CSM 61C Spring 2020

C Programming

Exam Question Compilation Solutions

This document is a PDF version of old exam questions by topic, ordered from least difficulty to greatest difficulty.

Questions:

- Fall 2017 Final Q1
- Fall 2015 Final MT1-2
- Fall 2019 Quest Q4, Q6
- Fall 2018 Quest Q2A, Q2B, Q3B
- Spring 2018 Final Q1C, Q1D
- Fall 2019 Midterm Q3
- Spring 2015 Final M1-2
- Summer 2019 Midterm 1 Q3
- Summer 2019 Midterm 1 Q6
- Spring 2018 Midterm 1 Q4
- Summer 2019 Final Q3
- Fall 2019 Final Q3
- Fall 2017 Final Q3
- Summer 2018 Midterm 1 Q4
- Spring 2018 Midterm 1 Q2
- Summer 2018 Final Q1
- Spring 2018 Final Q2
- Summer 2019 Midterm 1 Q4

CSM 61C Spring 2020

Parallelism

Exam Question Compilation Solutions

This document is a PDF version of old exam questions by topic, ordered from least difficulty to greatest difficulty.

Questions:

- Fall 2018 Midterm Q5 A,B,E
- Fall 2017 Final Q2
- Fall 2017 Final Q10
- Fall 2015 Final F2
- Fall 2015 Final F3
- Spring 2015 Final F2
- Fall 2019 Final Q10a, Q10b, Q10c
- Spring 2018 Final Q10
- Spring 2018 Final Q11
- Summer 2018 Final Q9
- Summer 2018 Final Q10
- Summer 2018 Final Q13

Q1: A bit of C here and there

1. What, if anything, is wrong with this code? Write your answer in 10 words or less. (Hint: Recall Project 2-1 and consider *all* possible values of m.)

```
int floor_divide_by_two(int m) { return m >> 1; }
```

Implementation doesn't work with negative numbers.

2. Please fill in `power_of_16` to calculate whether the given integer is a power of 16.

Be sure to only use bitwise operators, == and != and up to 1 subtraction. You may introduce constants but not any new variables.

Return 0 if it is not a power of 16, or 1 if it is. (**Note: 0 is not a power of 16**).

```
// sizeof(int) == 4
int power_of_16(int m) {
      # check sign bit & # of 1-bit in binary representation
      if (0 != ((m >> 31) & 1) || (m & (m-1)) != 0) {
            return 0;
      } else {
            return m != 0 & (m & 0xEEEEEEEE) == 0;
      }
}
```

MT1-2: *C*-ing images through a kaleidoscope (8 points)

Consider a grayscale image with a representation similar to the one you worked with in Project 4, where the image is represented by a 1-dimensional array of `chars` with length `n` x `n`. Fill out the following function `block_tile`. It returns a new, larger image array, which is the same image tiled `rep` times in both the x and y direction. You may or may not need all of the lines.

For a better idea of what must be accomplished, consider the following example:

```
char *image = malloc(sizeof(char) * 4);
image[0] = 1;
image[1] = 2;
image[2] = 3;
image[3] = 4;
char *tiled_image = block_tile(image, 2, 2);
```

The contents of `tiled_image` would then look like:

```
tiled_image: [1, 2, 1, 2,
              3, 4, 3, 4,
              1, 2, 1, 2,
              3, 4, 3, 4];
```

blank 1: n * rep. This is because we have n-width per single picture and rep pictures wide.
blank 2: new_width * new_width * sizeof(char). We need a square matrix with side length new_width. The sizeof(char) is needed because we weren't given the guarantee that a char is 1 byte.
blank 3: i % n. i represents the column. We want our old_x to be the relevant column in our smaller picture. Because the column of the small picture repeats every n, we can just use modulo.
blank 4: j % n. j represents the row. With the same logic as blank 3, we can use modulo.
blank 5: j * new_width + i. Remember that our final array is 1-d, so we index into it accordingly.
blank 6: Not needed as we finished creating the array already
blank 7: Not needed as we finished creating the array already

SID: _____

```
char *block_tile(char *block, int n, int rep) {
    int new_width = ____n * rep_____________________________________;
    char *new_block = malloc(__new_width * new_width * sizeof(char)____);
    for (int j = 0; j < new_width; j++) {
        for (int i = 0; i < new_width; i++) {
            int old_x = _______i % n_____________________________;
            int old_y = ___________j % n_____________________________;
            int new_loc = ___________j * new_width + i_____________;
            new_block[new_loc] = block[old_y * n + old_x];
        }
    }
    ____________________________________________;
    ____________________________________________;
    return new_block;
}
```

For this page, assume all `mallocs` are successful, all necessary libraries are `#included`, and any heap accesses outside what the program allocates is a segmentation fault.

Q4) [12 Points] Which of the following are possible, if perhaps unlikely, results of attempting to compile and run this code? (select ALL that apply)

```
int main() {
    int32_t *str = (int32_t *) malloc(sizeof(int32_t) * 3);
    printf("%s", (char *) str); // A char is 8 bits.
    return 0;
}
```

- ☐ Compilation error due to invalid typecast
- ☐ Runtime typecasting error
- ● A segmentation fault
- ● The program prints the empty string
- ● The program prints **`CS61C`**
- ☐ The program prints **`CS61C rocks!`**

- Compilation or runtime typecast error: Won't happen, the cast of the bits on the right matches the value expected on the left: `(int32_t *)` matches `str`'s type, and `(char *)` matches `%s`.
- A segmentation fault: Possible; this can occur if no byte in val is a null terminator (`'\0'`), since inside `printf` it will keep reading the string `str` until it finds the null terminator. Note that because `malloc` doesn't clear out the data already in that space, it is indeterminate whether this will occur, or if any valid string is outputted.
- The program prints the empty string "": Possible; this can occur if the 0th byte in `str` is a null terminator
- The program prints the string "`CS61C`": Possible; this can occur on the off chance that the first six bytes of `str` match the sequence "`C`", "`S`", "`6`", "`1`", "`C`", "`\0`"
- The program prints the string "`CS61C rocks!`": Impossible; "`CS61C rocks!`" is 12 characters, and since all strings must contain a null terminator, that would require 13 bytes to store. Since `str` is only 12 bytes long, "`CS61C rocks!`" cannot be outputted. Remember that we stated that any memory accesses the program did not allocate will cause a segfault. This means that the allocation of `str` is on the heap, which means that we only have 12 bytes to work with – accessing that 13th byte to get the "`\0`" would error.

Q5) [10 Points] Each of the following evaluate to an address in memory. In other words, they "point" somewhere. Where in memory do they **point**?

	Code	*Static*	*Stack*	*Heap*
`arr`	○	●	○	○
`arr[0]`	○	●	○	○
`dest`	○	○	●	○
`dest[0]`	○	○	○	●
`&arrPtr`	○	○	●	○

- `arr` is an array of character pointers, so it itself is a pointer to the first element in the array. Since the array is declared globally, its contents are placed in the static portion of memory, so `arr` points to **static**.
- `arr[0]` is a string literal (therefore, a pointer to

Q6) [10 Points] The program below runs through the array of strings, doing something to each of the characters and putting the results in the `dest` array.

What are the first 8 characters the program prints? (Note: The program DOES compile and run without error.)

G O _ B E A R S

This function capitalizes all letters in `arr`, the letter's ASCII encoding has its 5th bit turned off by the bitwise and. Let's go through that slowly.
A's binary representation is `0b0100 0001`
a's binary representation is `0b0110 0001`
(hmm, you might say, it only differs in the 32's place, and that remains true for all the letters since there are only 26 of them and 32 possible bit patterns for the lowest 5 bits)

```
int mystery (unsigned int N) {
      unsigned int counter = 0;
      while (N > 0) {
            counter += N & 1;
            N = N >> 1;
      }
      return counter == 1;
}
```

Q2a) What does the **mystery** return? (Select ONE)

◯ The number of 1s in the binary representation of **N**

◯ 1 if **N** is odd, otherwise 0

● **1 if N is a power of 2, otherwise 0 [it shifts N to the right, storing in counter all the 1s it sees. If it's exactly 1, meaning the only 1 is the MSB (most significant bit), then it's a power of 2]**

◯ 1 if the binary representation of **N** is all 1s, otherwise 0

◯ 1 if the binary representation of **N** has any 1s, otherwise 0

Q2b) Given this setup to **mystery**:

```
unsigned int myN =
GetNFromUser();
int mysteryReturn = mystery(myN);
```

...could **myN** be changed by the call to **mystery**? (Select ONE)

◯ Yes

◯ It depends on the value of **myN**

● **No [because C is *call by value*]**

```
// My project partner wrote this code to duplicate some elements of orig into copy
      int orig[] = {1,2,3,4,5,6,7,8}; // ints are 4 bytes wide      ...scratch space below...
      int main() {
            int *backup, *copy, **copyH;
            backup = copy = (int *) malloc (sizeof(int) * 100);
            copyH = &copy;
            for (int i = 0; i < 2; i++) {
                  *copy = orig[i];
                  *copyH = *copyH + 4;
            }
      }
```

Q3a) Right *before* the **for** loop, where in memory do the following **point**? (Select ONE per row)

	Code	*Static*	*Stack*	*Heap*
orig	○	●	○	○
backup	○	○	○	●
copyH	○	○	●	○

Q3b) Right *after* the **for** loop, what is the value of the following? If it'd be garbage, write "**G**".

backup[0]	**backup[1]**	**copy[0]**	**copy[1]**
1	**G**	**G**	**G**

backup,copy ↓ **backup = copy = (int *) malloc (sizeof(int) * 100);**

the memory below shows all the words (not bytes) of the malloced space

G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G

backup,copy ↓ ***copy = orig[0];**

1	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G

backup ↓ copy ↓ ***copyH = *copyH + 4;**

1	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G

backup ↓ copy ↓ ***copy = orig[1];**

1	G	G	G	**2**	G	G	G	G	G	G	G	G	G	G	G

backup ↓ copy ↓ ***copyH = *copyH + 4;**

1	G	G	G	**2**	G	G	G	G	G	G	G	G	G	G	G
backup[0]	backup[1]							copy[0]	copy[1]						

(c) Given the following function in C:

```
int shifter(int x, int shift) {
    if (x > 0) {
        return x >> shift;
    }
    return -1 * (x >> shift);
}
```

Given y is a negative integer, and that `shifter(y, 2)` outputs 4, what is the range of values of y?

hint: `-8 >> 1 = -4`

Solution: -16 ~ -13

(d) Implement the function `unsigned int base_convert(unsigned int num, unsigned int base)`. This function takes in non-negative integers `num` and `base`. You are guaranteed the following:
- `base` is an integer in the range [2, 10], no need to error check this
- `num` is comprised of "digits" with a value between 0 and `base` - 1.
- All values fit inside an `unsigned int`.

Your job is to make it so the function returns the decimal value of `num` in base `base`. For example, `base_convert(30, 4)` would return 12, since 30_4 is 12_{10}. You may not use additional lines (do not put multiple lines on the same line via ;) but you may not need all the lines provided. In addition, you may not include `<math.h>` or use `pow()`.

Solution:

```
unsigned int base_convert(unsigned int num, unsigned int base) {

    unsigned int value = 0, power = 1;

    while (num > 0) {
        value += (num % 10) * power;
        power *= base;
        num /= 10;
    }
    return value;
}
```

Q3) *I thought I needed to do a 2s but it was really just a sign-mag?!* **(20 pts = 7*2 + 6)**

You recover an array of critical 32-bit data from a time capsule and find it was encoded in sign-magnitude! Write the **`ConvertTo2sArray`** function in C that converts all the data to 2s complement. You are told that **`0x00000000`** was never used to record any *actual data*, and is the array terminator (just as you do for strings). **`ConvertTo2s`** does the actual conversion for each number. Select ONE per letter; for **<h>** fill in the blank.

```
void ConvertTo2sArray( <a> A ) {
    while ( <b> ) {
        if ( <c> )
            ConvertTo2s( <d> );
        <e> ;
    }
}

void ConvertTo2s( <f> B ) {
    <g> = <h> ;
}
```

<a>	○ int32_t		● int32_t *	
<b>	○ true	○ false	○ A	● *A
<c>	○ A < 0	● *A < 0	○ A	
	○ A > 0	○ *A > 0	○ *A	
	○ A <= 0	● *A <= 0	○ true	
	○ A >= 0	○ *A >= 0	○ false	
<d>	○ &A	● A	○ *A	
<e>	● A = A + 1		○ *A = *A + 1	
<f>	○ int32_t		● int32_t *	
<g>	○ &B	○ B	● *B	

<h>	Some valid answers:
	`~(*B & 0x7FFFFFFF)+1;`
	`-(*B & 0x7FFFFFFF);`
	`-(*B + (1<<31));`
	`(*B - 1) ^ 0x7FFFFFFF;`
	`~((*B<<1)>>1)+1;`
	`(*B ^ 0x7FFFFFFF) + 1;`			

With Sign and magnitude (SM), the most significant bit (MSB, aka leftmost bit) represents sign, and the remaining bits represent magnitude. A positive SM number has a MSB of 0, negative SM has MSB of 1. With 2's complement (2's), positive numbers look identical to how they would with SM. However, to get negative 2's numbers, remember we take the positive number representation, flip each bit, and then add 1. A key observation is that positive SM and 2's numbers are represented identically (thus our condition in <c> isn't called for these positive numbers). Thus the chief job of the function `ConvertTo2s` is to take a negative SM number and convert it to a negative 2's number.

The most intuitive strategy for this is to take the negative SM number, set the MSB to 0 to give us the positive magnitude, and then convert this positive magnitude to 2's by flipping all bits and adding 1. *B gives us the number we want to work with, and 0x7FFFFFFF is a mask of a single 0 bit followed by thirty-one 1 bits (note MSB for this mask is 0). 'and'ing with such a mask will set the MSB to 0 (any bit 'and'ed with 0 equals 0), and leaves all other bits unchanged (any bit 'and'ed with 1 is unchanged). Thus (*B & 0x7FFFFFFF) will flip the MSB of the negative SM number, giving us just the positive magnitude. We can then use the formula ~(positive magnitude)+1 to convert the positive magnitude to a negative 2's number, giving us the full expression of ~(*B & 0x7FFFFFFF)+1. Check out this article for a more concrete example explaining this process: http://cseweb.ucsd.edu/classes/fa99/cse141l/ass1update.htm

Alternatively we can just apply the unary negation operator '-' on the positive magnitude to get the negative 2's number: -(*B & 0x7FFFFFFF), since C uses 2's to store signed ints, so the '-' operator invokes 2's rules of ~(...)+1 to negate.

Another method is to cause the negative SM number to overflow: (*B + (1 << 31)). The (1 << 31) just sets the mask MSB to 1, and 'add'ing this mask effectively 'add's 1 to the SM number's MSB of 1, overflowing the resulting MSB to 0, giving us the positive magnitude. We now can use the unary negation operator '-' to get the negative 2's number. This approach gives a final formula of -(*B + (1 << 31)).

M1-2: *I'll believe it when I C it* (9 points)

Your friend wants to take 61C next semester, and is learning C early to get ahead. They try to implement the ROT13 function as practice, but the code they wrote has some bugs, so you've been called in, as the C expert, to help debug their program, reproduced below:

```
1  /* Applies the ROT13 cipher. rot13("happytimes") == "uncclgvzrf" */
2  void rot13(char *str) {
3      while (*str) {
4          if (str >= 'a' && str <= 'z') {
5              *str = (*str + 13) % 26;
6          }
7          str++;
8      }
9  }
10
11 int main(int argc, char *argv[]) {
12     char a[] = "happy";
13     char b[] = "times";
14     char *s = "XXXXXXXXXXX"; // 12 X's
15
16     // Apply cipher to a and b.
17     rot13(a);
18     rot13(b);
19     printf("%s%s\n", a, b);
20
21     // Concatenate and place in s.
22     int i = 0;
23     for (int j = 0; a[j]; ) s[i++] = a[j++];
24     for (int j = 0; b[j]; ) s[i++] = b[j++];
25
26     printf("%s\n", s);
27 }
```

we are comparing an address to a number; these should be *str instead

we want this to be a letter, so the ascii should be in the range ['a', 'z']. Taking the number mod 26 will not achieve this.

5 characters

5 characters

s is a pointer to a static string, so it cannot be mutated

makes a refer to "uncc" (If rot13 works)

makes b refer to "gvzrf" (If rot13 works)

ERROR: mutating the string in read-only static memory

If s is longer than len(a) + len(b), then there will be residue characters here unless we manually insert a NUL Byte

a) You want to impress your friend, so you predict the result of executing the program as it is written, just by looking at it. If the program is guaranteed to execute without crashing, describe what it prints, otherwise explain the bug that may cause a crash.

The program will crash on line 23 when it tries to modify the string literal referred to by s.

b) Now, fix all the errors in the program so that it executes correctly. Fill in the corrections you made in the table below. You may not need all the rows.

Line #	Insert Before / Replace / Delete	Change (Explanation or Code)
4	Replace	*str >= 'a' && *str <= 'z'
5	Replace	*str = (*str - 'a' + 13) % 26 + 'a'
14	Replace	char s[] = "XXXXXXXXXXX"
25	Insert	s[i] = '\0'

Question 3: Help, there's a moth in my computer! (Debuggin' ;) - 16 pts

For each of the following functions there are comments about what certain lines are meant to do. Mark any lines whose contents **DO NOT** accomplish what the comment asks it to do. If everything functions as described, mark "no errors". Note the comment ABOVE describes the line(s) BELOW.

We have also provided a comment about what the whole function is meant to do, but that should not be necessary to complete this question (although you may find it helpful when trying to understand the code). For this section you may assume that the file contains all necessary includes, that all calls to malloc succeed, and that all input arguments to the functions are valid.

1.

```
/* Function that takes in an array of integers, mallocs space for a new array,
 * and copies the integers from the first array into the new array. It returns
 * the new array.*/
int* copy_ints(int* arr) {
     /* Allocates space to store all integers in arr. */
[ ]  int* new = malloc(sizeof(arr)); // Can't use sizeof (arr), need to pass in a len

     /* Iterates over all the elements in arr. */
[ ]  for (int i = 0; i < sizeof(arr); i++) { // Can't use sizeof (arr), need a len

        /* Loads an element from arr and stores it in new. */
[ ]     *(new + i) = *(arr + i);
     }

     /* Returns a pointer that can be dereferenced in other functions. */
[ ]  return new;
}
```

[ ] no errors

2.

```
/* Function that takes in an integer, interprets it as a boolean value,
 * and returns a string that can be dereferenced outside the function
 * indicating if it was true or false.*/
char* bool_to_string (int i) {
     /* Allocates space for a pointer. */
[ ]  char* ret_val; // Allocates space for a pointer but not contents

     /* Evaluates to true on all false values and false on all true values. */
[ ]  if (i == 0) {
         ret_val = "false";
     } else {
         ret_val = "true";
     }
     // String literals have memory allocated, so the assignment works

     /* Returns a pointer that can be dereferenced in other functions. */
[ ]  return ret_val; // String literals last for the life of the program
}
```

[ ] no errors

3.

```
/* Function that takes in a non-null-terminated string and its length and
 * returns a pointer to a malloc'd, null-terminated copy of the string.*/
char* null_term(char* str, unsigned int len) {
     /* Allocates space to store a null-terminated version of str. */
[ ]  char* copy = (char*) malloc(sizeof(char) * len); // Missing +1 for '\0'

     /* Iterates over all the elements of str. */
[ ]  for (int i = 0; i < len; i++) {

        /* Loads an element from str and stores it in copy. */
[ ]     copy[i] = *(str++);
     }

     /* Appends a null terminator to the end of copy. */
[ ]  copy[len] = '\0';

     /* Returns the string, that can be dereferenced in other functions. */
[ ]  return &copy; // Need to just return copy, not the address of the string
}
```

[ ] no errors

4.

```
/* Function that takes in the start of a linked list, mallocs space for a
 * new element, appends that element to the front, and returns the new start
 * of a linked list.*/
typedef struct int_node {
  int value;
  struct int_node* next;
} int_node_t;

int_node_t* append_front(int_node_t* current, int value) {
     /* Allocates space for a new int node. */
[ ]  int_node_t* new = malloc(sizeof(int_node_t*)); // Should be sizeof (int_node_t)

     /* Assigns the value field to the value parameter. */
[ ]  new->value = value;

     /* Assigns the next field to the current front. */
[ ]  (*new).next = current; // The same as new->next

     /* Returns a pointer that can be dereferenced in other functions. */
[ ]  return new;
}

[ ] no errors
```

Question 6: More debugging!! Yay! - 9 pts

Morgan is interested in exchanging secret messages with Branden, but she doesn't want Nick to be able to read them. She writes the following secret_encoder function which takes in a string and its size and increments all the characters by thirteen to make a rotational cipher. Assume inputs never cause overflow and all necessary libraries are included.

```
void secret_encoder(char* arr, int len) {
      printf("Encoding: %s\n", arr);
      for (int i = 0; i < len; i++) {
            arr[i] += 13;
      }
      printf("Result: %s\n", arr);
}
```

Morgan decides, like any good CS61C student, she should test her code on a few examples. She writes the following function call.

```
int main(int argc, char* argv[]) {
      char hello[5] = "Hello";
      secret_encoder(hello, 5);
      return 0;
}
```

Using an ASCII table, she calculates her expected output to be:

```
Encoding: Hello
Result: Uryy|
```

However, when she runs the code, her actual output is:

```
Encoding: Hello??_??
Result: Uryy|??_??
```

1. Which of the following *best* describes Morgan's issue:

Ⓐ Null pointer exception
Ⓑ Uninitialised variable
Ⓒ Missing a null terminator
Ⓓ Memory management mistake

2. Branden and Morgan try to work together to fix their code, but they encounter some issues. Read through various implementations of the file below and select which implementations **are correct** for the program to produce the described output from above. Assume all necessary libraries are included.

Which implementation(s) are correct?: __C_____

Implementation A

```
void secret_encoder(char* arr)
{
  printf("Encoding: %s\n", arr);
  for (int i = 0; i < strlen(arr); i++)
  {
    arr[i] += 13;
  }
  printf("Result: %s\n", arr);
}

int main(int argc, char* argv[])
{
  char hello[5] = “Hello”;
  secret_encoder(hello);
  return 0;
}
```

Doesn’t include null terminator in ‘hello’, so strlen won’t work correctly.

Implementation B

```
void secret_encoder(char* arr, int len) {
  printf("Encoding: %s\n", arr);
  for (int i = 0; i < strlen(arr); i++)
  {
    arr[i] += 13;
  }
  printf("Result: %s\n", arr);
}

int main(int argc, char* argv[])
{
  char hello[5] = “Hello”;
  secret_encoder(hello, 5);
  return 0;
}
```

Doesn’t include null terminator in ‘hello’, so strlen won’t work correctly. Len is irrelevant.

Implementation C

```
void secret_encoder(char* arr, int len) {
  printf("Encoding: %s\n", arr);
  for (int i = 0; i < len; i++)
  {
    arr[i] += 13;
  }
  printf("Result: %s\n", arr);
}

int main(int argc, char* argv[])
{
  char hello[6] = “Hello”;
  secret_encoder(hello, strlen(hello));
  return 0;
}
```

Implementation D

```
void secret_encoder(char* arr, int len) {
  printf("Encoding: %s\n", arr);
  for (int i = 1; i < len + 1; i++)
  {
    arr[i] += 13;
  }
  printf("Result: %s\n", arr);
}

int main(int argc, char* argv[])
{
  char* hello = “Hello”;
  secret_encoder(hello, 5);
  return 0;
}
```

Includes null terminator, but iterates over it, changing it to a new character. Printing will fail.

Problem 4 ***Sorting With Pointers*** **(20 points)**

You are given the following implementation of insertion sort for strings:

```
1 void string_insertion_sort(char *str) {
2      char tmp;
3      int inner, outer;
4      outer = 1;
5      while (str[outer] != '\0') {
6           inner = outer;
7           while (inner != 0 && str[inner] > str[inner - 1]) {
8                tmp = str[inner - 1];
9                str[inner - 1] = str[inner];
10               str[inner] = tmp;
11               inner--;
                        <------ breakpoint here
12          }
13          outer++;
14     }
15 }
```

It is called like so:

```
char str[4] = "bdf";
string_insertion_sort(str);
```

(a) **Trace Execution:** Suppose that I've set a breakpoint after line 11 (such that the breakpoint triggers after line 11 executes). Fill in the table for the memory contents at the breakpoint for the first 2 times this breakpoint triggers. The initial state refers to the contents of memory on line 2 (right when we enter the function). You may write either the character or the ASCII code, memory location, OR variable value respectively.

Solution:

Addr	C Variable	Initial State	1st Break	2nd Break
0xA	str[0]	'b' (98)	'd' (100)	'd' (100)
0xB	str[1]	'd' (100)	'b' (98)	'f' (102)
0xC	str[2]	'f' (102)	'f' (102)	'b' (98)
0xD	str[3]	'\0' (0)	'\0' (0)	'\0' (0)
0xE	str	0xA	0xA	0xA
0x11	tmp	uninitialized	'b' (98)	'b' (98)
0xF	inner	uninitialized	0	1
0x10	outer	uninitialized	1	2

(b) How many more times will the breakpoint trigger?

○ 0 ○ 3

● 1 ○ 4

○ 2 ○ 5

(c) Is this program correct for all null-terminated strings?

○ Yes ● No

If you answered yes, leave this blank. If you answered no, provide a string that would serve as a counterexample.

Solution: '\0', as well as any statically defined string.

Suppose that we revise the above insertion sort algorithm to support a new character encoding that uses 16 bits (similar to Unicode). These characters are stored in a 16 bit unsigned integer type (called `uni_t`). The function is identical except that all variables are now of type `uni_t` instead of `char`. We will call this function `uni_string_insertion_sort`. We call this new sort in a similar fashion:

```
uni_t str[4] = "Ωβα";
uni_string_insertion_sort(str);
```

Suppose that the first character in `str` is stored at address `0xFFF8`. Answer the following questions assuming `uni_string_insertion_sort` is run from the start. If the answer cannot be determined or would cause an error from the provided information, write "Unknown".

(d) After line 4 has executed, what is the value of `str + outer`?

Solution: 0xFFFA (0xFFF8 + `sizeof(uni_t)*1`)

(e) After line 4 has executed, what is the value of `*(str + outer)`?

Solution: β

(f) After line 4 has executed, what is the value of `str[inner]`?

Solution: Unknown, `inner` is uninitialized.

Question 3: C Coding - 24 pts

In this question we are going to implement a double-ended queue data structure, which is a data structure in which you an insert to either end. To do so we will allocate a single array to store all data contiguously, but because we need to append to both ends we will implement our array as a circular buffer. A circular buffer is a way of wrapping around an array while maintaining the ordering. For example imagine the following implementation where we append to the left of our queue with an initial value of 3.

```
// Initially q->data = [garbage, 3, garbage];
append_left (q, 2) // Now q->data = [2, 3, garbage]
append_left (q, 1) // Now q->data = [2, 3, 1];
// We keep track of the order with additional struct fields.
```

Notice that we fill the array entirely and move from one end to the other when we run out of space. To implement our queue we have provided a struct and a constructor on the handout.

1. Implement `print_reverse_dqueue` which prints each valid element in the array from the end to the front (left to right) with each element on a newline. You may not need all lines.

```
#include <stdio.h>
void print_reverse_dqueue (int_dqueue_t* q) {

      for (int i = 0; i < q->occupied_size; i++) {

            int location = q->right_location - i - 1;

            if  (location < 0) {

                  location = location + q->allocated_size;

            }

            printf (“%d\n”, q->data[location]);

      }

}
```

One issue that complicates our queue is what happens when we need to resize. With other data structures we can use realloc, but imagine we have the following full data where the actual order of the data is 1, 2, 3, 4.

```
q->data = [3, 4, 1, 2];
```

If we were to reallocate the queue to size we would then get:

```
q->data = [3, 4, 1, 2, garbage, garbage, garbage, garbage]
```

Now if we only realloc we can't maintain our ordering, so we need to do some extra work when resizing.

2. Implement `expand_buffer` which takes in a queue that is **full** and reallocs circular buffer while maintaining the previous ordering. You can assume all calls to realloc succeed and you may not need all lines.

Recall the header for realloc is:

```
void* realloc (void* ptr, int size);
```

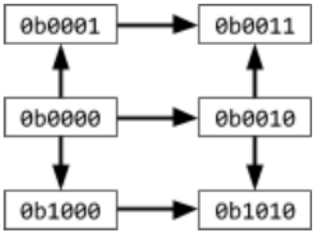
Hint: You probably only want to change either left_location or right_location, not both

```
#include <stdlib.h>
void expand_buffer (int_dqueue_t* q) {

      q->allocated_size *= 2;

      q->data = realloc (q->data, q->allocated_size * sizeof(int));

      for (int i = 0; i <= q->left_location; i++) {

            q->data[q->occupied_size + i] = q->data[i];

      }

      q->right_location = q->left_location + q->occupied_size + 1;

      if (q->right_location >= q->allocated_size) {

            q->right_location -= q->allocated_size;

      }

}
```

Q3) There's a Dr. Hamming to C you... **(14 pts)**

We are given an array of **N unique** `uint32_t` that represent nodes in a directed graph. We say there is an edge between **A** and **B** if **A < B** and the Hamming distance between A and B is **exactly 1**. A Hamming distance of 1 means that the bits differ in 1 (and only 1) place. As an example, if the array were {`0b0000, 0b0001, 0b0010, 0b0011, 0b1000, 0b1010`}, we would have the edges shown on the right:

A	B
0b0000	0b0001
0b0000	0b0010
0b0000	0b0100
0b0001	0b0011
0b0010	0b0011
0b0010	0b1010
0b1000	0b1010



Construct an `edgelist_t` (specified below) that contains all of the edges in this graph. Our solution used every line provided, but if you need more lines, just write them to the right of the line they're supposed to go after and put semicolons between them. All of the necessary `#include` statements are omitted for brevity; don't worry about checking for `malloc`, `calloc`, or `realloc` returning NULL. *Make sure `L->edges` has no unused space when `L` is eventually returned.*

```
typedef struct {
   uint32_t A;
   uint32_t B;
} edge_t;

typedef struct {
   edge_t *edges;
   int len;
} edgelist_t;
```

```
edgelist_t *build_edgelist(uint32_t *nodes, int N) {
   edgelist_t *L = (edgelist_t *) malloc (sizeof(edgelist_t));
   L->len   = 0;


   L->edges = (edge_t *) malloc (N * N * sizeof(edge_t));

   for (int i = 0; i < N; i++) {
       for (int j = 0; j < N; j++) {
           uint32_t tmp = nodes[i] ^ nodes[j];
           if ((nodes[i] < nodes[j]) && !(tmp & (tmp-1))) {
               L->edges[L->len].A = nodes[i];
               L->edges[L->len].B = nodes[j];
               L->len++;
           }
       }
   }
   L->edges = (edge_t *) realloc(L->edges, sizeof(edge_t) * L->len);
   return L;
}
```

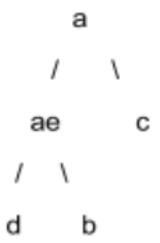
Q3: Now you C me, now you don't

Let's build a basic min-heap. Recall that a min-heap is similar to a binary tree where each child is greater than or equal to its parent. We will be using a single dimensional array to represent our min-heap, which other than `create`, supports `insert` and `free_heap` only.

Here's an example. Given the array,

```
char* arr[] = { "d", "b", "c", "ae", "a" };
```

Our min-heap will transform this array into
`["a", "ae", "c", "d", "b"]`



Which, diagrammatically, would be represented as on the right:

Here's the data structure we will be using, with its corresponding create function.

```
#define INIT_CAP 10

struct min_heap {

      char** data;

      size_t size; // current number of elements in heap

      size_t capacity; // max number of elements heap can handle

};

struct min_heap* create(char** arr, size_t size) {
      struct min_heap* m = malloc(sizeof(struct min_heap));
      m->size = m->capacity = 0;
      for(size_t i = 0; i < size; i++)
            insert(m, arr[i]);
      return m;
}
```

Please fill in the blanks to complete the functionality for the `free_heap` and `insert` functions on your **answer sheet**. As a reminder, to insert an element into a min heap, you will insert the element at the back of the array and then bubble up that element (this has already been implemented for you). You may or may not need all lines.

For the purposes of this problem, assume

- Access to the entire standard library
- `data` should contain copies of strings; memory allocation always succeeds.
- String comparisons are done lexicographically.
- For an empty heap, allocate `INIT_CAP` elements initially and every time afterwards, if we run out of space, double its size. Do not declare any new variables or introduce new conditional statements or loops.

```
#define INIT_CAP 10
#define RESIZE_FACTOR 2
void free_heap(struct min_heap* m) {

      for (size_t i = 0; i < m->size; i++)
            free(m->data[i]);
      free(m->data);
      free(m);

}

void insert(struct min_heap* m, char* elem) {

        if(m->capacity == 0) {

                m->data = malloc(sizeof(char*) * INIT_CAP);

                m->capacity = INIT_CAP;

        } else if(m->size >= m->capacity) { #check if we need to resize

                m->data = realloc(m->data, sizeof(char*)*(m->capacity*2));

                m->capacity = m->capacity * 2;

         }

         char* temp = malloc((strlen(elem) + 1) * sizeof(char));

         strcpy(temp, elem);

         int iter = m->size++;

      // Bubble up inserted element to correct position in heap
      while(iter != 0 && strcmp(temp, m->data[(iter - 1) / 2]) < 0)) {

                  m->data[iter] = m->data[(iter - 1) / 2];

                  iter = (iter - 1) / 2;

      }
      m->data[iter] = temp;

}
```

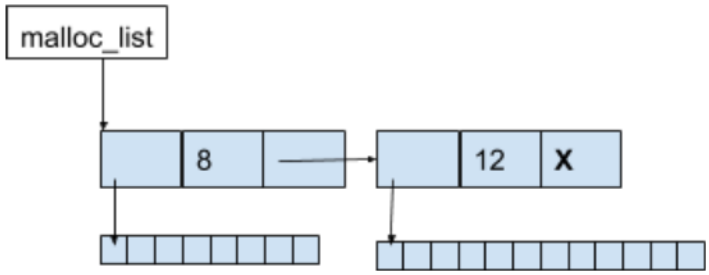
Question 4: C Coding (20 pts)

Recall that in C, pointers have no sense of their own bounds. Nick doesn't like this, so he decided to replace malloc and free with helper functions to keep track of memory bounds. To do so he decides to create wrapper functions for malloc and free that he will call instead of just malloc or free. To do so he creates a struct:

```
typedef struct malloc_node {
      void *data_ptr;
      size_t length;
      struct malloc_node *next;
} m_node;
```

Example:

```
user_malloc (12);
user_malloc (8);
```



This holds the value of the malloc'ed pointer along with the original length requested in bytes as a linked list node. He also creates a global variable:

```
m_node *malloc_list; // Assume this is initialized to NULL
```

Using these globally accessible structures you will implement a form of malloc and free that can keep track of the bounds on heap pointers. For this question assume all mallocs succeed. For both questions you may not need all lines.

/ The user wrapper function for malloc. This function is called instead of malloc when asking for N bytes of heap memory. This function should make a call to malloc to produce this memory, but it should also add a node to malloc_list to store the additional size information. A visual is shown above.*
*Hint: You may find it easier to add an element to the front of a linked list rather than the end . */*

```
void *user_malloc (size_t n) {
      m_node *node = malloc (sizeof (m_node));
      node->data_ptr = malloc (n);
      node->length = n;
```

```
        node->next = malloc_list;
        malloc_list = node;
        return node->data_ptr;
}
```

Finally we need an implementation of free which also frees this metadata.

```
void user_free (void *ptr) {
        if (! remove_ptr (& malloc_list, ptr)) {
                illict_free ();         // Assume this handles any errors from
illegal
                                        // attempts to free.
        }
}
```

Implement remove_ptr. This should free the PTR, remove the metanode node from the linked list and cleanup any metadata if the free is legal. It should return true if it was successful and otherwise false.

/ Takes in a NODE_PTR which points to a part of the list and a ptr and if the node stores the info about pointer handles any appropriate freeing and removes the node holding that pointer from the list. */*

```
bool remove_ptr (m_node **node_addr, void *ptr) {
        if (node_addr && *node_addr && ptr) {
                if ((*node_addr)->data_ptr != ptr) {
                        return remove_ptr (&(*node_addr)->next, ptr);
                }
                m_node *temp = (*node_addr);
                free (temp->data_ptr);
                *node_addr = temp->next;
                free (temp);
                return true;
        }
        return false;
}
```

Nick is considering revising the `struct malloc_node` definition by adding another field (bolded for your convenience):

```
typedef struct malloc_node {
     void *data_ptr;
     size_t length;
     struct malloc_node *next;
     size_t num_bytes;
} m_node;
```

Given this new struct definition, the value returned by `sizeof(next)` changes. Ⓣ Ⓕ

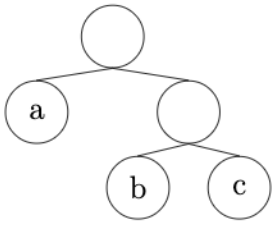
Problem 2 ***C Coding*** **(20 points)**

(a) **Find the End:** Given a **acyclic** singularly linked list, return a pointer to the last non-NULL node of the linked list. We have defined the `list_node` struct for the linked list below. Note that each node contains a string `str` as data. Return NULL if the list is empty. You may or may not need every blank but you may not add additional lines.

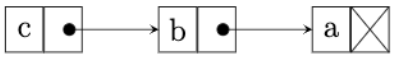
```
typedef struct list_node {
    char *str;
    struct list_node *next;
} list_node;
```

Solution:
```
list_node * find_end(list_node *start) {
    if (start == NULL) {
        return NULL;
    }
    while (start->next != NULL) {
        start = start->next;
    }
    return start;
}
```

(b) **Flatten:** Given a binary search tree (BST) **with data only stored in the leaf nodes**, generate a linked list with elements of the tree in reverse order (since this is a BST, the linked list will necessarily be in descending sorted order). The new linked list should have a **full copy** of each string in the BST. Return NULL if the BST is empty. For example, the following BST:



Should be turned into a linked list as such:



You are given a BST `tree_node` struct, the `list_node` struct and `find_end` (assume it's correctly implemented). You may use any C standard library functions and may assume that memory allocation always succeeds. Fill in the blanks to implement `flatten`. You may not need all the lines but you must not add additional lines.

```
typedef struct tree_node {
    char *str;
    struct tree_node *left, *right;
} tree_node;
char *strcpy(char *dest, const char *src);
list_node * find_end(struct list_node *start); // from part (a)
```

Solution:

```
list_node * flatten(tree_node *root) {
    if (root == NULL)
        return NULL;
    if (root->str != NULL) {
        list_node *curr = malloc(sizeof(list_node));
        curr->str = malloc(strlen(root->str)+1));
        strcpy(curr->str, root->str);
        return curr;
    }
    list_node *left_list = flatten(root->left);
    list_node *right_list = flatten(root->right);
    list_node *rend = find_end(right_list);
    if (rend == NULL) {
        return left_list;
    } else {
        rend->next = left_list;
        return right_list;
    }
}
```

SID: _____

Question 1: Main [Memory] Stacks (16 pts)

Recall the idea of the Stack section in our memory hierarchy: when functions are called, their function frames are **pushed** onto the stack; when a function returns, it is **popped** off the stack. We will be implementing the Stack section of memory in software by representing each function frame on the stack as a struct, defined below. Assuming we are running on a **32-bit machine** with **32-bit integers**.

```
typedef struct stack {
      void *frame;          // Pointer to the space allocated for this function frame
      struct stack *prev;   // Pointer to previous stack node
      char *func_name;      // Name of the stack frame's function
      int threads;          // The number of threads below this stack frame
      int in_use;           // The number of threads currently pointing to this stack frame
} StackNode;

StackNode *sp = NULL;
```

1) What would `sizeof (StackNode)` return?

5 * 4B = 20 B

Suppose that the staff has implemented a function `frame_size` which takes in a C function's name as a string and outputs how many **bytes** that the function's local variables would need in the function frame:

```
int frame_size (char *func_name) {
      \\ Assume this function has been correctly implemented for you.
}
int main_stacks (int argc, char *argv[]) {
      char arr[] = "61C rox";
      return 0;
}
```

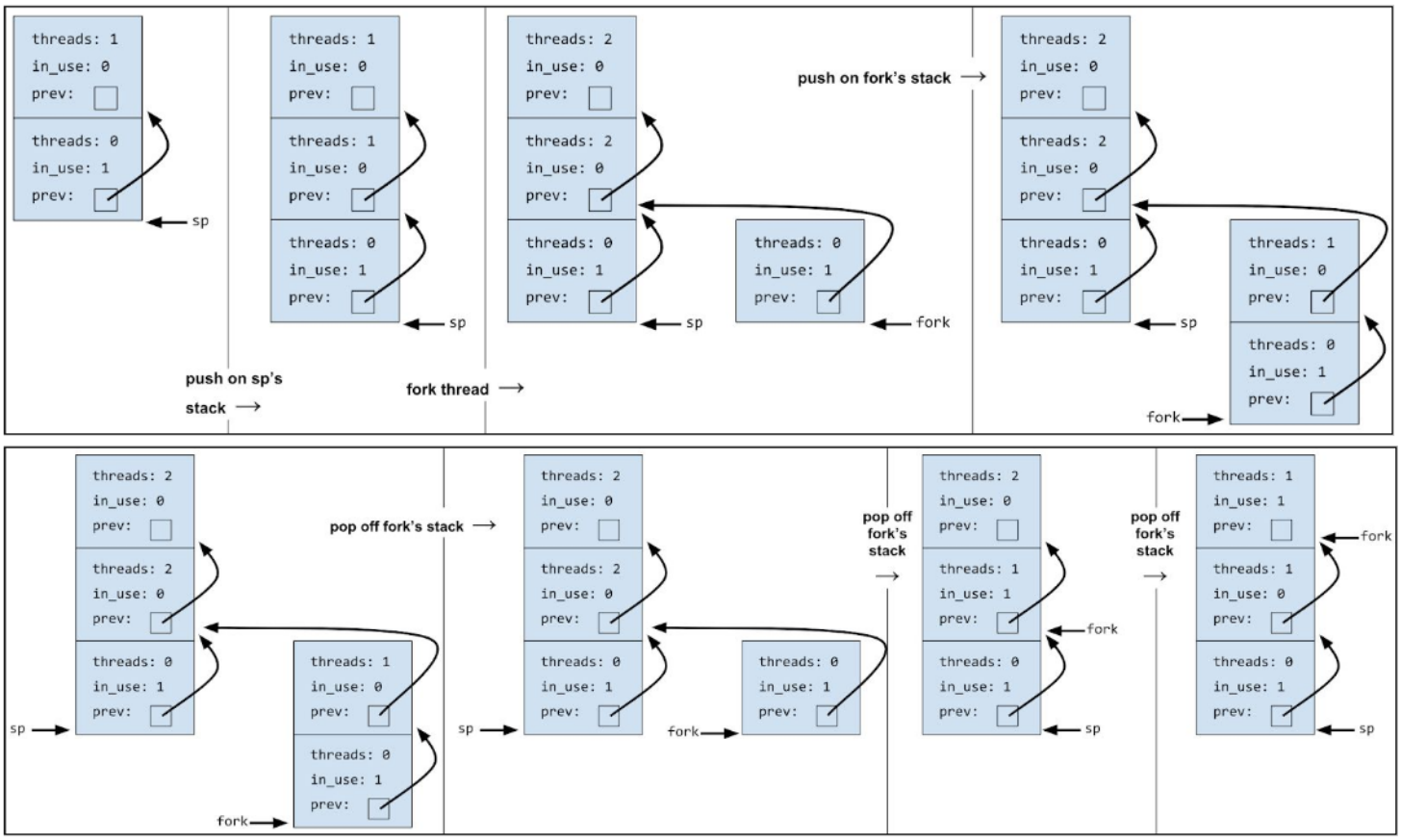
2) What would `frame_size ("main_stacks")` return?

2 * 4B + 8B = 16 B

Recall that for a single thread, its Stack section is essentially a linked list where the stack pointer **sp** points to the end of the stack. Also recall that when you **fork** a thread, the new thread will have their own copy of the stack. Instead of actually copying the function frames, we just point to the "shared" stack frames.

On the next page is a visual of the stack that we will be designing. The first set is an example of pushing function frames on the stack along with creating a new fork. The second set is continuously popping off from the fork's stack.

SID: _____



Fill in the following code sequence that will push the function frame onto the stack and modify the `sp` passed in. Assume `sp` is never `NULL` and `func_name` is always stored in static data—don't allocate space for it.

```
void push (char *func_name, StackNode **sp) {
      StackNode *temp = (StackNode *) malloc(sizeof(StackNode));
      temp->frame = malloc( frame_size(func_name) );
      temp->prev = *sp;
      temp->func_name = func_name;
      temp->threads = 0;
      temp->in_use = 1;
      if (*sp == NULL) {
            *sp = temp;
            return;
      }
      if ((*sp)->in_use > 0) {
            (*sp)->threads += 1;
            (*sp)->in_use -= 1;
      } else {    /* We are making a fork. */
            for (StackNode *trace = *sp; trace != NULL; trace = trace->prev) {
                  trace->threads += 1;
            }
      }
      *sp = temp;
}
```

SID: _____

Fill out the following code sequence that will pop the function frame off the stack, modifying the `sp` passed in. Also assume that `sp` is never `NULL`. Since our stack frame is possibly shared by multiple "threads" you will want to consider when we should free the memory for a frame we pop off.

```
void pop (StackNode **sp) {
          printf("%s has been returned\n", (*sp)->func_name);
          (*sp)->in_use--;
          bool free_frame = false;
          if ( (*sp)->threads == 0 && (*sp)->in_use == 0 ) {
             free_frame = true;
          }
          StackNode *temp = (*sp)->prev;
          if (free_frame) {
              free((*sp)->frame);
              free(*sp);
          }
          *sp = temp;
          if ((*sp) != NULL) {
              (*sp)->threads -= 1;
              (*sp)->in_use += 1;
          }
}
```

Problem 2 ***[MT1-2] Allocating an Order*** **(10 points)**

You are working on an e-commerce platform. Internally, orders are tracked through a struct called `order_t`. Your task is to write a function to allocate and initialize a new order. There's a catch though! This platform must be robust to errors, so you are required to return an error value from this function in addition to the newly allocated order. The possible errors are defined for you as preprocessor directives.

(a) **Write** `new_order`: Fill in the following code. Keep in mind the following requirements:

- You must return `BAD_ARG` if any inputs are invalid. The criteria for valid arguments is:
  - Unit price should be positive (no negative prices)
  - An order cannot be for more than `MAX_ORDER` items
  - Inputs must not cause your function to crash (or execute undefined behavior)
- You must return `NO_MEM` if there are any errors while allocating memory
- The tax rate is always initialized to `TAX_RATE`
- If your function returns `OK`, then `new` points to a valid struct that has been initialized with the provided values.

```
typedef struct order {
    int quantity;
    double unit_price;
    double tax_rate;
} order_t;

#define OK 0          /* Function executed correctly */
#define NO_MEM 1     /* Could not allocate memory for order */
#define BAD_ARG 2     /* An invalid argument was given */

#define TAX_RATE 1.08
#define MAX_ORDER 100
```

Solution:

```
/* Allocate and initialize a new order */
int new_order(order_t **new, int quantity, double unit_price) {
     /* Validate Arguments */
     if (new == NULL || quantity > MAX_ORDER || unit_price < 0)
          return BAD_ARG;

     /* Allocate "new" */
     *new = malloc(sizeof(order_t));
     if(*new == NULL) {
          return NO_MEM;

     /* Initialize "new" */
     (*new)->unit_price = unit_price;
     (*new)->quantity = quantity;
     (*new)->tax_rate = TAX_RATE;

     return OK;
}
```

(b) **Calling** `new_order`: How would you use `new_order()` to allocate and initialize `blue_monday` with a quantity of 10 and a unit price of 3.50 in the example below?

Solution:

```
order_t *blue_monday;
double total;
ret_t ret;

/* Fill in the arguments to new_order here */

ret = new_order(&blue_monday, 10, 3.50);

if (ret == OK) {
     printf("Total:  %lf\n",
               (blue_monday->unit_price *
                    blue_monday->quantity * blue_monday->tax_rate));
} else {
     printf("Error\n");
}
```

Question 4: The cat videos need better captions!! (C Programming) - 18 pts

We've decided that YouTube isn't doing a good enough job of captioning its videos, and we're going to do it for them. To do so, we created a structure that holds a caption and a structure that holds an array of these captions (enough for the entire video).

```
typedef struct {
      char* text; // pointer to a valid, null-terminated C string
      int timestamp;
} caption_t;

typedef struct {
      caption_t* array; // pointer to "length" consecutive caption_t structs
      int length;
} video_caption_t;
```

1. What is the size of each of the following variables in bytes? (i.e., what would `sizeof` return?)
   We are on a 32-bit architecture, and you can assume that `sizeof(int) == 4`. You can leave your answer in the form of an equation rather than multiplying out.

   You may not use `sizeof` as part of your answer.

   `caption_t many_captions[100];` ____(4+4)*100 [100 structs each of size 8]_____

   `caption_t* many_captions_ptr[200];` __4*200 [200 pointers, each of size 4]_____

   `caption_t** many_captions_double_ptr;` _____4 [1 pointer of size 4]_____

   `video_caption_t whole_video_captions;` _____8 [1 struct of size 8]_____

   `video_caption_t* whole_video_caption_ptr;` ___4 [1 pointer of size 4]_____

2. Implement the following function to find the length of the longest caption in a video. This length excludes the null terminator (i.e., it's a count of the number of characters in the text of the caption).

   You can use any functions in the `string.h` library to help you with this task. You can assume that all the memory for the input argument has been properly allocated and that all strings within it are null terminated. You may not need all lines for your code.

```
#include <string.h>

typedef struct {
      char* text; // pointer to a valid, null-terminated C string
      int timestamp;
} caption_t;

typedef struct {
      caption_t* array; // pointer to "length" consecutive caption_t structs
      int length;
} video_caption_t;

/* Returns the length of the longest caption text in the video.
 * (length does not include the null terminator) */
int longest_caption(video_caption_t* video) {
      int max_len = 0;


      for (int i=0; ________ i<video->length_____________________________; i++) {


            _int length = strlen(video->caption_array[i].text);_; // be careful of -> and . here


            if (_____length > max_len_____________________________) {


                  __________max_len = length;_____________________________;
            }
      }


      return ____________________max_len_____________________________;
}
```

3. Implement the following function to combine the captions from two videos together into a new `video_caption_t` structure and return a pointer to it. **Do not modify the contents of the input arguments**. You can copy the pointers to the text strings, and do not need to copy their characters. You can use any functions in `stdlib.h`. You can assume that all the memory for input arguments has been properly allocated and that all strings within them are null terminated. You may use `sizeof` when calculating sizes. You may not need all lines for your code.

```
#include <stdlib.h>

typedef struct {
      char* text; // pointer to a valid, null-terminated C string
      int timestamp;
} caption_t;

typedef struct {
      caption_t* array; // pointer to "length" consecutive caption_t structs
      int length;
} video_caption_t;

/* Concatenates two video captions together into a single new video caption structure */
video_caption_t* combine_captions(video_caption_t* video_one, video_caption_t* video_two){


      video_caption_t* new_video = malloc(______sizeof(video_caption_t)_________);
      int first_len = video_one->length;
      int second_len = video_two->length;
      new_video->length = first_len + second_len;


      new_video->array = malloc(________(first_len+second_len)*sizeof(caption_t)________);
      for (int i=0; i<first_len; i++) {


            ___________new_movie->caption_array[i] = first_movie->caption_array[i]___________;


            ____(can also assign members separately for full points)_________;
      }
      for (int i=0; i<second_len; i++) {


            _________new_movie->caption_array[first_len+i] = second_movie->caption_array[i]_____;


            _________(Be careful about -> and . here. Also use first_len+i)_____;
      }
      return new_video;
}
```