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

#define MAX_PATH_SIZE 2000

char *filetype(unsigned char type) {
    char *str;
    switch (type) {
        case DT_BLK:
            str = "block device";
            break;
        case DT_CHR:
            str = "character device";
            break;
        case DT_DIR:
            str = "directory";
            break;
        case DT_FIFO:
            str = "named pipe (FIFO)";
            break;
        case DT_LNK:
            str = "symbolic link";
            break;
        case DT_REG:
            str = "regular file";
            break;
        case DT_SOCK:
            str = "UNIX domain socket";
            break;
        case DT_UNKNOWN:
            str = "unknown file type";
            break;
        default:
            str = "UNKNOWN";
    }
    return str;
}

struct params;

typedef void (*funcTarget)(struct params p);

struct params {
    struct dirent *dirent;
    char *target;
    int tabSpaces;
    int count;
    struct stat *buf;
    ssize_t b;
    int SFlag;
    char *subStr;
    int fileSize;
    int depth;
    int argsFlag;
    char *fileType;
    funcTarget funcTargCall;
    char *file;
    char *unixCommand;
};
```

```
void displayForStringFlag(struct params p);

void displayForSizeFlag(struct params p);

void displayStatFlag(struct params p);

void displayFileTypeFlag(struct params p);

void printFile(struct params p, ssize_t b);


void chooseNextFunc(struct params p);


void forkFunc(char *target, char *command);

/**
 * Returns cwd but just the current folder
 * @return current folder
 */
char *get_current_directory_name() {
    char *cwd;

    size_t size = MAX_PATH_SIZE;
    cwd = malloc(sizeof(char) * size);

    if (getcwd(cwd, size) != NULL) {
        const char *directory_name = basename(cwd);

        char *result = malloc(sizeof(char) * (strlen(directory_name) + 1));
        strcpy(result, directory_name);

        free(cwd);

        return result;
    } else {
        free(cwd);
        return NULL;
    }
}

/**
 * Takes in stat for a file and a tab indentation for formatting and prints out
 * the file size, permissions, and last access time.
 * @param sb
 * @param tab
 */
void printStat(struct stat sb, int tab) {
    // The conversion formatting for the permissions was borrowed from a stack overflow thr
ead: https://stackoverflow.com/questions/10323060/printing-file-permissions-like-ls-l-using
-stat2-in-c
    printf("%*s  File Permissions:         ", tab, " ");
    printf((S_ISDIR(sb.st_mode)) ? "d" : "-");
    printf((sb.st_mode & S_IRUSR) ? "r" : "-");
    printf((sb.st_mode & S_IWUSR) ? "w" : "-");
    printf((sb.st_mode & S_IXUSR) ? "x" : "-");
    printf((sb.st_mode & S_IRGRP) ? "r" : "-");
    printf((sb.st_mode & S_IWGRP) ? "w" : "-");
    printf((sb.st_mode & S_IXGRP) ? "x" : "-");
    printf((sb.st_mode & S_IROTH) ? "r" : "-");
    printf((sb.st_mode & S_IWOTH) ? "w" : "-");
    printf((sb.st_mode & S_IXOTH) ? "x" : "-");
    printf("\n");
    // This formatting was taken from the man page
    if (S_ISDIR(sb.st_mode)) {
        printf("%*s  File size:                %lld bytes\n", tab, " ",
               (long long) 0);
    } else {
        printf("%*s  File size:                %lld bytes\n", tab, " ",
               (long long) sb.st_size);

    }
    printf("%*s  Last file access:         %s\n", tab, " ", ctime(&sb.st_atime));
}

/**
 * Method that takes in stat for a file and a size parameter and returns 1 if the file size
 is less or equal
 * to the parameter and 0 if it is not.
 * and 0 if it
 * @param sb
 * @param size
 * @return 0 or 1
 */
int checkSize(struct stat sb, int size) {
    int flag = 0;
    if (sb.st_size <= size) {
        flag = 1;
    }
    return flag;

}

/**
 * Takes in the fileName of the current file and its’ depth and also the substring and dept
h from the cmd args.
 * Then compares them and returns 1 if the substring is in the filename and the depth is le
ss than the depth limit.
 * @param fileName
 * @param substr
 * @param depth
 * @param depthLimit
 * @return 0 or 1
 */
int checkSubstr(char *fileName, char *substr, int depth, int depthLimit) {
    int flag = 0;
    if (strstr(fileName, substr) != NULL && depth <= depthLimit) {
        flag = 1;
    }
    return flag;

}

/**
 * Method to take in three strings and split the first one into two parts at the space char
acter.
 * It then assigns the seperated strings to the two other string params.
 * @param str
 * @param str1
 * @param str2
 */
void splitStringOnSpace(const char *str, char **str1, char **str2) {
    char *space_pos = strchr(str, ’ ’);
    if (space_pos == NULL) {
        *str1 = NULL;
        *str2 = NULL;
        return;
    }
    size_t len1 = space_pos - str;
    size_t len2 = strlen(space_pos + 1);
    *str1 = (char *) malloc(len1 + 1);
    *str2 = (char *) malloc(len2 + 1);
    strncpy(*str1, str, len1);
    strncpy(*str2, space_pos + 1, len2);
    (*str1)[len1] = '\0';
    (*str2)[len2] = '\0';
}

/**
 * Takes in information for a file and the flags from the cmd args.
 * it then call the printFile function to print the file
 * @param dirent
 * @param target
 * @param tabSpaces
 * @param count
 * @param buf
 * @param b
 * @param SFlag
 * @param subStr
 * @param fileSize
 * @param depth
 * @param argsFlag
 * @param fileType
 */
void displayForNoFlag(struct params p) {

    p.argsFlag--;
    if (p.argsFlag <= 0) {
        if (p.dirent->d_type == DT_LNK) {
            p.b = readlink(p.dirent->d_name, p.target, PATH_MAX - 1);
            printf("%*s [%s] (symbolic link to %s)\n", 4 * p.tabSpaces, " ", p.dirent->d_na
me, p.target);
        } else {
            printf("%*s [%s] (%s)\n", 4 * p.tabSpaces, " ", p.dirent->d_name, filetype(p.di
rent->d_type));

        }
        if (p.unixCommand != NULL && p.dirent->d_type != DT_DIR) {
            forkFunc(p.file, p.unixCommand);
        }
    }

}

/**
 * Takes in information for a file and the flags from the cmd args.
 * it then call the printFile function to print the file if requirements are met.
 * It then sets the SFlag to zero marking that the -S flag has been handled and
 * then calls the chooseNextFunc function to determine which function the file should be
 * sent to next depending on what flags are left to handle.
 * @param dirent
 * @param target
 * @param tabSpaces
 * @param count
 * @param buf
 * @param b
 * @param SFlag
 * @param subStr
 * @param fileSize
 * @param depth
 * @param argsFlag
 * @param fileType
 */
void displayStatFlag(struct params p) {
    p.argsFlag--;
    p.funcTargCall = NULL;
    printFile(p, 0);
    p.SFlag = 0;
    chooseNextFunc(p);


}
```

```
/**
 * This method takes in information for a file and the flags from the cmd args.
 * It then checks whether all flags have been handled and if they have it prints the files
 * that met the criteria for all flags
 * @param dirent
 * @param target
 * @param tabSpaces
 * @param count
 * @param buf
 * @param b
 * @param SFlag
 * @param subStr
 * @param fileSize
 * @param depth
 * @param argsFlag
 * @param fileType
 */
void printFile(struct params p, ssize_t b) {
    if (p.argsFlag <= 0) {
        if (p.dirent->d_type == DT_LNK) {
            p.b = readlink(p.dirent->d_name, p.target, PATH_MAX - 1);
            printf("%*s [%s] (symbolic link to %s)\n", 4 * p.tabSpaces, " ", p.dirent->d_na
me, p.target);
        } else {
            printf("%*s [%s] (%s)\n", 4 * p.tabSpaces, " ", p.dirent->d_name, filetype(p.di
rent->d_type));

        }
        if (p.SFlag != 0) {
            printStat(*p.buf, 4 * p.tabSpaces);
        }
        if (p.unixCommand != NULL && p.dirent->d_type != DT_DIR) {
            forkFunc(p.file, p.unixCommand);
        }
    }
}

/**
 * Takes in information for a file and the flags from the cmd args.
 * It then decides which flags have not been handled and calls the appropriate function.
 * @param dirent
 * @param target
 * @param tabSpaces
 * @param count
 * @param buf
 * @param b
 * @param SFlag
 * @param subStr
 * @param fileSize
 * @param depth
 * @param argsFlag
 * @param fileType
 * @param funcTargCall
 */
void chooseNextFunc(struct params p) {
    if (p.subStr != NULL) {
        p.funcTargCall = &displayForStringFlag;
    } else if (p.fileType != NULL) {
        p.funcTargCall = &displayFileTypeFlag;
    } else if (p.fileSize != 0) {
        p.funcTargCall = &displayForSizeFlag;
    } else if (p.SFlag == 1) {
        p.funcTargCall = &displayStatFlag;
    }
    if (p.funcTargCall != NULL) {
        (*p.funcTargCall)(p);
    }
```

```
}

/**
 * Takes in information for a file and the flags from the cmd args.
 * It then checks to see if the current file is a folder or a regular file and only
 * executes the rest of the function on the files that type matches the fileType arg.
 * it then call the printFile function to print the file if all requirements are met.
 * It then sets the fileType param to NULL marking that the -t flag has been handled and
 * then calls the chooseNextFunc function to determine which function the file should be
 * sent to next depending on what flags are left to handle.
 * @param dirent
 * @param target
 * @param tabSpaces
 * @param count
 * @param buf
 * @param b
 * @param SFlag
 * @param subStr
 * @param fileSize
 * @param depth
 * @param argsFlag
 * @param fileType
 */
void displayFileTypeFlag(struct params p) {
    p.argsFlag--;
    p.funcTargCall = NULL;
    int x;
    if (strcmp(p.fileType, "d") == 0) {
        x = DT_DIR;
    }
    if (strcmp(p.fileType, "f") == 0) {
        x = DT_REG;
    }
    if (p.dirent->d_type == x) {
        printFile(p, 0);
        p.fileType = NULL;
        chooseNextFunc(p);
    }
}

/**
 * Takes in information for a file and the flags from the cmd args.
 * Then uses the checkSize function to see if the current file is smaller than the
 * size param specified and only executes the rest of the function on those files.
 * it then call the printFile function to print the file if all requirements are met.
 * It then sets the fileType param to NULL marking that the -t flag has been handled and
 * then calls the chooseNextFunc function to determine which function the file should be
 * sent to next depending on what flags are left to handle.
 * @param dirent
 * @param target
 * @param tabSpaces
 * @param count
 * @param buf
 * @param b
 * @param SFlag
 * @param subStr
 * @param fileSize
 * @param depth
 * @param argsFlag
 * @param fileType
 */
void displayForSizeFlag(struct params p) {
    p.argsFlag--;
    p.funcTargCall = NULL;
    if (checkSize(*p.buf, p.fileSize) == 1) {
        // Changes for symbolic link
        printFile(p, 0);
        p.fileSize = 0;
        chooseNextFunc(p);
    }
}

/**
 * Takes in information for a file and the flags from the cmd args.
 * Then splits the subStr into the string to find and the depth.
 * Then uses the checkSubstr function to see if the file name contains the substr
 * and is within the specified depth and only executes the rest of the function on those fi
les.
 * it then call the printFile function to print the file if all requirements are met.
 * It then sets the subStr param to NULL marking that the -s flag has been handled and
 * then calls the chooseNextFunc function to determine which function the file should be
 * sent to next depending on what flags are left to handle.
 * @param dirent
 * @param target
 * @param tabSpaces
 * @param count
 * @param buf
 * @param b
 * @param SFlag
 * @param subStr
 * @param fileSize
 * @param depth
 * @param argsFlag
 * @param fileType
 */
void displayForStringFlag(struct params p) {
    p.argsFlag--;
    p.funcTargCall = NULL;
    if (p.dirent->d_type == DT_DIR) {
        return;
    } else {
        char *str1;
        char *str2;
        splitStringOnSpace(p.subStr, &str1, &str2);
        if (DT_DIR && checkSubstr(p.dirent->d_name, str1, p.depth, atoi(str2)) == 1) {
            printFile(p, 0);
            p.subStr = NULL;
            chooseNextFunc(p);
        }
        free(str1);
        free(str2);
    }
}


/***
 * Function to take in a command and an array and a number of token in the array and split
the command up by space
 * and put it in the array so that execvp can parse it correctly
 * @param command
 * @param array
 * @param numTokens
 */
void parseCommand(char *command, char ***array, int *numTokens) {
    char *token = strtok(command, " ");
    int i = 0;
    while (token != NULL) {
        *array = realloc(*array, (i + 1) * sizeof(char *));
        (*array)[i] = malloc(strlen(token) + 1);
        strcpy((*array)[i], token);
        i++;
        token = strtok(NULL, " ");
    }
    *numTokens = i;
}
```

```c
/***
 * Function to take in a target file and a command and execute that command on the target file using fork and wait
 * @param target
 * @param command
 */
void forkFunc(char *target, char *command) {
    // pass *command to parseCommand and return argsArray with space seperated commands in the array
    int len = strlen(command);
    char *cmd = malloc((len + 1) * sizeof(char));
    strcpy(cmd, command);
    char **argsArray = NULL;
    int numTokens;
    parseCommand(cmd, &argsArray, &numTokens);

    // now append the target file and the null character to the argsArray
    argsArray = realloc(argsArray, (numTokens + 2) * sizeof(char *));
    argsArray[numTokens] = malloc(strlen(target) + 1);
    strcpy(argsArray[numTokens], target);
    argsArray[numTokens + 1] = NULL;

    pid_t pid;
    int status;

    printf("===================================================\n");
    printf("Unix Command Output for ’%s %s’\n", command, target);
    printf("===================================================\n");
    printf("                          â\206\223                        \n");
    printf("---------------------------------------------------\n");
    pid = fork();
    if (pid == 0) { /* this is child process */
        execvp(argsArray[0], argsArray);
        printf("If you see this statement then execl failed ;-(\n");
        perror("execv");
        exit(-1);
    } else if (pid > 0) { /* this is the parent process */
//        printf("Wait for the child process to terminate\n");
        wait(&status); /* wait for the child process to terminate */
        if (WIFEXITED(status)) { /* child process terminated normally */
//            printf("Child process exited with status = %d\n", WEXITSTATUS(status));
        } else { /* child process did not terminate normally */
            printf("Child process did not terminate normally!\n");
            /* look at the man page for wait (man 2 wait) to determine
               how the child process was terminated */
        }
    } else { /* we have an error */
        perror("fork"); /* use perror to print the system error message */
        exit(EXIT_FAILURE);
    }
    printf("---------------------------------------------------\n\n\n");

//    printf("[%ld]: Exiting program .....\n", (long)getpid());
}

/**
 * This method loops through all the files in the starting directory given and calls the appropriate functions
 * based on input flags. A lot of this code was taken from the example given in lab.
 * @param path
 * @param tabSpaces
 * @param depth
 * @param argsFlag
 * @param subStr
 * @param fileSize
 * @param SFlag
 * @param fileType
 */
void traverseDirectory(char *path, int tabSpaces, int depth, int argsFlag, char *subStr, in
t fileSize, int SFlag,
                       char *fileType, char *unixCommand) {

    struct dirent *dirent;
    DIR *parentDir;

    // Open the directory.
    parentDir = opendir(path);
    if (parentDir == NULL) {
        printf("Error opening directory '%s'\n", path);
        exit(-1);
    }
    int count = 1;
    // After we open the directory, we can read the contents of the directory, file by file
.
    char *cwd = (char *) malloc(MAX_PATH_SIZE);
    struct params p;
    struct params empty = {0};
    while ((dirent = readdir(parentDir)) != NULL) {

        // If the file's name is "." or "..", ignore them. We do not want to infinitely rec
urse.
        if (strcmp(dirent->d_name, ".") == 0 || strcmp(dirent->d_name, "..") == 0) {
            continue;
        }

        // set up relative path
        char target[PATH_MAX];
        ssize_t b = -1;
        memset(cwd, 0, MAX_PATH_SIZE);
        strcat(cwd, path);
        strcat(cwd, "/");
        strcat(cwd, dirent->d_name);
        // create a stat struct to use if we need stat info
//        puts(cwd);
//        forkFunc(cwd);
        struct stat buf;
        funcTarget funcTargCall = NULL;
        if (lstat(cwd, &buf) < 0) {
            printf("lstat error for: %s\n", dirent->d_name);
        }
        // call the appropriate function depending on the flags
        p.argsFlag = argsFlag;
        p.funcTargCall = funcTargCall;
        p.dirent = dirent;
        p.target = target;
        p.file = cwd;
        p.tabSpaces = tabSpaces;
        p.count = count;
        p.buf = &buf;
        p.subStr = subStr;
        p.fileType = fileType;
        p.fileSize = fileSize;
        p.b = b;
        p.SFlag = SFlag;
        p.depth = depth;
        p.unixCommand = unixCommand;
        if (subStr != NULL) {
            funcTargCall = &displayForStringFlag;
        } else if (fileType != NULL) {
            funcTargCall = &displayFileTypeFlag;
        } else if (fileSize != 0) {
            funcTargCall = &displayForSizeFlag;
        } else if (SFlag == 1) {
            funcTargCall = &displayStatFlag;
        } else {
            funcTargCall = &displayForNoFlag;
        }
        (*funcTargCall)(p);
        p = empty;

        count++;
        // Check to see if the file type is a directory. If it is, recursively call travers
eDirectory on it.
        if (dirent->d_type == DT_DIR) {
            // Build the new file path.
            char *subDirPath = (char *) malloc(MAX_PATH_SIZE);
            strcpy(subDirPath, path);
            strcat(subDirPath, "/");
            strcat(subDirPath, dirent->d_name);
            traverseDirectory(subDirPath, tabSpaces + 1, depth + 1, argsFlag, subStr, fileS
ize, SFlag, fileType,
                              unixCommand);
        }
    }
}

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


    // Set up flags and parameters
    char *root = NULL;
    int tabSpaces = 0;
    int depth = 0;
    int opt;
    int argFlag = 0;
    char *subStr = NULL;
    int fileSize = 0;
    char *startingFolder = NULL;
    int SFlag = 0;
    char *fileType = NULL;
    char *UnixCommand = NULL;

    // check if the starting folder is provided or not
    if (argv[1] == NULL || argv[1][0] == ’-’) {
        startingFolder = ".";
        root = get_current_directory_name();
    } else {
        startingFolder = argv[1];
        root = argv[1];
    }
    printf("STARTING DIR\n[%s]\n", root);
    tabSpaces++;
    // Take in cmd args and set flags accordingly
    while ((opt = getopt(argc, argv, "Ss:f:t:e:")) != -1) {
        switch (opt) {
            case ’S’:
                argFlag++;
                SFlag = 1;
                break;
            case ’f’:
                subStr = optarg;
                argFlag++;
                break;
            case ’s’:
                argFlag++;
                fileSize += atoi(optarg);
                break;
            case ’t’:
                argFlag++;
                fileType = optarg;
                break;
            case ’e’:
                UnixCommand = optarg;
                break;
            default:
                printf("Invalid args");
        }

    }
    traverseDirectory(startingFolder, tabSpaces, depth, argFlag, subStr, fileSize, SFlag, f
ileType, UnixCommand);


    return 0;
}
```