CSCI 2240

Program 4

For this assignment we will implement a rather simple word processor program. There are two parts to the assignment: part one, and (wait for it…) part B!

Part one

Consider this file of text:

```
The Mason–Dixon Line (or Mason and Dixon's Line)
was surveyed between
1763 and 1767 by Charles Mason and Jeremiah Dixon
in the resolution
of a border dispute between British colonies in
Colonial America. It forms
a demarcation line among four U.S. states, forming part
of the borders of
Pennsylvania, Maryland, Delaware, and West Virginia
(then part of Virginia). In popular usage, especially
since the Missouri Compromise of 1820 (apparently the
first official use of the term "Mason's and Dixon's Line"),
the Mason–Dixon Line symbolizes a cultural boundary between
the Northeastern United States and the Southern United
States (Dixie).
```

The job of part one of the word processor is to read text from a file, reformat the text to make it look nice, and place the new text in another file. The output from the above file might look like this:

```
The Mason–Dixon Line (or Mason and Dixon's Line) was surveyed
between 1763 and 1767 by Charles Mason and Jeremiah Dixon in
the resolution of a border dispute between British colonies in
Colonial America. It forms a demarcation line among four U.S.
states, forming part of the borders of Pennsylvania, Maryland,
Delaware, and West Virginia (then part of Virginia). In popular
usage, especially since the Missouri Compromise of 1820
(apparently the first official use of the term "Mason's and
Dixon's Line"), the Mason–Dixon Line symbolizes a cultural
boundary between the Northeastern United States and the
Southern United States (Dixie).
```

Notice that the output does not have to be "justified" on the right. The idea is that we keep each line under a certain number of characters, and move words as necessary to ensure that this requirement is met. **Note that there is NO space after the last word on each line.**

The program based on arguments passed on the command line. For example:

```
$ ./wordproc 60 data
```

The first command line argument (60) indicates that all of the output text can be no more than 60 characters long per line (including the spaces, not including new line). You can assume that this is a "reasonable" number and that we will not need to worry about a wild value such as 10. Let's stipulate that the number is >= 25 and <= 100.

The second command line argument is the name of the file with the input text. You are to open this file and use the data that is in this file, but create a new output file with the input name and ".out". So the above invocation would read "data" and produce "data.out".

The program should validate that there are two input parameters and that the first is a number within the range mentioned above. The program should provide a reasonable error message, and terminate, if the input file does not exist.

The general algorithm is something like this:

```
Initialize the output line to nothing
While ( read a line of text from the input file )
/* you can either fgets it and use strtok or similar, */
/* or use fscanf and %s for the format.               */
{
      For each word
            Find the length of the word
            If ( that length + length of the output line < limit )
                  Append the word to the output line
            Else
                  Write out the output line
                  Copy the word into the output line
}
Print out anything remaining in output line
```

Part B

In addition the program will produce a second file; using the above example the filename would be "data.words". This file will contain a sorted list of all words which were encountered in the file. A *part* of the file for the above input might be

```
(apparently - 1
1763 - 1
1767 - 1
1820 – 1
America. - 1
a - 3
among - 1
and - 6
between - 3
```

and so on. Notice that the number of occurrences of each word is printed after the word. Also notice that we have a very simplistic notion of what a "word" is, and that these may include the punctuation. That's fine.

In order to perform this function we want you to create an array of pointers, and to expand this array as needed using the "`realloc`" function. Shown below is an example of how this works, (using some built-in words instead of words that are read from the file). It'll give you the general idea of how to use "`realloc`":

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

int main()
{

    char **wordlist = malloc(sizeof(char*)); /* pointer to pointer to words
*/
    int  i, numwords = 0;
    char *testwords[] = { "hello", "there", "folks", "in", "1840" };
    wordlist[0] = strdup( testwords[0]);

    /* This will show how you can add words to the array of pointers */
    for( i = 1; i < sizeof( testwords ) / sizeof( testwords[ 0 ] ); i++ )
    {
        /* Expand our array of pointers by one pointer's worth */
        wordlist = realloc( wordlist, (numwords + 1) * sizeof( char * ) );
        /* Make a duplicate of the word and save the pointer to it */
        wordlist[ numwords ] = strdup( testwords[ i ] );
        numwords++;
    }

    printf( "Added %d words to the array and they are:\n", numwords );
    for( i = 0; i < numwords; i++ )
        printf( "%s\n", wordlist[ i ] );

    return( 0 );
}
```

The variable "wordlist" needs some explanation. Remember that pointers and arrays are nearly the same thing. We are allocating an array of pointers to strings, but since this is an array that we're allocating, it is a "pointer to a bunch of pointers to character".

Use the "`qsort`" function to sort the array. You will need to supply a comparison function for this. I'll let you investigate.

In order to create the sorted word list in the example, the comparison function should be created such that:

1) Different alphabetic letters should be compared case-insensitive such that different letters appear in alphabetic order regardless of the case.
2) The same alphabetic letters should be compared case-sensitive such that the upper case always appears before the lower case of the same letter.

Hand-in instructions…

Note: After the programs are collected, any compiled program (executable) in that directory is deleted and the program is recompiled for testing. Specifically, make sure you compile the program with the same options that are used for grading:

```
gcc -Wall -ansi -pedantic format.c –o format
```

or whatever the correct file name is. If the compiler generates any errors or warnings you will be penalized.