Minishell

As beautiful as a shell

Summary: *The objective of this project is for you to create a simple shell. Yes, your own little bash or zsh. You will learn a lot about processes and file descriptors.*

Contents

Chapter I

Introduction

The existence of shells is linked to the very existence of IT. At the time, all coders 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 `to communicate with a computer using interactive lines of commands in a language somewhat close to english`.

With `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 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`, 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}`. 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	Yes
Arguments	
External functs.	printf, malloc, free, write, open, read, close, fork, wait, waitpid, wait3, wait4, signal, kill, exit, getcwd, chdir, stat, lstat, fstat, execve, dup, dup2, pipe, opendir, readdir, closedir, strerror, errno, termcap functions
Libft authorized	Yes
Description	Write a shell

Your shell should:

- Not use more than one global variable, think about it and be ready to explain why you do it.

- Show a prompt when waiting for a new command

- Search and launch the right executable (based on the PATH variable or by using relative or absolute path) like in bash

- It must implement the builtins like in bash:

    ○ `echo` with option '-n'

    ○ `cd` with only a relative or absolute path

    ○ `pwd` without any options

    ○ `export` without any options

    ○ `unset` without any options

    ○ `env` without any options and any arguments

    ○ `exit` without any options

- ; in the command should separate commands like in bash

- ' and " should work like in bash except for multiline commands

- Redirections < > ">>" should work like in bash except for file descriptor aggregation

- Pipes | should work like in bash except for multiline commands

- Environment variables ($ followed by characters) should work like in bash

- $? should work like in bash

- ctrl-C, ctrl-D and ctrl-\ should have the same result as in bash

- Use up and down arrows to navigate through the command using termcap (mandatory) history which we will then be able to edit (at least like we can for classic lines) if we feel like it (the line, not the history).

Chapter IV

Bonus part

- If the Mandatory part is not perfect don't even think about bonuses

- You don't need to do all the bonuses

- Redirection "`<<`" like in bash

- Advance history and line editing with Termcaps (`man tgetent` for examples)

  - Edit the line where the cursor is located.

  - Move the cursor left and right to be able to edit the line at a specific location. Obviously new characters have to be inserted between the existing ones similarly to a classic shell.

  - Cut, copy, and/or paste all or part of a line using the key sequence you prefer.

  - Move directly by word towards the left or the right using `ctrl+LEFT` and `ctrl+RIGHT`.

  - Go directly to the beginning or the end of a line by pressing `home` and `end`.

  - Write AND edit a command over a few lines. In that case, we would love that `ctrl+UP` and `ctrl+DOWN` allow to go from one line to another in the command while remaining in the same column or otherwise the most appropriate column.

- `&&`, `||` with parenthesis for priorities, like in bash

- wilcard `*` like in bash