

Due: Checkpoint 1 - Wednesday, October 21st, 2008 at 10:00pm Central Time
Checkpoint 2 - Wednesday, October 28th, 2008 at 10:00pm Central Time

Please read the entire document before starting this MP. Certain sections may only make sense after you read all the others.

Program Description

In this MP, you need to implement 2 separate C programs, one for each checkpoint. The first program, to be submitted for MP3.1, reads characters from the user, one at a time, counts the number of letters, digits, white spaces and special characters and displays the result to the user. The second program, to be submitted for MP3.2, reads 2 strings from the user, a substring and a main string, and counts the number of separate occurrences of the substring in the main string.

Given Code

You will be given 2 different skeleton files named mp3.1.c and mp3.2.c, which contain necessary declarations, an empty main function, empty subroutines, an implemented function PrintMsg and some comments. You should use PrintMsg to print any messages. For example, use PrintMsg(0) to print message 0. Note that the definition of the function PrintMsg is different in mp3.1.c and mp3.2.c. The given code for mp3.1.c contains the declarations of global variables CountLetter, CountDigit, CountSpace and CountSpecialChar. The file mp3.2.c contains the global variable CntInstances.

Tasks: Checkpoint 1

For checkpoint 1, you need to modify the file mp3.1.c and implement the Count() function, which reads in a character one at a time, using getchar(), until the 'Enter' key or '\n' has been entered. At the end of the user input, this function displays the total number of letters (both lowercase and uppercase), number of digits (0-9), white spaces (number of single spaces and tabs) and special characters (all other characters) that were entered by the user. The messages to be displayed are shown in Table 1. Use the appropriate message code to print a message. In the main function, you should include a function call to Count(), and any initializations, if necessary.

Tasks: Checkpoint 2

For checkpoint 2, you need to modify the file mp3.2.c and implement the CheckPattern() function, that reads in 2 strings from the user, a substring and a main string, using the gets() function. Assume that both the strings can contain a maximum of 50 characters, and that the substring is always smaller than or equal to the length of the main string. The function should count the number of separate occurrences of the substring in the main string, in both lowercase and uppercase, and display the count. For example: if the substring is "MP3", and the main string is "Mp3.1 and mP3.2 are easy", then the count should be two. The main function should include a function call to CheckPattern(), and any initializations, if necessary.

Table 1 and 2 give the messages corresponding to each message code implemented in the PrintMsg function in mp3.1.c and mp3.2.c respectively.

Table 1: PrintMsg in mp3.1.c

Message Code	Message that gets printed
0	Enter the string:
1	The number of alphabets in the string is: <CountLetter>
2	The number of digits in the string is: <CountDigit>
3	The number of white spaces in the string is: <CountSpace>
4	The number of special characters in the string is: <CountSpecialChar>

Table 2: PrintMsg in mp3.2.c

Message Code	Message that gets printed
0	Enter the substring to be checked for:
1	Enter the main string:
2	The first string is contained in the second string <CntInstances> times

Testing

Before you can test your program you must create an executable. For checkpoint 1, you may do so by typing “gcc -Wall -g -ansi mp3.1.c -o mp3.1” at the terminal, when you are in the same directory as your code. For checkpoint 2, it should be “gcc -Wall -g -ansi mp3.2.c -o mp3.2”. If there is no error found, it will generate a binary executable named mp3.1 or mp3.2. Otherwise you need to fix all errors and warnings that are reported. If you make the mistake of putting mp3.1.c after the -o argument, your source will be overwritten. Keep copies and be careful. You may execute the binary by simply typing “./mp3.1” or “./mp3.2” at the terminal. We will provide gold binary files for both checkpoints and some test cases to help you test your program. You should use the gold binary to generate the standard output and compare it with the output from your own program. Your output must match the gold output **EXACTLY**.

Specifics

- Your program files must be called **mp3.1.c** and **mp3.2.c**.
- You must implement the required functions. You are not required to use additional functions. You may if you think they are helpful.
- You must use the predefined message codes.
- You may not include any additional header files (other than the ones already included).
- You may not use any additional global variables (other than the ones provided).
- Your code should be in good coding style, with proper indentation on each line to indicate functional blocks and explanatory comments. A brief header describing the program and headers for main function and support functions are important.
- Your code must pass compilation. You will not receive any functionality points on a code that fails to generate a binary. You will also lose points for compiler warnings.

Handing in

Turn in your code using the ECE190 handin script. Your files must be called mp3.1.c and mp3.2.c. We will NOT grade files with any other name. To hand in the code, first login to any EWS machine and type ece190. Change to the directory containing your code and type "handin -MP 3.1 mp3.1.c" or "handin --MP 3.2 mp3.2.c". You may hand in as many times as you like, but only the last submission will be time stamped and graded. Note that only your .c file is submitted.

Grading

Your code functionality will be graded by a computer script. Use input/output prompting messages provided in the tables and test your program using our test cases. You must ensure that your program terminates properly and does not loop forever.

- **Functionality (65%)**
Program terminates correctly. Program reads each character and increments appropriate counter correctly. Program reads each string and counts instances of substring correctly. Uses predefined global variables as counters. Uses predefined message codes. Checks lowercase and uppercase.
- **Style (20%)**
10% - Program compiles without errors or warnings.
10% - Good and consistent indentations.
- **Documentation (15%)**
5% - Introductory paragraph explaining program's purpose and approach.
10% - Good comments and variable names.