```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>

/*
Name: Bhaskar Gopati
blazerID: gbhaskar.
Homework 3
To compile: gcc -wall hm3.c -o hw3
To run: ./<name of executable> <commands and arguments> <directory>
ex: ./hw3 -s 1024 -e "ls -l"
   ./hm3 -f jpg -E "tar cvf jpg.tar"
   ./hw3 -s 1024 -e "wc -l"
*/

typedef struct
{
   short S_flag; // is the S flag provided?
   short s_flag; // is the s flag provided?
   short f_flag; // is the f flag provided?
   short t_flag; // is the t flag provided?
   short e_flag; // is the e flag provided?
   short E_flag; // is the E flag provided?
```

```
    int fileSize;        // s flag value
    char filterTerm[300]; // f flag value
    char fileType[2];     // t flag value
    char unix_cmd_e[400]; // unix command for e
    char unix_cmd_E[400]; // unix command for E
} FlagArgs;

// for storing path required to exec commands of -E flag
char *fp_E[1000];

int fp_count_E = 0;

pid_t current_process_id;

// function pointer.
typedef void FileHandler(char *filePath, char *dirfile, FlagArgs flagArgs, int nestingCount);

void exec_E_flag_commands()
{
    int status_code;

    // forking process
    current_process_id = fork();

    if (current_process_id == 0)
    {
        execvp(fp_E[0], fp_E);
        perror("exec");
        exit(-1);
```

```
    }
    else if (current_process_id > 0)
    {
        wait(&status_code);
        // checking for the exit status_code of the child process
        if (WIFEXITED(status_code) != 1)
            printf("Child process closed abruptly \n");
    }
    else
    {
        perror("fork");
        exit(EXIT_FAILURE);
    }
}

// the function that will be used for this assignment
void myPrinterFunction(char *filePath, char *dirfile, FlagArgs flagArgs, int nestingCount)
{
    struct stat buf;       // buffer for data about file
    lstat(filePath, &buf); // very important that you pass the file path, not just file name
    char line[100];        // init some memory for the line that will be printed
    strcpy(line, "");      // verify a clean start
    strcat(line, dirfile); // init the line with the file name

    if (flagArgs.S_flag) // S case
    {
        char strsize[10];                       // allocate memory for the string format of the size
        sprintf(strsize, " %d", (int)buf.st_size); // assign the size to the allocated string
        strcat(line, strsize);                  // concatenate the line and the size
```

```
}
if (flagArgs.s_flag) // s case
{
   if (flagArgs.fileSize > (int)buf.st_size) // if the file size is less than the expected
   {
      strcpy(line, ""); // clear the line print
   }
}
if (flagArgs.f_flag) // f case
{
   if (strstr(dirfile, flagArgs.filterTerm) == NULL) // if the filter does not appear in the file
   {
      strcpy(line, ""); // clear the line print
   }
}
if (flagArgs.t_flag) // t case
{
   if (strcmp(flagArgs.fileType, "f") == 0) // if the provided t flag is "f"
   {
      if (S_ISDIR(buf.st_mode) != 0) // if the file is a dir
      {
         strcpy(line, ""); // clear the line print
      }
   }
   if (strcmp(flagArgs.fileType, "d") == 0) // if the provided t flag is "d"
   {
      if (S_ISREG(buf.st_mode) != 0) // if the file is a regular file
      {
         strcpy(line, ""); // clear the line print
```

```
        }
    }
}
if (strcmp(line, "") != 0) // check to prevent printing empty lines
{
    int i = 0;
    for (i = 0; i <= nestingCount; i++) // tab printer
        printf("\t");                  // print a tab for every nesting
    printf("%s\n", line);              // print the line after the tabs

    // adding required files to the array for the -E flag
    fp_E[fp_count_E] = line;
    fp_count_E++;

    if (flagArgs.e_flag == 1)
    {
        int status_code, index = 0;
        char *token = strtok(flagArgs.unix_cmd_e, " ");
        char *e_arg_list[500];

        while (token != NULL)
        {
            e_arg_list[index] = token;
            index++;
            token = strtok(NULL, " ");
        }

        e_arg_list[index] = filePath;
        e_arg_list[index + 1] = NULL;
```

```
        pid_t e_current_process_id = fork();

        if (e_current_process_id == 0)
        {
           execvp(e_arg_list[0], e_arg_list);
           perror("exec");
           exit(-1);
        }
        else if (e_current_process_id > 0)
        {
           wait(&status_code);

           if (WIFEXITED(status_code) != 1)
              printf("Child process closed abrubtly \n");
        }
        else
        {
           perror("fork");
           exit(EXIT_FAILURE);
        }
     }
  }
}

void readFileHierarchy(char *dirname, int nestingCount, FileHandler *fileHandlerFunction, FlagArgs
flagArgs)
{
   struct dirent *dirent;
```

```
    DIR *parentDir = opendir(dirname); // open the dir
    if (parentDir == NULL)          // check if there's issues with opening the dir
    {
        printf("Error opening directory '%s'\n", dirname);
        exit(-1);
    }
    while ((dirent = readdir(parentDir)) != NULL)
    {
        if (strcmp((*dirent).d_name, "..") != 0 &&
            strcmp((*dirent).d_name, ".") != 0) // ignore . and ..
        {
            char pathToFile[300];                       // init variable of the path to the current file
            sprintf(pathToFile, "%s/%s", dirname, ((*dirent).d_name)); // set above variable to be the path
            // printf("\n%s\n", pathToFile);                    // print the path
            fileHandlerFunction(pathToFile, (*dirent).d_name, flagArgs, nestingCount); // function pointer
call
            if ((*dirent).d_type == DT_DIR)                 // if the file is a dir
            {
                nestingCount++;                             // increase nesting before going in
                readFileHierarchy(pathToFile, nestingCount, fileHandlerFunction, flagArgs); // reccursive call
                nestingCount--;                             // decrease nesting once we're back
            }
        }
    }
    closedir(parentDir); // make sure to close the dir
}

int main(int argc, char **argv)
{
```

```
// init opt :
int opt = 0;
// init a flag struct with 0s
FlagArgs flagArgs = {
   .S_flag = 0,
   .s_flag = 0,
   .f_flag = 0,
   .t_flag = 0};

// Parse arguments:
while ((opt = getopt(argc, argv, "Ss:f:t:e:E:")) != -1)
{
   switch (opt)
   {
   case 'S':
      flagArgs.S_flag = 1; // set the S_flag to a truthy value
      break;

   case 's':
      flagArgs.s_flag = 1;            // set s_flag to true.
      flagArgs.fileSize = atoi(optarg); // set fileSize as given.
      break;

   case 'f':
      flagArgs.f_flag = 1;              // set f_flag to true.
      strcpy(flagArgs.filterTerm, optarg); // set filterTerm to what was provided
      break;

   case 't':
```

```
            flagArgs.t_flag = 1;              // set the t_flag to a truthy value
            strcpy(flagArgs.fileType, optarg); // set fileType to what was provided
            break;

        case 'e':
            flagArgs.e_flag = 1;              // set the e_flag to a truthy value
            strcpy(flagArgs.unix_cmd_e, optarg); // set unix_cmd to what was provided
            break;

        case 'E':
            flagArgs.E_flag = 1;                      // set the E_flag to a truthy value
            strcpy(flagArgs.unix_cmd_E, optarg);          // set unix_cmd to what was provided
            char *token = strtok(flagArgs.unix_cmd_E, " "); // parse the command as tokens and store them
in the fp_E array
            while (token != NULL)
            {
                fp_E[fp_count_E] = token;
                fp_count_E++;
                token = strtok(NULL, " ");
            }
            break;
        }
    }

    char *workingDir = argv[argc - 1];
    if (opendir(workingDir) == NULL) // check for if a dir is provided
    {
        workingDir = getcwd(NULL, 0); // if not, set workingDir to the current dir
    }
```

```
    printf("Current working dir: %s\n", workingDir); // prints the top-level dir
    readFileHierarchy(workingDir, 0, myPrinterFunction, flagArgs);

    // printf("%d", flagArgs.E_flag);
    if (flagArgs.E_flag == 1)
    {
        exec_E_flag_commands();
    }
    return EXIT_SUCCESS;
}
```