case: *testcase20.c*

5. Special Restrictions (*Important!*)

You must abide by the following restrictions in the *assignment01.c* file you submit. Most of these restrictions will only make sense if you've been programming for a while. If you're new to C, you probably wouldn't think to do any of the crazy things that would violate these restrictions:

- ★ You cannot call any of C's built-in functions except for *printf()*.
- ★ Do not declare new variables part way through a function. All variable declarations must occur at the *top* of a function.
- ★ All variables must be declared inside your functions or declared as function parameters.
- ★ Do not make calls to C's *system()* function.
- ★ Do not use *goto* statements in your code.
- ★ Do not read or write to files (using, e.g., C's *fopen()*, *fprintf()*, or *fscanf()* functions).
- ★ Do not write malicious code. (I would hope this would go without saying.)
- ★ No crazy shenanigans.

Failure to abide by any one of these restrictions could result in a catastrophic loss of points.

6. Style Restrictions (*Important!*)

Please conform as closely as possible to the style I use while coding in class. In particular:

- ★ Any time you open a curly brace, that curly brace should start on a new line, and it should be indented to align properly with the line above it. See my code in Webcourses for examples.
- ★ Any time you open a new code block, indent all the code within that code block one level deeper than you were already indenting.
- ★ Be consistent with the amount of indentation you're using, and be consistent in using either spaces or tabs for indentation throughout your source file. If you're using spaces for indentation, please use at least two spaces for each new level of indentation, because trying to read code that uses just a single space for each level of indentation is downright painful.

- ★ Please always use code blocks with if/else statements and loops, even if there's just one line of code within that code block.
- ★ Please avoid block-style comments: /* *comment* */
- ★ Instead, please use inline-style comments: // *comment*
- ★ Always include a space after the "//" in your comments: "// *comment*" instead of "//*comment*"
- ★ Any libraries you *#include* should be listed *after* the header comment at the top of your file that includes your name, course number, semester, NID, and so on.
- ★ Use end-of-line comments sparingly. Comments longer than three words should always be placed above the lines of code to which they refer. Furthermore, such comments should be indented to properly align with the code to which they refer. For example, if line 16 of your code is indented with two tabs, and line 15 contains a comment referring to line 16, then line 15 should also be intended with two tabs.
- ★ Please do not write overly long lines of code. Lines must be fewer than 100 characters wide.
- ★ Avoid excessive consecutive blank lines. In general, you should never have more than one or two consecutive blank lines.
- ★ Please leave a space on both sides of any arithmetic operator you use in your code. For example, use *(a + b) - c* instead of *(a+b)-c*.
- ★ When defining a function that doesn't take any arguments, put *void* in its parentheses. For example, define a function using *int do_something(void)* instead of *int do_something()*.

7. Running All Test Cases on Eustis

The test cases included with this assignment are designed to show you some ways in which we might test your code and to shed light on the expected functionality of your code. We've also included a script, *test-all.sh*, that will compile and run all test cases for you.

Super Important: Using the *test-all.sh* script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting.

To run the *test-all.sh* script on Eustis, first transfer it to Eustis along with with *assignment01.c*, *assignment01.h*, all the test case files, and the *sample_output* directory. Transferring all your files to Eustis with MobaXTerm isn't too hard, but if you want to transfer them from a Linux or Mac command line, here's how you do it:

1. At your command line on your own system, use *cd* to go to the folder that contains all your files for this project (*assignment01.c*, *assignment01.h*, *test-all.sh*, the test case files, and the *sample_output* folder).
2. From that directory, type the following command (replacing `YOUR_NID` with your actual NID) to transfer that whole folder to Eustis:

```
scp -r $(pwd) YOUR_NID@eustis.eecs.ucf.edu:~
```

> **Warning:** Note that the `$(pwd)` in the command above refers to your current directory when you're at the command line in Linux or Mac OS. The command above transfers the *entire contents* of your current directory to Eustis. That will include all subdirectories, so for the love of all that is good, please don't run that command from your desktop folder if you have a ton of files on your desktop!

Once you have all your files on Eustis, you can run the script by connecting to Eustis and typing the following:

```
bash test-all.sh
```

If you put those files in their own folder on Eustis, you will first have to *cd* into that directory. For example:

```
cd assignment01
```

That command (*bash test-all.sh*) will also work on Linux systems and with the bash shell for Windows. It will not work at the Windows Command Prompt, and it might have limited functionality in Mac OS.

Warning: When working at the command line, any spaces in file names or directory names either need to be escaped in the commands you type, or the entire name needs to be wrapped in double quotes. For example:

```
cd assignment\ files
```

```
cd "assignment files"
```

It's probably easiest to just avoid file and folder names with spaces.

8. Running Test Cases Individually

If the *test-all.sh* script is telling you that some of your test cases are failing, you'll want to compile and run those test cases individually to inspect their output. Here are two ways you can do that:

1. The ideal way:
    a. (*Optional*) Remove the *main()* function from your *assignment01.c* file.
    b. Compile *both* your *assignment01.c* file and the test case file you want to run into a single program. To compile multiple source files at the command line, simply type both filenames after *gcc*:

    ```
    gcc assignment01.c testcase01.c
    ```

    c. Run the program as usual:

    ```
    ./a.out
    ```

    d. If you want to check whether the output of your program is an *exact* match of the expected output for that case, you can force your program to dump its output into a text file and then use the *diff* command to check whether the newly created text file has the exact same contents as one of the sample output test files. For example:

```
./a.out > my_output.txt
diff my_output.txt sample_output/output01.txt
```

Note: If two files have the exact same contents, *diff* does not produce any output. If it produces an output message, that means the files differ somehow.

2. Following is the less ideal way to run a single test case. However, this is what you'll most likely want to do if you want to write your own *main()* function to test your code:
   a. Comment out the *#include "assignment01.h"* line in your *assignment01.c* source file.
   b. Copy and paste the *main()* function from one of the test case files (such as *testcase01.c*) into your *assignment01.c* source file, or write your own *main()* function for testing.
   c. Compile *assignment01.c* as usual:

```
gcc assignment01.c
```

   d. Run the program as usual:

```
./a.out
```

   e. When you're finished, don't forget to un-comment the *#include "assignment01.h"* line in your *assignment01.c* file so that your code will be compatible with our grading infrastructure!

Important Note: When grading your programs, we will use different test cases from the ones we've released with this assignment, to ensure that no one can game the system and earn credit by simply hard-coding the expected output for the test cases we've released to you. You should create additional test cases of your own in order to thoroughly test your code.

9. Deliverables (Submitted via Webcourses, Not Eustis)

Submit a single source file, named *assignment01.c*, via Webcourses. The source file should contain definitions for all the required functions (listed above), as well as any helper functions you've written to make them work. Don't forget to *#include "assignment01.h"* in your *assignment01.c* code.

Do not submit additional source files, and do not submit a modified *assignment01.h* file. Your source file must work with the *test-all.sh* script, and it must be able to compile and run with each individual test case, like so:

```
gcc assignment01.c testcase01.c
./a.out
```

Be sure to include your name, the course number, the current semester, and your NID in a comment at the top of your source file.

10. Grading

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

50%	Passing test cases with 100% correct output formatting. Points will be awarded for each individual test case you pass. (It's possible to pass some, but not others.)
25%	Adherence to all style restrictions listed above. We will likely impose significant penalties for small deviations, because we really want you to develop good coding style habits in this class.
20%	Implementation details and adherence to the special restrictions imposed on this assignment. This will likely involve some manual inspection of your code.
5%	Naming your file correctly (capitalization counts) and including all required information in a comment at the top of the source file you submit.

Note! Your program must be submitted via Webcourses, and it must compile and run on Eustis to receive credit. Programs that do not compile on Eustis will receive an automatic zero.

Your grade will be based largely on your program's ability to compile and produce the *exact* output expected. Even minor deviations (such as capitalization or punctuation errors) in your output will cause your program's output to be marked as incorrect, resulting in severe point deductions. The same is true of how you name your functions and their parameters. Please be sure to follow all requirements carefully and test your program throughly.

Start early. Work hard. Good luck!