ESIGELEC
ÉCOLE D'INGÉNIEURS-ES GÉNÉRALISTES

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

C Programming



File handling

Ángel Perles
angel.perles@upv.es
R02

Contents

Objectives
Concepts
Opening and closing
Text files
Binary files
Random access
Robust file code
Activities and summary

Objectives

- Explain what a file, a stream and the position indicator are
- Open and close files safely with fopen() and fclose()
- Write and read text files: fprintf(), fgets() + sscanf()
- Write and read binary files: fwrite(), fread()
- Move inside a file: fseek(), ftell(), rewind()
- Loop on the return value of the read function, then check ferror()
- Explain why a binary file is not portable

From Java to C

	Java	C
Handle	FileReader, BufferedWriter, … objects	One type for everything: `FILE *`
Open fails	IOException is thrown	`fopen()` returns `NULL`, sets `errno`
Close	try-with-resources closes for you	`fclose()`: your job, on every path
Read loop	`read()` returns -1 at the end	Same idea: test the return value
Binary format	DataOutputStream: fixed, big-endian	`fwrite()`: the raw memory image

Same ideas. What changes: no exceptions, no automatic close and you choose the byte layout.

CONCEPTS

FILE

A file is an array of bytes

- File: a **named sequence of bytes** in non-volatile storage
- The file does not know what the bytes mean: **your program** gives them a structure
- The extension (`.csv`, `.bin`, `.exe`) is only a convention
- It can grow; the OS keeps the metadata: size, dates, permissions

THE FILE hello.txt (6 BYTES)

Offset	0	1	2	3	4	5
Byte	68	65	6C	6C	6F	0A
Char	h	e	l	l	o	\n

The same bytes could be read as text, as numbers or as an image: only the program decides.

Streams: FILE *

- `fopen()` returns a `FILE *`: a handle to a **stream**
- The stream keeps a buffer, the position indicator and the EOF and error flags
- Never look inside `FILE`: use only `<stdio.h>` functions
- Three streams are open when `main()` starts: `stdin`, `stdout`, `stderr`

YOU ALREADY USE STREAMS

```
printf("x = %d\n", x);
/* is the same as */
fprintf(stdout, "x = %d\n", x);

fprintf(stderr, "Error: no sensor\n");
```

REDIRECTION: TWO DIFFERENT FILES

```
C:\...> build\app.exe > out.txt 2> err.txt
```

The position indicator

- Each open stream has its own **position indicator**
- Opening sets it to the start (to the end in mode `"a"`)
- Every read or write starts there and moves it forward: **sequential access**
- `fseek()` moves it anywhere: **random access** (later)

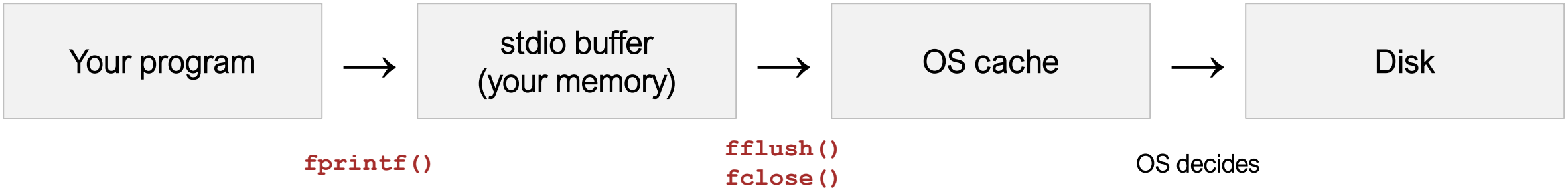
WRITING "hello" IN TWO CALLS

```
fputs("he", p_file);   /* 0 → 2 */
fputs("llo", p_file);  /* 2 → 5 */
```

Offset	0	1	2	3	4	5
Byte	h	e	l	l	o	↑

After the second call the indicator is at 5: the next write goes there. Checked with **`ftell()`**.

Buffering: why fclose() matters



- `fprintf()` writes to a buffer in memory, **not to the disk**
- `fflush()` and `fclose()` pass the buffer to the operating system
- Program crash or power cut before that: **the data is lost**
- Data logger: `fflush()` after each record (cost: speed and flash wear)

- On embedded systems with an SD card, the same happens with the FAT library buffers. Other file systems such as littleFS solve these issues.

OPENING AND CLOSING



fopen() and fclose()

```
#include <stdio.h>

FILE *fopen(const char *filename, const char *mode);
int   fclose(FILE *stream);
```

- `fopen()` returns `NULL` on failure and sets `errno`
- Typical failures: the file does not exist, no permission, wrong path
- `fclose()` returns 0, or `EOF` if the final flush failed
- After `fclose()` the pointer is no longer valid: do not use it again

Opening modes

Mode	Opens for	If the file exists	If it does not exist
`"r"`	Read	Read from the start	Error: `NULL`
`"w"`	Write	**Truncated to 0 bytes!**	Created
`"a"`	Append	Every write goes to the end	Created
`"r+"`	Read and write	From the start	Error: `NULL`
`"w+"`	Read and write	**Truncated to 0 bytes!**	Created
`"a+"`	Read and append	Reads anywhere, writes at the end	Created

Add b for binary files: **`"rb"`**, **`"wb"`**, **`"ab"`**, **`"r+b"`**… On Windows it matters (next slide).
Mode **`"x"`** (fail if the file exists) is C11, not C99.

Text mode and binary mode

- In memory `'\n'` is one byte on every platform
- Text mode on Windows: `'\n'` is written as CR LF and read back as `'\n'`
- Binary mode: the bytes go to the file unchanged
- Linux, macOS: no difference, but write the `b` anyway
- Rule: text files `"r"`, `"w"`; anything else `"rb"`, `"wb"`

THE SAME CALL

```
fputs("a\nb", p_file);
```

BYTES IN THE FILE ON WINDOWS

	0	1	2	3
"w" (text)	61	0D	0A	62
"wb" (binary)	61	0A	62	

A binary file opened with **`"w"`**: every byte 0A gets a 0D in front. The file is corrupted.

Check every fopen()

CRASHES WHEN THE FILE IS MISSING

```
FILE *p_file = fopen("config.txt", "r");

fgets(line, sizeof(line), p_file);
/* NULL if the file is missing: crash */
```

COURSE PATTERN

```
FILE *p_file = fopen(p_name, "r");

if (NULL == p_file)
{
    perror(p_name);
    return -1;
}
```

OUTPUT OF perror()

```
config.txt: No such file or directory
```

- `perror(s)` prints `s`, a colon and the reason taken from `errno`
- Use it right after the failing call: the next library call may change `errno`
- Errors go to `stderr`

Paths and the working directory

COMPILES, NEVER OPENS

```
fopen("C:\temp\data.csv", "r");
/* \t is a TAB: no warning */
```

WORKS ON WINDOWS

```
fopen("C:/temp/data.csv", "r");
fopen("C:\\temp\\data.csv", "r");
fopen("data.csv", "r");  /* relative */
```

- A backslash starts an **escape sequence** in a string literal
- Forward slashes work on Windows too
- A relative path starts at the **working directory**, not where the `.exe` is
- `mingw32-make run` runs from the project folder: the files appear there

TEXT FILES

fgets

Text files and binary files

- Let's start comparing the idea of "text" and "binary" content in a file

TEXT: "PI=3.14 RADIO=256" → 17 BYTES

50	49	3D	33	2E	31	34	20	52	41	44	49	4F	3D	32	35	36
P	I	=	3	.	1	4	␣	R	A	D	I	O	=	2	5	6

BINARY: float 3.14F + int16_t 256 → 6 BYTES (little-endian)

C3	F5	48	40	00	01
float				int16_t	

Text: readable, portable, bigger, needs conversion (`fprintf()`, `sscanf()`)

Binary: compact, fast, exact, not readable, not portable

Writing text: fprintf()

```
FILE *p_file = fopen("history.csv", "w");

if (NULL == p_file)
{
    perror("history.csv");
    return -1;
}
fprintf(p_file, "cycle;tank;level_m\n");

for (uint32_t cycle = 0U; cycle < 3U; cycle++)
{
    fprintf(p_file, "%u;%u;%.2f\n",
                    cycle, 1U, level_m[cycle]);
}
fclose(p_file);
```

history.csv

```
cycle;tank;level_m
0;1;1.50
1;1;2.50
2;1;3.50
```

- `fprintf()` = `printf()` + a stream
- Also `fputs()`, `fputc()`
- All the specifiers work, also `PRIu16`…
- Opens in LibreOffice Calc or Excel

Reading text: the loop pattern

```
char line[64];
unsigned int cycle;
unsigned int tank;
double level_m;

while (NULL != fgets(line, sizeof(line), p_file))
{
    if (3 == sscanf(line, "%u;%u;%lf",
                    &cycle, &tank, &level_m))
    {
        /* use the values */
    }
    else
    {
        fprintf(stderr, "Bad line: %s", line);
    }
}
```

- The read function is the loop condition
- `fgets()` returns `NULL` at the end of the file or on error
- It keeps the `'\n'` and never writes more than `size`
- `sscanf()` returns how many fields it converted: check it
- Header line: one `fgets()` before the loop

Reading byte by byte: fgetc()

COUNT THE LINES

```
int c = fgetc(p_file);   /* int! */
uint32_t num_lines = 0U;

while (EOF != c)
{
    if ('\n' == c)
    {
        num_lines++;
    }
    c = fgetc(p_file);
}
```

- `fgetc()` returns the byte (0…255) as an `int`, or `EOF` (-1)
- Store it in an `int`, not in a `char`
- `char c`: the byte `FF` looks like `EOF`, or `EOF` is never seen (unsigned `char`, ARM) → endless loop
- Like Java: `read()` returns an `int` too

End of file or error?

```
while (NULL != fgets(line, sizeof(line), p_file))
{
    /* ... */
}

if (0 != ferror(p_file))
{
    perror("history.csv");  /* read error */
}
/* else: normal end of file */
```

- A read function fails for two reasons: **end of file** or **error**
- After the loop, `ferror()` tells which one
- `feof()` only reports that a read **has already** failed at the end
- Rule: loop on the return value, ask `ferror()` after the loop

BINARY FILES

fread

fwrite() and fread()

```
size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream);
size_t fread(void *ptr, size_t size, size_t count, FILE *stream);
```

- Copy `count` elements of `size` bytes between memory and the file
- **No conversion**: the bytes in memory go to the file as they are
- Return the number of **complete elements**, not bytes
- Less than `count`: end of file or error (`ferror()`)
- Open with `"wb"` and `"rb"`

Writing and reading records

WRITE THE WHOLE ARRAY ("wb")

```
typedef struct
{
    uint8_t cycle;
    uint8_t tank;
    double  level_m;
} record_t;

record_t history[3] = {0};
/* ... fill history ... */

size_t n = fwrite(history, sizeof(record_t),
                  3U, p_file);
if (3U != n)
{
    perror("history.bin");
}
```

READ ONE RECORD AT A TIME ("rb")

```
record_t rec;

while (1U == fread(&rec, sizeof(rec),
                   1U, p_file))
{
    printf("%u %.2f\n",
           rec.cycle, rec.level_m);
}
```

Same pattern as with text: the read function is the loop condition.

Binary files are not portable

history.bin WITHOUT {0}: TWO RECORDS

```
$ od -A d -t x1 history.bin
0000000 00 01 00 00 00 00 00 00 00 00 00 00 00 00 f8 3f
0000016 01 01 40 2f fe 7f 00 00 00 00 00 00 00 00 04 40
```

- `sizeof(record_t)` is 16, the data is 10: **6 padding bytes** (alignment of `double`)
- Padding keeps whatever was in memory → initialise with `{0}`
- **Byte order**: x86 and most ARM are little-endian; Java `DataOutputStream` is big-endian
- **Type sizes**: `int`, `long` depend on the platform → `<stdint.h>`
- Same program, same machine: fine. To exchange files: define the format byte by byte, or use text

RANDOM ACCESS

fseek

Moving the position indicator

```
int  fseek(FILE *stream, long offset,
           int origin);
long ftell(FILE *stream);
void rewind(FILE *stream);
```

origin	offset counted from
`SEEK_SET`	The start of the file
`SEEK_CUR`	The current position
`SEEK_END`	The end of the file

- `fseek()` returns 0 if OK; `ftell()` returns the position, or -1L on error
- Use them in **binary mode**: in text mode on Windows the offsets are not byte counts
- Switching from writing to reading (`"r+"`, `"w+"`): call `fseek()` or `fflush()` in between

Example: read record N

```
int read_record(FILE *p_file, uint32_t index, record_t *p_rec)
{
    long offset = (long)index * (long)sizeof(record_t);

    if (0 != fseek(p_file, offset, SEEK_SET))
    {
        return -1;
    }
    if (1U != fread(p_rec, sizeof(record_t), 1U, p_file))
    {
        return -1;   /* past the end or error */
    }
    return 0;
}
```

- Fixed-size records: position = index × `sizeof`
- No need to read records 0 … N-1
- `fseek()` past the end is not an error: the `fread()` fails

HOW MANY RECORDS?

```
fseek(p_file, 0L, SEEK_END);
long num_records = ftell(p_file) / (long)sizeof(record_t);
rewind(p_file);
```

Robust file code

```
int save_history(const char *p_name,
                 const double *p_data, uint8_t n)
{
    int result = 0;
    FILE *p_file = fopen(p_name, "w");

    if (NULL == p_file)
    {
        perror(p_name);
        return -1;
    }
    for (uint8_t i = 0U; (i < n) && (0 == result); i++)
    {
        if (0 > fprintf(p_file, "%u;%.2f\n", i, p_data[i]))
        {
            result = -1;
        }
    }
    if (0 != fclose(p_file))
    {
        result = -1;
    }
    return result;
}
```

- Check `fopen()`: `NULL` → `perror()`, stop
- Loop on the return value of the read function
- Check writes: `fwrite()` count, `fprintf()` < 0
- `fclose()` on every path; check it when writing
- `fflush()` if the data must survive a crash
- Text for people and other programs; binary only with a defined format

Activity: find the four bugs

```
void log_level(double level_m)
{
    FILE *p_file = fopen("C:\logs\level.csv", "w");

    fprintf(p_file, "%.2f\n", level_m);
}
```

- The function is called once per cycle to log the tank level
- Find four bugs. Which one does gcc report?
- Rewrite it with the course pattern

Activity: how big is the file?

```
int16_t values[100];      /* all = 1000 */

/* A: text file, opened with "w" */
for (uint8_t i = 0U; i < 100U; i++)
{
    fprintf(p_a, "%" PRId16 "\n", values[i]);
}
/* B: binary file, opened with "wb" */
fwrite(values, sizeof(int16_t), 100U, p_b);
/* C: binary file, opened with "wb" */
fwrite(values, sizeof(values), 100U, p_c);
```

Size of A on Linux? On Windows? Size of B? What is wrong with C?
Check your answers with `ls -l` and `od -A d -t x1`.

Summary: Java → C

Topic	Remember in C
Handle	`FILE *`; `stdin`, `stdout`, `stderr` are streams too
Open	`fopen()`: check `NULL`, `perror()`; `b` for binary files
Close	`fclose()` on every path; data reaches the disk only then (or `fflush()`)
Text	`fprintf()` to write; `fgets()` + `sscanf()` to read
Binary	`fwrite()`, `fread()` copy memory as it is: padding, byte order, sizes
Loops	Loop on the return value of the read function; `ferror()` after it
Random access	`fseek()`, `ftell()`, `rewind()` in binary mode

Next lab: the history of the plant goes to a CSV file.

ESIGELEC

ÉCOLE D'INGÉNIEURS-ES GÉNÉRALISTES

UNIVERSITAT POLITÈCNICA DE VALÈNCIA