<u>Note: this exam is double sided, make sure to check all pages!</u>

CS107 Practice Midterm Exam

This is a closed book, closed note, closed electronic device exam, except for one double-sided US-Letter-sized (8.5"x11") page of your own prepared notes which you may refer to during the exam. You have 120 minutes to complete all problems. You don't need to **`#include`** any header files. For coding questions, the majority of the points are typically focused on the correctness of the code. However, there may be deductions for code that is roundabout/awkward/inefficient when more appropriate alternatives exist. For any coding questions, your answers should compile cleanly and not have any memory leaks or errors. Solutions that violate any specified restrictions may get partial credit. Style is secondary to correctness (e.g., there are no style deductions for using magic numbers). There is 1 point per minute of the exam.

Good luck!

SUNet ID (username): ________________**`@stanford.edu`**

First Name: ______________________________

Last Name: ______________________________

I accept the letter and spirit of the honor code.

[signed] ____

Problems:

1. Bits, Bytes and Numbers	16 points
2. C Strings	30 points
3. Pointers and Generics	54 points
4. Using **`qsort`**	10 points
5. **`void *`** and Function Pointers	10 points
	120 points total

Problem 1: Bits, Bytes and Numbers [16 points]

For this question, refer to the following **mystery** function:

```
unsigned char mystery(unsigned char n) {
     n |= n >> 1;
     n |= n >> 2;
     n |= n >> 4;
     n++;
     return (n >> 1);
}
```

A) [6 points] What does the following code print?

```
printf("%u\n", mystery(17));      // %u prints the integer value
printf("%u\n", mystery(88));      // for an unsigned char
printf("%u\n", mystery(150));
```

B) [5 points] For which values of **`n`** does **`mystery(n)`** return non-zero?

C) [5 points] When **`mystery(n)`** returns a non-zero value, what is the general bit pattern of the result? In other words, explain the return value in terms of the argument **`n`**.

Problem 2: C Strings [30 points]

The function

```
char *substr(const char *s, char start, char stop, char result[])
```

fills **`result`** with the substring that starts at the first instance of the **`start`** character and ends at the next instance of the **`stop`** character. The **`result`** buffer is guaranteed to be big enough to hold the substring, and the function should properly null-terminate **`result`**. If there isn't a substring that meets the criteria, **`result`** should contain the empty string. The **`result`** buffer is also returned to the calling function.

Here are some examples:

```
char *input = "Mississippi";
char buffer[strlen(input) + 1];

substr(input, 'i', 'p', buffer);     // fills buffer with "ississip"
substr(input, 's', 'i', buffer);     // fills buffer with "ssi"
substr(input, 's', 's', buffer);     // fills buffer with "ss"
substr(input, 'p', 's', buffer);     // fills buffer with empty string
```

Requirements:

- Your function should not allocate, deallocate, or resize any memory.
- Re-implementing functionality that is available in the standard library will result in loss of credit. For example, your code cannot have *any* explicit loops! Instead, call the library functions!

A) [25 points] Implement the **`substr`** function on the next page.

```
char *substr(const char *s, char start, char stop, char result[]) {
```

B) [5 points] Your colleague decides that it would make more sense to have a correctly-sized **`result`** buffer so you don't waste space. They suggest adding the following code before returning from the function (after **`result`** has been populated correctly):

```
// note: caller is responsible for freeing returned pointer
char *new_buffer = malloc(strlen(result));
strcpy(new_buffer, result);
free(result);
return new_buffer;
```

While you are happy that your colleague has left a nice comment about the caller being responsible for freeing the memory, you see two problems in the code. One problem is definitely an error, and the other problem has a big potential to be an error. Identify these two problems.

This page intentionally left blank.

Problem 3: Pointers and Generics [54 points]

Even though C doesn't have classes or data structures, with generics and structs we can approximate this functionality, with functions we write that use this struct. For this problem, you will be implementing code for part of a generic C queue, which has first-in-first-out behavior. The queue elements will be stored as a linked list of nodes:

```
typedef struct node {
    struct node *next;
    void *data;
} node;
```

The **queue** definition is as follows. Note that there is both a front and a back in a queue, and elements are enqueued onto the back of the queue, and dequeued from the front:

```
typedef struct queue {
    int width;
    node *front;
    node *back;
} queue;
```

The **queue_create** function initializes a **queue**:

```
queue *queue_create(int width) {
    // note: caller responsible for freeing queue
    queue *q = malloc(sizeof(*q));
    q->width = width;
    q->front = NULL;
    q->back = NULL;
    return q;
}
```

The **queue_enqueue** function works by copying the data into a **node**, and it **does not simply copy the pointer location**. The function looks like this:

```
// addr is where q->width bytes of data are to be copied from and
// stored into a queue node
void queue_enqueue(queue *q, const void *addr) {
    node *new_node = malloc(sizeof(*new_node));
    new_node->data = malloc(q->width);
    memcpy(new_node->data, addr, q->width);
    new_node->next = NULL;
    if (q->front == NULL) {
        q->front = new_node;
    } else {
        q->back->next = new_node;
    }
    q->back = new_node;
}
```

A) [35 points] Write the **`queue_dequeue`** function. It should take in a pointer to a **`queue`** and a **`void *addr`**, which is a pointer to an address that can hold **`queue->width`** bytes from a queue node. The data in the node at the front of the queue should be copied to the address pointed to by **`addr`**, and the node at the front of the queue should be removed and deallocated. The function should return true if the queue has any elements when called, or false if the queue is empty when called.

```
bool queue_dequeue(queue *q, void *addr) {
```

B) [19 points] In assign3, you wrote a **`mytail`** program with a circular queue of a fixed size. Another way to write the program would have been with the generic queue you just created. Fill in each of 8 blanks in the **`main`** function on the next page. Your program should *not leak any memory*. (For this program, assume that all input lines end with a newline **`\n`** character).

```
int main(int argc, char *argv[]) {

    char buffer[1024];

    int nlines = atoi(argv[1]);

    FILE *fp = fopen(argv[2], "r");

    queue *q = queue_create(_____________________________); // line 1

    int lines_read = 0;

    char *line;

    while (fgets(buffer, sizeof(buffer), fp)) {

        buffer[strlen(buffer) - 1] = '\0';

        // Make a persistent copy of the line and
        // enqueue into the queue.
        line = __________________________________________;  // line 2

        queue_enqueue(q, ________________________________);  // line 3

        if (++lines_read > nlines) {

            queue_dequeue(q, ___________________________);  // line 4

            _____________________________________________;  // line 5

        }

    }

    fclose(fp);

    while (queue_dequeue(q, ___________________________)) { // line 6

        printf("%s\n" ,line);

        _________________________________________________;  // line 7

    }

    _____________________________________________________;  // line 8

    return 0;

}
```

Problem 4: Using `qsort` [10 points]

Assume the following definition of a date:

```
typedef struct date {
    int month;
    int year;
} date;
```

Dates are compared first by year, and if year is the same, then compared by month. For example, **`{5,2018}`** (May 2018) is less than **`{6,2018}`** (June 2018), and **`{11,2000}`** (Nov 2000) is less than **`{4,2018}`** (April 2018). Implement the **`cmp_date`** comparison callback that can be used with **`qsort`** to sort an array of **`date`**s, as in the code below.

```
int main(int argc, char *argv[]) {
    struct date dates[] = {{1,2000}, {6,2018}, {2,2018}, {1,2005},
                           {8,2007}};
    int n = sizeof(dates) / sizeof(dates[0]);

    qsort(dates, n, sizeof(*dates), cmp_date);
    for (int i = 0; i < n; i++) {
        printf("%d/%d\n", dates[i].month, dates[i].year);
    }
    return 0;
}
```

```
int cmp_date(const void *a, const void *b) {
```

Problem 5: `void *` and Function Pointers [10 points]

The **`map`** function applies a client-supplied callback function to each element in a generic array. The sample code demonstrates using **`map`** to apply a callback function that adds one to each array element.

```
void increment(void *a) {
    int *pnum = (int *)a;
    (*pnum)++;
}

int main(int argc, char *argv[]) {
    int arr[] = {5, 8, 2, 0};
    int n = sizeof(arr) / sizeof(arr[0]);

    map(arr, n, sizeof(*arr), increment);
    // now arr holds {6, 9, 3, 1};
    ...
}
```

Implement the generic map function.

```
void map(void *arr, int n, size_t width, void (*fn)(void *)) {
```