Question 1: Functions and Pointers (10 pts)

#include "stdio.h"

// Compute the sum of the numbers in an array
// Inputs:
// sum: holds the final sum
// numbers: a list of number to sum
// size: the number of numbers to sum
void sum_list(int sum, int numbers[], int size)
{
    int i;
    sum = 0;

    for(i=0; i < size; i++)
    {
        sum = sum + numbers[i];
    }
}

void main()
{
    int sum;
    int numbers[3] = {10,20,30};

    sum_list(sum, numbers, 3);

    printf("The sum is %d", sum);
    // Desired output of printf: The sum is 60
}
a. [5 pts] The function does not work as intended. Explain Why.

Sum's address is not passed to the function so `sum_list` cannot modify the original variable.

b. [5 pts] Revise the code of `sum_list` and `main` so that they behave as desired. Test your code, and then cut and paste it in the space below. Highlight your changes in bold type.

**Note 1:** You may not modify the number of arguments passed to the functions.
**Note 2:** You may only modify the existing lines of code.

```c
void sum_list(int *sum, int numbers[], int size)
{
    int i;
    *sum = 0;

    for(i=0; i < size; i++)
    {
        *sum = *sum + numbers[i];
    }
}

void main()
{
    int sum;
    int numbers[3] = {10,20,30};

    sum_list(&sum, numbers, 3);

    printf("The sum is %d", sum);
    // Desired output of printf: The sum is 60
} 
```
Question 2: Pointers (10 pts)

Complete the table below (i.e. fill in the memory map) to show the state of memory after the following C fragment has been executed. Assume the ATmega128 platform, and all the variables are in the stack (and thus the storage is allocated from high address to low address). Remember the ATmega128 is a little endian architecture (i.e. least significant byte of a variable is stored in the lowest address).

Be very careful with this question.

typedef struct pixel
{
    unsigned char red;
    unsigned char green;
    unsigned char blue;
} pixel_t;

char num[3]= {0x44,0x55, 0x66};
char* num_ptr;

pixel_t my_pixel={0x77, 0x88, 0x99};
pixel_t *pixel_ptr = &my_pixel;

long x[0] = 0xDDCCBBAA;

pixel_ptr++;
pixel_ptr++;
pixel_ptr->green = 0xFF;

num_ptr = num - 6;
*num_ptr = 0x90;

x[0] = x[2];
num[0] = *num_ptr + 5;

<table>
<thead>
<tr>
<th>Address</th>
<th>Variable Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>num[2]</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFFFE</td>
<td>num[1]</td>
<td>0x55</td>
</tr>
<tr>
<td>0xFFFFD</td>
<td>num[0]</td>
<td>0x95</td>
</tr>
<tr>
<td>0xFFFEC</td>
<td>num_ptr</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFFEB</td>
<td></td>
<td>0xF7</td>
</tr>
<tr>
<td>0xFFFDA</td>
<td>my_pixel.blue</td>
<td>0x99</td>
</tr>
<tr>
<td>0xFFF9</td>
<td>my_pixel.green</td>
<td>0x88</td>
</tr>
<tr>
<td>0xFFF8</td>
<td>my_pixel.red</td>
<td>0x77</td>
</tr>
<tr>
<td>0xFFF7</td>
<td>pixel_ptr</td>
<td>0x90</td>
</tr>
<tr>
<td>0xFFF6</td>
<td></td>
<td>0xFE</td>
</tr>
<tr>
<td>0xFFF5</td>
<td></td>
<td>0x44</td>
</tr>
<tr>
<td>0xFFF4</td>
<td></td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFF3</td>
<td></td>
<td>0xF7</td>
</tr>
<tr>
<td>0xFFF2</td>
<td></td>
<td>0x99</td>
</tr>
</tbody>
</table>
Question 3: Strings and pointers (10 pts)

The function `strcat3` concatenates three C-strings. For example, if `str1` is “Hello ”, `str2` is “CPRE”, and `str3` is “288”. Then after calling `strcat3(str1, str2, str3)`, `printf(“%s”,str1)` should print: “Hello CPRE288”

Note: For both part a) and part b) you may not use ANY other library function calls (i.e. you cannot use `strlen`, `strcat`, `strcmp`, etc.). Using a library function call will result in zero points for this problem.

a. [5] Complete the function using array access, i.e. `str[i]`, but no pointer access.

```c
int strcat3(char str1[], char str2[], char str3[]) {
    int pos1 =0;
    int pos2 =0;
    int pos3 =0;

    // Find the end of str1
    while(str1[pos1] !=0) // Keep appending until end of str2
    {
        pos1++;
    }

    // Append str2 to str1
    while(str2[pos2] !=0) // Keep appending until end of str2
    {
        str1[pos1] = str2[pos2];
        pos1++;
        pos2++;
    }

    // Append str3 to str1
    while(str3[pos3] !=0) // Keep appending until end of str3
    {
        str1[pos1] = str3[pos3];
        pos1++;
        pos3++;
    }

    // Add NULL byte to str1
    str1[pos1] = 0;
}
```
b. [5] Rewrite the above function, this time using pointer access, i.e. *str, but no array access.

Note: In general, a function’s parameter declarations “char str[]” and “char *str” are equivalent (i.e. they are interchangeable)

```c
int strcat3(char *str1, char *str2, char *str3)
{
    int pos1 =0;
    int pos2 =0;
    int pos3 =0;

    // Find the end of str1
    while( *(str1+pos1) !=0)
    {
        pos1++;
    }

    // Append str2 to str1
    while( *(str2+pos2)!=0)//Keep appending until end of str2
    {
        *(str1+pos1) = *(str2+pos2);
        pos1++; pos2++;
    }

    // Append str3 to str1
    while( *(str3+pos3)!=0)//Keep appending until end of str3
    {
        *(str1+pos1) = *(str3+pos3);
        pos1++; pos3++;
    }

    // Add NULL byte to str1
    *(str1+pos1) = 0;
}
```
Question 4 (10 pts): Binary operations, 2pts each

Write a single line of C code to implement each of the following. Assume the following declarations:

```c
unsigned char ch;            // 8-bit value
unsigned int n;              // 16-bit value
```

a. Clear to 0 bits 6, 5, 4, 1, 0 of ch, and preserve the remaining bits.

```c
ch &= 0b1000_1100; // Note: _ does not compile but it makes the grading easier
OR
ch &= 0x8C;
```

b. Set to 1 bits 14, 12, 8, 4, 2 of n, and preserve the remaining bits.

```c
n |= 0b0101_0001_0001_0100;
OR
n |= 0x5114;
```

c. Toggle bits 13 and 4, and Set to 1 bits 14 and 1, of n and preserve the remaining bits.

```c
n = (n ^ 0b0010_0000_0001_0000) | 0b0100_0000_0000_0010;
OR
n = (n ^ 0x2010) | 0x4002;
```

d. Rotate (also known as barrel shift) all the bits of ch to the right 6 positions (e.g. bit 0 should end up at bit position 2).

```c
ch = (ch >> 6) | (ch << 2);
```

e. Multiply n by 40 and store the result back to n. You may not use the multiplication operator (*) and your code must be efficient. You may ignore any potential overflow. Hint: You may use shift operations combined with an efficient arithmetic operation.

```c
n = (n << 5) + (n << 3); // n = n*32 + n*8
```
**Question 5 (10 pts): Bit checking, 2 pts each**

Write a single if condition that will evaluate to true for the following checks. Assume the following declarations

```c
unsigned char ch;
unsigned int n;
```

**a. If at least one of bits 6, 4, 0 of ch are set to 1.**

```c
if (ch & 0b0101_0001) // Note: _ doesn't compile but it makes grading easier
OR
if (ch & 0x51)
```

**b. If at least one of bits 7, 4, 2 of ch are Cleared to 0.**

**c. If all bits 13, 7, 3, 1 of n are Set to 1.**

```c
if ( (n & 0b0010_0000_1000_1010) == 0b0010_0000_1000_1010)
OR
if ( (n & 0x208A) == 0x208A)
```

**d. If bit 4 and bit 0 of ch are equal.**

```c
if ( ((ch & 0b0001_000) >> 4) == (ch & 0b000_0001) )
OR
if ( ((ch & 0x10) >> 4) == (ch & 0x01) )
```

**e. If for n, all bits 4, 3 are Set to 1, all bits 15, 0 are Cleared to 0, at least one of bits 7, 6, 5 are Set to 1, at least one of bits 2, 1 are Cleared to 0, all other bit preserved.**

```c
if ( ((n & 0x0018) == 0x0018) && ((~n & 0x8001) == 0x8001)
&& (n & 0x0080) && (~n & 0x0003) )
```
Question 6 (10 pts): Code evaluation, 2 pts each

For each C code fragment, give the final value of the indicated variable after the code fragment is run. If the final value cannot be determined, then give N/A.

a. Final value of \( ch \) is \( 0xF0 \) (in hex)
   
signed char \( ch = 0x80; \)
   \( ch = ch >> 3; \)

b. Final value of \( var\_ptr \) is \( 0xFC04 \) (in hex)
   
   long \( my\_var = 0x7F00; \)  // Assume \( my\_var \) is located at address \( 0xFC00 \)
   long *\( var\_ptr; \)
   my_var = my_var >> 5;
   var_ptr = &my_var;
   var_ptr++;

c. Final value of \( my\_var \) is \( 0x44 \) (in hex)
   
   char \( my\_var = 0x55; \)  // Assume \( my\_var \) is located at address \( 0xFFFE \)
   char *\( var\_ptr; \)
   var_ptr = &my_var;
   my_var++;

   if(&var_ptr[1] == -1){
      my_var = 0x44;
   }
   else{
      my_var = 0x33;
   }

d. Final value of \( flag \) is \( 1 \) (in decimal)

   signed char \( ch = 127; \)
   char \( flag = 0; \)
   ch++;
   flag = (100 > ch);

   Note: In the C language, a relational operator such as ‘>’ returns either 0 or 1.

e. Final value of \( ch \) is \( 1 \) (in decimal)
   
   char \( ch = 5; \)
   char *\( ch\_ptr = &ch; \)
   
   \( ch\_ptr = ch\_ptr \mid 0xFF00; \)
   if(\( ch\_ptr\)){
      ch = 1;
   }
Question 7 (10 pts): Coding practice

Complete the following function, `count_bit_pattern()`, so that it returns the number of elements in `num_array` for which bits 7-4 are equal to the inversion of bits 3-0.

As an example, an element with the pattern `00001111` would be counted.

```c
int count_bit_pattern(unsigned char *num_array, int size) {
    int total = 0; // Count elements that match the pattern
    int i; // loop counter
    for(i = 0; i < size; i++) {
        if( (num_array[i] >> 4) == (~num_array[i] & 0x0F) ) {
            total++;
        }
    }
    return total;
}
```