```
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <limits.h>

/*
* Descend through the hierarchy, starting at "fullpath".
* If "fullpath" is anything other than a directory, we lstat() it,
* call func(), and return. For a directory, we call ourself
* recursively for each name in the directory.
*/

/* function type that is called for each filename */
typedef int Myfunc(const char *, const struct stat *, int);
static Myfunc myfunc;
static int myftw(char *, Myfunc *, char *[]);
static int dopath(Myfunc *, int, DIR *, char *[]);
static long nreg, ndir, nblk, nchr, nfifo, nslink, nsock, ntot;
int e = 0, E = 0, s = 0, S = 0, f = 0, tf = 0, td = 0, size = 0, isdir = 1, ftype, i, a
rc;
char *needle = "", *t, *command, *delim = " ";
char *arr[512], *array[512];

//calls the appropriate command if the -E flag is set
void Eflag(char *command, char *array[]){
    i = 0;
    // //printf("made it here\n");
    // while(array[i] != NULL){
    //     printf("%s, ", array[i++]);
    // }
    //printf("\nCommand: %s\n", command);
    pid_t pid = fork();
    if(pid == 0){
        //child
        execvp(command, array);
        printf("Jas has broken my code :(\n");
        exit(-1);
    }else if(pid > 0){
        //parent
        wait();
    }else{
        printf("The process did not fork correctly\n");
        exit(-1);
    }
}

//calls the appropriate command if the -e flag is set
void eflag(int index, char *command, char *args[], char *filename){
    //complete the args array by adding the filename
    args[index] = filename;

    pid_t pid = fork();
    if(pid == 0){
        //child
        execvp(command, args);
        printf("Something went wrong with the child :(\n");
        exit(-1);
    }else if(pid > 0){
        //parent
        wait();                              //wait for the child process to finish (avo
ids zombie process)
```

```
        //printf("Hello from the parent\n");
    }else{
        printf("The process did not fork correctly\n");
        exit(-1);
    }
}

int main(int argc, char *argv[])
{
    arc = argc;
    int ret;
    if (argc < 2){
        //err-quit
        printf("usage: ftw <starting-pathname>\n");
        exit(-1);
    }

    for(i = 2; i < argc; i++){
        //-e flag
        if( strcmp(argv[i], "-e") == 0){
            if(i != argc-2) {               //improper usage
                printf("The flag '-e' must be the last flag and its args must be surrou
nded by double quotes\n");
                exit(-1);
            }
            //To-Do
            e++;
        }

        //-E flag
        if( strcmp(argv[i], "-E") == 0){
            if(i != argc-2){                //improper usage
                printf("The flag -E must be the last flag and its args must be surround
ed by double quotes\n");
                exit(-1);
            }
            E++;
        }

        //-s flag
        if( strcmp(argv[i], "-s" ) == 0){
            if(argv[i+1] == NULL) {
                printf("The flag '-s' requires an integer \n");
                exit(-1);
            }
            s = 1;
            size = atoi(argv[i+1]);

        }
        //-S flag
        if( strcmp(argv[i], "-S" ) == 0) S = 1;
        //-f flag
        if( strcmp(argv[i], "-f" ) == 0){
            if(argv[i+1] == NULL) {         //improper usage
                printf("The flag '-f' requires a string \n");
                exit(-1);
            }
             f = 1;
            needle = argv[i+1];
        }

        //-t flag (not functional)
        if( strcmp(argv[i], "-t") == 0){
            if(argv[i+1] == NULL){          //improper usage
                printf("The flag '-t' requires a character \n");
                exit(-1);
            }
            if(strcmp(argv[i+1], "f") == 0){
                tf++;
            }else if(strcmp(argv[i+1], "d") == 0){
                td++;
            }else printf("The parameter for the flag '-t' must be either 'f' or 'd'\n")
;
        }

    }
    ret = myftw(argv[1], myfunc, argv); /* does it all */
    ntot = nreg + ndir + nblk + nchr + nfifo + nslink + nsock;
    if (ntot == 0)
        ntot = 1; /* avoid divide by 0; print 0 for all counts */
    exit(ret);
}
/*
* Descend through the hierarchy, starting at "pathname".
* The callerâ\200\231s func() is called for every file.
*/
#define FTW_F 1 /* file other than directory */
#define FTW_D 2 /* directory */
#define FTW_DNR 3 /* directory that canâ\200\231t be read */
#define FTW_NS 4 /* file that we canâ\200\231t stat */
static char *fullpath; /* contains full pathname for every file */
static size_t pathlen;

static int /* we return whatever func() returns */
myftw(char *pathname, Myfunc *func, char *argv[])
{
    fullpath = malloc(pathlen); /* malloc PATH_MAX+1 bytes */ //currently malloc path (
may need to add 1 byte later)
    /* (Figure 2.16) */
    if (pathlen <= strlen(pathname)) {
        pathlen = strlen(pathname) * 2;
        if ((fullpath = realloc(fullpath, pathlen)) == NULL)
            //err-sys
            printf("realloc failed\n");
    }
    strcpy(fullpath, pathname);
    return(dopath(func, 0, NULL, argv));
}
/*
* Descend through the hierarchy, starting at "fullpath".
* If "fullpath" is anything other than a directory, we lstat() it,
* call func(), and return. For a directory, we call ourself
* recursively for each name in the directory.
*/
static int /* we return whatever func() returns */
dopath(Myfunc* func, int tabs, DIR *prev, char *argv[])
{
    struct stat statbuf;
    struct stat sb;
    struct dirent *dirp;
    DIR *dp;
    int ret, n;
    if (lstat(fullpath, &statbuf) < 0) /* stat error */
        return(func(fullpath, &statbuf, FTW_NS));
    if (S_ISDIR(statbuf.st_mode) == 0){ /* not a directory */
        return(func(fullpath, &statbuf, FTW_F));
    }
    /*
    * Itâ\200\231s a directory. First call func() for the directory,
    * then process each filename in the directory.
    */
    if ((ret = func(fullpath, &statbuf, FTW_D)) != 0)
        return(ret);
    n=strlen(fullpath);
    if (n + NAME_MAX + 2 > pathlen) { /* expand path buffer */
        pathlen *= 2;
        if ((fullpath = realloc(fullpath, pathlen)) == NULL)
            //err-sys
            printf("realloc failed\n");
    }
    fullpath[n++] = '/';
    fullpath[n] = 0;
    if ((dp = opendir(fullpath)) == NULL) /* canâ\200\231t read directory */
        return(func(fullpath, &statbuf, FTW_DNR));

    //if the path is in a new directory, add a tab to show nested directory
    if( (prev != NULL) && (prev != dp) ){
        tabs++;
    }

    int index;
    //set arguments for e or E flag
    if(e || E){
        //var declaration
        i = 0;

        command = strtok(argv[arc-1], " ");                //last element in argv holds
 command parameters, 1st parameter is the cmd
        arr[i++] = command;                                 //duplicate 1st arg
        if(E == 0) index = i++;
        //arr[i] = "Placeholder";
        while( (t = strtok(NULL, " ")) != NULL){
            arr[i++] = t;
        }
        ++i;                                             //skip last arg for now (reserv
ed for filename)

        //note-to-self: this may need to change later to be the 2nd argument instead

        arr[i] = NULL;                                   //null-terminate
        // printf("Hello World!\n");
        // while(arr[i] != NULL){
        //     printf("%s,,,,,,,", arr[i++]);
        // }
    }

    int j = 0;                                           //index for -E args
    while ((dirp = readdir(dp)) != NULL) {
        if (strcmp(dirp->d_name, ".") == 0 || strcmp(dirp->d_name, "..") == 0){
            continue; /* ignore dot and dot-dot */
        }else{



            char *filename = dirp->d_name;
            char *tpath = fullpath;
            strcpy(&tpath[n], filename);
            //check if size is correct for -s flag
            if (lstat(fullpath, &sb) < 0) printf("ERROR\n");
            int filesize = sb.st_size;

            if (S_ISDIR(sb.st_mode) == 0) isdir = 0;
            if(td == 1){
                ftype = 1;
            }else ftype = 0;

            //ensure that file size is acceptable & substring is present if -s or -f fl
ags exist, ignore otherwise
            if( (filesize >= size) && (strstr(filename, needle) != NULL) ){
                //tabs to show nested files
                for(i = 0; i < tabs; i++){
                    printf("\t");
                }

                printf("%s", filename);
                if(S) printf("(%d)", filesize);                     //print the file si
ze if -S flag is set
                if(e) eflag(index, command, arr, filename);         //run the appropria
te command if -e flag is set
                if(E) array[j++] = filename;                        //print the file si
ze if -S flag is set
                printf("\n");
            }

        }

        strcpy(&fullpath[n], dirp->d_name); /* append name after "/" */
        if ((ret = dopath(func, tabs, dp, argv)) != 0) /* recursive */
            break; /* time to leave */
    }

    if(E == 1){
        E++;
        //array[0] = command;
        i = 0;
        int len = 0;
        while(arr[i++] != NULL){
            len = len + 1;
            //printf("%s\n", arr[i-1]);
        }
        i = 0;
        //printf("len: %d\n", len);
        while(array[i] != NULL){
            arr[i + len] = array[i];
            i++;
        }
        //printf("\n\n\nCalled Eflag\n\n\n");
        Eflag(command, arr);

    }
    fullpath[n-1] = 0; /* erase everything from slash onward */
    if (closedir(dp) < 0)
        //err-ret
        printf("can’t close directory %s\n", fullpath);
    return(ret);
}

/*
* for testing - replacing all err_XXXX(string) with printf(string) and exit(-1)
*/
static int
myfunc(const char *pathname, const struct stat *statptr, int type)
```

```
{
    switch (type) {
        case FTW_F:
            switch (statptr->st_mode & S_IFMT) {
                case S_IFREG: nreg++; break;
                case S_IFBLK: nblk++; break;
                case S_IFCHR: nchr++; break;
                case S_IFIFO: nfifo++; break;
                case S_IFLNK: nslink++; break;
                case S_IFSOCK: nsock++; break;
                case S_IFDIR: /* directories should have type = FTW_D */
                    //err-dump
                    printf("ERROR:\t\tfor S_IFDIR for %s\n", pathname);
            }
            break;
        case FTW_D:
        ndir++;
            break;
        case FTW_DNR:
            //err-ret
            printf("ERRPR:\t\tcan’t read directory %s\n", pathname);
            break;
        case FTW_NS:
            //err-ret
            printf("ERROR:\t\tstat error for %s\n", pathname);
            break;
        default:
            //err-dump
            printf("ERROR:\t\tunknown type %d for pathname %s\n", type, pathname);
    }
    return(0);
}
```