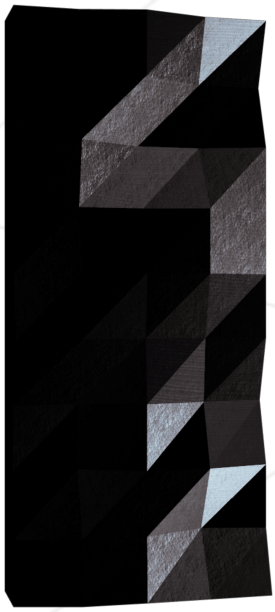

Minishell

As beautiful as a shell

Summary:
This project is about creating a simple shell.
Yes, your own little bash.
You will learn a lot about processes and file descriptors.

Version: 7.1

Contents

I	**Introduction**	**2**
II	**Common Instructions**	**3**
III	**Mandatory part**	**5**
IV	**Bonus part**	**8**
V	**Submission and peer-evaluation**	**9**

Chapter I

Introduction

The existence of shells is linked to the very existence of IT.

At the time, all developers agreed that *communicating with a computer using aligned 1/0 switches was seriously irritating.*

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

Thanks to `Minishell`, you'll be able to travel through time and come back to problems people faced when *Windows* didn't exist.

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 inside.
- Your functions should not quit unexpectedly (segmentation fault, bus error, double free, etc) apart from undefined behaviors. If this happens, your project will be considered non functional and will receive a `0` during the evaluation.
- All heap allocated memory space must be properly freed when necessary. No leaks will be tolerated.
- If the subject requires it, you must submit a `Makefile` which will compile your source files to the required output with the flags `-Wall`, `-Wextra` and `-Werror`, use cc, and your Makefile must not relink.
- Your `Makefile` must at least contain the rules `$(NAME)`, `all`, `clean`, `fclean` and `re`.
- To turn in bonuses to your project, you must include a rule `bonus` to your Makefile, which will add all the various headers, librairies or functions that are forbidden on the main part of the project. Bonuses must be in a different file `_bonus.{c/h}` if the subject does not specify anything else. Mandatory and bonus part evaluation is done separately.
- If your project allows you to use your `libft`, you must copy its sources and its associated `Makefile` in a `libft` folder with its associated Makefile. 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 **won't have to be submitted and won't be graded**. It will give you a chance to easily test your work and your peers' work. You will find those 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 your assigned git repository. Only the work in the git repository will be graded. If Deepthought is assigned to grade your work, it will be done

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

Chapter III

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).
- Avoid using more than **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 cannot provide any other information or data access than the number of a received signal. 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 **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 **builtins**:
  - 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 can cause memory leaks. You don't have to fix them. But that **doesn't 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 IV

Bonus part

Your program has to implement:

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

The bonus part will only be assessed if the mandatory part is PERFECT. Perfect means the mandatory part has been integrally done and works without malfunctioning. If you have not passed ALL the mandatory requirements, your bonus part will not be evaluated at all.

Chapter V

Submission and peer-evaluation

Turn in 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.

M L Q * d z 0 k v P
:)? e Q 6 u nk] R *
f *q lV - 0 , ?ZP
7 c 1V H
e 9 () 8
U
' > D s / w
; E ' > M
q < >

z b H (3
l X 0g E 6