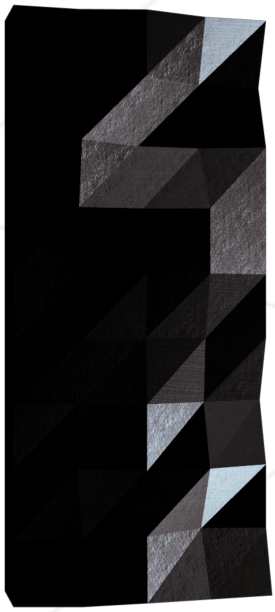

Get Next Line

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: 13

Contents

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.
- Your functions should not quit unexpectedly (segmentation fault, bus error, double free, etc.) except for undefined behavior. If this occurs, your project will be considered non-functional and will receive a `0` during the evaluation.
- All heap-allocated memory must be properly freed when necessary. Memory leaks will not be tolerated.
- If the subject requires it, you must submit a `Makefile` that compiles your source files to the required output with the flags `-Wall`, `-Wextra`, and `-Werror`, using `cc`. Additionally, your `Makefile` must not perform unnecessary relinking.
- Your `Makefile` must contain at least the rules `$(NAME)`, `all`, `clean`, `fclean` and `re`.
- To submit bonuses for your project, you must include a `bonus` rule in your `Makefile`, which will add all the various headers, libraries, or functions that are not allowed in the main part of the project. Bonuses must be placed in `_bonus.{c/h}` files, unless the subject specifies otherwise. The evaluation of mandatory and bonus parts is conducted separately.
- 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 **does not need to be submitted and will not be graded**. It will give you an opportunity to easily test your work and your peers' work. You will find these 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 the assigned Git repository. Only the work in the Git repository will be graded. If Deepthought is assigned to grade your work, it will occur

after your peer-evaluations. If an error happens in any section of your work during Deepthought's grading, the evaluation will stop.

Chapter III

AI Instructions

Context

This project is designed to help you discover the fundamental building blocks of your ICT 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 ICT curriculum, 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 IV

Mandatory part

Function name	`get_next_line`
Prototype	`char *get_next_line(int fd);`
Turn in files	`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 functs.	`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 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 is modified 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 V

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 VI

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`.

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.

/=∂/\/\[](_)§ /\/\@|V †|-|@¯|¯ /-/!570®1<|-\£1_`/ ¢@/\/\ε vv!7}{ ???