CSC 357 Lecture Notes Week 5

More on Files and Directories
Function Pointers in C
Standard I/O Library
System Data Files and Information
Administrative Matters

- The midterm is rescheduled to **Friday 11 May**

- It will be held during both lecture and lab.

- Details forthcoming.
I. Relevant reading:

A. Stevens Chs 4, 5, 6.

B. K&R Section 5.11.

C. Man pages for referenced functions.
II. Reading directories (Stevens Section 4.21).

A. Useful example on pp. 121-125.

1. Shows opendir, readdir, closedir.

2. Also illustrates use of struct dirent.
Reading directories, cont’d

B. Signatures of directory-related system functions:

DIR* opendir(const char *filename);
struct dirent *readdir(DIR* dirp);
int closedir(DIR* dirp);
Reading directories, cont’d

C. **DIR** type is internal, similar to **FILE**.

D. **struct dirent** has accessible info.

1. Def is implementation-dependent.

2. On falcon/hornet, it’s `<sys/dirent.h>`:
E. Here’s the falcon/hornet definition:

```c
typedef struct dirent {
    ino_t d_ino;        /* inode */
    off_t d_off;        /* offset */
    short d_reclen;     /* record len */
    char d_name[1];     /* name */
} dirent_t;
```
III. chdir, fchdir, getcwd (Section 4.22).

A. They are defined in `<unistd.h>`.

B. Signatures:

```c
int chdir(const char *pathname);
int fchdir(int fildes);
char* getcwd(char* buf, size_t size);
```
C. `chdir` and `fchdir` change the current working directory of a process, i.e., a running program.

D. `getcwd` returns path of current working dir

E. `chdir` is system-level analog of "cd".

F. `getcwd` is system-level analog of "pwd".
IV. \texttt{ftw} and \texttt{nftw}.

A. On page 121, Stevens notes these directory traversing functions.

1. They are extensions of POSIX.

2. Normal place is \texttt{<ftw.h>}.

3. You may use \texttt{nftw} in your Program 4.
nftw, cont’d

B. Signatures:

```c
int ftw(const char *path,
    int (*fn) (const char*,
        const struct stat*,
        int),
    int depth);
```
nftw, cont’d

int nftw(const char *path,
         int (*fn) (const char*,
                    const struct stat*,
                    int,
                    struct FTW*),
         int depth,
         int flags);
nftw, cont’d

1. The `path` arg is the root of the path that the functions recursively traverse.

2. The `fn` arg is the function that is called as each element of the directory hierarchy is visited.
nftw, cont’d

3. The depth arg is an estimate the depth of the traversed directory tree.

4. For nftw, fourth flags arg controls function operation, including the traversal order and whether symbolic links are followed.
nftw, cont’d

C. Example:

```c
#include <stdio.h>
#include <ftw.h>

/****
 *
 * Example use of nftw.
 *
 */
```
nftw, cont’d

/**
 * The visit function ...
 */

int visit(const char* path,
          const struct stat* stat,
          int flags,
          struct FTW* ftw) {
nftw, cont’d

```c
printf("path=%s, mode=%o\n", path, stat->st_mode);
return 0;
```
nftw, cont’d

/**
 * The main function ...
 */

main() {
    if (nftw(".", visit, 10, 0) != 0) {
        perror("nftw");
    }
}
}
nftw, cont’d

D. Second arg to \texttt{ftw} and \texttt{nftw} is a \textit{function pointer}, which topic we need to discuss further.
V. Review of function pointers in C.

A. Just what the term says -- a pointer to a function.

1. Function pointer value treated like other values.

2. Can be assigned, placed in an array, passed to other functions, returned by functions.
Function pointers, cont’d

B. Primary utility is params.

1. Allows called function to perform different work based on the function it is passed.

2. For example, the second argument to the `ftw` functions allows their user to define exactly what happens when each element of a directory hierarchy is visited.
Function pointers, cont’d

C. The subject of function pointers is covered in Section 5.11 of K&R, on pages 118-121.

1. An example illustrates the use of a function pointer as a parameter to a sorting function.

2. Specifically, the function parameter is the comparison function used in the body of the sort.
Function pointers, cont’d

3. Making this a function pointer allows different implementations of comparison to be provided, similar to the way `Object->compareTo` method has different implementations in Java.
Function pointers, cont’d

D. A variant of the K&R example.

```c
#include <stdio.h>
#include <unistd.h>
```
Function pointers, cont’d

/**
 * Compare to char* values as ints .
 */

int intcmp(const void* a1,
          const void* a2) {
    int v1 = *((int*) a1);
    int v2 = *((int*) a2);

    if (v1 < v2) return -1;
    if (v1 > v2) return 1;
    return 0;
}
Function pointers, cont’d

/**
 * Like intcmp, but reverses ...
 */

int intcmp_reverse(const void* a1,
                    const void* a2) {
    int v1 = (*((int*) a1));
    int v2 = (*((int*) a2));

    if (v1 > v2) return -1;
    if (v1 < v2) return 1;
    return 0;
}
Function pointers, cont’d

/**
 * The main function .
 */
main() {
    int i;
    int data[6] = {1, ... };
    size_t nelems = ... ;
Function pointers, cont’d

```c
qsort((void*)data, 6, sizeof(char*), intcmp);

qsort(data, 6, sizeof(char*), intcmp_reverse);
}
```
E. Decl of system `qsort`;

```c
void qsort(void* base,
           size_t nel,
           size_t width,
           int (*compar)(
               const void*,
               const void*)
);
```
Function pointers, cont’d

1. Fourth parameter declares a pointer to a function that takes two constant `void*` arguments and returns an `int`.

2. The parentheses around the parameter name are critical, since without them the type would be

   ```c
   int *compar(
       const void*, const void*);
   ```
Function pointers, cont’d

F. We can now revisit the declaration of the second parameter to the `nftw` function, which is

```c
int (*fn) (  
    const char*,  
    const struct stat*,  
    int,  
    struct FTW*)
```
Function pointers, cont’d

1. This is pointer to function of four arguments, returning an int.

2. Decl of \texttt{visit} function is consistent with this:
Function pointers, cont’d

```c
int visit(
    const char* path,
    const struct stat* stat,
    int flags,
    struct FTW* ftw)
```
Function pointers, cont’d

G. Syntax of function pointer types is rather tricky.

1. Using *typedefs* makes function pointer decls more readable.

2. E.g., here is a def of *qsort*:
Function pointers, cont’d

typedef int (* CompareFunc)(
    const void*, const void*);

void qsort(void* base,
    size_t nel,
    size_t width,
    CompareFunc compar);
Function pointers, cont’d

3. The following an equivalent:

```c
typedef int CompareFunc(
    const void*, const void*);

void qsort(void* base,
    size_t nel,
    size_t width,
    CompareFunc* compar);
```
VI. Other file- and directory-related topics (Stevens Sections 4.4 - 4.20).

A. Sections describe additional file and directory functions.
Stevens 4.4 - 4.20, cont’d

B. Useful explanatory info:

1. file access permissions in Section 4.5.
2. file size in Section 4.12.
3. symbolic links in Section 4.16.
4. file times in Section 4.18
5. file access permission bits in Section 4.24
C. Also read and understand Sections 4.4 and 4.14.
VII. Intro to Buffered Std I/O (Section 5.1).

A. Spec’d by ISO C standard.

B. Implemented in many OSs other than UNIX.
VIII. Streams and Files (Section 5.2).

A. Chapter 3 functions centered on file descriptors.

B. Standard I/O lib centered on *streams*.

C. Streams are associated with fopen’d files.

D. Streams can be one-char-per byte ASCII or multi-byte "wide" char sets.
Streams and Files, cont’d

E. Applications should never need to examine `FILE` objects.

F. Rather, `FILE*` pointers are passed to all functions that deal with stdio streams.
IX. Std In, Out, and Err (Sect 5.3).

A. As we’ve seen, these are pre-assigned streams.

B. The POSIX-def’d file descriptors are
   stdin_FILENO, stdout_FILENO, and stderr_FILENO.

C. The FILE* stream names are stdin, stdout, and stderr.
X. **Buffering (Sect 5.4).**

A. The goal of buffering is to minimize read/write calls.

B. Three types of buffering are provided:

1. Fully buffered.
2. Line buffered.
3. Unbuffered.
Buffering, cont’d

C. Observations:

1. Buffered streams typically associated with files, with programmer-`malloc`’d buffer.

2. Buffered streams can be flushed with `fflush`.

3. Line buffered streams are typically associated with terminal devices.

4. Stderr is normally unbuffered.
Buffering, cont’d

5. ISO C requires stdin and stdout to be fully buffered unless on interactive device.

6. ISO C says stderr is never fully buffered.
Buffering, cont’d

D. Buffering scheme can be changed with `setbuf` system function.

E. `setbuf` is called after stream is open.

F. As noted above, `fflush` flushes buffer output.
XI. Opening a Stream (Sect 5.5).

A. You’ve already used `fopen` and `fclose`.

B. There are also `freopen` and `fdopen`.

1. `freopen` closes if already open

2. `fdopen` associates an existing fd with a stream.
### Opening a Stream, cont’d

C. Pages 138 and 139 cover details.

<table>
<thead>
<tr>
<th>Restriction</th>
<th>r</th>
<th>w</th>
<th>a</th>
<th>r+</th>
<th>w+</th>
<th>a+</th>
</tr>
</thead>
<tbody>
<tr>
<td>file must exist contents discarded</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>can be read</td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>can be written</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>written at end</td>
<td></td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
XII. Reading and Writing a Stream (Sect 5.6).

A. There are three types of I/O:

1. char-at-a-time -- `fgetc`, `fputc`
2. line-at-a-time -- `fgets`, `fputs`
3. direct -- `fread`, `fwrite`
Reading and Writing, cont’d

B. Stream functions return \texttt{EOF} on end or error.

C. To distinguish, \texttt{ferror} and \texttt{feof}.
Reading and Writing, cont’d

D. You’ve already used `getc`, `fgetc`, and `getchar`.

E. These provide char-at-a-time input.

F. The output analogs are `putc`, `fputc`, and `putchar`. 
XIII. Line-at-a-Time I/O (Sect 5.7).

A. The functions are `fgets`, and `fputs`.

B. Read (again) the admonishment on Page 142 about never using `gets`.
XIV. Std I/O Efficiency (Sect 5.8).

A. Stevens presents a timing example analogous to the one in Chapter 3.

B. The results are that buffered I/O is slower across the board than unbuffered.
I/O Efficiency, cont’d

Sample read times for 95MB file:

<table>
<thead>
<tr>
<th>Function</th>
<th>Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>read, 500K buf</td>
<td>6.67</td>
</tr>
<tr>
<td>fgets</td>
<td>7.15</td>
</tr>
<tr>
<td>getc</td>
<td>12.07</td>
</tr>
<tr>
<td>fgetc</td>
<td>11.42</td>
</tr>
<tr>
<td>read, 1 char buf</td>
<td>288.64</td>
</tr>
</tbody>
</table>
I/O Efficiency, cont’d

C. Observations:

1. Line-at-a-time is faster than may be expected because it uses `memcpy`.

2. Other interesting details about "system" calls versus "function calls".
XV. Binary I/O (Sect 5.9).

A. The buffered analogs of read and write are fread and fwrite.

B. These are most commonly used for array and structure I/O, where size says exactly how many bytes to read or write.
Binary I/O, cont’d

C. A practical issue is the non-portable nature of binary data.

1. Different compilers may treat struct offsets differently.

2. Different computer architectures may vary in storing multi-byte ints and floats.
XVI. Positioning a Stream (Sect 5.10).

A. The analog of lseek is fseek.

B. The ftell stream function is the analog of an interrogating call to lseek.

C. There is also the stream rewind function.
Positioning a Stream, cont’d

D. For text files, ISO C supports `fgetpos` and `fsetpos`.

E. These are preferable for text files, since low-level byte offset of `fseek` is not really of interest.
XVII. Formatted I/O (Sect 5.11).

A. You’ve been here plenty.

B. Output functions are \texttt{printf}, \texttt{fprintf}, \texttt{sprintf}, \texttt{snprintf}.

C. You may be using \texttt{sprintf} for Program 4.
Formatted I/O, cont’d

D. The tables on Page 150 summarize "%" formatting options.

E. There are variable-length-argument versions of `print` functions, names prefixed with "v".

F. These are handy in ways we’ll see in coming weeks.
Formatted I/O, cont’d

G. Formatted input provided by the `scanf` series.

H. These can be tricky to use, but are helpful in reading structured input from stdin streams.
XVIII. Implementation Details (Sect 5.12).

A. Std I/O functions call unbuffered I/O functions presented in Chapter 3.

B. The fd associated with a stream is available via `fileno` function.

C. Example on Pages 154-155 illustrates implementation issues.
XIX. Temporary Files (Sect 5.13).

A. The functions `tmpnam` and `tmpfile` are useful for creating temporary files.

B. The files have unique names, up to a `TMP_MAX` limit, which varies for ISO C and POSIX (at least 25 versus at least 10,000).
XX. Alternatives to Std I/O (Sect 5.14).

A. A potential inefficiency of Std I/O lib is extra data copying.

B. More efficient implementations pass pointers in places where copying can be avoided.

C. There are also implementations specialized for embedded systems that require a small memory footprint.
XXI. Intro to System Data Files (Sect 6.1)

A. Like any OS, UNIX uses lots of system files.

B. The root of the file system is "/".

C. There are a number of typical subdirs, including 
   /etc/, bin, /usr, /dev, /var.
XXII. Password File (Sect 6.2)

A. Standard place is `/etc/passwd`

B. System program access through `struct passwd`

C. It’s in `<pwd.h>`.
Password file, cont’d

```c
struct passwd {
    char     *pw_name;
    char     *pw_passwd;
    uid_t    pw_uid;
    gid_t    pw_gid;
    char     *pw_age;
    char     *pw_comment;
    char     *pw_gecos;
    char     *pw_dir;
    char     *pw_shell;
};
```
Password file, cont’d

D. Access functions are `getpwnam`, `getpwid`.

E. Signatures:

```c
struct passwd *getpwnam(
    const char *name);

struct passwd *getpwuid(
    uid_t uid);
```
Password file, cont’d

F. See Pages 162 - 164 for details.
XXIII. Shadow Passwords (Sect 6.3)

A. Used to strengthen encryption.

B. Can be subsumed by NIS, LDAP access.
XXIV. Group File (Sect 6.4)

A. Structure defined in `<grp.h>`.

B. Defines group as list of user names.

C. Access functions are `getgrnam`, `getgrgid`.
XXV. **Supplementary Group IDs (Sect 6.5).**

A. Defined in `/etc/group`.

B. Allows user to belong to multiple groups.
XXVI. Implementation Differences (Sect 6.6).

A. Falcon/hornet use NIS+.

B. There is also LDAP.

C. These provide access to common network password file.
XXVII. Other Data Files (Sect 6.7)

A. Various network services are common.

B. /etc/services, /etc/protocols, /etc/networks.
XXVIII. System Identification (Sect 6.9).

A. POSIX requires `uname` function.

B. Signature:

```c
int uname(struct utsname *name);
```
System Id, cont’d

C. struct utsname defined in 
<sys/utsname.h>
System Id, cont’d

```c
struct utsname {
    char    sysname[SYS_NMLN];
    char    nodename[SYS_NMLN];
    char    release[SYS_NMLN];
    char    version[SYS_NMLN];
    char    machine[SYS_NMLN];
};
```