ECE551
Midterm Version 2

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, you may use your notebook, which must contain only handwritten notes, written 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	Multiple Choice Concepts		12
2	Reading Code		9
3	Debugging		8
4	Coding 1: Strings		7
5	Coding 2: Arrays		8
6	Coding 3: Dynamic Allocation		13
7	Coding 4: File IO		18
	Total		75
	Percent		100

The next two pages contains 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 *s1);`
  The `strdup` function allocates sufficient memory for a copy of the string `s1`, 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.

- `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 getdelim() function , delimited by the character delimiter. 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`.

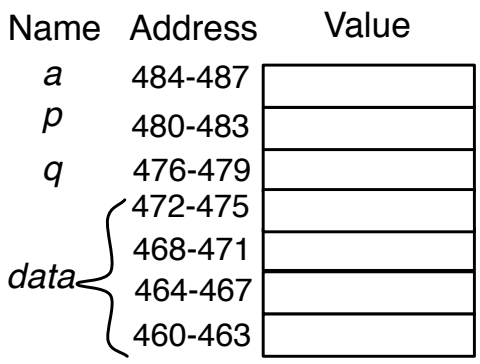
Question 1 Multiple Choice Concepts [12 pts]

Q1.1 Consider the code on the left of the following figure, and the frame layout on the right (we assume that `sizeof(int)=4`, and `sizeof(int*)=4` on this system):

Code

```
int a = 9;
int * p = &a;
int ** q= &p;
int data[2][2];
data[0][0] = 35;
data[0][1] = 87;
data[1][0] = 12;
data[1][1] = 200;
```

Frame Layout



1. In the diagram above, fill in each box in the frame with the **numerical** (not conceptually: write numbers, do not draw arrows for pointers) value that it contains when the code shown here finishes executing.

2. What is the **type** of `data[1]`?

3. What is the **numerical value** of `data[1]`?

4. What is the **type** of `data[1][0]`?

5. What is the **type** of `&data[1][1]`?

6. What is the **numerical value** of `&data[1][1]`?

Q1.2 Consider the following two declarations:

```
const char * s1 = "Hello";
char s2[] = "Hello";
```

For each of the following statements, select whether or not it is true of s1, s2, neither or both (place a check mark in the correct box)

Statement	**True of s1**	**True of s2**	**Both**	**Neither**
s1 and/or s2 is an lvalue				
&s1[3] and/or &s2[3] is valid				
s1[0] and/or s2[0] is in read only memory				
s1 and/or s2 occupies 6 bytes in the frame				
s1 and/or s2 point at a null terminated string				
s1 and/or s2 should be freed				

Q1.3 For each question, fill in the blank with the word or words which most correctly complete the sentence.

1. A program makes a _____ if it needs to interact with the outside world.
2. The difference between perror and just using fprint to stderr is that perror also prints a description of _____.
3. If you need to resize a block allocated by malloc, you would use _____.
4. _____ is the separation of interface from implementation.
5. When declaring a variable which points at a string literal, its most correct type is _____.
6. If you need to find out how many bytes a type occupies in memory, you would use the _____ operator.
7. If you are debugging in `gdb`, and are currently at a line with a function call, you would use the _____ command to go past the function call, without going inside it.

Question 2 Reading Code [9 pts]

Execute the following code by hand, and fill in the output at the bottom of this page (the start of each line is written for you, write the correct number at the end of each line).

```
#include <stdio.h>
#include <stdlib.h>
void f(int x, int * p, int ** q) {
  x = x + 7;
  *p = **q - x;
  *q = p;
}
int main(void) {
  int a = 6;
  int b = 3;
  int c = 1;
  int * data[] = {&a, &b, &c};
  int ** q = &data[1];
  **q = 43;
  q[0] = q[1];
  printf("a=%d, b=%d, c=%d\n",a,b,c);
  for (int i = 0; i < 3; i++) {
    *data[i] = *data[i] + 11;
  }
  printf("a=%d, b=%d, c=%d\n",a,b,c);
  f(a, &b, &data[2]);
  *q[1] = *q[1]+2;
  printf("a=%d, b=%d, c=%d\n",a,b,c);
  return EXIT_SUCCESS;
}
```

Fill in output below:

a= , b= , c=

a= , b= , c=

a= , b= , c=

Question 3 Debugging [8 pts]

A C programmer wrote the following buggy code to read in many lines of numbers, sort them, and print them out:

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3: int cmplong(const void * vp1, const void * vp2) {
 4:   const long * p1 = vp1;
 5:   const long * p2 = vp2;
 6:   return *p1 - *p2;
 7: }
 8: int main(void) {
 9:   char * line= NULL;
10:   size_t sz =0;
11:   long * array = NULL;
12:   size_t n = 0;
13:   while(getline(&line, &sz, stdin) > 0){
14:     n++;
15:     array=realloc(array, n * sizeof(*array));
16:     array[n] = strtol(line, NULL, 0);
17:   }
18:   free(line);
19:   qsort(&array, n, sizeof(*array), cmplong);
20:   for (size_t i = 0; i < n; i++) {
21:     printf("%ld\n", array[i]);
22:     free(&array[i]);
23:   }
24:   return EXIT_SUCCESS;
25: }
```

Error 1 The first problem is:

```
Invalid write of size 8
   at 0x400788: main (broken.c:16)
 Address 0x51fc108 is 0 bytes after a block of size 8 alloc'd
   at 0x4C2AB80: malloc (in ...)
   by 0x4C2CF1F: realloc (in ...)
   by 0x400759: main (broken.c:15)
```

This problem can be correctly fixed by:

A. Using `malloc` instead of `realloc` on line 15

B. Changing line 16 to have `array[n-1]` instead of `array[n]`

C. Changing line 13 to read `size_t n = 1;`

D. Moving line 14 after line 15.

E. None of the above—specify what to do instead:

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

```
Conditional jump or move depends on uninitialised value(s)
   by 0x4E726CB: qsort_r (...)
   by 0x4007CF: main (broken.c:19)
```

This problem can be correctly fixed by:

A. Changing `qsort(&array,` to `qsort(array,` on line 19.

B. Changing `qsort(&array,` to `qsort(*array,` on line 19.

C. Changing `sizeof(*array)` to `sizeof(array)` on line 19.

D. Changing `sizeof(*array)` to `sizeof(&array)` on line 19.

E. None of the above—specify what to do instead:

Error 3 After correctly fixing the first two problems, the next error is:

```
Invalid read of size 8
   at 0x4007ED: main (broken.c:21)
Address 0x51fd408 is 8 bytes inside a block of size 224 free'd
   at 0x4C2BDEC: free (in ...)
   by 0x40081C: main (broken.c:22)

Invalid free() / delete / delete[] / realloc()
   at 0x4C2BDEC: free (in ...)
   by 0x40081C: main (broken.c:22)
Address 0x51fd408 is 8 bytes inside a block of size 224 free'd
   at 0x4C2BDEC: free (in ...)
   by 0x40081C: main (broken.c:22)
```

This problem can be correctly fixed by:

A. Changing line 22 from `free(&array[i]);` to `free(array[i]);`

B. Changing line 22 from `free(&array[i]);` to `free(array);`

C. Changing line 22 from `free(&array[i]);` to `free(*array[i]);`

D. Changing line 22 from `free(&array[i]);` to `free(array+i);`

E. None of the above—specify what to do instead:

Error 4 After correctly fixing the first three errors, the code runs, but leaks memory:

```
224 bytes in 1 blocks are definitely lost in loss record 1 of 1
   at 0x4C2CE8E: realloc (...)
   by 0x400759: main (broken.c:15)
```

This error can be corrected by:

A. Placing `free(array[i]);` between lines 22 and 23 (inside the for loop)

B. Placing `free(array)` between lines 23 and 24 (after the for loop)

C. Placing `free(line);` between lines 16 and 17 (inside the while loop)

D. Changing `sizeof(*array) to \verb`sizeof(array)+ on line 19

E. None of the above—specify what to do instead:

Question 4 Coding 1: Strings [7 pts]

For this problem you will write your own, slightly simplified implementation of the `atoi` function from the C library, **without using any library functions**. Specifically, your function should take a `const char *` as input, and return the integer which that string represents. If passed, `"123"`, your function should return 123. You may make the following simplifying assumptions:

- The string will only contain digits from `'0'` to `'9'` (and will, of course, be terminated by a `'\0'`).
- The number will be positive (which follows from the first point).
- The string will not be empty (it will include at least one digit).
- The resulting number will not overflow an `int`.

```
int atoi(const char * str){




}
```

Question 5 Coding 2: Arrays [8 pts]

Write the method `findSmallestSum` which takes two arrays of integers, `a1` and `a2`, and `n` which is the size of both arrays. This method returns the smallest sum of elements in the same positions in `a1` and `a2`. For example, if

```
a1 = {1, 4, 6, 7, 9}
a2 = {7, 5, 1, 3, 8}
```

Then this method would return 7, since the sums of elements in the same positions are 8, 9, 7, 10, and 17, respectively, and 7 is the smallest of those sums. If `n` is zero, your method should return 0.

```
int findSmallestSum(int * a1, int * a2, size_t n) {




}
```

Question 6 Coding 3: Dynamic Allocation [13 pts]

Write the function `char * int2Str(unsigned int x)` which takes an `unsigned int`, allocates memory for a string, and fills it in with the digits of the decimal representation of `x`. For example, if your function is passed `"123"`, then it should return 123. Note that `x` is unsigned, so you do not have to deal with negative numbers. You should not make assumptions about the maximum size of a number that is representable in an `unsigned int` (*e.g.*, if I run your code on a weird platform where `unsigned int` is 1024 bits, it should still work right).

You may use any of the following library functions (but only the following library functions): `malloc`, `realloc`, and `free`..

```
char * int2Str(unsigned int x) {




}
```

Question 7 Coding 4: File IO [18 pts]

Write a program which takes 2 command line arguments. The first argument is the name of an input file, and the second is an arbitrary string, which you will treat as a set of characters. Your program should read the file named by its first argument, and count how many times each character in its second argument appears in that file. After reading the entire file, it should print each character from its second argument, followed by a `:`, and the count of how many of that character it found in the input. For example, if the input file is

```
Hello World
I like to program!
```

and the second argument is `i!o`, your program should print:

```
i:1
!:1
o:4
```

(because `i` appears once, `!` appears once, and `o` appears 4 times).

For this problem, we will relax the "no assumptions" restriction by allowing you to assume that (1) two command line arguments are provided (2) the input file requested exists and is readable (fopen succeeds) (3) the characters in the second argument are distinct (none are repeated) (4) malloc and realloc always succeed and (5) fclose succeeds. You may use any library functions you wish. You may *NOT* assume anything about the length of the second argument, nor the contents of the input file.

(Answer on next page)

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