



Get Next Line

Reading a line from a file descriptor is far too tedious.

Summary:

This project is about programming a function that returns a line read from a file descriptor.

Version: 12.01

Contents

I	Goals	2
II	Common Instructions	3
III	Mandatory part	5
IV	Bonus part	7
V	Submission and peer-evaluation	8

Chapter I

Goals

This project will not only allow you to add a highly useful function to your collection, but it will also teach you an important concept in C programming: static variables.

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, libraries 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

Function name	<code>get_next_line</code>
Prototype	<code>char *get_next_line(int fd);</code>
Turn in files	<code>get_next_line.c</code> , <code>get_next_line_utils.c</code> , <code>get_next_line.h</code>
Parameters	<code>fd</code> : The file descriptor to read from
Return value	Read line: correct behavior NULL: there is nothing else to read, or an error occurred
External functs.	<code>read</code> , <code>malloc</code> , <code>free</code>
Description	Write a function that returns a line read from a file descriptor

- Repeated calls (e.g., using a loop) to your `get_next_line()` function should let you read the text file pointed to by the file descriptor, **one line at a time**.
- Your function should return the line that was read.
If there is nothing left to read or if an error occurs, it should return `NULL`.
- Make sure that your function works as expected both when reading a file and when reading from the standard input.
- **Please note** that the returned line should include the terminating `\n` character, except when the end of the file is reached and the file does not end with a `\n` character.
- Your header file `get_next_line.h` must at least contain the prototype of the `get_next_line()` function.
- Add all the helper functions you need in the `get_next_line_utils.c` file.



A good start would be to know what a `static variable` is.

- Because you will have to read files in `get_next_line()`, add this option to your compiler call: `-D BUFFER_SIZE=n`
It will define the buffer size for `read()`.
The buffer size value will be adjusted by your peer evaluators and the Moulinette to test your code.



We must be able to compile this project with and without the `-D BUFFER_SIZE` flag in addition to the usual flags. You may choose any default value you prefer.

- You will compile your code as follows (a buffer size of 42 is used as an example):
`cc -Wall -Wextra -Werror -D BUFFER_SIZE=42 <files>.c`
- `get_next_line()` exhibits undefined behavior if the file associated with the file descriptor changes after the last call, while `read()` has not yet reached the end of the file.
- `get_next_line()` also exhibits undefined behavior when reading a binary file. However, you can implement a logical way to handle this behavior if you want to.



Does your function still work if the `BUFFER_SIZE` value is 9999? If it is 1? 10000000? Do you know why?



Read as little data as possible each time `get_next_line()` is called. If a newline character is encountered, return the current line immediately.
Don't read the whole file and then process each line.

Forbidden

- You are not allowed to use your `libft` in this project.
- `lseek()` is forbidden.
- Global variables are forbidden.

Chapter IV

Bonus part

This project is straightforward and does not support complex bonus features. However, we trust your creativity. If you have completed the mandatory part, consider attempting this bonus section.

Here are the bonus part requirements:

- Develop `get_next_line()` using only one static variable.
- Your `get_next_line()` can manage multiple file descriptors at the same time. For example, if you are reading from file descriptors 3, 4, and 5, you should be able to read from a different file descriptor with each call, without losing track of the reading state of each file descriptor or returning a line from a different one. This means you should be able to call `get_next_line()` to read from fd 3, then fd 4, then fd 5, then again from fd 3, then fd 4, and so forth, without losing track of the reading state for each file descriptor.

Append the `_bonus.[c|h]` suffix to the bonus part files.

It means that, in addition to the mandatory part files, you will turn in the 3 following files:

- `get_next_line_bonus.c`
- `get_next_line_bonus.h`
- `get_next_line_utils_bonus.c`



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

Submit your assignment in your `Git` repository as usual. Only the content inside your repository will be evaluated during the defense. Make sure to double-check the file names for accuracy.



When writing your tests, remember that:

1) Both the buffer size and the line size can be of very different values.

2) A file descriptor does not only point to regular files.

Be thorough and cross-check your work with your peers. Prepare a comprehensive set of diverse tests for the defense.

Once passed, do not hesitate to add your `get_next_line()` to your `libft`.



`/=ð/\^[](_)$ /\^@|V †|-|@~|~ /-!/570@1<|-£1_~/ ¢@/\^ε vv!7}{ ???`