Programming Assignment #1: Orcheeseo

COP 3502, Fall 2020

Due: Sunday, September 6, *before* 11:59 PM

Abstract

In this assignment, you will write a few string-processing functions that are designed to get you thinking in C again. You will write a *main()* function that can process command line arguments – parameters that are typed at the command line and passed into your program right when it starts running (as opposed to using *scanf()* to get input from a user *after* your program has already started running). Those parameters will be passed to your *main()* function as an array of strings, and so this program will also jog your memory with respect to using arrays and strings in C.

On the software engineering side of things, you will learn to construct programs that use multiple source files and custom header (*.h*) files. This assignment will also help you hone your ability to acquire new knowledge by reading technical specifications. If you end up pursuing a career as a software developer, the ability to rapidly digest technical documentation and work with new software libraries will be absolutely critical to your work, and this assignment will provide you with an exercise in doing just that.

Finally, this assignment is specifically designed to require fewer lines of code than our remaining programming assignments so that you can get acclimated to the assignment testing setup in this class and make your first foray into the world of Linux without a huge, unwieldy, and intimidating program to debug. As the semester progresses, the programming assignments will become more lengthy and complex, but for now, this assignment should provide you with a gentle(ish) introduction to a development environment and software engineering concepts that will most likely be quite foreign to you at first.

Deliverables

Orcheeseo.c

Note! The capitalization and spelling of your filename matter!

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

1. Assignment Overview (Also: What is an Orcheeseo?)

This summer, a few of my TAs told me they had started placing Goldfish crackers inside Oreo cookies, like so:

Figure 1. A TA holds an Orcheeseo. Taken July 2020. Note that this approach to the Goldfish crackers is suboptimal. Ideally, one should crumble the Goldfish crackers a bit so they can be mushed into the crème filling and make the Orcheeseo more cohesive and less unwieldy.

If you're put off by this idea, you're not alone. A few of the TAs objected to these concotions on moral grounds, and others thought they just looked kind of nasty.[1] You might even object to Oreos in general on the grounds that they're incredibly bad for your health (did you know there was actually a lawsuit in California aimed at getting Oreos banned throughout the state?), or that the crème filling has an odd, synthetic taste. You might also object to Goldfish crackers on the grounds that they're not as good as Cheez-Its.

But I digress.

Some of the TAs involved with these creations (which I've dubbed "Orcheeseos")[2] started experimenting with various quantities and placements of layers within the mixed sandwich cookies (since there wasn't much else to do while everyone was locked down this summer). An Orcheeseo consists of the following three types of layers:

---

1 At the time of this writing, I myself have not yet tried an Orcheeseo, but I'm keeping an open mind. My plan is to try them after I've finished writing this assignment, as I don't want my ultimate opinion on them to bias my writing. As university students, I hope you, too, will work to keep open – yet critical – minds when confronted with ideas that challenge societal norms and/or your preconceived notions about the world.

2 Etymological note: "Orcheeseo" comes from sandwiching the word "cheese" right in the middle of the word "Oreo," much in the same way the cheesy Goldfish crackers are being stuffed inside the Oreo cookies.

1. The chocolate cookies from the Oreos
2. The sweet crème filling from the Oreos
3. Crushed Goldfish crackers

In this assignment (which is inspired by those TAs' experiments), you will write a function called *printOrcheeseo()* that takes a single argument – a string representing the composition of an Orcheeseo – and prints a diagram depicting the Orcheeseo represented by that argument. Each string received by the *printOrcheeseo()* function is guaranteed to contain only the lowercase characters 'o', 'c', and 'g', and to be terminated by a null sentinel ('\0'). The Orcheeseo diagrams must be printed according to the following rules:

1. Letters in the string are processed from left to right.
2. An 'o' (for "Oreo") causes us to print a cookie layer (10 equals symbols) (see diagrams below).
3. A 'c' (for "crème") causes us to print a layer of crème filling (8 tilde symbols preceded by a space).
4. A 'g' (for "Goldfish") causes us to print one or more layers of goldfish crackers, depending on how many consecutive 'g' characters we encounter. Each 'g' corresponds to a single Goldfish. The rules for printing Goldfish are as follows:
    a. Given a sequence of consecutive Goldfish crackers ('g' characters), we first count how many crackers there are in that sequence.
    b. If the number of goldfish crackers is exactly divisible by three, we print one standard layer of Goldfish crumbles for every three goldfish. A standard layer of Goldfish is denoted using four pairs of 'x' and 'o' characters, preceded by a single space: " xoxoxoxo"
    c. If attempting to divide the goldfish into groups of three leaves us with two leftover goldfish, we will print one impoverished layer of Goldfish crumbles above any standard layers of Goldfish for this particular sequence of crackers (with one standard layer for each group of three goldfish). An impoverished layer of Goldfish is denoted using only three pairs of 'x' and 'o' characters, preceded by two spaces: " xoxoxo"
    d. If attempting to divide the goldfish into groups of three leaves us with one leftover goldfish, we have two options:
        i. If this is the only goldfish (i.e., this particular goldfish is not *immediately* preceded or followed by any other goldfish in this part of the string), then it goes by itself in a very thin layer of crumbs denoted using three pairs of '.' and ',' characters, preceded by two spaces: " .,.,.,"
        ii. If, on the other hand, this leftover goldfish was part of a sequence of consecutive 'g' characters, then it should combine with the *last* standard layer of Goldfish crumbles generated by this particular sequence (where there's one standard layer for each group of three goldfish) to form one extended layer of Goldfish. An extended layer of Goldfish is denoted using five pairs of 'x' and 'o' characters: "xoxoxoxoxo"

For example, consider the string "ococccgggggggggco". Processing the string from left to right, we would first print a cookie layer (for the 'o'), followed by a crème filling layer (for the 'c'), followed by one cookie layer and three crème layers. We then encounter a sequence of eight consecutive goldfish. When attempting to divide them into groups of three, we end up with two groups of three goldfish and two leftover goldfish. Accordingly, we invoke rule 4(c) (given above) and print an impoverished layer of crackers (for the two leftovers), followed by

two complete layers of crackers (for the two groups of three goldfish). We finish with a crème layer and a cookie layer (for the final ‘c’ and ‘o’ in the string). The resulting diagram looks like so (minus the highlighted text I’ve added at the end of each line for clarity):

```
==========  <- cookie                                (height: 0.4 mm)
 ~~~~~~~~   <- crème filling                         (height: 0.25 mm)
==========  <- cookie                                (height: 0.4 mm)
 ~~~~~~~~   <- crème filling                         (height: 0.25 mm)
 ~~~~~~~~   <- crème filling                         (height: 0.25 mm)
 ~~~~~~~~   <- crème filling                         (height: 0.25 mm)
  xoxoxo    <- goldfish (impoverished layer)         (height: 0.25 mm)
 xoxoxoxo   <- goldfish (standard layer)             (height: 0.25 mm)
 xoxoxoxo   <- goldfish (standard layer)             (height: 0.25 mm)
 ~~~~~~~~   <- crème filling                         (height: 0.25 mm)
==========  <- cookie                                (height: 0.4 mm)
```

Next, let’s examine what happens with the string “ogggoggggogoggggggggo”. In particular, here’s what happens with the layers of Goldfish crackers:

- The first sequence of goldfish has three crackers, so we print a standard layer of crackers using rule 4(b).
- The second sequence of goldfish has four ‘g’ characters, so that one extra goldfish unites with the other three to create an extended layer of crackers using rule 4(d)(ii).
- The third sequence of goldfish has a single ‘g’ character, so it generates a very thin layer of crumbs using rule 4(d)(i).
- The fourth sequence of goldfish has seven ‘g’ characters. This triggers rule 4(d)(ii), giving us a standard layer of crackers, followed by an extended layer of crackers.

The resulting Orcheeseo looks like so (minus the highlighted text I’ve added at the end of each line for clarity):

```
==========  <- cookie                                (height: 0.4 mm)
 xoxoxoxo   <- goldfish (standard layer)             (height: 0.25 mm)
==========  <- cookie                                (height: 0.4 mm)
xoxoxoxoxo  <- goldfish (extended layer)             (height: 0.25 mm)
==========  <- cookie                                (height: 0.4 mm)
  .,.,.,    <- goldfish (very thin layer of crumbs)  (height: 0.1 mm)
==========  <- cookie                                (height: 0.4 mm)
 xoxoxoxo   <- goldfish (standard layer)             (height: 0.25 mm)
xoxoxoxoxo  <- goldfish (extended layer)             (height: 0.25 mm)
==========  <- cookie                                (height: 0.4 mm)
```

There are a few final considerations that affect the output of the *printOrcheeseo()* function:

1. Is this a well-formed Orcheeseo? To be considered well-formed, the sequence must start and end with a cookie layer, and it must contain at least one Goldfish cracker. If the top and bottom layers aren’t both cookies, then we have a big mess. And if there are no Goldfish in the sequence, then this is just an Oreo.
2. Will the Orcheeseo in question fit in the average person’s mouth, or has it gotten too tall? Here, let’s assume each cookie layer is 0.4 mm tall, each very thin layer of goldfish crumbs is 0.1 mm tall, and all other layers (all crème layers and all other goldfish layers) are each 0.25 mm tall. Let’s also assume the average person can only comfortably consume a cookie that is 4.0 mm tall or shorter.

After the Orcheeseo diagram, *printOrcheeseo()* prints **exactly one** of the following:

1. If there were no characters in the string – i.e., if it were the empty string, which has a string length of zero – then the function simply prints "No cookie. :(" (followed by a newline character).
2. If the Orcheeseo is non-empty but messy (i.e., it has at least one layer, but it doesn't start *and* end with a cookie layer), then on the line immediately after the Orcheeseo diagram, print "Too messy. :(" (followed by a newline character).
3. If the Orcheeseo is non-empty and non-messy, but the cookie was too tall to be consumed, then on the line immediately after the Orcheeseo diagram, print "Oh nooooo! Too tall. :(" (followed by a newline character).
4. If the Orcheeseo is non-empty, non-messy, and not too tall to consume, but it wasn't a well-formed Orcheeseo (as defined above), then on the line immediately after the Orcheeseo diagram, print "Om nom nom! Oreo!" (followed by a newline character).
5. If the Orcheeseo is non-empty, non-messy, not too tall to consume, and it's well-formed (as defined above), then on the line immediately after the Orcheeseo diagram, print "Om nom nom! Orcheeseo!" (followed by a newline character).

As part of this assignment, you will also write a *main()* function that will allow users to pass a string to your program as a command line argument. Your *main()* will then have to pass that string to the *printOrcheeseo()* function described above. The process for setting up your program's *main()* function to accept command line arguments is described below in Appendix A ("Processing Command Line Arguments") (pg. 14).

2. Orcheeseo.h (*Super Important!*)

Your code for this assignment will go in a file named *Orcheeseo.c*. At the very top of that 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 "Orcheeseo.h"
```

The "quotes" (as opposed to <brackets>) indicate to the compiler that this header file is found in the same directory as your source file, not a system directory. Note that filenames are case sensitive in Linux, so if you use *#include "orcheeseo.h"* (all lowercase), your program might compile on Windows, but it won't compile when we test it. You must capitalize *Orcheeseo.h* correctly.

The *Orcheeseo.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* any other standard libraries your code relies upon (such as *stdio.h* and *ctype.h*).

From this point forward, you will always have to have *Orcheeseo.h* in the same folder as your *Orcheeseo.c* file any time you want to compile your code. You should not send *Orcheeseo.h* when you submit your assignment, as we will use our own copy of *Orcheeseo.h* when compiling your code. You should also be very careful not to modify *Orcheeseo.h* while working on this assignment, except as described below in Section 8 of this PDF.

3. 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 on Windows, 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 test case files should look just fine.

4. Function Requirements

In the source file you submit, *Orcheeseo.c*, you must implement the following functions. 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. You may also write and call additional functions ("helper functions") if you find that doing so will make it easier for you to write some of these required functions.

```
int main(int argc, char **argv)
```

Description: This function should simply pass the command line argument stored in argv[1] to the *printOrcheeseo()* function and then return zero. Note that the string we pass to your program at the command line could be arbitrarily long. If no arguments are passed to your program at the command line (other than the program name in *argv[0]*), then you should not produce any output whatsoever.

Special Restrictions (Super Important!): Please be sure to see Section 5 on pg. 8 of this PDF for additional restrictions you must abide by when writing this program.

Return Value: Your *main()* function should return zero.

Related Test Cases: *arguments01.txt* through *arguments12.txt*

```
void printOrcheeseo(char *str)
```

Description: This function takes a single string argument (*str*) and prints the Orcheeseo it represents using the format described in Section 1 of this PDF: "Assignment Overview (Also: What is an Orcheeseo?)" Be sure to consult the output files included with this assignment for examples of the *exact* output formatting we expect. You may assume *str* will never be NULL, although it might be an empty string (i.e., a string whose length is zero).

Special Restriction (Super Important!): This function **must** call *printGoldfish()* (described below) once for each separate sequence of goldfish encountered in the argument string.

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

Related Test Cases: *arguments01.txt* through *arguments12.txt* and *UnitTest06.c* through *UnitTest08.c*

`double printGoldfish(int numGoldfish)`

Description: This function takes a single integer argument (*numGoldfish*) representing a number of consecutive Goldfish that have been detected in the string. It then prints the appropriate layer(s) for that number of Goldfish using the rules described above and returns the height of those goldfish layers (as defined earlier in this PDF). For example, *printGoldfish(6)* would print two standard layers of Goldfish (and return 0.5), and *printGoldfish(8)* would print an impoverished layer of Goldfish followed by two standard layers of Goldfish (and return 0.75). If *numGoldfish* < 1, this function should simply return 0.0 without printing anything.

Note: Splitting this task off into a separate function will make your *printOrcheeseo()* function *vastly* more readable, compact, and easy to debug. Breaking a problem into appropriate functions like this is called "functional decomposition."

Return Value: Return the height (as defined above) of the goldfish layers this function just printed.

Related Test Case: *UnitTest05.c*

`int returnThirtyTwo(void)`

Description: This function simply returns the integer 32. This function is here strictly to reinforce the difference between *returning* a value and *printing* a value.

Related Test Case: *UnitTest03.c*

`void printThirtyTwo(void)`

Description: This function simply prints the integer 32 to the screen, followed by a newline character ('\n'). This function is here strictly to reinforce the difference between *returning* a value and *printing* a value.

Related Test Case: *UnitTest04.c*

`double difficultyRating(void)`

Description: Return a double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult). If you are thinking about using *printf()* or *scanf()* in this function, you are off track and should visit a TA in office hours straight away to discuss this misconception.

Related Test Case: *UnitTest01.c*

`double hoursSpent(void)`

Description: Return an estimate (greater than zero) of the number of hours you spent on this assignment. Your return value must be a realistic and reasonable estimate. Unreasonably large values will result in loss of credit. If you are thinking about using *printf()* or *scanf()* in this function, you are off track and should visit a TA in office hours straight away to discuss this misconception.

Related Test Case: *UnitTest02.c*

5. Special Restrictions (*Super Important!*)

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

- ★ Please do not create any string variables or arrays in this assignment other than the command line arguments passed to *main()* and the *str* variable in *printOrcheeseo()*. You should not create any new copies of the command line argument strings, and you should not modify the contents of those strings.
- ★ You can only call *strlen()* **once** for each string processed by *printOrcheeseo()*. That's because *strlen()* is a slow function that counts up the characters in a string every single time you call it. Note that if we pass "hello" to the following function, it will call *strlen()* not just once, but **six times** before the function terminates, because *strlen()* is called every single time we perform the *i* < *strlen(str)* comparison (once for each character in the string, and one final time when the comparison evaluates to false and causes the loop to end. So, this code would violate the restriction on how many times we can call *strlen()*:

```
// This is a bogus printOrcheeseo() function.

void printOrcheeseo(char *str)
{
   int i;
   for (i = 0; i < strlen(str); i++)  // BAD! Calling strlen() so many times!
      printf("%d...\n", i);
}
```

- ★ Do not read or write to files (using, e.g., C's *fopen()*, *fprintf()*, or *fscanf()* functions). Also, please do not use *scanf()* to read input from the keyboard.
- ★ Do not declare new variables part way through a function. All variable declarations should occur at the *top* of a function, and all variables must be declared inside your functions or declared as function parameters.
- ★ Do not use *goto* statements in your code.
- ★ Do not make calls to C's *system()* function.
- ★ 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.

6. Style Restrictions (*Super Important!*)

Please conform as closely as possible to the style I use while coding in class. To encourage everyone to develop a commitment to writing consistent and readable code, the following restrictions will be strictly enforced:

- ★ Any time you open a curly brace, that curly brace should start on a new line.

- ★ 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 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*"
- ★ The header comments introducing your source file (including the comment(s) with your name, course number, semester, NID, and so on), should always be placed ***above*** your *#include* statements.
- ★ 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 excessively long lines of code. Lines must be no longer than 100 characters wide.
- ★ Avoid excessive consecutive blank lines. In general, you should never have more than one or two consecutive blank lines.
- ★ When defining a function that doesn't take any arguments, always put *void* in its parentheses. For example, define a function using *int do_something(void)* instead of *int do_something()*.
- ★ When defining or calling a function, do not leave a space before its opening parenthesis. For example: use *int main(void)* instead of *int main (void)*. Similarly, use *printf("...")* instead of *printf ("...")*.
- ★ Do leave a space before the opening parenthesis in an *if* statement or a loop. For example, use use *for (i = 0; i < n; i++)* instead of *for(i = 0; i < n; i++)*, and use *if (condition)* instead of *if(condition)* or *if(condition)*.
- ★ Please leave a space on both sides of any binary operators you use in your code (i.e., operators that take two operands). For example, use *(a + b) - c* instead of *(a+b)-c*. (The only place you do ***not*** have to follow this restriction is within the square brackets used to access an array index, as in: *array[i+j]*.)
- ★ Use meaningful variable names that convey the purpose of your variables. (The exceptions here are when using variables like *i*, *j*, and *k* for looping variables or *m* and *n* for the sizes of some inputs.)

7. Running All Test Cases on Eustis (*test-all.sh*)

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 *test-all.sh* on Eustis, first transfer it to Eustis in a folder with *Orcheeseo.c*, *Orcheeseo.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 (*Orcheeseo.c*, *Orcheeseo.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 *test-all.sh* 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 orcheeseo_project
```

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 orcheeseo\ files
```

```
cd "orcheeseo files"
```

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

8. Checking the Output of Individual Test Cases

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. This section tells you how to do that.

There are two types of test cases included with this assignment: (1) the test cases where you compile your program and run it with the command line argument listed in one of our text files (*arguments01.txt* through *arguments12.txt*), and (2) the test cases where you have to compile one of our source files (*UnitTest01.c* through *UnitTest08.c*) along with your source file (*Orcheeseo.c*) in order to run.

If you want to run one of these test cases individually in order to examine its output outside of the *test-all.sh* script, here's how you do it:

1. **Instructions for running your program with the command line argument given in one of the *.txt* files:**
   a. Place all the test case files released with this assignment in one folder, along with your *Orcheeseo.c* file.
   b. In *Orcheeseo.h*, make sure line 15 is commented out *exactly* as follows, with no space directly following the "//". This is the *only* line of *Orcheeseo.h* that you should ever modify:

      ```
      //#define main __hidden_main__
      ```

   c. At the command line, *cd* to the directory with all your files for this assignment, and compile your program:

      ```
      gcc Orcheeseo.c
      ```

   d. To run your program and redirect the output to *output.txt*, copy and paste the contents of one of the *argumentsXX.txt* files to the command line after `./a.out`, like so:

      ```
      ./a.out ogggco > output.txt
      ```

   e. As an alternative to step (d), type the following to have the command line automatically insert the argument from one of the text files for you:

      ```
      ./a.out $(cat arguments01.txt) > output.txt
      ```

   f. Use *diff* to compare your output to the expected (correct) output for the program:

      ```
      diff output.txt sample_output/arguments01-output.txt
      ```

2. **Instructions for compiling your program with one of the unit test cases:**
   a. Place all the test case files released with this assignment in one folder, along with your *Orcheeseo.c* file.
   b. In *Orcheeseo.h*, make sure line 15 uncommented and appears *exactly* as follows. This is the *only* line of *Orcheeseo.h* that you should ever modify:

      ```
      #define main __hidden_main__
      ```

   c. At the command line, *cd* to the directory with all your files for this assignment, and compile your program with *UnitTestLauncher.c* and any *one* of the *UnitTestXX.c* files you would like to test:

      ```
      gcc Orcheeseo.c UnitTestLauncher.c UnitTest01.c
      ```

d. To run your program and redirect the output to *output.txt*, execute the following command:

```
./a.out > output.txt
```

e. Use *diff* to compare your output to the expected (correct) output for the program:

```
diff output.txt sample_output/UnitTest01-output.txt
```

Super Important: *Remember, 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.*

9. Deliverable (Submitted via Webcourses, not Eustis)

Submit a single source file, named *Orcheeseo.c*, via Webcourses. The source file should contain definitions for all the required functions (listed above). Be sure to include your name and NID as a comment at the top of your source file. Also, don't forget *#include "Orcheeseo.h"* in your source code (with correct capitalization). Your source file must work on Eustis with the *test-all.sh* script, and it must also compile and run on Eustis like so:

```
gcc Orcheeseo.c
./a.out ogggco
```

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. In creating your own test cases, you should always ask yourself, "What kinds of inputs could be passed to this program that don't violate any of the input specifications, but which haven't already been covered in the test cases included with the assignment?"

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

50%	Passes test cases with 100% correct output formatting. This portion of the grade includes tests of the *difficultyRating()* and *hoursSpent()* methods.
20%	Adequate comments and whitespace. To earn these points, you must adhere to the style restrictions set forth above. We will likely impose huge penalties for small deviations, because we really want you to develop good style habits in this class. Please include a header comment with your name and NID (***not*** your UCF ID).
20%	Implementation details and adherence to the special restrictions imposed on this assignment. This will likely involve some manual inspection of your code.
10%	Source file is named correctly. Spelling and capitalization count.

Note! Your program must be submitted via Webcourses, and it must compile and run on Eustis to receive credit. Programs that do not compile 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. Your best bet is to submit your program in advance of the deadline, then download the source code from Webcourses, re-compile, and re-test your code in order to ensure that you uploaded the correct version of your source code.

Additional points will be awarded for style (appropriate commenting and whitespace) and adherence to implementation requirements. For example, the graders might inspect your code to make sure you're not creating any arrays or extra copies of command line argument strings.

11. One Last Thing

Okay, so, this might all be overwhelming, and you might be thinking, "Where do I even start with this assignment?! I'm in way over my head!" The main thing here is:

There's a helpful video posted in Webcourses where I actually walk you through how to get started on this assignment. (Check it out!) Other than that, it's a matter of starting the assignment early, reading everything carefully before you start coding, and embracing a growth mindset as you work. Please also take advantage of the abundance of TA office hours ASAP if you get stuck on anything.

Start early. Work hard. Good luck!

Appendix A:
Processing Command Line Arguments

1. Guide to Command Line Arguments

All your interactions with Eustis this semester will be at the command line, where you will use a text-based interface (rather than a graphical interface) to interact with the operating system and run programs.

When we type the name of a program to run at the command line, we often type additional parameters *after* the name of the program we want to run. Those parameters are called "command line arguments," and they are passed to the program's *main()* function upon execution without us having to call *scanf()*.

For example, in class, you've seen that I run the program called *gcc* to compile source code, and after typing "*gcc*," I always type the name of the file I want to compile, like so:

```
gcc Orcheeseo.c
```

Straight up, *gcc* is a program someone wrote, and it has a *main()* function. In this example, the string "*Orcheeseo.c*" is passed to the *gcc* program's *main()* function as a string, which tells the program which file it's supposed to open and compile.

In this assignment, your *main()* function will have to processes command line arguments. This appendix shows you how to get that set up.

2. Passing Command Line Arguments to *main()*

Your program must be able to process an arbitrary number of string arguments. For example:

```
seansz@eustis:~$ ./a.out cgo
 ~~~~~~~~
  .,.,.,
==========
Too messy. :(
```

To get command line arguments (such as the "cgo" string in the above example) into your program, you just have to tweak the function signature for the *main()* function you're writing in *Orcheeseo.c*. Whereas we've typically seen *main()* defined using *int main(void)*, you will now use the following function signature instead:

```
int main(int argc, char **argv)
```

Within *main()*, *argc* is now an integer representing the number of command line arguments passed to the program, including the name of the executable itself. So, in the example above, *argc* would be equal to 2. *argv* is an array of strings that stores all those command line arguments. *argv[0]* always stores the name of the program being executed (*./a.out*), and in the example given above, *argv[1]* would be the string "cgo".

3. Example: A Program That Prints All Command Line Arguments

For example, here's a small program that would print out all the command line arguments it receives (including the name of the program being executed). Note how we use *argc* to loop through the *argv* array:

```
int main(int argc, char **argv)
{
   int i;

   for (i = 0; i < argc; i++)
      printf("argv[%d]: %s\n", i, argv[i]);

   return 0;
}
```

If we compiled that code into an executable file called *a.out* and ran it from the command line by typing *./a.out lol hi*, we would see the following output:

```
argv[0]: ./a.out
argv[1]: lol
argv[2]: hi
```

4. Side Note

There are certain characters that cause Linux to do wonky things with command line arguments. Accordingly, if a command line argument contains any of the characters in the table below, that could lead to unexpected results in the strings passed to a program. To get around this problem, one could use the `$(cat argumentsXX.txt)` technique described in Section 8 of this PDF (which is what we use in *test-all.sh*).

dollar sign ('$')	semi-colon (';')
opening parenthesis ('(')	tilde ('~')
closing parenthesis (')')	asterisk ('*')
exclamation point ('!')	period ('.')
hash ('#')	space (' ')
ampersand ('&')	single quotes (' and ')
backslash ('\')	double quotes (" and ")
pipe ('|')	redirection symbols ('<' and '>')

Keep in mind that we will only use lowercase characters 'o', 'c', and 'g' in the strings we pass to your program, so for this assignment, there shouldn't be any reason to worry about the characters above. This information is just included here for the sake of completeness.