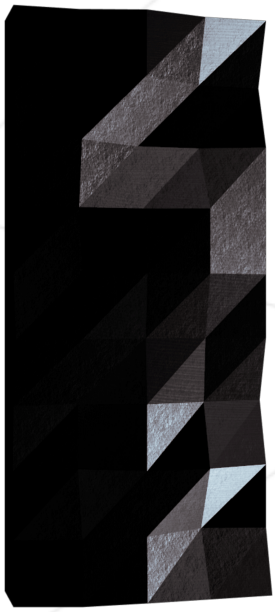

Minishell

As beautiful as a shell

Summary:
This project is about creating a simple shell.
Yes, your very own little Bash.
You will gain extensive knowledge about processes and file descriptors.

Version: 9.0

Contents

Chapter I

Introduction

Shells have existed since the very beginning of IT.

Back then, all developers agreed that communicating with a computer via aligned 1/0 switches was extremely frustrating.

It was only logical that they came up with the idea of creating software to communicate with a computer using interactive command lines in a language somewhat close to human language.

With `Minishell`, you'll travel back in time and experience the challenges developers faced before *Windows* existed.

Chapter II

Common Instructions

- Your project must be written in C.
- Your project must be written in accordance with the Norm. If you have bonus files/functions, they are included in the norm check, and you will receive a `0` if there is a norm error.
- Your functions should not quit unexpectedly (segmentation fault, bus error, double free, etc.) except for undefined behavior. If this occurs, your project will be considered non-functional and will receive a `0` during the evaluation.
- All heap-allocated memory must be properly freed when necessary. Memory leaks will not be tolerated.
- If the subject requires it, you must submit a `Makefile` that compiles your source files to the required output with the flags `-Wall`, `-Wextra`, and `-Werror`, using `cc`. Additionally, your `Makefile` must not perform unnecessary relinking.
- Your `Makefile` must contain at least the rules `$(NAME)`, `all`, `clean`, `fclean` and `re`.
- To submit bonuses for your project, you must include a `bonus` rule in your `Makefile`, which will add all the various headers, libraries, or functions that are not allowed in the main part of the project. Bonuses must be placed in `_bonus.{c/h}` files, unless the subject specifies otherwise. The evaluation of mandatory and bonus parts is conducted separately.
- If your project allows you to use your `libft`, you must copy its sources and its associated `Makefile` into a `libft` folder. Your project's `Makefile` must compile the library by using its `Makefile`, then compile the project.
- We encourage you to create test programs for your project, even though this work **does not need to be submitted and will not be graded**. It will give you an opportunity to easily test your work and your peers' work. You will find these tests especially useful during your defence. Indeed, during defence, you are free to use your tests and/or the tests of the peer you are evaluating.
- Submit your work to the assigned Git repository. Only the work in the Git repository will be graded. If Deepthought is assigned to grade your work, it will occur

after your peer-evaluations. If an error happens in any section of your work during Deepthought's grading, the evaluation will stop.

Chapter III

AI Instructions

Context

During your learning journey, AI can assist with many different tasks. Take the time to explore the various capabilities of AI tools and how they can support your work. However, always approach them with caution and critically assess the results. Whether it's code, documentation, ideas, or technical explanations, you can never be completely sure that your question was well-formed or that the generated content is accurate. Your peers are a valuable resource to help you avoid mistakes and blind spots.

Main message

- Use AI to reduce repetitive or tedious tasks.
- Develop prompting skills — both coding and non-coding — that will benefit your future career.
- Learn how AI systems work to better anticipate and avoid common risks, biases, and ethical issues.
- Continue building both technical and power skills by working with your peers.
- Only use AI-generated content that you fully understand and can take responsibility for.

Learner rules:

- You should take the time to explore AI tools and understand how they work, so you can use them ethically and reduce potential biases.
- You should reflect on your problem before prompting — this helps you write clearer, more detailed, and more relevant prompts using accurate vocabulary.
- You should develop the habit of systematically checking, reviewing, questioning, and testing anything generated by AI.
- You should always seek peer review — don't rely solely on your own validation.

Phase outcomes:

- Develop both general-purpose and domain-specific prompting skills.
- Boost your productivity with effective use of AI tools.
- Continue strengthening computational thinking, problem-solving, adaptability, and collaboration.

Comments and examples:

- You'll regularly encounter situations — exams, evaluations, and more — where you must demonstrate real understanding. Be prepared, keep building both your technical and interpersonal skills.
- Explaining your reasoning and debating with peers often reveals gaps in your understanding. Make peer learning a priority.
- AI tools often lack your specific context and tend to provide generic responses. Your peers, who share your environment, can offer more relevant and accurate insights.
- Where AI tends to generate the most likely answer, your peers can provide alternative perspectives and valuable nuance. Rely on them as a quality checkpoint.

✓ Good practice:

> I ask AI: "How do I test a sorting function?" It gives me a few ideas. I try them out and review the results with a peer. We refine the approach together.

✗ Bad practice:

> I ask AI to write a whole function, copy-paste it into my project. During peer-evaluation, I can't explain what it does or why. I lose credibility — and I fail my project.

✓ Good practice:

> I use AI to help design a parser. Then I walk through the logic with a peer. We catch two bugs and rewrite it together — better, cleaner, and fully understood.

✗ Bad practice:

> I let Copilot generate my code for a key part of my project. It compiles, but I can't explain how it handles pipes. During the evaluation, I fail to justify and I fail my project.

Chapter IV

Mandatory part

Program name	`minishell`
Turn in files	`Makefile, *.h, *.c`
Makefile	`NAME, all, clean, fclean, re`
Arguments	
External functs.	`readline, rl_clear_history, rl_on_new_line, rl_replace_line, rl_redisplay, add_history, printf, malloc, free, write, access, open, read, close, fork, wait, waitpid, wait3, wait4, signal, sigaction, sigemptyset, sigaddset, kill, exit, getcwd, chdir, stat, lstat, fstat, unlink, execve, dup, dup2, pipe, opendir, readdir, closedir, strerror, perror, isatty, ttyname, ttyslot, ioctl, getenv, tcsetattr, tcgetattr, tgetent, tgetflag, tgetnum, tgetstr, tgoto, tputs`
Libft authorized	`Yes`
Description	`Write a shell`

Your shell should:

- Display a **prompt** when waiting for a new command.
- Have a working **history**.
- Search and launch the right executable (based on the `PATH` variable or using a relative or an absolute path).
- Use at most **one global variable** to indicate a received signal. Consider the implications: this approach ensures that your signal handler will not access your main data structures.

`Be careful. This global variable must only store the signal number and must not provide any additional information or access to data. Therefore, using "norm" type structures in the global scope is forbidden.`

- Not interpret unclosed quotes or special characters which are not required by the subject such as \ (backslash) or ; (semicolon).
- Handle ' (single quote) which should prevent the shell from interpreting the meta-characters in the quoted sequence.
- Handle " (double quote) which should prevent the shell from interpreting the meta-characters in the quoted sequence except for $ (dollar sign).
- Implement the following **redirections**:
  - < should redirect input.
  - > should redirect output.
  - << should be given a delimiter, then read the input until a line containing the delimiter is seen. However, it doesn't have to update the history!
  - >> should redirect output in append mode.
- Implement **pipes** (| character). The output of each command in the pipeline is connected to the input of the next command via a pipe.
- Handle **environment variables** ($ followed by a sequence of characters) which should expand to their values.
- Handle $? which should expand to the exit status of the most recently executed foreground pipeline.
- Handle ctrl-C, ctrl-D and ctrl-\ which should behave like in bash.
- In interactive mode:
  - ctrl-C displays a new prompt on a new line.
  - ctrl-D exits the shell.
  - ctrl-\ does nothing.
- Your shell must implement the following **built-in** commands:
  - echo with option -n
  - cd with only a relative or absolute path
  - pwd with no options
  - export with no options
  - unset with no options
  - env with no options or arguments
  - exit with no options

The `readline()` function may cause memory leaks, but you are not required to fix them. However, this **does not mean your own code, yes the code you wrote, can have memory leaks**.

You should limit yourself to the subject description. Anything that is not asked is not required.

If you have any doubt about a requirement, take bash as a reference.

Chapter V

Bonus part

Your program must implement:

- `&&` and `||` with parenthesis for priorities.
- Wildcards `*` should work for the current working directory.

The bonus part will only be evaluated if the mandatory part is completed perfectly. Perfect means the mandatory part is fully implemented and functions without any issues. If you have not passed ALL the mandatory requirements, your bonus part will not be evaluated at all.

Chapter VI

Submission and peer-evaluation

Submit your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double-check the names of your files to ensure they are correct.

During the evaluation, a brief **modification of the project** may occasionally be requested. This could involve a minor behavior change, a few lines of code to write or rewrite, or an easy-to-add feature.

While this step may **not be applicable to every project**, you must be prepared for it if it is mentioned in the evaluation guidelines.

This step is meant to verify your actual understanding of a specific part of the project. The modification can be performed in any development environment you choose (e.g., your usual setup), and it should be feasible within a few minutes — unless a specific timeframe is defined as part of the evaluation.
You can, for example, be asked to make a small update to a function or script, modify a display, or adjust a data structure to store new information, etc.

The details (scope, target, etc.) will be specified in the **evaluation guidelines** and may vary from one evaluation to another for the same project.