Practice Final

- Like the midterms, the final is closed-book, closed technology.
- The examination period is 180 minutes.
- The final covers everything that we have covered in the course.
- The final will have 10 questions in total—think two midterms stapled that you have three hours to complete.
- There are be 6 non-programming questions and 4 programming questions.
- Like the midterms, 60% of the points are in the non-programming questions and 40% of the points are in the programming questions.
- The questions on this practice final are more difficult than those you will find on the final.
- All relevant C APIs and documentation will be given to you for the exam.
- For this exam, we will use Gradescope to streamline grading and feedback at the end of the semester. Because of this, the exam will be double-sided with two additional sheets of scratch paper available at the end of the exam. Please make sure to not detatch any pages from the exam and clearly indicate where your work for each problem is located if it is not obvious.

Solutions!
Heaps of Stack (Diagrams)

Give complete stack-and-heap diagrams outlining the state of memory at each of the given points below, starting execution in main. The style of your diagrams should be identical to the style presented in class. You should reproduce the diagrams completely at each point of the program.

```c
void f(char *arr, int n) {
    for (int i = 0; i < n; i++) {
        arr[i] = arr[i] + 1;  // the next char
    }
    // Point C
}

void g(char *arr, int *n) {
    char buf[4] = { 'd', 'e', 'f', 'g' };  // Point B
    *n = 4;
    f(buf, *n);
}

int main(void) {
    char arr[3] = { 'a', 'b', 'c' };  // Point A
    int n = 3;
    g(arr, &n);  // Point D
    // Point C
}
```

// Diagram for Point A

// Diagram for Point B

// Diagram for Point C

// Diagram for Point D
typedef struct {
    int* v1;
    int* v2;
} cell_t;

void f(cell_t c) {
    *c->v1 = *c->v1 + *c->v2;
    *c->v2 = *c->v1 + *c->v2;
    // Point C
}

void g(cell_t *c) {
    int z = 11
    // Point B
    *c->v1 = *c->v1 + *c->v2;
    c = (cell_t*) malloc(sizeof(cell_t));
    c->v1 = &z;
    c->v2 = &z;
    f(c);
}

int main(void) {
    int x = 3;
    int y = 7;
    cell_t *c = (cell_t*) malloc(sizeof(cell_t));
    c->v1 = &x;
    c->v2 = &y;
    // Point A
    g(c);
    // Point D
}

// Diagram for Point A

// Diagram for Point B

// Diagram for Point C

// Diagram for Point D

(memory | c.k.)
Reasons

Consider the following C function:

```c
int calculate(int x) {
    int y;
    scanf("%d", &y);
    int count = 0;

    // Point A
    while (y < x) {
        if (y == 0) {
            count++;
            // Point B
        }
        scanf("%d", &y);
    }

    // Point C
    return count;
}
```

For each of the propositions below, determine if the proposition never holds (\(\times\)), sometimes holds (\(?\)), or always holds (\(\checkmark\)) at the given program points.

<table>
<thead>
<tr>
<th></th>
<th>(y &lt; x)</th>
<th>(y == 0)</th>
<th>(\text{count} &gt; 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>?</td>
<td>?</td>
<td>(\times)</td>
</tr>
<tr>
<td>B</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>C</td>
<td>(\times)</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

In a sentence or two, describe what the function computes.

\(\text{calculate (x)}\) counts the number of times the user types in \(0\) before typing a value that is at least \(x\).
Represent Yourself Now.

Consider the following number written in base-10: 135.

(a) Write this number in base-2 (binary) using eight bits (interpreted as an unsigned integer).

```
1000011
```

(b) If we interpreted your answer in part (a) as a signed integer represented in twos complement, what is its value in base-10?

```
011110001
```

(c) Consider the given variables containing integer values (written in base-2 for convenience): \( x = 11100, y = 00100 \). What are the results of the following expressions over these variables?

\[
x \& y \quad x \mid y
\]

```
00100
11100
```

\[
x \ll 3 \quad x ^ y
\]

```
00000
11000
```

Consider a bit set defined by the three bit masks:

\[
\#define A 1 \mid 001 \\
\#define B 2 \mid 010 \\
\#define C 4 \mid 100
\]

Write expressions using bitwise operators that perform the following operations:

(d) Checks to see if either option A or option C are set for an integer variable \( x \).

\[
(A \mid C) \& x
\]

(e) Sets all three options from an empty bitset.

\[
A \mid B \mid C
\]
To Err is Human

Consider the following program:

typedef struct {
    char* data [100];
    int sz;
} stack_t;

stack_t* make_stack() {
    stack_t *ret = (stack_t*) malloc(sizeof(stack_t));
    ret->sz = 0;
    return ret;
}

void free_stack(stack_t* s) {
    for (int i = 0; i < s->sz; i++) {
        free(data[i]);  \[Double Free\]
    }
    \[Memory leak\]
    void stack_push(stack_t* s, char *v) {
        s->data[s->sz++] = v;
    }

int main(void) {
    char *t = (char*) malloc(sizeof(char) * 8);
    strcpy("element", t);
    stack_t *s = make_stack();
    stack_push(s, t);
    stack_push(s, t);
    free_stack(s);
    free(t);
}

(a) List the memory errors (dangling pointer, buffer overflow, memory leak, double free, use-after-free) in the above code. For each such memory error, write one sentence describing the error and its cause with respect to the program.

(b) For each of the memory errors you listed, fix the code above so that it no longer has that memory error. You may inline your changes in the code above, striking out relevant code and using arrows to insert relevant snippets as necessary.
One Last Time

Let eucl_dist and make_picture be functions defined by

double eucl_dist(int x1, int y1, int x2, int y2) {
    int dx = x1 - x2;
    int dy = y1 - y2;
    return sqrt(dx*dx + dy*dy);
}

Picture make_picture(int x, int y, double r) {
    Picture pic;
    pic.height = 192;
    pic.width = 256;
    for (int i = 0; i < pic.height; i++) {
        for (int j = 0; j < pic.width; j++) {
            double dist = eucl_dist(i, j, x, y);
            if (dist < r) {
                pic.pix_array[i][j] = {255,0,0};
            } else if (i < y) {
                pic.pix_array[i][j] = {0,255,0};
            } else {
                pic.pix_array[i][j] = {0,0,255};
            }
        }
    }
    return pic;
}

(a) Describe in your own words what make_picture is doing.

Draws a red circle centered at (x,y).
Also color the pixels above the circle green,
and those below the circle blue.
The radius of the circle is r.

(b) Draw the pictures generated by the calls make_picture(128,96,64) and make_picture(0,0,192).
Be sure to indicate the color of the different regions of the pictures.
A New Job in Management

Consider the following implementation of the list ADT:

```c
typedef struct node {
    char *value;
    struct node *next;
} node_t;

typedef struct {
    node_t *first;
} list_t;

node_t* make_node(char* value, node_t *next) {
    node_t *ret = (node_t*) malloc(sizeof(node_t));
    ret->value = (char*) malloc(sizeof(char) * (strlen(value) + 1));
    strcpy(ret->value, value);
    ret->next = next;
    return ret;
}

list_t* make_list() {
    list_t *ret = (list_t*) malloc(sizeof(list_t));
    ret->first = NULL;
    return ret;
}

void add_to_front(list_t *l, char* s) {
    l->first = make_node(s, l->first);
}
```

(a) Imagine a client using this list library. For each of the following pieces of data, determine if it is the library’s or the client’s responsibility to initiate the freeing of that data.

- A list_t: __client__
- The next field of the list_t: __library__

(b) On the following page, provide definitions for the following functions given the memory management policy you outlined above:

- void free_node(node_t *n)
- void free_list(list_t *l)
(Your definitions of free_node and free_list go here.)

```c
void free_node (node_t *n)
{
    free (n->value);
    free (n);
}

void free_list (list_t *lst)
{
    node_t *cur = lst->first;
    while (cur != NULL)
    {
        node_t *next = cur->next;
        free_node (cur);
        cur = next;
    }
    free (lst);
}
```
String 'em Up

On the next page, write a function char* tokenize(char *s) that takes a string s and splits the string up into its next token, delimited by whitespace. Here we'll define a whitespace character to be one of ' ', '	', '
', ''. tokenize proceeds by advancing the input pointer as follows:

- First, tokenize skips over all initial whitespace that it finds.
- Once tokenize finds a non-whitespace character, it skips over all non-whitespace characters it finds.
- Once tokenize encounters a whitespace character, it overwrites that character with a null terminator (\0) and advances one position forward.
- Finally, tokenize continues skipping over whitespace until it encounters another non-whitespace character and returns a pointer to that non-whitespace character.

If our pointer ever points to a pre-existing null-terminator character in this process, we return the pointer immediately (i.e., a pointer to that null-terminator).

As an example, consider the following string (written in an array style) where the input pointer initial points to the first character:

[' ', ' ', ' ', ' ', 'h', 'e', '1', '1', 'o', ' ', ' ', ' ', ' ', 'w', 'o', 'r', '1', 'd', '

First we advance the input pointer to the first non-whitespace character, 'h' :

[' ', ' ', ' ', ' ', 'h', 'e', '1', '1', 'o', ' ', ' ', ' ', ' ', 'w', 'o', 'r', '1', 'd', '

We then advance over the non-whitespace characters we find until we hit the first space after 'o' :

[' ', ' ', ' ', ' ', 'h', 'e', '1', '1', 'o', ' ', ' ', ' ', ' ', 'w', 'o', 'r', '1', 'd', '

We overwrite this whitespace character with a null-terminator and then advance the pointer to the first non-whitespace character found after this new null-terminator which is 'w' :

[' ', ' ', ' ', ' ', 'h', 'e', '1', '1', 'o', ' ', ' ', ' ', ' ', 'w', 'o', 'r', '1', 'd', '

We then return our pointer which points to the string "hello" and the returned pointer points to the string "world";
(Write your implementation of tokenize here.)

```c
char *tokenize (char *s)
{
    while (is_whitespace (*s)) { s++; }
    while (!is_whitespace (*s)) {
        if (s == '101') { return s; }
        s++;
    }
    s = '101';
    while (is_whitespace (*s)) { s++; }
    if (s == '101') { return s; }
    return s;
}

bool is_whitespace (char c)
{
    return (c == ' ' || c == 't' || c == 'r' || c == 'n');
}
```
In n‘ Out

Write a function void report_averages(char *infile) that reads the text file denoted by infile that contains lines of single-digit numbers separated by spaces and reports the average of the numbers found on each line. For example, if the text file contains:

3 1 8 0 9
2 1 3
7 9 8 1 2 5 1

Then your program should print the following to the console:

4.2
2
4.714285714

(Note that the output is given as a double rather than an integer! The format specifier for double is %lf.) The following functions of the <stdio.h> header will help in creating this function:

• FILE* fopen(char *filename, char *mode). Opens a file for reading or writing. Recall that the mode for reading is "r".
• int fclose(FILE *stream). Closes a file.
• int fgetc(FILE *stream). Gets the next character from a file, returning EOF if the file has no more characters left.
• int fprintf(FILE *stream, char *fmt, ...). Prints the given formatted string to the given file.

To convert a single digit represented as a char c into an integer, using the expression c = '0'. Assume that the file is well-formatted, i.e., the numbers are all single-digit and only white space separates digits found on each line.

```c
void report_averages(char *infile)
{
    FILE *fp = fopen(infile, "r");
    int c = fgetc(fp), count = 0, sum = 0;
    while (c != EOF)
    {
        if (c == \n")
            printf("%d\n", (1.0 * sum) / count);
        sum = 0;
        count = 0;
    }
    else if (\'0\' <= c && c <= \'9\')
    {
        sum += c - \'0\';
        count += 1;
    }
    fclose(fp);
    close(fp);
}
```
Problem 9: A Link to the Past

Consider the following implementation of a linked list:

typedef struct node {
    int value;
    struct node *next;
} node_t;

typedef struct {
    node_t first;
} list_t;

list_t* make_list(void);
node_t* make_node(int value, node_t *next);

Write a function called interleave that takes two lists as input and constructs a new list that is the result of interleaving the two input lists together. For example, if the two input lists contain the values [1, 2, 3, 4] and [4, 5, 6, 7], then interleaving them together produces the lists [1, 4, 2, 5, 3, 6, 4, 7]. The output list should be independent of the input lists, i.e., it should not share any nodes in common with the input lists. If either of the input lists is longer than the second, then the remainder of the longer list’s elements appears at the end of the output list. For example, if we interleave [1, 2] and [3, 4, 5, 6], we obtain the list [1, 3, 2, 4, 5, 6]. You may not change the definition of the linked list or use any additional list functions beyond the ones specified above in your implementation.

```
list_t* interleave (list_t *l1, list_t *l2) {
    list_t* ret = make_list();
    node_t* cur1 = l1->first;
    node_t* cur2 = l2->first;
    while (cur1 != NULL || cur2 != NULL) {
        if (cur1 != NULL && cur2 != NULL) {
            ret->first = make_node(cur1->value, NULL);
            ret->first->next = make_node(cur2->value, NULL);
            cur1 = cur1->next;
            cur2 = cur2->next;
        } else if (cur1 == NULL) {
            ret->first = make_node(cur2->value, NULL);
            cur2 = cur2->next;
            while (cur1 != NULL && cur2 != NULL) {
                cur3 = cur2->next;
                cur2->next = make_node(cur1->value, NULL);
                cur1 = cur1->next;
                cur2 = cur3;
            }
        } else {
            ret->first = make_node(cur1->value, NULL);
            ret->first->next = make_node(cur2->value, NULL);
            cur1 = cur1->next;
            cur2 = cur2->next;
        }
    }
    return ret;
}
```
Problem 10: Line 'em Up

In the Hanoi project, we studied the stack abstract data type. A stack is a sequential container that provides first in, last out (FILO) access to its elements. That is, the first element we push onto the stack will be the last element that is popped. For this problem, we will study the queue abstract data type which, like a stack, is a sequential container. However, a queue provides first in, first out (FIFO) behavior; the first element enqueued onto the queue will be the first element that is dequeued.

As an example, consider first making a queue, initially empty:

[ ]

Suppose that we enqueue the elements 3, 8, 1, and 2. The queue then has the following shape:

[3, 8, 1, 2]

Where the left-hand side is the front of the queue and the right-hand side is the back (although this not need to always be the case). If we dequeue an element from the list, we will end up removing and returning the 3, leaving the remaining elements:

[8, 1, 2]

(a) Draw the state of a queue called q that is initially empty after performing the following function calls:

enqueue(q, 12)

enqueue(q, 9)

enqueue(q, 7)

dequeue(q) (also, what value does dequeue produce?)

enqueue(q, 8)

dequeue(q) (also, what value does dequeue produce?)
Next, we will implement the queue ADT in terms of our list ADT. Assume that you have a list implementation with the following interface:

- `list_t* make_list(void);`
- `void free_list(list_t *l);`
- `int list_size(list_t *l);`
- `void list_add_front(list_t *l, int v);`
- `void list_add_back(list_t *l, int v);`
- `int list_get(list_t *l, int index);`
- `int list_remove(list_t *l, int index);`

(b) Write the definition of a type called `queue_t` that represents a queue as described above.

```c
typedef list_t *queue_t; // A queue is a list
```

(c) Write a pair of functions, `queue_t* make_queue(void)` and `void free_queue(queue_t* q)` that creates an empty queue and frees a queue, respectively.

```c
queue_t* make_queue(void) {
    return make_list();
}

free_queue(queue_t* q) {
    free_list(q);
    free_queue(q);
}
```
(d) Write the queue_size function that reports the number of elements inside the queue.

```c
int queue_size(queue_t *q) {
    return list_size(q);
}
```

(e) Write the queue_enqueue function that enqueues an integer v to a queue q.

```c
void queue_enqueue(queue_t *q, int v) {
    list_add_back(q, v);
}
```

(f) Write the queue_dequeue function that dequeues the next element from the queue and returns it. As a pre-condition to the function, the queue must contain at least one element in it.

```c
// pre: size(q) > 0
int queue_dequeue(queue_t *q) {
    return list_remove(q, 0);
}
```