A Guide to File Handling in C

1. Introduction to File Handling

In C programming, a **file** is a place on a storage medium (like a hard disk, SSD, or USB drive) where data is stored permanently. File handling allows us to create, read, write, and update these files directly from our C programs. This is essential for any application that needs to store data beyond a single run, such as saving user progress, configuration settings, or processing large datasets.

Types of Files

There are two primary types of files you'll work with:

- **Text Files:** These files store data as a sequence of characters (like the letters, numbers, and symbols you're reading now). They are human-readable and can be opened by any standard text editor (like Notepad or VS Code). In C, lines in a text file are typically terminated by a newline character (\n).
- **Binary Files:** These files store data in the same format the computer uses in memory—raw bytes. This includes integers, floats, and complex data structures. Binary files are not human-readable but are much more efficient for storing and retrieving data because there's no need for conversion between text and internal formats.

2. The Basics: FILE Pointer and Core Functions

All file operations in C are performed using a special pointer called the FILE pointer. It acts as a communication link between your program and the file on the disk.

Key Functions

Function	Description
fopen()	Opens a file and returns a FILE pointer to it.
fclose()	Closes a file, flushing any buffered data to the disk.
fgetc()	Reads a single character from a file.
fputc()	Writes a single character to a file.
fgets()	Reads a line of text (a string) from a file.
fputs()	Writes a string to a file.
fprintf()	Writes formatted data to a file (like printf).
fscanf()	Reads formatted data from a file (like scanf).
fread()	Reads a block of binary data from a file.
fwrite()	Writes a block of binary data to a file.
fseek()	Moves the file pointer to a specific position in the file.
ftell()	Returns the current position of the file pointer.
rewind()	Resets the file pointer to the beginning of the file.

Opening a File: fopen()

Before you can work with a file, you must open it.
Syntax: FILE *fopen(const char *filename, const char *mode);
File Modes:

Mode	Meaning	Behavior
"r"	**Read**	Opens an existing text file for reading. Fails if the file doesn't exist.
"w"	**Write**	Creates a new text file for writing. **Deletes the file's contents** if it already exists.
"a"	**Append**	Opens a text file for writing at the end. Creates the file if it doesn't exist.
"r+"	**Read/Update**	Opens an existing text file for both reading and writing.
"w+"	**Write/Update**	Creates a new text file for both reading and writing. **Deletes contents** if it exists.
"a+"	**Append/Update**	Opens or creates a text file for reading and appending (writing at the end).

*To work with **binary files**, simply append a b to the mode string (e.g., "rb", "wb", "ab+").*

Always Check the FILE Pointer

fopen() returns NULL if it fails to open the file (e.g., file not found, no permissions). You **must** always check for this.

```
#include <stdio.h>

int main() {
    FILE *file_pointer;

    // Attempt to open a file that might not exist
    file_pointer = fopen("my_file.txt", "r");

    if (file_pointer == NULL) {
        // perror provides a descriptive error message
        perror("Error opening file");
        return 1; // Exit with an error code
    }

    printf("File opened successfully!\n");

    // ... do work with the file ...

    fclose(file_pointer); // Close the file when done
    return 0;
}
```

Closing a File: fclose()

It is crucial to close a file using fclose() when you are finished with it. This function writes any pending data from internal buffers to the disk and frees up system resources associated with the file.
Syntax: int fclose(FILE *stream);

3. Working with Text Files

Character by Character: fgetc() and fputc()

These are the most fundamental I/O functions.

- int fgetc(FILE *stream); reads the next character and returns it as an int. It returns EOF (End of File) when the end of the file is reached or an error occurs.
- int fputc(int character, FILE *stream); writes a character to the file.

Example: Copying a File

```
#include <stdio.h>

int main() {
    FILE *source_file, *dest_file;
    int ch;

    source_file = fopen("source.txt", "r");
    if (source_file == NULL) {
        perror("Error opening source file");
        return 1;
    }

    dest_file = fopen("destination.txt", "w");
    if (dest_file == NULL) {
        perror("Error opening destination file");
        fclose(source_file); // Clean up the already opened file
        return 1;
    }

    // Read from source and write to destination
    while ((ch = fgetc(source_file)) != EOF) {
        fputc(ch, dest_file);
    }

    printf("File copied successfully.\n");

    fclose(source_file);
    fclose(dest_file);

    return 0;
}
```

Line by Line: fgets() and fputs()

These functions are more efficient for reading and writing lines of text.

- char *fgets(char *str, int num_chars, FILE *stream); reads a line of text into the str buffer. It stops when num_chars - 1 characters are read, a newline is encountered, or EOF is reached. It's safer than gets() because it prevents buffer overflows.
- int fputs(const char *str, FILE *stream); writes a string to the file. It does *not* add a newline character automatically.

Example: Reading Student Names

```
#include <stdio.h>

int main() {
    FILE *fp;
    char student_name[100];

    // Writing names to a file
    fp = fopen("students.txt", "w");
    if (fp == NULL) return 1;
    fputs("Alice\n", fp);
    fputs("Bob\n", fp);
    fputs("Charlie\n", fp);
    fclose(fp);

    // Reading names from the file
    fp = fopen("students.txt", "r");
    if (fp == NULL) return 1;

    printf("List of Students:\n");
    while (fgets(student_name, sizeof(student_name), fp) != NULL) {
        printf(" - %s", student_name); // fgets includes the newline
    }
    fclose(fp);

    return 0;
}
```

Formatted I/O: fprintf() and fscanf()

These work just like printf and scanf but operate on files.

Example: Storing Records

```
#include <stdio.h>

int main() {
    FILE *fp;
    char name[50];
    int age;
    float score;

    // Write formatted data
    fp = fopen("records.txt", "w");
```

```
    if (fp == NULL) return 1;
    fprintf(fp, "David %d %.2f\n", 25, 88.5f);
    fprintf(fp, "Eve %d %.2f\n", 31, 92.0f);
    fclose(fp);

    // Read formatted data
    fp = fopen("records.txt", "r");
    if (fp == NULL) return 1;

    printf("Reading Records:\n");
    // fscanf returns the number of items successfully read
    while (fscanf(fp, "%s %d %f", name, &age, &score) == 3) {
        printf("Name: %s, Age: %d, Score: %.2f\n", name, age, score);
    }
    fclose(fp);

    return 0;
}
```

4. Working with Binary Files

Binary files are ideal for storing structured data, like C structs, because they map directly to memory.

Block I/O: fread() and fwrite()

These functions read and write whole blocks of data at once.
Syntax:

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

Parameters:

- ptr: Pointer to the data block to be read/written.
- size: Size (in bytes) of each individual element.
- count: Number of elements to read/write.
- stream: The FILE pointer.

Example: Saving and Loading a Struct

```
#include <stdio.h>

// Define a simple structure
struct Person {
    char name[50];
    int age;
    int id;
};

int main() {
    FILE *fp;
    struct Person p1 = {"John Doe", 30, 101};
    struct Person p2_read; // To store data read from file
```

```
    // --- Writing the struct to a binary file ---
    fp = fopen("person.bin", "wb"); // Note the "wb" for write binary
    if (fp == NULL) {
        perror("Error writing file");
        return 1;
    }

    // fwrite(&data, size_of_one_element, number_of_elements,
file_pointer)
    fwrite(&p1, sizeof(struct Person), 1, fp);
    fclose(fp);
    printf("Struct data written to person.bin\n");

    // --- Reading the struct from the binary file ---
    fp = fopen("person.bin", "rb"); // Note the "rb" for read binary
    if (fp == NULL) {
        perror("Error reading file");
        return 1;
    }

    // fread(&destination, size_of_one_element, number_of_elements,
file_pointer)
    fread(&p2_read, sizeof(struct Person), 1, fp);
    fclose(fp);

    printf("\nData read from file:\n");
    printf("Name: %s\n", p2_read.name);
    printf("Age: %d\n", p2_read.age);
    printf("ID: %d\n", p2_read.id);

    return 0;
}
```

5. Intermediate: Random File Access

Sometimes you need to jump to a specific part of a file without reading everything before it. This is called random access.

fseek(): Moving the Pointer

This is the primary function for file positioning.
Syntax: int fseek(FILE *stream, long int offset, int origin);

- offset: The number of bytes to move.
- origin: The starting point for the offset.
    - SEEK_SET: Beginning of the file.
    - SEEK_CUR: Current position of the file pointer.
    - SEEK_END: End of the file.

ftell(): Finding the Current Position

This function tells you the current value of the file pointer (as a long int).

Example: Reading the 2nd Record from a Binary File

Imagine records.bin contains several Person structs. We want to read only the second one.

```
#include <stdio.h>

struct Person {
    char name[50];
    int age;
    int id;
};

int main() {
    FILE *fp;
    struct Person person_to_read;
    int record_number = 2;

    fp = fopen("records.bin", "rb"); // Assume this file exists and
has data
    if (fp == NULL) {
        perror("Error");
        return 1;
    }

    // Move the file pointer to the beginning of the 2nd record.
    // The offset is (record_number - 1) * size_of_record.
    long offset = (record_number - 1) * sizeof(struct Person);
    fseek(fp, offset, SEEK_SET);

    // Now, read the record at the new position
    if (fread(&person_to_read, sizeof(struct Person), 1, fp) == 1) {
        printf("Record %d:\n", record_number);
        printf("Name: %s, Age: %d, ID: %d\n", person_to_read.name,
person_to_read.age, person_to_read.id);
    } else {
        printf("Could not read record %d.\n", record_number);
    }

    fclose(fp);
    return 0;
}
```