Version: 1.0

Get Next Line

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

Summary

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

#C #Imperative #FD

Intellectual Property Disclaimer

All content presented in this training module, including but not limited to texts, images, graphics, and other materials, is protected by intellectual property rights held by Association 42.

Terms of Use:

- **Personal use:** You are permitted to use the contents of this module solely for personal purpose. Any commercial use, reproduction, distribution, modification, or public display is strictly prohibited without prior written permission from Association 42.
- **Respect for Integrity:** You must not alter, transform, or adapt the content in any way that could harm its integrity.

Protection of Rights:

Any violation of these terms constitutes an infringement of intellectual property rights and may result in legal action. We reserve the right to take all necessary measures to protect our rights, including but not limited to claims for damages.

For any questions regarding the use of the content or to obtain authorization, please contact: `legal@42.fr`

Contents

Chapter 1

Introduction

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

Chapter 2

Common Instructions

- Your project must be written in C.
- Your project must be written in accordance with the Norm. If you have bonus files or 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.) except for 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 memory leaks will be tolerated.
- If the subject requires it, you must submit a `Makefile` that will compile your source files to the required output with the flags `-Wall`, `-Wextra`, and `-Werror`, using `cc`, and your `Makefile` must not relink.
- Your `Makefile` must at least contain the rules `$(NAME)`, `all`, `clean`, `fclean`, and `re`.
- If your project allows you to use your `libft`, you must copy its sources and its associated `Makefile` into a `libft` folder. 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 **will not be submitted and will not be graded**. It will give you a chance to easily test your work and your peers' work. You will find these tests especially useful during your defense. Indeed, during the defense, 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 occurs in any section of your work during Deepthought's grading, the evaluation will stop.

Chapter 3

AI Instructions

Context

This project is designed to help you discover the fundamental building blocks of your 42 training.

To properly anchor key knowledge and skills, it's essential to adopt a thoughtful approach to using AI tools and support.

True foundational learning requires genuine intellectual effort — through challenge, repetition, and peer-learning exchanges.

For a more complete overview of our stance on AI — as a learning tool, as part of the 42 training, and as an expectation in the job market — please refer to the dedicated FAQ on the intranet.

Main message

- Build strong foundations without shortcuts.
- Really develop tech & power skills.
- Experience real peer-learning, start learning how to learn and solve new problems.
- The learning journey is more important than the result.
- Learn about the risks associated with AI, and develop effective control practices and countermeasures to avoid common pitfalls.

Learner rules:

- You should apply reasoning to your assigned tasks, especially before turning to AI.
- You should not ask for direct answers to the AI.
- You should learn about 42 global approach on AI.

Phase outcomes:

Within this foundational phase, you will get the following outcomes:

- Get proper tech and coding foundations.
- Know why and how AI can be dangerous during this phase.

Comments and example:

- Yes, we know AI exists — and yes, it can solve your projects. But you're here to learn, not to prove that AI has learned. Don't waste your time (or ours) just to demonstrate that AI can solve the given problem.
- Learning at 42 isn't about knowing the answer — it's about developing the ability to find one. AI gives you the answer directly, but that prevents you from building your own reasoning. And reasoning takes time, effort, and involves failure. The path to success is not supposed to be easy.
- Keep in mind that during exams, AI is not available — no internet, no smartphones, etc. You'll quickly realise if you've relied too heavily on AI in your learning process.
- Peer learning exposes you to different ideas and approaches, improving your interpersonal skills and your ability to think divergently. That's far more valuable than just chatting with a bot. So don't be shy — talk, ask questions, and learn together!
- Yes, AI will be part of the curriculum — both as a learning tool and as a topic in itself. You'll even have the chance to build your own AI software. In order to learn more about our crescendo approach you'll go through in the documentation available on the intranet.

✓ Good practice:

> I'm stuck on a new concept. I ask someone nearby how they approached it. We talk for 10 minutes — and suddenly it clicks. I get it.

✗ Bad practice:

> I secretly use AI, copy some code that looks right. During peer evaluation, I can't explain anything. I fail. During the exam — no AI — I'm stuck again. I fail.

Chapter 4

Mandatory part

Function Name	`get_next_line`
Prototype	`char *get_next_line(int fd);`
Files to Submit	`get_next_line.c, get_next_line_utils.c, get_next_line.h`
Parameters	`fd: The file descriptor to read from`
Return Value	`Read line: correct behavior` `NULL: there is nothing else to read, or an error occurred`
External Functions	`read, malloc, free`
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 else to read or if an error occurred, 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 if the end of file was reached and 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 modified by your peer-evaluators and the Moulinette in order 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 can choose the default value of your choice.

- 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`
- We consider that `get_next_line()` has an undefined behavior if the file pointed to by the file descriptor changed since the last call whereas `read()` didn't reach the end of file.
- We also consider that `get_next_line()` has an 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?

Try to read as little as possible each time `get_next_line()` is called. If you encounter a new line, you have to return the current line.
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 5

Readme Requirements

A README.md file must be provided at the root of your Git repository. Its purpose is to allow anyone unfamiliar with the project (peers, staff, recruiters, etc.) to quickly understand what the project is about, how to run it, and where to find more information on the topic.
The README.md must include at least:

- The very first line must be italicized and read: *This project has been created as part of the 42 curriculum by <login1>[, <login2>[, <login3>[...]]].*
- A "**Description**" section that clearly presents the project, including its goal and a brief overview.
- An "**Instructions**" section containing any relevant information about compilation, installation, and/or execution.
- A "**Resources**" section listing classic references related to the topic (documentation, articles, tutorials, etc.), as well as a description of how AI was used — specifying for which tasks and which parts of the project.

➠ **Additional sections may be required depending on the project** (e.g., usage examples, feature list, technical choices, etc.).

Any required additions will be explicitly listed below.

- A detailed explanation and justification of the algorithm selected for this project must also be included.

The choice of language is at your discretion. It is recommended to write in English, but it is not mandatory.

Chapter 6

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.

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 smart and cross-check with your peers. Prepare a full set of diverse tests for defense.

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

During the evaluation, a brief **modification of the project** may occasionally be requested. This could involve a minor behavior change, a few lines of code to write or rewrite, or an easy-to-add feature.

While this step may **not be applicable to every project**, you must be prepared for it if it is mentioned in the evaluation guidelines.

This step is meant to verify your actual understanding of a specific part of the project. The modification can be performed in any development environment you choose (e.g., your usual setup), and it should be feasible within a few minutes — unless a specific timeframe is defined as part of the evaluation.
You can, for example, be asked to make a small update to a function or script, modify a display, or adjust a data structure to store new information, etc.

The details (scope, target, etc.) will be specified in the **evaluation guidelines** and may vary from one evaluation to another for the same project.