DEPARTMENT OF COMPUTER & SOFTWARE ENGINEERING

COLLEGE OF E&ME, NUST, RAWALPINDI

File Storage System

Project Report

SUBMITTED TO:
Lec Anum Abdul Salam/LE Kashaf Raheem

Course Name:
Data Structures and Algorithms (DSA)

SUBMITTED BY:

Aleeza Rizwan	**M Ibrahim Abdullah**	**M Shaheer Afzal**
456143	**454188**	**481560**
DE-45 Dept CE A	**DE-45 Dept CE A**	**DE-45 Dept CE A**

Submission Date: 14th May 2025

Table of Contents

Project Overview

Introduction

This project presents a custom-built Virtual File Storage System implemented entirely in C++. The system simulates a simple yet functional file system using a single 10MB binary file named File_system.bin, effectively acting as a virtual hard disk. Within this environment, users can perform basic file operations such as creating, viewing, modifying, deleting, importing, and exporting text files.

Purpose and Motivation

Most operating systems abstract away the complexity of how file systems manage memory, organize files, and store metadata. While convenient, this abstraction limits opportunities for students and developers to explore the inner workings of disk storage and file allocation. This project aims to demystify those internal mechanisms by providing a simplified model of a file system that mimics essential behavior found in real-world formats like FAT or ext.

Virtual Disk Structure

The virtual disk (File_system.bin) is divided into three main logical sections:

1. **Metadata Section (1 MB):**
   - Stores file entries, each containing a file name and the starting address of its data blocks.
   - Entries are fixed-size (500 bytes) and zero-padded for consistency.

2. **Free Block List (1 MB):**
   - Maintains a comma-separated list of unallocated block indices.
   - Helps in managing space and allocating blocks dynamically during file operations.

3. **Data Section (8 MB):**
   - Stores the actual contents of files.
   - File contents are stored linearly in memory, starting from a computed free address and extending by the size of the file.

File Storage Method

- Files are stored linearly in memory, with direct addressing for faster access and simpler implementation.
- Each 1KB block contains:
  - A fixed-size chunk of the file's content.
- A special delimiter ('\0') marks the end of file content, simplifying traversal and read

operations.

Design and Classes

- **File:** Represents a file with its name, starting address, and size.
- **FileSystem:** Manages the entire system, including:
    - Disk initialization
    - Metadata management
    - Free block allocation
    - File creation, deletion, modification, viewing
    - Import/export operations with Windows

Interfacing with the Host OS

The project supports interoperability with the Windows file system:

- **Import:** Reads content from a .txt file on disk and stores it in the virtual hard disk.
- **Export**: Retrieves content from a virtual file and saves it as a physical .txt file on the host system. This bridges the gap between the simulated and real environments, enhancing both usability and realism.

In addition to import/export support, the project integrates a graphical user interface (GUI) built using Windows Forms via C++/CLI. This allows users to interact with the file system using buttons, drop-downs, and dialog boxes rather than through the command line. Features like file creation, viewing, and deletion can be performed through GUI elements, enhancing usability and accessibility

Graphical User Interface (GUI) Integration

To enhance usability and provide a more intuitive user experience, a graphical user interface (GUI) was integrated into the project using **C++/CLI and Windows Forms**. This GUI enables users to perform all essential file operations—such as creating, viewing, deleting, and importing/exporting files—through interactive controls like buttons, text fields, and native Windows file dialogs.

The GUI eliminates the need for command-line interaction, making the system more accessible to non-technical users. It leverages managed .NET components like MessageBox, TextBox, and MenuStrip, and visually represents key operations through form-based navigation. This not only improves usability but also bridges the gap between theoretical data structure concepts and real-world software application design.

By adding a GUI, the project demonstrates how low-level file system logic can be abstracted into a user-friendly interface, reinforcing core software engineering principles.

Educational Value

By building a file system from scratch, the project:

- Reinforces concepts like memory allocation, block-based storage, and metadata handling.

- Demonstrates practical use of fstream, vectors, and maps for binary data handling.
- Encourages understanding of real-world file system challenges like fragmentation and block reuse.

Scalability and Extensions

The current implementation forms a strong foundation for further development. Potential future enhancements include:

- Implementing defragmentation
- Supporting directories and subdirectories
- Adding file permissions and timestamps
- Visualizing block allocation as a debugging or learning tool

Problem Statement

Traditional file systems are complex and managed at the operating system level, making it difficult to understand their internal working. This project aims to simplify and simulate file storage at a lower level using a custom-managed virtual disk. The goal is to:

- Simulate disk block management
- Provide insight into file allocation, metadata storage, and fragmentation
- Allow file operations without interacting with the actual OS file system

Objectives

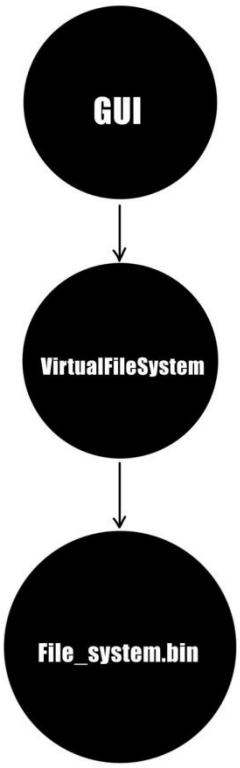
- Create a 10MB virtual disk inside a binary file.
- Divide the disk logically into metadata, free block list, and data sections.
- Support creation, modification, deletion, and viewing of text files.
- Enable importing and exporting .txt files to/from the virtual disk.
- Manage free space through a custom free block list.
- Implement basic block-level file allocation using linked lists.

System Design

Class Diagram

VirtualFileSystem
- files : vector<File> - disk : char* - freeBlocks : bool* - metadataFilePath : string
+ VirtualFileSystem ( ) + ~VirtualFileSystem ( ) + findFreeBlock (int) : (int) + createFile (string, string) : void + listFile ( ) : void + viewFile (string) : void + deleteFile (string) : void + copyToWindows (string) : void + copyFromWindows (string) : void + deleteAll ( ) : void + resetSystem ( ) : void

System Level Diagrams



Implementation Details

- **Disk File:** A 10MB binary file File_system.bin is initialized.
- **Structure:**
  - **1 MB** – Metadata (file name + starting address).
  - **1 MB** – Free block list (CSV-style).
  - **8 MB** – Actual file data, stored in 1KB blocks.
- **File Metadata:** Stored in fixed-size entries of 500B using simple serialization.
- **Free Blocks:** Managed in-memory using a vector<int> and stored as a comma-separated list.
- **File Data:** Stored sequentially within a memory buffer and written directly to virtual disk space using address offsets.
- **Classes:**
  - File: Represents a file with name, starting address, and size.
  - FileSystem: Handles all logic including file creation, deletion, disk management, metadata, and user I/O
- **Graphical Interface:** The system includes a C++/CLI-based Windows Forms GUI for performing file operations, using managed components like Button, TextBox, and OpenFileDialog.

Time Complexity Analysis

createFile(name, content)

- **Best/Average Case:** O(n)
- **Why:** The function finds a sequence of n free bytes, writes n bytes of content, and updates metadata.

viewFile(name)

- **Time Complexity:** O(n)
- **Why:** File data is read sequentially from its starting address until the file size is reached.

deleteFile(name)

- **Time Complexity:** O(n)
- **Why:** It marks n bytes as free and removes the file metadata.

deleteAllFiles( )

- **Time Complexity:** O(f + m)
  - f = number of files
  - m = total bytes occupied
- **Why:** Clears metadata and resets all occupied blocks.

modifyFile(name)

- **Time Complexity:** O(n + m)
  - n = original file size
  - m = appended content size
- **Why:** Old content is copied into memory and new content is appended. A new location may be used.

copyToWindows(name)

- **Time Complexity:** O(n)
- **Why:** File content is read and written byte by byte to the Windows filesystem.

copyFromWindows(filename)

- **Time Complexity:** O(n)
- **Why:** Reads file content from disk and calls createFile() internally, so it inherits its complexity.

listFiles()

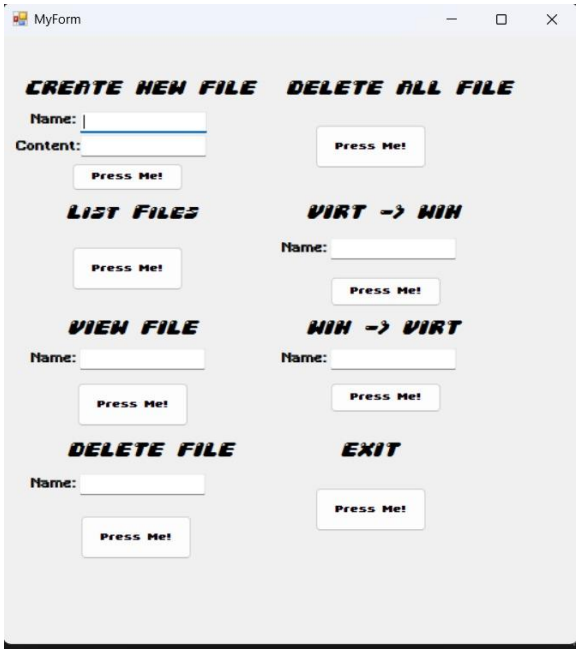
- **Time Complexity:** O(f)
- **Why:** Iterates through all stored file metadata and prints file names and sizes.

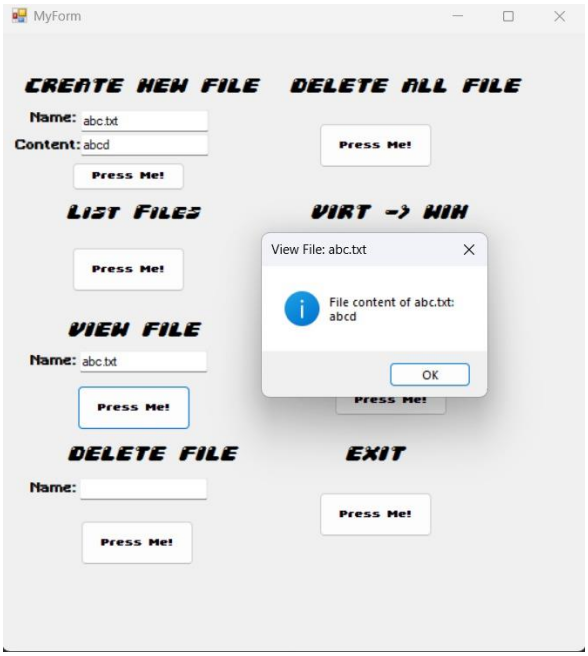
.

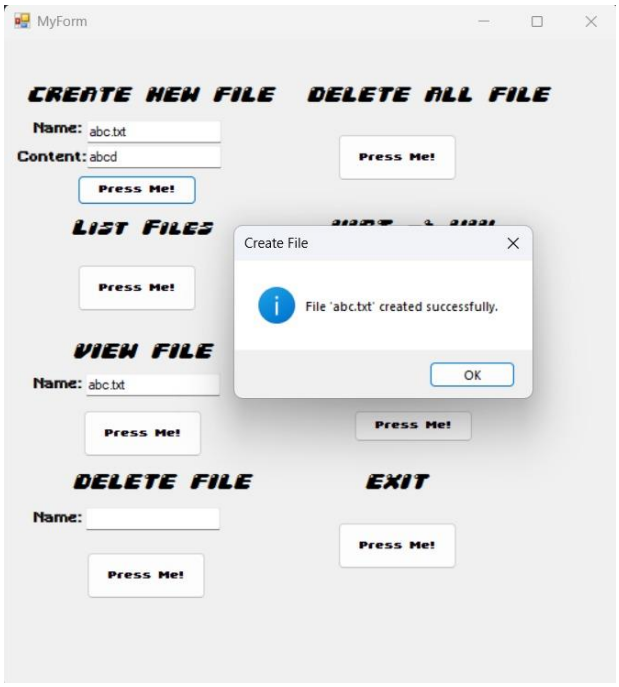
Key Features

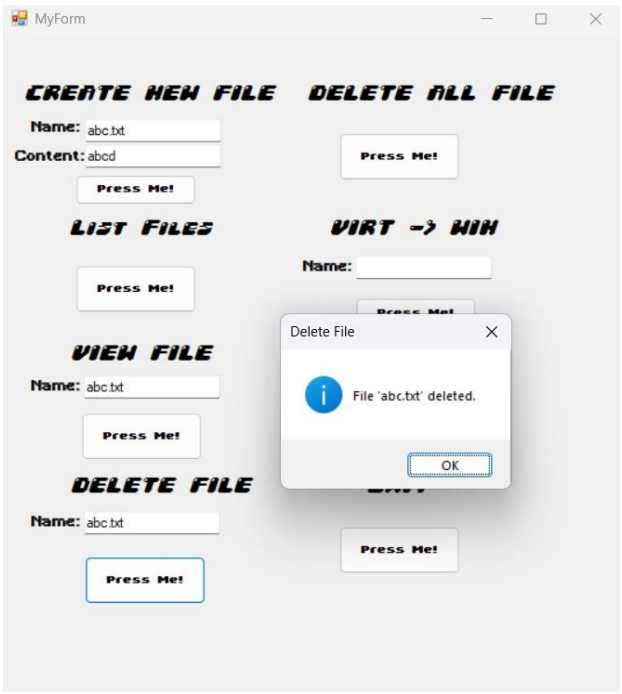
- **Create File:** Enter a filename and content; data is stored sequentially in virtual memory.
- **View File:** Reads file content from its stored address and displays it.
- **Modify File:** Appends data directly after the existing content using address-based offset calculation.
- **Delete File:** Frees blocks and wipes metadata entry.
- **Free Block Management:** Dynamically updates available block list.
- **Import File:** Reads a Windows file and copies its content into the virtual system.
- **Export File:** Retrieves file content and writes it to the real file system.
- **GUI Support:** A graphical interface allows file operations through form controls like buttons and dialog boxes. File selection uses native Windows file pickers.

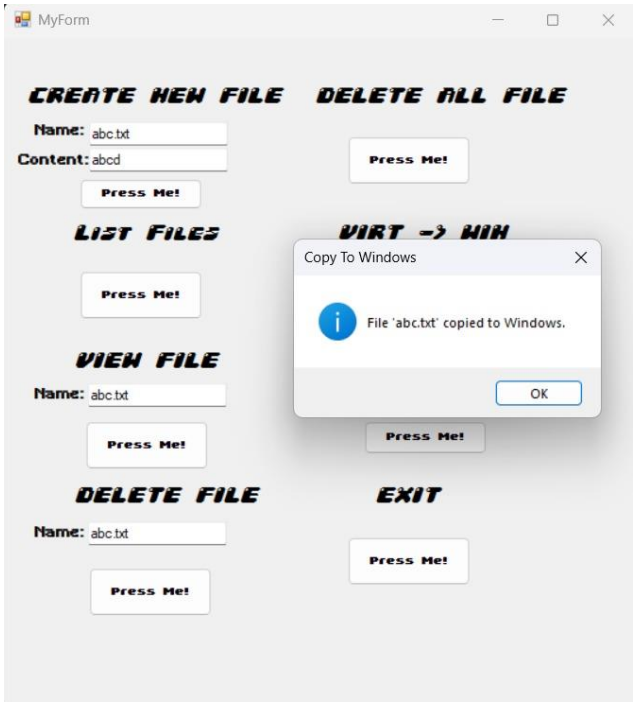
Results and Outputs

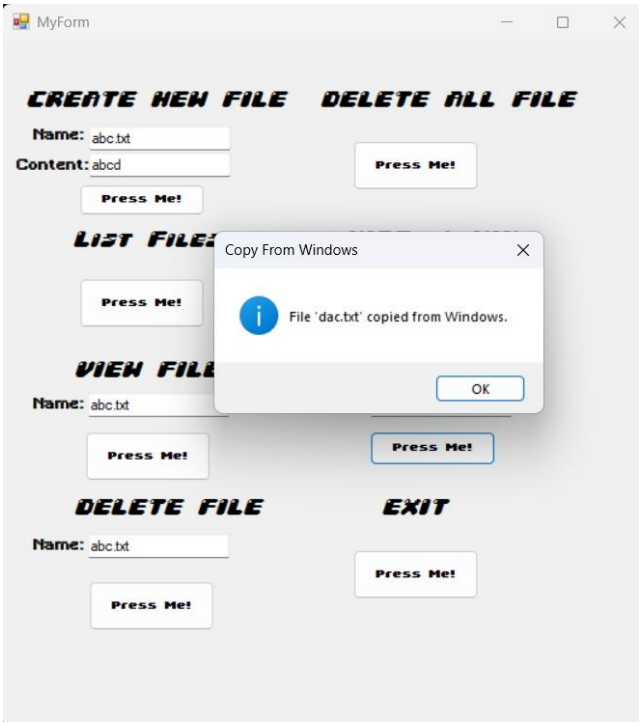


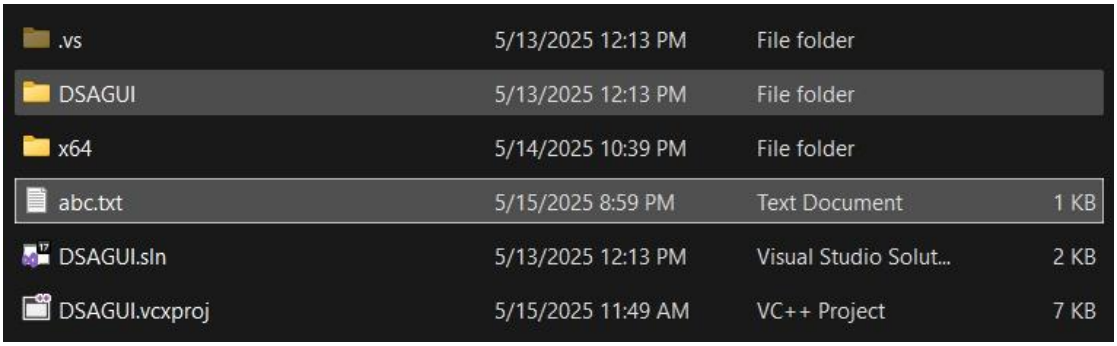



MyForm
CREATE NEW FILE
DELETE ALL FILE
Name: abc.txt
Content: abcd
Press Me!
Press Me!
LIST FILES
Press Me!
Create File
File 'abc.txt' created successfully.
OK
VIEW FILE
Name: abc.txt
Press Me!
Press Me!
DELETE FILE
EXIT
Name:
Press Me!
Press Me!


MyForm
CREATE NEW FILE
DELETE ALL FILE
Name: abc.txt
Content: abcd
Press Me!
Press Me!
LIST FILES
VIRT -> WIN
Name:
Press Me!
Delete File
File 'abc.txt' deleted.
OK
VIEW FILE
Name: abc.txt
Press Me!
DELETE FILE
Name: abc.txt
Press Me!
Press Me!


MyForm
CREATE NEW FILE
Name: abc.txt
Content: abcd
Press Me!
DELETE ALL FILE
Press Me!
LIST FILES
Press Me!
VIRT -> WIN
Copy To Windows
File 'abc.txt' copied to Windows.
OK
Press Me!
VIEW FILE
Name: abc.txt
Press Me!
DELETE FILE
Name: abc.txt
Press Me!
EXIT
Press Me!


MyForm
CREATE NEW FILE
Name: abc.txt
Content: abcd
Press Me!
DELETE ALL FILE
Press Me!
LIST FILES
Press Me!
Copy From Windows
File 'dac.txt' copied from Windows.
OK
VIEW FILE
Name: abc.txt
Press Me!
Press Me!
DELETE FILE
Name: abc.txt
Press Me!
EXIT
Press Me!



Challenges and Solutions

- **Managing memory inside a single file:**
  - Solution: Simulated block-based memory management with metadata and offsets.
- **Keeping track of used/free blocks:**
  - Solution: Custom free block list managed in a 1MB section of the file.
- **Handling variable file sizes:**
  - Solution: Used sequential memory allocation with dynamic sizing and delimiter-based termination.
- **Exporting/Importing file data:**
  - Solution: Used string processing and fstream to integrate with real files.

Conclusions

This project successfully simulates a simplified file system within a virtual hard disk. It provides a hands-on learning experience in managing memory, metadata, and file storage without relying on the OS's native capabilities. It can be extended further with features like defragmentation, file permissions, directories, and optimization techniques for storage.

References

1. Carrano, F. M. (2013). *Data abstraction and problem solving with C++: Walls and mirrors* (6th ed.). Pearson.
2. *cppreference.com*. (2019). Cppreference.com. https://en.cppreference.com/w/
3. Deitel, P., & Deitel, H. (2014). *C++ how to program* (9th ed.). Pearson.
4. Goodrich, M. T., Tamassia, R., & Mount, D. (2011). *Data structures and algorithms in C++* (2nd ed.). Wiley.
5. Weiss, M. A. (2013). *Data structures and algorithm analysis in C++* (4th ed.). Pearson.