**Project 3: Improved UNIX/LINUX Command Line Interpreter**

Duc Vo, Ty Gehrke

Grand Canyon University

CST-315: Operating Systems Lecture and Lab

Professor Citro

February 24th, 2024

**Team Members**

- Ty Gehrke

- Duc Vo

**Project Description**

The Improved Unix/Linux Command Line Interpreter (CLC) project represents a significant enhancement to a foundational command-line tool developed previously. Recognizing the importance of solidifying our understanding of UNIX/Linux environments, shell scripting, and C programming, this project aims to refine and expand upon the capabilities of our existing shell interpreter. Through careful consideration of feedback and exploration of advanced features, we have crafted a more versatile command-line experience.

In this project, we prioritize the incorporation of new features, improvements in programming style, and enhancements to the overall design and approach. Our objective is to create a user-friendly and efficient shell interpreter that not only executes commands reliably but also offers a seamless and customizable interaction environment.

**Theoretical Background**

The code for the Improved Unix/Linux Command Line Interpreter (CLC) is grounded in several key theoretical concepts related to operating systems, process management, and shell scripting. Understanding these concepts provides the necessary foundation for comprehending the design and functionality of the interpreter.

1. Operating Systems Fundamentals:
   1. At the core of the interpreter lies an understanding of operating system principles. This includes concepts such as process management, file systems, and system calls. Operating systems coordinate the hardware and software resources of a computer system, and a shell interpreter interacts closely with the operating system to execute commands and manage processes.
2. Process Management:
   1. Process management is central to the operation of the shell interpreter. Processes are instances of executing programs, and the interpreter creates child processes to execute user commands. Process creation, forking, and execution are fundamental to understanding how the interpreter interacts with the operating system to execute commands.
3. Shell Scripting:
   1. Shell scripting involves writing scripts or programs to automate tasks using a shell, such as Bash on Unix/Linux systems. The interpreter utilizes shell scripting techniques to parse user input, execute commands, and manage the shell environment. Understanding shell scripting concepts, including command parsing, input/output redirection, and command execution, is crucial for implementing the interpreter's functionality.
4. Input/Output Handling:
   1. Input/output (I/O) handling is essential for interacting with users and executing commands. The interpreter reads user input from the command line, processes it, and displays output to the user. Additionally, the interpreter supports input/output redirection and piping, allowing users to manipulate the flow of data between commands.
5. System Calls:
   1. System calls are the interface between user programs and the operating system kernel. The interpreter utilizes system calls to perform operations such as process creation, file I/O, and process synchronization. Understanding system calls and their parameters is essential for implementing the core functionality of the interpreter.

**Implementation Approach**

The implementation of the Improved Unix/Linux Command Line Interpreter (CLC) involved a systematic and iterative approach, focusing on enhancing functionality, improving programming style, and refining overall design. The following steps outline the approach taken to develop the interpreter:

1. Requirements Analysis:
   1. The initial phase involved a thorough analysis of requirements, including feedback from the previous project and instructor guidance. Key requirements included adding new features, improving programming style, and enhancing the overall user experience.
2. Feature Identification:
   1. Based on the requirements analysis, a list of new features and enhancements was identified. These features included color customization for the command prompt, command history functionality, a help command, batch mode execution, and improved error handling.
3. Design Planning:
   1. With the features identified, the next step was to plan the design and architecture of the interpreter. This included determining the structure of the main program, defining data structures for storing command history, and outlining the logic for executing commands and handling user input.
4. Coding and Iterative Development:
   1. The coding phase involved implementing the planned design in C++, adhering to best practices in coding standards and style. Each feature was developed incrementally, with regular testing and debugging to ensure correctness and reliability. Code was structured into functions and modules to promote modularity and ease of maintenance.
5. Testing and Validation:
   1. Throughout the development process, extensive testing and validation were conducted to verify the correctness and robustness of the interpreter. This included unit testing of individual functions, integration testing of feature interactions, and validation against test cases and user scenarios.
6. Refinement and Optimization:
   1. As the implementation progressed, ongoing refinement and optimization were performed to improve performance, readability, and user experience. This involved revisiting code sections for potential optimizations, enhancing error handling and error messages, and refining user interface elements.
7. Documentation and Commenting:
   1. Concurrent with development, comprehensive documentation and code commenting were maintained to provide clear explanations of functionality, algorithms, and code structure. This documentation served as a reference for understanding the codebase and facilitating future maintenance and extension.

**Code & Screenshots**

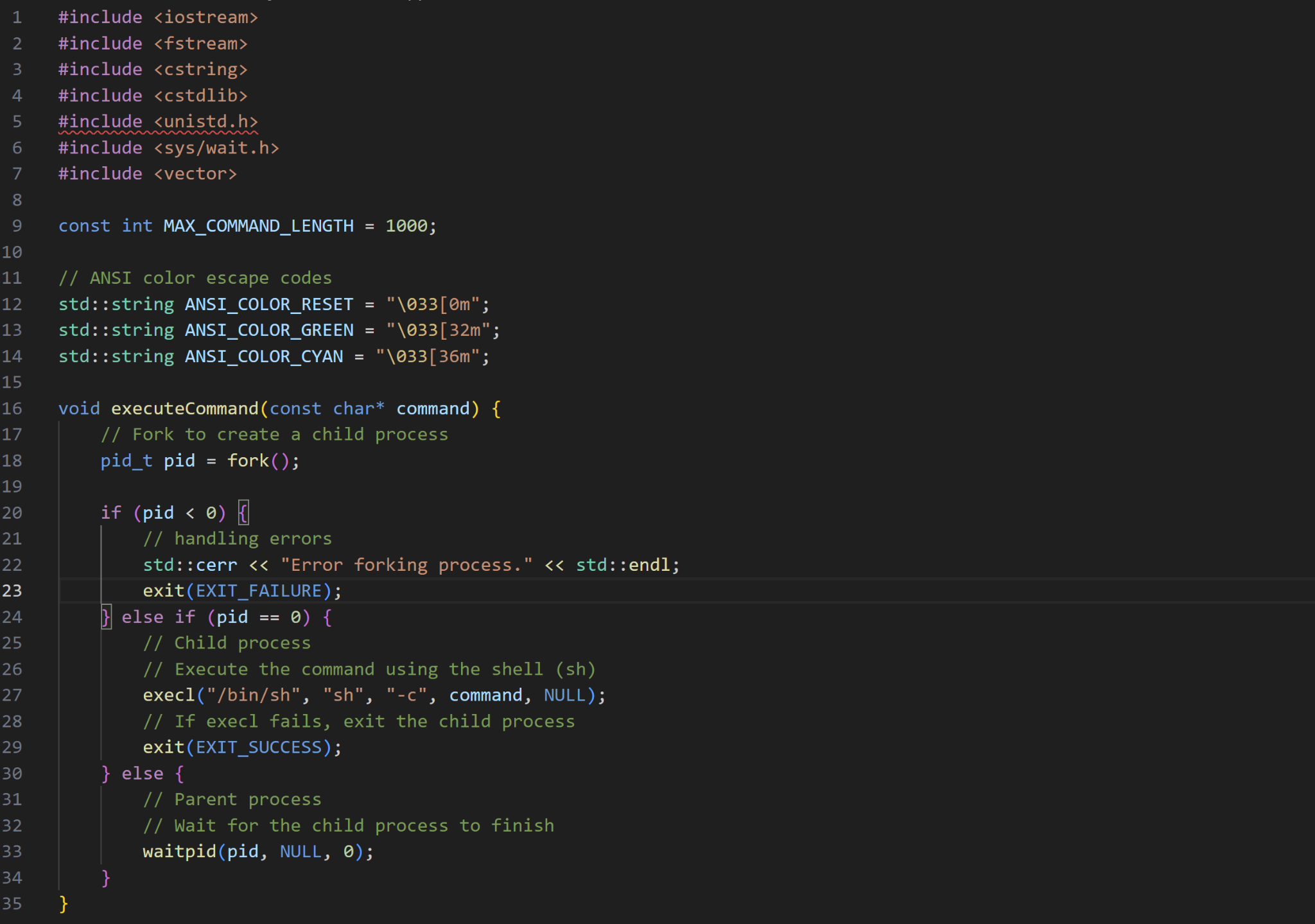
The algorithm for parsing and processing shell commands in the Improved Unix/Linux Command Line Interpreter (CLC) underwent significant enhancements to improve efficiency, robustness, and flexibility. The following description outlines the key aspects of the improved algorithm:

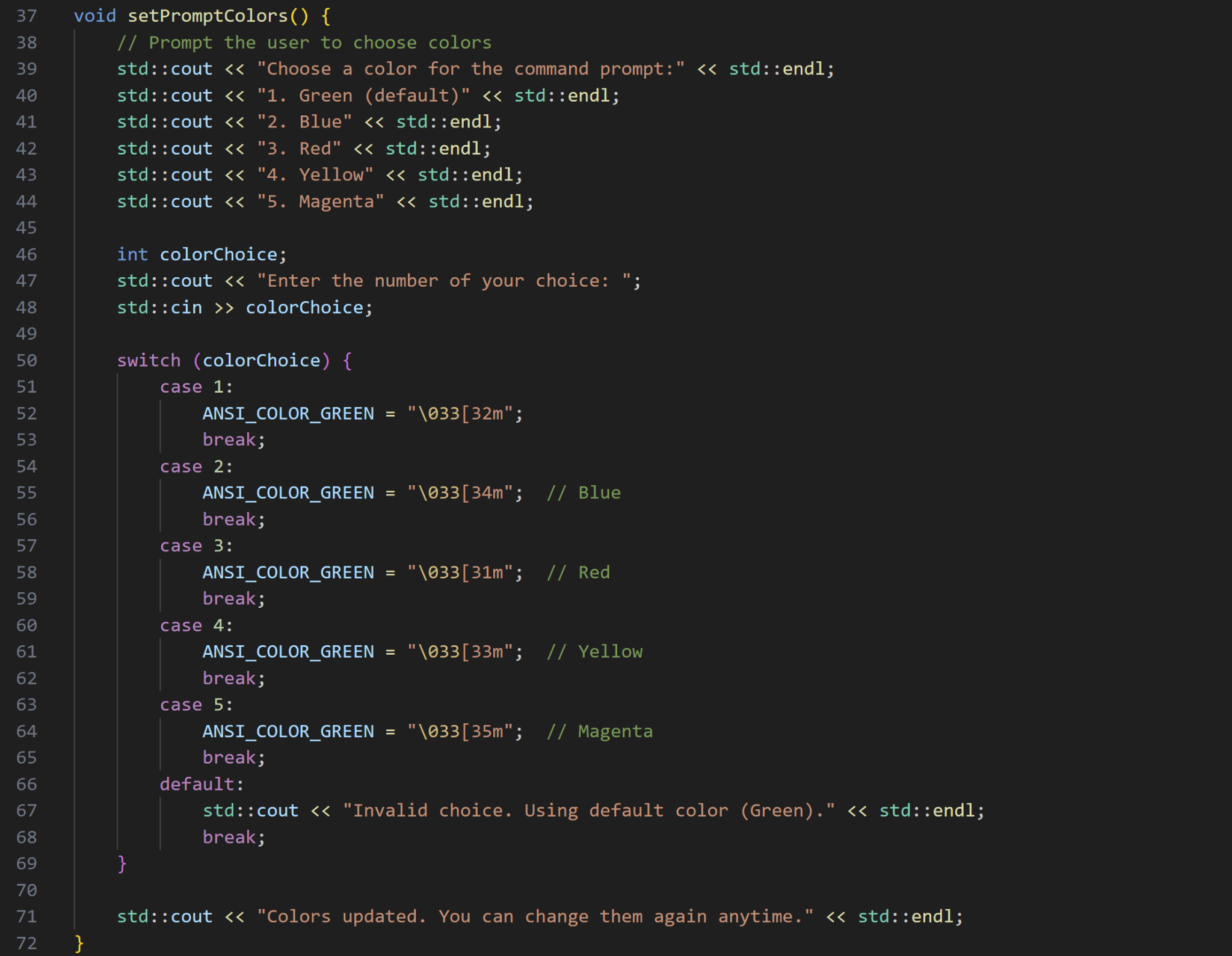
1. Input Parsing:
   1. The algorithm begins by parsing user input obtained from the command line. Input parsing involves breaking down the user-provided command into individual tokens or segments, separating commands, arguments, and special characters such as semicolons (;) or exclamation marks (!). This process ensures that each component of the command is correctly identified and processed.
2. Command Execution:
   1. Once the input is parsed, the algorithm proceeds to execute the parsed commands. For each parsed command, the interpreter determines the appropriate action to take, which may include executing a single command, handling command history references, or executing multiple commands separated by semicolons. The algorithm ensures that each command is executed in the correct sequence and with the necessary parameters.
3. Process Creation and Management:
   1. During command execution, the algorithm creates and manages child processes to execute the commands. This involves using system calls such as fork() to create a new process, exec() to replace the current process image with a new one, and waitpid() to wait for child processes to complete execution. Process management ensures that commands are executed asynchronously and that the interpreter remains responsive during command execution.
4. Error Handling:
   1. The improved algorithm includes robust error handling mechanisms to detect and handle errors that may occur during command parsing and execution. This includes checking for syntax errors in user input, handling errors arising from system calls, and providing informative error messages to the user. Error handling ensures that the interpreter gracefully handles unexpected situations and maintains stability and reliability.
5. Command History Management:
   1. An essential aspect of the algorithm is the management of command history. The interpreter maintains a history of previously executed commands, allowing users to recall and re-execute commands using history references (e.g., !n). The algorithm ensures efficient storage and retrieval of command history entries and handles user requests to access and manipulate command history effectively.
6. Interactive and Batch Mode Support:
   1. The algorithm supports both interactive and batch mode operation. In interactive mode, the interpreter reads user input from the command line, processes it in real-time, and displays output interactively. In batch mode, the interpreter reads commands from a batch file, executes them sequentially, and displays the output accordingly. The algorithm adapts to the mode of operation dynamically, providing a seamless user experience.

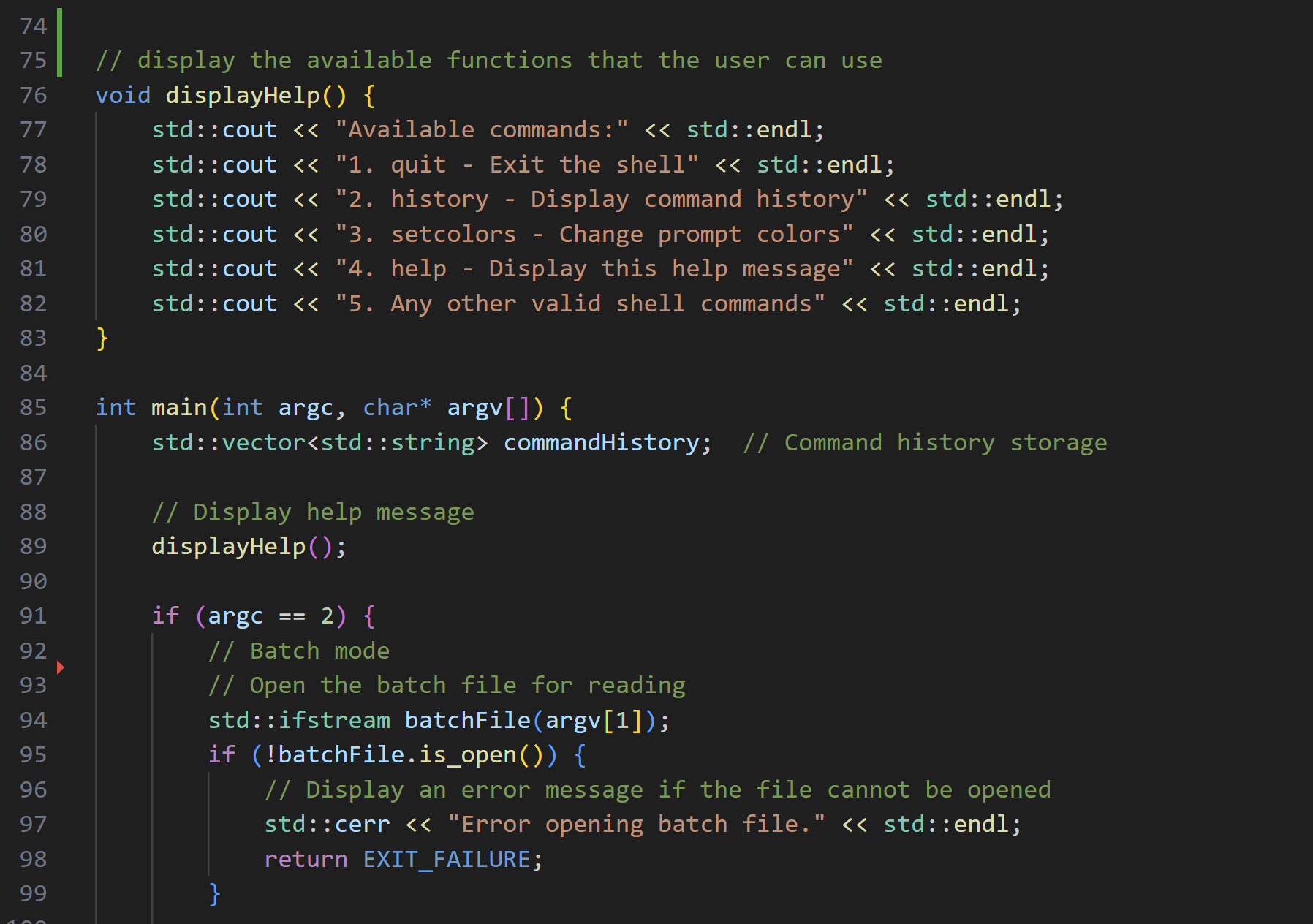
**Function Explanations**

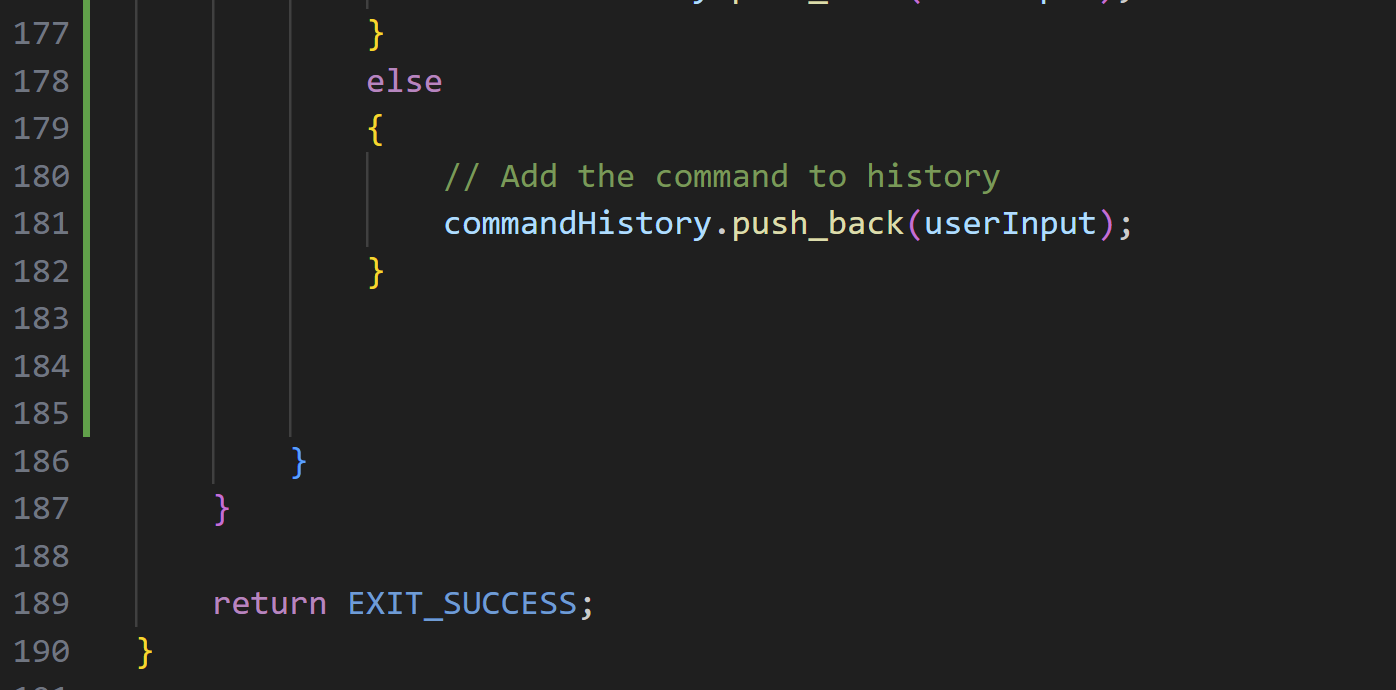
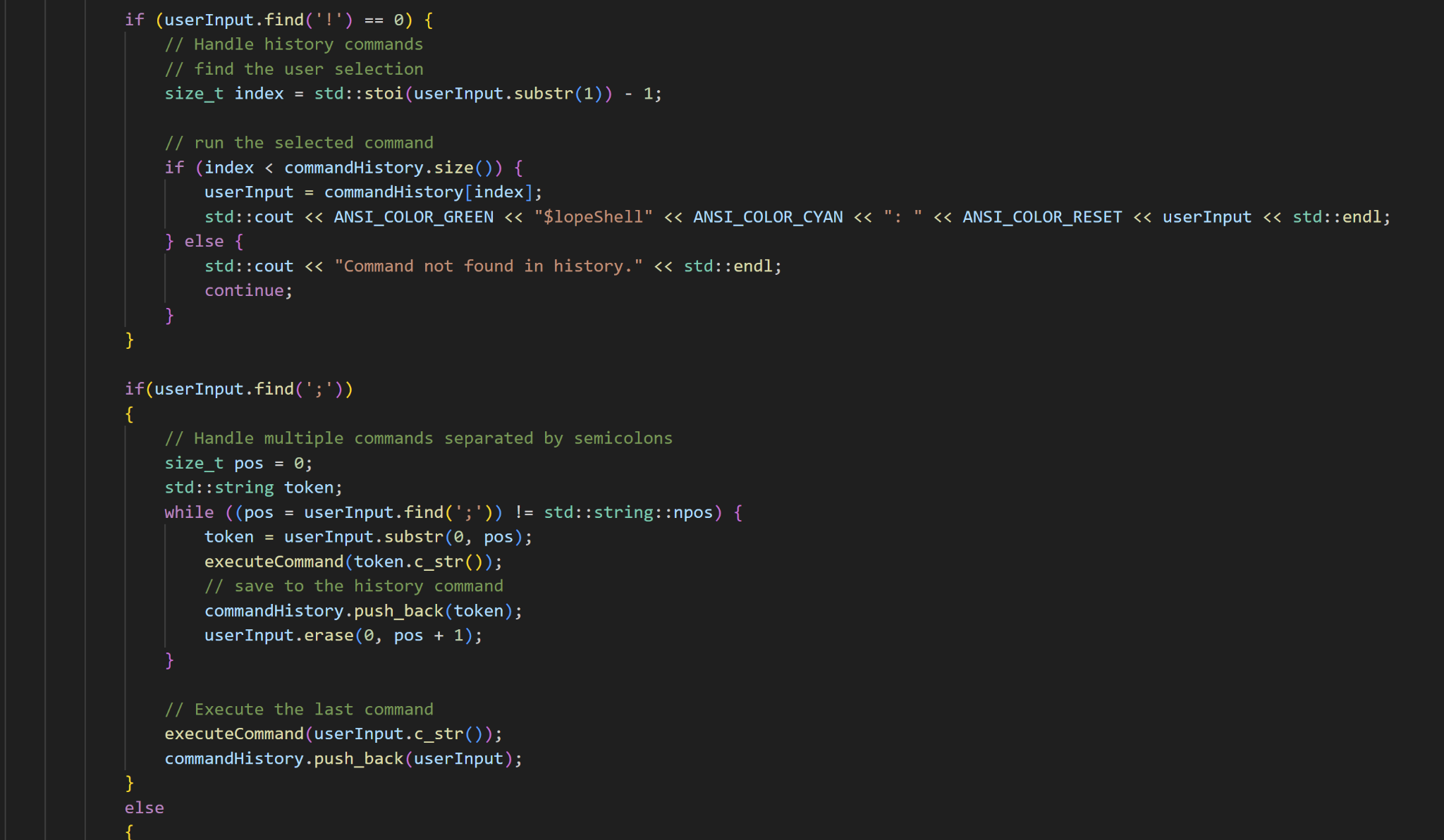
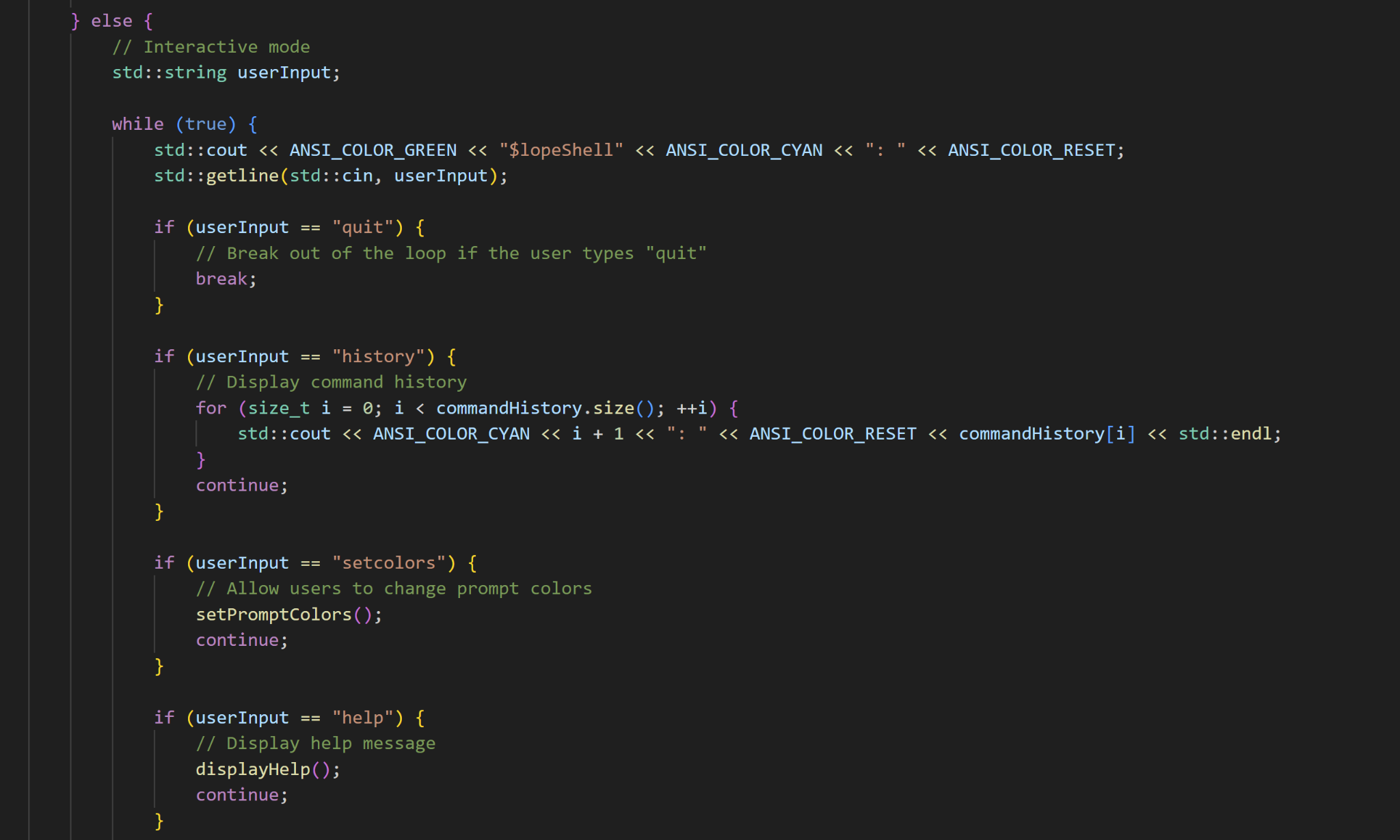
* executeCommand(const char\* command):
  + This function is responsible for executing a given command.
  + It forks a child process using fork() to execute the command asynchronously.
  + In the child process, it uses execl() to execute the command via the shell (/bin/sh).
  + The parent process waits for the child process to finish using waitpid().
* setPromptColors():
  + This function allows the user to customize the color of the command prompt.
  + It displays a menu of color choices and prompts the user to enter their selection.
  + Based on the user's choice, it updates the ANSI color escape codes used for the command prompt.
* displayHelp():
  + This function displays a help message listing available commands and their descriptions.
  + It provides guidance to the user on how to interact with the shell and utilize its features.
* main(int argc, char\* argv[]):
  + The main entry point of the interpreter.
  + It initializes variables, sets up command history storage, and displays the help message.
  + Depending on the presence of command-line arguments (argc), it either enters interactive mode or batch mode.
  + In interactive mode, it continuously prompts the user for input, executes commands, and handles special commands like quit, history, setcolors, and help.
  + In batch mode, it reads commands from a batch file, executes them sequentially, and displays the output.
  + It also handles error conditions and ensures proper termination of the interpreter.

**Screenshots**

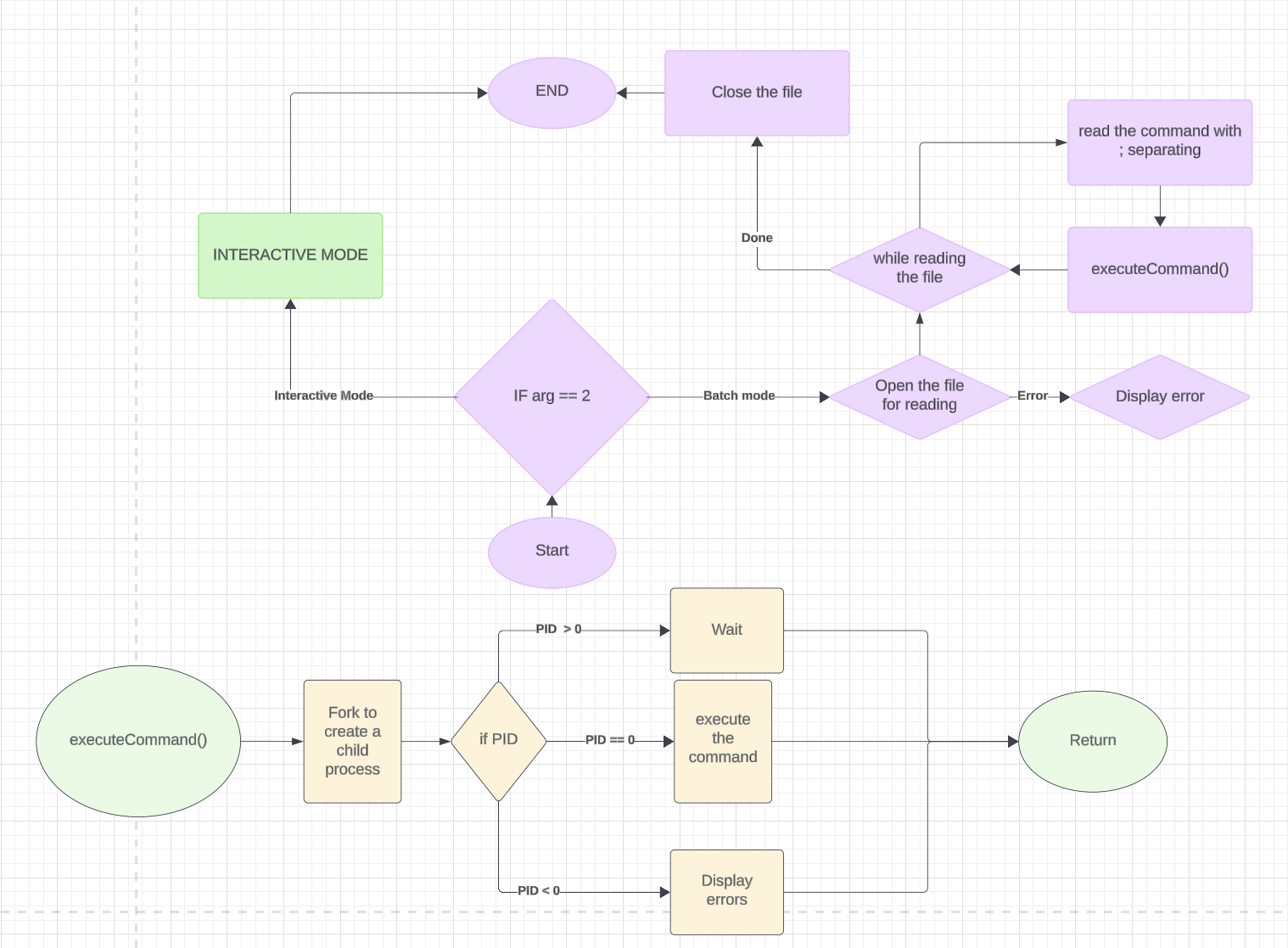
****

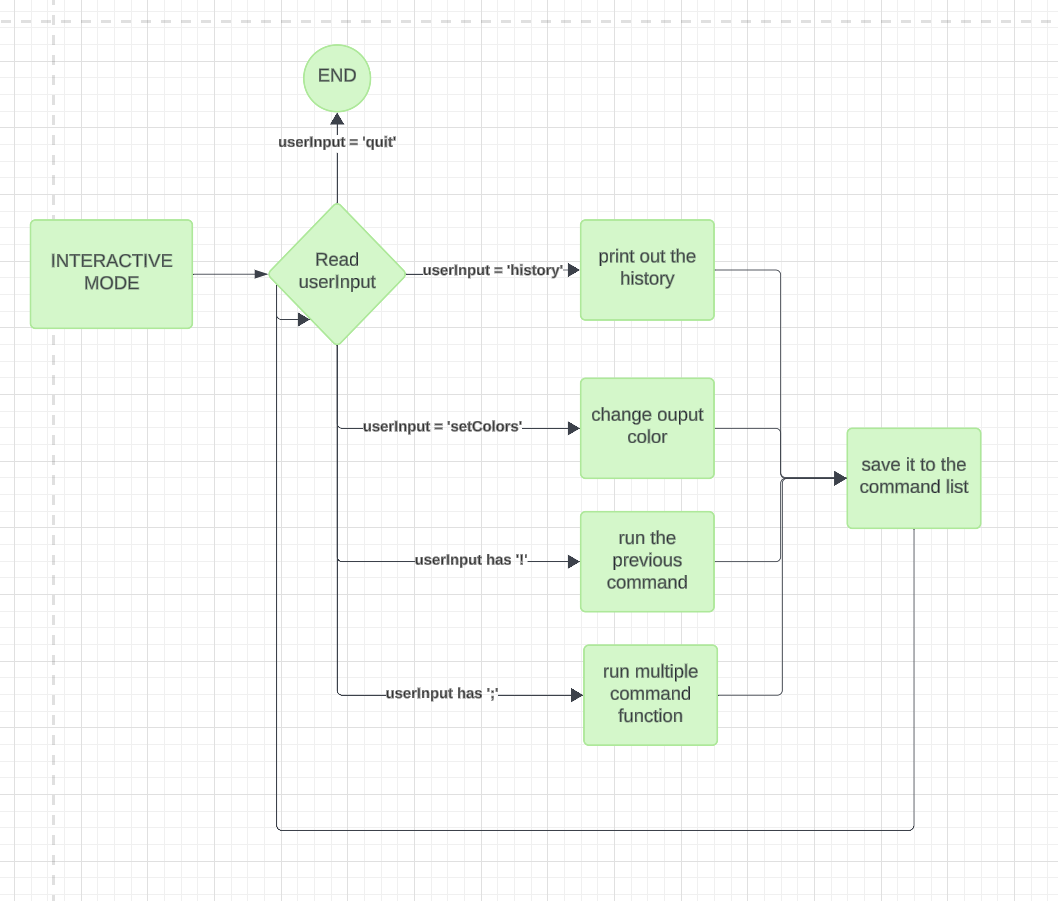
****

****

****

**Flowchart**





**References**

Ali, S. (2023, May 7). *“Creating your own command interpreter: Main concepts guide: Simple*

*shell - part 2.”* “Creating Your Own Command Interpreter: Main Concepts Guide: Simple Shell - Part 2.” https://shazaali.substack.com/p/creating-your-own-command-interpreter-5c7

Fisher, T. (2023, September 22). *What is a command line interpreter?*. Lifewire.

https://www.lifewire.com/what-is-a-command-line-interpreter-2625827

*UPCASE : Let’s build a CLI*. Command Line Interface Tutorial by thoughtbot. (n.d.).

https://thoughtbot.com/upcase/videos/lets-build-a-cli