University of Maryland College Park
Department of Computer Science
CMSC216 Spring 2019
Exam #2

FIRSTNAME, LASTNAME (PRINT IN UPPERCASE):

STUDENT ID (e.g., 123456789):

Instructions

- Please print your answers and use a pencil.
- This exam is a closed-book, closed-notes exam with a duration of 75 minutes and 200 total points.
- **Do not remove the exam's staple.** Removing it will interfere with the scanning process (even if you staple the exam again).
- Write your directory id (e.g., terps1, not UID) at the bottom of pages with the text DirectoryId, provide answers in the rectangular areas, and do not remove any exam pages. Even if you don't use the extra pages for scratch work, return them with the rest of the exam.
- Your code must be efficient and as short as possible.
- If you continue a problem on the extra page(s) provided, make a note on the particular problem.
- For multiple choice questions you can assume only one answer is expected, unless stated otherwise.
- You don't need to use meaningful variable names; however, we expect good indentation.
- **You must write your name and id at this point (we will not wait for you after time is up).**
- You must stop writing once time is up.
- MACHINE PARAMETERS: sizeof(char) == 1; sizeof(int) == 4; sizeof(long) == 8; sizeof(void *) == 8;
- **If the answer to a question depends on the architecture involved, assume the question applies to behavior on grace.umd.edu.**

Grader Use Only

#1	Problem #1 (Miscellaneous)	(60)	
#2	Problem #2 (File I/O)	(40)	
#3	Problem #3 (Linked Lists)	(100)	
Total	Total	(200)	

Problem #1 (Miscellaneous)

1. (3 pts) Define a Unix command using **grep** that lists the lines(s) in the file(s) ending with a .h extension containing the word **DEC**.

grep DEC *.h

2. (3 pts) Define a Unix **chmod** command that will set the file permissions for a file named **exp** as follows:

   a. The file can only be read and executed by the owner (owner has no write permissions).
   b. Users other than the owner can only execute the file (they do not have read nor write permissions).

chmod 511 exp

3. (3 pts) Which of the following expressions when assigned to new_val (the expression will replace **HERE**) will print **7**? Circle all that apply.

```
float m = 7.0;
int *a = (int *)&m;

int new_val = HERE;
printf("%d\n", new_val);
```

   a. *(int *)&m
   b. *a
   c. (int)m
   d. None of the above.

4. (3 pts) What will happen when the following code fragment is executed?

```
double *m = NULL;
void *p = m;
free(p);
```

   a. A segmentation fault might occur.
   b. A segmentation fault will always occur.
   c. Nothing bad will take place.
   d. Some memory will be freed in the heap.

5. (3 pts) Using typedef define a type named **Values** that will allow us to perform the following assignment:

```
Values values = {1.5, 1.7};
```

typdef float[2] Values;

```
typedef struct values {
    float f1, f2;
} Values;
```

6. (3 pts) The following function compiles and it is supposed to create a copy of the string parameter. Do you see any problems with the function? If you see no problems write **No** (no explanation needed); otherwise briefly describe the problem.

```
char *gen_dup(const char *p) {
   char *dup = (char *)malloc(sizeof(p) + 1);
   strcpy(dup, p);

   return dup;
}
```

sizeof(p) == 8 no matter what, its size of char* not length of p
we need strlen(p) instead for correct implementation

DirectoryId:

7. (3 pts) Define a function pointer variable named **fp** to which we can assign a function that has the following prototype:

double *linearize(int *m, char a)

double *(*linearize)(int*, char);

8. (3 pts) What is the 2's complement representation of -17 using 5 bits.

17 = 00010001
-17 = 11101111

9. (3 pts) Write the value **0xabcd1234** as it will appear in increasing locations of memory if we were using little endian.

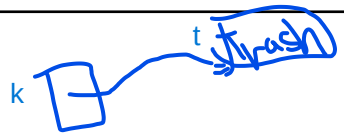
10. (3 pts) How many dangling pointer variables do we have after free(a) is executed?

```
int *d = malloc(34);
int *a = d, *b = d;
free(a);
```

3

11. (3 pts) Draw a memory map for the following code fragment. You don't need to draw the stack; just the variables **t, k** and their values. If a variable has an address write ADDR; if it is trash write TRASH.

```
char **t;       /* using two * */
char ***k = &t; /* using three * */
```



12. (4 pts) Assume a **Banana** structure has been defined. Set the variable **b** to point to a dynamically-allocated array of pointers to **Banana** structures. The size of the array is 10. Each entry of the array must be initialized to NULL.

```
Banana **b;
```

b = (Banana**)calloc(sizeof(Banana*), 10);

13. (6 pts) What is the output (**in hexadecimal**) of the following program? If you provide your answer is binary you will lose credit.

```
unsigned char x = 0x57, y = 0x4b;
printf("V1 %02x\n", x | y);
printf("V2 %02x\n", x ^ y);
```

V1 5f
V2 1c

x=0101 0111
y=0100 1010

14. (8 pts) The file **pt.c** has the code below. Define a single makefile rule that will create an executable named **pt**.

```
#include <stdio.h>
#include "seg.h"

int main() {
   printf("SEG_CONST %d\n", SEG_CONST);

   return 0;
}
```

```
pt: pt.c seg.h
    gcc -o pt pt.c
```

15. (9 pts) Recall that a nibble is 4 bits. The function **flip_clear_nibbles** flips the second and third nibbles and clears (set to 0) the first and fourth nibble of the provided value. For example, **flip_clear_nibbles(0xabcd)** will return **0x0cb0.** If you use loops you will not receive any credit for this problem. A uint16_t represents an unsigned integer of 16 bits.

```
uint16_t flip_clear_nibbles(uint16_t x){

    return ((x<<8)>>4) | ((x>>8)<<4);

      or

    return (x>>4 & 0x00f0) | (x<<4 & 0x0f00)
```

aaaabbbbccccdddd

0000ccccbbbb0000

DirectoryId:

Problem #2 (File I/O)

1. **(40 pts)** Implement the function **process_items** that has the prototype shown on the next page. The function reads and process data from the input file associated with the **in** parameter and sends output to file associated with the **out** parameter. The data to process is a file with requests for computer items. There are three types of requests:

   a. **software request** – (e.g., **software 3.4**) - Identified by a line with the word **software** followed by a float value representing a version number.
   b. **cables request** –(e.g., **cables TypeA 2) -** Identified by a line with the word **cables** followed by a string (representing the cable's type) and an integer (representing the number of cables ordered).
   c. Any line that does not start with the words **software** or **cables** is considered an invalid request. You can assume that if the **software** or **cable** words are seen at the beginning of a line, correct information will follow.

   The **process_items** function will read each request and perform the following processing based on the request's type:

   a. **software request** – The function will print (using the **out** parameter) the message "software version <VERSION> requested" where <VERSION> is the float value associated with the request.
   b. **cables request** – The function will print (using the **out** parameter) the message "<NUMBER> cable(s) type <TYPE> requested" where <NUMBER> is the number of cables and <TYPE> the specified type (e.g., TypeA in the example above).
   c. For any invalid request the program will print the message "Invalid request".

   After every request has been processed, the function will print (using the **out** parameter) how many total items need to be shipped. A **software** request counts as one item; the number of items for a **cables** request corresponds to the specified integer. The total number of items will be printed using the message "Total items:". Below we have provided an example of running the program using standard input/output. Remember your code must work for different kinds of input data. The % represents the Unix prompt. You do not need to implement a main function.

```
% cat items.txt
software 3.4
cables TypeA 2
software 2.1
pc
cables TypeB 10
% a.out < items.txt
software version 3.400000 requested
2 cable(s) type TypeA requested
software version 2.100000 requested
Invalid request
10 cable(s) type TypeB requested
Total items: 14
%
```

Standard I/O Cheat Sheet

```
FILE *fopen(const char *path, const char *mode);
int fscanf(FILE *stream, const char *format, ...);
char *fgets(char *s, int size, FILE *stream);
int sscanf(const char *str, const char *format, ...);
int fputs(const char *s, FILE *stream);
int fprintf(FILE *stream, const char *format, ...);
int fclose(FILE *fp);
```

String Library Functions Cheat Sheet

```
size_t strlen(const char *);
char *strcpy(char *restrict, const char *restrict);
int strcmp(const char *, const char *);
char *strcat(char *restrict, const char *restrict);
void *memcpy(void *restrict, const void *restrict,
size_t);
void *memmove(void *, const void *, size_t);
```

```
int process_items(FILE * in, FILE * out) {
    char line[MAX_LEN + 1], type[MAX_LEN + 1], data[MAX_LEN + 1];
    float version;
    int quantity, total_items = 0;

    while(fgets(line, MAX_LEN + 1, in){
        sscanf(line, "%s", data);
        if(strcmp(data, "software") == 0){
            sscanf(line, "%s %f", data, version);
            total_items++;
            fprintf(out, "software version %f requested\n", version);
        } else if(strcmp(data, "compare") == 0){
            sscanf(line, "%s %s %d", data, type, quantity);
            total_items+=quantity;
            fprintf(out, "%d cable(s) type %s requested\n", quantity, type);
        } else{
            fprintf(out, "Invalid request\n");
        }

        return total_items;
    }
```

DirectoryId:

Problem #3 (Linked Lists)

The following structures and definitions will be used for the questions below. A linked list is used to keep track of TV shows. A **List** structure keeps track of the actual head of the list, its name and size. A show is defined by its name and the channel that broadcasts the show. The same show can appear in the list multiple times as it can be broadcasted by several TV stations. For this problem you can assume memory allocations are always successful (do not check whether allocation fails). Feel free to use string library functions (cheat sheet can be found in the previous problem). Below we have provided a driver to illustrate the use of the functions you need to implement. The driver relies on functions you are not responsible for (e.g., add_to_list). You MAY NOT use any of these additional functions to implement the required functions. You may not add auxiliary functions in order to implement a function. Feel free to ignore the driver if you know what to implement.

```
#define MAX_LEN 80

typedef struct show {
   char name[MAX_LEN + 1];
   int channel;
} Show;
```

```
typedef struct node {
   Show *show;
   struct node *next;
} Node;
```

```
typedef struct list {
   Node *head;
   char *name;
   int size;
} List;
```

```
Show s1 = { "friends", 5 }, s2 = {"bbt", 10};
Show s3 = { "Mom", 5 };
List *list;
char result[MAX_SHOWS][MAX_LEN + 1];
int i, found;

create_list("comedy shows", &s1, &list);
add_to_list(list, &s2);
add_to_list(list, &s3);
print_list(list);

printf("Retrieving shows\n");
found = find_shows(list, 5, result);
for (i = 0; i < found; i++) {
   printf("%s\n", result[i]);
}

printf("\nAfter remove\n");
remove_first_show(&list);
print_list(list);

remove_first_show(&list);
remove_first_show(&list);
remove_first_show(&list);
```

```
List size: 3
Show: "Mom", Channel: 5
Show: "bbt", Channel: 10
Show: "friends", Channel: 5
Retrieving shows
Mom
friends

After remove
List size: 2
Show: "bbt", Channel: 10
Show: "friends", Channel: 5
```

1. **(30 pts)** Implement the function **create_list** that has the prototype shown below. The function creates a linked list with a single show. The function will dynamically allocate memory for the following items:

   a. A List structure that will be returned using the out parameter **list**.
   b. The name of the list. Use the **name** parameter to initialize the list's name.
   c. A node for the new element.
   d. A **Show** structure. Initialize the new **Show** structure using the information provided via the **show** parameter.
   e. Any other item you understand is needed.

   You can assume all pointers parameters are not NULL. Make sure you update any necessary fields.

```
void create_list(const char *name, const Show * show, List ** list){

    *list = (List*)malloc(sizeof(List));
    *list->name = (char*)malloc(strlen(name)+1);
    strcpy(*list->name, name);
    *list->head = (Node*)malloc(sizeof(Node));
    *list->head->show = (Show*)malloc(sizeof(Show));
    memcpy(*list->head->show, show, sizeof(Show));
    *list->size = 1
```

DirectoryId:

2. **(40 pts)** Implement the function **find_shows** that has the prototype shown below. The function initializes the **result** out parameter with the names of shows that are broadcasted by the specified **channel** parameter. After the last name is added to **result**, the function will add the empty string to mark the end of data. Even if no names were added, you will add the empty string to **result**. You can assume no show has as name the empty string. The function returns the number of shows found (if any). The function will return -1 and perform no processing if the pointers parameters are null and/or **channel** is less than 1. You can assume **result** is large enough to fit the answer.

```
int find_shows(List * list, int channel, char result[][MAX_LEN + 1]) {

    Node *curr = list->head;
    int num_of_shows = 0;

    if(channel < 1 || list < 1 || result == NULL)
        return -1;

    while(curr != NULL){
        if(curr->show->channel == channel){
            strcpy(result[num_of_shows++], curr->show->name);
        }
        curr = curr->next;
    }

    result[num_of_shows] = "";
    return num_shows;
```

3. **(30 pts)** Implement the function **remove_first_show** that has the prototype shown below. The function removes the first element (node) from the list and adjust the head accordingly. You need to free any dynamically allocated memory that is associated with the node. If the list is empty, the function will free any dynamically allocated memory associated with the List structure and will initialize the pointer variable associated with the out parameter **list** to NULL.

```
void remove_first_show(List ** list)

    Node *curr = *list->head;

    if(*list != NULL){
        *list->head = *list->head->next;
        free(curr->show);
        free(curr);
        *list->size--;
        if(*list->size == 0){
            free(*list->name);
            free(*list);
            *list = NULL;
        }
    }
```

DirectoryId:

EXTRA PAGE IN CASE YOU NEED IT (RETURN WITH THE EXAM)

LAST PAGE