ECE 551D
Spring 2018
Midterm Exam

Name: SOLUTIONS NetID:

There are 6 questions, with the point values as shown below. You have 75 minutes with a total of 75 points. Pace yourself accordingly.

This exam must be individual work. You may not collaborate with your fellow students. However, this exam is open notes, so you may use your class notes, which must be handwritten by you.

I certify that the work shown on this exam is my own work, and that I have neither given nor received improper assistance of any form in the completion of this work.

Signature:

#	Question	Points Earned	Points Possible
1	Concepts		14
2	Reading Code		7
3	Fixing Valgrind Errors		12
4	Coding: Recursion		10
5	Coding: Arrays		14
6	Coding: String and More		18
	Total		75
	Percent		100

These two pages contain some reference (an abbreviated version of the man pages) for common, useful library functions.

- `char * strchr(const char *s, int c);`
  The `strchr` function locates the first occurrence of `c` (converted to a char) in the string pointed to by `s`. It returns a pointer to the located character, or `NULL` if the character does not appear in the string.

- `char * strdup(const char *s);`
  `char * strndup(const char *s, size_t n);`
  The `strdup` function allocates sufficient memory for a copy of the string `s`, does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function `free`. If insufficient memory is available, `NULL` is returned. `strndup` behaves the same way, except the length of the string to be copied is specified.

- `size_t strlen(const char *s);`
  The `strlen` function computes the length of the string `s` and returns the number of characters that precede the terminating `\0` character.

- `int strcmp(const char *s1, const char *s2);`
  `int strncmp(const char *s1, const char *s2, size_t n);`
  The `strcmp` and `strncmp` functions lexicographically compare the null-terminated strings `s1` and `s2`. The `strncmp` function compares not more than `n` characters. Because `strncmp` is designed for comparing strings rather than binary data, characters that appear after a `\0` character are not compared. These functions return an integer greater than, equal to, or less than 0, according as the string `s1` is greater than, equal to, or less than the string `s2`. The comparison is done using unsigned characters.

- `char *strstr(const char *s1, const char *s2);`
  The `strstr` function locates the first occurrence of the null-terminated string `s2` in the null-terminated string `s1`. If `s2` is an empty string, `s1` is returned; if `s2` occurs nowhere in `s1`, `NULL` is returned; otherwise a pointer to the first character of the first occurrence of `s2` is returned.

- `void * malloc(size_t size);`
  The `malloc` function allocates `size` bytes of memory and returns a pointer to the allocated memory.

- `void * realloc(void *ptr, size_t size);`
  The `realloc` function creates a new allocation of `size` bytes, copies as much of the old data pointed to by `ptr` as will fit to the new allocation, `free`s the old allocation, and returns a pointer to the allocated memory. If `ptr` is `NULL`, `realloc` is identical to a call to `malloc` for `size` bytes.

- `void free(void *ptr);`
  The `free` function deallocates the memory allocation pointed to by `ptr`. If `ptr` is a `NULL` pointer, no operation is performed.

- `int fgetc(FILE *stream);`
  The `fgetc` function obtains the next input character (if present) from the stream pointed at by `stream`, or `EOF` if `stream` is at end-of-file.

- `char * fgets(char *  str, int size, FILE *  stream);`
  The `fgets` function reads at most one less than the number of characters specified by `size` from the given `stream` and stores them in the string `str`. Reading stops when a newline character is found, at end-of-file or error. The newline, if any, is retained. If any characters are read and there is no error, a `\0` character is appended to end the string. Upon successful completion, it returns a pointer to the string. If end-of-file occurs before any characters are read, it returns `NULL`.

- `ssize_t getline(char ** linep, size_t * linecapp, FILE * stream);`
  The `getline` function reads a line from `stream`, which is ended by a newline character or end-of-file. If a newline character is read, it is included in the string. The caller may provide a pointer to a `malloc`ed buffer for the line in `*linep`, and the capacity of that buffer in `*linecapp`. These functions expand the buffer as needed, as if via `realloc`. If `linep` points to a `NULL` pointer, a new buffer will be allocated. In either case, `*linep` and `*linecapp` will be updated accordingly. This function returns the number of characters written to the string, excluding the terminating `\0` character. The value -1 is returned if an error occurs, or if end-of-file is reached.

Question 1 Concepts [14 pts]

For all parts of this question, assume that integers and floats occupy 4 bytes and doubles and pointers occupy 8 bytes.

1. There is a suggestion on StackOverflow that one find the size of an array (such as `int a[17];`) in C by doing the following:

```
int n = sizeof(a)/sizeof(a[0]);
```

   Which one of the following describes the problem with this approach?

   a) `sizeof(a)` gives the size of a pointer.
   b) `sizeof(a[0])` gives the size of an int.
   c) This should be `int n = sizeof(*a);`
   d) This only works within a stack frame. ←
   e) None of the above

2. If `p` is declared as `int * p;`, and the numerical value of `p` is 0x87048e, then what is the numerical value of p+3?

   a) 0x870491
   b) 0x87049a ←
   c) 0x870506
   d) 0x8705a6
   e) None of the above

3. Consider the following two declarations for `arr1` and `arr2`:

```
int arr1[2][3];
int r1[3];
int r2[3];
int * arr2[2] = {r1, r2};
```

   Which one of the following correctly gives their sizes?

   a) `sizeof(arr1)` = 8 bytes and `sizeof(arr2)` = 8 bytes
   b) `sizeof(arr1)` = 24 bytes and `sizeof(arr2)` = 16 bytes ←
   c) `sizeof(arr1)` = 24 bytes and `sizeof(arr2)` = 24 bytes
   d) `sizeof(arr1)` = 48 bytes and `sizeof(arr2)` = 32 bytes
   e) None of the above

4. Debugging is best performed by

   a) Guess and check
   b) Making sure you've `free`d all memory
   c) Applying the scientific method ←
   d) Converting all array indexing to pointer arithmetic
   e) None of the above

5. If you write the following line of code:

   ```
   char * mystr = "Hello!";
   ```

   What will happen if you do `mystr[0] = 'J';`?

   a) The program will seg fault. ←
   b) The compiler will give an error.
   c) `mystr` will have the value `Jello!`.
   d) `mystr` will be unchanged.
   e) None of the above

6. What is *tail recursion*?

   a) The recursive call is the last line of a function.
   b) The function returns immediately after the recursive call. ←
   c) A recursive implementation of an algorithm that cannot be written iteratively.
   d) Converting an iterative algorithm to a recursive one.
   e) None of the above

7. *All* of the following are errors in using `free` *except*:

   a) Freeing the middle of a block of memory
   b) Freeing memory not allocated by malloc
   c) Freeing a `NULL` pointer ←
   d) Double freeing a block of memory
   e) None of the above (all are errors)

2 points (all or nothing) each

Question 2 Reading Code [7 pts]

What is the output of the following code?

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

int * aFunction(int ** p, int ** q) {
  printf("**p=%d\n",**p);
  (*p)++;
  *q = *p+1;
  printf("**p=%d\n",**p);
  printf("**q=%d\n",**q);
  **q = **p - (*p)[-1];
  int ** temp = p;
  p = q;
  q = temp;
  return *q - 1;
}

int main(void) {
  int arr[3];
  for (size_t i = 0; i < 3; i++) {
    arr[2-i] = 2*i;
    printf("arr[%zu]=%zu\n",2-i,2*i);
  }
  int * curr = &arr[0];
  int * next = NULL;
  curr = aFunction(&curr, &next);
  printf("*curr=%d\n",*curr);
  printf("*next=%d\n",*next);
  for (size_t i = 0; i < 3; i++) {
    printf("arr[%zu]=%d\n",i,arr[i]);
  }
  return EXIT_SUCCESS;
}
```

Output:

```
arr[2]=0
arr[1]=2
arr[0]=4
**p=4
**p=2
**q=0
*curr=4
*next=-2
arr[0]=4
arr[1]=2
arr[2]=-2
```

Question 3 Fixing Valgrind Errors [12 pts]

A C programmer wrote the following buggy code to do some calculations with points:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct _point_t {
5    double x;
6    double y;
7  };
8  typedef struct _point_t point_t;
9
10 point_t * init_points(size_t sz) {
11   point_t * answer = malloc(sz+sizeof(*answer));
12   for (size_t i = 0; i < sz; i++) {
13     answer[i].x = i;
14     answer[i].y = i*i;
15   }
16   return answer;
17 }
18
19 float ** calculate(point_t * arr1, point_t * arr2, size_t sz1, size_t sz2) {
20   float ** answer = malloc(sz1*sizeof(*answer));
21   for (size_t i = 0; i < sz1; i++) {
22     answer[i] = malloc(sz2*sizeof(*answer[i]));
23     for (size_t j = 0; j <= sz2; j++) {
24       answer[i][j] = arr1[i].x + arr1[i].y + arr2[j].x + arr2[j].y;
25     }
26   }
27   return answer;
28 }
29
30 int main(void) {
31   size_t n1;
32   size_t n2;
33   point_t * arr1 = init_points(n1);
34   point_t * arr2 = init_points(n2);
35   float ** results = calculate(arr1, arr2, n1, n2);
36   for (size_t i = 0; i < n1; i++) {
37     for (size_t j = 0; j < n2; j++) {
38       printf("%2.2f ",results[i][j]);
39     }
```

```
40      printf("\n");
41    }
42    free(arr1);
43    free(arr2);
44    free(results);
45    return EXIT_SUCCESS;
46  }
```

1. The first problem is:

```
Conditional jump or move depends on uninitialised value(s)
    at 0x4C2DB3C: malloc
    by 0x400621: init_points (errors.c:11)
    by 0x40081B: main (errors.c:33)
  Uninitialised value was created by a stack allocation
    at 0x400808: main (errors.c:30)
```

   This problem can be correctly fixed by:

   a) Initializing answer to NULL on line 11.
   b) Initializing `n1` to 2 and `n2` to 3 on lines 31 and 32. ←
   c) Initializing `arr1` and `arr2` to `NULL` on lines 33 and 34.
   d) Initializing `results` to `NULL` on line 35.
   e) None of the above. Describe what to do instead:

2. After correctly fixing the first problem, the next error is:

```
Invalid write of size 8
    at 0x400673: init_points (errors.c:13)
    by 0x40082B: main (errors.c:33)
  Address 0x5204050 is 16 bytes inside a block of size 18 alloc'd
    at 0x4C2DB8F: malloc
    by 0x400621: init_points (errors.c:11)
    by 0x40082B: main (errors.c:33)
```

   This problem can be correctly fixed by:

   a) Changing the '`+`' to a '`*`' on line 11. ←
   b) Changing `*answer` to `point_t` on line 11.
   c) Changing the '`<`' to a '`<=`' on line 12.
   d) Changing `answer[i].x` to `answer[i]->x` on line 13.
   e) None of the above. Describe what to do instead:

3. After correctly fixing the previous problem, the next error is:

```
Invalid read of size 8
    at 0x4007AC: calculate (errors.c:24)
    by 0x400857: main (errors.c:35)
  Address 0x52040d0 is 0 bytes after a block of size 48 alloc'd
    at 0x4C2DB8F: malloc
    by 0x400621: init_points (errors.c:11)
    by 0x40083B: main (errors.c:34)
```

This problem can be correctly fixed by:

a) Changing the '`<`' to an '`<=`' on line 21.

b) Changing the '`<=`' to an '`<`' on line 23. ←

c) Changing `answer[i][j]` to `answer[j][i]` on line 24.

d) Changing each `arr[i].` to `arr[i]->` on line 24.

e) None of the above. Describe what to do instead:

4. After correctly fixing the previous problem, the program leaks memory:

```
HEAP SUMMARY:
      in use at exit: 24 bytes in 2 blocks
    total heap usage: 6 allocs, 4 frees, 1,144 bytes allocated

LEAK SUMMARY:
     definitely lost: 24 bytes in 2 blocks
     indirectly lost: 0 bytes in 0 blocks
       possibly lost: 0 bytes in 0 blocks
     still reachable: 0 bytes in 0 blocks
          suppressed: 0 bytes in 0 blocks
```

This problem can be correctly fixed by:

a) Adding the line `free(results[i][j]);` after line 37 and before line 39.

b) Adding the line `free(results[i]);` after line 39 and before line 41. ←

c) Moving the line `free(results);` from line 44 to after line 39 and before line 41.

d) Adding the lines `free(arr1[i]);` and `free(arr2[i]);` to after line 39 and before line 41.

e) None of the above. Describe what to do instead:

3 points (all or nothing) each

Question 4 Coding: Recursion [10 pts]

Write a recursive implementation of the function
`int isPalindrome(const char * str, size_t n)`, which takes a string and its size as parameters and returns a 0 if it is not a palindrome and a 1 if it is. A palindrome is a word that has letters that are symmetrical about the middle of the word, so that it reads the same forwards and backwards. For example, "radar" is a palindrome, but "banana" is not. Your algorithm should be case-sensitive (`'A'` and `'a'` are different), and it does not matter if the character is alphabetic or not.

```
int isPalindrome(const char * str, size_t n) {









}
```

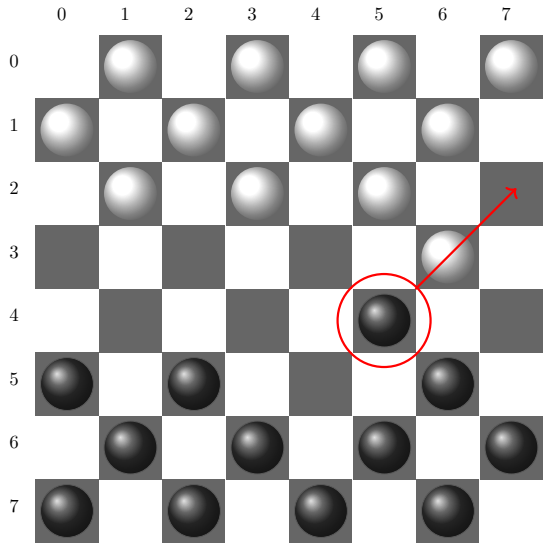
One possible solution:

```
int isPalindrome(const char * str, size_t n) {
  if (str == NULL) {
    return 0;
  }
  if (n < 2) {
    return 1;
  }
  if (str[0] == str[n-1]) {
    return isPalindrome(str+1, n-2);
  }
  return 0;
}
```

Question 5 Coding: Arrays [14 pts]

Consider the game of checkers, where players can "jump" an opponent's checker to remove it from play. Jumps are diagonal, and each player moves in a forward (from their perspective) direction. For a jump to be possible, an opponent's checker must be diagonally adjacent, and there must be a blank space diagonally beyond it.

Our checkerboard is represented by an `int board[8][8]`, where each space can have a black checker (1), white checker (0), or no checker (-1). Write the function `jump_t canBlackJump(pos_t p, int b[8][8])`, which takes a `pos_t` to indicate the player position and a pointer to the board (initialize elsewhere), and returns a `jump_t` indicating the jump options. A `pos_t` has two fields: the x-position and y-position (square (0,0) is the top left-most square).



For example, The black checker shown has position (5,4). `canBlackJump` would return `RIGHT`, since that is the only available jump.

```
struct pos_tag {
  int x;
  int y;
};
typedef struct pos_tag pos_t;

enum jump_tag {
  LEFT,
  RIGHT,
  EITHER,
  NO_JUMP
};
typedef enum jump_tag jump_t;
```

```
jump_t canBlackJump(pos_t p, int b[8][8]) {









}
```

One possible solution:

```
jump_t canBlackJump(pos_t p, int b[8][8]) {
  jump_t answer = NO_JUMP;
  if (p.x-2 < 0 || p.y-2 < 0) {
    answer = NO_JUMP;
  }
  else if (b[p.y-1][p.x-1] == 0 && b[p.y-2][p.x-2] == -1) {
    answer = LEFT;
  }
  if (p.x+2 >= 8 || p.y-2 < 0) {
    return answer;
  }
  else if (b[p.y-1][p.x+1] == 0 && b[p.y-2][p.x+2] == -1) {
    if (answer == LEFT) {
      return EITHER;
    }
    return RIGHT;
  }
  return answer;
}
```

Question 6 Coding: String and More [18 pts]

Write the function `void printFormattedRoster(const char * str)`, which takes a string of the Duke men's Basketball 2017–18 roster, extracts the player information, and prints a formatted player list in sorted order by jersey number. (A roster is a list of players and their information, such as jersey number.) For example, if your function is passed

```
3Allen35Bagley1Duval34Carter41White15Connell
```

you should print to the screen as follows,

```
1 Duval
3 Allen
15 Connell
34 Carter
35 Bagley
41 White
```

You may make the following assumptions:

- There are no spaces in the input string;
- The input string always has the correct NumberNameNumberName format;
- The uniform number of each player will not repeat. For example, you do not need to consider "1Giles1Duval".
- The total number of players are determined by the input and cannot be presumed.
- You may assume that the uniform number is 0 to 99 inclusive, and their name is at least one letter long.
- You have a function `void sortData(char ** data, size_t count)`, which will sort an array of strings by the number at the beginning of each string.

You may want to use the following library functions: `malloc`, `realloc`, `free`, and `strndup`. You *should* abstract out some tasks into helper functions.

Begin your answer on the next page.

```
void printFormattedRoster (const char * str) {









}
```

One possible solution:

```
const char * nextPlayer(const char * str) {
  const char * ans = str;
  while (*ans != 0) {
    if (!isalpha(*ans)) {
      return ans;
    }
    ans++;
  }
  return ans;
}

void printFormatted(const char * str) {
  int first = 1;
  for (size_t i = 0; i < strlen(str); i++) {
    if (isalpha(str[i]) && first) {
      printf(" ");
      first = 0;
    }
    printf("%c", str[i]);
  }
  printf("\n");
}

void printFormattedRoster(const char * str) {
  const char * curr = str;
  const char * next = NULL;
  char ** lines = NULL;
  size_t num = 0;
  while (*curr != 0) {
    next = nextPlayer(curr+2);
    lines = realloc(lines, (num+1)*sizeof(*lines));
    lines[num] = strndup(curr, next-curr);
    curr = next;
    num++;
  }
  sortData(lines, num);
  for (size_t i = 0; i < num; i++) {
    printFormatted(lines[i]);
    free(lines[i]);
  }
  free(lines);
}
```