TP2: Input/Output

Reading and writing in a file : `fopen`, `fclose`,...

a. Write a program `readfile` that takes one argument – the name of a file – and prints the file line by line. Be careful to deal with the different errors that may happen (no argument, the file does not exist, is not readable,...)

Useful functions : `fopen`, `fgets`, `perrror`,...

b. Modify your program to indicate at the end the number of lines, characters and words.

c. Modify your program to write the number of words of each line in a new file (the output will be a file where each line will be "line_number number_of_words").

Program with multiple source-files

Unless a program is very small, it is often more convenient to write it separating the code in different files.

a. Separate your program in three files : one containing the functions, one containing the main function and one header file.

b. Modify your Makefile so that it only compiles the files that are necessary.

In the following, remember to write your code in separated files.

Command-line interpreter : part one

The aim of the next sessions will be to write a command-line interpreter which will be a simplified version of the shell you are using. The idea is to implement the following algorithm :

```
while ( the_user_does_not_want_to_quit )
   print a prompt
   read the line typed
   execute it and print the result
```

The aim of our work is to be as complete as possible but more important, we want to write a program with as less bug as possible. In particular, it is more important to deal with the different errors than to write a very complicated program. If you are quick enough, your interpreter will be able to :

1. contain build-in functions (ex : `cd my_dir` or `echo 1` or `quit`)
2. execute an external program with its arguments (ex : `ls -l my_directory`)
3. pipe the result of a program to an other program (ex : `cat my_file | grep hello`)
4. write the result of a program in a file (ex : `ls > my_file`)
5. ... And more functions if you want (deal with &, ;, add an history,...)

We will implement the functions step by step. At first, we begin by step 0 : print the prompt, wait for the answer and store the line that the program should interpret. We will assume for now that commands do not contain any & or > or other special characters and we will only consider lines of less than 255 characters.

a. Write a program that prints a prompt, waits for the user to type one line and then prints "you want to execute x" (where x is the line of the user) until the user enters the command "quit".

A command may have 0 or more arguments, each of them are separated by one or more whitespace. For example the arguments of the function `cp -r dir1 dir2` are {cp, -r, dossier1, dossier2, NULL}.

b. Implement a function `char** command_args(char * a_line)` where `a_line` is a command line and the result is an array containing the different arguments. Be aware of multiple spaces.

In most cases, execute a command means executing an external program (and we will implement that later). In some cases, the command is just an internal function.

c. Implement the functions `quit`, `cd` and `echo`.

d. (bonus) Modify the prompt to print the current directory.