The following are full-credit answers copied from some of your responses (without attribution or permission). Some of the answers are extremely complete; others just sufficient. For the programming problems, there are many ways to solve each problem. I just picked one. For those of you who hand-wrote answers; I did not use your answers here. They were no less correct, just harder to use and anonymize.

Problem 1:

```
**
     * @param replaced – a sting a characters to be replaced
     * @param replacers – a string of characters to be used in place of
     * the replacements. replacers must be the same length as replaced
     * @param target – the string in which the replacement is to be done
     * @return the number of characters in target that were changed
     **/
     int replaceChars(char *replaced, char *replacers, char *target){
         int count,i;
        //  char current = *target;
         char *replacedStartHolder = replaced;
         char *replacersStartHolder = replacers;
        //assume the input of replaced and replacers are the same length

         while (*target!= '\0'){
             while (*replaced != '\0'){
                 if(*target == *replaced){
                     *target = *replacers;
                     count++;
                 } else {
                     replaced ++;
                     replacers ++;
```

```
            }
        }
        target ++;
        replaced = replacedStartHolder;
        replacers = replacersStartHolder;
    }

    return count;
}
```

Problem 2:

Visualization of the 2D array: {

{**1**, 2} {3, **4**} {**5**, 6} {7, 8} {9, 10}

}

LINE 1:

- - For line 1, the first value is 1 because the position on the 0 index for row and 0 index for column in the 2d array is being accessed, which, as is seen in the instantiation of arr[][], is 1.

- - Similarly, for the second output, 4, the value at row index 1 and column index 1 is 4.

- - For the third output, something a bit different is happening. Though the print statement calls the index value 2 for the column, this does not exist in the 1st row, and so the program moves on to read the next value, 5, which is technically in position [2][0]. Since C does not use official 2D arrays, and rather a 1D array with pointers designating the new rows, this kind of overflow can happen and it will just read the next value.
  LINE 2:

- - For the first printed number, the memory address 245041312 is being output, because %d is returning a reference to the pointer rather than the value (because the array is made up of doubles, so a call to print an integer would not return the value at that spot but rather the memory address). 245041312 is the memory location of the beginning of the array.

- - For the second printed number, a slightly different memory address, 245041344, is being output, again because of the printf statement calling for an integer rather than a double. This address is different from the memory address of arr, even though arr[2] is a position on arr[][], because there are values between arr[0][0] and arr[2][0], specifically 4 values in this case. Each of these 4 values takes up 8 bits in memory, making a difference of 32 bits between the first memory address and the second.

- - The third printed number is a similar memory address to the first two, in this case 56 bits or 7 values down from the beginning of the array. It is additionally printed as a memory address because of use of &, which references a variable's memory address rather than value (although since the printf is still asking for an integer rather than a double, this would return a memory address regardless).

LINE 3:

- - The first value of line 3, 245041352, is again a memory address, this time from the
double pointer variable dp. The reason why this memory address is the same as the one at arr[0][1], is because when dp was instantiated, it was declared as a pointer to the memory address of arr[0][1], given by the use of &.

- - The second value of line 3 prints as 0 because the printf statement is calling for an integer value from a pointer in an array of doubles, and since this is not possible, it returns no value, or 0, instead.

- - The third value of line 3 prints as 6 because the printf statement is calling for a double value, and the dp* now points to the same memory address as arr[0][1], which was changed from 2 to 6 with the addition of +4 in the line above.

-

Problem 3:

```
define LINE_LEN 256
int main(void){
    char line[LINE_LEN];
    int lines = 0;
    int characters;
    while(1){
        if(NULL == fgets(line, LINE_LEN, stdin)){
            break;
} lines++;
        while(line[characters] != '\0'){
            characters++;
}
}
    printf("%d%3d", lines, characters);
}

gcc -o counter counter.c
gcc -o echocl echoCL.c
./echocl this is a test | ./counter
```

PROBLEM 4: I'm only explaining why the numbers are what they are "after the colon".

LINE 0: i is 5, because 5 is the value stored in i when it is initiated. The same goes for j = 7, and k = 9.

They are all of type long. They each print out the values that are stored: 5 7 9.

LINE 1: each of the commands evaluate to 1 being true, or 0 being false. i = 5 is less than j = 7;

it evaluates to 1, true. j = 7 is not greater than k = 9; it evaluates to 0, false. k = 9 is not

less than i = 5; it evaluates to 0, but 0 is less than j = 7; it evaluates to true, 1. So the output is 1 0 1.

LINE 2: First, we know the address of i gets saved as the value of k. Then the pointer/address of k gets

saved as the variable, kk. That means *kk holds the same value as k and i, which is 5. But then, *kk is

set equal to k, and *kk points to i, so changing *kk to k means that i is now also k, so they now have all

the same address which is why they each print out the same address.

P5:

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

int sev2dec(char *seviString)
{
    int dec = 0;
    int f = 0; //bool for "first"
    for (int i = 0; i < strlen(seviString); i++)
    {
        int c = tolower(seviString[i]);
        if (c >= 97 && c <= 122)
        {
            c -= 87;
            if (f == 0)
            {
                dec = 17 * c;
                f++;
            }
            else
            {
                dec += c;
            }
        }
        else
        {
            c -= 48;
            if (f == 0)
```

```
            {
                dec = 17 * c;
                f++;
            }
            else
            {
                dec += c;
            }
        }
    }
    return dec;
}

int main(void)
{
    printf("%d\n", sev2dec("c5")); //ask are preconditions that its
only digits or alphabet?
}
```

EC:

1. head -90 file | tail -10 > bbb
2. ls -l | grep \.[hc]$
3. grep [1-9][0-9][0-9][0-9][0-9][0-9]* file