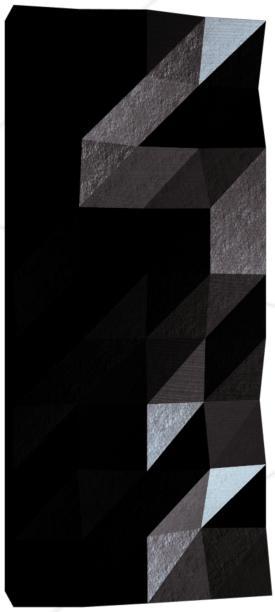

get_next_line

Reading a line on a fd is way too tedious

Pedago pedago@42.fr

Summary: The aim of this project is to make you code a function that returns a line ending with a newline, read from a file descriptor.

Contents

Chapter I

Foreword

Neill Blomkamp (born 17 September 1979) is a South African-Canadian film director, film producer, screenwriter, and animator. Blomkamp employs a documentary-style, hand-held, cinéma vérité technique, blending naturalistic and photo-realistic computer-generated effects. He is best known as the co-writer and director of the critically acclaimed and financially successful science fiction film District 9 and the dystopian science fiction film Elysium, which garnered moderately positive reviews and a good box office return. He is also known for his collaborations with South African actor Sharlto Copley. He is based in Vancouver, British Columbia.

Time named Blomkamp as one of the 100 Most Influential People of 2009. Forbes magazine named him as the 21st most powerful celebrity from Africa.

Figure I.1: Neill Blomkamp

Chapter II

Introduction

You are now starting to understand that it will get tricky to read data from a file descriptor if you don't know its size beforehand. What size should your buffer be? How many times do you need to read the file descriptor to retrieve the data ?

It is perfectly normal and natural that, as a programmer, you would want to read a "line" that ends with a line break from a file descriptor. For example each command that you type in your shell or each line read from a flat file.

Thanks to the project `get_next_line`, you will finally be able to write a function that will allow you to read a line ending with a newline character from a file descriptor. You'll be able to add this function to your `libft` if you feel like it and most importantly, use it in all the future projects that will require it.

Chapter III

Objectives

This project will not only allow you to add a very convenient function to your collection, but it will also allow you to learn a highly interesting new concept in `C` programming: static variables.

You will also gain a deeper understanding of allocations, whether they happen on the stack memory or in the heap memory, the manipulation and the life cycle of a buffer, the unexpected complexity implied in the use of one or many static variables.

Your respect of the Norm will improve the rigor of your programming. We also suspect that your approach to coding will change when you will discover that the initial state of a variable in a function can vary depending on the call of that very function.

Chapter IV

General Instructions

- You must only submit two files : `get_next_line.c` and `get_next_line.h`
- If you are clever, you will use your libft. If so, submit your folder `libft` at the root of your repository.
- There cannot be a `main` function in your program.
- Do not push a `Makefile`.
- Your project must be written in accordance with the Norm.
- You have to handle errors carefully. In no way can your program, or in this particular case your function, quit in an unexpected manner (Segmentation fault, bus error, double free, etc).
- All heap allocated memory space must be properly freed when necessary.
- You must submit a file called `author` containing your username followed by a '\n' at the root of your repository:

```
$>cat -e author
xlogin$
```

- If you choose to submit this project using your library `libft`, it is strictly `FORBIDDEN` to bypass the limitation of the Norm by adding some specific functions from your `get_next_line`. That would be considered cheating during your defence. Your `get_next_line` must hold in 5 functions of 25 lines max. The respect of this instruction will be carefully checked during your defence. There is no need for you ask permission to the staff to add a function to your library. Use your head instead and ask yourself if your function respects this instruction. If you do everything right, you will of course be encouraged to expand your library with generic functions. You'll discover their usage as you complete this project.
- The `libc` functions allowed on this project are `read`, `malloc` and `free`.

Chapter V

Mandatory part

- Write a function that returns a line read from a file descriptor.
- What we call a “line” is a succession of characters that end with ’\n’ (ascii code 0x0a) or with End Of File (EOF).

https://latedev.wordpress.com/2012/12/04/all-about-eof/ : taht article can prove useful for someone who can read.

- Your function must be prototyped as follow :

```
int     get_next_line(const int fd, char **line);
```

- The first parameter is the file descriptor that will be used to read.
- The second parameter is the address of a pointer to a character that will be used to save the line read from the file descriptor.
- The return value can be 1, 0 or -1 depending on whether a line has been read, when the reading has been completed, or if an error has happened respectively.
- Your function get_next_line must return its result without ’\n’.
- Calling your function get_next_line in a loop will then allow you to read the text available on a file descriptor one line at a time until the end of the text, no matter the size of either the text or one of its lines.
- Make sure that your function behaves well when it reads from a file, from the standard output, from a redirection etc.
- In you header file get_next_line.h you must have at least the prototype of the function get_next_line and a macro that allows to choose the size of the reading

buffer for the `read` function. This value will be modified during the defence to evaluate the strength of your function. That macro `must` be named BUFF_SIZE. For example:

```
#define BUFF_SIZE 32
```

Is your function still working if your BUFF_SIZE value is 9999 ? And if BUFF_SIZE is valued 1 ? And 10000000 ? Do you know why ?

- We consider that `get_next_line` has an undefined behavior if, between two calls, the same file descriptor designs two distinct files although the reading from the first file was not completed.
- We consider also that a call to `lseek(2)` will never take place between two calls of the function `get_next_line` on the same file descriptor.
- Finally we consider that `get_next_line` has an undefined behavior when reading from a binary file. However, if you wish, you can make this behavior coherent.
- Global variables are forbidden.
- Static variables are allowed.

To know what is a static variable is a good start: https://en.wikipedia.org/wiki/Static_variable

Chapter VI

Bonus part

The project `get_next_line` is straight forward and leaves very little room for bonuses, but I am sure that you have a lot of imagination. If you ace perfectly the mandatory part, then by all means complete this bonus part to go furher. I repeat, no bonus will be taken into consideration if the mandatory part isn't perfect.

- To succeed `get_next_line` with a single static variable.
- To be able to manage multiple file descriptor with your `get_next_line`. For example, if the file descriptors 3, 4 and 5 are accessible for reading, then you can call `get_next_line` once on 3, once on 4, once again on 3 then once on 5 etc. without losing the reading thread on each of the descriptors.

Chapter VII

Submission and peer correction

Submit your work on your `GiT` repository as usual. Only the work on your repository will be graded during defence and by the Deepthought.

Deepthought will go over your repository once you're finished with defence. It will compile like this :

```
$> make -C libft/ fclean && make -C libft/
```

Then :

```
$> clang -Wall -Wextra -Werror -I libft/includes -o get_next_line.o -c get_next_line.c
$> clang -Wall -Wextra -Werror -I libft/includes -o main.o -c main.c
$> clang -o test_gnl main.o get_next_line.o -I libft/includes -L libft/ -lft
```

Make sure that your repository compiles in the same way (obviously the `main` will be ours).

Good luck everyone !