University of California, Berkeley – College of Engineering
Department of Electrical Engineering and Computer Sciences
Summer 2019   Instructors: Branden Ghena, Morgan Rae Reschenberg, Nicholas Riasanovsky   2019-07-08

CS61C MIDTERM 1 SOLUTIONS

***Last** Name (Please print clearly)*				
***First** Name (Please print clearly)*				
Student ID Number				
Circle the name of your Lab TA	**Ayush Maganahalli** **John Yang**	**Chenyu Shi** **Lu Yang**	**Gregory Jerian** **Ryan Searcy**	**Jenny Song** **Ryan Thornton**
Name of the person to your: Left | Right				
All my work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CS61C who haven't taken it yet. ***(please sign)***				

Instructions

- This booklet contains 14 pages including this cover page. **The back of each page of this exam is blank and can be used for scratch work, but will not be graded**.
- Please turn off all cell phones, smartwatches, and other mobile devices. Remove all hats and headphones. Place *everything* except your writing utensil(s), cheat sheet, and beverage underneath your seat.
- You have 80 minutes to complete this exam. The exam is closed book: no computers, tablets, cell phones, wearable devices, calculators, or cheating. You are allowed one page (US Letter, double-sided) of *handwritten* notes.
- There may be partial credit for incomplete answers; write as much of the solution as you can.
- Please write your answers within the boxes and blanks provided within each problem!
- For all coding questions you may only use functions from .h files that **have already been included** and any functions you use must work on **all common platforms** (Windows, Mac, Linux)

Question	**1**	**2**	**3**	**4**	**5**	**6**	**Total**
Possible Points	17	14	16	18	16	9	90

If you have the time, **feel free to doodle on the front page!**

Question 1: We're bored of Euclid. (Number Representation) - 17 pts

Morgan, Nick, and Branden are disappointed in the selection of food available near Soda, so they open a store selling many different kinds of products. They need YOUR help to come up with a barcode scheme for everything they sell.

1. Branden wants to assign each product in the store its own unique number. This will be encoded as the barcode. If they sell 29 unique items, what is the smallest number of bits they can use to encode the barcode?

Product Number

```
Barcode = _____5 (ceiling(log2(29)))________ bits
```

2. When the store runs out of a particular item, it would be helpful to see what other kinds of that item there are in stock. Morgan proposes adding a "product group" field to the barcode in addition to the existing product number. Note that now each product number does not need to be globally unique and instead just needs to be unique within its product group. If there are 5 unique product groups, what is the smallest number of bits they can use for the product group field?

Product Group	Product Number

```
Product Group = ____3 (ceiling(log2(5)))____ bits
```

3. We expand to have 12 product groups. The largest has 15 items in it while the smallest has one item. Nick argues the entire barcode can now be condensed to only 6 bits without losing product grouping or unique identifiers. Is he correct? If yes, explain why, if no, what is the actual minimum size?

   `[ ] Yes, he is correct` `[X] No, he is incorrect`

   ____At least 4 bits for items ceiling(log2(15))____________________

   ____At least 4 bits for groups ceiling(log2(12))____

   ____So it can't be less than 8 bits____

4. The team decides on the following barcode field sizes (which may or may not reflect your answers above).

Product Group (4 bits)	Product Number (5 bits)

   Morgan loads all the barcodes into the database but runs into a problem; she'd like to reserve the barcode of all zeros (so, product group = 00...0, product number = 00...0) for products that are out of stock. Assuming there are 8 product groups holding between 1 and 31 products each, can she implement the all-zero barcode without adding bits to the scheme? Explain.

   `[X] Yes, she can` `[ ] No, she can't`

   ____There are 4 bits for product group (possible 2^4 = 16 groups)_____________________________

   _____But only 12 groups are used, leaving 4 possible groups unclaimed______________________

   __So, we just make product group = 0b0000 the "empty group", and product number 0b00000 is in it__

5. Business is booming and the team has the opportunity to expand! They purchase a new store and modify the barcode to keep track of products sold at store 0 and store 1 separately.

Store Code (1 bit)	Product Group (4 bits)	Product Number (5 bits)

   Assuming the same item sold across stores differs ONLY in the top bit (that is, they have the same product group and product number regardless of the store they're sold at) what is the maximum number of items Morgan, Nick, and Branden can uniquely identify with this barcode scheme? You may leave your answer as an unsimplified equation.

   `_2^9 = 512 (Unique items are product group and product number, not store)_ Unique Items`

Question 2: Remember remember the segments of memory. - 14 pts

```
#include <stdlib.h>
#include <stdbool.h>

bool fetch_data (char* buf);

char* receive_buffer;
bool is_complete = false;

int main(int argc, char* argv[]) {
      receive_buffer = malloc(100*sizeof(char));
      if (!receive_buffer) {
            return -1;
      }
      fetch_data(receive_buffer);
      free(receive_buffer);
      return 0;
}

/* Function that takes in a buffer for storing characters
 * and places data in the buffer by calling receive_data.*/
void fetch_data(char* buf) {
      int len = receive_data(receive_buffer);
      if (len == 0) {
            return; // HERE
      } else {
            fetch_data(buf + len);
      }
}
```

All of the following expressions **evaluate** to an address value. State in which region of memory each value corresponds to: stack, heap, static, or code. Assume we are about to execute the line marked `HERE`.

1.	receive_buffer	Ⓐ Code	Ⓑ Static	Ⓒ Stack	[✓ Ⓓ Heap] (set by malloc(100…))
2.	&(receive_buffer[0])	Ⓐ Code	Ⓑ Static	Ⓒ Stack	[✓ Ⓓ Heap] (same as question 1)
3.	&receive_buffer	Ⓐ Code	[✓ Ⓑ Static]	Ⓒ Stack	Ⓓ Heap (is a global variable)
4.	&argc	Ⓐ Code	Ⓑ Static	[✓ Ⓒ Stack]	Ⓓ Heap (arguments are local on stack)
5.	&is_complete	Ⓐ Code	[✓ Ⓑ Static]	Ⓒ Stack	Ⓓ Heap (is a global variable)
6.	&fetch_data	[✓ Ⓐ Code]	Ⓑ Static	Ⓒ Stack	Ⓓ Heap (function pointers in code)
7.	buf	Ⓐ Code	Ⓑ Static	Ⓒ Stack	[✓ Ⓓ Heap] (buf was receive_buffer passed in, so this is the same as question 1)

Now consider the following different program:

```
// function prototypes
int a(int x);
int b(int y);

int main(void) {
      a(3);
      return 0;
}

int a(int x) {
      return b(x - 1);
}

int b(int y) {
      if (y <= 1) {
            return 7; // HERE
      } else {
            return a(y - 1);
      }
}
```

Assume we are about to execute the line marked `HERE`. Label all the stack frames below with either the function that created the frame: a, b, or main, or with UNUSED if the frame is not in use.

Main calls A(3) calls B(2) calls A(1) calls B(0) which hits the \`return 7\` line

Memory (Top)

1	1.	Ⓐ a	Ⓑ b	[✓ Ⓒ main]	Ⓓ UNUSED
2	2.	[✓ Ⓐ a]	Ⓑ b	Ⓒ main	Ⓓ UNUSED
3	3.	Ⓐ a	[✓ Ⓑ b]	Ⓒ main	Ⓓ UNUSED
4	4.	[✓ Ⓐ a]	Ⓑ b	Ⓒ main	Ⓓ UNUSED
5	5.	Ⓐ a	[✓ Ⓑ b]	Ⓒ main	Ⓓ UNUSED
6	6.	Ⓐ a	Ⓑ b	Ⓒ main	[✓ Ⓓ UNUSED]
7	7.	Ⓐ a	Ⓑ b	Ⓒ main	[✓ Ⓓ UNUSED]

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 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;
}
```

Question 5: I GOT RISC-V ON IT. - 16 pts

Fill in the blanks to implement `strncpy` in RISC-V. You may not need all lines.

`char* strncpy(char* destination, char* source, unsigned int n);`

`strncpy` takes in two `char*` arguments and copies up to the first n characters from `source` into `destination`. **If it reaches a null terminator, then it copies that value into `destination` and stops copying in characters**. If there is no null terminator among the first n characters of `source`, the string placed in `destination` will not be null-terminated. You may not store anything on the stack for this problem.

`strncpy` returns a pointer to the `destination` string.

Hint: Assume the calling convention earned in lecture!

```
strncpy:

      add t0 x0 x0 # Current length

      _(Students could have other solutions, e.g. may save a0 here and restore it after end)_

Loop:

      beq t0 a2 End # Check if we've iterated over the length of the string

      add t1 a1 t0 # calculate the address of the t0-th (count-th) char in the string

      lb t2 0(t1) # load this char

      add t1 a0 t0 # calculate the same address, but off of the dst string

      sb t2 0(t1) # store into the dst string

      addi t0 t0 1 # increment our counter

      bne t1 x0 Loop # check if the loaded char was null

End:

      Jr ra
```

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.