ECE 264 Exam 2

6:30-7:30PM, March 9, 2011

I certify that I will not receive nor provide aid to any other student for this exam.

Signature:

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

Read the questions carefully.

Please write legibly. Your exam is not graded if your writing is hard to read.

This exam is printed **double sided**. Please read the questions carefully. Two common mistakes are answering a wrong question and failing to answer all questions.

This is an *open-book, open-note* exam. You can use any book or note or program printouts.
Please turn off your cellular phone and iPod. No electronic device is allowed.

Outcomes 2 and 3 are tested in this exam. To pass an outcome, you must receive 50% or more points in the corresponding questions.

Contents

Passed Outcomes: outcome 2 outcome 3

Total Score: out of 15.

This page is blank. You can write answers here.

1 Structure (Outcome 2, 5 points)

Assume you have a C structure for a student database with this definition:

```
typedef struct {
    char *name;          // name of student
    int year;            // birth year
} Student;
```

Please fill in your answers to the questions in the skeleton code given below.
Hint: The functions may be useful:

```
strcpy(char *dest, const char *src)
strdup(const char *s)
size_t strlen(const char *s);
malloc(size_t size)
free(void * ptr)
```

You can treat `size_t` as `int`.

(a) Write a constructor function that allocates a new Student record (i.e. object) and fills it with data passed as arguments.

(b) Write a function for retrieving the **name** of a student from a student record.

(c) Write a function for retrieving the **year** of a student from a student record.

(d) Write a function for copying a student record. Bear in mind that the copy should be **deep**, i.e., you will need to duplicate strings and not just their pointers.

(e) Write a function for deallocating all of the memory associated with a Student, including the Student record itself. Remember to deallocate all strings as well.

```
Student *Student_create(char *name, int year)
{
  /* === (a) Create a new student record (1.7 point) */











}
```

```
char *Student_getName(Student *s)
{
  /* === (b) Retrieve the name for a student record (0.4 point) */





}

int Student_getYear(Student *s)
{
  /* === (c) Retrieve the birth year for a student record (0.4 point) */





}


Student *Student_clone(Student *s)
{
  /* === (d) Copy a student record (deep copy) (1.6 point) */
  /* The new Student object should not share memory with the argument s */




}
```

```
void Student_destroy(Student *s)
{
  /* === (e) Deallocate all memory for a student record (0.9 point) */








}

int main(int argc, char **argv)
{
  Student *s1 = Student_create("John Doe", 1985);
  printf("Name: %s, Year: %d\n", Student_getName(s1),
         Student_getYear(s1));
  /* Name: John Doe, Year: 1985 */

  Student *s2 = Student_clone(s1);
  printf("Name: %s, Year: %d\n", Student_getName(s2),
         Student_getYear(s2));
  /* Name: John Doe, Year: 1985 */

  strcpy(s2 -> name, "Amy");
  s2 -> year = 1990;
  printf("Name: %s, Year: %d\n", Student_getName(s2),
         Student_getYear(s2));
  /* Name: Amy, Year: 1990 */

  printf("Name: %s, Year: %d\n", Student_getName(s1),
         Student_getYear(s1));
  /* Name: John Doe, Year: 1985 */

  Student_destroy(s1);
  Student_destroy(s2);

  return 0;
}
```

2 Linked List (Outcome 3, 7 points)

Assume you have a C structure for a linked list of integers with this definition:

```
/* Node.h */
struct Node {
    int value;
    struct Node *next;
};
```

Implement the missing code for the below four functions for inserting values at the beginning, end, and at a specific index in a linked list of integers. You may use the provided functions.

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

#include "Node.h"

struct Node *Node_create(int value)
{
  struct Node *node = malloc(sizeof(struct Node));
  node->value = value;
  node->next = NULL;
  return node;
}

void Node_destroy(struct Node *n)
{
  free(n);
}

void List_destroy(struct Node *head)
{
  struct Node * p;
  while (head != NULL)
    {
      p = head;
      head = head -> next;
      free(p);
    }
}

void List_print(struct Node *head)
{
  while (head != NULL)
    {
      printf("%d ", head->value);
      head = head->next;
    }
  printf("\n");
}

struct Node *List_insertFront(struct Node *head, int value)
{
  /* Insert an element (i.e. a Node) with an integer value at the
     beginning of the list. */
  /* head = the first element of the original list */
  /* value = the value of a new element of the list */
  /* The original list may be empty */
  /* Returns the pointer to the first element. */
  /* === FILL IN CODE BELOW (1 point) */















}

struct Node *List_insertBack(struct Node *head, int value)
{
  /* Insert an element (i.e. a Node) with an integer value at the end of
     the list. */
  /* head = the first element of the original list */
  /* value = the value of a new element of the list */
  /* The original list may be empty */
  /* Returns the pointer to the first element. */
  /* === FILL IN CODE BELOW (2 points) */
```

```
}

struct Node *List_insertAt(struct Node *head, int index, int value)
{
  /* Inserts an element (i.e. a Node) with an integer value at a
     specific location, specified by the index, in list. */
  /* If index is negative, the new element is at the beginning of the
     list */
  /* If index is larger than the list's length, the new element is
     at the end of the list. */
  /* Returns the pointer to the first element. */
  /* Hint: you can use List_insertFront and List_insertBack */
  /* === FILL IN CODE BELOW (2 points) */


}

struct Node *List_reverse(struct Node *head)
{
  /*
     NOTE: This function should
                    ===NOT=== allocate any new memory.
  */
  /* Reverse the integer values in the list so that the order of the
     element is reverse. */
  /* If a Node is the first element in the original list, this Node
     becomes the last element in the new list */
  /*  If the head pointer is NULL, this function returns NULL. */
  /* Returns the pointer to the new (reversed) list. */

  /* === FILL IN CODE BELOW (2 points) */















}

int main(int argc, char *argv[])
{
  struct Node *head = NULL;

  head = List_insertFront(head, 42);
  head = List_insertFront(head, 10);
  List_print(head); /* Output: 10 42 */

  head = List_insertBack(head, 66);
  List_print(head); /* Output: 10 42 66 */

  head = List_insertAt(head, 0, 1);
  head = List_insertAt(head, 70, 72);
  head = List_insertAt(head, -9, 0);
  List_print(head); /* Output: 0 1 10 42 66 72 */

  head = List_reverse(head);
  List_print(head); /* Output: 72 66 42 10 1 0  */
  List_destroy(head);
  return 0;
}
```

3 File (3 points)

The most basic file reading operation is `fgetc` that reads a single character from a file record pointer. It has the following function prototype:

```
int fgetc ( FILE *stream );
```

- `stream`: Pointer to a FILE object that identifies the stream the character is to be read from.
- **Return value:** `fgetc()` reads the next character from stream and returns it as an unsigned char cast to an int, or `EOF` on end of file or error.

We want to implement a new function that reads more than just one character. This function is

```
int fgetLine (char *buf, int n, FILE *stream );
```

The function reads a line from a text file. A line is defined as a sequence of (i) at most (`n` - 1) characters or (ii) until a newline ('\n') is found. The characters, in case (ii) including the newline, are stored in an array called `buf`. A '\0' character is appended to the end of the string. The function `fgetLine` does **not** check whether `buf` has enough space. This is the caller's responsibility.

- `buf`: An array of characters. It stores the characters read from the file.
- `n`: At most (`n` - 1) characters are read from the file.
- `stream`: Pointer to a FILE object that identifies the stream the character is to be read from.
- **Return value:** `fgetLine` returns the number of characters read from the file.

Please implement this `fgetLine` function.

Hint: Since `fgetc` casts a character to an integer, you can assign the return value of `fgetc` to a character:

```
char ch;
FILE * fhd;
fhd = fopen(.......);
......
ch = fgetc(fhd);
```

```
#include <stdio.h>
int fgetLine(char *buf, int n, FILE *fhd)
{
















}

int main(int argc, char **argv)
{
    if (argc < 2)
      {
        return -1;
      }
    FILE *f = fopen(argv[1], "r");
    const int BUFFER_SIZE = 10;
    char buffer[BUFFER_SIZE];
    while (fgetLine(buffer, BUFFER_SIZE, f) != 0)
      {
        printf("%s", buffer);
      }
    fclose(f);
    return 0;
}
```