**Submitted by:**

**Hunaina Ehsan**

**Zainab Kashif**

Project Report

# Introduction:

In this project, we worked on modifying an existing code file called "[Original Code](https://github.com/pfdinc/Operating_Systems_Mini_Shell)" to enhance its functionality and add new features in “[Modified Code](https://github.com/Hunaina-Ehsan/project)”. The code is designed to create a simple command-line shell where users can execute various commands.

The modifications we made to the original code include adding the following commands: **rm**, **cd**, **touch**, **grep**, **write**, **writeat**, **read**, **readat**, **head**, **tail**, **man**, **rmdir**, and command concatenation. These commands provide additional functionality to the shell, allowing users to perform common file operations, search for patterns in files, manipulate file contents, and navigate the directory structure.

# Application Overview:

The CLI application provides a command prompt where users can enter commands to perform various tasks. The application supports a range of commands, including file creation (touch), printing the current working directory (pwd), file removal (rm), pattern matching in files (grep), displaying command manuals (man), directory removal (rmdir), file concatenation (cat), file read/write operations, and more. The application also supports piping commands together using the '|' symbol.

# Design Considerations:

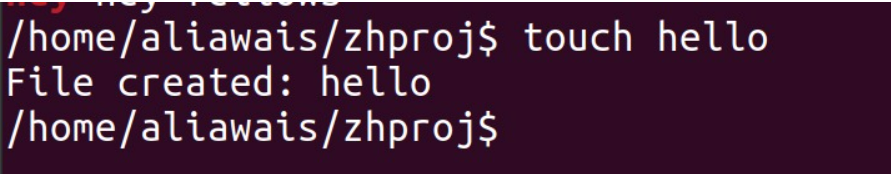
The application follows a modular design approach, with each command implemented as a separate function. This design allows for easy extensibility and maintainability. The application leverages standard C++ libraries such as **<iostream>**, **<string>**, **<fstream>**, **<unistd.h>**, and **<sys/wait.h>** for input/output handling, string manipulation, file operations, process management, and more.

# Implementation Details:

The application utilizes functions such as **ExecuteCommand**, **TouchCommand**, **PwdCommand**, **RmCommand**, **GrepCommand**, **ManCommand**, **RmdirCommand**, **CatCommand**, **WriteCommand**, **ReadCommand**, **HeadCommand**, **TailCommand**, and more to implement the respective command functionalities. The input is processed using the **getline** function, and commands are tokenized for execution. The application uses **fork** and **execvp** functions for creating child processes and executing commands.

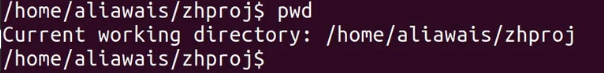
Following are how we implemented the commands and its outputs.

## Touch: The **TouchCommand** function creates a new file with the specified filename. It uses the **fopen** function to open the file in write mode ("w"). If the file creation is successful, the function closes the file using **fclose**. If the file creation fails, an error message is displayed. The purpose of this function is to mimic the behavior of the "touch" command in Unix systems, allowing the user to create an empty file with a single function call.



## Pwd:

The **PwdCommand** function retrieves the current working directory and stores it in the **cwd** variable. It uses the **getcwd** function to get the current working directory path and saves it into **cwd** using the specified buffer size. If the operation is successful, the function displays the current working directory using the **cout** statement. If the operation fails, an error message is displayed. This function mimics the behavior of the "pwd" command in Unix systems, allowing the user to obtain the current working directory path with a single function call.



## Rm:

The **RmCommand** function attempts to remove a file or directory specified by the **filename** parameter. It uses the **remove** function to delete the file or directory. If the operation is successful (i.e., the return value of **remove** is 0), the function displays a success message indicating that the file or directory has been removed. However, if the operation fails, an error message is displayed. This function replicates the functionality of the "rm" command in Unix systems, allowing the user to delete files or directories with a single function call.

## Grep:

The **GrepCommand** function searches for a given **pattern** within a specified **filename**. It first opens the file in read mode using **fopen** and checks if the file exists. If the file cannot be opened, an error message is displayed, and the function returns.

The function then reads the file line by line using **fgets** and stores each line in the **line** buffer. It checks if the **pattern** exists within the line using **strstr**. If a match is found, it performs the following actions:

1. Prints the part of the line before the matched pattern.
2. Applies highlighting to the matched pattern using ANSI escape codes (**\033[1;31m** for red color).
3. Prints the rest of the line after the matched pattern.

This approach visually emphasizes the matched pattern in the output. The function continues to search for matches in subsequent lines until the end of the file is reached.

Finally, the function closes the file using **fclose**. This function replicates the functionality of the "grep" command in Unix systems, allowing the user to search for patterns within a file.



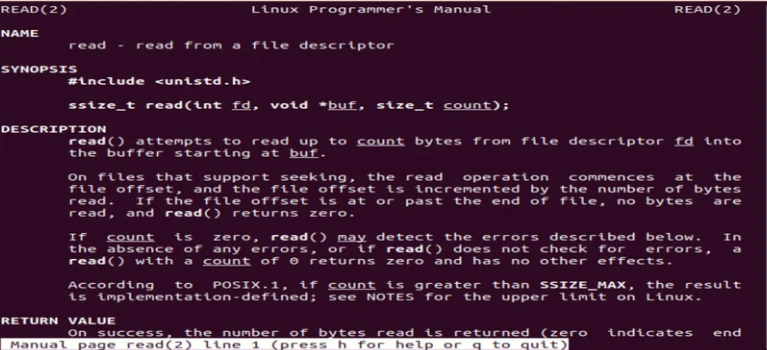
## Man:

The **ManCommand** function is a simple wrapper around the **man** command in Unix systems. It takes a **command** as input and constructs a string **manualCommand** by appending the provided command to the string "man ".

The constructed command is then executed using the **system** function. The **system** function allows the execution of a command in the underlying operating system's shell.

By invoking the **man** command with the specified **command**, the function opens the corresponding manual page for that command in the terminal. This provides the user with access to the manual documentation, which contains detailed information about the usage, options, and examples of the specified command.

In summary, the **ManCommand** function facilitates accessing the manual pages of various commands by executing the **man** command with the specified command as an argument.



## Rmdir:

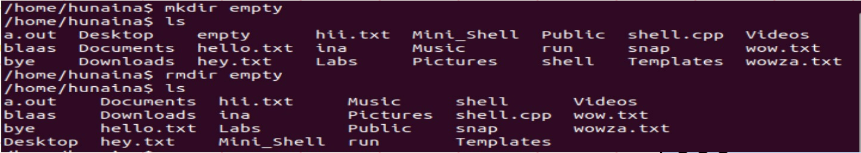
The **RmdirCommand** function is responsible for removing a directory in the file system. It takes a **directory** as input and constructs a string **removeCommand** by appending the provided directory to the string "rmdir ".

The constructed command is then executed using the **system** function. The **system** function allows the execution of a command in the underlying operating system's shell.

By invoking the **rmdir** command with the specified **directory**, the function attempts to remove the specified directory from the file system. If the directory removal is successful, the **system** function returns a status of 0. If the removal fails, the returned status will be non-zero.

The function checks the status returned by the **system** function. If the status is not 0, indicating a failure to remove the directory, an error message is displayed to the user using **cout**.

In summary, the **RmdirCommand** function facilitates the removal of a directory by executing the **rmdir** command with the specified directory as an argument. It provides feedback to the user in case of any errors during the directory removal process.



## Cat:

The **CatCommand** function is responsible for displaying the contents of a file to the console. It takes a **filename** as input and attempts to open the file using **fopen** in read mode.

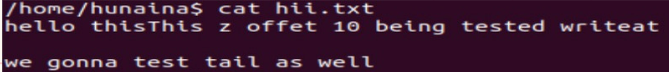
If the file cannot be opened (i.e., **file** is **NULL**), an error message is displayed using **cout**, indicating a failure to open the file. The function then returns, terminating further execution.

If the file is successfully opened, the function enters a loop where it reads each line from the file using **fgets** and stores it in the **line** array. The maximum length of each line is set to 1000 characters.

Within the loop, each line is printed to the console using **cout**, allowing the file contents to be displayed line by line.

Once all lines have been read and printed, the file is closed using **fclose** to release any resources associated with it.

In summary, the **CatCommand** function opens a file, reads its contents line by line, and prints each line to the console. It provides feedback to the user if there is an error opening the file and ensures proper resource management by closing the file after reading.



## Write:

The **WriteCommand** function is designed to allow the user to write content into a file. It takes a **filename** as input and creates an **ofstream** object named **outfile** to handle the file operations.

The function checks if the **outfile** object was successfully created, indicating that the file was successfully opened. If the file opening is successful, the function enters a loop where it reads each line of input from the user using **getline** and stores it in the **line** string.

The loop continues until the user enters **":wq"** to indicate they are done writing and want to save and exit. If **":wq"** is entered, the loop is terminated using the **break** statement.

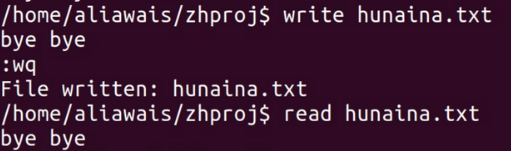
Inside the loop, each line of input is written to the file using the **outfile** object and the **<<** operator. A newline character (**endl**) is added after each line to separate them.

Once the user has finished writing and exits the loop, the **outfile** is closed using the **close** method to save the changes and release any associated resources.

Finally, a success message is displayed using **cout**, indicating that the file has been written with the provided **filename**.

If the **outfile** object could not be created, indicating a failure to open the file, an error message is displayed using **cout**.

In summary, the **WriteCommand** function creates or opens a file, allows the user to enter content until they indicate they are done, writes the content into the file, saves it, and displays a success message. It provides error handling if the file cannot be opened.



## WriteAt:

The **WriteAtCommand** function allows writing to a file at a specified offset position. It takes a **filename** and an **offset** as inputs.

The function creates an **fstream** object named **file** to handle both input and output file operations. It opens the file with the specified **filename** in both input and output modes, indicated by the **ios::in | ios::out** flags.

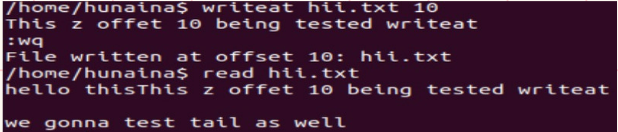
The function checks if the **file** object was successfully created, indicating that the file was successfully opened. If the file opening is successful, the function uses the **seekp** function to set the write position in the file to the specified **offset**. This allows writing to the file starting from the desired position.

The function enters a loop where it reads each line of input from the user using **getline** and stores it in the **line** string. The loop continues until the user enters the **:wq** command, which signifies the end of input and breaks the loop. Within the loop, each line is written to the file using the output operator **<<**, which appends the line to the existing content of the file. The **endl** manipulator is used to insert a newline after each line.

After writing the lines to the file, the **file** is closed using the **close** method to release any associated resources.

If the **file** object could not be created, indicating a failure to open the file, an error message is displayed using **cout**.

In summary, the **WriteAtCommand** function opens a file for both input and output, sets the write position to the specified offset, and then allows the user to input lines of text to be written to the file. It provides error handling if the file cannot be opened. The function terminates input and saves the changes when the user enters the **:wq** command.



## Read:

The **ReadCommand** function reads and displays the content of a file. It takes a **filename** as input.

The function creates an **ifstream** object named **file** to handle input file operations. It opens the file with the specified **filename**.

The function checks if the **file** object was successfully created, indicating that the file was successfully opened. If the file opening is successful, the function enters a loop where it reads each line of the file using **getline** and stores it in the **line** string.

Inside the loop, each line of the file is displayed using **cout**, which prints the line to the console. A newline character (**endl**) is added after each line to separate them.

Once the loop has read all the lines in the file, the **file** is closed using the **close** method to release any associated resources.

If the **file** object could not be created, indicating a failure to open the file, an error message is displayed using **cout**.

In summary, the **ReadCommand** function opens a file for input, reads its content line by line, displays the content to the console, and then closes the file. It provides error handling if the file cannot be opened.



## ReadAt:

The **ReadAtCommand** function reads and displays the contents of a file starting from a specified **offset** position. It takes a **filename** and an **offset** as inputs.

The function creates an **ifstream** object named **file** to handle input file operations. It opens the file with the specified **filename**.

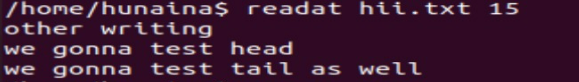
The function checks if the **file** object was successfully created, indicating that the file was successfully opened. If the file opening is successful, the function uses the **seekg** function to set the read position in the file to the specified **offset**. This allows reading the file starting from the desired position.

The function enters a loop where it reads each line of the file using **getline** and stores it in the **line** string. The loop continues until the end of the file is reached or there are no more lines to read. Within the loop, each line is displayed on the console using **cout**.

After reading all the lines from the file, the **file** is closed using the **close** method to release any associated resources.

If the **file** object could not be created, indicating a failure to open the file, an error message is displayed using **cout**.

In summary, the **ReadAtCommand** function opens a file for input, sets the read position to the specified offset, and then reads and displays the contents of the file from that position onwards. It provides error handling if the file cannot be opened.



## Head:

The **HeadCommand** function reads and displays the first **lines** number of lines from a file. It takes a **filename** and the number of **lines** as inputs.

The function creates an **ifstream** object named **file** to handle input file operations. It opens the file with the specified **filename**.

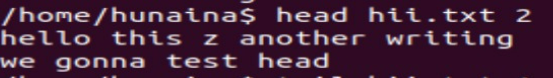
The function checks if the **file** object was successfully created, indicating that the file was successfully opened. If the file opening is successful, the function enters a loop where it reads each line of the file using **getline** and stores it in the **line** string.

Inside the loop, each line of the file is displayed using **cout**, which prints the line to the console. A newline character (**endl**) is added after each line to separate them. The **count** variable keeps track of the number of lines that have been read, and the loop continues as long as both **getline** and the line count are within the specified limits.

Once the loop has read the desired number of lines or reached the end of the file, the **file** is closed using the **close** method to release any associated resources.

If the **file** object could not be created, indicating a failure to open the file, an error message is displayed using **cout**.

In summary, the **HeadCommand** function opens a file for input, reads and displays the specified number of lines from the beginning of the file, and then closes the file. It provides error handling if the file cannot be opened



## Tail:

The **TailCommand** function reads and displays the last **lines** number of lines from a file. It takes a **filename** and the number of **lines** as inputs.

The function creates an **ifstream** object named **file** to handle input file operations. It opens the file with the specified **filename**.

The function checks if the **file** object was successfully created, indicating that the file was successfully opened. If the file opening is successful, the function initializes an empty vector called **buffer** to store the lines read from the file.

The function enters a loop where it reads each line of the file using **getline** and stores it in the **line** string. Within the loop, each line is added to the **buffer** vector using the **push\_back** method. If the size of the **buffer** exceeds the specified number of **lines**, the oldest line is removed from the beginning of the **buffer** using the **erase** method with **buffer.begin()**.

After reading all the lines from the file, the function enters another loop that iterates over the **buffer** vector and displays each line using **cout**. Each line is printed to the console followed by a newline character (**endl**) to separate them.

Once the lines have been displayed, the **file** is closed using the **close** method to release any associated resources.

If the **file** object could not be created, indicating a failure to open the file, an error message is displayed using **cout**.

In summary, the **TailCommand** function opens a file for input, reads and stores the last specified number of lines from the file in a vector, and then displays those lines. It provides error handling if the file cannot be opened.



# User Interaction:

The user interacts with the CLI application by entering commands at the command prompt. The application displays the current working directory as the prompt, allowing users to navigate the file system and perform operations accordingly. The output of executed commands is displayed in the terminal window.

# Future Improvements:

While the CLI application provides a functional interface for system interaction, there are several areas for potential improvement. Some suggestions for future enhancements include:

* Adding support for more commands and functionalities, such as file/directory renaming, file permission management, sorting files, etc.
* Implementing command history and command auto-completion features to improve usability.
* Enhancing error handling and providing informative error messages for better user feedback.
* Implementing additional input validation and sanitization to prevent potential security vulnerabilities.
* Adding support for command-line arguments and options to customize the behavior of commands.
* Providing a more user-friendly and intuitive help system.

# Conclusion:

In conclusion, the CLI application developed in C++ provides a versatile and efficient means of interacting with the system through a command-line interface. The application offers a range of commands for file manipulation, directory navigation, and system operations. While the current implementation provides a solid foundation, there is room for further improvement and expansion to enhance functionality and user experience. The CLI application demonstrates the power and flexibility of C++ in building robust command-line tools.