

Programming Assignment #6: Structs and Struct Pointers

COP 3223H, Fall 2019

Due: Monday, December 2, *before* 11:59 PM

Table of Contents

1. Super Important: Initial Setup (<i>Same As Usual</i>).....	2
2. Note: Test Case Files Might Look Wonky in Notepad.....	2
3. Assignment Overview.....	2
4. Overview: The <i>pancake_info</i> Struct.....	2
5. Function Requirements.....	2
6. Special Restrictions (<i>Important!</i>).....	5
7. Style Restrictions (<i>Important!</i>).....	6
8. Running All Test Cases on Eustis.....	7
9. Running Test Cases Individually.....	8
10. Deliverables (Submitted via Webcourses, Not Eustis).....	9
11. Grading.....	9

Deliverables

assignment06.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 (*Same As Usual*)

At the very top of your *assignment06.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 "assignment06.h"
```

If you do not *#include* that file properly, your program will not compile on our end, and it will not receive credit. Note that you should also *#include* any other standard libraries your code relies upon (*stdio.h*, *ctype.h*, etc.). In your code, writing *main()* is optional.

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

You already know this from previous assignments. All the test case files included with this assignment should be opened with a good coding text editor (not Notepad).

3. Assignment Overview

This assignment is designed to help you solidify your understanding of structs and struct pointers in C. In this assignment, you will write several functions (listed below). Some will be more straightforward than others. Before you start working on this assignment, your best bet is to read and comprehend all the notes posted in Webcourses on these topics.

4. Overview: The *pancake_info* Struct

This assignment brings everything full circle by referencing the pancake functions from Assignment #1. In this program, you'll have all the information about someone's pancake-consuming activities wrapped up in a struct:

```
typedef struct pancake_info
{
    char *name;                // Individual's full name.
    double pancake_count;      // Number of pancakes eaten.
    double pancakes_per_minute; // Pancakes consumed per minute.
    double minutes_spent_munching; // Minutes spent eating pancakes.
} pancake_info;
```

Super Important: You should **not** type up this struct definition in your *assignment06.c* file! It gets imported into your program automatically when you *#include "assignment06.h"*.

5. Function Requirements

You must implement the following functions in a file named *assignment06.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.

```
void print_pancake_report(pancake_info p);
```

Description: This function takes a *pancake_info* struct (*p*) and prints the data it contains in the following format. Always use exactly three decimal places of precision when printing the doubles in one of these structs. Note: The name will be a valid, non-empty, non-NULL string.

```
Name: Bo Bobbity Frank
Pancakes Consumed: 23.394
Pancakes Per Minute: 2.100
Minutes Spent Munching: 11.140
```

If you need any clarification on the expected output format here, please be sure to look at the output files for this function’s test cases. Note that we should print the data contained within the struct, even if it doesn’t make sense (i.e., even if the numbers don’t check out mathematically).

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

Related Test Cases: *testcase01a.c*, *testcase01b.c*, *testcase01c.c*

```
double get_missing_pancake_info(pancake_info p);
```

Description: This function takes a *pancake_info* struct (*p*) in which exactly one of the doubles is guaranteed to be zero; the other two will be positive (i.e., strictly greater than zero) real numbers. From the two non-zero members, calculate and return the correct value for the member that was equal to zero.

For example, if the *pancake_count* member is 5.0, the *pancakes_per_minute* member is 2.5, and the *minutes_spent_munching* member is 0.0, you need to calculate and return the number of minutes spent munching (which, in this case, is 2.0).

Return Value: Return a *double* as described above.

Related Test Cases: *testcase02a.c*, *testcase02b.c*, *testcase02c.c*

```
void update_missing_pancake_info(pancake_info *p);
```

Description: This function has the same behavior as *get_missing_pancake_info()*, with two exceptions: (1) the input parameter is a pointer to a *pancake_info* struct, and (2) instead of returning the appropriate value for whichever member was set to zero, this function should update the value of that member inside the struct pointed to by *p*.

For example, if the *pancake_count* member is 5.0, the *pancakes_per_minute* member is 2.5, and the *minutes_spent_munching* member is 0.0, the *minutes_spent_munching* member of the struct pointed to by *p* should be set to 2.0.

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

Related Test Cases: *testcase03a.c, testcase03b.c, testcase03c.c*

```
pancake_info *make_pancake_info(char *name, double count, double rate, double minutes);
```

Description: This function takes four arguments: a string (*name*), a double indicating how many pancakes that person has eaten (*count*), a double indicating the number of pancakes per minute that person can consume (*rate*), and the number of minutes that person has spent eating pancakes (*minutes*). Within this function, you must then do the following:

- Dynamically allocate a new *pancake_info* struct.
- Set the *pancake_count*, *pancakes_per_minute*, and *minutes_spent_munching* members of your dynamically allocated struct to the corresponding values passed to this function.
- Dynamically allocate a string (a *char* array) that is long enough to hold the name passed to this function. Store the address of that newly allocated string in the *name* field of your dynamically allocated struct, and copy the name passed to this function into that newly allocated string space.
- Return the address of the struct you dynamically allocated within this function.

You may assume *name* is a valid, non-empty, non-NULL string.

Return Value: Return the address of the struct you dynamically allocated within this function.

Related Test Cases: *testcase04a.c, testcase04b.c, testcase04c.c*

```
pancake_info *clone_pancake_info(pancake_info source);
```

Description: This function has the same behavior as *make_pancake_info()*, except that instead of receiving four separate parameters, it receives all the information it needs inside a single struct parameter. As above, this function should do the following:

- Dynamically allocate a new *pancake_info* struct.
- Set the *pancake_count*, *pancakes_per_minute*, and *minutes_spent_munching* members of your dynamically allocated struct to the corresponding values in the *source* struct.
- Dynamically allocate a string (a *char* array) that is long enough to hold the name stored within the source struct passed to this function. Store the address of that newly allocated string in the *name* field of your dynamically allocated struct, and copy the name from the source struct into that newly allocated string space.

You may assume the name within the struct passed to this function is a valid, non-empty, non-NULL string.

Return Value: Return the address of the struct you dynamically allocated within this function.

Related Test Cases: *testcase05a.c, testcase05b.c, testcase05c.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).

Related Test Case: *testcase06a.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.

Related Test Case: *testcase06b.c*

```
double prior_experience(void)
```

Description: This is the same function as in Programming Assignment #1. 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).

Related Test Case: *testcase06c.c*

6. Special Restrictions (*Important!*)

You must abide by the following restrictions in the *assignment06.c* file you submit. Failure to abide by any one of these restrictions could result in a catastrophic loss of points.

- ★ (**New!**) You cannot use typecasting in this assignment. (That's where you put a data type in parentheses in front of a variable in order to force C to view it as a particular data type – possibly a data type other than what it really is. I'm imposing this restriction so people can't use type casting to artificially suppress warnings related to the misuse of pointers.)
- ★ Do not read or write to any files (using, e.g., C's *fopen()*, *fprintf()*, or *fscanf()* functions).
- ★ 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 write malicious code, including code that attempts to open files it shouldn't be opening, whether for reading or writing. (I would hope this would go without saying.)
- ★ No crazy shenanigans.

7. 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 indented 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()`.

8. 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 *assignment06.c*, *assignment06.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 (*assignment06.c*, *assignment06.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 assignment06
```

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.

9. 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 *assignment06.c* file.
- b. Compile *both* your *assignment06.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 assignment06.c testcase01a.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 "assignment06.h"* line in your *assignment06.c* source file.
- b. Copy and paste the *main()* function from one of the test case files (such as *testcase01a.c*) into your *assignment06.c* source file, or write your own *main()* function for testing.

- c. Compile *assignment06.c* as usual:

```
gcc assignment06.c
```

- d. Run the program as usual:

```
./a.out
```

- e. When you're finished, don't forget to un-comment the *#include "assignment06.h"* line in your *assignment06.c* file so that your code will be compatible with our grading infrastructure!

10. Deliverables (Submitted via Webcourses, Not Eustis)

Submit a single source file, named *assignment06.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 "assignment06.h"` in your *assignment06.c* code.

Do not submit additional source files, and do not submit a modified *assignment06.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 assignment06.c testcase01a.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.

11. Grading

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.

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

- | | |
|-----|--|
| 90% | 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.) |
| 10% | Code includes useful and appropriate comments and adheres to all style restrictions and special 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. Points may also be deducted from this category for missing required information in the header comment(s), or for incorrect placement of the header comment(s), or for naming the source file incorrectly. |

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 thoroughly.

Please also note that failure to abide by any the special restrictions listed above on pg. 5 (Section 6, "Special Restrictions") could result in a catastrophic loss of points.

Start early. Work hard. Good luck!