

# Minishell (pre)documentation

A pathway to shell glory, from zero to hero

The aim of this (pre)doc is to help approach the Minishell project at 42 School and make it feel like a less daunting project. It will go through an overview of the project, list the required knowledge and concepts needed before starting to write the code, make an inventory of the core implementations asked by the subject, and specify the main challenges that will arise during the whole process. It will also be accompanied by a checklist and workflow organization to help students divide and conquer Minishell.

The workflow and objective division is done following what, to my understanding, is the usual approach: **parsing-tokenizing on one side, executing on the other**. This is by no means a requirement, but it is my personal recommendation that, whenever possible, students build their teams this way so they can work and develop with enough independence so that they don't interfere in each other's flow, and when the time comes, work together in the contact point of this division (basically, the point in which the parser sends its tokens to the executor). Whatever the case may be, the point of this project (or at least one of them) is for both members of the group to hone up their skills and consolidate their knowledge around all minishell topics, so all of this has more to do with the practical approach to the project (i.e., how to tackle day 1 and confront the dreaded question, "how are we supposed to start building this thing"?). In this last regard, this document also contains a chapter dedicated to help navigate that initial contact with the building process. My only hope is that if this doc ends up in your hands or screens, it succeeds at making Minishell feel more manageable, approachable and even enjoyable.

This doc is the result of a personal research of the Minishell project, the information included in the subject and evaluation sheets and the consultation of direct sources with similar or complementary aims, like the project entry in the [Gitbook](#) written by Laura Fabbiano and Simon Aeby, or the [Ultimate Github Collaboration Guide](#) by Johnathan Mines. I also used chatGPT to expand on some areas and to help me build the checklists, examples and diagrams that are included in this file, and since credit must be given where credit is due, here is to the funny little demon that lives inside the vast, entangled mesh of prompts that has taken over the world with incredible force and immovable presence.

Good luck with the project.

This, too, shall pass.

PD: I strongly recommend that every student team makes their own pre-documentation instead of using someone else's, as the process of writing it is in itself a skill that they should practice. Use this doc however you like, as a template for your own or as a ready-to-use guide, just have that advice in mind ;D

# Contents

<b>Contents.....</b>	<b>2</b>
<b>1. Overview.....</b>	<b>4</b>
What is Minishell?.....	4
<b>2. Required Knowledge.....</b>	<b>6</b>
2. 1. System Calls.....	6
2. 2. Parsing and Tokenization.....	6
2. 3. Environment variables.....	7
2. 4. Signals.....	7
2. 5. Built-Ins.....	7
2. 6. File Descriptors and Redirections.....	8
2. 7. Debugging and Error Handling.....	8
2. 8. Github collaboration.....	8
<b>3. Deliverables.....</b>	<b>9</b>
3. 1. Core deliverables.....	9
3. 2. Mandatory Functional Requirements.....	10
3. 3. Bonus Functional Requirements.....	15
3. 4. Non-Functional Requirements.....	20
3. 5. Basic Checklist for Success.....	21
<b>4. Proposed Work Organization.....</b>	<b>22</b>
4. 1. Expanded Workflow Organization.....	23
4. 2. Day One Approach.....	24
4. 3. Work Breakdown Table (with phases).....	26
4. 4. Proposed Directory Structure.....	27
4. 5. Proposed Git Branch Management.....	29
<b>5. Mandatory In-Depth Checklist.....</b>	<b>31</b>
5.1 Person 1 (Minitalker) Responsibilities:.....	31
5.1.1 Initial Setup and Planning:.....	31
5.1.2 Shell Loop and Signal Handling:.....	31
5.1.3 Process Creation and Management:.....	32
5.1.4 Built-in Commands:.....	33
5.1.5 Garbage collector.....	34
5.2 Person 2 (Pipexer) Responsibilities:.....	34
5.2.1 Parsing and Tokenizing Input:.....	34
5.2.2 Redirection and Pipes:.....	35
<b>6. Bonus In-Depth Checklist.....</b>	<b>36</b>
6.1.1 Advanced Signal Handling:.....	37
6.1.2 Job Control:.....	37
6.2 Person 2 (Pipexer) Responsibilities:.....	39
6.2.1 Advanced Redirection and Piping:.....	39

6.2.2 Complex Command Structures:.....	40
<b>7. Workflow Diagrams.....</b>	<b>42</b>
<b>8. Documentation to-do list.....</b>	<b>43</b>

# 1. Overview

Minishell can feel like a gigantic, mind-shattering project the first time you even dare to think about it, so before we go through the basics of the subject, let's start by processing what it's asking from us, what are the main objectives and what the key concepts are inscribed in the process (i.e., what does one learn while doing Minishell).

## What is Minishell?

Minishell challenges the student to create **a small-scale, functional Unix shell** from scratch. **A shell serves as an interface between the user and the operating system**, allowing users to execute commands, manage files, and interact with system processes.

In Minishell, the student will develop **a simplified version of popular shells like [bash](#) or [zsh](#)**. The task is to replicate fundamental shell behaviors while gaining a deeper understanding of system-level programming in Unix.

The primary goal of Minishell is to replicate the behavior of a basic Unix shell while adhering to specific requirements and restrictions. This involves **creating a small, lightweight shell program that reads user input, interprets commands, and executes them, either as built-in functions or via external programs**.

In this project, you will:

- Implement a **read-execute loop**: This is the foundation of a shell, repeatedly prompting the user for input, processing it, and executing commands until the user chooses to exit.
- Handle **command parsing**: Break down user input into meaningful components ([tokens](#)) and resolve complexities like **quotes**, **pipes**, and **redirections**.
- Manage **processes**: Use system calls like **fork** and **execve** to execute commands in child processes while maintaining control in the parent shell.
- Implement **built-in commands**: Certain commands, such as **echo**, **cd**, and **env**, will be handled directly by Minishell without relying on external binaries.
- Support **redirection and piping**: Allow users to redirect input and output using symbols like **<**, **>**, and **|**.

Unlike full-fledged shells, Minishell focuses only on core features, allowing the student to master the essentials in a complexity-controlled environment. The student will work with strict guidelines, such as **avoiding external libraries**, which forces them to rely on system calls and deepens the understanding of low-level programming. All of this while implementing a **robust error handling** to ensure that Minishell behaves predictably even in edge cases or invalid input scenarios.

All together, this is a project about gaining hands-on experience with **fundamental Unix concepts**:

- **Processes**: Understand how processes are created, managed, and terminated.
- **File Descriptors**: Learn how input/output redirection is implemented at a system level (**Pipex** is a very useful project to learn the basics of i/o redirection).
- **Signals**: Handle real-time interruptions such as **CTRL-C** and **CTRL-D**. (**Minitalk** is a very useful project to learn the basics of signal handling).
- **Memory Management**: Ensure proper resource allocation and avoid leaks in a long-running program.

By completing this project, the student will acquire skills that form the backbone of many advanced programming tasks, including creating server applications, debugging complex systems, and writing efficient code for resource-constrained environments.

## 2. Required Knowledge

To successfully build Minishell, there are several core Unix and C programming concepts that the student needs to master. This section outlines the foundational topics and provides brief explanations to help get started.

### 2. 1. System Calls

Minishell heavily relies on Unix system calls, which are **low-level functions that allow programs to interact with the operating system**. Key system calls to master include:

- **fork**: Creates a new process (child) that is a copy of the calling process (parent). Both processes continue executing from the same point, but with different process IDs (**pid**). Forking is essential for running commands in child processes. The parent shell forks a child to execute external programs, leaving the shell itself free to continue running.
- **execve**: Replaces the current process image with a new program. After forking, the child process uses **execve** to run the command. Unlike **fork**, it does not create a new process.
- **wait / waitpid**: Makes the parent process wait for child processes to finish. Ensures proper synchronization and avoids zombie processes. Use of **waitpid** gives more control.
- **pipe**: Creates a pair of connected file descriptors for inter-process communication. Data written to one end of the pipe is read from the other.

### 2. 2. Parsing and Tokenization

Handling user input is one of the most complex aspects of Minishell. The student will need to **break input strings into meaningful components**, or **tokens**, while respecting shell syntax.

- **Tokenization**: Split input into commands, arguments, and operator using delimiters like spaces, pipes (**|**), and redirection symbols (**<**, **>**).
  - Example: **echo "Hello, world!" | grep Hello** becomes:
    - Command: **echo**
    - Argument: **"Hello, world!"**
    - Pipe: **|**
    - Command: **grep**
    - Argument: **Hello**
- **Quote Handling**: Manage single ( **'** ) and double ( **"** ) quotes to ensure arguments with spaces or special characters are treated as a single unit.

- Example: `echo "Hello, world!"` treats `"Hello, world!"` as one argument.
- **Escape Characters**: Handle `\` for escaping special characters.
  - **This is not asked in the subject**, so the choice to manage the escape character is up to the student. If the choice is not to accept the character, though, `minishell` needs to know how to act if it is encountered (e.g.: rejecting the input).
- **Syntax Validation**: Check for common errors, such as unclosed quotes or invalid command placement.

## 2. 3. Environment variables

Shells use environment variables to **store configuration data and share information across processes**.

- **Definition**: Key-value pairs (e.g., `PATH=/usr/bin:/bin`) that influence program behavior.
- **Commands to Handle**:
  - `env`: Print all environment variables.
  - `export`: Add or modify an environment variable.
  - `unset`: Remove an environment variable.
- **Usage**:
  - Access environment variables using the `environ` array or `getenv` function.

## 2. 4. Signals

Signals allow processes to **communicate asynchronously**, typically to interrupt or terminate them.

- **Common Signals**:
  - **SIGINT (CTRL-C)**: Interrupt the current process.
  - **SIGQUIT (CTRL-\)**: Quit the current process and dump core.
  - **EOF (CTRL-D)**: Signals the end of input.
- **Signal Handling**:
  - Use `signal` or `sigaction` to customize how the shell responds to these signals.

## 2. 5. Built-Ins

Minishell requires the student to implement several built-in commands. These are **commands executed directly by the shell without creating a new process**. Some of the required built-ins are:

- **echo**: Print text to the terminal.
- **cd**: Change the current working directory.
- **pwd**: Print the current working directory.
- **export** / **unset**: Manage environment variables.
- **env**: Display all environment variables.
- **exit**: Terminate the shell.

## 2. 6. File Descriptors and Redirections

Shells rely on file descriptors (FDs) to **manage input and output streams**.

- **Standard FDs:**
  - **STDIN (0)**: Standard input.
  - **STDOUT (1)**: Standard output.
  - **STDERR (2)**: Standard error.
- **Redirections:**
  - **<**: Redirect input from a file.
  - **>**: Redirect output to a file (overwrite).
  - **>>**: Append output to a file.
  - **2>**: Redirect error output to a file (**not in subject**)
  - **2>>**: Append error output to a file (**not in subject**)
- **Implementation**: Use **dup** and **dup2** to duplicate file descriptors for redirection.

## 2. 7. Debugging and Error Handling

- Ensure meaningful error messages for invalid input (e.g., “**command not found**”).
- Prevent segmentation faults by validating input and handling edge cases.
- Use tools like **valgrind**, **fsanitize** or **gdb** to debug memory issues.



## 2. 8. Github collaboration

While not an explicit necessary piece of knowledge, Minishell is a great opportunity to learn how to set up collaborating repositories with github and how to handle different working branches so that successive merges are safe, trackable and correctly inserted in the main project tree. A good resource to tackle this possible issue is the guide made by Johnathan Mines in their Medium page: [The Ultimate Github Collaboration Guide](#).

## 3. Deliverables

For this section, we'll outline the **expected outputs and artifacts for the Minishell project**. A clear understanding of deliverables ensures the students can stay focused on what needs to be achieved.

### 3. 1. Core deliverables

These are the deliverables that will form the foundation of the Minishell project. Each component is essential to the final success of the project, and together they ensure that the project is functional, well-documented, and properly structured.

#### Executable Program:

- File Name: **minishell**
- Purpose: A functional shell program that replicates a subset of Bash functionality.
- Location: The executable should be created at the root of the project directory after running the Makefile.

#### Documentation:

- ReadMe file
  - Brief description of the project.
  - How to compile and run the shell.
  - List of features implemented.
  - Known limitations (optional but helpful).
- Inline Comments:
  - Your code should include comments explaining critical sections, especially in complex parsing or execution logic.

## Norminette Compliance:

- Basic checklist:
  - Functions  $\leq 25$  lines.
  - Proper indentation and spacing.
  - Clear, descriptive variable names.
  - Maximum of one global variable (exclusively for signal handling)

## Makefile:

- Supporting at least the following rules:
  - **all**: Compiles the program.
  - **clean**: Deletes object files.
  - **fclean**: Deletes object files and the executable.
  - **re**: Recompiles everything from scratch.
- Must handle dependencies efficiently to avoid unnecessary recompilation (it is a good opportunity to learn how to use .d files).

## 3. 2. Mandatory Functional Requirements

This list of functional requirements define the expected behavior and features that the Minishell must have. They cover a wide range of features, from basic built-in commands to complex functionalities like piping, redirection, and handling signals. Each requirement focuses on a specific aspect of the shell's functionality and is critical for ensuring that the shell operates correctly and meets the project objectives.

### 3.2.1 Shell Prompt

- **Objective:** The shell should **print a prompt to the user indicating that it is ready for input**. This prompt should be customizable or predefined.
- **Behavior:**
  - When the shell is running, it should print a prompt (typically **minishell\$** ) after completing a command or when the shell is idle.
  - The prompt should update or remain static depending on whether the user is in a child process or if any specific conditions are met (e.g., a change in the current working directory after a **cd** command, if a compound prompt is implemented).

```
shell Copiar código

minishell$ ls
file1.txt  file2.txt
minishell$ cd /path/to/dir
minishell$ pwd
/path/to/dir
```

### 3.2.2 Reading User input

- **Objective:** The shell must **be capable of reading commands from the user** in a loop and executing them.
- **Behavior:**
  - The shell should accept and read input from the user in the form of a command line. The user should be able to press Enter after typing a command, and the shell should process and execute the command.
  - If the user types an invalid command or syntax error, the shell should display an appropriate error message.
  - The input should be parsed and handled by the Minishell, ensuring correct handling of spaces, quotes, and special characters.

```
shell Copiar código

minishell$ echo "Hello, world!"
Hello, world!
minishell$ exit
```

### 3.2.3 Built-In Commands

- **Objective:** The shell must **support a set of basic built-in commands** for interacting with the operating system.
- **Commands to Implement:**
  - **echo:** Prints arguments to the console.
  - **cd:** Changes the current directory.

- **pwd**: Prints the current directory.
- **exit**: Exits the shell.
- **Behavior:**
  - Each built-in command should be implemented as a function that can be called from the main shell loop.
  - The **cd** command should correctly handle relative and absolute paths.
  - The **exit** command should allow the user to exit the shell, terminating the program gracefully, with a specified exit code (or 0 if no argument is specified).
  - Implement error handling for commands. For instance, if **cd** is given an invalid path, the shell should print an appropriate error message.

```
shell Copiar código

minishell$ cd /path/to/directory
minishell$ pwd
/path/to/directory
minishell$ exit
```

### 3.2.4 External commands

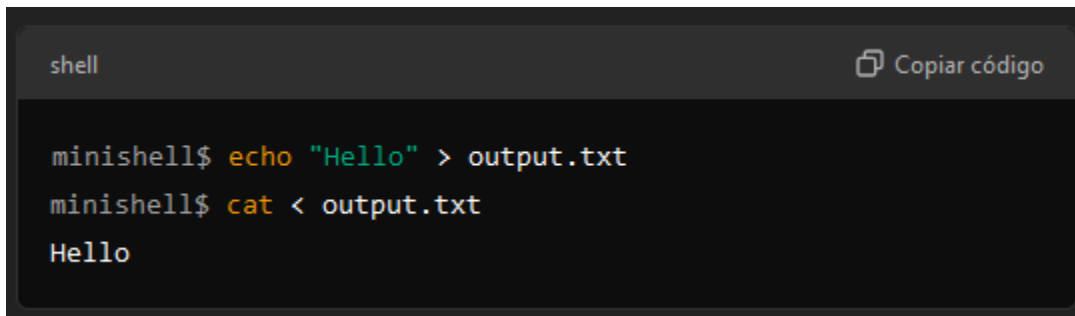
- **Objective:** The shell must be able to **execute external commands**, those that are not built-in.
- **Behavior:**
  - The shell should search for the executable commands in the directories listed in the **PATH** environment variable.
  - If the command exists, the shell should execute it in a separate process using **fork()** and **exec()** system calls.
  - If the command does not exist, the shell should print an error message and return to the prompt.

```
shell Copiar código

minishell$ ls
file1.txt  file2.txt
minishell$ /usr/bin/ls
file1.txt  file2.txt
```

### 3.2.5 Redirection

- **Objective:** The shell must support input/output redirection, allowing users to **direct the output of a command to a file or the input from a file.**
- **Types of Redirection:**
  - **>:** Redirects the output of a command to a file (overwrites/truncate the file).
  - **>>:** Appends the output of a command to a file.
  - **<:** Redirects input from a file.
  - **<<:** Redirects input from a here document (heredoc).
- **Behavior:**
  - The shell should parse the command to identify redirection operators.
  - For output redirection (**>**), the shell should open the specified file and write the command's output to it.
  - For input redirection (**<**), the shell should open the specified file and read from it as if it were typed by the user.
  - Implement error handling for cases where the file cannot be opened, accessed or created.

A terminal window titled 'shell' with a 'Copiar código' button in the top right. The terminal shows a sequence of commands: 'minishell\$ echo "Hello" > output.txt', 'minishell\$ cat < output.txt', and the output 'Hello'.

### 3.2.6 Here Documents

- **Objective:** The shell should support "here documents," **allowing the user to create multiline input** for a command directly within the shell.
- **Behavior:**
  - A here document is a special type of redirection that allows you to provide input to a command from within the shell itself, rather than from a file.
  - The user can specify a delimiter, and everything between the delimiter and the next occurrence of the same delimiter will be used as input for the command.
  - This feature is commonly used in commands that require multiline input (e.g., **cat**, **grep**, or **sed**).

```
arduino Copiar código

minishell$ cat << EOF
This is
a multi-line
text input.
EOF
This is
a multi-line
text input.
```

### 3.2.7 Piping

- **Objective:** The shell must support piping, **allowing the output of one command to be used as the input of another command.**
- **Behavior:**
  - The shell should parse the command to detect the pipe (|) operator.
  - The shell should create a pipe and fork processes to run the commands on each side of the pipe.
  - The output of the first command should be passed as input to the second command.
  - The shell should handle multiple pipes (e.g., **cmd1 | cmd2 | cmd3 | cmdn**).

```
shell Copiar código

minishell$ ls | grep "file"
file1.txt  file2.txt
```

### 3.2.8 Handling Environment Variables

- **Objective:** The shell must support environment variables, **allowing users to get, set, and manipulate environment variables.**
- **Behavior:**

- **env**: The shell should support the **env** command to display all environment variables.
- **setenv and unsetenv**: Implement functions to set and unset environment variables, respectively.
- **Variable Expansion**: The shell should expand environment variables in commands (e.g., **echo \$HOME** should print the value of the **HOME** environment variable).
- **Exporting Variables**: Allow environment variables to be passed to child processes via the **export** command.

```

shell Copiar código

minishell$ echo $HOME
/home/user
minishell$ export VAR=value
minishell$ echo $VAR
value

```

### 3.2.9 Signal Handling

- **Objective**: The shell should **handle signals appropriately**, particularly when a process is running in the foreground or background.
- **Behavior**:
  - **SIGINT (Ctrl + C)**: When the user presses Ctrl + C, the shell should stop the current running process and print a new prompt.
  - **SIGQUIT (Ctrl + \)**: Handle signal to terminate the current process.
    - The shell should not crash or freeze when these signals are received; instead, it should recover and present the prompt again.
  - **SIGTSTP (Ctrl + Z)**: should be ignored (doesn't mean not-implemented!)

### 3.2.10 Exit Status Codes

- **Objective**: The shell must **track and return exit statuses** for commands that were executed.
- **Behavior**:
  - Each command should return an exit status code (**0** for success, **non-zero for failure**).

- The shell should print the exit status of the last command when requested (e.g., by using `echo $?`).
- The `exit` command should allow the user to terminate the shell and return the exit status of the shell.

```
shell Copiar código  
  
minishell$ ls  
file1.txt file2.txt  
minishell$ echo $?  
0  
minishell$ nonexistent_command  
minishell$ echo $?  
127
```

### 3.2.11 Memory Management

- **Objective:** Proper **memory allocation and deallocation** should be handled throughout the shell's operations to avoid memory leaks.
- **Behavior:**
  - Ensure that memory is allocated for variables, strings, and data structures when needed and is freed when no longer necessary (e.g., when the program exits or after processing a command).
  - Handle errors gracefully by freeing any allocated memory before exiting due to a failure.

### 3.2.13 Autocompletion

- **Objective:** The shell should **offer basic autocompletion for commands, arguments, file names, and directories**.
- **Behavior:**
  - Implement command-line autocompletion by listening to keyboard events (typically using libraries like `readline`).
  - The shell should suggest possible completions for commands, file paths, and arguments based on what the user types.
  - For example, if the user types `cd /us` and presses Tab, the shell might suggest `/usr` or `/user`.



```
bash Copiar código

minishell$ echo "Hello"
minishell$ echo "World"
minishell$ history
1  echo "Hello"
2  echo "World"
minishell$ !1
echo "Hello"
```

### 3.3.14 Customizing the Prompt (NOT REQUIRED)

- **Objective:** Allow the user to **customize the prompt with dynamic information, such as the current directory, username, hostname, or even the exit status** of the last command.
- **Behavior:**
  - Allow the user to specify a custom prompt using environment variables or a configuration file.
  - The prompt could change dynamically based on the current state, for example:
    - Displaying the current working directory.
    - Showing the username or hostname.
    - Displaying the exit status of the last command.

```
ruby Copiar código

minishell$ export PS1="\u@\h:\w\$ "
user@hostname:/current/directory$
```

### 3.3.15 Quoting and Escaping

- **Objective:** Implement full support for single quotes ('), double quotes ("), and escape characters (e.g., \).
- **Behavior:**
  - Support for both single and double quotes:
    - Single quotes should preserve the literal value of characters inside the quotes (no variable expansion).

- Double quotes should allow variable expansion (e.g., `$HOME`).
- Handle escape characters to escape special characters like spaces, quotes, and backslashes.

```
bash Copiar código  
  
minishell$ echo "My home is $HOME"  
My home is /home/user  
minishell$ echo 'My home is $HOME'  
My home is $HOME
```

### 3. 3. Bonus Functional Requirements

The bonus requirements define additional features that enhance the Minishell's functionality. These features include logical operators, parentheses for prioritizing commands, and wildcard expansion. While not mandatory, implementing these requirements demonstrates advanced understanding and capabilities.

#### 3.3.1 Logical Operators: `&&` and `||`

- **Objective:** The shell must support logical operators `&&` (AND) and `||` (OR) for conditional execution of commands.
- **Behavior:**
  - `&&`: The second command executes only if the first command succeeds (i.e., returns an exit status of 0).
  - `||`: The second command executes only if the first command fails (i.e., returns a non-zero exit status).

#### 3.3.2 Parentheses for Prioritization

- **Objective:** The shell must support parentheses to group commands and define the execution priority.
- **Behavior:**
  - Commands enclosed in parentheses are treated as a single unit and executed with priority.

- Example: `(command1 && command2) || command3`. `command1 && command2` is evaluated first. If both succeed, `command3` is skipped. Otherwise, `command3` executes.

### 3.3.3 Parentheses for Prioritization

- **Objective:** The shell must support wildcard expansion using `*` to match files and directories in the current working directory.
- **Behavior:**
  - **Wildcard Matching:** The `*` wildcard matches zero or more characters in file and directory names.
  - **Integration with Commands:** Wildcards should work seamlessly with other commands (e.g.: `ls *.txt`)

## 4. Proposed Work Organization

This proposed work organization divides the Minishell project into two main areas, with tasks allocated to each team member based on the two main minishell branches: parsing and execution. This organization allows each team member to leverage their strengths and work efficiently on complementary components, while collaborating on shared tasks like error handling and debugging, while minimizing the points of contact and synchronization.

**Person 1 will focus on tasks primarily related to parsing. This includes:**

- Tokenizing user input into manageable components.
- Parsing commands, arguments, and special characters (e.g., `|`, `>`, `<`).
- Ensuring proper handling of quotes and escape sequences.
- Implementing the logic for command grouping and parentheses (bonus: `&&`, `||` support).

**Person 2 will handle tasks related to execution. This includes:**

- Managing process creation and execution for both built-in and external commands.
- Implementing redirection (e.g., `<`, `>`, `>>`) and piping functionality.
- Handling environment variables and their expansion.
- Ensuring proper execution order and state management.

Both partners will need to collaborate on:

- Error handling & debugging to ensure smooth integration between parsing and execution.
- Managing edge cases, such as invalid syntax or unsupported operations.

Keep in mind that the proposed workflow organization and the GitHub branch management don't need to follow the exact same task division. For example, the Parsing section can be divided into separate branches for tokenization, syntax validation, and command grouping. Similarly, Execution can be split into branches for process management, redirection, and environment variable handling. This flexibility allows the team to adapt the workflow to their preferences and ensure efficient development.

## 4. 3. Proposed Directory Structure

To streamline development and maintain clarity, the **src/** directory is divided into subdirectories based on the main areas of concern within the Minishell project. This organization reflects the core components of a functional shell, such as parsing, signal handling, execution, and environment management. By grouping related functionality together, the structure facilitates collaboration, simplifies debugging, and enhances maintainability as the project scales.

```

Minishell/
├─ includes/           # Header files
├─ libs/              # External libraries (if any)
├─ tests/             # Test files
├─ Makefile           # Makefile for building the project
├─ src/               # Source code
│   ├─ builtins/      # Built-in command implementations
│   │   ├─ cd.c
│   │   ├─ echo.c
│   │   ├─ env.c
│   │   └─ exit.c
│   ├─ executor/      # Command execution (piping, redirection, execve)
│   │   ├─ executor.c
│   │   ├─ piping.c
│   │   └─ redirection.c
│   ├─ parser/        # Input tokenization and syntax checking
│   │   ├─ parser.c
│   │   ├─ tokenizer.c
│   │   └─ syntax_checker.c
│   ├─ signals/       # Signal handling
│   │   └─ signals.c
│   ├─ env/           # Environment variable management
│   │   ├─ env.c
│   │   └─ env_utils.c
│   ├─ utils/         # Utility functions
│   │   ├─ string_utils.c
│   │   └─ error_handler.c
│   └─ main/          # Shell loop and main entry point
│       ├─ main.c
│       └─ shell_loop.c

```

### **builtins/:**

- Implementations for built-in commands.
- Example files: `cd.c`, `echo.c`, `env.c`, `exit.c`.

### **executor/:**

- Functions to execute commands.

- Handles piping (`|`), redirection (`<`, `>`), and external commands (`execve`).
- Example files: `executor.c`, `pipng.c`, `redirection.c`.

#### **parser/:**

- Handles input tokenization and syntax validation.
- Splits user input into commands, arguments, and operators.
- Example files: `parser.c`, `tokenizer.c`, `syntax_checker.c`.

#### **signals/:**

- Manages signal behavior for the shell.
- Handles signals like `SIGINT` and `SIGQUIT`.
- Example files: `signals.c`.

#### **env/:**

- Functions for managing environment variables (`getenv`, `setenv`, `unsetenv`).
- Example files: `env.c`, `env_utils.c`.

#### **utils/:**

- Common utility functions shared across modules.
- Example files: `string_utils.c`, `error_handler.c`.

#### **main/:**

- Core shell loop, prompt display, and project entry point.
- Example files: `main.c`, `shell_loop.c`.

## 4. 5. Proposed Git Branch Management

In order to maintain a clean and scalable working tree connecting local and github repositories, it is advised to organize progress branches in different types, regarding the consecutive phases of development. This, of course, means that **before starting to work on any feature, the student must check that they are in the correct corresponding branch, or create it if it doesn't yet exist.** With that out of the way, the following branch management proposal divides the workflow in **3 tiers of branches**, and closely aligns with Git branching best practices, particularly the **Git Flow model**.

This strategy is built with several aims in mind: it allows for **isolated workflows** for each new functionality, minimizing possible conflicts; it is **collaboration friendly**, both at the develop level (the stages through which the project advances) and at the feature level (each individual implementation around which the project is divided); it **keeps the main branch stable** and production-ready; it has **testing support** to open an independent channel for rigorous checking before integration; it is **structured around incremental progress**, mimicking... well, a tree.

### Feature Branches

- **Purpose:** To develop specific features or functionalities.
- **Naming Convention:** Use clear, descriptive names (e.g., **feature-shell-loop** or **feature-signal-handling**).
- **Workflow:**
  - Create a feature branch off the **develop** branch (or **main** if no **develop** exists yet).
  - Work on the feature, committing changes incrementally.
  - When the feature is complete and tested, merge it back into **develop** via a pull request.
- **Benefits:** Keeps the main branch clean and allows parallel development of multiple features.

### Develop Branch

- **Purpose:** A central branch for integrating all feature branches.
- **Role in Workflow:**
  - Feature branches are merged into **develop** as they are completed.
  - The **develop** branch reflects the latest state of the project with all ongoing features combined but not yet released.
  - Bug fixes or enhancements from **test** branches can also be merged into **develop**.
  - Once the work in **develop** is stable, it can be merged into **main** for release.

- **Benefits:** Serves as a staging area for ongoing development, keeping **main** stable.

## Test Branches

- **Purpose:** To experiment, debug, or test specific parts of the codebase without risking disruption to other branches.
- **Workflow:**
  - Create a test branch off **develop** or a feature branch (e.g., **test-shell-loop**).
  - Run tests, debug, or try experimental changes.
  - Decide whether to merge changes back to the parent branch or discard the test branch after testing.
- **Benefits:** Allows safe experimentation without affecting production or ongoing development.

## Suggested Workflow Summary:

1. **Start with a clean **main** branch.**
2. Create a **develop** branch from **main**.
3. Develop features on **feature-** branches, merging into **develop**.
4. Use **test-** branches as needed for debugging or experiments.
5. Regularly merge **develop** into **main** once everything is stable and ready for release.



# 5. Mandatory In-Depth Checklist

This checklist provides a step-by-step guide to building the Minishell project. It is important to tackle the core requirements first (such as process creation, built-in commands, and redirection) before moving on to more advanced features like job control and signal handling. Ensure that the core functionality is solid before moving on to the bonus requirements.

This checklist should serve as a useful organizational tool for teams to track progress throughout the project. It breaks down the main tasks into manageable steps, making the project less overwhelming and more structured.

## 5.1 Person 1 Responsibilities

### 5.1. Initial Setup and Planning

#### 5.1.1. Set up the project repository and .gitignore:

- Initialize a new Git repository.
- Create the `srcs/`, `includes/`, and `libs/` directories in the project.
- Add appropriate `README.md` and `LICENSE` files (if necessary).
- Set up `.gitignore` to ignore common build files (e.g., `*.o`, `*.d`, `*.a`, `*.out`).
- Exclude IDE-specific files (e.g., `.vscode`, `.idea`, etc.) and system-specific files (`Thumbs.db`, `.DS_Store`, etc.).

#### 5.1.2. Plan the directory structure:

- Organize the project with the following structure:
  - `srcs/`: For source code files.
  - `includes/`: For header files.
  - `libs/`: For any external libraries.
- Ensure that headers are placed in the `includes/` directory and source files are placed in `srcs/`.

### 5.2. Shell Loop and Signal Handling

#### 5.2.1. Infinite Shell Loop:

- **Prompt Display:**
  - Create a function to display the shell prompt (e.g., `$` or a custom prompt).
  - Display username and current working directory (CWD).
- **User Input Handling:**
  - Implement `readline()` or `get_next_line()` to capture user input.
  - Strip any leading or trailing whitespace.
- **Loop Logic:**

- Set up the loop to continuously read and process commands until exit (`exit` or `Ctrl+D`).
- Add logic for handling empty input (e.g., do nothing if the user presses Enter with no input).

### 5.2.2. Signal Handling:

- **SIGINT (Ctrl+C):**
  - Create a custom signal handler to intercept `SIGINT` and prevent the shell from terminating immediately.
  - Print a new prompt when `Ctrl+C` is pressed, without exiting the shell.
- **SIGTSTP (Ctrl+Z):**
  - Implement a custom handler for `SIGTSTP` to catch `Ctrl+Z` and suspend background processes, showing a message indicating suspension.
  - Ensure the shell remains responsive after `Ctrl+Z`.

### 5.2.3. Env Variable Copy:

- Before running the infinite loop, store a copy of the environment variables into a list or an array.
- If there is no environment, have a fallback username (e.g., `root@localhost`) taken from `/etc/passwd`.

### 5.2.4. Edge Execution Cases:

- Minishell should run from a non-existing directory.
  - With `env`, it should display the same CWD from where it was launched and be able to `cd` backwards.
  - With no `env`, it should not crash and have a fallback default route (`/root`).
  - Display a specific CWD when no `env` and non-existing directory.

## 5.3. Built-in Commands

### 5.3.1. `cd` (Change Directory):

- Implement the `cd` command to change the current working directory using `chdir()`.
- Handle cases for absolute paths, relative paths, `cd -`, `cd /`, and no argument (fallback to home directory).
- Normalize paths (e.g., handle `../`, `./`, and special characters like `~`).
- Ensure no crashes in unset `env` execution.

### 5.3.2. `echo` (Print to Standard Output):

- Parse arguments and print them to the standard output, separated by spaces.

- Handle special options, like `-n` to prevent a newline at the end.

#### 5.3.3. `exit` (Exit the Shell):

- Implement the `exit` built-in command to terminate the shell.
- Allow an optional exit status code.
- Ensure proper cleanup (close open file descriptors, release memory).

#### 5.3.4. `env` (Show Environment Variables):

- Implement the `env` command to print all environment variables in `KEY=VALUE` format.

#### 5.3.5. `export` (Set or Export Environment Variables):

- Implement `export` to create or modify environment variables.
- If no arguments are given, print the list of exported variables.
- Handle invalid cases gracefully.

#### 5.3.6. `unset` (Remove Environment Variables):

- Implement `unset` to remove environment variables.
- Ensure proper handling of edge cases, like attempting to unset non-existent variables.

### 5.4. Garbage Collector

#### 5.4.1. Create a garbage collector struct:

- Make it able to store both single and double pointer cases.

#### 5.4.2. Create a garbage collector builder:

- Enable it to build nodes that point to single and double pointer cases.

#### 5.4.3. Create a garbage collector cleanup protocol:

- Traverse the list and free single pointers and double pointers appropriately.

## 5.2 Person 2 Responsibilities

### 5.2.1 Parsing and Tokenizing Input

#### 5.2.1.1. Write a tokenizer function to split input by spaces and handle special characters (quotes, pipes, redirection):

- Implement a tokenizer that splits the user input string into tokens based on spaces and special characters.
- Handle quotes (single and double) as single tokens and escape sequences.
- Tokenize and separate out pipes (|) and redirection operators (<, >, >>, <<).
- Handle edge cases, like empty tokens or unclosed quotes, and report errors where applicable.

#### 5.2.1.2. Detect and handle background execution (&):

- Parse the input string to detect the & symbol at the end of a command.
- Set a flag for background process execution.

## 2. Redirection and Pipes

#### 5.2.2.1. Implement input redirection (<), output redirection (>, >>), and heredoc redirection (<<):

- Handle opening files for input/output using `open()` and redirecting streams using `dup2()`.
- Manage temporary files for heredoc input until the set delimiter is found.

#### 5.2.2.2. Test and handle combined redirections:

- Implement logic for combined redirections (e.g., `2>` for `STDERR`).
- Ensure correct handling of `STDOUT` and `STDERR` redirections.

#### 5.2.2.3. Design and test pipes (|) functionality:

- Detect the pipe (|) symbol in tokenized input and separate commands.
- Implement pipe communication using `pipe()`, `fork()`, `dup2()`, and `execvp()`.
- Handle multiple pipes and errors like empty commands.

#### 5.2.2.4. Test pipe and redirection integration:

- Test scenarios combining pipes and redirections (e.g., `cmd1 < file1 | cmd2 > file2`).
- Ensure seamless integration of piping and redirection.

## 6. Bonus In-Depth Checklist

### 6.1 Wildcard Expansion

#### 6.1.1 Implement wildcard parsing and matching:

- Develop logic to detect wildcard patterns (e.g., `*`, `?`, and `[ ]`) in user input.
- Ensure wildcard expansion works for directory listings by integrating with `opendir()` and `readdir()`.
- Handle edge cases such as invalid patterns or no matching files.

#### 6.1.2 Sort and format the results:

- Implement sorting logic to organize the matching files alphabetically (if required).
- Ensure proper formatting when integrating expanded wildcards into the command.

#### 6.1.3 Test wildcard expansion in different contexts:

- Test with commands such as `ls *.c`, `echo file[1-3]`, and `rm test*`.
- Ensure wildcard expansion does not modify the original arguments for cases where no match is found (e.g., `echo "no match *"`).

### 6.2 Logical Operators (&& and ||)

#### 6.2.1 Parse logical operators:

- Extend the tokenizer to detect `&&` and `||` as distinct tokens.
- Ensure proper tokenization when logical operators are surrounded by spaces (e.g., `command1 && command2`) or not (e.g., `command1&&command2`).

#### 6.2.2 Implement logical execution flow:

- Add logic to execute commands conditionally based on the return status of the previous command:
  - `&&`: Execute the next command only if the previous command succeeds (exit status 0).
  - `||`: Execute the next command only if the previous command fails (non-zero exit status).
- Manage the chaining of multiple logical operators in a single command line (e.g., `command1 && command2 || command3`).

#### 6.2.3 Handle parentheses for grouping:

- Parse and handle parentheses to group commands and define execution precedence (e.g., `(command1 && command2) || command3`).
- Implement a recursive or stack-based approach to evaluate nested groups.

#### 6.2.4 Test various combinations of logical operators:

- Test simple cases: `command1 && command2`, `command1 || command2`.
- Test mixed cases: `command1 && command2 || command3`.

- Test grouped cases: `(command1 && command2) || (command3 && command4)`.
- Test edge cases: Logical operators with invalid commands, empty parentheses, or syntax errors.

## 6.3 Parentheses for Command Grouping

### 6.3.1 Update the parser to detect parentheses:

- Extend the tokenizer to treat `(` and `)` as special tokens.
- Ensure proper error handling for mismatched or misplaced parentheses.

### 6.3.2 Implement grouped command execution:

- Develop logic to execute commands within parentheses as a single unit.
- Redirect the grouped commands to be executed in a subshell if required.

### 6.3.3 Handle nested parentheses:

- Support multiple levels of nesting (e.g., `((command1 && command2) || command3)`).
- Use a stack or recursive approach to maintain proper evaluation order.

### 6.3.4 Test parentheses functionality:

- Test simple cases: `(command1)`.
- Test nested cases: `((command1 && command2) || (command3 && command4))`.
- Test edge cases: Empty parentheses, invalid grouping, or unbalanced parentheses.

### 6.3.4 Additional Integration Testing

- Combine all bonus features into complex command lines for integration testing:
  - Examples: `ls *.c && (echo "Success" || echo "Failure")`.
  - Handle wildcards, logical operators, and parentheses in the same command line.
- Ensure smooth interaction between the core functionality and bonus features.