Sec. 7: File I/O & Testing

Diba Mirza , CSE30, Fall 2016

Slides by
Ashish Kashinath

Style Guidelines for ARM Code

- No Rule set in stone. Subjective, each person has their own.
- Good practice to have comments (demo)
  - Comments can be the equivalent C statement.
  - Comments can be in plain English too.
- Testing :
  - Tests should be unit Tests, meaning one test should be independent of another. This helps in isolation of bugs.
  - For example, in createList what would be the cases you'd be checking?
    - Does size get updated?
    - Does maxSize get updated?
    - Is the list* returned by createList( ) NULL?
    - Is the sortedList* NULL?

Style Guidelines for ARM Code

- Demo
  - Style1.txt
  - Style2.txt

File I/O

- User-space application: Reading in terms of blocks, not effective.

- Reading Speed: Single byte 1024 times Vs. Single 1024 block at once. So buffering in done in user-space, transparently (application i.e., the C program doesn't know)
  - time dd bs=1 count=2097152 if=/dev/zero of=pirate (Vs)
  - time dd bs=1024 count=2048 if=/dev/zero of=pirate
  - Stat command can be used for block size. `stat -f /dev/sda1`

- So, can we do all the I/O in 4 or 8KB chunks and everything is great?

Not so fast!

- Programs rarely deal in terms of blocks.
  - Instead lines, characters & numbers.
- User buffered I/O aims to close the gap between the filesystem, which speaks in blocks and the application, that talks in its own abstactions.
  - HUGE Performance benefits here.
- Impossible to develop buffering by hand in our own programs. So we use the standard I/O library (part of standard C library) called <stdio.h>
  - ANSI C Standard in 1989 (C89)
  - Changes in C96, C99, C11 and so on.

FILE*

- Operating System(Kernel) views files as file descriptors (an integer which indexes into a table called file table).
- <stdio.h> views files as *file pointers* (FILE *)
- User buffered I/O aims to close the gap between the filesystem, which speaks in blocks and the application, that talks in its own abstactions.
  - HUGE Performance benefits here.
- Impossible to develop buffering by hand in our own programs. So we use the standard I/O library (part of standard C library) called <stdio.h>
  - ANSI C Standard in 1989 (C89)
  - Changes in C96, C99, C11 and so on.

FILE*

- The file pointer is represented by a pointer to the file typedef, defined in <stdio.h>. (Open GLIBC codebase).
- Why is it all capitals (historically I/O implemented as MACROS).

FILE* Operations (Or,stream operations)

- Opening
  - Opening files (Normal)
  - Opening file via File Descriptor
- Closing
  - Closing files (Normal)
  - Closing all Streams
- Reading from a stream
  - Reading a character at a time
  - Reading an entire line
  - Reading Binary data

FILE* Operations (Or,stream operations)

- Writing
  - Writing a single character at a time.
  - Writing a string of characters
- Sample Program using streams.

Opening…

fopen()

- **FILE** * fopen (**const char** *path, **const char** *mode

Modes

- r, r+
- w, w+
- a, a+

Return Value

- Success: a valid FILE pointer
- Failure: a NULL pointer.

Opening…

Mode	Purpose	Where is the stream now? (Always note this)
r	Open the file for reading.	The stream is positioned at the start of the file.
r+	Open the file for both reading and writing.	The stream is positioned at the start of the file.
w	Open the file for writing.(If the file exists, it is truncated to zero length. If the file does not exist, it is created)	The stream is positioned at the start of the file.
w+	Open the file for both reading and writing.(If the file exists, it is truncated to zero length. If the file does not exist, it is created)	The stream is positioned at the start of the file.
a	Open the file for writing in append mode. The file is created if it does not exist.	The stream is positioned at the end of the file. All writes will append to the file.
a+	Open the file for both reading and writing in append mode. The file is created if it does not exist.	The stream is positioned at the end of the file. All writes will append to the file.

(The character b in mode) Text & Binary files are same in all POSIX-conforming systems (like Linux). As we are on Linux, b can be ignored.

Opening…

- Example of fopen( )
  - Demo (fopen_example.c)

Opening…

fdopen()

- **FILE** * fopen (**int fd, const char** *mode);

Modes

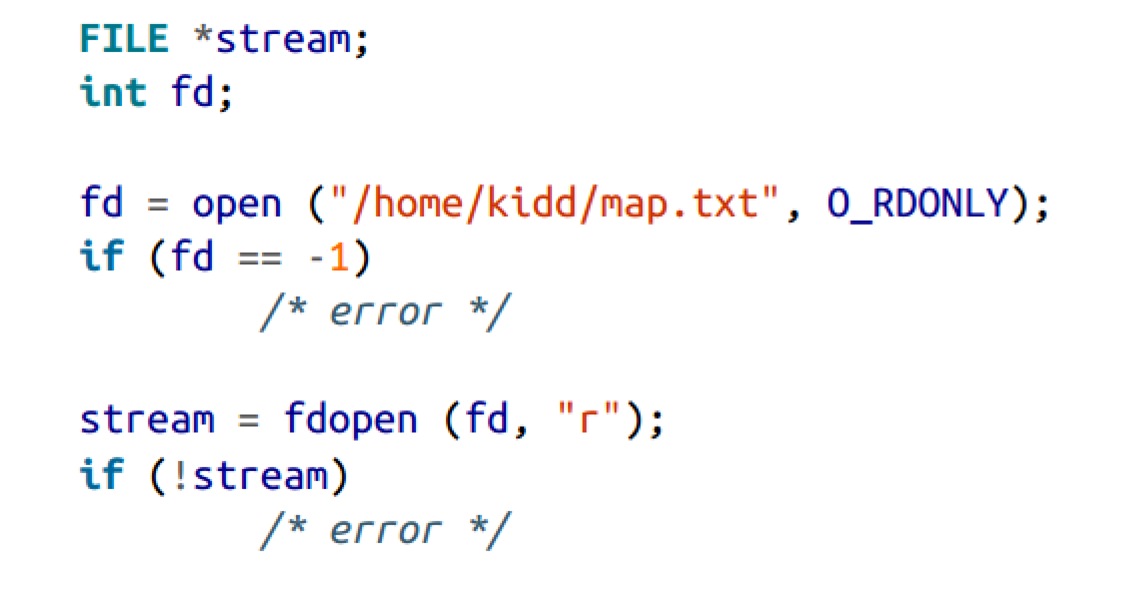
- Same as fopen( ).

Return Value

- Success: a valid FILE pointer
- Failure: a NULL pointer.

Opening...

- Example of fdopen( )



```c
FILE *stream;
int fd;

fd = open ("/home/kidd/map.txt", O_RDONLY);
if (fd == -1)
        /* error */

stream = fdopen (fd, "r");
if (!stream)
        /* error */
```

Closing…

fclose()

- `int fclose (FILE *stream);`

Return Value

- Success: returns 0
- Failure: returns EOF

fcloseall( )

- `int fcloseall();`

Return Value

- Success: returns 0
- Failure: returns EOF

Note

❑ Any buffered and not-yet-written data is first flushed.

Closing…

- Example of fclose( )
  - Demo

Reading...

- Condition for reading
  - Our file should be opened in any valid mode **except w or a**

Reading… (Reading a char at a time)

fgetc()

- **`int fgetc (FILE *stream);`**

Return Value

- Success: Returns the read character (cast to an int so that it can accommodate error values too)
- Failure: EOF.

Reading… (Reading a char at a time)

- Example of fgetc( )
  - Demo (char_example.c)

- Still interested? Explore ungetc( ) . Allows us to have a ‘peek’ at the stream.

Reading… (Reading an Entire Line)

- Syntax
  - **char** * fgets (**char** *str, **int** size, **FILE** *stream);
- Purpose:
  - (size-1) bytes read from the stream
  - NULL character (\0) is stored in the buffer after the last read char.
- Returns:
  - Success: return str
  - Failure: NULL is returned

Reading… (Reading an Entire Line)

- Demo (string_example.c)

  -
    ```
    char buf[LINE_MAX];
    if (!fgets (buf, LINE_MAX, stream))
      /* error */
    ```

- Implementing fgets( ) using fgetc( ) – What if we want a delimiter other than newline?

Reading… (Reading Binary Data)

- Syntax
  - **size_t** fread (**void** *buf, **size_t** size, **size_t** nr, **FILE** *stream);
- Purpose:
  - Read nr elements, each of size bytes, from stream, into buffer pointed to by buf
- Returns:
  - Success: Number of elements read , which should be nr
  - Failure: EOF or a value < nr
- Example of fread()
  - Demo

Writing…

- Condition for writing
  - Our file should be opened in any valid mode **<u>except r</u>**

Writing… (Writing a char at a time)

fputc()

- `int fputc (int c, FILE *stream);`

Purpose

Writes the byte specified by c (cast to an unsigned char) to the stream pointed at by stream.

Return Value

- Success: Returns the read character (cast to an int so that it can accommodate error values too)
- Failure: EOF.

Writing… (Writing a char at a time)

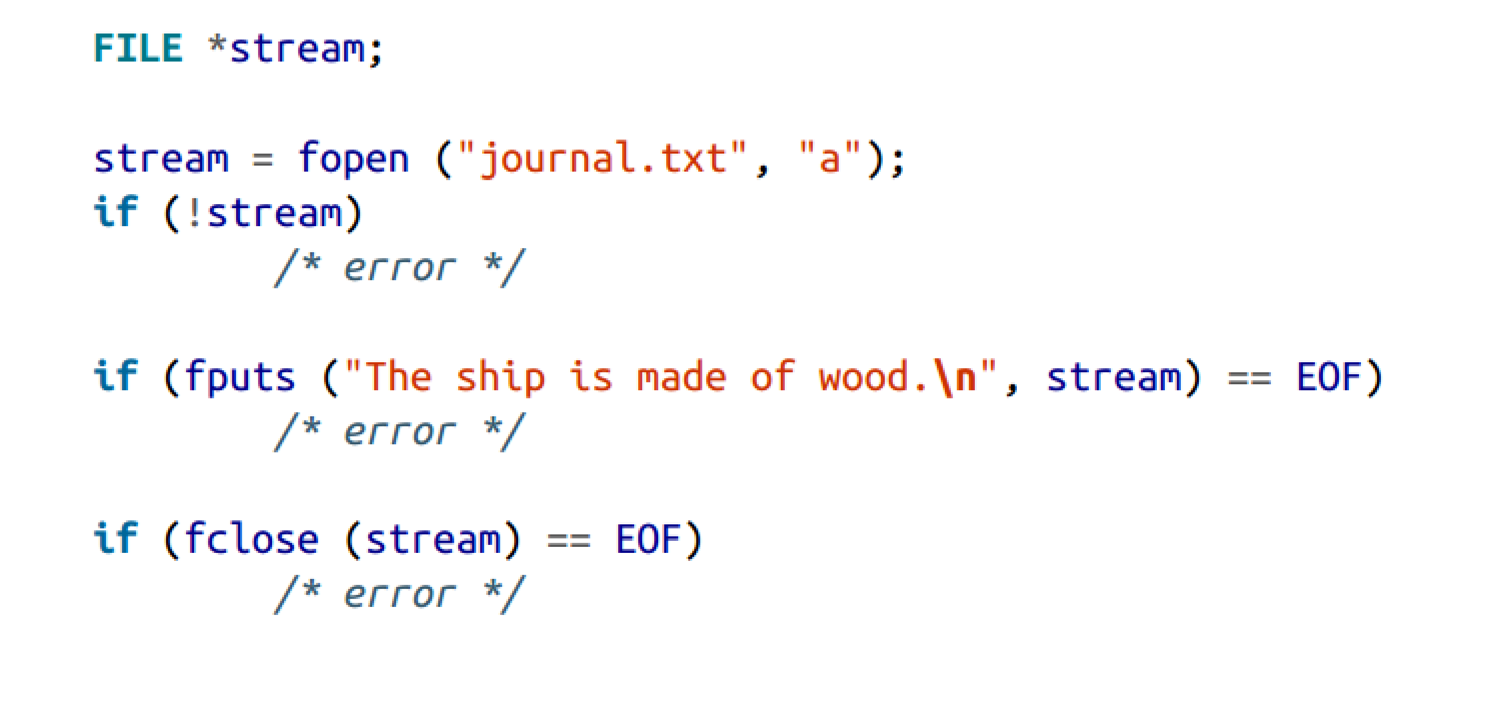
- Example of fputc( )
  - Demo
  - **if** (fputc ('p', stream) == EOF)
    \/ error */*

Writing… (Writing an Entire Line)

- Syntax:
  - **int** fputs (**const char** *str, **FILE** *stream);
- Purpose:
  - writes all of the null-terminated string pointed at by str to the stream pointed at by stream.
- Returns:
  - Success: return non-negative number
  - Failure: return EOF.

Writing… (Writing an Entire Line)

- Demo



```
FILE *stream;

stream = fopen ("journal.txt", "a");
if (!stream)
        /* error */

if (fputs ("The ship is made of wood.\n", stream) == EOF)
        /* error */

if (fclose (stream) == EOF)
        /* error */
```

Writing… (Writing Binary Data)

- Syntax
  - **size_t** fwrite (**void** *buf, **size_t** size, **size_t** nr, **FILE** *stream);
- Purpose:
  - write to stream up to nr elements, each size bytes in length, from the data pointed at by buf.
  - The file pointer will be advanced by the total number of bytes written.
- Returns:
  - Success: Number of elements successfully written, which should be nr
  - Failure: A value < nr
- Example of fwrite()
  - Demo (fread_fwrite_example.c)

Sample Program using all concepts

- Demo (buffered_io.c)

Advanced FILE* Operations

- Seeking a stream
- Flushing a stream
- Obtaining the associated file descriptor

Seeking a Stream

- Seeking in YouTube
- The same thing can be done for a file
  - Manipulate the stream position
- 3 functions: fseek( ), fsetpos( ) and rewind( )
  - `int fseek (FILE *stream, long offset, int whence);`
  - `int fsetpos (FILE *stream, fpos_t *pos);`
  - `void rewind (FILE *stream);`
- fsetpos( ) is normally used in non-Unix platforms
- Rewind( ) is like rewind in music player, setting the file position to the beginning

TLDR; Use only fseek( )

Seeking a Stream

whence	Where is the stream now? (Always note this)
SEEK_SET	The file position is at offset
SEEK_CUR	The file position is at (current position + offset)
SEEK_END	The file position is at (EOF + offset)

- Demo
    - fseek_example.c

Obtaining the Current Stream Position

- fseek() does not return the updated position.
- Standard I/O has 2 interfaces for this puspose
  - **long** ftell (**FILE** *stream);
  - **int** fgetpos (**FILE** *stream, **fpos_t** *pos);
- fgetpos( ), like fsetpos( ) is used in non-Unix platforms.
- Demo
  - Stream Positioning

Flushing a stream

- Why do we need to flush streams? What *can* go wrong?
- Interface for writing out the user buffer to the kernel, calls write( ) internally.
- Syntax:
  - `**int** fflush (**FILE** *stream);`
- Returns:
  - `Success: 1`
  - `Failure: EOF`
- `Usually followed by fsync() to ensure that the kernel buffer is written to the disk.`

Formatted I/O

- 5 family of printf functions:

Category of printf( )	Purpose
printf( )	Writes to standard output
fprintf()	Writes to the specified file pointer
dprintf()	Writes to the specified file descriptor
sprintf()	Writes to the array *buf*
snprintf()	Writes to the array *buf* n characters

– Demo (fprintf_example.c)

Formatted I/O

- 3 family of scanf functions:

Category of scanf( )	Purpose
scanf( )	Reads from standard input
fscanf()	Reads from the specified file pointer
sscanf()	Reads from the specified string.

- Demo (sscanf_example.c)

And we have only scratched the surface…

- Advanced I/O concepts
  - Non-blocking I/O
  - I/O Multiplexing
  - Asynchronous I/O
  - Memory-mapped I/O
  - Directory Entry (DIR*) like FILE *

References

- “Advanced Programming in the Unix Enviroment” 3rd Edition, W.Richard Stevens (Source code from Book under GPL License)
- “Linux System Programming” O’Reily
- Tutorialspoint.com