Gorgo C for C++ Programmers

April 11, 2026

11. Low-Level I/O

The `<stdio.h>` functions you learned in the previous chapter are built on top of a lower-level I/O interface provided by the operating system. These system calls — `read`, `write`, `open`, and `close` — work directly with **file descriptors** rather than `FILE *` pointers. You will encounter them in systems programming, and understanding them helps you see what `stdio` is actually doing under the hood.

File Descriptors

A file descriptor is a small non-negative integer that the operating system uses to identify an open file (or pipe, socket, device, etc.). When your program starts, three file descriptors are already open:

File Descriptor	POSIX Name	Purpose
0	`STDIN_FILENO`	Standard input
1	`STDOUT_FILENO`	Standard output
2	`STDERR_FILENO`	Standard error

These constants are defined in `<unistd.h>`. They correspond to `stdin`, `stdout`, and `stderr` from `<stdio.h>`, but at a lower level.

`read` and `write`

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

The `read` and `write` system calls transfer raw bytes between a file descriptor and a buffer:

```c
#include <unistd.h>

// write(fd, buffer, count) — returns bytes written
write(1, "Blue Monday\n", 12);   // write 12 bytes to stdout

// read(fd, buffer, count) — returns bytes read
char buf[100];
ssize_t n = read(0, buf, sizeof(buf));  // read from stdin
write(1, buf, n);                        // echo it back
```

`read` returns the number of bytes actually read (which may be less than requested), 0 at end of file, or -1 on error. `write` returns the number of bytes actually written, or -1 on error.

Unlike `printf` and `scanf`, these functions perform no formatting — they transfer raw bytes. There are no format specifiers, no newline handling, no buffering.

`open` and `close`

```c
int open(const char *path, int flags, ... /* mode_t mode */);
int close(int fd);
```

To open a file at the system call level, use `open` from `<fcntl.h>`:

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

int fd = open("data.txt", O_RDONLY);
if (fd == -1) {
    write(2, "Cannot open file\n", 17);
    return 1;
}

char buf[256];
ssize_t n = read(fd, buf, sizeof(buf));
write(1, buf, n);

close(fd);
```

The second argument to `open` is a set of flags combined with bitwise OR:

Flag	Purpose
`O_RDONLY`	Open for reading only
`O_WRONLY`	Open for writing only
`O_RDWR`	Open for reading and writing
`O_CREAT`	Create the file if it does not exist
`O_TRUNC`	Truncate the file to zero length
`O_APPEND`	Append to the file

To **create a new file** (or truncate an existing one), combine flags:

```c
int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
```

The third argument (`0644`) is the **file permissions** — only used with `O_CREAT`. The value `0644` means the owner can read and write, and everyone else can only read.

There is also `creat`, which is equivalent to `open` with `O_WRONLY | O_CREAT | O_TRUNC`:

```c
int creat(const char *path, mode_t mode);
```

```c
int fd = creat("output.txt", 0644);
// same as: open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644)
```

Wut: Yes, it is `creat` with no `e`. Ken Thompson was once asked what he would do differently if he were redesigning Unix. His answer: "I'd spell creat with an e."

Seeking: `lseek`

```c
off_t lseek(int fd, off_t offset, int whence);
```

`lseek` repositions the file offset for an open file descriptor:

```c
#include <unistd.h>

lseek(fd, 0, SEEK_SET);     // go to beginning
lseek(fd, 0, SEEK_END);     // go to end
lseek(fd, 100, SEEK_SET);   // go to byte 100

off_t pos = lseek(fd, 0, SEEK_CUR);  // get current position (no move)
```

The three `SEEK_` constants control where the offset is relative to:

Constant	Meaning
`SEEK_SET`	Relative to the beginning of the file
`SEEK_CUR`	Relative to the current position
`SEEK_END`	Relative to the end of the file

`lseek` returns the new offset from the beginning of the file, or -1 on error.

Tip: In `<stdio.h>`, the equivalent functions are `fseek` and `ftell`. The low-level `lseek` combines both: calling `lseek(fd, 0, SEEK_CUR)` returns the current position without moving, just like `ftell`.

`pread` and `pwrite`

```c
ssize_t pread(int fd, void *buf, size_t count, off_t offset);
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
```

`pread` and `pwrite` are like `read` and `write` but take an explicit offset instead of using (or modifying) the file's current position:

```c
// Read 100 bytes starting at offset 200, without changing the file position
ssize_t n = pread(fd, buf, 100, 200);

// Write 50 bytes at offset 0, without changing the file position
pwrite(fd, data, 50, 0);
```

These are useful in multi-threaded programs where multiple threads share a file descriptor — since they do not modify the file position, there is no race condition.

Try It: Low-Level I/O Starter

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

int main(void) {
    // write to stdout using file descriptor 1
    const char *msg = "Low-level I/O starter\n";
    write(STDOUT_FILENO, msg, strlen(msg));

    // open, write, close
    int fd = open("/tmp/tryit_lowio.txt",
                  O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd == -1) {
        write(STDERR_FILENO, "open failed\n", 12);
        return 1;
    }

    const char *lines[] = {"Come As You Are\n", "Year: 1992\n"};
    for (int i = 0; i < 2; i++)
        write(fd, lines[i], strlen(lines[i]));
    close(fd);

    // open, read, print
    fd = open("/tmp/tryit_lowio.txt", O_RDONLY);
    char buf[256];
    ssize_t n = read(fd, buf, sizeof(buf));
    write(STDOUT_FILENO, buf, n);

    // lseek: go back to start and read again
    lseek(fd, 0, SEEK_SET);
    n = read(fd, buf, sizeof(buf));
    printf("Read %zd bytes on second pass\n", n);

    close(fd);
    return 0;
}
```

Key Points

- File descriptors are small integers: 0 is stdin, 1 is stdout, 2 is stderr.
- `read` and `write` transfer raw bytes — no formatting, no buffering.
- `open` returns a file descriptor; `fopen` returns a `FILE *`. They are different levels of abstraction.
- Use `O_CREAT` with `open` to create files. Always provide a permissions argument when using `O_CREAT`.
- `lseek` repositions the read/write offset. Use `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`.
- `pread` and `pwrite` read/write at a specific offset without changing the file position.

Exercises

1. **Think about it:** Why would you use low-level `read`/`write` instead of `fprintf`/`fscanf`? When would `stdio` be the better choice?
2. **What does this print?**

```
write(1, "ABC", 3);
write(1, "DEF\n", 4);
```

3. **Calculation:** If `read(fd, buf, 1024)` returns 512, what does that tell you? Does it mean there was an error?
4. **Where is the bug?**

```
int fd = open("newfile.txt", O_WRONLY | O_CREAT);
write(fd, "Hello\n", 6);
close(fd);
```

5. **Think about it:** Explain the difference between `lseek(fd, 0, SEEK_END)` and `lseek(fd, -1, SEEK_END)`. What does each return?
6. **Write a program** that uses low-level I/O (`open`, `read`, `write`, `close`) to copy the contents of one file to another. The source and destination filenames should be taken from `argv`.