Programming Assignment #2: So Many Loops

COP 3223H, Fall 2019

Due: Sunday, September 29, *before* 11:59 PM

Table of Contents

Deliverables

assignment02.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 *assignment02.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 "assignment02.h"
```

Yes, that needs to be in "double quotes" instead of <angled brackets>, because *assignment02.h* is a local header file that we've included with this assignment, rather than a standard system header file.

The *assignment02.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. Note that you should also *#include <stdio.h>* as usual.

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 "assignment02.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 *assignment02.h* (provided in the ZIP file for this assignment) in the same folder as your *assignment02.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

The primary purpose of this assignment is to give you exercises to help solidify your understanding of loops and stretch your brain in ways that will make you think more like an elite coding ninja.

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.

4. Function Requirements

You must implement the following functions in a file named *assignment02.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_one_through_n(int n)`

Description: This function should print all integers from 1 through *n*. Each integer should be followed by a single newline character ('\n'). You may assume the integer passed to this function will always be positive (i.e., strictly greater than zero). For example, *print_one_through_n(2)* should produce the following output:

```
1
2
```

Output: See *output01.txt* for an example of the *exact* output format expected for this function.

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

Related Test Case: *testcase01.c*

`void print_n_through_one(int n)`

Description: This function should print all integers from *n* down through 1. Each integer should be followed by a single newline character ('\n'). You may assume the integer passed to this function will always be positive (i.e., strictly greater than zero). For example, *print_n_through_one(2)* should produce the following output:

```
2
1
```

Output: See *output02.txt* for an example of the exact output format expected for this function.

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

Related Test Case: *testcase02.c*

`void tada(int n)`

Description: Print the integers 1 through *n*, with the following formatting restrictions:

1. After each integer, print "..." (without the quotes), followed by a newline character ('\n').
2. The first line of output should have no spaces before the integer being printed, the second line of output should have one space before the integer being printed, the third line of output should have two spaces before the integer being printed, and so on, with one additional space for each successive line of output.
3. The last line of output should print "Tada!" (without the quotes), followed by a newline character ('\n'). That line should have one more space than the line before it.

For example, *tada(4)* should produce the following output:

```
1...
 2...
  3...
   4...
    Tada!
```

You may assume the integer passed to this function will always be positive (i.e., strictly greater than zero).

Output: See *output03.txt* for an example of the *exact* output format expected for this function.

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

Related Test Case: *testcase03.c*

`void print_multiples_of_ten(int n)`

Description: This function takes a single integer parameter (*n*) and prints that many successive multiples of 10, beginning with 10 itself and working up from there. Each multiple of 10 should be followed by a newline character. For example, *print_multiples_of_ten(4)* would print:

```
10
20
30
40
```

You may assume the integer passed to this function will always be positive (i.e., strictly greater than zero).

Output: See *output04.txt* for an example of the *exact* output format expected for this function.

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

Related Test Case: *testcase04.c*

`void print_comma_separated_multiples_of_ten(int n)`

Description: This function is similar to *print_multiples_of_ten()*, but it should print a comma and a

space after each multiple of ten, *including* the last multiple of ten that it prints. At the end of that output, this function should also print a newline character (‘\n’). *print_comma_separated_multiples_of_ten(4)*, for example, would print the following:

```
10, 20, 30, 40,
```

Note that there is an invisible space after the last comma in the output shown above.

You may assume the integer passed to this function will always be positive (i.e., strictly greater than zero).

Output: See *output05.txt* for an example of the exact output format expected for this function.

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

Related Test Case: *testcase05.c*

`void print_comma_separated_multiples_of_ten_fancy(int n)`

Description: This function should produce the same output as the previous function (the *print_comma_separated_multiples_of_ten()* function), with the following exception: after the last multiple of ten is printed, it should ***not*** print a comma followed by a space. Instead, it should simply print a newline character (‘\n’) after that last multiple of ten. For example, a call to *print_comma_separated_multiples_of_ten_fancy(4)* would print the following:

```
10, 20, 30, 40
```

You may assume the integer passed to this function will always be positive (i.e., strictly greater than zero).

You might need an *if* statement inside of a loop in order to make this one print correctly, but there’s also a way to do this without any *if* statements.

Output: See *output06.txt* for an example of the *exact* output format expected for this function.

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

Related Test Case: *testcase06.c*

`void print_duet_symmetric(int n, char yin, char yang)`

Description: This function takes a single integer parameter (*n*), followed by two character parameters (*yin* and *yang*). This function must print *n* instances of the *yin* character, followed immediately by *n* instances of the *yang* character, followed by a newline character (‘\n’). For example, *print_duet_symmetric(5, 'x', 'o')* should print the following:

```
xxxxxooooo
```

You may assume the integer passed to this function will always be non-negative (i.e., we guarantee that we will only ever call this function with $n \geq 0$).

Output: See *output07.txt* for examples of the *exact* output format expected for this function.

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

Related Test Case: *testcase07.c*

`void print_duet(int m, int n, char yin, char yang)`

Description: This function takes two integer parameters (*m* and *n*), followed by two character parameters (*yin* and *yang*). This function must print *m* instances of the *yin* character, followed immediately by *n* instances of the *yang* character, followed by a newline character (‘\n’). For example, *print_duet(5, 3, 'x', 'o')* should print the following:

```
xxxxxooo
```

You may assume the integers passed to this function will always be non-negative (i.e., we guarantee that we will only ever call this function with $m \geq 0$ and $n \geq 0$).

Output: See *output08.txt* for examples of the *exact* output format expected for this function.

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

Related Test Case: *testcase08.c*

`void print_duet_half_and_half(int n, char yin, char yang)`

Description: This function takes a single integer parameter (*n*), followed by two character parameters (*yin* and *yang*). This function must print *n* characters in total. The first half of them must be *yin* characters, and the second half of them must be *yang* characters. The final *yang* character must be followed immediately by a newline character (‘\n’). For example, *print_duet_half_and_half(6, 'x', 'o')* should print the following:

```
xxxooo
```

You may assume the integer passed to this function will always be an even integer, and it will always be non-negative (i.e., we guarantee that we will only ever call this function with $n \geq 0$).

Output: See *output09.txt* for examples of the *exact* output format expected for this function.

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

Related Test Case: *testcase09.c*

`void print_duet_with_yin_dominating(int n, char yin, char yang)`

Description: This function takes a single integer parameter (*n*), followed by two character parameters (*yin* and *yang*). This function must print *n* characters in total. The first half of them must be *yin* characters, and the second half of them must be *yang* characters. The final *yang* character must be followed immediately by a newline character (‘\n’). If *n* is odd, this function should print exactly one

more *yin* character than *yang* characters. For example, *print_duet_with_yin_dominating(7, 'x', 'o')* should print the following:

```
xxxxooo
```

You may assume the integer passed to this function will always be non-negative (i.e., we guarantee that we will only ever call this function with $n \geq 0$).

Output: See *output10.txt* for examples of the *exact* output format expected for this function.

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

Related Test Case: *testcase10.c*

`void print_duet_with_yang_dominating(int n, char yin, char yang)`

Description: This function takes a single integer parameter (*n*), followed by two character parameters (*yin* and *yang*). This function must print *n* characters in total. The first half of them must be *yin* characters, and the second half of them must be *yang* characters. The final *yang* character must be followed immediately by a newline character (‘\n’). If *n* is odd, this function should print exactly one more *yang* character than *yin* characters. For example, *print_duet_with_yang_dominating(7, 'x', 'o')* should print the following:

```
xxxoooo
```

You may assume the integer passed to this function will always be non-negative (i.e., we guarantee that we will only ever call this function with $n \geq 0$).

Output: See *output11.txt* for examples of the *exact* output format expected for this function.

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

Related Test Case: *testcase11.c*

> ***Important Note:*** The next three functions should be possible without any *if* statements. If you find yourself writing *if* statements in your loops for these next three, you might be over-complicating the logic here.

`void print_glowie_basic(int num_spaces, int num_segments)`

Description: This function takes two integer parameters (*num_spaces* and *num_segments*) and does the following:

1. First, print *num_spaces* number of consecutive space characters.
2. Secondly, print a tilde (‘~’).

3. Thirdly, print *num_segments* number of lowercase ‘o’ characters.
4. Next, print a capital ‘O’ character followed by a capital ‘G’ character.
5. Finally, print a newline character (‘\n’).

For example, *print_glowie_basic(3, 4)* should print the following:

```
   ~ooooOG
```

The output above has exactly three spaces before the ‘~’ character.

The little thing you just printed out is an adorable little glowworm. The ‘~’ is its tail, the ‘G’ is its head, and the ‘o’ and ‘O’ characters are segments of the worm.

You may assume the integers passed to this function will always be non-negative (i.e., we guarantee that we will only ever call this function with *num_spaces* ≥ 0 and *num_segments* ≥ 0).

Output: See *output12.txt* for examples of the *exact* output format expected for this function.

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

Related Test Case: *testcase12.c*

`void print_glowie_with_platform(int num_spaces, int num_segments)`

Description: This function produces the same output as *print_glowie_basic()*, but with a platform beneath the glowworm. The platform is made up entirely of equal signs (‘=’), and is followed by *<u>two</u>* newline characters (“\n\n”). There should be equal signs from the very beginning of the platform line, all the way up to and including the spot directly beneath the glowworm’s ‘G’ character. For example, *print_glowie_with_platform(3, 4)* should print the following:

```
   ~ooooOG
==========

```

The first line of the output above has exactly three spaces before the ‘~’ character.

You may assume the integers passed to this function will always be non-negative (i.e., we guarantee that we will only ever call this function with *num_spaces* ≥ 0 and *num_segments* ≥ 0).

Output: See *output13.txt* for examples of the *exact* output format expected for this function.

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

Related Test Case: *testcase13.c*

`void print_glowie_from_coordinates(int tail_coordinate, int head_coordinate)`

Description: This function is very similar to *print_glowie_with_platform()*, but the parameters are given to you in such a way that you have to do a bit of math to figure out how to print the glowie.

The first parameter tells you the "coordinate" of the glowworm's tail ('~'), where 0 corresponds to the first character in the output line, 1 corresponds to the second character on the output line, 2 corresponds to the third character on the output line, and so on. So, if *tail_coordinate* = 0, that means the tail should be printed at the very beginning of the line; if *tail_coordinate* = 1, that means there should be a single space, followed by the tail; if *tail_coordinate* = 2, that means there should be *two* spaces, followed by the tail; and so on. In other words, you can think of *tail_coordinate* as the number of spaces you have to print before the tail.

The second parameter tells you the "coordinate" of the glowworm's head ('G'), using the same scheme as above. That coordinate effectively indicates how many characters must be printed *before* we print the 'G'. The composition of the glowworm is the same as in the previous functions, though: there's a single tail ('~'), followed by some number of lowercase 'o' segments, followed by a single 'O' segment, followed by a single head ('G'), followed by a newline character ('\n').

After printing the glowworm, you should print a platform of equal signs ('=') as described in the previous function, followed by *two* newline characters ("\n\n").

You will have to do some math here. It might be helpful if you calculate how many spaces to print and how many little segments ('o') to print before printing the final "OG" of the glowworm. Once you calculate those numbers, you will probably find that the rest of the function will fall into place quite nicely.

For example, *print_glowie_from_coordinates(0, 5)* should print the following. Note that we print *zero* characters before the tail ('~') and a total of *five* characters before the head ('G'):

```
~oooOG
======
```

Similarly, *print_glowie_from_coordinates(0, 2)* should print the following. Note that we print *zero* characters before the tail ('~') and a total of *two* characters before the head ('G'):

```
~OG
===
```

Similarly, *print_glowie_from_coordinates(2, 8)* should print the following. Note that we print *two* characters (spaces) before the tail ('~') and a total of *eight* characters before the head ('G'):

```
  ~ooooOG
=========
```

Note: The first line of the output above has exactly two spaces before the '~' character.

And finally, note that *print_glowie_from_coordinates(2, 4)* should print the following. Note that we print *two* characters (spaces) before the tail ('~') and a total of *four* characters before the head ('G'):

```
  ~OG
=====
```

Note: The first line of the output above has exactly two spaces before the '~' character.

To help make the math manageable in this function, whenever we call this function, we will ***always*** ensure that *head_coordinate* is greater than *tail_coordinate* by ***at least*** 2. So, if *tail_coordinate* = 7, we ***guarantee*** that we will give you *head_coordinate* ≥ 9. That means you will always print a '~', 'O', and 'G', but there might be zero lowercase 'o' segments.

Output: See *output14.txt* for examples of the *exact* output format expected for this function.

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

Related Test Case: *testcase14.c*

`void print_glowie_dead_or_alive(int tail_coordinate, int head_coordinate, int is_alive)`

Description: This function produces the exact same output as *print_glowie_from_coordinates()*, with the following twist: there is an *is_alive* parameter, which will either be 1 or 0. If *is_alive* is 1, then the glowworm is alive, and the output of this function should be exactly the same as the *print_glowie_from_coordinates()* function. If *is_alive* is 0, then the glowworm is dead, so in place of the "OG" that typically prints out for the far-right end of the glowworm, you should print "Xx".

For example, *print_glowie_dead_or_alive(3, 9, 1)* should produce the following output:

```
   ~ooooOG
==========
```

Note: The first line of the output above has exactly three spaces before the '~' character.

In contrast, *print_glowie_dead_or_alive(3, 9, 0)* should produce the following output:

```
   ~ooooXx
==========
```

The first line of the output above has exactly three spaces before the '~' character.

In this function, we will follow the same constraints for *tail_coordinate* and *head_coordinate* as in the *print_glowie_from_coordinates()* function. Also, *is_alive* is guaranteed to be either 0 or 1.

Output: See *output15.txt* for examples of the *exact* output format expected for this function.

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

Related Test Case: *testcase15.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: *testcase16.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: *testcase17.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).

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*

5. Special Restrictions (*Important!*)

You must abide by the following restrictions in the *assignment02.c* file you submit. Failure to abide by any one of these restrictions could result in a catastrophic loss of points.

- ★ (***New!***) No nested loops! Please break everything up into functions instead.
- ★ (***New!***) You cannot use arrays or pointers in this assignment.
- ★ 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.

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 *assignment02.c*, *assignment02.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 (*assignment02.c*, *assignment02.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 assignment02
```

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 *assignment02.c* file.
    b. Compile *both* your *assignment02.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 assignment02.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 "assignment02.h"* line in your *assignment02.c* source file.
    b. Copy and paste the *main()* function from one of the test case files (such as *testcase01.c*) into your *assignment02.c* source file, or write your own *main()* function for testing.
    c. Compile *assignment02.c* as usual:
    ```
    gcc assignment02.c
    ```
    d. Run the program as usual:

```
./a.out
```

e. When you're finished, don't forget to un-comment the *#include "assignment02.h"* line in your *assignment02.c* file so that your code will be compatible with our grading infrastructure!

9. Deliverables (Submitted via Webcourses, Not Eustis)

Submit a single source file, named *assignment02.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 "assignment02.h"* in your *assignment02.c* code.

Do not submit additional source files, and do not submit a modified *assignment02.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 assignment02.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

> ***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:

60%	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.
10%	Implementation details. 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.

Please also note that failure to abide by any the special restrictions listed above on pg. 11 (Section 5, “Special Restrictions”) could result in a catastrophic loss of points.

Start early. Work hard. Good luck!