Custom CLI

Generated by Doxygen 1.10.0

Chapter 1

Coding Standards

1.1 Introduction

This document outlines the coding standards and guidelines that are followed in the project. Adhering to these standards ensures consistency, readability, and maintainability of the codebase.

1.2 Naming Conventions

Naming conventions should follow:

- Functions: `functionsLikeThis()`
- Variables: `variables_like_this`
- Constants: `CONSTANTS_LIKE_THIS`
- Enums: `EnumLikeThis`
- Structs: `StructsLikeThis`
- Avoid using all uppercase names except for constants.

1.3 Formatting Guidelines

- Indentation: Use 2 spaces for indentation, not tabs.
- Line Length: Keep lines limited to a maximum of 80 characters.
- Braces: Place opening braces on the same line as the control statement.
- Comments: Use descriptive comments to explain complex code sections. Follow a consistent commenting style.

1.4 Coding Practices

- Follow the `Google C++ Style Guide` for general formatting and style.
- Use meaningful variable and function names.
- Write clear and concise code. Avoid unnecessary complexity.

1.5 Version Control

- Use Git for version control to manage the codebase effectively.
- Follow the Git branching model:
  - Create a new branch for each feature or bug fix.
  - Branch off from the `main` branch for development.
  - Name the branches descriptively, reflecting the feature or bug being addressed.
  - Avoid committing directly to the `main` branch.
  - Regularly merge feature branches back into `main` after thorough testing.

1.6 Documentation

- Use Doxygen for documenting code to automatically generate API documentation.
  - Follow the `Doxygen` syntax for comments to document functions, variables, and code blocks.
  - Include brief descriptions, parameter descriptions, return value descriptions, and example usage in the comments.
  - Make sure to provide descriptive and meaningful documentation to enhance code understanding.
  - Use Doxygen tags such as `@brief`, `@param`, `@return`, `@example`, etc., to structure the comments properly.
- Set yourself as the author in the file header comments if you create new files or significant sections of code.
- If you modify an existing file, add a `@section Modifications` below the file header comments and provide details of the changes made.
  - Include a brief description of the modifications and specify the date and your name.
  - Here's an example of how to document modifications:

1.7 Conclusion

Adhering to these coding standards is essential for maintaining a high-quality codebase. By following these guidelines, we ensure consistency across the project and make it easier for developers to understand, contribute to, and maintain the code.

Chapter 2

UNIX-CLI

2.1 Overview

This project involves implementing a set of commands for file manipulation and creating a custom command-line interpreter. The project is structured with specific requirements and constraints to challenge and improve comprehension of low-level system calls to the Linux operating system and process management.

2.2 Features

2.2.1 File manipulation commands:

- Show file contents
- Copy files
- Append contents of one file to another
- Count lines in a file
- Delete files
- Display filesystem information for a file
- List files and directories in a specified or current directory

2.2.2 Custom command-line interpreter:

- Read and execute user-entered commands
- Indicate readiness with "%" symbol
- Execute commands using process execution primitives
- Suspend interpreter until command completion
- Support sequential execution of multiple commands
- Terminate interpreter with a special command

2.3 Compilation

To compile, run the `make` command. This command will compile both the custom commands and the command-line interpreter (CLI) together. Alternatively, if you wish to compile only the CLI, you can execute `make cli`. Similarly, to compile only the custom commands, use `make commands`.

```
# Compile both the CLI and custom commands
make

# Compile only the CLI
make cli

# Compile only the custom commands
make commands
```

Upon successful compilation, the compiled program and the different commands will be placed inside the `build` folder for easy access and execution.

2.4 Usage

1. **Add to PATH:** Ensure that the folder containing the compiled commands is added to your system's PATH variable. This step allows the system to recognize and execute the custom commands. You can achieve this by appending the directory path to your PATH variable in your shell configuration file (e.g., `~/.bashrc`, `~/.bash_profile`, `~/.zshrc`).
2. **Launch UNIX-CLI:** Once the directory is added to the PATH, launch the UNIX-CLI executable file located inside the `build` folder. You'll be greeted with a command-line interface similar to the standard Bash shell, denoted by the "%" symbol indicating readiness for command input.
3. **Enjoy:** Enjoy the functionality of UNIX-CLI for file manipulation and command execution.

2.5 Commands

Some of the available commands include:

- `acrescenta` - allows you to append content from one file to another.
- `apaga` - allows you to delete a file.
- `conta` - allows you to count the number of lines in a file.
- `copia` - allows you to copy a file.
- `informa` - gives information about a file.
- `lista` - lists all files and directories under a given (or current by default) directory
- `mostra` - displays content of a file.

2.6 Conclusion

In conclusion, the project has been a valuable learning experience, providing hands-on exploration of low-level system calls and process management in the Linux environment. Through the implementation of essential file manipulation commands and a custom command-line interpreter, we have gained a deeper understanding of how the operating system interacts with files and processes.

By working on features such as displaying file contents, copying, appending, counting lines, and deleting files, we have honed my skills in file system operations and learned about the underlying mechanisms involved. The development of the custom command-line interpreter has further reinforced our understanding of command execution and process management.

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

File Documentation

4.1 CLI/include/constants.h File Reference

Header file containing constant definitions.

Macros

- #define **PROGRAM_NAME** "UNIX-CLI"
- #define **VERSION** "0.1"
- #define **EXIT_CMD** "termina"
- #define **MAX_ARGS** 64
- #define **BUFFER_SIZE_BYTES** 4096
- #define **FILE_INFO_STR_SIZE** 50

4.1.1 Detailed Description

Header file containing constant definitions.

Author

Enrique Rodrigues (a28602@alunos.ipca.pt)

This header file defines various constants that are used throughout the application. These constants are used to represent specific values or settings that are used in multiple parts of the codebase.

Constants in this file are organized into logical groups based on their purpose or usage. Each constant is given a descriptive name to indicate its meaning or significance.

For example, constants related to file permissions may be grouped together, while constants representing error codes may be in a separate group.

Constants defined in this file are intended to improve code readability, reduce the risk of errors due to magic numbers, and provide a centralized location for managing shared values.

Version

0.1

Date

2024-03-20

Copyright

4.2 constants.h

Go to the documentation of this file.

```
00001
00026 #ifndef CONSTANTS_H
00027 #define CONSTANTS_H
00028
00029 /* GENERAL CONSTANTS */
00030 #define PROGRAM_NAME "UNIX-CLI"  // program name
00031 #define VERSION "0.1"            // current version number
00032 #define EXIT_CMD "termina"       // command to exit CLI
00033 #define MAX_ARGS 64              // maximum number of arguments
00034
00035 /* BUFFERS */
00036 #define BUFFER_SIZE_BYTES 4096  // max buffer size
00037
00038 /* FILE INFORMATION */
00039 #define FILE_INFO_STR_SIZE 50  // size of strings in `FileInfo` structure
00040
00041 #endif /* CONSTANTS_H */
```

4.3 CLI/include/execute.h File Reference

Header file for functions that execute files.

Functions

- void execute_command (const char *command_path, char *args[])

  Executes a command with the given arguments.

4.3.1 Detailed Description

Header file for functions that execute files.

Author

Enrique Rodrigues (a28602@alunos.ipca.pt)

Version

0.1

Date

2024-04-21

Copyright

Copyright (c) 2024

4.3.2 Function Documentation

4.3.2.1 execute_command()

```
void execute_command (
            const char * command_path,
            char * args[] )
```

Executes a command with the given arguments.

This function creates a child process using fork() and executes the specified command with the provided arguments using execvp(). If the fork or execvp operation fails, an error message is printed, and the function returns.

Parameters

command_path	The path to the command to be executed.
args	An array of strings containing the arguments for the command.

4.4 execute.h

Go to the documentation of this file.

```
00001
00012 #ifndef EXECUTE_H
00013 #define EXECUTE_H
00014
00025 void execute_command(const char *command_path, char *args[]);
00026
00027 #endif /* EXECUTE_H */
```

4.5 CLI/include/find.h File Reference

Header file for functions that find and check for executable files.

```
#include <stdbool.h>
```

Functions

- bool is_executable_file (const char ∗path)

  Checks if a file is executable.
- bool find_command_in_path (const char ∗command, char ∗command_path)

  Finds the full path of a command in the PATH environment variable.

4.5.1 Detailed Description

Header file for functions that find and check for executable files.

Author

Enrique Rodrigues (a28602@alunos.ipca.pt)

Version

0.1

Date

2024-04-21

Copyright

Copyright (c) 2024

4.5.2 Function Documentation

4.5.2.1 find_command_in_path()

```
bool find_command_in_path (
            const char * command,
            char * command_path )
```

Finds the full path of a command in the PATH environment variable.

This function searches for the specified command in the directories listed in the PATH environment variable. If the command is found, its full path is copied to the provided buffer.

Parameters

command	The name of the command to search for.
command_path	A buffer to store the full path of the command.

Returns

true if the command is found, false otherwise.

4.5.2.2 is_executable_file()

```
bool is_executable_file (
            const char * path )
```

Checks if a file is executable.

This function checks if the file at the specified path is executable.

Parameters

path	The path to the file.

Returns

true if the file is executable, false otherwise.

4.6 find.h

Go to the documentation of this file.

```
00001
00012 #ifndef FIND_H
00013 #define FIND_H
00014
00015 #include <stdbool.h>
00016
00025 bool is_executable_file(const char *path);
00026
00038 bool find_command_in_path(const char *command, char *command_path);
00039
00040 #endif /* FIND_H */
```

4.7 CLI/include/input_parser.h File Reference

Header file for input parsing functions.

Functions

- int parse_input (char *input, char *args[], int max_args)
  Parses an input string into arguments.

4.7.1 Detailed Description

Header file for input parsing functions.

Author

Enrique Rodrigues (a28602@alunos.ipca.pt)

Version

0.1

Date

2024-04-21

Copyright

4.7.2 Function Documentation

4.7.2.1 parse_input()

```
int parse_input (
            char * input,
            char * args[],
            int max_args )
```

Parses an input string into arguments.

This function tokenizes the input string by spaces and stores the tokens in the provided array of strings (`args`). The maximum number of arguments that can be stored in the `args` array is specified by `max_args`.

Parameters

input	The input string to be parsed.
args	An array of strings to store the parsed arguments.
max_args	The maximum number of arguments that can be stored.

Returns

int The number of arguments parsed and stored in the `args` array.

4.8 input_parser.h

Go to the documentation of this file.

```
00001
00012 #ifndef INPUT_PARSER_H
00013 #define INPUT_PARSER_H
00014
00027 int parse_input(char *input, char *args[], int max_args);
00028
00029 #endif /* INPUT_PARSER_H */
```

4.9 CLI/include/utils.h File Reference

Contains utility declarations.

`#include <stdbool.h>`

Functions

- bool should_exit (const char *input)

  Checks if the input string indicates the program should exit.

4.9.1 Detailed Description

Contains utility declarations.

Author

Enrique Rodrigues (a28602@alunos.ipca.pt)

Version

0.1

Date

2024-04-21

Copyright

Copyright (c) 2024

4.9.2 Function Documentation

4.9.2.1 should_exit()

```
bool should_exit (
            const char * input )
```

Checks if the input string indicates the program should exit.

This function checks if the input string starts with the exit command defined in the constants header file. If the input string matches the exit command, the function returns true; otherwise, it returns false.

Parameters

input	The input string to check.

Returns

true if the input string indicates program exit, false otherwise.

4.10 utils.h

Go to the documentation of this file.

```
00001
00011 #ifndef UTILS_H
00012 #define UTILS_H
00013
00014 #include <stdbool.h>
00015
00026 bool should_exit(const char *input);
00027
00028 #endif /* UTILS_H */
```

4.11 CLI/src/execute.c File Reference

Contains functions for executing external commands.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
```

Functions

- void execute_command (const char *command_path, char *args[])

  Executes a command with the given arguments.

4.11.1 Detailed Description

Contains functions for executing external commands.

Author

Enrique Rodrigues (a28602@alunos.ipca.pt)

Version

0.1

Date

2024-04-21

Copyright

Copyright (c) 2024

4.11.2 Function Documentation

4.11.2.1 execute_command()

```
void execute_command (
            const char * command_path,
            char * args[] )
```

Executes a command with the given arguments.

This function creates a child process using fork() and executes the specified command with the provided arguments using execvp(). If the fork or execvp operation fails, an error message is printed, and the function returns.

Parameters

command_path	The path to the command to be executed.
args	An array of strings containing the arguments for the command.

4.12 CLI/src/find.c File Reference

Contains functions for finding executable files in the PATH.

```
#include <constants.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
```

Macros

- #define **_XOPEN_SOURCE** 700
- #define **_DEFAULT_SOURCE**

Functions

- bool is_executable_file (const char ∗path)

  Checks if a file is executable.
- bool find_command_in_path (const char ∗command, char ∗command_path)

  Finds the full path of a command in the PATH environment variable.

4.12.1 Detailed Description

Contains functions for finding executable files in the PATH.

Author

Enrique Rodrigues (a28602@alunos.ipca.pt)

Version

0.1

Date

2024-04-21

Copyright

4.12.2 Function Documentation

4.12.2.1 find_command_in_path()

```
bool find_command_in_path (
            const char * command,
            char * command_path )
```

Finds the full path of a command in the PATH environment variable.

This function searches for the specified command in the directories listed in the PATH environment variable. If the command is found, its full path is copied to the provided buffer.

Parameters

command	The name of the command to search for.
command_path	A buffer to store the full path of the command.

Returns

true if the command is found, false otherwise.

4.12.2.2 is_executable_file()

```
bool is_executable_file (
            const char * path )
```

Checks if a file is executable.

This function checks if the file at the specified path is executable.

Parameters

path	The path to the file.

Returns

true if the file is executable, false otherwise.

4.13 CLI/src/input_parser.c File Reference

Contains functions for parsing input strings into arguments.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

Functions

- int parse_input (char *input, char *args[], int max_args)
  Parses an input string into arguments.

4.13.1 Detailed Description

Contains functions for parsing input strings into arguments.

Author

Enrique Rodrigues (a28602@alunos.ipca.pt)

Version

0.1

Date

2024-04-21

Copyright

Copyright (c) 2024

4.13.2 Function Documentation

4.13.2.1 parse_input()

```
int parse_input (
            char * input,
            char * args[],
            int max_args )
```

Parses an input string into arguments.

This function tokenizes the input string by spaces and stores the tokens in the provided array of strings (`args`). The maximum number of arguments that can be stored in the `args` array is specified by `max_args`.

Parameters

input	The input string to be parsed.
args	An array of strings to store the parsed arguments.
max_args	The maximum number of arguments that can be stored.

Returns

int The number of arguments parsed and stored in the `args` array.

4.14 CLI/src/main.c File Reference

This file contains the main entry point of the program.

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include "constants.h"
#include "execute.h"
#include "find.h"
#include "input_parser.h"
#include "utils.h"
```

Macros

- #define **_XOPEN_SOURCE** 700

Functions

- int main ()

  Main entry point of the program.

4.14.1 Detailed Description

This file contains the main entry point of the program.

Author

Enrique Rodrigues (a28602@alunos.ipca.pt)

Unix-CLI is a versatile command-line utility featuring a custom command interpreter, allowing users to execute a variety of commands directly from the terminal. With a focus on efficiency and user-friendliness, Unix-CLI utilizes system calls for low-level operations, ensuring broad compatibility across Unix-like operating systems.

Version

0.2

Date

2024-03-20

4.14.2 Modifications

- 2024-04-18: Updated program to v0.2, documented on the Github repo. Enrique George Rodrigues (a28602@alunos.ipca.pt)
- 2024-04-22: CLI tries to execute as a file first and if it fails it looks in the users PATH variable. Enrique George Rodrigues (a28602@alunos.ipca.pt)

4.14.3 Function Documentation

4.14.3.1 main()

```
int main ( )
```

Main entry point of the program.

The main function serves as the entry point of the program. It executes the command-line interface, allowing users to execute commands and programs.

Returns

int Returns 0 upon successful execution or 1 in the case of error.

4.15 CLI/src/utils.c File Reference

Contains utility functions.

```
#include <stdbool.h>
#include <string.h>
#include <unistd.h>
#include "constants.h"
```

Functions

- bool should_exit (const char *input)
  Checks if the input string indicates the program should exit.

4.15.1 Detailed Description

Contains utility functions.

Author

Enrique Rodrigues (a28602@alunos.ipca.pt)

Version

0.1

Date

2024-04-21

Copyright

Copyright (c) 2024

4.15.2 Function Documentation

4.15.2.1 should_exit()

```
bool should_exit (
            const char * input )
```

Checks if the input string indicates the program should exit.

This function checks if the input string starts with the exit command defined in the constants header file. If the input string matches the exit command, the function returns true; otherwise, it returns false.

Parameters

input	The input string to check.

Returns

true if the input string indicates program exit, false otherwise.

Index