## Chapter 10 - Files

## Working with files

### Introduction

File handling is a fundamental aspect of programming, allowing us to store and retrieve data that can be used across multiple runs of a program. In C programming, file handling is achieved using the stdio.h library, which provides a variety of functions for reading from and writing to files. In this chapter, we will explore the basics of file handling in C, including opening and closing files, reading and writing data to files, and handling errors that may occur during file input/output.

### Text Files vs Binary Files

Before diving into working with files in C, it is important to understand the differences between text files and binary files. Text files are files that contain human-readable text, such as plain text documents or CSV files. These files are composed of a series of characters, usually encoded in ASCII or Unicode format. Binary files, on the other hand, are files that contain machine-readable data, such as executable files or images. These files are composed of a series of bytes that represent binary data and are not human-readable.

The choice between text files and binary files depends on the type of data being stored and the intended use of the file. Text files are usually used for storing data that can be easily edited by humans, such as configuration files or log files. On the other hand, binary files are typically used for storing data that is not easily editable by humans, such as multimedia files or database files.

One of the key differences between text files and binary files is the way that they are read and written. Text files can be read and written using standard functions such as fgets() and fprintf(), while binary files require more specialized functions such as fread() and fwrite().

### Opening and Closing Files

Once we have decided on the type of file we want to use, the next step is to open the file. In C, we use the fopen() function to open a file, which takes two parameters: the name of the file to be opened and the mode in which it is to be opened. The mode parameter specifies whether the file is to be opened for reading, writing, or both, as well as whether the file is to be created if it does not exist.

There are several modes in which a file can be opened, including:

* r: Open file for reading
* w: Open file for writing (overwrite if file already exists)
* a: Open file for appending (write to end of file)
* r+: Open file for both reading and writing
* w+: Open file for both reading and writing (overwrite if file already exists)
* a+: Open file for both reading and writing (write to end of file)

Once a file has been opened, we can perform operations on it, such as reading data from or writing data to the file. When we are finished working with a file, it is important to close it using the fclose() function. This ensures that any changes made to the file are saved and any resources used by the file are freed up.

## Reading from Text Files

Imagine we have a text file called "students.txt" containing a list of students' names, one name per line. We want to read the data from this file and display it on the screen. The following example demonstrates how to read data from a text file using fgets():

FILE \*file = fopen("students.txt", "r");

if (file != NULL)

{

char name[256];

printf("List of Students:\n");

while (fgets(name, sizeof(name), file) != NULL) {

printf("%s", name);

}

fclose(file);

}

## Writing to Text Files

Suppose we want to create a text file called "grades.txt" to store the grades of students in a course. We have a struct called "Student" containing the student's name and grade. The following example demonstrates how to write data to a text file using fprintf():

typedef struct

{

char name[50];

int grade;

} Student;

Student students[] =

{

{"Alice", 85},

{"Bob", 92},

{"Carol", 78}

};

FILE \*file = fopen("grades.txt", "w");

if (file != NULL)

{

for (int i = 0; i < 3; i++)

{

fprintf(file, "Name: %s, Grade: %d\n", students[i].name, students[i].grade);

}

fclose(file);

}

## Reading from Binary Files

Consider a binary file called "inventory.bin" containing a list of products in a store. Each product has a struct called "Product" containing a product ID and its stock quantity. The following example demonstrates how to read data from a binary file using fread():

typedef struct

{

int productID;

int stock;

} Product;

FILE \*file = fopen("inventory.bin", "rb");

if (file != NULL)

{

Product products[5];

size\_t read\_elements = fread(products, sizeof(Product), 5, file);

printf("Product Inventory:\n");

for (int i = 0; i < read\_elements; i++)

{

printf("Product ID: %d, Stock: %d\n", products[i].productID, products[i].stock);

}

fclose(file);

}

## Writing to Binary Files

Suppose we want to create a binary file called "sales.bin" to store the sales data of a store. Each sale record has a struct called "Sale" containing a transaction ID, product ID, and quantity sold. The following example demonstrates how to write data to a binary file using fwrite():

typedef struct {

int transactionID;

int productID;

int quantity;

} Sale;

Sale sales[] =

{

{1, 101, 5},

{2, 102, 3},

{3, 103, 8}

};

FILE \*file = fopen("sales.bin", "wb");

if (file != NULL)

{

fwrite(sales, sizeof(Sale), 3, file);

fclose(file);

}

## File Pointers and Positioning

File pointers are an important concept in C file handling, as they allow us to keep track of our current position within a file. A file pointer is a variable that holds the position of the next byte to be read or written in a file. By manipulating the file pointer, we can control where in the file we are reading or writing data.

The ftell() function is used to determine the current position of the file pointer within a file. It takes the file pointer as an argument and returns the current position in bytes from the beginning of the file.

The fseek() function is used to move the file pointer to a specific position within a file. It takes three arguments: the file pointer, the offset (number of bytes to move the pointer), and the origin (starting position for the pointer movement). The origin can be one of the following:

* SEEK\_SET: Beginning of file
* SEEK\_CUR: Current position of file pointer
* SEEK\_END: End of file

For example, to move the file pointer to the beginning of the file, we would use:

fseek(file\_pointer, 0, SEEK\_SET);

The rewind() function is a simpler way to move the file pointer to the beginning of the file. It takes the file pointer as an argument and moves the pointer to the beginning of the file, equivalent to using fseek() with an offset of 0 and SEEK\_SET as the origin.

When working with files, it is important to be aware of common errors that can occur. One common error is failing to check if a file was opened successfully before attempting to read or write to it. This can lead to unexpected behavior or even crashes. Another common error is not properly closing a file when we are finished using it. This can lead to resource leaks and cause issues if we attempt to open the file again later.

## Finding the size of a file

Determining the size of a file can be useful in many scenarios, such as when reading or writing binary data to a file. In C, we can use the ftell() and fseek() functions to determine the size of a file.

The ftell() function returns the current position of the file pointer within a file. We can use this function to determine the size of a file by first moving the file pointer to the end of the file using fseek(), and then calling ftell() to get the position of the file pointer.

Here is an example of how to use ftell() and fseek() to determine the size of a file:

FILE \*file = fopen("example.txt", "rb");

if (file != NULL)

{

fseek(file, 0L, SEEK\_END);

long int size = ftell(file);

printf("The size of the file is %ld bytes.\n", size);

fclose(file);

}

In this example, we first open the file "example.txt" in binary mode using fopen(). We then use fseek() to move the file pointer to the end of the file by passing the constant SEEK\_END as the second argument. The first argument to fseek() is the file pointer, and the third argument specifies the offset from the current position, which is 0 in this case.

After moving the file pointer to the end of the file, we call ftell() to get the current position of the file pointer, which is the size of the file in bytes. We then print the size of the file using printf(), and close the file using fclose().

It is important to note that when using ftell() to determine the size of a file, we must first move the file pointer to the end of the file using fseek(). If we attempt to call ftell() without first moving the file pointer, we may get an incorrect result.

## Common File Handling Errors

Working with files can lead to a variety of errors, such as file not found errors, permission errors, or errors related to file input/output. It is important to handle these errors properly in our code to prevent our program from crashing or producing unexpected results.

One common error that can occur when working with files is attempting to open a file that does not exist. When opening a file using fopen(), if the specified file does not exist and the mode is set to "r" (read), fopen() will return NULL. We can check for this error by checking if the file pointer returned by fopen() is NULL.

## Here is an example of how to check for a file not found error:

FILE \*file = fopen("nonexistent.txt", "r");

if (file == NULL)

{

printf("Error: File not found.\n");

}

In this example, we attempt to open a file called "nonexistent.txt" in read mode. If the file does not exist, fopen() will return NULL, indicating that an error has occurred. We can check for this error by comparing the file pointer returned by fopen() to NULL. If the file pointer is NULL, we print an error message indicating that the file was not found.

Another common file handling error is attempting to read or write to a file that is not open or has been closed. This can occur if the file pointer is not properly initialized or if the file has already been closed with fclose(). To avoid this error, always check if the file pointer is not NULL before attempting to read or write to the file, and make sure to properly close the file when finished using it.

## Glossary

File handling

The process of reading from and writing to files in a program.

Text file

A file that contains human-readable text, such as plain text documents or CSV files.

Binary file

A file that contains machine-readable data, such as executable files or images.

ASCII

A character encoding standard that represents each character as a unique 7-bit code.

Unicode

A character encoding standard that represents each character as a unique 16-bit or 32-bit code.

Mode

A parameter used when opening a file that specifies the intended use of the file (e.g. read, write, append).

Error handling

The process of handling errors that may occur during file input/output.

File pointer

A variable that points to a specific position in a file.

Permission is granted to copy, distribute, transmit and adapt this work un- der the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 Inter- national License: https://creativecommons.org/licenses/by-nc/4.0/. If you are interested in distributing a commercial version of this work, please contact the author(s). The LATEX source and code for this book is available from: https://github.com/mkpenta/ThinkC