Chapter 13: File Handling

Introduction

In modern software development, interacting with files is essential for tasks like data persistence, logging, configuration management, and data interchange. **File Handling** in programming refers to the process of creating, reading, writing, and manipulating files stored on disk using programming constructs. This chapter delves into advanced file handling techniques, beyond the basics, focusing on robust practices, performance optimization, error handling, and working with different file formats in **C++, Java, and Python**—languages often used in BTech CSE curricula.

13.1 File Basics

13.1.1 What is a File?

A file is a named location on disk that stores data persistently. It can contain data in different formats such as plain text, binary, CSV, JSON, XML, etc.

13.1.2 Types of Files

- **Text Files**: Contain human-readable characters.
- **Binary Files**: Store data in binary format (non-readable by humans).

13.2 File Operations

13.2.1 Common Operations

- **Create**: Make a new file.
- **Open**: Access an existing file.
- **Read**: Extract data from a file.
- **Write**: Insert data into a file.
- **Append**: Add data at the end.
- **Close**: Free resources used for file access.

13.2.2 File Modes

Mode	Description
`r`	Read-only
`w`	Write (overwrites existing)
`a`	Append
`rb`	Read binary

Mode	Description
`wb`	Write binary
`r+`	Read and write (no overwrite)
`w+`	Read and write (overwrite)

13.3 File Handling in C++

13.3.1 fstream Library

```
#include <fstream>
using namespace std;
```

13.3.2 Reading and Writing

```
ofstream fout("data.txt");
fout << "Hello File!";
fout.close();

ifstream fin("data.txt");
string line;
getline(fin, line);
cout << line;
fin.close();
```

13.3.3 File Pointers

- `seekg()` / `seekp()` – Set position for reading/writing
- `tellg()` / `tellp()` – Get current position

13.4 File Handling in Java

13.4.1 FileReader and FileWriter

```
import java.io.*;

FileWriter fw = new FileWriter("output.txt");
fw.write("Hello Java File!");
fw.close();

FileReader fr = new FileReader("output.txt");
int i;
while ((i = fr.read()) != -1)
    System.out.print((char)i);
fr.close();
```

13.4.2 BufferedReader and BufferedWriter

Efficient for large text files.

```
BufferedReader br = new BufferedReader(new FileReader("file.txt"));
BufferedWriter bw = new BufferedWriter(new FileWriter("file.txt"));
```

13.4.3 File Class

Used for file metadata and manipulation.

```
File f = new File("data.txt");
System.out.println(f.exists());
```

13.5 File Handling in Python

13.5.1 Opening and Reading

```
f = open("data.txt", "r")
content = f.read()
print(content)
f.close()
```

13.5.2 Writing to a File

```
with open("data.txt", "w") as f:
    f.write("Hello Python File!")
```

13.5.3 File Context Manager

Python uses `with` to automatically close files.

13.6 Working with Binary Files

13.6.1 C++ Example

```
ofstream fout("data.bin", ios::binary);
int x = 100;
fout.write((char*)&x, sizeof(x));
fout.close();
```

13.6.2 Python Example

```
with open("data.bin", "wb") as f:
    f.write(b'\x64')  # 100 in hex
```

13.7 Error and Exception Handling

13.7.1 C++

Use `fail()` or `bad()` methods on file streams:

```
if (fin.fail()) {
    cerr << "Error opening file.";
}
```

13.7.2 Java

Use `try-catch` blocks.

```
try {
    FileReader fr = new FileReader("file.txt");
} catch (IOException e) {
    e.printStackTrace();
}
```

13.7.3 Python

```
try:
    f = open("file.txt", "r")
except FileNotFoundError:
    print("File not found.")
```

13.8 Advanced Topics

13.8.1 File Locking

To prevent race conditions when multiple programs access the same file.

- **Java**: `FileChannel.lock()`
- **Python**: `fcntl` or `msvcrt` module

13.8.2 Random Access Files

- **C++**: `seekg()`, `seekp()`
- **Java**: `RandomAccessFile`
- **Python**: `seek()` and `tell()`

13.9 Working with Different File Formats

13.9.1 CSV Files

- **Python**: `csv` module
- **Java**: OpenCSV
- **C++**: Manual parsing

13.9.2 JSON Files

- **Python**: `json` module
- **Java**: Gson or Jackson

- **C++**: nlohmann/json library

13.9.3 XML Files

- **Python**: `xml.etree.ElementTree`
- **Java**: DOM/SAX parser

13.10 Best Practices in File Handling

- Always close files after use.
- Use context managers where possible.
- Handle exceptions robustly.
- Validate file paths and names.
- Avoid hardcoded paths – use dynamic or relative paths.
- Use buffers for large file operations.
- Avoid race conditions in multi-threaded environments.

Summary

This chapter explored the depths of file handling, covering not only basic I/O operations but also advanced practices such as file locking, working with structured formats like JSON/XML, and random access techniques. Mastery of file handling empowers developers to build applications with robust data storage, logging, and configuration capabilities—key requirements in real-world programming.
