File handling in C is a crucial concept that enables programs to interact with files stored on the disk. It involves reading data from files, writing data to files, and manipulating files in various ways.

Introduction to File Handling

In C, file handling provides a way to store data permanently on the disk. This is important for data that needs to persist beyond the execution of a program, such as user data, configurations, and logs. Files are used to store data in a structured way, making it easy to read and write information.

File Operations

The common operations performed on files are:

- **Creating a file:** Creating a new file if it doesn't exist.
- **Opening a file:** Accessing an existing file for reading, writing, or both.
- **Reading from a file:** Extracting data from a file.
- **Writing to a file:** Adding data to a file.
- **Closing a file:** Releasing resources associated with a file.

File Pointer

A file pointer is a special pointer in C that points to a file. It is used to interact with files and is declared as `FILE *`. This pointer is used with various file handling functions provided by the standard I/O library.

File Modes

When opening a file, a mode must be specified to indicate the type of operation. Common modes include:

- `"r"`: Open a file for reading.
- `"w"`: Create a file for writing. If the file exists, its contents are truncated.
- `"a"`: Open a file for appending. Data is added to the end of the file.
- `"r+"`: Open a file for both reading and writing.
- `"w+"`: Create a file for reading and writing. If the file exists, its contents are truncated.
- `"a+"`: Open a file for reading and appending.

Binary Modes

In addition to the above modes, each mode can be suffixed with "b" to indicate that the file is a binary file. This distinction is more significant on systems where text and binary files are handled differently (e.g., Windows).

Examples:

- `"rb"`: Open for reading in binary mode.
- `"wb"`: Open for writing in binary mode.
- `"ab"`: Open for appending in binary mode.
- `"r+b"` or `"rb+"`: Open for reading and writing in binary mode.
- `"w+b"` or `"wb+"`: Open for writing and reading in binary mode.
- `"a+b"` or `"ab+"`: Open for appending and reading in binary mode.

Error Handling

It's crucial to handle errors in file operations. For example, if a file cannot be opened, `fopen()` returns `NULL`. Checking for this and other error conditions ensures that the program can handle unexpected situations gracefully.

EOF and File Position

The end of a file is indicated by the `EOF` (End of File) marker. The `fgetc()` and similar functions return `EOF` when the end of the file is reached. The position within the file can be manipulated using functions like `fseek()` and `ftell()`.

Binary vs. Text Files

Files can be either text files or binary files. Text files contain readable characters, while binary files contain data in binary format. Different functions are used to handle these file types, such as `fwrite()` and `fread()` for binary files.

Advantages of File Handling

- **Data Persistence:** Data can be stored permanently.
- **Data Sharing:** Files can be shared across different programs.
- **Large Data Storage:** Files can store large amounts of data.

```
FILE *fp = fopen("filename.txt", "r");
FILE *fp = fopen("filename.txt", "w");
FILE *fp = fopen("filename.txt", "a");
FILE *fp = fopen("filename.txt", "r+");
FILE *fp = fopen("filename.txt", "w+");
FILE *fp = fopen("filename.txt", "a+");
```

Summary Table

Function	Input/output	Description
`getc()`	Input	Reads a single character from a file stream.
`getchar()`	Input	Reads a single character from stdin.
`getch()`	Input	Reads a single character from keyboard, no echo.
`getche()`	Input	Reads a single character from keyboard, with echo.
`gets()`	Input	Reads a string from stdin (unsafe).
`fgets()`	Input	Reads a string from a file/stream (safe).
`putc()`	Output	Writes a single character to a file stream.
`putch()`	Output	Writes a single character to the console.
`putchar()`	Output	Writes a single character to stdout.
`puts()`	Output	Writes a string to stdout with a newline.

`stdin` (Standard Input)

- **Purpose**: It is the default input stream, typically used to receive input from the user through the keyboard.
- **Default Input Source**: Usually the keyboard, but it can be redirected to read from a file or another input device.

`stdout` (Standard Output)

- **Purpose**: It is the default output stream, typically used to display output on the console or terminal.
- **Default Output Destination**: Usually the console or terminal, but it can be redirected to write to a file or another output device.

Difference between Text and Binary Files

Text files and binary files differ fundamentally in how they store and represent data:

Aspect	Text Files	Binary Files
Data Representation	Stores data in human-readable text (ASCII/Unicode characters).	Stores data in its native binary format, which is not human-readable.
File Extensions	Common extensions include `.txt`, `.csv`, `.html`.	Common extensions include `.bin`, `.exe`, `.dat`, `.jpg`.

Aspect	Text Files	Binary Files
Storage Efficiency	Less efficient because characters are stored as their ASCII/Unicode equivalents, requiring more space for certain types of data.	More efficient for non-textual data, as it stores data in its original format, saving space.
Accessibility	Can be opened and edited using text editors (e.g., Notepad, Vim, etc.).	Requires specific programs (e.g., image editors, media players, compilers) to interpret and modify.
End of Line	Uses newline characters (`\n` or `\r\n`) to denote the end of a line.	Does not follow specific line-ending conventions; data is stored as continuous bytes.
Interpretation	Interpreted as characters or strings by programs.	Interpreted based on data type (integers, floats, images, sounds, etc.) as defined by the file's format.
Use Case	Ideal for storing plain text, such as source code, configuration files, and log files.	Used for media files (images, audio, and video), compiled programs, and structured data (e.g., databases).
Portability	More portable between different platforms, but encoding differences (e.g., Unicode, UTF-8) can cause issues.	Requires consistent format interpretation across platforms (e.g., little-endian vs big-endian byte order).
Corruption Handling	Easier to recover from partial corruption (since it's line-based and human-readable).	Harder to recover from corruption as the entire structure may be compromised.

A file pointer is a pointer to a structure of type FILE.

Opening a file:

- **Example**: FILE *fp;
- The function **fopen ( )** is used to open a file.

- **Syntax**: fp=fopen(“path:\\filename.ext”, “file_operation_mode”);

Closing a file:

- **Syntax**: fclose(name_of_file_pointer);
- **Example**: fclose(fp);

Input/output Functions:

The functions used for file input/output are

Character I/O

fputc( ) :

• This function writes a character to the specified file at the current file position and then increments the file position pointer.
• Syntax: fputc(character,file_pointer);

fgetc( ) :

• This function reads a single character from a given file and increments the file pointer.
• Syntax: fgetc(file_pointer);

Integer I/O:

putw( ):

• This function writes an integer value to the file pointed to by file pointer.
• Syntax: putw(Integer,file_pointer);

getw( ):

• This function returns the integer value from the file associated with file pointer.
• Syntax: getw(file_pointer);

String I/O:

fputs():

• This function writes the null terminated string pointed by given character pointer to a file.
• Syntax: fputs(string,file_pointer);

fgets():

• This function is used to read characters from a file and these characters are stored in the string pointed by a character pointer.
• Syntax: fgets(string,length_of_string,file_pointer);

Formatted Input/output Functions:

fprintf( )

-This function is same as the printf( ) function but it writes formatted data into the file instead of the standard output(screen).
-This function has same parameters as in printf( ) but it has one additional parameter which is a pointer of FILE type, that points to the file to which the output is to be written.
Syntax: fprintf(fp, “control_string”,list of variables);

fscanf( )

- This function is similar to the scanf( ) function but it reads data from file instead of standard input, so it has one more parameter which is a pointer of FILE type and it points to the file from which data will be read.
Syntax: fscanf(fp, “control_string”, &list_of_variables);