This document contains the questions to the CS107 midterm given in Winter 2018 by instructor Chris Gregg. This was a 120-minute exam.

Midterm questions

Problem 1: Bits, bytes, and numbers

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

1a) 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));
```

1b) For which values of `n` does `mystery(n)` return non-zero?

1c) 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

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!

2a) Implement the `substr` function.

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

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.

2b) Identify these two problems.

Problem 3: Pointers and generics

In class, we discussed a generic stack, with last-in-first-out behavior. For this problem, you will be creating a generic *queue*, which has first-in-first-out behavior. The queue elements will be stored as a linked list of nodes:

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

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;
    struct node *front;
    struct 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) {
    struct 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;
}
```

3a) Write the `queue_dequeue` function:

```
/* return value: true if queue has any elements when called,
                 false if queue is empty when called
   addr: pointer to address that can hold queue->width
         bytes from queue node. The data in the node
         at the front of the queue should be copied
         to the address pointed to by addr, and node
         should be removed from the queue and deallocated.
*/

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

3b) In assign3, you wrote a *tail* 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 below. Your program should *not leak any memory*.

```
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
Assume the following definition of a date:

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

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 `dates`, 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

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 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 *)) {
```