Master IOT & AI4CI – Bash & C

Lab Handout: File Access in C (Low-Level vs. High-Level)

Sami Taktak
sami.taktak@cnam.fr

2025

Learning Objectives

By the end of this lab, you will be able to:

1. Use both low-level and high-level C file I/O functions.
2. Understand the differences between `open/read/write` and `fopen/fread/fwrite`.
3. Implement a simplified version of the Unix `cp` tool using both approaches

Background

C provides two main ways of accessing files:

Low-level I/O (system calls)

- Functions: `open`, `read`, `write`, `close`
- Work with *file descriptors* (integers)

High-level I/O (stdio library)

- Functions: `fopen`, `fread`, `fwrite`, `fclose`
- Work with *FILE** pointers

Part 1 : Low-level File I/O

We first use the POSIX system calls for file I/O. These operate directly with file descriptors (integers).

open

```
int open(const char *pathname, int flags, mode_t mode);
```

- `pathname`: the file to open
- `flags`: for this lab we use:

  - `O_WRONLY` – open for writing only
  - `O_RDONLY` – open for reading only
  - `O_CREAT` – create the file if it does not exist
  - `O_TRUNC` – truncate the file to length 0 if it exists

- `mode`: file permissions (used only when creating a new file, e.g. `0644`)
- Returns a file descriptor (non-negative integer), or -1 on error

write

```
ssize_t write(int fd, const void *buf, size_t count);
```

- `fd`: file descriptor obtained from `open`
- `buf`: pointer to memory containing the data
- `count`: number of bytes to write
- Returns number of bytes written, or -1 on error

read

```
ssize_t read(int fd, void *buf, size_t count);
```

- `fd`: file descriptor obtained from `open`
- `buf`: pointer to memory where to store the data
- `count`: maximum bytes to read
- Returns: number of bytes read, 0 at EOF, -1 on error

a) Writing to a file (low-level)

Task: Complete the program so that it writes `"Hello, low-level world!\n"` into `lowlevel.txt` using `open` and `write`.

```c
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main() {
    // 1. Open (or create) the file "lowlevel.txt" for writing
    int fd = /* TODO: call open(...) */;

    if (fd == -1) {
        perror("open");
        return 1;
    }

    const char *msg = "Hello, low-level world!\n";

    // 2. Write msg into the file
    /* TODO: call write(...) */

    // 3. Close the file
    /* TODO: call close(...) */

    return 0;
}
```

b) Reading from a file (low-level)

Task: Complete the program so that it reads from `lowlevel.txt` and prints its content using `open` and `read`.

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    char buffer[128];

    // 1. Open "lowlevel.txt" for reading
    int fd = /* TODO: call open(...) */;
    if (fd == -1) {
        perror("open");
        return 1;
    }

    // 2. Read data into buffer
    int bytes = /* TODO: call read(...) */;

    // 3. Null-terminate and print
    if (bytes > 0) {
        buffer[bytes] = '\0';
        printf("Read: %s", buffer);
    }

    // 4. Close the file
    /* TODO: call close(...) */

    return 0;
}
```

Part 2: High-level File I/O

C also provides buffered file I/O through the standard library, using `FILE *` pointers.

fopen

```c
FILE *fopen(const char *path, const char *mode);
```

- `filename`: name of the file
- `mode`: for this lab we use `"w"` (write, truncate or create), `"r"` (read), `"rb"` (read binary), `"wb"` (write binary)
- Returns a pointer to a `FILE` object, or `NULL` on error

fwrite

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

- Writes up to `nmemb` objects of size `size` from `ptr` to `stream`
- Returns the number of objects successfully written

fread

`size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);`

- Reads up to `nmemb` objects of size `size` into `ptr`
- Returns the number of objects successfully read

a) Writing to a file (high-level)

Task: Complete the program so that it writes `"Hello, high-level world!\n"` into `highlevel.txt` using `fopen` and `fwrite`.

```c
#include <stdio.h>

int main() {
    // 1. Open file "highlevel.txt" for writing
    FILE *fp = /* TODO: call fopen(...) */;
    if (!fp) {
        perror("fopen");
        return 1;
    }

    const char *msg = "Hello, high-level world!\n";

    // 2. Write the message into the file
    /* TODO: call fwrite(...) */

    // 3. Close the file
    /* TODO: call fclose(...) */

    return 0;
}
```

b) Reading from a file (high-level)

Task: Complete the program so that it reads from `highlevel.txt` and prints its content using `fopen` and `fread`.

```c
#include <stdio.h>

#define BUFSIZE 256

int main() {
    char buffer[BUFSIZE];

    // 1. Open "highlevel.txt" for reading
    FILE *fp = /* TODO: call fopen(...) */;
    if (!fp) {
        perror("fopen");
        return 1;
    }

    // 2. Read data into buffer
    size_t bytes = /* TODO: call fread(...) */;

    // 3. Null-terminate and print
    buffer[bytes] = '\0';
    printf("Read:␣%s", buffer);

    // 4. Close the file
    /* TODO: call fclose(...) */

    return 0;
}
```

Part 3: Mini Project — Reimplementing `cp`

a) `cp` with low-level I/O

Task: Complete the missing parts to implement a simple `cp` program using low-level functions.

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

#define BUFSIZE 256

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage:␣%s␣source␣dest\n", argv[0]);
        return 1;
    }

    // 1. Open source file for reading
    int in_fd = /* TODO */;

    // 2. Open destination file for writing (create or truncate)
    int out_fd = /* TODO */;

    char buffer[BUFSIZE];
    ssize_t n;

    // 3. Loop: read into buffer, then write buffer into dest
    while (/* TODO: call read(...) */) {
        /* TODO: call write(...) */
    }

    // 4. Close both files
    /* TODO */

    return 0;
}
```

b) cp with high-level I/O

Task: Complete the missing parts to implement a simple cp program using high-level functions.

```c
#include <stdio.h>

#define BUF_SIZE 256

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s source dest\n", argv[0]);
        return 1;
    }

    // 1. Open files
    FILE *in = /* TODO */;
    FILE *out = /* TODO */;

    char buffer[BUFSIZE];
    size_t n;

    // 2. Loop: fread into buffer, fwrite buffer into dest
    while (/* TODO: call fread(...) */) {
        /* TODO: call fwrite(...) */
    }

    // 3. Close both files
    /* TODO */

    return 0;
}
```

c) Compare low-level and high-level implementation of cp

Task: Compare the performances between the to implementation of `cp`. You can use the `time` command which allow to monitor how long a program has been running. Try to play with different buffer size.

```
$ time cp src_file dest_file
```

To generate huge file to test `cp`, you can use `dd` tool:

```
$ dd if=/dev/urandom of=bigfile.raw bs=1M count=1024 status=progress
```