

# CS 2211

# Systems Programming

External Files in a C Program

## FILE \*

In C, we use a FILE \* data type to access files.

FILE \* is defined in /usr/include/stdio.h

An example:

```
#include <stdio.h>
int main()
{
    FILE *fp;
    fp = fopen("tmp.txt", "w");
    fprintf(fp, "This is a test\n");
    fclose(fp);
    return 0;
}
```

## Opening a File

You must include `<stdio.h>`

Prototype Form:

```
FILE * fopen (const char * filename, const char * mode)
```

**FILE** is a structure type declared in `stdio.h`.

You don't need to worry about the details of the structure.  
In fact it may vary from system to system.

**fopen ()** returns a pointer to the FILE structure type.

You must declare a pointer of type FILE to receive that value when it is returned.

Use the returned pointer in all subsequent references to that file.

If **fopen ()** fails, NULL is returned.

The argument `filename` is the name of the file to be opened.

## Opening a File

Values of **mode**

Enclose in double quotes or pass as a string variable

Modes:

**r** : open the file for reading (NULL if it doesn't exist)

**w** : create for writing. destroy old if file exists

**a** : open for writing. create if not there. start at the end-of-file

**r+** : open for update (r/w). create if not there. start at the beginning.

**w+** : create for r/w. destroy old if there

**a+** : open for r/w. create if not there. start at the end-of-file

In the text book, there are other binary modes with the letter **b**. They have no effect in today's C compilers.

## stdin, stdout and stderr

Every C program has three files opened for them at start-up:

**stdin**, **stdout**, and **stderr**

**stdin** is opened for reading, while **stdout** and **stderr** are opened for writing

They can be used wherever a FILE \* can be used.

Examples:

```
fprintf(stdout, "Hello there!\n");
```

This is the same as `printf("Hello there!\n");`

```
fscanf(stdin, "%d", &int_var);
```

This is the same as `scanf("%d", &int_var);`

```
fprintf(stderr, "An error has occurred!\n");
```

This is useful to report errors to standard error - it flushes output as well, so this is really good for debugging!

## the exit() function

This is used to leave the program at anytime from anywhere before the “normal” exit location.

### Syntax:

```
exit (status);
```

### Example:

```
#include <stdlib.h>

.....

if( (fp=fopen("a.txt","r")) == NULL) {
    fprintf(stderr, "Cannot open file a.txt!\n");
    exit(1);
}
```

## Four ways to read and write files

Formatted file I/O

Get and put a character

Get and put a line

Block read and write

## Formatted File input and output

Formatted File input is done through `fscanf()` :

```
int fscanf (FILE * fp, const char * fmt, ...);
```

Formatted File output is done through `fprintf()` :

```
int fprintf(FILE *fp, const char *fmt, ...);
```

```
{  
    FILE *fp1, *fp2;  
    int n;  
    fp1 = fopen("file1", "r");  
    fp2 = fopen("file2", "w");  
    fscanf(fp1, "%d", &n);  
    fprintf(fp2, "%d", n);  
    fclose(fp1);  
    fclose(fp2);  
}
```



## Get and Put a character

```
#include <stdio.h>
int fgetc(FILE * fp);
int fputc(int c, FILE * fp);
```

- These two functions read or write a single byte from or to a file.
- **fgetc()** returns the character that was read, converted to an integer.
- **fputc()** returns the same value of parameter c if it succeeds, otherwise, return EOF.

## Get and Put a line

```
#include <stdio.h>
char *fgets(char *s, int n, FILE * fp);
int fputs(char *s, FILE * fp);
```

- These two functions read or write a string from or to a file.
- **fgets()** reads an entire line into **s**, up to **n-1** characters in length (pass the size of the character array **s** in as **n** to be safe!)
- **fgets()** returns the pointer **s** on success, or NULL if an error or end-of-file is reached.
- **fputs()** returns the number of characters written if successful; otherwise, return EOF.

## fwrite() and fread()

- **fread()** and **fwrite()** are binary file reading and writing functions
  - Prototypes are found in stdio.h

- Generic Form:

```
int fwrite (void *buf, int size, int count, FILE *fp);  
int fread  (void *buf, int size, int count, FILE *fp);
```

- **buf**: is a pointer to the region in memory to be written/read
    - It can be a pointer to anything (more on this later)
  - **size**: the **size** in bytes of each individual data item
  - **count**: the number of data items to be written/read

## fwrite() and fread()

- For example a 100 element array of integers
  - `fwrite( buf, sizeof(int), 100, fp);`
- The `fwrite()` [ and the `fread()` ]  
returns the number of items actually written (read).

## fwrite() and fread()

Testing for errors:

```
if ((fwrite(buf, size, count, fp)) != count)
    fprintf(stderr, "Error writing to file.");
```

Writing a single double variable x to a file:

```
fwrite (&x, sizeof(double), 1, fp) ;
```

This writes the double x to the file in raw binary format  
i.e., it simply writes the internal machine format of x

Writing an array text[50] of 50 characters can be done by:

```
fwrite (text, sizeof(char), 50, fp) ;
```

or

```
fwrite (text, sizeof(text), 1, fp) ;
/* text must be a local array name */
```

**fread()** and **fwrite()** are more efficient than **fscanf()** and **fprintf()**

## Closing and Flushing Files

- Syntax:

```
int fclose (FILE * fp) ;
```

- closes `fp` -- returns 0 if it works -1 if it fails

- You can clear a buffer without closing it

```
int fflush (FILE * fp) ;
```

- Essentially this is a force to disk.
  - Very useful when debugging.
- Without `fclose()` or `fflush()`, your updates to a file may not be written to the file on disk.

(Operating systems like Unix usually use “write caching” disk access.)

## Sequential and Random Access

- In the FILE structure, there is a long type to indicate the position of your next reading or writing.
- When you read/write, the position move forward.
- You can “rewind” and start reading from the beginning of the file again:

```
void rewind (FILE * fp) ;
```

- To determine where the position indicator is use:

```
long ftell (FILE * fp) ;
```

- Returns a long giving the current position in bytes.
- The first byte of the file is byte 0.
- If an error occurs, `ftell()` returns -1.

## Random Access

- One additional operation gives slightly better control:  
`int fseek (FILE * fp, long offset, int origin) ;`
  - `offset` is the number of bytes to move the position indicator
  - `origin` says where to move from
- Three options/constants are defined for `origin`
  - `SEEK_SET`
    - move the indicator offset bytes from the beginning
  - `SEEK_CUR`
    - move the indicator offset bytes from its current position
  - `SEEK_END`
    - move the indicator offset bytes from the end



## Detecting End of File

### □ Text mode files:

```
while ( (c = fgetc (fp) ) != EOF )
```

- Reads characters until it encounters the EOF
- The problem is that the byte of data read may actually be indistinguishable from EOF.

### □ Binary mode files:

```
int feof (FILE * fp) ;
```

- Note: the `feof` function realizes the end of file only **after** a reading failed (`fread`, `fscanf`, `fgetc` ... )

```
fseek (fp, 0, SEEK_END) ;
printf ("%d\n", feof (fp)) ;      /* zero value          */
fgetc (fp) ;                     /* fgetc returns -1 */
printf ("%d\n", feof (fp)) ;      /* nonzero value     */
```

## An Example ...

```
#define BUFSIZE 100

int main ()
{
    char buf[BUFSIZE];

    if ( (fp=fopen("file1", "r"))==NULL) {
        fprintf (stderr,"Error opening file.");
        exit (1);
    }

    while (!feof(fp)) {
        fgets  (buf,BUFSIZE,fp);
        printf ("%s",buf);
    }

    fclose (fp);
    return 0;
}
```

## File Management Functions

### □ Erasing a file:

```
int remove (const char * filename);
```

- This is a character string naming the file.
- Returns 0 if deleted; otherwise -1.

### □ Renaming a file:

```
int rename (const char * oldname, const char * newname);
```

- Returns 0 if successful or -1 if an error occurs.
- error: file **oldname** does not exist
- error: file **newname** already exists
- error: try to rename to another disk

## Using Temporary Files

- Files that only exist during the execution of the program.
- Generic Form:  
`char *tmpnam(char *s);`
  - Included in `stdio.h`.
  - Creates a valid filename that does not conflict with any other existing files.
- Note this does not create the file
  - Just the NAME!
  - You then go and open it and presumably write to it.
  - The file created will continue to exist after the program executes unless you delete it.

## An Example ...

```
#include <stdio.h>

int main () {
    char buffer[25];
    tmpnam(buffer);

    printf ("Temporary name 1: %s", buffer);

    return 0;
}
```

### OUTPUT:

```
Temporary name 1:  /var/tmp/aaaceaywB
```

# External Files in a C Program

**END OF Files ...**

