**Detailed Explanation of the Assignment**

In this assignment, we implemented a file system using C++ with an Object-Oriented Programming (OOP) approach. The primary goal was to create a functional file system that could handle basic file operations such as creation, deletion, reading, and writing, as well as directory management. This required designing and implementing several key classes: File, Directory, File System, and File Descriptor.

#### **Design Phase**

**1. UML Class Diagram**

The design phase began with creating a UML class diagram to visualize the relationships between different classes in the system. The main classes identified were:

* **File:** Represents a file with a name and data.
* **Directory:** Represents a directory that can contain files and other directories.
* **FileSystem:** Manages the root directory and provides methods for file and directory operations.
* **FileDescriptor:** Represents a reference to a file with a position, used for handling file operations.

**2. Use Case Diagram**

A use case diagram was created to illustrate the interactions between the user and the system functions. This diagram includes use cases for creating, deleting, reading, and writing files, as well as creating and deleting directories and listing directory contents.

### **Implementation Phase**

#### **1. file.hpp and file.cpp**

**Purpose:** The File class represents a file in the file system, encapsulating its name and content.

***Implementation Details:***

* **file:** Declares the File class with its private members (name and data) and public methods (get Name, read, write).
* **file.cpp:** Implements the methods declared in filehpp, enabling operations such as reading and writing data to a file.

**Explanation:**

* The File class has methods to get the file name, read data from the file, and write data to the file. This class is essential for handling individual file operations within the file system.

#### **2. directory.hpp and directory.cpp**

**Purpose:** The Directory class represents a directory, which can contain files and other directories.

***Implementation Details:***

* **directory.hpp:** Declares the Directory class with its private members (name, files, directories) and public methods (get Name, addFile, removeFile, addDirectory, removeDirectory, listContents).
* **directory.cpp:** Implements the methods declared in directory, enabling operations such as adding and removing files and directories, and listing directory contents.

**Explanation:**

* The Directory class is crucial for organizing the file system's hierarchy. It allows adding and removing files and subdirectories and listing all contents within a directory.

#### **3. filesystem.hpp and filesystem.cpp**

**Purpose:** The FileSystem class provides a user interface for performing file and directory operations.

***Implementation Details:***

* **filesystem.hpp:** Declares the FileSystem class with its private member (root directory) and public methods (createFile, deleteFile, readFile, writeFile, createDirectory, deleteDirectory, listDirectoryContents).
* **filesystem.cpp:** Implements the methods declared in filesystem.hpp, providing the core functionality for interacting with the file system.

**Explanation:**

* The FileSystem class acts as a manager for the entire file system, starting with a root directory. It offers methods to create, delete, read, and write files, as well as create and delete directories and list directory contents. This class integrates the functionality of the File and Directory classes to manage the file system.

#### **Makefile**

**Purpose:** Automates the build process for the project.

***Implementation Details:***

* **Makefile:** Specifies the compiler options, source files, and the build rules required to compile the project.

**Explanation:**

* The Makefile includes rules for compiling each source file into an object file and linking the object files to create the final executable. It also includes a clean rule to remove the compiled files.

#### **Version Control**

**Purpose:** Tracks changes to the project's codebase over time using Git.

**Implementation Details:**

* A Git repository was created, and commits were made regularly with clear messages describing the changes.

**Explanation:**

* Version control is essential for managing changes to the code. It allows tracking progress, reverting to previous versions if necessary, and collaborating with others. A screenshot of the GitHub repository was included to show the commit history and messages.

### **Testing Phase**

#### **Testing Approach**

**Purpose:** Ensure the functionality and correctness of the file system implementation.

**Implementation Details:**

* Used the Catch2 framework to write unit tests for the FileSystem class and its components.

**Explanation:**

* Testing was conducted to verify that files could be created, read, written to, and deleted correctly. Directory operations were also tested to ensure that directories could be created, listed, and managed properly.

#### **Test Cases**

**Purpose:** Validate specific functionalities of the file system.

**Implementation Details:**

* Test cases were written to cover various scenarios, including file creation, deletion, reading, writing, and directory operations.

**Explanation:**

* The test cases ensured that the implemented methods performed as expected. Each test case focused on a particular functionality, verifying the correct behavior of the system under different conditions.

### **Conclusion**

#### **1. Summary**

**Purpose:** Summarize the project and its achievements.

**Explanation:**

* The project successfully implemented a basic file system using C++ and OOP principles, providing essential file and directory management functionalities.

#### **2. Limitations**

**Purpose:** Identify the current limitations of the implementation.

**Explanation:**

* The current implementation does not support advanced features such as file permissions, multi-user environments, or persistent storage across sessions.

#### **3. Future Improvements**

**Purpose:** Suggest potential improvements for future projects.

**Explanation:**

* Future projects could incorporate comprehensive error handling, optimize performance, and add support for persistent storage to enhance the file system's capabilities.