Report on Command-Line Interface (CLI) Shell Implementation in C

Your Name
Your Institution
Your Email

***Abstract*—This C program implements a simple command-line interface (CLI) shell for Windows, similar to popular Unix/Linux shells such as Bash. The shell processes user input, manages command history, changes directories, lists files, and performs basic file system operations. The primary objective of this shell is to allow users to execute basic commands (cd, ls, pwd, echo, hist) and interact with the Windows operating system through a terminal interface.**

***Index Terms*—CLI, shell, C, Windows, command history, directory navigation, file listing**

I. Introduction

This document presents a report on the implementation of a simple command-line interface (CLI) shell in C for the Windows operating system. The shell provides basic functionality such as command history, directory navigation, file listing, and the execution of simple commands like cd, ls, pwd, echo, and hist. The primary objective of this project is to familiarize oneself with the fundamental concepts of shell programming and interacting with the Windows API.

II. Program Structure and Components

A. Libraries and Macros

The program utilizes various standard C libraries and defines several macros for enhanced code organization. The libraries included are:

- `stdio.h`: Provides standard input/output functions.
- `stdlib.h`: Offers standard library functions such as `getenv()` and `strtok()`.
- `string.h`: Provides functions for string manipulation, including `strncpy()` and `strcmp()`.
- `windows.h`: Contains Windows-specific system calls, like `GetComputerName()` and `FindFirstFile()`.
- `direct.h`: Includes directory-related functions such as `_getcwd()` and `_chdir()`.
- `limits.h`: Defines constants like `PATH_MAX` (maximum allowable path length).

The program defines the following macros:

- `MAX_INPUT`: Sets the maximum size for user input (1024 characters).
- `MAX_ARGS`: Determines the maximum number of command arguments (64).
- `MAX_HISTORY`: Defines the size of the command history buffer (10).

B. Command History Management

The program maintains a history of the last 10 commands executed by the user. This functionality is implemented using global variables and functions:

- **Global Variables:**
  - `history[MAX_HISTORY][MAX_INPUT]`: A 2-dimensional array storing the history of the last 10 commands.
  - `history_count`: Keeps track of the current number of commands in the history.
- **Functions:**
  - `add_to_history(const char *command)`: Adds the latest command to the history. If the history buffer is full, the oldest command is removed, and the new command is added.
  - `show_history()`: Displays the command history to the user.

C. Directory Navigation

The shell provides functionality for changing and printing the current directory. This is accomplished through the following functions:

- `change_directory(char *path, char *prev_dir)`: Changes the current working directory. If no path is provided or the user specifies " ", it defaults to the user's home directory. If the user enters "-", the program switches to the previous directory.
- `print_working_directory()`: Prints the current working directory using `_getcwd()`.

D. File Listing

The `list_files()` function lists the files in the current directory using Windows API functions. It leverages `FindFirstFile()` and `FindNextFile()` to traverse files and directories in the current location.

E. Command Parsing and Execution

The shell reads user input, tokenizes the command, and executes appropriate actions. The input parsing process is as follows:

1) The shell continuously waits for user input via `fgets()`.
2) The input is split into tokens using `strtok()`, and each token is stored in the `args` array.

3) Commands are matched using `strcmp()` and executed based on a set of predefined commands.

The following commands are supported:

- `cd`: Changes the current directory. Handles special cases like `cd -` (switch to the previous directory) and `cd` (switch to the home directory).
- `ls`: Lists files in the current directory.
- `pwd`: Prints the current working directory.
- `echo`: Prints the provided arguments to the terminal.
- `hist`: Displays the command history.
- `exit`: Exits the shell program.

F. Command Execution Workflow

The command execution workflow is as follows:

1) **Prompt Display**: The shell prints a prompt in the format `[username@hostname cwd]$` where:
   - `username`: Retrieved from the environment variable `USERNAME`.
   - `hostname`: Obtained using the Windows function `GetComputerName()`.
   - `cwd`: The current working directory, fetched using `_getcwd()`.
2) **Input Handling**:
   - The shell reads user input and processes it with `fgets()`.
   - If the command is not empty, it is added to the history using `add_to_history()`.
3) **Command Execution**:
   - The shell tokenizes the input into commands and arguments.
   - It checks for each supported command (cd, ls, pwd, echo, etc.) and invokes the corresponding function.
   - If an unsupported command is entered, the shell prints an error message.
4) **Looping**: The program continuously loops until the user types `exit`.

III. Error Handling

The shell incorporates basic error handling mechanisms to ensure robust operation. It handles errors such as:

- If `getcwd()` or `chdir()` fails, it prints an appropriate error message.
- If too many arguments are passed to the `cd` command, the program prints an error and ignores the command.
- The `FindFirstFile()` function handles cases where the directory cannot be read.

IV. Key Functions

- `change_directory(char *path, char *prev_dir)`: Handles changing directories and supports navigating to the home directory ( ) and switching to the previous directory (-).
- `list_files()`: Uses Windows API functions to list files and directories in the current location.
- `echo_command(char *input)`: This function prints the users input to the terminal, implementing an echo command.

V. Conclusion

This report has presented a simple command-line shell implemented in C for the Windows operating system. The shell successfully integrates essential features like command history, directory navigation, file listing, and the execution of basic commands. The use of Windows API functions enables interaction with the file system, while standard C functions handle input/output and string manipulation.

The shell could be further enhanced by adding support for more commands and features such as piping, file redirection, or job control to make it more powerful and versatile.