University of Maryland College Park
Department of Computer Science
CMSC216 Summer 2023
Exam #2 Key

FIRSTNAME, LASTNAME (PRINT IN UPPERCASE):

STUDENT ID (e.g., 123456789):

Problem #1 (Linux Commands)

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

   Answer: grep IMP *.c

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

   a. The file can only be read and executed by the owner (the owner has no write permissions).
   b. Users other than the owner can only execute the file (they do not have read or write permissions).

   Answer: chmod 511 process

Problem #2 (C Constructs)

1. (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 terrible will take place.
   d. Some memory will be freed in the heap.

   Answer: c.

2. (3 pts) What is the 2's complement representation of -17 using 6 bits.

   Answer: 101111

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

```
int *d = malloc(48);
int *a = d, *b = d;
free(a);
```

   Answer: 3

4. (3 pts) Declare a function pointer named **task** that allows us to complete the assignment below.

```
double *process(int *p, float y) {
   return NULL;
}

int main() {
   /* Declare variable here */
   task = process;

   return 0;
}
```

   Answer: **double *(*task)(int *, float);**

5. (2 pts) The following code compiles and has no errors. Simplify the code so we can have the same functionality. Feel free to cross out/edit the code provided. You may not remove the **clean** function.

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

void clean(char *v) {
   if (v != NULL) {
      free(v);
   }
}

int main() {
   char *k = malloc(sizeof(char));
   *k = 'A';
   printf("%c\n", *k);
   clean(k);

   return 0;
}
```

Answer:

(2 pts) No need for the if statement in the clean function (ignore the others)

Problem #3 (Makefile)

Write a makefile to allow us to build an executable for the system associated with the code below. There are four files: spec.h, monitor.h, monitor.c and computer.c. The content of each file is shown below. Without a makefile we can build an executable named **computer** by executing gcc *.c and renaming **a.out** to **computer** (assuming only these .c files are in the current directory).

spec.h

```
int large = 20;
```

monitor.h

```
void start_monitor();
```

monitor.c

```
#include <stdio.h>
#include "monitor.h"
#include "spec.h"

void start_monitor() {
  printf("monitor started\n");
}
```

computer.c

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

int main() {
   start_monitor();
   printf("Computer system running\n");

   return 0;
}
```

The specifications associated with the makefile you need to write are:

1. **You may not use implicit rules. If you do, you will lose most of the credit for this problem.**
2. **Targets/rules**
   a. We should be able to create an executable named **computer** when we type "make".
   b. **clean** - This rule will remove any object files and the **computer** executable.
   c. **back -** This rule will copy the .c and .h files to a directory called **backup** that already exists in the current directory. The message "backup done" will be printed after the backup has been completed.
   d. Feel free to add any other targets you understand are necessary.
3. You don't need to use any compiler flags aside from -c or -o.
4. The only macro you need to use is CC and it should be initialized to gcc.

Answer:

```
CC = gcc

computer: monitor.o computer.o
        $(CC) -o computer monitor.o computer.o

monitor.o: spec.h monitor.h monitor.c
        $(CC) -c monitor.c

computer.o: monitor.h computer.c
        $(CC) -c computer.c

clean:
        rm *.o computer

back:
        @cp *.c *.h backup
        @echo "backup done"
```

Problem #4 (I/O)

The **gen_histogram** function generates a histogram based on integer values provided in a file. For this problem:

1. The file consists of one number per line.
2. A line in a file is ignored (the program moves on to the next line) if the first word in the line is IGNORE.
3. If the function's parameter is NULL, standard input will be used; otherwise, input will come from the specified file. If the file cannot be opened, the function will print the " error " message to **standard error** and return (without further processing). The program will not end because a file cannot be opened.
4. All the output is sent to standard output.
5. Use asterisks ("*") to generate the histogram.
6. Each line in the file has a maximum of 80 characters.
7. Below we provide an example of calling the function with the file data.txt (gen_histogram("data.txt")). The histogram appears to the right of the output of the cat data.txt command.
8. Cheat sheets can be found below.

```
% cat data.txt
5
  IGNORE  20
7
4
```

```
*****
*******
****
```

```
void gen_histogram(const char *file){
```

One Possible Answer:

```
#define MAX 80

void gen_histogram(const char *file) {
   FILE *input = stdin;
   char line[MAX + 1], str[MAX + 1];

   if (file) {
      if ((input = fopen(file, "r")) == NULL) {
         fprintf(stderr, "error");
         return;
      }
   }
   while (fgets(line, MAX + 1, input)) {
      int i, cnt;

      sscanf(line, "%s", str);
      if (strcmp(str, "IGNORE")) {
         sscanf(line, "%d", &cnt);
         for (i = 1; i <= cnt; i++) {
            printf("*");
         }
         printf("\n");
      }
   }
}

/* An alternative since we never specify what to do on input other than number or IGNORE */


#define MAX 80

void gen_histogram(const char *file) {
   FILE *input = stdin;
   char line[MAX + 1], str[MAX + 1];

   if (file) {
      if ((input = fopen(file, "r")) == NULL) {
         fprintf(stderr, "error");
         return;
      }
   }
   while (fgets(line, MAX + 1, input)) {
      int i, cnt;

      if (sscanf(line, "%d", &cnt) == 1) {
         for (i = 1; i <= cnt; i++) {
            printf("*");
         }
         printf("\n");
      }
   }
}
```

Problem #5 (Linked Lists)

The structures and definitions below will be used for the questions that follow. A linked list keeps track of cars in a car lot. A **Car_lot** structure keeps track of the cars' lot name (**name** field), the actual head of the list (**cars** field), and the total number of cars (**total_cars** field). The **free_car_lot_struct** field will be explained in the **init_car_lot** function description**.** A **Car** structure keeps track of a car's make and price. In addition, it has a link to the next car on the list. For this problem, you can assume memory allocations are always successful (do not check whether a dynamic-memory allocation fails). Feel free to use string library functions (a 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., print_lot). You may not add auxiliary functions to implement a function. Feel free to ignore the driver if you know what to implement.

1. Implement the **init_car_lot** function that has the prototype shown below. The function will dynamically allocate space for a **Car_lot** structure if the **lot** parameter is NULL; otherwise, it will rely on the **Car_lot** structure associated with the **lot** pointer. If dynamically-allocated space is created for the **Car_lot** structure, the function must set the **free_car_lot_struct** field to true; otherwise, it will be set to false. This field will determine whether the **Car_lot** structure must be freed when destroyed (see **destroy_lot** function description below). The function will also dynamically allocate memory for the lot's name and initialize the name using the **name** parameter. The function will set the **cars** field to NULL and the **total_cars** field to 0. The function will return a pointer to the **Car_lot** structure (created or received via the **lot** parameter). You can assume the **name** parameter is not NULL.

   ```
   Car_lot *init_car_lot(Car_lot * lot, const char *name) {
   ```

Answer:

```
Car_lot *init_car_lot(Car_lot * lot, const char *name) {
   if (lot == NULL) {
      lot = malloc(sizeof(Car_lot));
      lot->free_car_lot_struct = 1;
   } else {
      lot->free_car_lot_struct = 0;
   }
   /* It is OK if the following code only appears in the if (lot == NULL) section */
   lot->name = malloc(strlen(name) + 1);
   strcpy(lot->name, name);
   lot->cars = NULL;
   lot->total_cars = 0;

   return lot;
}
```

2. Implement the **add_car** function that has the prototype shown below. The function adds a car to the linked list associated with the **Car_lot** structure, adding cars based on their make (in increasing alphabetical order). The same make can appear multiple times in the list (cars with the same make can be inserted in any order). The function will increase the **total_cars** field after adding a car. The function will add a car (and return true) only if the pointer parameters are not NULL and the price is greater than 0; otherwise, the function will not perform any processing and return false. Make sure you initialize any other field as needed.

   ```
   int add_car(Car_lot * lot, const char *make, int price){
   ```

Answer:

```
int add_car(Car_lot * lot, const char *make, int price) {
   if (lot && make && price > 0) {
      Car *prev = NULL, *curr = lot->cars, *new_car;

      while (curr != NULL && strcmp(make, curr->make) > 0) {
         prev = curr;
         curr = curr->next;
      }

      new_car = calloc(1, sizeof(Car));
      strcpy(new_car->make, make);
      new_car->price = price;

      if (prev == NULL) {       /* beginning of the list */
         new_car->next = lot->cars;
         lot->cars = new_car;
      } else if (curr == NULL) {        /* end of the list */
         prev->next = new_car;             /* Note: This branch is redundant */
      } else {                  /* in-between */
         new_car->next = curr;  /* Note: This will initialize curr->next to NULL in inserting to the end case */
         prev->next = new_car;
      }

      lot->total_cars++;
      return 1;
   }

   return 0;
}
```

3. Implement the **destroy_lot** function that has the prototype shown below. The function frees any dynamically-allocated memory associated with the **Car_lot** structure, with one exception: if the **free_car_lot_struct** field is false the function will not free the **Car_lot** structure; otherwise, the structure will be freed. You can assume the **lot** parameter is not NULL.

   ```
   void destroy_lot(Car_lot * lot) {
   ```

Answer:

```
void destroy_lot(Car_lot * lot) {
   Car *curr = lot->cars, *to_delete;

   while (curr) {
      to_delete = curr;
      curr = curr->next;

      free(to_delete);
   }

   free(lot->name);
   if (lot->free_car_lot_struct) {
      free(lot);
   }
}
```