Lecture # 4 – Standard I/O Library (Chapters 5)

- Streams and FILE Objects

Unbuffered I/O (i.e. open) returns a file descriptor (i.e. nonnegative integer)
Standard I/O (i.e. fopen) returns a file stream (i.e. FILE *)

This FILE object contains things like:

- the file descriptor
- a pointer to the buffer for the stream
- the size of the buffer
- a count of the number of characters in the buffer
- an error flag
- etc.

- Buffering

The goal of buffering in the stdio lib is to use the minimum number of read/write calls.

There are 3 types of buffering provided:

1. Fully buffered

   Actual I/O takes place when the buffer is full.
   The term **flush** describes the writing of a stdio buffer.
   A buffer can be flushed automatically (i.e. when full), or by fflush fct call.

2. Line buffered

   Actual I/O takes place when newline character is encountered.
   Line buffered I/O is typically used when output stream is a terminal.

   Note: it is possible to have lines longer than the buffer, which may cause a flush.

3. Unbuffered

   Actual I/O takes place with every stdio call.

   Note: stderr is normally unbuffered so error messages are displayed quickly.

Note: ANSI C requires:

stderr never to be fully buffered
stdin & stdout are fully buffered only when not an interactive device (terminal).

Note: Most UNIX systems use these conventions:

Stderr is always unbuffered.
All other streams are line buffered if a terminal, and fully buffered otherwise.

Setting buffering mechanism:

```
void    setbuf(FILE *fp, char *buf);
int     setvbuf(FILE *fp, char *buf, int mode, size_t size);
```

Returns 0 if OK, nonzero on error.

These functions must be called after stream is opened, but before use.

The setbuf function turns buffering on/off, and buf must point to a buffer of size BUFSIZ as defined in stdio.h. Providing a NULL for buf disables buffering.

The setvbuf function offers more control over buffering. We can use a mode of:

_IOFBF	fully buffered
_IOLBF	line buffered
_IONBF	unbuffered

Note: Be careful about using automatic variables for stdio buffers. You MUST close the stream BEFORE exiting the function.

- Opening a steam

```
FILE *fopen(const char *pathname, const char *type);
FILE *freopen(const char *pathname, const char *type; FILE *fp);
FILE *fdopen(int filedes, const char *type);
int fclose(FILE *fp);
```

fopen opens a specfied file
freopen opens a specfied file on a specific stream, closing it first (I/O redirection).
fdopen opens a stream from an open file descriptor

Type can be:

Type	Description
r or rb	Open for read
w or wb	Truncate or create for write
a or ab	Append or create for write
r+ or r+b or rb+	Open for read and write
w+ or w+b or wb+	Truncate, create for read and write
a+ or a+b or ab+	Append or create for read and write

Note: When opening for read and write, you must flush or position the file when switching from input to output or vice versa.

Note: Default file creation mask (permissions) is 666 unless affected with umask.

- Reading and Writing a stream

Different techniques:

1. character-at-a-time
2. line-at-a-time (i.e. fgets / fputs)
3. direct I/O (i.e. fread / fwrite)

Input functions:

```
int     getc(FILE *fp);
int     fgetc(FILE *fp);
int     getchar(void);
```

Returns next character if OK, EOF for end of file or error.

getchar is equivalent to getc(stdin)
getc can be implemented as a macro while fgetc will not

```
int     ferror(FILE *fp);
int     feof(FILE *fp);
void    clearerr(FILE *fp);

int     ungetc(int c, FILE *fp);
```

Note: Some implementations only support single character putback.

Output functions:

```
int     putc(int c, FILE *fp);
int     fputc(int c, FILE *fp);
int     putchar(int c);
```

Returns c if OK, EOF on error

Putchar(c) is equivalent to putc(c, stdout)
Putc can be implemented as a macro while fputc will not

- Line-at-a-time I/O

```
char    *fgets(char *buf, int n, FILE *fp);
char    *gets(char *buf);
```

Returns buf if OK, NULL of EOF or error

fgets returns a buffer of input including newline character (no more than n-1 characters)
gets is a deprecated function (no buffer size specification = OVERFLOW)
gets does not store the newline in the buffer

```
int     fputs(const char *str, FILE *fp);
int     puts(const char *str);
```

puts adds a newline while fputs does not

- Standard I/O Efficiency

Example:

```
int main(void)
{
        int     c;

        while ((c = getc(stdin)) != EOF)
                if (putc(c, stdout) == EOF)
                {
                        fprintf(stderr, "output error\n");
                        exit (1);
                }

        if (ferror(stdin))
        {
                fprintf(stderr, "input error\n");
                exit(1);
        }

        exit(0);
}
```

Note: see page 144 for the same program using fgets / fputs
Note: see Figure 5.6 page 144 for timing results with this program

- Binary I/O

size_t fread(void *ptr, size_t size, size_t nobj, FILE *fp);
size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *fp);

Returns number of objects read / written

These functions are often using to read / write a binary array or to read/write a structure.

Example:

```
float	data[10];

if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
	fprintf(stderr, "fwrite failed\n");
```

Example:

```
struct {
	short	count;
	long	total;
	char	name[NAMESIZE];
} item;

if (fwrite(&item, sizeof(item), 1, fp) != 1)
	fprintf(stderr, "fwrite failed\n");
```

Note: Files that are written using this technique must be read from the same system dur to differences in storage techniques (especially for integers and floats).

- Positioning a stream

long ftell(FILE *fp);
Returns file pos if OK, -1L on error

int fseek(FILE *fp, long offset, int whence);
Returns 0 ik OK, non zero on error

void rewind(FILE *fp);

Note: new functions are:

int fgetpos(FILE *fp, fpos_t *pos);
int fsetpos(FILE *fp, const fpos_t *pos);

Returns 0 if OK, nonzero on error

- Formatted I/O

Output:

```
int     printf(const char *format, …);
int     fprintf(FILE *fp, const char *format, …);
int     sprintf(char *buf, const char *format, …);

int     vprintf(const char *format, va_list arg);
int     vfprintf(FILE *fp, const char *format, va_list arg);
int     vsprintf(char *buf, const char *format, va_list arg);
```

Note: see C programming book for dealing with variable length args

Input:

```
int     scanf(const char *format, …);
int     fscanf(FILE *fp, const char *format, …);
int     sscanf(const char *buf, const char *format, …);
```

- Implementation details

```
int     fileno(FILE fp);          /* Return file des for a stream */
```

We can examine stdio.h to see implementation details for stdio library.

- Temporary Files

```
char    *tmpnam(char *ptr);
FILE    *tmpfile(void);
char    *tempnam(const char *dir, const char *prefix);
```

tmpnam generates a string that is a valid pathname that is unique.
tmpnam generates a different name each time it's called up to TMP_MAX times.

tmpfile creates a temporary binary file that is automatically removed when it is closed or on program termination.

tempnam allows the caller to specify directory AND prefix
See notes on page 157 concerning directory choices.

Example:

```
int main(void)
{
        char    name[L_tmpnam], line[MAXLINE];
        FILE    *fp;

        printf("%s\n", tmpnam(NULL));

        tmpnam(name);
        printf("%s\n", name);

        if ((fp = tmpfile()) == NULL)
        {
                fprintf(stderr, "cannot create temp file\n");
                exit(1);
        }

        fputs("one line of output\n", fp);
        rewind(fp);

        if (fgets(line, sizeof(line), fp) == NULL)
        {
                fprintf(stderr, "fgets error\n");
                exit(1);
        }

        fputs(line, stdout);

        exit(0);
}
```

Note: See also mkstemp

System Files (Chapter 6)

- Password File

The system password file is /etc/passwd, and its fields are colon delimited:

Description	Struct passwd member
User name	char *pw_name
Encrypted password	char *pw_passwd
UID	uid_t pw_uid
Numerical GID	gid_t pw_gid
Comment Field	char *pw_gecos
Home directory	char *pw_dir
Shell	char *pw_shell

The encrypted password is generated from a one-way algorithm using (crypt).
This routine generates 13 printable characters from the 64-char set [a-zA-Z0-9./]
The first two characters of the encrypted password is called the salt (picked at random).
Two identical passwords with different salts will have different encrypted passwords.

Quick lookups:

```
struct passwd *getpwuid(uid_t uid);
struct passwd *getpwnam(const char *name);
```

File searches:

```
struct passwd *getpwent(void);          /* return the next passwd entry */
void          setpwent(void);           /*  rewind the file */
void          endpwent(void);           /* close the file */
```

- Shadow Passwords

For security reasons, we will like to restrict access to the encrypted passwd portion of the password file.

Some systems support the use of a shadow password file, which contains usernames and encrypted passwords. The traditional password file then contains an "x" where the encryped password should be.

This shadow file is restricted to read only by root where the normal password file must be readable by all for programs like "ls".

- Group File

The system group file is /etc/group.

Description	Struct group member
Group name	char *gr_name
Encrypted password	char *gr_passwd
Numerical GID	int gr_gid
Array of user names	char **gr_mem

Quick lookups:

```
struct group *getgrgid(gid_t gid);
struct group *getgrnam(const char *name);
```

File Searches:

```
struct group    *getgrent(void);
void            setgrent(void);
void            endgrent(void);
```

- Supplementary Group ID's

In recent times, UNIX has evolved to allow a user to belong to multiple groups.

```
int     getgroups(int gidsetsize, gid_t grouplist[]);
int     setgroups(int ngroups, const gid_t grouplist[]);
int     initgroups(const char *username, gid_t basegid);
```

The function **initgroups** takes a username, determines group memberships, and calls **setgroups** to populate the summplementary group ID's for that user.

- Example (passwd_test.c):

```
int main(int argc, char **argv)
{
    struct stat      buf;
    struct passwd    *pw;
    struct group     *gr;
    char             owner[64], group[64];

    if (argc != 2)
    {
        fprintf(stderr, "usage: %s <filename>\n", argv[0]);
        return(1);
    }
```

```
        if (lstat(argv[1], &buf) < 0)
        {
                perror("stat call failed");
                return(1);
        }

        /*  Use the uid to get the owner's name */

        if ((pw = getpwuid(buf.st_uid)) == NULL)
                sprintf(owner, "%d", buf.st_uid);
        else
                strcpy(owner, pw->pw_name);

        /*  Use the gid to get the group owner's name */

        if ((gr = getgrgid(buf.st_gid)) == NULL)
                sprintf(group, "%d", buf.st_gid);
        else
                strcpy(group, gr->gr_name);

        printf("filename (%s) owner (%s) group (%s)\n", argv[1], owner, group);

        return(0);
}
```

- Other data files

  There are several important data files in UNIX, and here is a sample:

  /etc/services
  /etc/hosts
  /etc/resolv.conf
  /etc/nsswitch.conf
  <etc>

- Login Accounting

  utmp file tracks users currently logged in
  wtmp file tracks login/logouts.

- System Identification

```
int     uname(struct utsname *name);

struct utsname {
        char    sysname[9];
        char    nodename[9];
        char    release[9];
        char    version[9];
        char    machine[9];
};
```

Note: Information same as returned by uname utlility.

```
int     gethostname(char *name, int namelen);
```

Note: Information same as returned by hostname utility.

- Time and Date Routines

```
time_t  time(time_t *calptr);
```

Returns the value of time if OK, -1 on error; also puts value of time in calptr if non-null.

```
struct tm    gmtime(const time_t calptr);        /* UTC */
struct tm    localtime(const time_t calptr);     /* local time include DST */

struct tm {
        int     tm_sec;
        int     tm_min;
        int     tm_hour;
        int     tm_mday;
        int     tm_mon;         /* 0..11 */
        int     tm_year;
        int     tm_wday;
        int     tm_yday;
        int     tm_isdst;
}

time_t  mktime(struct tm tmptr);                 /* convert struct tm to time_t */
char    *asctime (const struct tm tmpptr);       /* give output similar to date */
char    *ctime(const time_t *calptr);

size_t  strftime(char *buf, size_t maxsize, const char *format, const struct tm *tmptr);
```

Note: See table in textbook for list of format options

- Example (timediff.c):

```
int main(argc, argv)
int     argc;
char    **argv;
{
        struct tm     *tm, *oldtm, logtm;
        time_t        now, old;
        int           nsecs;
        char          buf[1024];

        if (argc != 2)
        {
                fprintf(stderr, "usage: %s <secs>\n", argv[0]);
                return(1);
        }

        nsecs = atoi(argv[1]);

        if (nsecs < 0 || nsecs > 100000)
        {
                fprintf(stderr, "%s: secs out of range 0 .. 100000\n", argv[0])
;                return(1);
        }

        /*  Get the current time */

        now = time(0);
        tm = localtime(&now);

        strftime(buf, 1024, "%m/%d/%Y %T", tm);
        fprintf(stderr, "%s is the current time\n", buf);

        /*  Back it up by "nsecs" seconds */

        tm->tm_sec -= nsecs;

        old = mktime(tm);
        oldtm = localtime(&old);

        /*  Print out the results */

        strftime(buf, 1024, "%m/%d/%Y %T", tm);
        fprintf(stderr, "%s is time %d seconds ago\n", buf, nsecs);

}
```