**CS 305 Lab 3: Introduction to structs, files, and C preprocessor**

The purpose of this lab is to give you experience with structs, files, and the C preprocessor. This lab has a total of 100 points (30 points from pre-lab and 70 points from lab).

**Objectives**

Upon completion of this laboratory exercise, you will be able to do the following:

* Write functions that take pointers to return multiple values from a function
* Define and use structs
* Read from and write to files
* Use C pre-processor directives

*If you would like a lab partner, please ask the instructor and one will be assigned to you. To work with a partner you should have the MS Teams desktop client downloaded so that you can do screen sharing. You are welcome to do this lab exercise alone if you would prefer.*

**Part 1: Log in (both partners)**

* Start up your C programming environment.
* The Driver should set up an MS Teams screen-share if you are working with a partner.
* Make a new lab3 directory for this lab’s exercises (see Lab 1 for instructions if you need more details).
* Download the starter code for this lab from Moodle and unzip it into your lab3 directory. *Note*: the Linux terminal command to unzip a zip file looks like this, though you can use a graphical tool if you prefer:

unzip cs305\_lab3\_starter.zip

* There should now be five files under lab3.

**Part 2: Use pointers to return multiple values from a function and using < to read from a file instead of stdin (one partner driving)**

Open count\_chars.c in your preferred text editor.

The file count\_chars.c contains a main function that is defined to count vowels, consonants, punctuation characters, and total number of characters across all lines of text in a file. It does this by reading lines of a file one-by-one, and then calling a helper-function, count\_em, which is defined above main. The purpose of the count\_em function is to count those respective kinds of characters on a *single* string of text.

The count\_em function needs to return *four* values to its caller:

* the total number of characters
* the number of vowels
* the number of consonants
* the number of punctuation characters

Because C only allows a function to have a single return-value, count\_em will get the effect of four return-values using pointers as follows:

* The **actual** return value will give the total number of characters.
* The caller will create variables to receive each of the other three return-values, and passes pointers to those variables as parameters to the function call. The called function "writes through" the pointers, in effect, returning the additional values to the caller.

(By the way, this is how the function scanf works – you pass a pointer to the variable to which to store the data.)

The header for the count\_em function is:

int count\_em(char \*str, int \*vow, int \*cons, int \*punct)

where str is the string to be counted, and the last three parameters are "write-through" pointers or **reference** parameters. After each call, the main function will add those values into its counters.

Complete the loop in the count\_em function. (Do not modify main.) Counting should be done as follows:

* The characters that are *vowels* are upper- and lower-case 'a', 'e', 'i', 'o' and 'u'.
* A *consonant* is any upper- or lower-case letter that is not a vowel.
* The *punctuation* characters are period ('.'), comma (','), colon (':'), semicolon (';') question mark ('?'), exclamation point('!'), and left and right parentheses ('(' and ')').

Hints:

1. A switch statement has been written for you.

Compile count\_chars.c:

gcc –o count\_chars count\_chars.c

You can test your program on any text file, as in:

./count\_chars < caseCons.txt

The < means to use the file in place of stdin. This is a handy way to re-test the same input for a program without having to re-type it – just put what you would type to the screen in a file.

**Checkpoint 1 [20 points]: Take a screenshot of you running count\_chars with caseCons.txt as an input file. Make sure both partners have the screenshot and the modified count\_chars.c for the final submission. If during class, send the instructor a message to let him know you’re done checkpoint 1; ask him to check it if you’re not sure about your solution.**

**Part 3: Defining a struct, reading from a text file, writing to a text file (switch driver and navigator)**

1. Compile player.c:

gcc –o player player.c

What error messages do you get? \_\_\_\_\_\_\_\_No Struct for player\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What do you think needs to be defined in player.c? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Player struct\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2. Open player.c in your text editor. Examine the code.

In general, this program:

* opens an input file that stores the game state (players); think of this as loading a saved game state into a video game
* stores the data from the input file into an malloc’ed array of player objects
* finds the player with the most points
* writes the player’s data with the most points to the output file

3. First, define the player struct so it has the following members:

* name (char \*)
* x\_loc (int)
* y\_loc (int)
* points (int)

Once you have finished the struct definition, try to compile it again:

gcc –o player player.c

Hopefully, you now have a compiled program, but it is not fully functional.

4. Complete the function find\_max. It takes an array of player objects and the length of the array as parameters. It should return the *index* of the player who has the most points. Part of the function is already completed for you.

5. Go to the main function. Around line 100, after the line FILE \*out, open the file with name stored in argv[2] for writing and store it to the file pointer out. This should be one line of code.

6. Now, after checking that the out file stream is not NULL, write the data about the highest scoring player to the out file pointer. fprintf works just as printf, except you put the name of the stream as the first argument. For example, to write “Highest Scoring Player:\n” to the stream out, you write:

fprintf(out, "Highest Scoring Player:\n");

7. Compile player.c and run the executable:

./player input.txt high\_scores.txt

The data in the output file should be formatted as follows:

Highest Scoring Player:

Name: Jill

X position: 40

Y position: 20

Points: 1000

**Checkpoint 2 [30 points]: Take a screenshot of you running player with input.txt as an input file and highscores.txt as an output file. Make sure both partners have the screenshot, the modified player.c, highscores.txt, and this lab document edited with your question answers for the final submission. If during class, send the instructor a message to let him know you’re done checkpoint 2; ask him to check it if you’re not sure about your solution.**

**Part 4: Using the preprocessor (switch driver and navigator)**

1. Open list.c with your text editor. Examine the code for sorting an array of integers. You will see a bunch of #defines in the code. Ignore those for now and look at main and sort. This is an implementation of selection sort.

3. Run the C preprocessor on the file list.c without actually running the compiler and put the results into list2.c. Type at the shell prompt:

cpp list.c > list2.c

4. Examine the file list2.c. This is the output of the preprocessor.

What are the first four lines of list2.c?

|  |
| --- |
| # 1 "list.c"  # 1 "<built-in>"  # 1 "<command-line>"  # 31 "<command-line>" |

Compile and run **both** programs.

gcc –o list list.c

gcc –o list2 list2.c

5. Now, execute both programs:

./list

./list2

Does the output match? **YES<-** NO

6. Now, let’s look at list.c again. At the top, you’ll see two global variables (AUTHOR and ARRAYCONTENTS) that get defined if they are not already defined. You might ask, where would these values be defined elsewhere? Well, they could be defined in other code files that are compiled together with this program, should the program comprise several files. They could also be defined at the command line.

Try this:

gcc –g –D'AUTHOR="Tammy"' –o listTammy list.c

Run:

./listTammy

Who is the author? \_\_\_\_\_\_\_\_\_Tammy\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Run:

./list

Who is the author? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Captain Fantastic\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

7. The –D flag in gcc defines variables for the program. This can be handy if you want to test different inputs from the command line, rather than changing the values in the code and recompiling before running it each time.

8. Try setting the variable for ARRAYCONTENTS from the command line.

gcc -g -D'ARRAYCONTENTS={12, -6, 4, 10}' -o listarray list.c

Run your listarray program. What does your program print?

|  |
| --- |
| -6  4  10  12 |

9. Another helpful thing we can do with preprocessor directives is to run code for debugging purposes. Examine the code list.c again. At the top, you will see #define DEBUG commented out. Uncomment that, compile the code, and rerun it.

gcc –o list list.c

./list

What is printed just under the “Sorting program by … Captain Fantastic”? (just write down the next line)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Value of i: 0, array value at position 0 is 3\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Examine the code list.c. You’ll see that there are print statements that are only executed if DEBUG is true. This is a great way to debug your programs without adding/deleting a bunch of print statements. Just turn them on and off with the macro. Now, let’s do that.

10. Comment out the #define DEBUG line again. Save the file. Now, we’ll set that variable at the command line when compiling:

gcc -D'DEBUG=1' -o listdebug list.c

Run it:

./listdebug

Does the debugging printing happen? **YES<-** NO

Now, create an executable without the debug set:

gcc –o list list.c

Run it:

./list

Does the debugging printing happen? YES **NO**

11. Finally, look at list.c again. You’ll see a swap macro. In C, you can define function-like definitions as macros and the code for swap(arr,a,b) will get replaced by the code that follows when the code compiles. In this case, anytime the code swap(arr,a,b) appears, it will be replaced by:

int temp = arr[a];

arr[a] = arr[b];

arr[b] = temp;

These macros are handy if you want to create “shorthand” code blocks.

What do you see at the end of the lines for the swap macro definition? \_\_\_\_\_\_\_\_\_\_\_\\_\_\_\_\_

Macros must be single-line definitions, so the symbol in the above answer allows macros to be defined on multiple lines.

**Checkpoint 3 [20 points]: Make sure both partners have a copy of this lab document edited with your question answers for the final submission. If during class, send the instructor a message to let him know you’re done checkpoint 3; ask him to check it if you’re not sure about your solution.**

You are done Lab 3!

* Collect a copy of this document (edited with your answers), as well as your code for count\_chars.c and player.c, along with your output file highscores.txt and your screenshots from checkpoints 1 & 2.
* If you worked with a partner, also collect a text file named partner.txt with both your name and your partner’s name.
* Submit all the collected files to the “Lab 3” activity on Moodle. If you worked with a partner, both partners should submit this.

*Acknowledgement*

This lab was written by Dr. Tammy VanDeGrift.