ECE 551D
Fall 2019
Midterm Exam

Name: NetID:

There are 7 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**
1	Concepts	10
2	Testing	10
3	Debugging	8
4	Reading Code	9
5	Coding: Arrays	12
6	Coding: File IO	11
7	Coding: Reverse Region	15
	Total	75

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.

- `long int strtol(const char *nptr, char **endptr, int base);`
  `long long int strtoll(const char *nptr, char **endptr, int base);`
  `unsigned long int strtoul(const char *nptr, char **endptr, int base);`
  `unsigned long long int strtoull(const char *nptr, char **endptr, int base);`
  These functions convert the initial part of the string in `nptr` to a long (or long long, or unsigned long, or unsigned long long) integer value according to the given `base`, which must be between 2 and 36 inclusive, or be the special value 0. If `endptr` is not `NULL`, the address of the first invalid character is stored in `*endptr`.

- `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.

- `void * memcpy(void * dst, const void *src, size_t n);`
  The `memcpy` function copies `n` bytes from memory area `src` to memory area `dst`.

Question 1 Concepts [10 pts]

For all parts of this question, you must blacken the circle of the answer you choose.

1. Assume integers and floats occupy 4 bytes and doubles and pointers occupy 8 bytes. `p` is declared `float ** p`. How many bytes is `*p` incremented in the expression `(*p) += 3` ?

   (a) 3

   (b) 4

   (c) 12

   (d) 24

   (e) None of the above

2. Still assuming that integers and floats occupy 4 bytes and doubles and pointers occupy 8 bytes, what does the following code print?

```
void f(int data[]) {
  printf("%lu", sizeof(data));
}

int main(void) {
  int data[] = {0, 1, 2, 3};
  f(data);
  return EXIT_SUCCESS;
}
```

   (a) 4

   (b) 8

   (c) 16

   (d) 32

   (e) None of the above

3. Why should you use the `const` modifier when declaring a string literal `const char * str`?

   (a) This causes the compiler to store the literal in read only memory.

   (b) This keeps the program from crashing when you modify a character in the string.

   (c) This causes a compiler error when a character in the string is modified.

   (d) This causes a compiler error when `str` is modified.

   (e) None of the above

4. Why might you want to define an enumerated type?

   (a) They enable the programmer to group multiple variables into a single entity.

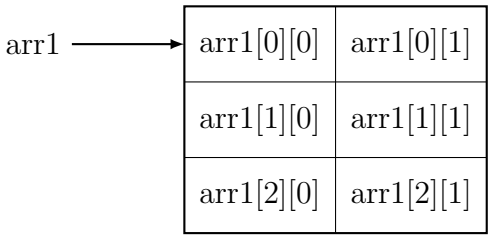
   (b) Defining a type at one place in the code base makes it easier for a programmer to make changes later.

   (c) They can represent a larger range of numbers than the integer type.

   (d) Giving each value in a set a conceptual name enhances readability.

   (e) None of the above

5. For the following conceptual picture, which one of the following could be the declaration for `arr1`?



   (a) `int * arr1[] = {(int *){0, 1}, (int *){2, 3}, (int *){4, 5}};`

   (b) `int * arr1[] = {(int *){0, 1, 2}, (int *){3, 4, 5}};`

   (c) `int arr1[][2] = {{0, 1}, {2, 3}, {4, 5}};`

   (d) `int arr1[][3] = {{0, 1, 2}, {3, 4, 5}};`

   (e) None of the above

Question 2 Testing [10 pts]

Suppose you needed to write test cases for the following function:

```
int getSquareRoot(int n);
```

This function should return the non-negative square root if `n` is a perfect square and -1 otherwise.

A perfect square is a whole number whose square root is also a whole number. For example, 4 is a perfect square because $2 \times 2 = 4$, so on input 4, the function would return 2. 6 is not a perfect square because there is no real integer that equals 6 when squared, so the function would return -1. For this function, we define 0 as a perfect square.

You should write five test cases for this function **on the next page**. For each test case, you should fill in all fields on the testing worksheet. For full credit, the test cases should test distinct types of errors.

Test Case 1
Description of Mistake:
Input:
Expected Behavior:
Behavior if Mistake Were Made:

Test Case 2
Description of Mistake:
Input:
Expected Behavior:
Behavior if Mistake Were Made:

Test Case 3
Description of Mistake:
Input:
Expected Behavior:
Behavior if Mistake Were Made:

Test Case 4
Description of Mistake:
Input:
Expected Behavior:
Behavior if Mistake Were Made:

Test Case 5
Description of Mistake:
Input:
Expected Behavior:
Behavior if Mistake Were Made:

Question 3 Debugging [8 pts]

Consider the following code (which has valgrind errors):

```
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[sz];
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 size) {
20   float ** answer = malloc(size * sizeof(float));
21   for (size_t i = 0; i < size; i++) {
22     answer[i] = malloc(size * sizeof(**answer));
23     for (size_t j = 0; j < size; j++) {
24       float sum = arr1[i].x + arr1[i].y + arr2[j].x + arr2[j].y;
25       answer[i][j] = sum;
26     }
27   }
28   return answer;
29 }
30
31 int main(int argc, char ** argv) {
32   size_t size = strtoull(argv[2], NULL, 0);
33   point_t * arr1 = init_points(size);
34   point_t * arr2 = init_points(size);
35   float ** results = calculate(arr1, arr2, size);
36   for (size_t i = 0; i < size; i++) {
37     for (size_t j = 0; j < size; j++) {
38       printf("%2.2f ", results[i][j]);
39     }
40     printf("\n");
41   }
42   free(arr1);
43   free(arr2);
44   for (size_t i = 0; i < size; i++) {
45     free(results[i]);
46   }
47   return EXIT_SUCCESS;
48 }
```

The program is compiled (ignoring warnings—eek!), and run as follows:

```
valgrind ./myProgram 42
```

For each of the following questions, assume each problem is fixed in order, and the program is recompiled and run again after each question.

1. The first error is

   ```
   ==36241== Invalid read of size 1
   ==36241==    at 0x4E81970: ____strtoul_l_internal (strtol_l.c:292)
   ==36241==    by 0x108A60: main (valgrinderr.c:32)
   ==36241==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
   ```

   Which one of the following describes how to fix this error?

   (a) Change line 31 to be `int main(void) {`

   (b) Change line 31 to be `void main(int argc, char ** argv) {`

   (c) Change line 32 to be `size_t size = strtoull(argv[1], NULL, 0);`

   (d) Change line 32 to be `size_t size = strtoull(argv[2], argv + argc, 0);`

2. The next error is

   ```
   ==59470== Invalid read of size 8
   ==59470==    at 0x108987: calculate (valgrinderr.c:24)
   ==59470==    by 0x108A9B: main (valgrinderr.c:35)
   ==59470==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
   ```

   Which one of the following describes how to fix this error?

   (a) Change line 11 to be `point_t * answer = malloc(sz * sizeof(*answer));`

   (b) Change line 20 to be `float ** answer = malloc(size * sizeof(*answer));`

   (c) Change line 21 to be `for (size_t i = 0; i < size - 1; i++) {`
   and line 23 to be `for (size_t j = 0; j < size - 1; j++) {`

   (d) Change line 22 to be `answer[i] = malloc(size * sizeof(*answer[i]));`

3. The next error is

```
==62437== Invalid write of size 8
==62437==    at 0x108887: calculate (valgrinderr.c:22)
==62437==    by 0x1089BD: main (valgrinderr.c:35)
==62437==  Address 0x522d6a8 is 0 bytes after a block of size 168 alloc'd
==62437==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_[...]
==62437==    by 0x108851: calculate (valgrinderr.c:20)
==62437==    by 0x1089BD: main (valgrinderr.c:35)
```

Which one of the following describes how to fix this error?

(a) Change line 20 to be `float ** answer = malloc(size * sizeof(*answer));`

(b) Change line 21 to be `for (size_t i = 0; i < size - 1; i++) {`
and line 23 to be `for (size_t j = 0; j < size - 1; j++) {`

(c) Change line 22 to be `answer[i] = malloc(size * sizeof(*answer[i]));`

(d) Change line 25 to be `answer[i - 1][j - 1] = sum;`

4. There are no more errors, but the program leaks memory. When run

```
valgrind --leak-check=full ./myProgram 42
```

valgrind reports:

```
==70170== HEAP SUMMARY:
==70170==     in use at exit: 336 bytes in 1 blocks
==70170==   total heap usage: 46 allocs, 45 frees, 9,760 bytes allocated
==70170==
==70170== 336 bytes in 1 blocks are definitely lost in loss record 1 of 1
==70170==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload[...]
==70170==    by 0x108851: calculate (valgrinderr.c:20)
==70170==    by 0x1089BD: main (valgrinderr.c:35)
```

Which one of the following describes how to fix this leak?

(a) After line 15, add the line `free(answer);`

(b) After line 32, add the line `free(size);`

(c) After line 40, add the line `free(results[i]);`

(d) After line 46, add the line `free(results);`

Question 4 Reading Code [9 pts]

What is the output of the following code?

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

int * myFn(int * p, int ** q) {
  p++;
  printf("*p = %d\n", *p);
  *p += 2;
  **q -= 4;
  (*q)++;
  printf("**q = %d\n", **q);
  return p;
}

int main(void) {
  int data[] = {6, 93, 87, 4};
  int * p = &data[1];
  int * x = &data[2];
  int ** ptrs[] = {&p, &x};
  for (int i = 0; i < 2; i++) {
    printf("%d\n", **ptrs[i]);
  }
  int * r = myFn(p, ptrs[1]);
  printf("*r = %d\n", *r);
  **ptrs[0] += 100;
  **ptrs[1] += 1000;
  for (int i = 0; i < 4; i++) {
    printf("%d\n", data[i]);
  }
  return EXIT_SUCCESS;
}
```

Output:

Question 5 Coding: Arrays [12 pts]

Given the following struct declaration:

```
struct array_tag {
  int * arr;
  size_t n;
};
typedef struct array_tag array_t;
```

Write the function

```
array_t * findRoots(int * arr, size_t n);
```

This function should return a *pointer* to an `array_t` (allocated in the heap), which has the fields `arr`, a pointer to an array of `int`s (also in the heap) representing the square roots of any perfect squares in the input array, and the number of elements in that array `n`.

For example, if the function were called on the array `{2, 4, 12, 16, 4}`, and `n = 5`, it should return an `array_t` that has an array with three elements: `arr` is `{2, 4, 2}` because 2 is the square root of 4, 4 is the square root of 16, and 2 is the square root of 4. The other elements in the input array, `{2, 12}` are not perfect squares, so they are ignored. If there are no roots of perfect squares, it should return an `array_t` where `arr` is `NULL` and `n` is 0.

You should abstract out the helper function `int getSquareRoot(int n);` which you have already written test cases for in a previous question. Recall: this function should return the non-negative square root if `n` is a perfect square and -1 otherwise. A perfect square is a whole number whose square root is also a whole number. For this function, we define 0 as a perfect square. You may not use math library functions.

Write your answer on the next page.

```
int getSquareRoot(int n) {









}

array_t * findRoots(int * arr, size_t n) {









}
```

Question 6 Coding: File IO [11 pts]

Write a program that reads from `stdin` and uses `findRoots`, which you wrote in the previous question, to print the roots of perfect squares. You are provided the function

```
array_t * splitLine(const char * str);
```

which reads a string ending in a newline character and separates it into an array of integers, dynamically allocating the return value on the heap. Your task is to use this function, along with `findRoots`, to print the square roots to `stdout`. The output should be comma-separated, with newlines where they appear in the input.

For example, if the input were

```
2,4,12,16,4
9,0,4,-1
```

it would print out

```
2,4,2
3,0,2
```

If the input string does not have the proper format, `splitLine` returns `NULL`, and you should print a message to `stderr` and continue reading the next line. This program should not leak memory.

You may not assume the length of the input. You may assume `splitLines` and the code you wrote in previous questions is defined elsewhere, and you do not need to declare the struct or functions again.

Write your answer on the next page.

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

// your code goes here
```

Question 7 Coding: Reverse Region [15 pts]

For this problem, you will write the function

```
void revRegion(char * str);
```

which reverses each region inside a string that is enclosed by a set of matching parentheses, defined as a pair of open and close parentheses within which the number of open parentheses is the same as the number of close parentheses.

For example, given `"ab(cd-e4).fg(ymk)zz"` your function should modify the string (in place, as in your string reverse assignment) to produce `"ab(4e-dc).fg(kmy)zz"`. Note that the answer was produced by leaving the parts of the string *outside* the parentheses unmodified and reversing the parts *inside* the parentheses. If there are no parentheses, the string should be unchanged. If there is an open parenthesis with no matching close parenthesis, you should leave that portion of the string unmodified. For example, `"ab(cd)ef(g"` turns into `"ab(dc)ef(g"`, since the last parenthesis is unmatched.

For this question, if you write code that works correctly when no nested parentheses are present, you will earn 12 points. If you want full credit, you should handle nested parentheses. The specification for nested parentheses is that the region within the outermost parentheses should be reversed, but then you are done with that portion of the string—you should NOT search for the inner parentheses and should NOT re-reverse those regions. For example, given `"ab(c(de(fg)h)ik)xyz"` you should produce `"ab(ki)h)gf(ed(c)xyz"`. Note that the imbalanced parenthesis in the resulting string is fine. As before, if there is an unmatched parenthesis at the end, you should leave that final portion of the string unmodified.

Hint: abstract out the function `char * findMatchingClose(char * str)`. For this helper function, it is useful to keep a counter of how many open parentheses you have seen, decrementing it when you see each close parenthesis.

Here are some more examples.

Without handling nested parentheses:

- `""` → `""`
- `"abc"` → `"abc"`
- `"a)(bc)"` → `"a)(cb)"` (ignore close paren before open)
- `"a()b(cd)"` → `"a()b(dc)"` (reverse empty region is fine, unchanged)
- `"(ab)))"` → `"(ba)))"`

With handling nested parentheses:

- `"((abc))"` → `"()cba()"`
- `"a(()b"` → `"a(()b"`. (no matching close, unchanged)

Write your answer on the next page.

```
// write any helper functions you want here (hint: good idea).

void revRegion(char * str) {

}
```