

1. Introduction. This is a routine to search through a directory hierarchy, find all the files matching the input specifications, and list them out in order according to size. It is based on **Program 4.7** in W. Richard Stevens wonderful book *Advanced Programming in the UNIX Environment*.

2. This program is written in WEB, a preprocessor for C or Pascal. This style of programming is called “Literate Programming.” For Further information see the paper *Literate Programming*, by Donald Knuth in *The Computer Journal*, Vol 27, No. 2, 1984; or the book *Weaving a Program: Literate Programming in WEB* by Wayne Sewell, Van Nostrand Reinhold, 1989.

3. The following is the top-down structure of all my CWEB programs. CWEB is just a variant of WEB that handles Standard C.

```

⟨ Global # includes 4 ⟩
⟨ Global structures 5 ⟩
⟨ Global variables 6 ⟩
⟨ Functions 9 ⟩
⟨ The main calling routine 7 ⟩

```

4.

```

⟨ Global # includes 4 ⟩ ≡
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <limits.h>
#include "ourhdr.h"

```

See also section 16.

This code is used in section 3.

5. Function type that’s called for each filename.

```

#define MAXSYSFILES 200000
⟨ Global structures 5 ⟩ ≡
typedef int(Myfunc)(const char *, const struct stat *, int);
static long filesize[MAXSYSFILES], indices[MAXSYSFILES];
static char *filenames[MAXSYSFILES];

```

This code is used in section 3.

6.

```

⟨ Global variables 6 ⟩ ≡
static Myfuncmyfunc; static int myftw (char *, Myfunc * ) ; static int dopath ( Myfunc * ) ;
static long nreg, ndir, nblk, nchr, nfifo, nslink, nsock, ntot;

```

See also sections 13, 15, and 18.

This code is used in section 3.

7. Here is the top calling routine. I recursively descend the directory saving the file names which match the parameters.

⟨The main calling routine 7⟩ ≡

```
int main(int argc, char *argv[]) { int ret;
    extern int optind; ⟨Parse input parameters 14⟩
    for (; optind < argc; optind++) ret = myftw(argv[optind], myfunc);    /* does it all */
    ⟨Print statistics 8⟩
    ⟨Sort names 17⟩
    ⟨Output data 19⟩
    exit(ret); }
```

This code is used in section 3.

8. Here I print out the statistics.

⟨Print statistics 8⟩ ≡

```
if ((ntot = nreg + ndir + nblk + nchr + nfifo + nslink + nsock) == 0) ntot = 1;
printf("regular_files=%7ld, %5.2f%%\n", nreg, nreg * 100.0 / ntot);
printf("directories=%7ld, %5.2f%%\n", ndir, ndir * 100.0 / ntot);
printf("block_special=%7ld, %5.2f%%\n", nblk, nblk * 100.0 / ntot);
printf("char_special=%7ld, %5.2f%%\n", nchr, nchr * 100.0 / ntot);
printf("FIFOs=%7ld, %5.2f%%\n", nfifo, nfifo * 100.0 / ntot);
printf("symbolic_links=%7ld, %5.2f%%\n", nslink, nslink * 100.0 / ntot);
printf("sockets=%7ld, %5.2f%%\n", nsock, nsock * 100.0 / ntot);
if (do_uid) printf("\n%s_files=%7ld, %5.2f%%\n", pwd->pw_name, ureg * 100.0 / ntot);
```

This code is used in section 7.

9. The routine *myftw()* is lifted whole from *APUE*. Descend through the hierarchy, starting at "pathname". The caller's *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 can't be read */
#define FTW_NS 4   /* file that we can't stat */

⟨Functions 9⟩ ≡
static char *fullpath;    /* contains full pathname for every file */
static int    /* we return whatever func() returns */
myftw(char *pathname, Myfunc *func)
{
    fullpath = path_alloc(Λ);    /* malloc's for PATH_MAX + 1 bytes */    /* (Prog pathalloc) */
    strcpy(fullpath, pathname);    /* initialize fullpath */
    return (dopath(func));
}
```

See also sections 10, 11, and 20.

This code is used in section 3.

10. Function `dopath()`. 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.

(Functions 9) +≡

```
static int      /* we return whatever func() returns */
dopath(Myfunc *func)
{
    struct stat statbuf;
    struct dirent *dirp;
    DIR *dp;
    int ret;
    char *ptr;
    if (lstat(fullpath, &statbuf) < 0) return (func(fullpath, &statbuf, FTW_NS)); /* stat error */
    if (S_ISDIR(statbuf.st_mode) == 0) return (func(fullpath, &statbuf, FTW_F));
        /* not a directory */      /* * It's 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);
    ptr = fullpath + strlen(fullpath); /* point to end of fullpath */
    *ptr++ = '/';
    *ptr = 0;
    if ((dp = opendir(fullpath)) == 0) return (func(fullpath, &statbuf, FTW_DNR));
        /* can't read directory */
    while ((dirp = readdir(dp)) != 0) {
        if (strcmp(dirp->d_name, ".") == 0 ∨ strcmp(dirp->d_name, "..") == 0) continue;
            /* ignore dot and dot-dot */
        strcpy(ptr, dirp->d_name); /* append name after slash */
        if ((ret = dopath(func)) != 0) /* recursive */
            break; /* time to leave */
    }
    ptr[-1] = 0; /* erase everything from slash onwards */
    if (closedir(dp) < 0) err_ret("can't close directory %s", fullpath);
    return (ret);
}
```

11. This is my function.

⟨Functions 9⟩ +≡

```
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++; ⟨Process regular file 12⟩
        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: err_dump("for_ S_IFDIR_for_ %s", pathname);
        /* directories should have type = FTW_D */
        } break;
    case FTW_D: ndir++;
        break;
    case FTW_DNR: err_ret("can't_read_directory_ %s", pathname);
        break;
    case FTW_NS: err_ret("stat_error_for_ %s", pathname);
        break;
    default: err_dump("unknown_type_ %d_for_ pathname_ %s", type, pathname); } return (0); }
```

12.

⟨Process regular file 12⟩ ≡

```
if ((uid == statptr->st_uid) ∨ (¬do_uid)) {
    ureg++;
    filenames[ureg] = malloc(strlen(pathname) + 1);
    strcpy(filenames[ureg], pathname);
    filesize[ureg] = statptr->st_size;
}
```

This code is used in section 11.

13. Let me declare the new variables used above.

⟨Global variables 6⟩ +≡

```
long ureg;
```

14.

⟨ Parse input parameters 14 ⟩ ≡

```

    yrcnt = 50;
    while ((c = getopt(argc, argv, "n:u:")) != EOF) {
        switch (c) {
            case 'n': yrcnt = atoi(optarg);
                break;
            case 'u': pwd = getpwnam(optarg);
                if (pwd == Λ) {
                    fprintf(stderr, "Error: username %s not found.\n", optarg);
                    exit(2);
                }
            else {
                uid = pwd->pw_uid;
                do_uid = TRUE;
            }
            break;
            case '?': errflg++;
        }
    }
    if (optind ≥ argc) errflg++;
    if (errflg) {
        err_quit("usage: _dircsort_ [-n<#>] [-u<username>] _path1...");
    }

```

This code is used in section 7.

15. Now let me declare all the new variables used above.

```

#define TRUE 1
#define FALSE 0
⟨ Global variables 6 ⟩ +=
    extern char *optarg;
    int errflg, yrcnt, c;
    int do_uid = FALSE;
    uid_t uid;
    struct passwd *pwd;

```

16.

```

⟨ Global # includes 4 ⟩ +=
#include <pwd.h>

```

17.

```

⟨ Sort names 17 ⟩ ≡
#if defined (DEBUG)
    printf("Sorting_file_sizes.\n");
#endif
    if (ureg > 1) indexx(ureg, filesize, indices);
#if defined (DEBUG)
    for (j = 1; j ≤ ureg; j++) {
        printf("%s, indices[%d]=%d\n", filenames[j], j, indices[j]);
    }
#endif

```

This code is used in section 7.

18.

```

⟨ Global variables 6 ⟩ +≡
    long j;

```

19.

```

⟨ Output data 19 ⟩ ≡
#if defined (DEBUG)
    printf("Outputting_data.\n");
#endif
    if (yrcnt > ureg) yrcnt = ureg;
    for (j = ureg; j > ureg - yrcnt; j--) {
        fprintf(stdout, "%3d. (%4d_bytes)\t\t%s\n", (ureg - j + 1), filesize[indices[j]], filenames[indices[j]]);
    }

```

This code is used in section 7.

20. Sort the indices of an array. Lifted from *Numerical Recipes*.

(Functions 9) +≡

```

void indexx(n, arrin, indx)
    int n, indx[];
    float arrin[];
{
    int l, j, ir, indxt, i;
    float q;
    for (j = 1; j ≤ n; j++) indx[j] = j;
    l = (n ≫ 1) + 1;
    ir = n;
    for ( ; ; ) {
        if (l > 1) q = arrin[(indxt = indx[--l])];
        else {
            q = arrin[(indxt = indx[ir])];
            indx[ir] = indx[1];
            if (--ir ≡ 1) {
                indx[1] = indxt;
                return;
            }
        }
        i = l;
        j = l ≪ 1;
        while (j ≤ ir) {
            if (j < ir ∧ arrin[indx[j]] < arrin[indx[j + 1]]) j++;
            if (q < arrin[indx[j]]) {
                indx[i] = indx[j];
                j += (i = j);
            }
            else j = ir + 1;
        }
        indx[i] = indxt;
    }
}

```

21. Index.

argc: [7](#), [14](#).
argv: [7](#), [14](#).
arrin: [20](#).
atoi: [14](#).
c: [15](#).
closedir: [10](#).
d_name: [10](#).
DEBUG: [17](#), [19](#).
DIR: [10](#).
dirent: [10](#).
dirp: [10](#).
do_uid: [8](#), [12](#), [14](#), [15](#).
dopath: [6](#), [9](#), [10](#).
dp: [10](#).
EOF: [14](#).
err_dump: [11](#).
err_quit: [14](#).
err_ret: [10](#), [11](#).
errflg: [14](#), [15](#).
exit: [7](#), [14](#).
FALSE: [15](#).
filenames: [5](#), [12](#), [17](#), [19](#).
filesize: [5](#), [12](#), [17](#), [19](#).
fprintf: [14](#), [19](#).
FTW_D: [9](#), [10](#), [11](#).
FTW_DNR: [9](#), [10](#), [11](#).
FTW_F: [9](#), [10](#), [11](#).
FTW_NS: [9](#), [10](#), [11](#).
fullpath: [9](#), [10](#).
func: [9](#), [10](#).
getopt: [14](#).
getpwnam: [14](#).
i: [20](#).
indexx: [17](#), [20](#).
indices: [5](#), [17](#), [19](#).
indx: [20](#).
indxt: [20](#).
int: [5](#).
ir: [20](#).
j: [18](#), [20](#).
l: [20](#).
lstat: [10](#).
main: [7](#).
malloc: [12](#).
MAXSYFILES: [5](#).
myftw: [6](#), [7](#), [9](#).
myfunc: [6](#), [7](#), [11](#).
Myfunc: [5](#), [6](#), [9](#), [10](#).
n: [20](#).
nblk: [6](#), [8](#), [11](#).
nchr: [6](#), [8](#), [11](#).
ndir: [6](#), [8](#), [11](#).
nfifo: [6](#), [8](#), [11](#).
nreg: [6](#), [8](#), [11](#).
nslink: [6](#), [8](#), [11](#).
nsock: [6](#), [8](#), [11](#).
ntot: [6](#), [8](#).
opendir: [10](#).
optarg: [14](#), [15](#).
optind: [7](#), [14](#).
passwd: [15](#).
path_alloc: [9](#).
PATH_MAX: [9](#).
pathname: [9](#), [11](#), [12](#).
printf: [8](#), [17](#), [19](#).
ptr: [10](#).
pw_name: [8](#).
pw_uid: [14](#).
pwd: [8](#), [14](#), [15](#).
q: [20](#).
readdir: [10](#).
ret: [7](#), [10](#).
S_IFBLK: [11](#).
S_IFCHR: [11](#).
S_IFDIR: [11](#).
S_IFIFO: [11](#).
S_IFLNK: [11](#).
S_IFMT: [11](#).
S_IFREG: [11](#).
S_IFSOCK: [11](#).
S_ISDIR: [10](#).
st_mode: [10](#), [11](#).
st_size: [12](#).
st_uid: [12](#).
stat: [5](#), [10](#), [11](#).
statbuf: [10](#).
statptr: [11](#), [12](#).
stderr: [14](#).
stdout: [19](#).
strcmp: [10](#).
strcpy: [9](#), [10](#), [12](#).
strlen: [10](#), [12](#).
TRUE: [14](#), [15](#).
type: [11](#).
uid: [12](#), [14](#), [15](#).
uid_t: [15](#).
ureg: [8](#), [12](#), [13](#), [17](#), [19](#).
yrcent: [14](#), [15](#), [19](#).

- ⟨ Functions [9](#), [10](#), [11](#), [20](#) ⟩ Used in section [3](#).
- ⟨ Global structures [5](#) ⟩ Used in section [3](#).
- ⟨ Global variables [6](#), [13](#), [15](#), [18](#) ⟩ Used in section [3](#).
- ⟨ Global # **includes** [4](#), [16](#) ⟩ Used in section [3](#).
- ⟨ Output data [19](#) ⟩ Used in section [7](#).
- ⟨ Parse input parameters [14](#) ⟩ Used in section [7](#).
- ⟨ Print statistics [8](#) ⟩ Used in section [7](#).
- ⟨ Process regular file [12](#) ⟩ Used in section [11](#).
- ⟨ Sort names [17](#) ⟩ Used in section [7](#).
- ⟨ The main calling routine [7](#) ⟩ Used in section [3](#).

Directory Sort

(Version 1.0, May 1993
(Ansi C Version)

	Section	Page
Introduction	1	1
Index	21	8