File I/O in Standard C

File content is accessed as a stream—sequential read/write.

Can also do random access ("seek") if file is regular file.

(Exercise: Example of not-regular file? Such that random access does not make sense, can only be sequential?)

File functions work with 'FILE *' all the time:

FILE = Type representing stream state. Actual definition varies by platform, but most likely a struct, so we pass around pointers.

(Poor naming, STREAM would be better, and "stream functions" instead of "file functions".)

Header to #include: stdio.h

Open

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

mode	meaning	if file exists	if not
"r"	read	cool	error
"w"	write	truncate	create
"a"	append	cool	create
"r+"	read+write	cool	error
"w+"	write+read	truncate	create
"a+"	append+read	cool	create

"read" and "write" start at the beginning of the file.
"append" start at the end of the file.
"truncate" = erase original content.

Returns NULL if error. (This one you should always check.)

Windows: Text vs Binary Files

Windows uses two consecutive bytes 13, 10 ("carriage return", "linefeed") for end of line. (Unix just uses a 10.) Other OSes can be different too.

Windows also puts byte 26 at end of text file.

Normally, C I/O functions auto-convert to/from 10 and hide the 26, you code like it's always Unix. Makes sense for text files.

In binary files (raw bytes), 10, 13, and 26 stand for themselves, you want no conversion. Add 'b' to mode to indicate this, e.g, `"rb"`, `"r+b"`, `"rb+"`. (Just not `"br"`, `"br+"`.)

On Unix, the additional 'b' makes no difference.

"Carriage return" and "linefeed" are from typewriter jargon.

Close

Close stream when you're done:

```
int fclose(FILE *stream)
```

Returns 0 if success, EOF if error.

Why need to close ASAP:

► Limit on how many streams are open per process.

► Writing may be delayed (buffered) until closing. (More on buffering later.)

► Windows: Restrictions on multiple processes opening the same file. (Allowed only when all open in read-only mode.)

Formatted I/O

Like printf and scanf but for arbitrary streams:

```
int fprintf(FILE *stream, const char *format, ...)
int fscanf(FILE *stream, const char *format, ...)
```

In fact:

```
printf(format, args) = fprintf(stdout, format, args)
scanf(format, args) = fscanf(stdin, format, args)
```

stdin, stdout, stderr are pre-opened, and no need to manually close.

Exercise:
Find out what printf and fprintf return.
Find out what scanf and fscanf return.

Character I/O: Output

Output one single character:

```
int putchar(int c)   /* stdout */
int putc(int c, FILE *stream)
```

Returns the character written if success, EOF if error.

Character I/O: Input

Input one single character:

```
int getchar(void)    /* stdin */
int getc(FILE *stream)
```

Returns the character read if success, EOF if error or end of stream.

Worth repeating: EOF is not a character in the file. It is a special return value to indicate "none".

Worth repeating: EOF fits in int but not char. Correct usage of getchar and getc involves:

1. Store return value in int variable, not char.
2. Compare to EOF.
3. If not EOF, safe to down-convert to char.

Demo: count.c

String I/O

```
int fputs(const char *string, FILE *stream)
```

Returns EOF if error. Also difference from 'puts': 'fputs' does not add newline at the end, 'puts' does.

```
char *fgets(char *dest, int n, FILE *stream)
```

Reads at most $n - 1$ characters or until (and including) newline.

Returns NULL if error or end of stream, dest if success.

Exercise 1: Why does it need you to provide n?
Exercise 2: Why $n - 1$ characters read, not n?

Arbitrary Data I/O

```
size_t fread(void *dest, size_t s, size_t n,
             FILE *stream)
size_t fwrite(const void *data, size_t s, size_t n,
              FILE *stream)
```

Read/Write n items, each item s bytes (ask 'sizeof'); in-memory raw bytes used. Returns how many items read/written.

Some use cases:

- A whole array.
- Record (struct) (single or array of).
- Raw bytes (array of 'unsigned char' usually).

Cross-platform watchout: The same raw bytes can mean different values on different platforms. Check compatibility!

Seeking

(Again, these can be unsupported for non-real files.)

Seek (so next read/write happens at specified position):
`int fseek(FILE *stream, long i, int origin)`

origin	go to i bytes from:
SEEK_SET	beginning
SEEK_END	end
SEEK_CUR	current position

Returns -1 on error, 0 on success.
Restrictions apply to text files on systems where this matters.

Ask current position:
`long ftell(FILE *stream)`
Returns -1 if error.

Error vs End-of-Stream Disambiguation

If end-of-stream OR error,
`getc()` returns `EOF`, `fgets()` returns `NULL`, etc.
How to tell end-of-stream from read error:

`int feof(FILE *stream)`
Returns true if end-of-stream status.
Important: Does **not** predict next read; only **remembers** past read.
You must read before asking. Further complications next slide.

`int ferror(FILE *stream)`
Returns true if error status. Ditto.

`void clearerr(FILE *stream);`
Clears end-of-stream and error status.

Demo code: `eoferr.c`

feof(), ferror(): Correct Model

FILE has boolean vars end, fault.

initial, clearerr(), rewind(): reset both to false.
fseek(): reset end to false.

read, write:
  if read and end of file: set end to true
  if error: set fault to true
  (∴ e.g., if neither: **no change**)

feof(): return end
ferror(): return fault
That's it. Only a dumb getter.

Error Information

(#include <errno.h>)

Global(!) `int` variable `errno` stores error code of most recent error.

(Worse, shared by all of standard library and system calls. So, easily overwritten by most recent call.)

(Even worse, successful calls don't reset to 0, but can overwrite arbitrarily. Assume nothing.)

(Multi-threading note: Each thread has its own `errno`.)

Many possible values, see `man errno`. Examples:
File does not exist: `ENOENT`
No permission: `EACCESS`
No space left: `ENOSPC`

Standard Error Messages

perror prints error message for current errno to stderr:
void perror(const char *prefix)

strerror and strerror_r gives error message:
char *strerror(int errnum)
int strerror_r(int errnum, char *buf, size_t buflen)

Exercise: Try to open a non-existent file for mode "r", check for
error (should get NULL), use perror to see the error message.

Buffering

C stream I/O delays writing: accumulates data in buffer until large chunk, then requests kernel to write that chunk.

And reads ahead: requests kernel to read a large chunk into buffer, then serves your puny 'getc' etc. from buffer.

Why: Huge overhead per kernel request (system call, "syscall") regardless of data size. 1 syscall for 1000 bytes beats 1000 syscalls for 1 byte each.

But can be disabled/reconfigured if your application needs.

(Story for another day: Kernel does its own buffering too—huge overhead talking to hardware. . .)

Buffer Operations

`int fflush(FILE *stream)`
Output stream: Write the buffer now!
Input stream: Clear buffer (will re-read, no real data loss).
Returns 0 if success, `EOF` if error.

`int setvbuf(FILE *stream, char *buf, int mode, size_t n)`
Configure buffering:

mode	meaning
_IOFBF	full buffering
_IOLBF	line buffering, write when newline
_IONBF	no buffering

Buffer space will be at `buf`; but if `NULL`: `setvbuf` allocates its own.

Demo: `buf.c` shows slowness of no-buffering.

Default Buffering of stdin, stdout, stderr

	if terminal	if file or pipe
stdout	line	full
stderr	no	no
stdin	complicated	full

stdin on terminal: C requests a chunk, **but** kernel undercuts it with line buffering:

Until you press enter, kernel puts C on hold. This gives you a chance to correct mistakes before committing.

After you press enter, kernel gives the whole chunk to C.