**1. Introduction**

The Minimalist Linux Shell (Minishell) is a lightweight command-line interpreter developed in C, inspired by the Unix philosophy of simplicity and modularity. It serves as an interface between the user and the operating system, interpreting commands and translating them into system calls. Minishell supports essential features such as built-in commands (cd, pwd, exit), execution of external commands via execvp, input/output redirection (<, >, >>), pipelines using the | operator, and process management through fork, wait, and Unix file descriptors. The primary goal is to provide a minimal yet functional shell that adheres to POSIX standards, ensuring clarity, correctness, and robustness.

**2. Objectives of the Project**

The project aims to:

1. **Understand Core Shell Concepts**: Develop a practical understanding of how Unix-like shells interpret commands, create processes, handle input/output redirection, and implement pipelines using fundamental system calls such as fork, execvp, wait, pipe, and dup2.
2. **Implement a Minimalist Shell**: Build a lightweight command-line interpreter in C that supports built-in commands (cd, pwd, exit), execution of external programs, input/output redirection (<, >, >>), and command pipelines (|).
3. **Apply Modular Software Design**: Utilize a modular structure and clearly defined interfaces (module1\_parser, module2\_builtin, module3\_execute, module4\_pipeline, and main) aimed at maintainability, extensibility, and separation of concerns.
4. **Develop Robust Error Handling**: Ensure graceful detection and reporting of syntax and runtime errors—including invalid commands, redirection syntax errors, file access issues, or pipe failures—using structured error messages and safe recovery strategies.
5. **Create and Execute Comprehensive Tests**: Design and conduct functional and integration-level test suites that validate built-in behavior, external command execution, file-based input/output redirection, pipelines of varying length and complexity, and edge and error conditions.
6. **Enhance Understanding of OS Interfaces**: Deepen comprehension of operating system abstractions through practical use of file descriptors, environment variables (getenv, getcwd, chdir), inter-process communication, and process lifecycle management.

**3. System Architecture & Methodology**

**3.1 High‑Level Workflow**

A streamlined view of how commands move through your shell:

**CLI Parsing**

1. Display prompt and read user input via fgets().

2. Tokenize input into tokens (normal words, <, >, >>, |).

3. Build one or more command segments, filling argv[], in\_file/out\_file, and flags.

4. Decide execution path:

* If ncmd == 1 and first argv matches a built-in → call handle\_builtin()
* Else if single external → call execute\_command()
* Else (ncmd > 1) → call execute\_pipeline()
* execute\_command(cmd)
  1. fork() a child process
  2. In child: apply I/O redirection (open, dup2) as needed
  3. execvp() the external command
  4. In parent: waitpid() for the child's completion
* execute\_pipeline(commands[], ncmd)
  1. Create (ncmd‑1) pipes
  2. Loop for each command:
     + fork() child
     + In child: connect stdin/stdout to pipe ends via dup2(), set up redirection, then execvp()
  3. Close all pipe FDs in parent
  4. wait() for all children

This flow ensures modular handling of built-ins, single commands, and pipelines, maintaining separation between parsing, execution logic, and error handling.

**3.2 Data Structures**

Key C structs that drive command representation and parsing:

// token\_t: Represents each word or operator in the input

typedef struct {

token\_type\_t type; // TOK\_NORMAL, TOK\_INPUT, TOK\_OUTPUT, TOK\_OUTPUT\_APPEND, TOK\_PIPE

char \*text; // actual token data

} token\_t;

typedef struct {

int num\_tokens;

token\_t tokens[MAX\_TOKENS];

} token\_list\_t;

// command\_t: A structured command for exec with redirection information

typedef struct {

char \*argv[MAX\_TOKENS];

char \*in\_file; // filename after '<'

char \*out\_file; // filename after '>' or '>>'

int append; // boolean for append mode

} command\_t;

* token\_list\_t serves as an intermediate between raw input and executable commands.
* command\_t[] arrays represent each stage in a pipeline, encapsulating all argv and I/O redirection details.

**3.3 Tools & Libraries**

Your shell relies only on POSIX standard libraries and UNIX system calls—keeping it compact and portable:

| **Tool/Library** | **Purpose** |
| --- | --- |
| stdio.h | Input/output functions: fgets, printf, perror |
| stdlib.h | Memory allocation: malloc, strdup, and termination with exit |
| string.h | String operations: strncpy, strcmp, strdup |
| unistd.h | Core system calls: fork, execvp, pipe, dup2, chdir, getcwd |
| fcntl.h | File control: open file descriptors with proper flags |
| sys/wait.h | Process synchronization: wait, waitpid |
| errno.h | Error reporting via errno for system call failures |
| sys/stat.h | File permission constants for redirection (0644) |

No external or third-party dependencies are required—just standard libraries and direct system calls, making your shell lightweight and easy to build or port across UNIX-like platforms.

**4. Implementation Details**

**4.1 Source Files**

* **module1\_parser.c/.h** – Tokenizes input and constructs command\_t structures.
* **module2\_builtin.c/.h** – Implements built-in commands: cd, pwd, exit.
* **module3\_execute.c/.h** – Executes a single command with I/O redirection using fork, execvp, and wait.
* **module4\_pipeline.c/.h** – Chains multiple command\_t instances into pipelines using pipe, fork, and dup2.
* **main.c** – The shell driver: reads user input, invokes parser, built-ins or pipeline exec, loops until exit.

**4.2 Key Functions**

|  |  |
| --- | --- |
| **Function** | **Description** |
| int parse\_input(const char\*, token\_list\_t\*) | Splits the input line into tokens of type normal, <, >, >>, or ` |
| int build\_command\_segments(token\_list\_t\*, command\_t\*) | Organizes tokens into command segments, populates argv, in\_file, out\_file, and append flags. |
| int handle\_builtin(char\* argv[]) | Detects and executes built-in commands (cd, pwd, exit), returns flag to skip fork/exec. |
| int execute\_command(command\_t\*) | Forks a child, sets up I/O redirection, and uses execvp to run a single (non-piped) command. |
| void execute\_pipeline(command\_t commands[], int ncmd) | Sets up all pipes, forks child processes for each command in the pipeline, applies redirection, and waits for them. |
| main() – loop | Reads user input via fgets, tokenizes, builds commands, dispatches built-ins or pipelines, frees tokens. |

**4.3 Error Handling**

* **System call failures** (fork, pipe, open, dup2, execvp) trigger perror (or fprintf) in the child or parent, followed by an exit with non-zero status.
* **Missing filenames** after redirection operators result in skipped assignment and possible null in\_file/out\_file.
* **Built-in misuse** (e.g., cd to invalid directory, incorrect exit argument) prints a clear error using perror("cd") or exit() with default behavior.
* **Pipeline errors**: failure during pipe creation leads to immediate exit; failures in any child are printed to stderr, but parent waits for all children before proceeding.
* **Input line too long**: longer than MAX\_CMD\_LENGTH leads to truncation; shell continues afterward without crash.

**4.4 System Calls Used**

* fork() – Create child processes
* execvp() – Replace child process image with external command
* waitpid() / wait() – Parent waits for child termination
* pipe() – Establish inter-process communication via a pipeline
* dup2() – Redirect file descriptors for I/O or pipe handling
* open() – Open files for redirection
* close() – Close unused file descriptors
* perror() / fprintf(stderr,…) – Report system call failures with error info
* getcwd(), chdir() – Handle pwd and cd built-in commands
* exit() – Terminate shell via built-in exit

These calls form the foundation of your shell’s core logic – parsing, redirection, pipeline creation, command execution, and error management.

**5. Testing and Verification**

**5.1 Test Environment**

* **Platform**: Ubuntu 24.04 LTS running on a virtual machine.
* **Compiler**: GCC 12.2 with -Wall -Werror.
* **Shell**: /bin/bash for driving tests in both interactive and batch modes.
* **Tools & Frameworks**: Custom bash scripts for automated functional and edge-case testing.
* **Sandbox**: A temporary directory where all input/output redirections occur, ensuring no interference with host filesystem.

**5.2 Test Cases**

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Description** | **Expected Outcome** |
| Built-in:  cd <existing\_dir> | Change to a directory that exists | Shell’s current directory becomes <existing\_dir>; pwd reflects change. |
| Built-in:  cd (no args) | Invoke cd without arguments, using $HOME | Shell changes to HOME directory; pwd outputs $HOME. |
| Built-in:  pwd | Print current working directory | Displays the current directory path. |
| Built-in:  exit <n> | Exit shell with specified status code | Shell exits with status n. |
| External:  ls -la | List directory contents with options | Displays full listing; exit status 0. |
| External:  echo hello world | Print a simple string | Outputs hello world; exit status 0. |
| Input redirection:  wc < file.txt | wc reads from a file via < | Shows counts for file.txt; exit status 0. |
| Output redirection:  echo x > out | echo writes to file via > | out contains x\n; existing content is overwritten; exit status 0. |
| Output append:  echo y >> out | Append to out | out now ends with appended y\n; exit status 0. |
| Combined redirection: grep a < in > out | grep reads in, writes matches to out | out contains matches; exit status 0 if matches found, else non-zero. |
| Pipeline:  `ls | wc -l` | Pass output of ls to wc -l |
| Multi-pipe:  `cat a | grep x | sort` |
| Nonexistent cmd:  foo | External command not found | Error message to stderr; exit status 127. (“command not found”) |
| Bad redirection:  ls > | Redirect without specifying file name | Prints syntax error or message to stderr; exit status non-zero. |
| Redirection error:  < nofile | Input redirect to non-existent file | Error to stderr; exit status non-zero. |
| Empty input | Press Enter or submit blank line | Shell ignores and prompts again; no crash, exit status remains unchanged. |
| Consecutive pipes:  `ls |  | wc` |
| Redirection on built-in:  cd > f | Attempt redirect on built-in | Generates controlled error; shell remains active; no file overwritten. |
| Over-length input (>1024 bytes) | Input exceeding MAX\_CMD\_LENGTH | Shell truncates or prints error; must not crash or hang. |

**5.3 Test Procedure**

**1. Compile the Shell Program**

* **Action**: Compile all source files of the shell program.
* **Command**:

gcc -o minishell module1\_parser.c module2\_builtin.c module3\_execute.c module4\_pipeline.c main.c

* **Expected Outcome**: The minishell executable is created without errors.

**2. Run the Shell Program**

* **Action**: Launch the shell program.
* **Command**:

./minishell

* **Expected Outcome**: The shell prompt minishell> appears, indicating the shell is ready to accept commands.

**3. Test Built-in Commands**

* **Action**: Execute built-in commands like pwd, cd, and exit.
* **Commands**:

pwd

cd /home/user

exit

* **Expected Outcome**:
  + pwd: Displays the current working directory.
  + cd /home/user: Changes the directory to /home/user.
  + exit: Exits the shell program.

**4. Test External Commands**

* **Action**: Execute external commands such as ls and gcc.
* **Commands**:

ls

gcc --version

* **Expected Outcome**:
  + ls: Lists the contents of the current directory.
  + gcc --version: Displays the version of the GCC compiler.

**5. Test Input Redirection**

* **Action**: Test input redirection using <.
* **Commands**:

./minishell < input.txt

* **Expected Outcome**: The shell reads commands from input.txt and executes them.

**6. Test Output Redirection**

* **Action**: Test output redirection using >.
* **Commands**:

echo "Hello, World!" > output.txt

cat output.txt

* **Expected Outcome**:
  + echo "Hello, World!" > output.txt: Writes "Hello, World!" to output.txt.
  + cat output.txt: Displays "Hello, World!" from output.txt.

**7. Test Append Output Redirection**

* **Action**: Test append output redirection using >>.
* **Commands**:

echo "First Line" > output.txt

echo "Second Line" >> output.txt

cat output.txt

* **Expected Outcome**: output.txt contains:

First Line

Second Line

**8. Test Command Pipeline**

* **Action**: Test command pipelines using |.
* **Commands**:

echo "Hello" | awk '{print $1}'

* **Expected Outcome**: Displays "Hello".

**9. Test Background Execution**

* **Action**: Test background execution using &.
* **Commands**:

sleep 5 &

* **Expected Outcome**: The command runs in the background, and the shell prompt returns immediately.

**10. Test Exit Status**

* **Action**: Test the exit status of commands.
* **Commands**:

ls /nonexistent

echo $?

* **Expected Outcome**: ls returns a non-zero exit status, and echo $? displays the exit status.

**6.Results**

* **Built‑in commands** (cd, pwd, exit): Operate correctly including default cd to HOME and exit codes.
* **External commands**: Executed as expected via execvp; output and exit statuses are accurate.
* **Redirections**:
  + **Input (<)**: Works reliably; missing files produce proper errors.
  + **Output overwrite (>)**: Correctly truncates or creates files.
  + **Output append (>>)**: Appends without overwriting.
* **Pipelines (|)**: Single and multi-stage pipelines (e.g. ls | wc‑l, cat | grep | sort) produce correct output.
* **Error handling**: Invalid commands or bad syntax produce clear error messages without crashing. Redirection misuse on built-ins is safely handled.
* **Robustness**: Blank or overly long inputs are managed gracefully; no crashes or undefined behavior observed.

**7. Conclusion**

Minishell fulfills its primary aims with a clean, minimal implementation that remains reliable and modular. It correctly supports built-ins (cd, pwd, exit), external command execution via execvp, and seamless input/output redirection (<, >, >>), as well as single and multi-stage pipelines using fork, pipe, and dup2.

Errors—whether from invalid commands, misused redirection, or missing files—are handled gracefully without crashes. The modular separation (parsing, built-in handling, execution, pipeline) ensures clarity and extensibility.

Through this project you've gained practical experience with POSIX process and I/O management, shell behavior, and inter-process communication. Minishell demonstrates how essential shell capabilities can emerge from straightforward, well-structured code.

**8. Learning Outcomes**

The project facilitated hands-on experience with:

* **Unix Shell Fundamentals**: Implemented built-in commands (cd, pwd, exit), external command execution, and command pipelines.
* **System Calls and Process Management**: Utilized fork(), execvp(), pipe(), dup2(), and waitpid() for process creation, execution, and synchronization.
* **I/O Redirection**: Implemented input/output redirection using open() and dup2().
* **Modular Software Design**: Structured code into distinct modules for parsing, built-ins, execution, and pipelines, enhancing maintainability.
* **Error Handling**: Developed robust error reporting mechanisms using perror() and structured error handling strategies.
* **Testing and Validation**: Created comprehensive test suites to validate built-in behavior, external command execution, redirection, pipelines, and error conditions.

**9. References**

1. Kerrisk, M. (2010). *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press. ISBN: 978-1593272203.
2. Weiss, S. N. (2025). *System Programming in Linux: A Hands-On Introduction*. No Starch Press. ISBN: 978-1718503564.
3. "Making your own Linux Shell in C." GeeksforGeeks. Retrieved from <https://www.geeksforgeeks.org/making-linux-shell-c/>.