# FILE HANDLING

Sercan Külcü | 21.06.2023

# Contents

# 1 Working with files in C

In the world of programming, dealing with files is an essential skill. Whether you're reading data from a file or writing data to it, understanding how to work with files in C opens up a world of possibilities. In this chapter, we will explore the fundamentals of file handling in C and equip you with the knowledge and skills to manipulate files with ease. So, let's dive in and uncover the secrets of working with files in C!

Before we can perform any operations on a file, we need to open it. The fopen() function is used to open a file and returns a file pointer that we can use to refer to the file in our code. It takes two arguments: the file name (along with the path, if necessary) and the mode in which we want to open the file. The mode can be "r" for reading, "w" for writing, "a" for appending, and more.

Once we have finished working with a file, it is important to close it using the fclose() function. This step ensures that any pending data is written to the file and releases system resources associated with it. Failing to close a file can lead to data loss or resource leaks.

To read data from a file, we can use the fscanf() or fgets() functions. The fscanf() function allows us to read formatted data from a file, while the fgets() function reads a line of text. Both functions require the file pointer and a variable to store the read data.

It's important to handle file errors gracefully. After performing any file operation, we should always check the return value to ensure that the operation was successful. In case of an error, we can use the feof() and ferror() functions to determine whether the end of the file has been reached or if an error occurred during the file operation.

To write data to a file, we can use the fprintf() or fputs() functions. The fprintf() function allows us to write formatted data to a file, while the fputs() function writes a string. Similar to reading, these functions require the file pointer and the data to be written.

When writing to a file, we have the option to overwrite the existing content ("w" mode), append to the existing content ("a" mode), or create a new file if it doesn't exist ("w+" mode). Choosing the appropriate mode ensures that our data is written correctly.

In some cases, we may need to move the file position indicator to a specific location within a file. The fseek() function allows us to accomplish this task. It takes the file pointer, the offset from a reference position, and the reference position itself as arguments. The reference position can be the beginning of the file (SEEK_SET), the current position (SEEK_CUR), or the end of the file (SEEK_END).

Using ftell(), we can determine the current position of the file pointer. This information can be useful when we need to remember a specific position or implement custom file reading algorithms.

When working with files, we must be prepared to handle errors that may occur during file operations. The perror() function can be used to print a descriptive error message when an error occurs, helping us identify and resolve the issue quickly.

Before performing any file operation, we can check whether a file exists using the access() function. This function takes the file name and the desired mode of access as arguments. It returns zero if the file exists and we have the required permissions, or -1 otherwise.

# 2 File I/O using standard libraries

In this chapter, we will explore the exciting realm of File Input/Output (I/O) using the standard libraries in C. Working with files is a fundamental aspect of programming, allowing us to store and retrieve data efficiently. By mastering file I/O, you'll gain a powerful tool in your programming arsenal. So, let's dive in and learn how to manipulate files using the standard libraries in C!

File I/O is a critical aspect of programming, enabling us to interact with files and handle data persistence. It allows us to read data from files, write data to files, and perform various operations on them. Whether you're working with text files, binary files, or even device files, understanding file I/O is essential for many real-world applications.

C provides us with standard libraries that simplify file I/O operations. The most commonly used library is <stdio.h>, which offers a set of functions for file I/O, such as fopen(), fclose(), fread(), fwrite(), and more. These libraries provide an abstraction layer that simplifies file operations, making it easier for us to work with files.

To work with a file, we first need to open it. The fopen() function allows us to open a file, specifying the file name (including the path, if necessary) and the mode in which we want to open it. The mode can be "r" for reading, "w" for writing, "a" for appending, and more.

After we finish working with a file, it's important to close it using the fclose() function. This ensures that any pending data is written to the file and releases system resources associated with it. Neglecting to close files can lead to data loss or resource leaks.

To read data from a file, we have several functions at our disposal. The fscanf() function allows us to read formatted data from a file, similar to using scanf() for standard input. Alternatively, the fgets() function is useful for reading a line of text from a file. Additionally, the fread() function enables us to read binary data directly into memory.

While reading from files, it's important to handle errors gracefully. We should always check the return value of file read operations to ensure they succeed. We can

use the feof() and ferror() functions to determine if we've reached the end of the file or encountered an error during the read operation.

To write data to a file, we can use functions like fprintf(), fputs(), and fwrite(). fprintf() allows us to write formatted data to a file, similar to printf() for standard output. fputs() is useful for writing a string to a file, while fwrite() is ideal for writing binary data.

Similar to reading, it's crucial to handle errors when performing file write operations. The ferror() function helps us identify any errors that occurred during the write operation, allowing us to handle them appropriately.

In some cases, we may need to move the file position indicator to a specific location within a file. The fseek() function comes to our aid. It allows us to set the position indicator using a file pointer, an offset from a reference position, and the reference position itself (beginning, current, or end of the file).

Additionally, we can use ftell() to determine the current position of the file pointer. This information can be useful for remembering a specific position or implementing custom file reading algorithms.

# 3 Binary File I/O

In this chapter, we will explore the fascinating world of binary file I/O. While text files are suitable for storing human-readable data, binary files allow us to work with complex data structures and optimize storage efficiency. By mastering binary file I/O, you'll gain the ability to handle structured data and perform advanced file operations. So, let's dive in and unlock the power of binary file I/O in C!

Binary files store data in its raw, binary representation. Unlike text files, which use character encoding, binary files store data in its pure form, byte by byte. This characteristic makes binary files ideal for storing complex structures, such as arrays, structures, and even multimedia data.

The process of opening and closing binary files is similar to text files. We use the fopen() function to open a binary file, specifying the file name (including the path) and the mode. The mode can be "rb" for reading in binary mode, "wb" for writing in binary mode, and "ab" for appending in binary mode.

Closing a binary file is done using the fclose() function, just like with text files. It ensures that any pending data is written to the file and releases system resources associated with it.

To read binary data from a file, we use the fread() function. This function allows us to read a specified number of bytes directly into memory. It takes the destination buffer, the size of each element to be read, the number of elements to read, and the file pointer as parameters.

When reading binary data, it's crucial to handle errors and check for the successful completion of the fread() function. The return value of fread() indicates the number of elements successfully read, which allows us to verify if the desired amount of data was obtained.

To write binary data to a file, we utilize the fwrite() function. This function allows us to write a specified number of bytes from memory to a file. It takes the source buffer, the size of each element to be written, the number of elements to write, and the file pointer as parameters.

Similar to reading, it's important to handle errors and check the return value of the fwrite() function. This allows us to confirm if the desired amount of data was successfully written to the file.

When working with binary files, we may need to move the file position indicator to a specific location. The fseek() function, as discussed in Chapter 2, allows us to achieve this. By setting the file pointer, offset, and reference position, we can position the indicator anywhere within the binary file.

In addition, we can use the ftell() function to determine the current position of the file pointer in the binary file. This information is valuable for remembering positions or implementing custom file reading algorithms.

Working with binary files often involves dealing with structured data, such as arrays or structures. To ensure proper storage and retrieval, it's important to define a clear binary file structure. This includes specifying the size and layout of each data element, ensuring proper alignment, and accounting for any necessary padding.

# 4 Streams

Streams provide a powerful mechanism for input and output operations in C, allowing us to interact with external devices, files, and the standard input/output channels. So, let's embark on this exploration of streams together!

In C programming, a stream is an abstract representation of a sequence of data. Streams provide a uniform way to perform input and output operations, regardless of the specific device or medium involved. They serve as a bridge between the program and the external world, enabling data to flow in and out of the program.

Categorically, there are three types of streams:

- Standard Streams: C provides three standard streams that are automatically available in every program: stdin, stdout, and stderr. stdin represents the standard input stream, which is typically associated with the keyboard. stdout represents the standard output stream, which is associated with the console or terminal. stderr represents the standard error stream, used for error messages and diagnostics.
- File Streams: File streams allow us to perform input/output operations on files. By associating a file with a stream, we can read data from files or write data to files using stream-based functions.
- Custom Streams: It is also possible to create custom streams for specific devices or data sources. These streams can be created using library functions or by implementing your own stream-handling functions.

C provides a set of functions to perform input and output operations on streams. These functions allow you to read data from input streams, write data to output streams, and manipulate the stream's state. Some commonly used stream functions include:

- fopen(): Used to open a file and associate it with a file stream.
- fclose(): Closes a file stream.
- fprintf(): Writes formatted data to a file stream.
- fscanf(): Reads formatted data from a file stream.
- getchar(): Reads a character from the standard input stream.
- putchar(): Writes a character to the standard output stream.

- fgets(): Reads a line of text from a file stream.
- fputs(): Writes a string to a file stream.

These functions, along with many others, provide a wide range of capabilities for stream-based input and output operations.

Streams can be manipulated and controlled using various functions and techniques. Some common operations include:

- Positioning within a Stream: Functions like fseek() and ftell() allow you to move the file position indicator within a file stream, enabling random access to different parts of the file.
- Flushing Streams: The fflush() function is used to flush the data from output streams, ensuring that the data is written immediately. This is particularly useful when working with buffered output streams.
- Error Handling: Stream functions often return a value to indicate success or failure. Additionally, the ferror() and feof() functions help in error detection and end-of-file detection, respectively.

Proper error handling is crucial when working with streams to handle exceptional situations gracefully and provide meaningful feedback to the user.

One powerful feature of streams is the ability to redirect the standard streams. This allows you to change the default input and output channels of a program. By redirecting the standard input and output streams to files, you can automate input, capture output, or redirect error messages to a file.

You can redirect standard streams using command-line shell operators or by using the freopen() function within your program.