get_next_line: Reading from a File Descriptor

By Mia Combeau
In 42 School Projects
January 22, 2022
18 Min read
Add comment

One of the first projects at 42 school is get_next_line. The goal is to create a function that can return a single line read from a file descriptor, without loosing track of the next line and with a random buffer size.

This is not a step-by step tutorial and there will be no ready-made solutions. It is a walkthough, a guide to explore the issues and concepts behind the subject, the possible approaches we might take, and some tips to test our code.

In this project, we must write the get_next_line function so that it:

- can read the entire contents of a file, one line at a

time, with successive calls to the get_next_line function,
- returns the contents of the read line, followed by the newline character `\n` (unless we're at the end of the file and it does not end with a `\n`),
- returns `NULL` when there is nothing more to read, or in case of an error,
- can read from a file, but also from the standard input.

Read the subject [pdf] at the time of this writing.

Table of Contents

Concepts Behind get_next_line

There are two important concepts to master in order to accomplish this project. The first is, of course, the manipulation of file descriptors (fd) and the reading of files. But the subject also insists on static variables, which will quickly reveal themselves central to the smooth operation of get_next_line.

File Descriptors (fd)

To manipulate a file in C, we must first inform the operating system of our intentions with the `open` function of the `fcntl.h` library. In this system call, we need to specify the path towards the file we'd like to open, as well as the way in which we want to access

it with flags, like this:

```
open("file.txt", O_RDONLY); // read only
open("file.txt", O_WRONLY); // write only
open("file.txt", O_RDWR);   // read write
```

The system ensures that the file does exist and that we have the correct permissions to open it. If all goes well, the system sends a small non-negative integer called a file descriptor (or fd, for short). From now on, we will only use this file descriptor to refer to the file, instead of its name.

Note: file descriptors 0, 1, and 2 are reserved for the standard input, the standard output, and the standard error output respectively. Our file might be assigned a number 3 or higher. That will depend on how many files the operating system already has opened.

If there is any issue, for exemple if we ask to open a file that doesn't exist, or a file we don't have the permission to open, the function will return -1.

Finally, when we are done manipulating a file, we must de-reference the file descriptor with the `close` function of the `unistd.h` library.

Here is a basic example of opening and closing a file in C:

```c
#include <fcntl.h>
#include <unistd.h>

int     main(void)
{
        int     fd;

//      We open the file and assign the returned fi
//      to the fd variable in order to use it later
        fd = open("fichier.txt", O_RDONLY);
//      Then we make sure the file was correctly ope
        if (fd == -1)
                return (-1);
//      ...
//      And we close the file with the reference to
        close(fd);
        return (0);
}
```

The `get_next_line` function won't have to open or close any file, though. It only has to worry about reading from a file descriptor it receives as a parameter. For that, it will use the `read` function in order to... well, read the contents of a file.

Reading a File

The `read` function, from the `unistd.h` library, loads the contents of a file into memory, in part or in full, using its file descriptor. Here is its prototype:

```c
ssize_t read(int fd, void *buf, size_t count);
```

Its parameters as well as how we are going to make use of them in get_next_line are as follows:

- the file descriptor. This is the `fd` we receive as a parameter in the `get_next_line` function,
- a buffer, a pointer towards a memory area where we can temporarily store the characters we read,
- a size in bytes to read, in other words, the number of characters to read. In this project, it will be our `BUFFER_SIZE`, which will only be defined at compilation time.

The `read` function also returns the number of characters that it has read, or -1 in case of error.

Let's take a moment to note that the read function stops reading the instant it gets to the given number of characters, or the end of the file (EOF). It will remember the position of the last read character, so if we call it again later with the same file descriptor, it will resume reading where it left off, or stay at the end of file if it has already reached it. Like a handy little bookmark!

The BUFFER_SIZE

In get_next_line, we can't just read characters one by one until we hit a `\n`. We are forced to read a

certain number of characters at once, determined by a `BUFFER_SIZE` that we don't know in advance. For instance, a file with the following text has 34 total characters:

```
1 = one
2 = two
3 = three
4 = four
```

If we have a `BUFFER_SIZE` of 1, we will have to call the `read` function 35 times to read the entire text. In this case, we can relatively easily stop at the 8th call, when we hit a `\n` and return the first line. Then `read` will be at the correct position if we call `get_next_line` a second time to get the second line.

However, if our `BUFFER_SIZE` is 100, we will read the entire text with a single call to the `read` function. In that case, all we have to do is get the characters before the `\n` from the buffer and return that, right?

Yes, but that won't be enough: what is going to happen when we call get_next_line again to get the second line of our text? Nothing. The `read` function will be at the end of the file with nothing left to read. And the rest of the characters in the buffer will have disappeared...

Static Variables

Indeed, regular local variables (like our buffer) are very short-lived. The minute the function in which they were declared ends, they are de-referenced from the stack and slip out of memory.

A static variable doesn't lose its memory that easily. Its data persists until the end of the program, whether or not the function in which it was declared ends. In this project, a static variable is necessary to store the extra characters, read after a `\n`. That way, when we call `get_next_line` again to get the next line, those characters are not lost.

Exemple

Here is an example of a simple program in which a local static variable does not lose its memory no matter how many times the function in which it was declared is called.

```
#include <stdio.h>

void function(void)
{
        static int      x;
//      The default value of a static variable is 0

        printf("x = %d", x);
        x = x + 1;
}
int main(void)
{
        int i;
```

```
        i = 0;
        while (i <= 10)
        {
                function();
                i++;
        }
        return (0);
} //     It is only here, at the end of the program,
//      that the static variable ceases to exist
```

Output

At each function call, the static variable `x` never loses its value:

```
x = 0
x = 1
x = 2
x = 3
x = 4
x = 5
x = 6
x = 7
x = 8
x = 9
x = 10
```

If the characters saved in our static variable since the last read don't contain a `\n` and that `read` is not at the end of the file, all we need to do is read again and add the new read characters. Then, if there is a `\n` in

the static string, we will be able to extract the part before the `\n` to return it as a line, all the while saving the rest of the characters in the static variable, over and over.

Laying Out the get_next_line Code

Now that we have a firm grip on the concepts, it is time to organize our code. There are of course as many ways to code get_next_line as there are programmers, what follows is only one method out of many!

We can break down get_next_line into a few main functions.

A Function to Read From the File Descriptor

This function will read from the file descriptor in a loop until we detect a `\n` or the end of the file:

- create a reading loop that stops when the read function returns 0, which means we are at the end of the file and there is nothing left to read,

    - `read` from the file descriptor,
    - if `read` returns -1, there was an error, free all allocated memory and stop everything,
    - save the read characters in a static variable (`strjoin` the buffer to the static),
    - check to see if there is a `\n` in the static variable: if yes, stop the loop, if no, continue.

A Function to Get the Line to Return

If this function is called, it means we know for a fact that there is a `\n` in the static variable (or that there is nothing left to read in the file). We will extract the characters up to the `\n` to get the line we must later return:

- count the number of characters to and including the `\n` if there is one,
- malloc a string of the counted size, + 1 for the final `\0`,
- copy the characters until the `\n` or the `\0` and add the final `\0`,
- return the malloc'd string.

A Function to Save the Extra Read Characters

As with the previous function, by the time we call this function, we know that there is a `\n` in the static variable. If we have reached the end of the file, there is no need to call this function because we will end up returning the last line with the previous function. We must now "reset" the static variable by taking all the characters after the `\n`:

- measure the size of the string in the static variable, minus the number of characters until the `\n` (included!), + 1 for the final `\0`,
- malloc a string of that size,
- copy the characters and add the final `\0`,
- return the new malloc'd string.

The get_next_line Function

The get_next_line function will coordinate and call the other functions:

- declare the static variable,
- if the file descriptor `fd` or the `BUFFER_SIZE` is invalid, return `NULL` right away,
- call the reading function,
- if the static variable is not null or empty:
  - assign the result of the function to save the extra characters to the static variable,
  - assign the result of the function to get the line to another variable that we will return shortly,

- if the variable we want to return, the line, is null or empty, free any allocated memory and return `NULL`,
- otherwise, return the line.

Now, all that's left to do is code all that in our own way! Without forgetting to test everything as we go...

Tips for Testing get_next_line

A get_next_line's `main.c` must first `open` a file to get the file descriptor to feed into the `get_next_line` function and then print out the result. A simple example would be:

```
 1 #include <stdio.h>
 2 #include <fcntl.h>
 3 #include <unistd.h>
 4 #include "get_next_line.h"
 5
 6 int     main(int argc, char **argv)
 7 {
 8         int     fd;
 9         char    *line;
10
11         (void)argc;                     //
   Casting argc to quiet the compiler's
   warnings.
12         fd = open(argv[1], O_RDONLY);   //
```

```
    Open the file given as an argument at
    program launch
13          line = "";                          //
    Initialize this variable to be able to use
    it in the following loop
14          while (line != NULL)
15          {
16                  line = get_next_line(fd);
17                  printf("%s", line);     //
    We're omitting any \n in this printf because
    get_next_line is supposed to include the \n
    for each line
18          }
19          fd = close(fd);
20          return (0);
21  }
```

Then, we compile (many times, with many different `BUFFER_SIZE`s!):

```
gcc -Werror -Wextra -Wall -D BUFFER_SIZE=42
get_next_line.c get_next_line_utils.c main.c
```

Next, we can create test files, for example a file with a poem, a completely empty file or a file containing only a single `\n`. We might also simply give our program one of our `.c` files, like `get_next_line.c`.

```
./a.out get_next_line.c
```

But that's not all! We also need to test the standard input, and that is easy enough to do with:

```
./a.out /dev/random
```

No worries, it is meant to endlessly display random garbage like this:

```
B}66ńn)
V@vkm@z,7@9N@^ﾍe),@<=@3@@@V@o@z@Pi@@@S@ﾉ@}
@ɣ@@@ﾞ@ȯMɾ;@@@pX5戠b@@z*r@@|@醹@@E+@[2@X@$@Ú@nS@*@m@
@@@@@@@@"u@d@L@WF@@t<+@@@@g
```

The file `/dev/random` is a special one in Unix type operating systems: it generates random numbers. And since we are interpreting those numbers as characters... If it doesn't segfault, it's a victory. A victory we can kill at any time with `ctrl-c`.

We can also test the standard input this way :

```
./a.out /dev/tty
```

The program will start running and wait for input. The minute we are done typing and press the enter key, the program should print out the line we wrote, and then the line returned by get_next_line. They should both be identical. Again, we can stop this process with `ctrl-c`.

```
Hello!
Hello!
How are you?
```

```
How are you?
```

Finally, once we have done all our own testing, we might, as a last resort, just to make sure we haven't overlooked anything, run Tripouille's gnlTester [Github].

If our function passes all these tests with flying colors, the moulinette should be happy. Still, we must verify that we have correctly secured and freed all of our mallocs and that we have paid close attention to the following problems. It is not the sort of thing that will pass by an attentive corrector!

Common Issues with get_next_line

While coding get_next_line, there are two small issues we typically run into: a cute little *stack overflow* and a few misfiring `free`s.

The Buffer and BUFFER_SIZE Problem: Stack Overflow

Once we've coded `get_next_line` so that it works properly, we will very quickly realize that there is

still a problem to fix. If we try an extremely large `BUFFER_SIZE`, like 2147483647, we might notice a fatal error : *stack overflow*.

At 42, we can't allow this type of crash, even if in practice, when will we ever ask for that huge a buffer? The better question is simply: why?

This is because of the buffer variable, in which the `read` function stores the characters it has read. This variable is declared and initialized in the stack, which is the memory area in RAM where all regular local variables are stored and referenced by default. But the stack is not infinite, it has a limit. That limit varies from one operating system to the next, and even from one machine to the next. By default, it's between 1MB on Windows and 8MB on Linux. Its value can be adjusted, but that's unnecessary and outside the scope of this project.

This means that if we declare our buffer this way, for example:

```
//      ...
        char    buffer[BUFFER_SIZE + 1];
//      ...
        reader = read(fd, buffer, BUFFER_SIZE);
//      ...
```

There will inevitably be a problem when our `BUFFER_SIZE` exceeds the stack limit, which will no

longer be able to contain our enormous variable. So what can we do about it?

Limiting the Buffer Size?

We could be tempted to limit the value of our `BUFFER_SIZE`, by declaring something like this at the beginning of our function:

```
if (BUFFER_SIZE > 8000000)      // If the buffer's s
        return (NULL);          // Stop the whole th
```

But it is clearly stated in the subject that we should return `NULL` only when there is nothing more to read, or in case of error, which isn't the case here. If we are clever we might decide to simply redefine the `BUFFER_SIZE` when we detect that it is too big with preprocessor commands:

```
# if BUFFER_SIZE > 8000000      // If the size is to
#  undef BUFFER_SIZE            // We want to redefi
#  define BUFFER_SIZE 8000000  // to the maximum si
# endif                         // we can have in th
```

But this won't work very well. Since the limit of the stack varies from one computer to the next, we can't be sure of the exact value we must limit our `BUFFER_SIZE` to. On top of that, there may very well be other data in the stack that would reduce the

available space even further !

Malloc the Buffer!

Therefore, the solution is to allocate memory in the heap with the malloc function. The heap is another area of memory in RAM which doesn't have the same constraints as the stack. Without any real limit, the heap could easily contain our buffer variable, no matter its size. This is in fact the best practice to deal with very large or potentially large variables.

```
//      ...
        char    *buffer;

        buffer = malloc(sizeof * buffer * (BUFFER_SI
        if (!buffer)
                return (NULL);
//      ...
        reader = read(fd, buffer, BUFFER_SIZE);
//      ...
```

No BUFFER_SIZE? No Problem!

So, what happens if we compile without defining a `BUFFER_SIZE` at all? This is a question that isn't asked in the get_next_line subject, but it might be a good idea to take a look at it, especially if we plan to use

this function later on, in other projects.

We could decide to let the compiler grumble in that case. But we could also define a small command to give the preprocessor a default value to use for the `BUFFER_SIZE`. We can say 1, or 42, or 4000. But we could also use the `BUFSIZ` (no final E!) of the `stdio.h` library. Its value varies depending on the compiler (typically 1024 on MacOS, 8129 on Linux), but it is a defined constant:

```
# include <stdio.h>

# ifndef BUFFER_SIZE                // if BUFFER_SIZE wa
#  define BUFFER_SIZE BUFSIZ        // define it with a
# endif
```

The Free Problem

The buffer is probably not the only variable that we will need to malloc in get_next_line. And, as every good 42 student knows, any memory allocated on the heap must be freed.

Chances are that in this project, we have to free in a loop, or without knowing if the variable we want to free was actually malloc'd or not. Navigating a dangerous landscape of double-frees, failed frees, and attempted frees of non-allocated strings is not

easy! We need patience, compilations with –`fsanitize=address`, or even better, with Valgrind on Linux, to be sure we've correctly freed all our mallocs without any memory leaks.

A little trick that could make the hunt for mallocs a little easier is to wrap the `free` function in another function, which can at least help detect if a pointer is already `NULL`, so as not to try to free it a second time:

```
#include <stdlib.h>

// This function takes the address of the string poi
// free. That way we can determine if the pointer wa
// NULL or not. If yes, nothing happens. If not, the
// freed and the pointer set to NULL. This helps avo

void    ft_free_str(char **str)
{
        if (str != NULL && *str != '\0')
        {
                free(*str);
                *str = NULL;
        }
}
```

Bonuses for get_next_line

The first bonus consists of using **only a single static variable**. This is pretty easy to do. The only variable that needs to save its value for the lifetime

of the program is the one in which we store the extra characters read after a `\n`. Other strings can simply be malloc'd and freed as needed, to make their pointers easy to pass from one function to the next.

Then, the second bonus asks us to be able to **read from multiple file descriptors** without getting mixed up. All we need to do here is transform the static variable that saves the extra characters into an array of strings, one for each fd.

Since there is a limit to the number of files an operating system can open simultaneously, we can use that number to define the size of the array. By default, the `fd` limit is 1024. This is a "soft" limit, which means that it is possible to open more files than that at the same time. But 1024 should be more than enough in most cases.

```
1  char    *get_next_line(int fd)
2  {
3  //      static char     *storage;
4          static char     *storage[1024];
5
6  //      We shouldn't forget to add the
   condition if fd > 1024, since we won't be
   able to store anything at storage[1025] or
   at storage[2000].
7          if (fd < 0 || fd > 1024)
8                  return (NULL);
9  //      ...
10 //      Each reference to storage must be
   changed to storage[fd] in the rest of the
```

```
    code.
11          storage[fd] = "...";
12  //      ...
13  }
```

And that's it! Now there is no reason to get anything less than 125% on this project!

Completed Code

My complete get_next_line code is available over here, on GitHub. If you are a 42 student, I encourage you to use the elements above to build your own code in your own way, before glancing at mine.

Sources and Further Reading

- Linux Programmer's Manual:
  - *open(2)* [man]
  - *close(2)* [man]
  - *read(2)* [man]
- B. W. Kernighan, D. M. Ritchie, 1988, *The C Programming Language, Second Edition* :
  - Chapter 8: The UNIX System Interface, p 169.
  - Chapter 4: Functions and Program Structure,

Static Variables, p 83.
- Wikipédia, *Stack overflow* [Wikipédia]
- Stackexchange, *How much stack usage is too much?* [Stackexchange]
- C for Dummies Blog, *How Big is Your BUFSIZ?* [C for Dummies Blog]

42 cursus buffer C file descriptor free malloc read static variable walkthrough

ABOUT THE AUTHOR

Mia Combeau

Student at 42Paris, digital world explorer. I code to the 42 school norm, which means for loops, switches, ternary operators and all kinds of other things are out of reach... for now!

VIEW ALL POSTS

ADD COMMENT

Comment

Name *

Email *

Website

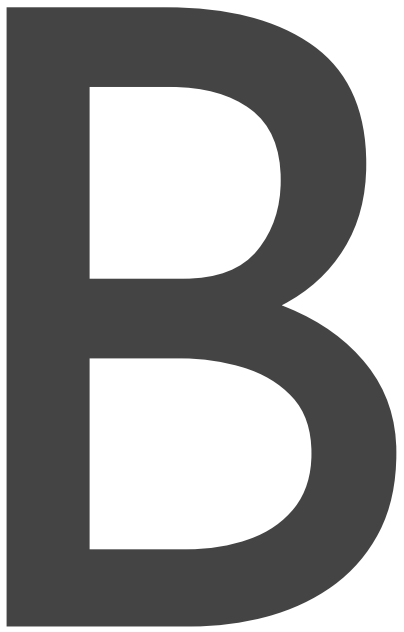
Save my name, email, and website in this browser for the next time I comment.

SUBMIT COMMENT

READ MORE

Binary 010: The Uses of Bit Shifting and Bitwise Operations

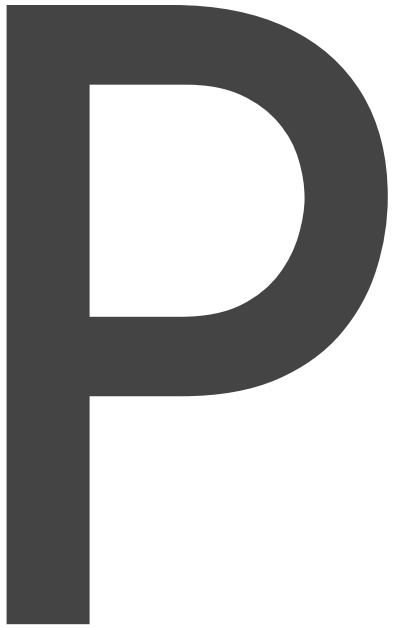
May 7, 2022



Pipex: Reproducing the Pipe Operator

“|” in C

April 2, 2022

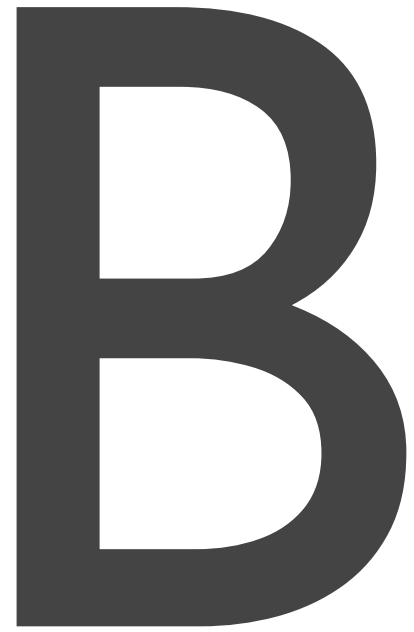


Born2beroot 03: Installing WordPress on a Debian Server

March 13, 2022

Born2beroot 02: Configuring a Debian Virtual Server

March 11, 2022



By Mia Combeau January 22, 2022

To Top

Born2beroot 01: Creating a Debian Virtual Machine

To Top

MENU

- Legal Notice
- Contact

CATEGORIES

- 42 School Projects
- C Programming
- Computer Science

CATEGORIES

- 42 School Projects
- C Programming
- Computer Science

MIA COMBEAU

Student at 42Paris, digital world explorer. I code to the 42 school norm, which means for loops, switches, ternary operators and all kinds of other things are out of reach... for now!

- github

- linkedin