ECE264 Spring 2016
Exam 3, 630-730PM, April 14

In signing this statement, I hereby certify that the work on this exam is my own and that I have not copied the work of any other student while completing it. I understand that, if I fail to honor this agreement, I will receive a score of ZERO for this exam and will be subject to possible disciplinary action.

Signature:

You must sign here. Otherwise you will receive a **1-point** *penalty.*

Read the questions carefully.
Some questions have conditions and restrictions.

This is an *open-book, open-note* exam. You may use any book, notes, or program printouts. No personal electronic device is allowed. You may **not** borrow books from other students.

This exam tests four learning objectives:

- File (Question 1)
- Structure (Questions 2 and 3)
- Recursion (Question 3.1)
- Dynamic Structure (Question 3)

You must obtain 50% or more points in the corresponding question to pass the learning objective.

Contents

Learning Objective

File	Pass	Fail
Recursion	Pass	Fail
Structure	Pass	Fail
Dynamic Structure	Pass	Fail

The ASCII Table

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
00	00	NUL	32	20	SP	64	40	@	96	60	‘
01	01	SOH	33	21	!	65	41	A	97	61	a
02	02	STX	34	22	"	66	42	B	98	62	b
03	03	ETX	35	23	#	67	43	C	99	63	c
04	04	EOT	36	24	$	68	44	D	100	64	d
05	05	ENQ	37	25	%	69	45	E	101	65	e
06	06	ACK	38	26	&	70	46	F	102	66	f
07	07	BEL	39	27	’	71	47	G	103	67	g
08	08	BS	40	28	(	72	48	H	104	68	h
09	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	˜
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

1 File (3 points)

The following about `fseek` and `ftell` is for your reference.

```
SYNOPSIS
       #include <stdio.h>
       int fseek(FILE *stream, long offset, int whence);
       long ftell(FILE *stream);
DESCRIPTION
       The fseek() function sets the file position indicator for the
       stream pointed to by stream.  The new position, measured in
       bytes, is obtained by adding offset bytes to the position
       specified by whence.  If whence is set to SEEK_SET, SEEK_CUR,
       or SEEK_END, the offset is relative to the start of the file,
       the current position indicator, or end-of-file, respectively.
       A successful call to the fseek() function clears the
       end-of-file indicator for the stream and undoes any effects of
       the ungetc(3) function on the same stream.

       The ftell() function obtains the current value of the file
       position indicator for the stream pointed to by stream.
```

The following information about `bcopy` is for your reference.

```
SYNOPSIS
       #include <strings.h>

       void bcopy(const void *src, void *dest, size_t n);

DESCRIPTION

       The bcopy() function copies n bytes from src to dest.  The
       result is correct, even when both areas overlap.
```

The following about `fread` and `fwrite` is for your reference.

```
NAME
       fread, fwrite - binary stream input/output

SYNOPSIS
       #include <stdio.h>
       size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
       size_t fwrite(const void *ptr, size_t size, size_t nmemb,
                     FILE *stream);
```

```
DESCRIPTION
       The  function  fread()  reads  nmemb  elements of data, each size bytes
       long, from the stream pointed to by stream, storing them at  the  loca
       tion given by ptr.

       The  function  fwrite()  writes nmemb elements of data, each size bytes
       long, to the stream pointed to by stream, obtaining them from the loca
       tion given by ptr.

       For nonlocking counterparts, see unlocked_stdio(3).

RETURN VALUE
       On  success,  fread()  and  fwrite() return the number of items read or
       written.  This number equals the number of bytes transferred only  when
       size  is 1.  If an error occurs, or the end of the file is reached, the
       return value is a short item count (or zero).

       fread() does not distinguish between end-of-file and error, and callers
       must use feof(3) and ferror(3) to determine which occurred.
```

Please write down the output of the program (stored in the file called `output`) for the given input file (called `input`). Assume all file function calls are successful and the program returns EXIT_SUCCESS.

1.

2.

3.

4.

5.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * * argv)
{
  if (argc < 2) { return EXIT_FAILURE; }
  int size = 5;
  // assume malloc succeeds
  int * arr = malloc(sizeof(int) * size);
  // initialize every elements
```

```c
  int ind;
  for (ind = 0; ind < size; ind ++) { arr[ind] = ind; }
  FILE * foutptr = NULL;
  foutptr = fopen(argv[1], "w");
  // assume fopen succeeds
  fwrite(arr, sizeof(int), size, foutptr);

  // back to the beginning of the file
  fseek(foutptr, 0, SEEK_SET);
  fwrite(& arr[2], sizeof(int), size - 2, foutptr);
  long loc1 = ftell(foutptr);
  printf("1. %ld\n", loc1);
  fclose(foutptr);

  // open the same file for read now
  FILE * finptr = NULL;
  finptr  = fopen(argv[1], "r");
  fread(arr, sizeof(int), size, finptr);
  printf("2. %d\n", arr[0]);
  printf("3. %d\n", arr[4]);
  loc1 = ftell(finptr);
  printf("4. %ld\n", loc1);

  fseek(finptr, 0, SEEK_SET);
  int val;
  loc1 = ftell(finptr);
  fread(&val, sizeof(int), 1, finptr);
  long loc2 = ftell(finptr);
  printf("5. %ld\n", loc2 - loc1);
  fclose(finptr);
  free(arr);
  return EXIT_SUCCESS;
}
/* for your reference
   sizeof(char)   = 1
   sizeof(int)    = 4
   sizeof(int *)  = 8
   sizeof(double) = 8
*/
```

2 Structure (3 points)

For each question, select the correct answer. The following information about bcopy is for your reference.

```
SYNOPSIS
       #include <strings.h>

       void bcopy(const void *src, void *dest, size_t n);

DESCRIPTION

       The bcopy() function copies n bytes from src to dest.  The
       result is correct, even when both areas overlap.
```

Q1.
Q2.
Q3.
Q4.
Q5.
Q6.

```c
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
typedef struct
{
  int size;
  double * element;
} Array;

// Create an Array object, the size attribute is sz
// copy the elements from em to the Array object's elements
// must use deep copy, must not use shallow copy
Array * Array_create(int sz, const double * em)
{
  Array * arr;
  // Q1. allocate memory for arr

  A. arr = malloc(sizeof(int));
  B. arr = malloc(sizeof(Array));
  C. arr = malloc(sizeof(double) * sz);
  D. arr = malloc(sizeof(Array)  * sz);
  E. arr = malloc(sizeof(double));

  // do not worry about checking whether malloc fails
  // assign sz to arr's size
  arr -> size = sz;

  // Q2. allocate memory for arr's elements

  A. arr = malloc(sizeof(double) * sz);
  B. arr = malloc(sizeof(Array) * sz);
  C. arr -> element = malloc(sizeof(double) * sz);
  D. arr -> element = malloc(sizeof(double));
  E. arr -> element = malloc(sizeof(double) * em);

  // do not worry about checking whether malloc fails

  // Q3. copy the elements from em to arr's element
  A. array -> element = em;
  B. & (array -> element[0]) = em;
  C. array -> element = & em[0];
  D. bcopy(em, arr -> element, sizeof(double) * sz);
  E. bcopy(em, arr, sizeof(double));
  F. bcopy(em, arr, sizeof(Array));

  return arr;
}

void Array_destroy(Array * arr)
{
  // Q4. release the memory of arr's element

  A. free (arr -> size);
  B. free (arr);
  C. malloc (arr);
  D. free (Array);
  E. free (arr -> element);

  // Q5. release the memory of arr object
  A. free (arr);
  B. free (arr -> size);
  C. free (Array);
  D. free (arr -> Array);
  E. free (int);
}

// create a new Array object
//    the new Array object has the same size as arr's size
//    the new Array's i-th element has the same value as
//         arr's i-th element's value
//
// MUST use deep copy (i.e., do not share memory
// space)
// assume arr is valid and do not need to check
Array * Array_copy(Array * arr)
{
  // Q6.
  A. return Array_create(arr -> size, arr -> element);
  B. return Array_create(arr);

  C.
    Array * arr2 = malloc(sizeof(Array));
    arr2 -> size = arr -> size;
    arr2 -> element = arr -> element;
    return arr2;

  D.
    Array * arr2 = malloc(sizeof(Array));
    bcopy(arr -> element, arr2 -> element, arr -> size);
    return arr2;

  E.
    Array * arr2;
    arr2 = malloc(sizeof (Array));
    arr2 = arr;
    return arr2;
}

void Array_print(Array * arr)
{
  int ind;
  printf("size = %d\n", arr -> size);
  for (ind = 0; ind < arr -> size; ind ++)
    {
      printf("element[%d] = %f\n",
             ind, arr -> element[ind]);
    }
}

int main(int argc, char * * argv)
{
  double dbarr[] = {-1.1, 2.2, 3.3, 4.4, -5.5,
                    -6.6, 0.7, 8.8, 9.9, -7.2};
  Array * arr1 = Array_create(10, dbarr);
  Array * arr2 = Array_copy(arr1);
  arr2 -> element[0] = 26.4;
  Array_print(arr1);
  Array_print(arr2);
  Array_destroy(arr1);
  Array_destroy(arr2);
  return EXIT_SUCCESS;
}
```

3 Recursion and Dynamic Structure (6.5 points)

Consider the following structure for linked lists.

```c
// file: list.h
#include <stdio.h>
#include <stdlib.h>
#ifndef LIST_H
#define LIST_H
typedef struct listnode
{
  struct listnode * next;
  double value;
} Node;
#endif
```

```c
// construct.c
#include "list.h"
Node * Node_construct(int val)
{
  Node * n = malloc(sizeof(Node));
  n -> value = val;
  n -> next = NULL;
  return n;
}
```

```c
// insert1.c
// This function is correct. The newly inserted
// value is at the beginning of the list.
#include "list.h"
Node * Node_construct(int val);
Node * List_insert1(Node * head, int val)
{
  Node * p = Node_construct(val);
  p -> next = head;
  return p;   /* insert at the beginning */
}
```

```c
// print.c
#include "list.h"
void List_print(Node * head)
{
  printf("\nPrint the list:\n");
  while (head != NULL)
```

```c
    {
      printf("%6.2f ", head -> value);
      head = head -> next;
    }
  printf("\n\n");
}
```

3.1 Insertion (2.5 points)

What is the output of this program? Please notice that there is a mistake in the program.

```
// insert.c
#include "list.h"
void List_print(Node * head);
Node * Node_construct(int val);
Node * List_insert1(Node * head, int val);
Node * List_insert2(Node * head, int val)
{
  if (head == NULL)
    {
      Node * ptr = Node_construct(val);
      return ptr;
    }
  // --->>> ERROR <<<---
  // should be
  // head -> next = List_insert2(head -> next, val);
  head = List_insert2(head -> next, val);
  return head;
}
int main(int argc, char * argv[])
{
  Node * head = NULL;
  int iter;
  for (iter = 0; iter < 5; iter ++)
    {
      head = List_insert1(head, iter);
    }
  List_print(head);
  // Print the list:
  // 4.00   3.00   2.00   1.00   0.00
  for (iter = 6; iter < 10; iter ++)
    {
      head = List_insert2(head, iter);
      // --->>> what is the output? <<<---
      List_print(head);
    }
  // do not worry about memory leak in this program
  return EXIT_SUCCESS;
}
```

3.2 Deletion (2.5 points)

What is the output of this program? Please notice that there is a mistake in the program.

```
// delete.c
#include "list.h"
void List_print(Node * head);
Node * List_insert1(Node * head, int val);
Node * List_delete(Node * head, int v)
{
  Node * p = head;
  if (p == NULL) /* empty list, do nothing */
    {
      return p;
    }
  /* delete the first node (i.e. head node)? */
  if ((p -> value) == v)
    {
      p = p -> next;
      free (head);
      return p;
    }
  /* not deleting the first node */
  Node * q = p -> next;


  // --->>> ERROR <<<---
  // should be
  // while ((q != NULL) && ((q -> value) != v))
  while (q != NULL)
    {
      // --->>> what is the output <<<---
      List_print(q);

      p = p -> next;
      q = q -> next;
    }
  if (q != NULL)
    {
      /* find a node whose value is v */
      p -> next = q -> next;
      free (q);
    }
  return head;
```

```c
}
int main(int argc, char * argv[])
{
  Node * head = NULL;
  int iter;
  for (iter = 0; iter < 5; iter++)
    {
      head = List_insert1(head, iter);
    }
  List_print(head);
  /*
    Print the list:
    4.00   3.00   2.00   1.00   0.00
  */
  head = List_delete(head, 13);

  // --->>> what is the output <<<---
  List_print(head);

  // do not worry about memory leak in this program
  return EXIT_SUCCESS;
}
```

3.3 Memory Leak (1.5 points)

The following program has memory leak. How many bytes are leaked (0.5 point). Explain the method to obtain the answer (1 point).

```
#include "list.h"
void List_print(Node * head);
Node * List_insert1(Node * head , int val);
void List_destroy(Node * head)
{
  // do nothing
}
int main(int argc , char * argv[])
{
  Node * head = NULL;
  int iter;
  for (iter = 0; iter < 5; iter ++)
    {
      head = List_insert1(head , iter);
    }
  List_destroy(head);
  // for your reference
  /*
     sizeof(char)   = 1
     sizeof(struct listnode *) = 8
     sizeof(int *)  = 8
     sizeof(double) = 8
     sizeof(Node) = 16
  */
  return EXIT_SUCCESS;
}
```