# Comprehensive Explanation of the Custom Shell with I/O Redirection

## 1. Header Includes

```cpp

#include <iostream> // For input/output streams

#include <string> // For string manipulation

#include <vector> // For dynamic arrays

#include <algorithm> // For find() algorithm

#include <unistd.h> // POSIX API (fork, exec, etc.)

#include <sys/wait.h> // For waitpid()

#include <sys/stat.h> // For file status

#include <fcntl.h> // For file control options

#include <cstring> // For C string functions

```

These headers provide all the necessary functionality for:

- Basic I/O operations

- String and vector manipulation

- Process management (fork/exec)

- File operations (open/close/dup)

- System calls

## 2. Function Prototypes

```cpp

void print\_prompt();

vector<string> parse\_command(const string &input);

int execute\_command(vector<string> &args);

void handle\_cd(const vector<string> &args);

void handle\_help();

void handle\_exit();

bool handle\_redirection(vector<string> &args);

bool is\_builtin(const string &cmd);

int execute\_builtin(const vector<string> &args);

```

These declarations inform the compiler about functions that will be defined later, allowing them to be called before their actual implementation.

## 3. Main Function

```cpp

int main() {

string input;

vector<string> args;

int status = 0;

while (true) {

print\_prompt();

getline(cin, input);

if (input.empty()) continue;

args = parse\_command(input);

if (args.empty()) continue;

if (is\_builtin(args[0])) {

status = execute\_builtin(args);

if (status == -1) break;

continue;

}

if (handle\_redirection(args)) {

status = execute\_command(args);

} else {

cerr << "Redirection error" << endl;

}

}

return 0;}

\*\*Execution Flow:\*\*

1. Displays prompt

2. Reads user input

3. Skips empty input

4. Parses command into arguments

5. Handles built-in commands separately

6. Processes redirections if present

7. Executes external commands

8. Repeats until exit command

## 4. print\_prompt()

```cpp

void print\_prompt() {

// ANSI color codes

const string COLOR\_RESET = "\033[0m";

const string COLOR\_RED = "\033[31m";

const string COLOR\_GREEN = "\033[32m";

const string COLOR\_YELLOW = "\033[33m";

const string COLOR\_BLUE = "\033[34m";

const string COLOR\_MAGENTA = "\033[35m";

const string COLOR\_CYAN = "\033[36m";

const string COLOR\_WHITE = "\033[37m";

const string COLOR\_BOLD = "\033[1m";

char cwd[1024];

if (getcwd(cwd, sizeof(cwd)) != nullptr) {

// Colored prompt format: [shellname:path] $

cout << COLOR\_BOLD << COLOR\_GREEN << "myshell"

<< COLOR\_RESET << ":"

<< COLOR\_BLUE << cwd

<< COLOR\_RESET << " "

<< COLOR\_RED << "$ "

<< COLOR\_RESET;

} else {

// Fallback prompt if path unavailable

cout << COLOR\_BOLD << COLOR\_GREEN << "myshell"

<< COLOR\_RESET << " "

<< COLOR\_RED << "$ "

<< COLOR\_RESET;

}

cout.flush();

}```

\*\*Functionality:\*\*

- Gets current working directory using `getcwd()`

- Displays shell prompt with path

- Falls back to simple prompt if path unavailable

- `flush()` ensures prompt appears immediately

Explanation of Changes:

1. **ANSI Color Codes**:
   * Added constants for different ANSI color escape sequences
   * \033[ starts the escape sequence
   * 0m resets all attributes
   * Other numbers set specific colors/styles (31=red, 32=green, etc.)
2. **Colored Prompt Components**:
   * Shell name (myshell) in bold green
   * Colon separator in default color
   * Current directory in blue
   * Prompt symbol ($) in red
   * Proper color reset after each component
3. **Fallback Prompt**:
   * Maintains colored elements even when path isn't available
   * Same color scheme as main prompt
4. **Color Reset**:
   * Always reset colors at end to avoid "bleeding" into command output

Available Color Options:

You can customize the prompt by changing these color constants:

| **Color Code** | **Color** |
| --- | --- |
| \033[30m | Black |
| \033[31m | Red |
| \033[32m | Green |
| \033[33m | Yellow |
| \033[34m | Blue |
| \033[35m | Magenta |
| \033[36m | Cyan |
| \033[37m | White |
| \033[1m | Bold |
| \033[4m | Underline |

## 5. parse\_command()

```cpp

vector<string> parse\_command(const string &input) {

vector<string> tokens;

string token;

bool in\_quote = false;

for (char c : input) {

if (c == '"') {

in\_quote = !in\_quote;

} else if (isspace(c) && !in\_quote) {

if (!token.empty()) {

tokens.push\_back(token);

token.clear();

}

} else {

token += c;

}

}

if (!token.empty()) {

tokens.push\_back(token);

}

return tokens;

}

```

\*\*Parsing Logic:\*\*

1. Initializes empty token list and current token

2. Tracks quote state for handling spaces within quotes

3. Processes each character:

- Toggles quote state on `"`

- Splits on whitespace outside quotes

- Otherwise accumulates characters

4. Adds final token if non-empty

5. Returns token vector

## 6. handle\_redirection()

```cpp

bool handle\_redirection(vector<string> &args) {

auto it = args.begin();

while (it != args.end()) {

if (\*it == ">" || \*it == ">>" || \*it == "<") {

string op = \*it;

it = args.erase(it); // Remove operator

if (it == args.end()) return false;

string filename = \*it;

it = args.erase(it); // Remove filename

int fd;

if (op == ">") {

fd = open(filename.c\_str(), O\_WRONLY|O\_CREAT|O\_TRUNC, 0644);

} else if (op == ">>") {

fd = open(filename.c\_str(), O\_WRONLY|O\_CREAT|O\_APPEND, 0644);

} else if (op == "<") {

fd = open(filename.c\_str(), O\_RDONLY);

}

if (fd == -1) {

perror("open");

return false;

}

if (op == ">" || op == ">>") {

dup2(fd, STDOUT\_FILENO);

} else {

dup2(fd, STDIN\_FILENO);

}

close(fd);

} else {

++it;

}

}

return true;

}

```

\*\*Redirection Handling:\*\*

1. Iterates through command arguments

2. When finding `>`, `>>`, or `<`:

- Removes the operator and filename from args

- Opens file with appropriate flags:

- `>`: Create/truncate (O\_TRUNC)

- `>>`: Create/append (O\_APPEND)

- `<`: Read-only (O\_RDONLY)

3. Uses `dup2()` to redirect:

- STDOUT for output redirection

- STDIN for input redirection

4. Closes original file descriptor

5. Returns false on errors

## 7. execute\_command()

```cpp

int execute\_command(vector<string> &args) {

pid\_t pid = fork();

if (pid == 0) { // Child process

vector<char\*> argv;

for (const auto &arg : args) {

argv.push\_back(const\_cast<char\*>(arg.c\_str()));

}

argv.push\_back(nullptr);

execvp(argv[0], argv.data());

perror("execvp");

exit(EXIT\_FAILURE);

} else if (pid < 0) {

perror("fork");

return -1;

} else {

int status;

waitpid(pid, &status, 0);

// Reset standard I/O

dup2(STDIN\_FILENO, STDIN\_FILENO);

dup2(STDOUT\_FILENO, STDOUT\_FILENO);

dup2(STDERR\_FILENO, STDERR\_FILENO);

return status;

}

}

```

\*\*Process Execution:\*\*

1. Forks to create child process

2. In child:

- Converts args to C-style array

- Executes command with `execvp()`

- Exits on error

3. In parent:

- Waits for child completion

- Resets standard I/O descriptors

- Returns exit status

## 8. Built-in Command Handlers

### is\_builtin()

```cpp

bool is\_builtin(const string &cmd) {

static const vector<string> builtins = {"cd", "help", "exit"};

return find(builtins.begin(), builtins.end(), cmd) != builtins.end();

}

```

Checks if command is in builtins list.

### execute\_builtin()

```cpp

int execute\_builtin(const vector<string> &args) {

if (args[0] == "cd") {

handle\_cd(args);

} else if (args[0] == "help") {

handle\_help();

} else if (args[0] == "exit") {

handle\_exit();

return -1; // Exit signal

}

return 0;

}

```

Routes to appropriate built-in handler.

### handle\_cd()

```cpp

void handle\_cd(const vector<string> &args) {

if (args.size() == 1) {

const char \*home = getenv("HOME");

if (home && chdir(home) != 0) {

perror("cd");

}

} else if (args.size() == 2) {

if (chdir(args[1].c\_str()) != 0) {

perror("cd");

}

} else {

cerr << "cd: too many arguments" << endl;

}

}

```

Changes directory with error checking.

### handle\_help()

```cpp

void handle\_help() {

cout << "Simple C++ Shell\nBuilt-in commands:\n"

<< " cd [dir] - Change directory\n"

<< " help - Show help\n"

<< " exit - Exit shell\n"

<< "Other commands execute external programs\n";

}

```

Displays help message.

### handle\_exit()

```cpp

void handle\_exit() {

cout << "Goodbye!\n";

}

```

Prints exit message.

This implementation provides a solid foundation for a custom shell with basic functionality and I/O redirection capabilities.