Programming Assignment #1: Ohce[1]

COP 3502, Spring 2018

Due: Wednesday, January 24, *before* 11:59 PM

Table of Contents

Abstract....3
1. Overview....4
2. Important Note: Test Case Files Look Wonky in Notepad....5
3. Adopting a Growth Mindset....5
4. Ohce.h....6
5. Guide to Command Line Arguments....6
6. Dissecting the Contents of the *test_cases* Folder....9
7. Running and Compiling the Two Flavors of Test Cases....9
8. Test Cases and the test-all.sh Script....11
9. Function Requirements....11
10. Special Restrictions....13
11. Compilation and Testing (CodeBlocks)....14
12. Compilation and Testing (Linux/Mac Command Line)....15
13. Compiling and Running Unit Tests (Linux/Mac Command Line)....16
14. Getting Started: A Guide for the Overwhelmed....17
15. General Tips for Working on Programming Assignments....18
16. Deliverables (Submitted via Webcourses, Not Eustis)....19
17. Grading....19

1 "Ohce" is "echo" spelled backwards.

Abstract

In this assignment, 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 require you to use some of your knowledge of array and string manipulation 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 relatively few lines of code so that you can 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 introduction to a development environment and software engineering concepts that will most likely be quite foreign to you at first.

Attachments

Ohce.h, *arguments{01-06}.txt*, *UnitTest{07-10}.c*, *UnitTestLauncher.c*, *output{01-10}.txt*, and *test-all.sh*

Deliverables

Ohce.c

Note! The capitalization and spelling of your filename matter!

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

1. Overview

Linux has a program called "*echo*" that takes any number of command line arguments (i.e., strings that are typed after the *echo* command at the command line) and prints them to the screen. For example, the following shows the results of running `echo Drink Me` at the Linux command line:

```
seansz@eustis:~$ echo Drink Me
Drink Me
seansz@eustis:~$ _
```

In the example above, *echo* is the name of the program being run, and the two command line arguments it receives (the strings "Drink" and "Me") are passed to the program's `main()` function for processing.

In this assignment, you will write a program that is very similar to Linux's *echo* program, but with a few twists. Firstly, each command line argument will be printed in reverse order. For example:

```
seansz@eustis:~$ ./a.out zebra giraffe
arbez effarig
seansz@eustis:~$ _
```

Note that the example above assumes that your program's executable file is named "*a.out*," which is the default name for executable files produced by the gcc compiler in Linux.

The second difference between Linux's *echo* program and your program is that if the word "echo" appears in any of the command line arguments passed to your program, you should also print "Echo, echo, echo!" (followed by a newline character, '\n') after your program's normal output. For example:

```
seansz@eustis:~$ ./a.out zebra giraffechobomb
arbez bmobohceffarig
Echo, echo, echo!
seansz@eustis:~$ _
```

Note that your program should detect occurrences of "echo" regardless of how they're capitalized. For example, the occurrence of "EcHo" in a command line argument should trigger the printing of "Echo, echo, echo!" as shown here:

```
seansz@eustis:~$ ./a.out zebra giraffEcHobomb
arbez bmoboHcEffarig
Echo, echo, echo!
seansz@eustis:~$ _
```

There are two other functions you have to write for this assignment. Details about those functions, as well as additional formatting requirements for your output, are given below in Section 9, "Function Requirements."

By completing this assignment, you will dust off your C programming skills, get some additional experience working with string manipulation and for-loops, learn how C programs process command line arguments, and gain experience working at the command line in Linux. You will also gain experience working with custom header files and learn a bit about compiling multiple source files into a single program.

You will submit a single source file, named `Ohce.c`, that contains all required function definitions, as well as any auxiliary functions you deem necessary. In `Ohce.c`, you will have to #include any header files necessary for your functions to work, including the custom `Ohce.h` file we have distributed with this assignment (see Section 4, "Ohce.h").

We have also included some test cases with this program and a script (`test-all.sh`) for testing your code on Eustis. Some of the test cases are designed to test the functionality of your program when you run it with command line arguments, and others are source files that you can run by compiling them alongside your own code for this program (i.e., by compiling multiple source files into a single program). Descriptions of these different types of test cases and detailed instructions on how to work with them are included in the sections that follow.

Although we have included a variety of test cases to get you started with testing the functionality of your code, we encourage you to develop your own test cases, as well. Ours are by no means comprehensive. We will use much more elaborate test cases when grading your submission.

2. Important Note: Test Case Files 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 Notepad, they will appear to contain one long line of text. That's because Notepad handles end-of-line characters differently from Linux and Unix-based systems. One solution is to view those files in an IDE (like CodeBlocks), or you could use a different text editor (like Atom or Sublime).

3. Adopting a Growth Mindset

A word of advice before we dive in to the details: When faced with an assignment like this, which has many different facets, some of which might appear foreign and/or challenging, it's important not to look at it as an instrument that is being used to measure how much you already know (whether that's knowledge of programming, operating systems, or even what some might call "natural intellectual capability"). Rather, it's important to view this assignment as a chance to learn something new, grow your skill set, and embrace a new challenge.

It's also important to view intellectual capability as something that can grow and change over one's lifetime. Adopting that mindset will allow you to reap greater benefits from this assignment (and from all your college-level coursework) regardless of the grades you earn. For more on the importance of adopting a growth mindset throughout your academic career and how that can impact your life, see the following:

> Growth Mindset vs Fixed Mindset: An Introduction (Watch time: 2 min 42 sec)
>
> The Power of Belief – Mindset and Success (Watch time: 10 min 20 sec)

4. Ohce.h

Included with this assignment is a customer header file that includes functional prototypes for all the functions you will be implementing. You **must** #include this file from your `Ohce.c` file, like so:

```
#include "Ohce.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 `#include "ohce.h"` (with a lowercase ‘o’), your program will not compile. You must use an uppercase ‘O’.

You should not send `Ohce.h` when you submit your assignment, and you should be very careful about modifying `Oche.h`. We will use our own unmodified copy of `Ohce.h` when compiling your program.

If you write auxiliary functions (“helper functions”) in your `Ohce.c` file (which should not be necessary for this particular assignment), you should **not** add those functional prototypes to `Ohce.h`. Our test case programs will not call your helper functions directly, so they do not need any awareness of the fact that your helper functions even exist. (We only list functional prototypes in a header file if we want multiple source files to be able to call those functions.) So, just put the functional prototypes for any helper functions you write at the top of your `Ohce.c` file.

Think of `Ohce.h` as a bridge between source files. It contains functional prototypes for functions that might be defined in one source file (such as your `Ohce.c` file) and called from a different source file (such as the `UnitTestXX.c` files we have provided with this assignment).

5. 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.

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 Ohce.c
```

In this example, the string “*Ohce.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. The following sections show you how to get that set up.

5.1 Passing Command Line Arguments to *main()*

Your program must be able to process any number of command line arguments, which are the strings to be printed in reverse order. For example:

```
seansz@eustis:~$ ./a.out zebra giraffe
arbez effarig
seansz@eustis:~$ _
```

It's super easy to get command line arguments (like the strings "zebra" and "giraffe" in this example) into your program. You just have to change the function signature for the `main()` function you're writing in `0hce.c`. Whereas we have 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 where we ran `./a.out zebra giraffe`, `argc` would be equal to 3). `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]` stores the string "zebra," and `argv[2]` contains "giraffe."

5.2 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 array:

```
#include <stdio.h>

int main(int argc, char **argv)
{
    int i;
    for (i = 0; i < argc; i++)
        printf("argv[%d]: %s\n", i, argv[i]);
    return 0;
}
```

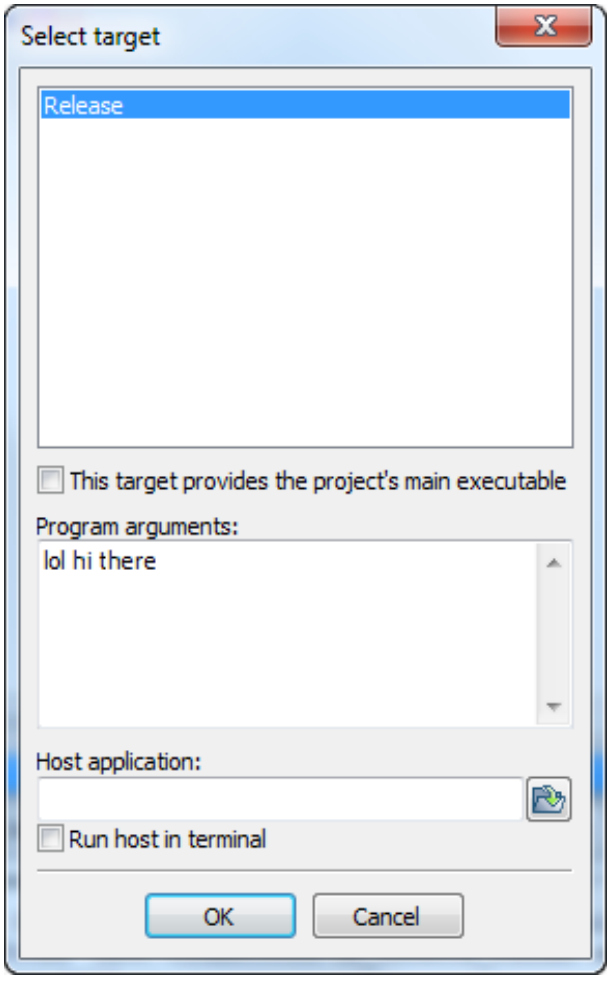
If we compiled that code into an executable file called `a.out` and ran it from the command line by typing `./a.out lol hi there!`, we would see the following output:

```
argv[0]: ./a.out
argv[1]: lol
argv[2]: hi
argv[3]: there!
```

5.3 Passing Command Line Arguments in CodeBlocks

You can set up CodeBlocks to automatically pass command line arguments to your program every time you hit *Build and run (F9)* within the IDE. Simply go to *Project → Set program's arguments...*, and in the box labeled *Program arguments*, type your desired command line arguments.

For example, the following setup that passes `lol hi there` as command line arguments to the program when compiled and run within CodeBlocks. (I don't think you need to check off the box next to "*This target provides...*." CodeBlocks seems to take care of that automatically after you hit "OK.")



Those three strings will be passed as `argv[1]`, `argv[2]`, and `argv[3]`. `argv[0]` will contain the name of your executable file automatically and should not be listed in the *Program arguments* box.

If you compile your program like this in CodeBlocks and later run your executable from the command line on Eustis (or even on your own machine), you will still need to supply fresh arguments for your program at the command line.

At any rate, don't forget to test your code on Eustis before submitting, even if you do most of your development in CodeBlocks. Throughout the semester, the safest way to test your program is by using the `test-all.sh` script on Eustis.

6. Dissecting the Contents of the *test_cases* Folder

Included with this project is a folder named "*test_cases*." In that folder, you'll find the following goodies:

1. The `test-all.sh` script (discussed below in Section 8, "Test Cases and the test-all.sh Script"). This little beast is made with love and can compile and run your code on all the test cases.

2. Test cases named `arguments01.txt` through `arguments06.txt`. These files contain command line arguments to pass to your program for testing. Instructions on how to use these files are given below in Section 7.1.

3. Test cases named `UnitTest07.c` through `UnitTest10.c`. A unit test is a test case that runs a specific segment of your code to check that it's producing the correct results. In this case, each unit test will call one of the three functions you're required to write for this assignment. These unit tests are source files that you will have to compile into a program along with your `0hce.c` source file and the `UnitTestLauncher.c` source file included in this folder. Instructions for working with these unit tests are given below in Section 7.2.

4. A source file named `UnitTestLauncher.c`. This source file has to be compiled into your program if (and only if) you are running a unit test. This process is described below in Section 7.2.

5. A folder named `sample_output`. This folder contains output files named `output01.txt` through `output10.txt`. These files show exactly what your output should look like for different test cases (provided that you do **not** open them in Notepad; see Section 2, "Important Note: Test Case Files Look Wonky in Notepad").

   The `arguments01.txt` through `arguments06.txt` test cases should produce the output given in `output01.txt` through `output06.txt`, and the `UnitTest07.c` through `UnitTest10.c` test cases should produce the output given in `output07.txt` through `output10.txt`. For details on how to ensure that your output is a 100% match for the expected output given in these text files, see Section 8 ("Test Cases and the test-all.sh Script"). Additional guides to compiling and running test cases in different contexts are included in Sections 11, 12, and 13.

7. Running and Compiling the Two Flavors of Test Cases

7.1 Running Standard Test Cases (*arguments01.txt* through *arguments06.txt*)

There are four ways to run the standard test cases (`arguments01.txt` through `arguments06.txt`) included with this assignment:

1. Compile your program at the command line, and then copy and paste the arguments from one of those files into the command prompt after "./a.out." You can use *diff* to check the difference between your output and the expected output. For example, `arguments01.txt` contains the text "`zebra giraffe  `." (Yes, there are spaces at the end of that text. Those spaces help `test-all.sh` make all of its output look pretty, and spaces at the end of your command line

input get ignored when the arguments are passed to `main()` anyway, so you needn't worry about them.) You can run that test case and check your output against `output01.txt` like so:

```
seansz@eustis:~$ gcc Ohce.c
seansz@eustis:~$ ./a.out zebra giraffe       > myoutput.txt
seansz@eustis:~$ diff myoutput.txt sample_output/output01.txt
seansz@eustis:~$ _
```

The lack of response from the *diff* command indicates that your output was correct. Yay!

2. Instead of copying and pasting text from `arguments01.txt`, you can also use the following magical incantations to make the Linux command line pass the contents of `arguments01.txt` to your program as command line arguments for you. (This works with the Mac terminal, too!)

```
seansz@eustis:~$ gcc Ohce.c
seansz@eustis:~$ ./a.out $(cat arguments01.txt) > myoutput.txt
seansz@eustis:~$ diff myoutput.txt sample_output/output01.txt
seansz@eustis:~$ _
```

Again, the lack of any grumbling from *diff* indicates that your output was correct.

3. If you're using CodeBlocks, you can copy and paste the contents of `arguments01.txt` (or any of the other argument-based test case files) into the program arguments interface as described above in Section 5.3, "Passing Command Line Arguments in CodeBlocks," and then hit the button to compile and run your program.

4. Alternatively, at a Linux or Mac command line (or in the new bash shell in Windows), you can run the `test-all.sh` script and let it do all the work for you. It will compile and run your source code on all the test cases, and give you a report of the results. For details on how to do that, see Section 8, "Test Cases and the test-all.sh Script."

7.2 Running Unit Test Cases (*UnitTest07.c* through *UnitTest10.c*)

The unit tests are a bit more tricky to get running. Here are your options:

1. The easiest option is to simply let the `test-all.sh` script do all the work for you, but you'll learn more if you try one of the following options as well. For details on how to use the `test-all.sh` script, see Section 8, "Test Cases and the test-all.sh Script."

2. You can run each unit test individually at the command line (Linux, Mac, or the new bash shell in Windows) by following the detailed instructions in Section 13, "Compiling and Running Unit Tests (Linux/Mac Command Line)."

3. You can run each unit test individually with CodeBlocks by following the detailed instructions in Section 11.1, "Running Unit Tests with CodeBlocks."

8. Test Cases and the test-all.sh Script

The multiple 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.

You can run the script on Eustis by placing it in a directory with `Ohce.c`, `Ohce.h`, the `sample_output` directory, and all the test case files, and then typing:

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

Please note that these test cases are not comprehensive. You should also create your own test cases if you want to test your code comprehensively. In creating your own test cases, you should always ask yourself, "How could these functions be called in ways that don't violate the function descriptions, but which haven't already been covered in the test cases included with the assignment?"

9. Function Requirements

In the source file you submit, `Ohce.c`, you must implement the following functions. You may implement auxiliary functions (helper functions) to make these work, as well, although that is probably unnecessary for this assignment. Please be sure the spelling, capitalization, and return types of your functions match these prototypes exactly.

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

> **Description:** Print each of the program's command line arguments (except `argv[0]`, which corresponds to the name of the executable file that you're running) in reverse order, as described above in Section 1 ("Overview"). There should be a single space between each reversed argument that you print, with a newline character ('\n') after the last reversed argument. Note that there should **not** be a space after the final argument that you print.
>
> If any of the arguments contain the string "echo" (in upper-, lower-, or mixed-case), you should also print "Echo, echo, echo!" (followed by a newline character, '\n') after printing the reversed command line arguments.
>
> If no arguments are passed to the program at the command line, your program should produce no output whatsoever.
>
> **Special Restriction:** You may only call the strlen() function once per command line argument. For details on this restriction, see Section 10.4, "Restrictions on Calling strlen()," below.
>
> **Special Restriction:** You cannot create any new arrays in your solution (whether 1D or 2D). That also means you cannot create any new strings. See Section 10.3, "No Array Creation."

Note: There are certain characters that cause Linux to do wonky things with command line arguments. Accordingly, we guarantee that the following characters will never appear in the command line arguments passed to your program: dollar sign (‘$’), opening parenthesis (‘(‘), closing parenthesis (‘)’), exclamation point (‘!’), hash (‘#’), ampersand (‘&’), backslash (‘\’), pipe (‘|’), semi-colon (‘;’), tilde (‘~’), asterisk (‘*’), single quotes (‘ and ’), double quotes (“ and ”), and redirection symbols (‘<’ and ‘>’). I think any other standard keyboard characters should be fair game, but if you have any questions about odd behaviors that you’re getting with non-alphabetic and non-numeric input characters, feel free to ask – although the answer is almost certainly, “Don’t worry about that.”

Return Value: You must return zero (0) from this function. Returning any value other than zero from `main()` could result in catastrophic test case failure when we grade your program.

```
double difficultyRating(void);
```

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

Return Value: A double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult).

```
double hoursSpent(void);
```

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

Return Value: A reasonable estimate (greater than zero) of the number of hours you spent on this assignment.

The fun continues on the following page!

10. Special Restrictions

You must adhere to the following special restrictions when writing this assignment.

10.1 No Global Variables

You may not use any global variables in this assignment. (A global variable is a variable that is declared outside of a function – e.g., just below a `#define` statement – and can therefore be accessed by any and all functions in your source file.)

10.2 No File I/O

Your program cannot read or write to any files.

10.3 No Array Creation

You cannot create any arrays (statically or dynamically) anywhere in the code you submit, other than the `argv` array. That means you cannot create new strings (since strings are just `char` arrays), you cannot create copies of any of the command line arguments (because that would involve the creation of a new string), and you definitely cannot create any 2D or 3D arrays.

10.4 Restrictions on Calling *strlen()*

If you call the `strlen()` function, you may only call it **once per command line argument**. That's because `strlen()` is a slow function that counts up all the letters in a string every single time you call it.

Note that the following code calls `strlen()` not just once, but **six times** before the program terminates, because `strlen()` is called once for each iteration of the for-loop (since the `i < strlen(str)` condition is checked at every single iteration), and so a for-loop like this, if used to process one of the command line arguments, would violate the restriction on calling `strlen()` only once per argument and could result in a loss of points:

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

int main(void)
{
   int i;
   char *str = "hello";

   for (i = 0; i < strlen(str); i++)  // BAD! Calling strlen() so many times!
      printf("%d...\n", i);

   return 0;
}
```

11. Compilation and Testing (CodeBlocks)

The key to getting multiple files to compile into a single program in CodeBlocks (or any IDE) is to create a project. Here are the step-by-step instructions for creating a project in CodeBlocks, which involves importing `Ohce.h` and the `Ohce.c` file you've created (even if it's just an empty file so far).

1. Start CodeBlocks.
2. Create a New Project (*File -> New -> Project*).
3. Choose "Empty Project" and click "Go."
4. In the Project Wizard that opens, click "Next."
5. Input a title for your project (e.g., "Ohce").
6. Choose a folder (e.g., Desktop) where CodeBlocks can create a subdirectory for the project.
7. Click "Finish."

Now you need to import your files. You have two options:

1. Drag your source and header files into CodeBlocks. Then right click the tab for **<u>each</u>** file and choose "Add file to active project."

   – or –

2. Go to *Project -> Add Files...*. Browse to the directory with the source and header files you want to import. Select the files from the list (using CTRL-click to select multiple files). Click "Open." In the dialog box that pops up, click "OK."

You should now be good to go. Try to build and run the project (F9).

11.1 Running Unit Tests with CodeBlocks

If you want to run a unit test (one of the test cases that is a source file), you must also do the following:

1. Add the unit test you want to run (e.g., `UnitTest07.c`) to your project. Note that you can only have one unit test in your project at a time. For example, if you import both `UnitTest07.c` *and* `UnitTest08.c`, the compiler will complain that you have multiple definitions for `main()`.

2. Add `UnitTestLauncher.c` to your project.

3. Find the line in `Ohce.h` that says `#define main __hide_ohce_main__` and ensure that it is **<u>not</u>** commented out. (I.e., remove the "`//`" from the beginning of the line).

If you want to go back to running normal test cases, you must remove the `UnitTestNN.c` and `UnitTestLauncher.c` files from your CodeBlocks project, and you must once again comment out the line in `Ohce.h` that says `#define main __hide_ohce_main__`. You **<u>must</u>** use "`//`" to comment out that line, with no space after the "`//`," like so: `//#define main __hide_ohce_main__`.

12. Compilation and Testing (Linux/Mac Command Line)

To compile your source file (`.c` file) at the command line:

```
gcc Ohce.c
```

By default, this will produce an executable file called `a.out`, which you can run with command line arguments like so:

```
./a.out zebra giraffe
```

If you want to name the executable file something else, use:

```
gcc Ohce.c -o Ohce.exe
```

...and then run the program with command line arguments like so:

```
./Ohce.exe zebra giraffe
```

Running the program could potentially dump a lot of text to the screen. If you want to redirect your program's output to a text file in Linux, it's easy. Just run the program using the following command, which will create a file called `whatever.txt` that contains the output from your program:

```
./Ohce.exe zebra giraffe > whatever.txt
```

Linux has a helpful command called `diff` for comparing the contents of two files, which is really helpful here since we've provided several sample output files. You can see whether your output matches ours exactly by typing, e.g.:

```
diff whatever.txt sample_output/output01.txt
```

If the contents of `whatever.txt` and `output01.txt` are exactly the same, `diff` won't have any output. It will just look like this:

```
seansz@eustis:~$ diff whatever.txt sample_output/output01.txt
seansz@eustis:~$ _
```

If the files differ, it will spit out some information about the lines that aren't the same. For example:

```
seansz@eustis:~$ diff whatever.txt sample_output/output01.txt
1c1
< zebra giraffe
---
> arbez effarig
seansz@eustis:~$ _
```

13. Compiling and Running Unit Tests (Linux/Mac Command Line)

Compiling a unit test at the command line requires you to compile three source files into one program. For example, if you want to compile `UnitTest07.c`, you must compile it along with your `Ohce.c` file and the `UnitTestLauncher.c` file we have included with this project.

Important: In order for this to work, you **must** ensure the `#define main __hide_ohce_main__` line in `Ohce.h` is **not** commented out. (I.e., remove the "`//`" from the beginning of that line).

Once you've done that, you can compile these three source files into a single program like so:

```
gcc Ohce.c UnitTest07.c UnitTestLauncher.c
```

By default, that gcc command will produce an executable file called `a.out`. You can now run the unit test like so:

```
./a.out
```

You can redirect the output of the unit test to a text file like so:

```
./a.out > whatever.txt
```

Linux has a helpful command called `diff` for comparing the contents of two files, which is really helpful here since we've provided several sample output files. You can see whether your output matches ours exactly by typing, e.g.:

```
diff whatever.txt sample_output/output07.txt
```

If the contents of `whatever.txt` and `output01.txt` are exactly the same, `diff` won't have any output. It will just look like this:

```
seansz@eustis:~$ diff whatever.txt sample_output/output07.txt
seansz@eustis:~$ _
```

If the files differ, it will spit out some information about the lines that aren't the same, as we saw on the previous page.

13.1 An Important Note About Messing with Ohce.h

Super Important: After running unit tests, before you can run standard test cases again, you will need to comment out the `#define main __hide_ohce_main__` line in `Ohce.h`. You **must** use "`//`" to comment out that line, with no space after the "`//`," like so: `//#define main __hide_ohce_main__`.

Uncommenting that line essentially kills your `main()` function in `Ohce.c` and allows one of the unit test functions to take over the program. Commenting out that line again brings your `main()` function back to life.

14. Getting Started: A Guide for the Overwhelmed

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!”

Don’t panic! There are plenty of TA office hours where you can get help, and here’s my general advice on starting the assignment:

1. First and foremost, start working on this assignment early. Nothing will be more frustrating than running into unexpected errors or not being able to figure out what the assignment is asking you to do on the day that it is due.

2. Secondly, glance through all the section headings in this PDF to get an idea of what kind of information it contains.

3. Thirdly, read over the PDF to get an idea of what the assignment is asking you to do. Section 1, “Overview,” will be the most important part to start out. Some sections won’t make sense right away, and it’s okay to skim them if that’s the case, but don’t blow them off entirely.

4. After you’ve read through the PDF, but before you start programming, open up the test cases and sample output files included with this assignment and trace through a few of them to be sure you have an accurate understanding of what you’re supposed to be doing. Refer back to relevant sections of the PDF (the ones you might have skimmed in your first pass through the PDF) to clarify any questions you might have about those test cases.

5. Once you’re ready to begin coding, start by creating a skeleton `Ohce.c` file. Add a header comment with your name and NID, add one or two standard `#include` directives, and be sure to `#include "Ohce.h"` from your source file. Then copy and paste each functional prototype from `Ohce.h` into `Ohce.c`, transform those prototypes into functions by removing the semicolon and setting up curly braces for each function signature, and set up all those functions to return dummy values (some arbitrary `int` or `float`, as appropriate).

6. Set up your `main()` function to take arguments as described above in Section 5.1, “Passing Command Line Arguments to main().” Be sure your `main()` function returns zero.

7. Test that your `Ohce.c` source file compiles. Do this before you’ve even taken a stab at making the functions work! If you’re at the command line on a Mac or in Linux, your source file will need to be in the same directory as `Ohce.h`, and you can test its ability to compile like so:

   ```
   gcc Ohce.c
   ```

   Alternatively, you can try compiling it with one of the unit test source files using the instructions set forth in Section 13, “Compiling and Running Unit Tests (Linux/Mac Command Line).”

   If you’re using CodeBlocks, you’ll want to start by creating a project using the instructions above in Section 11, “Compilation and Testing (CodeBlocks).” Import `Ohce.h` and your new `Ohce.c` source file into your project, and get the program compiling and running before you

move forward. (Note that CodeBlocks is the only IDE we officially support in this class.) Instructions for compiling unit tests in CodeBlocks are included in section 11.1, "Running Unit Tests with CodeBlocks."

8. Once you have your project compiling, go back to Section 9, "Function Requirements," and read through the function requirements. Do some brainstorming on how you want to solve the problem. Don't even try to write any code yet. Just assume you'll be able to get your `main()` function to read all the command line arguments one by one, and think about how you'll print each argument in reverse order. At this point, you might also want to revisit Sections 1 ("Overview") and 5 ("Guide to Command Line Arguments").

9. Once you have an idea of how you want to write `main()` (even if it feels a bit vague at first), dive in fearlessly. Be bold! Don't hesitate to run code that you think might not work; you won't break anything.

15. General Tips for Working on Programming Assignments

Here's some general advice that might serve you well in this and all remaining assignments for the course:

1. In general, you'll want to stop to check whether your code compiles after every one or two significant blocks or chunks of code you throw down. That's rather moot in this assignment, since there isn't much actual code to write. However, throughout the semester, if you frequently stop to check whether your code is compiling (rather than writing 30-100 lines of code before checking whether any of it is even syntactically correct), you will save yourself a lot of headaches.

2. If you get stuck while working on an assignment, draw diagrams on paper or a whiteboard. Make boxes for all the variables in your program. Trace through your code carefully, step by step, using these diagrams.

3. You're bound to encounter errors in your code at some point. Use `printf()` statements liberally to verify that your code is producing the results you think it should be producing (rather than making assumptions that certain components are working as intended). You should get in the habit of being immensely skeptical of your own code and using `printf()` to provide yourself with evidence that your code does what you think it does.

4. If you encounter a segmentation fault, you should always be able to use `printf()` and `fflush()` to track down the *exact* line you're crashing on.

5. When you find a bug, or if your program is crashing on a huge test case, don't trace through hundreds of iterations of some for-loop to track down the error. Instead, try to cook up a very small, new test case (as few lines as possible) that causes your code to crash. The smaller the test case that you have to trace through, the easier your debugging task will be.

6. You will eventually want to read up on how to set break points and use a debugger. Some helpful Google queries might be: CodeBlocks debugger and gdb debugging tutorial.

16. Deliverables (Submitted via Webcourses, Not Eustis)

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

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

```
gcc Ohce.c
./a.out zebra giraffe
```

Be sure to include your name and NID as a comment at the top of your source file.

17. Grading

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

65%	correct output for test cases (not all of which have been release with this project)
10%	implementation details (manual inspection of your code)
5%	`difficultyRating()` is implemented correctly
5%	`hoursSpent()` is implemented correctly
5%	source file is named correctly (`Ohce.c`); spelling and capitalization count
10%	adequate comments and whitespace; source includes student name and NID

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.

Additional points will be awarded for style (appropriate commenting and whitespace) and adherence to implementation requirements. For example, the graders might inspect your `main()` function to see whether it meets the special `strlen()` requirement listed in the function description above.

Start early. Work hard. Good luck!