

# Programming Assignment #4: Char Processing and File I/O

COP 3223H, Fall 2019

**Due:** Wednesday, November 6, *before* 11:59 PM

## Table of Contents

1. Super Important: Initial Setup ( <i>Same As Usual</i> ).....	2
2. Note: Test Case Files Might Look Wonky in Notepad.....	2
3. Assignment Overview.....	2
4. Overview: Letter Shifting.....	3
5. Overview: Caesar Cipher.....	4
6. Function Requirements.....	5
7. Special Restrictions ( <i>Important!</i> ).....	8
8. Style Restrictions ( <i>Important!</i> ).....	9
9. Running All Test Cases on Eustis.....	10
10. Running Test Cases Individually.....	11
11. Deliverables (Submitted via Webcourses, Not Eustis).....	12
12. Grading.....	12

## Deliverables

*assignment04.c*

**Note!** The capitalization and spelling of your filename matter!

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

## 1. Super Important: Initial Setup (*Same As Usual*)

At the very top of your *assignment04.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 "assignment04.h"
```

Yes, that needs to be in “double quotes” instead of <angled brackets>, because *assignment04.h* is a local header file that we’ve included with this assignment, rather than a standard system header file.

The *assignment04.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 (*stdio.h*, *ctype.h*, etc.).

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 "assignment04.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 *assignment04.h* (provided in the ZIP file for this assignment) in the same folder as your *assignment04.c* file any time you want to compile your code.

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

(*You already know this. You can skip reading it.*) 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

This assignment is designed to help you solidify your understanding of character processing and file I/O in C. 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 and to work through the file I/O lab activity. (Note that a solution to that activity is already posted.)

## 4. Overview: Letter Shifting

One of the functions you have to write in this assignment, *shift\_letter()*, takes two parameters: a character (*ch*), and an integer (*offset*), and returns the letter you get if you shift that character over in the alphabet by *offset* number of letters. For example, if you call *shift\_letter()* with an *offset* of 1, the function should return the next letter in the alphabet:

```
shift_letter('a', 1) returns 'b'
shift_letter('b', 1) returns 'c'
shift_letter('c', 1) returns 'd'
...
shift_letter('x', 1) returns 'y'
shift_letter('y', 1) returns 'z'
shift_letter('z', 1) returns 'a'
```

Notice that if shifting a letter causes us to go off the end of the alphabet, we have to wrap back around to the beginning of the alphabet.

Similarly, calling *shift\_letter()* with some character (*ch*) and an *offset* of 2 returns the alphabetic character that comes two letters after *ch* in the alphabet (again, wrapping back around to the beginning of the alphabet if necessary):

```
shift_letter('a', 2) returns 'c'
shift_letter('b', 2) returns 'd'
shift_letter('c', 2) returns 'e'
...
shift_letter('z', 2) returns 'b'
```

Note that *offset* doesn't necessarily have to be less than 26. If our *offset* is 26, that brings us full circle, and we return the letter passed to the function:

```
shift_letter('a', 26) returns 'a'
shift_letter('b', 26) returns 'b'
shift_letter('c', 26) returns 'c'
...
shift_letter('z', 26) returns 'z'
```

Similarly, if *offset* is 27, that has the same effect as if *offset* had been 1 (or 53, 79, 105, etc.):

```
shift_letter('a', 27) returns 'b'
shift_letter('b', 27) returns 'c'
shift_letter('c', 27) returns 'd'
...
shift_letter('z', 27) returns 'a'
```

Furthermore, the *offset* value could be negative, in which case we have to shift *backwards* in the alphabet by that number of characters (wrapping back around to the end of the alphabet if necessary). For example:

```
shift_letter('a', -1) returns 'z'
```

```
shift_letter('b', -1) returns 'a'
shift_letter('c', -1) returns 'b'
...
shift_letter('z', -1) returns 'y'
```

**Important Notes:** If your function receives a capital letter, it should return a capital letter. If it receives a lowercase letter, it should return a lowercase letter. If your function receives a non-alphabetic character, it should simply return that character without applying any offset.

**Hint:** (Highlight and/or copy and paste to reveal.)

**Another Hint:** (Highlight and/or copy and paste to reveal.) There are at least two ways to implement this:

1. 

```
def shift_letter(letter, offset):
    """Shift letter by offset"""
    if letter.isupper():
        start = ord('A')
    elif letter.islower():
        start = ord('a')
    else:
        return letter
    shift = offset % 26
    new_letter = chr((ord(letter) - start + shift) % 26 + start)
    return new_letter
```
2. 

```
def shift_letter(letter, offset):
    """Shift letter by offset"""
    if letter.isupper():
        start = ord('A')
    elif letter.islower():
        start = ord('a')
    else:
        return letter
    shift = offset % 26
    new_letter = chr((ord(letter) - start + shift) % 26 + start)
    return new_letter
```

## 5. Overview: Caesar Cipher

A [Caesar cipher](#) is a mechanism for encrypting a body of text. With a Caesar cipher, every letter in some body of text gets shifted by the same number of characters in the alphabet (using the shifting mechanism described above). To decrypt a body of text that has been encrypted using a Caesar cipher, you simply have to shift each letter by the same offset, but in the opposite direction. The *offset* we use to encrypt some text is called the *key*.

For example, suppose we have a text file with the following contents:

*My dog-powered robot helped me knit this amazing sweater.*

If we apply a Caesar cipher with *key* = 1, then all the letters get shifted over by one, and we get the following encrypted text:

*Nz eph-qpxfsfe spcpu ifmqfe nf loju uijt bnbajoh txfbufs.*

To decrypt that text, we would then apply a Caesar cipher with *key* = -1.

Note that your Caesar cipher function should **only** modify alphabetic characters. Non-alphabetic characters should remain unchanged by your Caesar cipher function.

## 6. Function Requirements

You must implement the following functions in a file named *assignment04.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.

You may assume that will never pass invalid pointers (such as NULL) to these functions.

```
int is_consonant(char ch);
```

**Description:** This function takes a single character (*ch*) as its only argument and returns 1 if *ch* is a consonant (any alphabetic character other than a vowel. Otherwise, return 0. Note that the letter Y is not a vowel.

For example:

```
is_consonant('x') returns 1
is_consonant('y') returns 1
is_consonant('u') returns 0
is_consonant('#') returns 0
```

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

**Return Value:** Return 1 or 0 as described above.

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

```
int is_terminating_punctuator(char ch);
```

**Description:** This function takes a single character (*ch*) as its only argument and returns 1 if *ch* is one of the following three punctuators that are typically used to terminate English sentences: ‘.’, ‘!’, or ‘?’ (period, exclamation point, or question mark). Otherwise, return 0.

For example:

```
is_terminating_punctuator('.') returns 1
is_terminating_punctuator('y') returns 0
is_terminating_punctuator('4') returns 0
is_terminating_punctuator('#') returns 0
```

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

**Return Value:** Return 1 or 0 as described above.

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

```
char shift_letter(char ch, int offset);
```

**Description:** This function takes two arguments: a character (*ch*) and an integer (*offset*). If *ch* is an alphabetic character, the function returns an alphabetic character using the shifting mechanism described above in Section 4, “Overview: Letter Shifting” (pg. 3). If *ch* is a non-alphabetic character, this function simply returns *ch* without any change whatsoever.

**Input Note:** When we grade your program, we will only use *offset* values on the range -250,000 through 250,000.

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

**Return Value:** This function should return a single character as described above.

**Related Test Cases:** *testcase03a.c* through *testcase03f.c*

```
int cipher(char *filename, int key);
```

**Description:** This function takes two arguments: a string (*filename*), which is the name of a file to open in read-only mode (“r”), and an integer (*key*). This function must open the file specified by *filename* and print the resulting text after applying a Caesar cipher with the specified *key*. You will most likely want to call *shift\_letter()* on each character you read from the file, passing *key* as the *offset* value for that function. For details, see Section 5, “Overview: Caesar Cipher” (above on pg. 4).

As mentioned in the section on Caesar ciphers, this function should only modify alphabetic character. Non-alphabetic characters should be printed to the screen without any change.

If the file opens successfully, you must close it before returning from this function.

If *filename* refers to a file that does not exist, or if the file cannot be opened for any reason, this function should print the following error message (followed by a newline character) and return -1:

```
Could not open file. Womp womp. :(
```

**Input Notes:** We will never pass a NULL pointer for the *filename* string, but *filename* could refer to a file that does not exist. When we run your program for grading, we will only use *offset* values on the range -250,000 through 250,000.

**Output:** Print the text that results from applying a Caesar cipher to the entire input text file. (For examples, see the test cases released with this assignment.) If the file cannot be opened, print the error message described above.

**Return Value:** Return 0 if the input file opens successfully. Otherwise, return -1.

**Related Test Cases:** *testcase04a.c* through *testcase04f.c*

```
int print_first_word_beginning_with_character(char *filename, char ch);
```

**Description:** This function takes two arguments: a string (*filename*), which is the name of a file to open in read-only mode (“r”), and a char (*ch*). This function must open the file specified by *filename* and then

print the first word it finds that begins with the character specified by *ch* (followed by a newline character). For the purposes of this function, we consider any consecutive non-space characters to be part of the same word.

For example, suppose we call this function with *ch* = 'd', and the input file we open has the following text:

```
My dog-powered robot helped me knit this amazing sweater!
```

The function call should produce the following output (followed by a newline character):

```
dog-powered
```

Similarly, if we call that function with *ch* = 's' and the same input file as above, it should produce the following output:

```
sweater!
```

Notice that the exclamation point (!) is part of the output, since all consecutive non-space characters are considered to be part of the same word.

The function should be case sensitive, too. So, if we call this function with *ch* = 'm' and the same input file as above, it should print the first word that begins with a lowercase 'm' (not the word that begins with an uppercase 'M'):

```
me
```

If there is no word that begins with *ch* in the file, simply print the following message (followed by a newline character) and return -1:

```
No such word in input file. :(
```

If *filename* refers to a file that does not exist, or if the file cannot be opened for any reason, this function should print the following error message (followed by a newline character) and return -1:

```
Could not open file. Womp womp. :(
```

If the file opens successfully, you must close it before returning from this function.

**Input Notes:** We will never pass a NULL pointer for the *filename* string, but *filename* could refer to a file that does not exist. The character in *ch* is always guaranteed to be a visible, non-space character, but it won't necessarily be an alphabetic character; it could be a digit or other symbol, such as '\$' or '?'.

**Output:** The function produces output as described above.

**Return Value:** Return 0 if the input file opens successfully and we find a word beginning with *ch*.

Otherwise, return -1.

**Related Test Cases:** *testcase05a.c* through *testcase05f.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:** *testcase06.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:** *testcase07.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:** *testcase08.c*

## 7. Special Restrictions (**Important!**)

You must abide by the following restrictions in the *assignment04.c* file you submit. Failure to abide by any one of these restrictions could result in a catastrophic loss of points.

- ★ (**Restriction Lifted!**) As in Assignment #3, you can use nested loops if you want.



- ★ (**New Restriction!**) Please do not create any strings or arrays in this assignment (other than the filename strings passed to the functions above).
- ★ (**Modified Restriction!**) For this assignment, you are allowed to read files (duh), but you are not permitted to write to any files. 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 must occur at the *top* of a function.
- ★ All variables must be declared inside your functions or declared as function parameters.
- ★ Do not make calls to C's *system()* function.
- ★ Do not use *goto* statements in your code.
- ★ Do not write malicious code, including code that attempts to open files it shouldn't be opening, whether for reading or writing. (I would hope this would go without saying.)
- ★ No crazy shenanigans.

## 8. Style Restrictions (**Important!**)

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

- ★ Any time you open a curly brace, that curly brace should start on a new line, and it should be indented to align properly with the line above it. See my code in Webcourses for examples.
- ★ Any time you open a new code block, indent all the code within that code block one level deeper than you were already indenting.
- ★ Be consistent with the amount of indentation you're using, and be consistent in using either spaces or tabs for indentation throughout your source file. If you're using spaces for indentation, please use at least two spaces for each new level of indentation, because trying to read code that uses just a single space for each level of indentation is downright painful.
- ★ Please always use code blocks with if/else statements and loops, even if there's just one line of code within that code block.
- ★ Please avoid block-style comments: */\* comment \*/*
- ★ Instead, please use inline-style comments: *// comment*
- ★ Always include a space after the *//* in your comments: *// comment* instead of *//comment*
- ★ Any libraries you *#include* should be listed *after* the header comment at the top of your file that includes your name, course number, semester, NID, and so on.
- ★ Use end-of-line comments sparingly. Comments longer than three words should always be placed above the lines of code to which they refer. Furthermore, such comments should be indented to properly align

with the code to which they refer. For example, if line 16 of your code is indented with two tabs, and line 15 contains a comment referring to line 16, then line 15 should also be indented with two tabs.

- ★ Please do not write overly long lines of code. Lines must be fewer than 100 characters wide.
- ★ Avoid excessive consecutive blank lines. In general, you should never have more than one or two consecutive blank lines.
- ★ Please leave a space on both sides of any arithmetic operator you use in your code. For example, use  $(a + b) - c$  instead of  $(a+b)-c$ .
- ★ When defining a function that doesn't take any arguments, put *void* in its parentheses. For example, define a function using *int do\_something(void)* instead of *int do\_something()*.

## 9. 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 *assignment04.c*, *assignment04.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 (*assignment04.c*, *assignment04.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 assignment04
```

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.

## 10. 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 *assignment04.c* file.
- b. Compile *both* your *assignment04.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 assignment04.c testcase01a.c
```

- c. Run the program as usual:

```
./a.out
```

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

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

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

2. Following is the less ideal way to run a single test case. However, this is what you'll most likely want to do if you want to write your own *main()* function to test your code:
  - a. Comment out the *#include "assignment04.h"* line in your *assignment04.c* source file.
  - b. Copy and paste the *main()* function from one of the test case files (such as *testcase01a.c*) into your *assignment04.c* source file, or write your own *main()* function for testing.

- c. Compile *assignment04.c* as usual:

```
gcc assignment04.c
```

- d. Run the program as usual:

```
./a.out
```

- e. When you're finished, don't forget to un-comment the *#include "assignment04.h"* line in your *assignment04.c* file so that your code will be compatible with our grading infrastructure!

## 11. Deliverables (Submitted via Webcourses, Not Eustis)

Submit a single source file, named *assignment04.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 "assignment04.h"* in your *assignment04.c* code.

Do not submit additional source files, and do not submit a modified *assignment04.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 assignment04.c testcase01a.c  
./a.out
```

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

## 12. 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.)   |
| 10% | 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% | Code includes useful and appropriate comments. See my commenting guidelines in the notes in Webcourses from <a href="#">Monday, Sept. 30</a> . Points may also be deducted from this category for missing required information in the header comment(s), or for incorrect placement of the header comment(s), or for naming the source file incorrectly. |

20%      Code compiles without warnings (and without the use of typecasting to artificially suppress warnings).

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

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

Please also note that failure to abide by any the special restrictions listed above on pg. 8 (Section 7, "Special Restrictions") could result in a catastrophic loss of points.

*Start early. Work hard. Good luck!*