GNL Captain's Log (。◕‿◕。) by mvaldeta

Hello everyone, welcome to my notes!
Since I'm just an average person please do your own fact checking.
If you find something I've written is wrong please send an email/slack me.

A few considerations on this project:

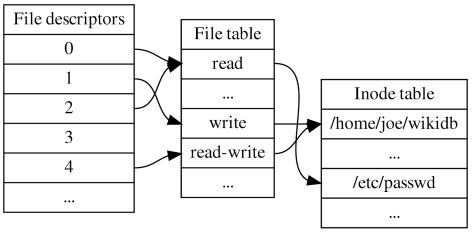
1. Try small tests of reading from files etc.
2. Read about **FD, Static Vars, Heap, Stack, Read & Open.**
3. If you have the same block of code several times, that can be possibly a function.
4. There are several ways to call other functions inside your function and transfer "the command". read below more info.
5. Consider doing one single version that works for both mandatory and bonus versions, see how below.
6. If you get gibberish results, you are leaking somewhere in your function.
7. Write blocks of code/modular code instead of one single function with 100 lines.It's better practice, and easier to debug.
8. Write a debug function/file to help you see your code, and better debug. Or use this small one I've made.(Thanks Ben for helping out)
   https://github.com/m4r11/simple_debug
   I also wrote about finding memory leaks in the last page. Check it out.

BASIC NOTIONS

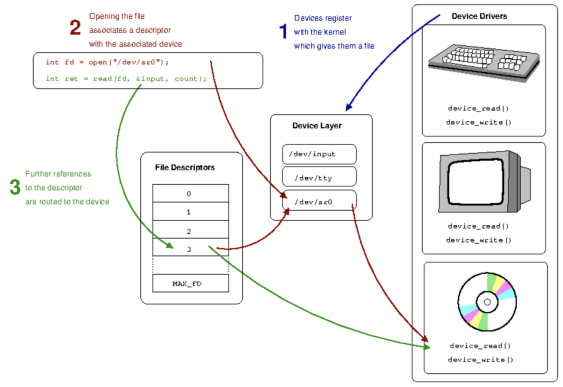
File Descriptor: First things first, what is a file descriptor?
Is it a C concept or a computer science concept? After some research in the www, I came to the conclusion that is a CS concept.
Quoting Wikipedia: "In Unix and related computer operating systems, a **file descriptor** (**FD**, less frequently **fildes**) is an abstract indicator (handle) used to access a file or other input/output resource, such as a pipe or network socket. File descriptors form part of the POSIX application programming interface.
A file descriptor is a non-negative integer, generally represented in the C programming language as the type int (negative values being reserved to indicate "no value" or an error condition)."
Before saying this is too hard to understand, let's go step by step.
Handle: In computer programming, a **handle** is an **abstract reference to a resource** that is used when application software references blocks of memory or objects that are managed by another system like a database or an operating system.
POSIX: Portable Operating System Interface aka, family of standards for maintaining compatibility between operating systems.

So, a file descriptor is a handle that allows us and the computer to speak the same language for an operation. **We meet halfway, at a table.**
It's not only for reading from and to files, but also receives input such as a keyboard, outputs for example our program's executable. **(remember 1 in write()?)**
There are no fixed values for the handles, except for the first three: 0, 1, 2.

Integer value	Name	<unistd.h> symbolic constant[1]	<stdio.h> file stream[2]
0	Standard input	STDIN_FILENO	stdin
1	Standard output	STDOUT_FILENO	stdout
2	Standard error	STDERR_FILENO	stderr



What we are looking at is a File Descriptor table. This table's index increases and decreases as necessary. For example: when opening "myFile.txt" an int 3 will be the forth index in our table. The value returned by an open call is termed a ***file descriptor*** and is essentially an index into an array of open files kept by the kernel.
Check out this website for the source: http://www.bottomupcs.com/file_descriptors.xhtml
I'm adding a chart for visualisation:



The kernel creates a file descriptor in response to an open call and associates the file descriptor with some abstraction of an underlying file-like object, be that an actual hardware device, or a file system or something else entirely. Consequently a process's read or write calls that reference that file descriptor are routed to the correct place by the kernel to ultimately do something useful.

OPEN AND READ

How it works?

Data from a file is read by calling the read function

The read system call takes three arguments:

1. The file descriptor of the file.
2. the buffer where the read data is to be stored and
3. the number of bytes to be read from the file.

this is the prototype: ssize_t read(int *fd*, void **buf*, size_t *nbyte*)

Reading data into a buffer:

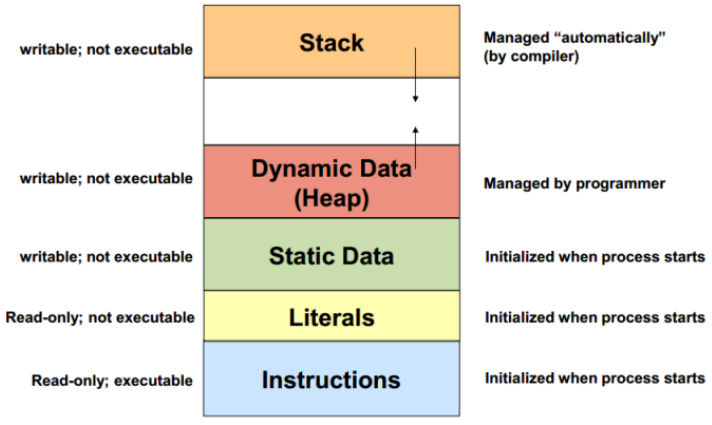
The following example reads data from the file associated with the file descriptor *fd* into the buffer pointed to by *buf*.

```
#include <sys/types.h>
#include <unistd.h>
...
char buf[20];
size_t nbytes;
ssize_t bytes_read;
int fd;
...
nbytes = sizeof(buf);
bytes_read = read(fd, buf, nbytes);
```

MEMORY: Heap vs Stack

Why is this relevant? In GNL we will have to show OFF our memory skills, because this is the first time we'll use 3 different memory "storage" areas.

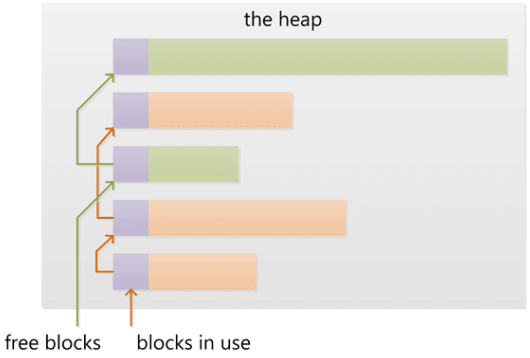
When we declare a variable, it is stored in the stack. We need to know how much of it we need like char *str = "john". When we don't know how much of it we need, we call malloc(3), and dynamically allocate memory line str = malloc(sizeof(str)); The heap is essentially a tree of linked lists.(further below there's representation) It's management has to be done manually therefore free();
In GNL we'll use static variables as well.
Take a look at this visual rep. of memory areas:



Quoting and amazing answer on Stck overflow : https://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap/1213360#1213360

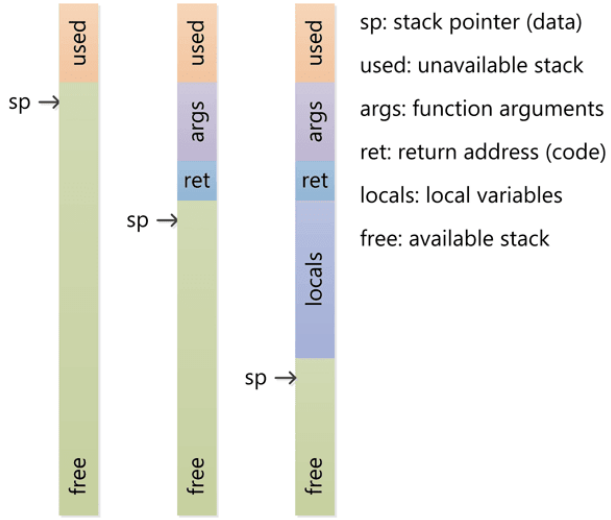
HEAP

- The heap contains a linked list of used and free blocks. New allocations on the heap (by new or malloc) are satisfied by creating a suitable block from one of the free blocks. This requires updating the list of blocks on the heap. This *meta information* about the blocks on the heap is also stored on the heap often in a small area just in front of every block.
- As the heap grows new blocks are often allocated from lower addresses towards higher addresses. Thus you can think of the heap as a *heap* of memory blocks that grows in size as memory is allocated. If the heap is too small for an allocation the size can often be increased by acquiring more memory from the underlying operating system.
- Allocating and deallocating many small blocks may leave the heap in a state where there are a lot of small free blocks interspersed between the used blocks. A request to allocate a large block may fail because none of the free blocks are large enough to satisfy the allocation request even though the combined size of the free blocks may be large enough. This is called *heap fragmentation*.
- When a used block that is adjacent to a free block is deallocated the new free block may be merged with the adjacent free block to create a larger free block effectively reducing the fragmentation of the heap.



THE STACK

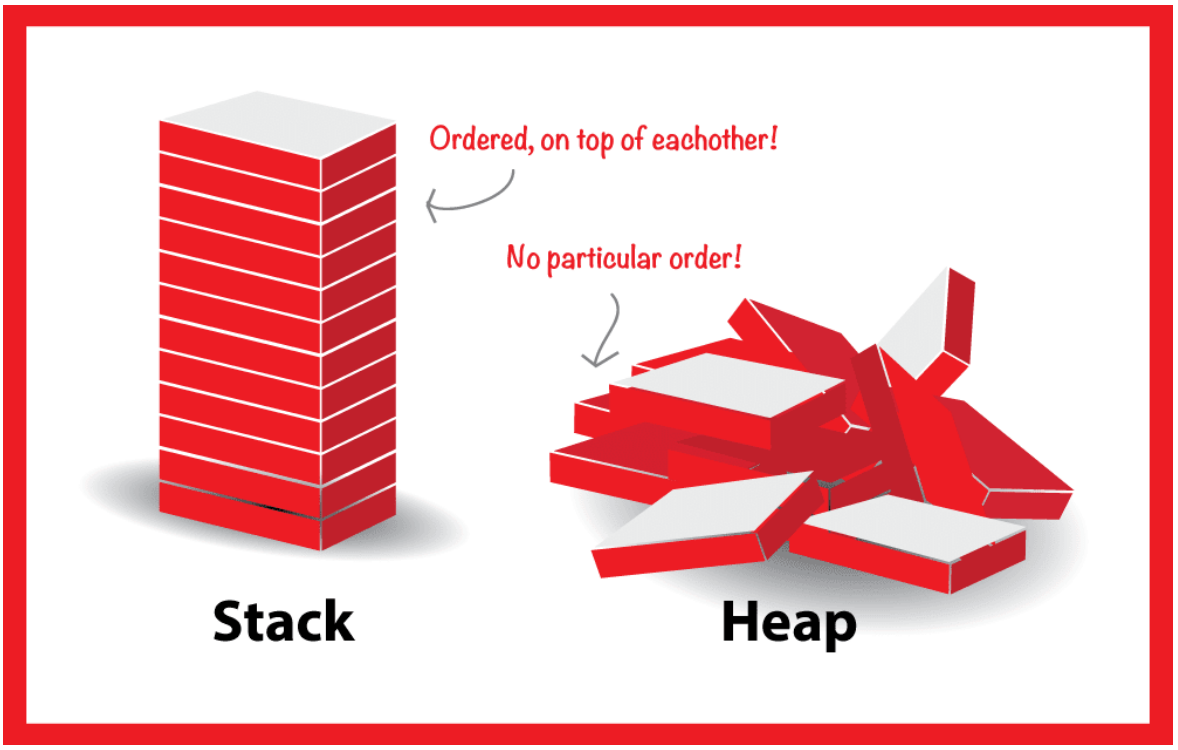
- The stack often works in close tandem with a special register on the CPU named the *stack pointer*. Initially the stack pointer points to the top of the stack (the highest address on the stack).
- The CPU has special instructions for *pushing* values onto the stack and *popping* them back from the stack. Each *push* stores the value at the current location of the stack pointer and decreases the stack pointer. A *pop* retrieves the value pointed to by the stack pointer and then increases the stack pointer (don't be confused by the fact that *adding* a value to the stack *decreases* the stack pointer and *removing* a value *increases* it. Remember that the stack grows to the bottom). The values stored and retrieved are the values of the CPU registers.
- When a function is called the CPU uses special instructions that push the current *instruction pointer*, i.e. the address of the code executing on the stack. The CPU then jumps to the function by setting the instruction pointer to the address of the function called. Later, when the function returns, the old instruction pointer is popped from the stack and execution resumes at the code just after the call to the function.
- When a function is entered, the stack pointer is decreased to allocate more space on the stack for local (automatic) variables. If the function has one local 32 bit variable four bytes are set aside on the stack. When the function returns, the stack pointer is moved back to free the allocated area.
- If a function has parameters, these are pushed onto the stack before the call to the function. The code in the function is then able to navigate up the stack from the current stack pointer to locate these values.
- Nesting function calls work like a charm. Each new call will allocate function parameters, the return address and space for local variables and these *activation records* can be stacked for nested calls and will unwind in the correct way when the functions return.
- As the stack is a limited block of memory, you can cause a *stack overflow* by calling too many nested functions and/or allocating too much space for local variables. Often the memory area used for the stack is set up in such a way that writing below the bottom (the lowest address) of the stack will trigger a trap or exception in the CPU. This exceptional condition can then be caught by the runtime and converted into some kind of stack overflow exception.



Can a function be allocated on the heap instead of a stack?
No, activation records for functions (i.e. local or automatic variables) are allocated on the stack that is used not only to store these variables, but also to keep track of nested function calls.
How the heap is managed is really up to the runtime environment. C uses malloc and C++ uses new, but many other languages have garbage collection.
However, the stack is a more low-level feature closely tied to the processor architecture. Growing the heap when there is not enough space isn't too hard since it can be implemented in the

library call that handles the heap. However, growing the stack is often impossible as the stack overflow only is discovered when it is too late; and shutting down the thread of execution is the only viable option.

Not yet clear? Here is an image for you:



STATIC VARIABLES

Static Variables: In computer programming, a **static variable** is a variable that has been allocated "statically", meaning that its lifetime (or "extent") is the entire run of the program. Holds value between calls.
in https://en.wikipedia.org/wiki/Static_variable

1. A static variable inside a function keeps its value between invocations.
2. A static global variable or a function is "seen" only in the file it's declared in

Example:

```
#include <stdio.h>
void foo()
{
int a = 10;
static int sa = 10;
a += 5;
sa += 5;
printf("a = %d, sa = %d\n", a, sa);
}
int main()
{
int i; for (i = 0; i < 10; ++i) foo();
}
```

This prints:

```
a = 15, sa = 15
a = 15, sa = 20
a = 15, sa = 25
a = 15, sa = 30
a = 15, sa = 35
a = 15, sa = 40
a = 15, sa = 45
a = 15, sa = 50
a = 15, sa = 55
a = 15, sa = 60
```

https://stackoverflow.com/questions/572547/what-does-static-mean-in-c/572550#572550

This is useful for cases where a function needs to keep some state between invocations, and you don't want to use global variables. Beware, however, this feature should be used very sparingly - it makes your code not thread-safe and harder to understand.

Quoting Wikipedia:
In the C programming language, static is used with global variables and functions to set their scope to the containing file. In local variables, static is used to store the variable in the statically allocated memory instead of the automatically allocated memory. While the language does not dictate the implementation of either type of memory, statically allocated memory is

typically reserved in data segment of the program at compile time, while the automatically allocated memory is normally implemented as a transient call stack.

Obviously you have a question at this point, happens when you strdup a static variable? how can one “thing” be at two places at the same time? (meaning the heap and static memory pools?) After a bit of inquiring all the answers I found had to do with disassemblers, in short: the pointer to the data is static and the data is stored in the heap. Hope this relieves you of that question for now.

QUESTIONS :
“Does your function still work if the BUFFER_SIZE value is 9999? And if the BUFFER_SIZE value is 1? And 10000000? Do you know why?”
Because I’m trying to make the function read in a loop until a new line is found, so a buffer of 1 will be in loop until a newline is found.

Yes, but how can you write this function? (or function architecture, is this a thing? I don’t know but let’s pretend it does)

There are clearly some things that just don’t work.

```
while ((r = read(fd, buf, BUFF_SIZE + 1) && r >= EOF_R))
 {
   if (r == -1)
     return (-1);
   *archive = ft_strdup(buf);
   *line = ft_substr(*archive, 0, (ft_find_line_len(*archive, '\n')));
   temp = ft_get_the_rest(buf);
   *archive = ft_strjoin(temp, buf);
   ft_freemem(temp);
   return (1);
 }
```

this for example, reproduces the following:
Reads the first line(BUFFER is set to 100 so, it’s normal) skip’s part of the second line but returns some of it. It’s not an ideal behavior. But it’s something.

r = 1 line = |The Hitch Hiker's Guide to the Galaxy | r = 1 line = |or tea, sympathy, and a sofa| r = 1 line = | western spiral arm of the Galaxy lies a small unregarded| r = 1 line = |ce of roughly ninety-two million miles|	r = 1 line = |The Hitch Hiker's Guide to the Galaxy r = 1 line = | for Jonny Brock and Clare Gorst r = 1 line = |and all other Arlingtoniansfor tea, sympathy, and a sofa
currcurrent behaviour	expected behaviour

if buf size 1 one, crazy things happen:

r = 1 line = || r = 1 line = || r = 1 line = |nd a sofa|	r = 1 line = |The Hitch Hiker's Guide to the Galaxy r = 1 line = | for Jonny Brock and Clare Gorst r = 1 line = |and all other Arlingtoniansfor tea, sympathy, and a sofa
currcurrent behaviour	exp expected behaviour

Conclusion: the while loop is not correct.

```
while ((r = read(fd, buf, BUFF_SIZE + 1) && r >= EOF_R))
```

Yes it's reading until the end of the file, independently of the buffer, but is skipping a lot lines and characters.

If read() it's outside of the while loop, the entire file is read.
We need to read the least amount possible until a line is found.

my second version is as stated:

```
while ((r = read(fd, buf, BUFF_SIZE) && r > EOF_R))
{
  if (r == -1)
    return (-1);
  archive = ft_strjoin(archive, buf);
  int contains_line = ft_find_line_len(archive, '\n');
  // debug_number(contains_line);
  if (contains_line != -1)
  {
    write(1, archive, ft_strlen(archive));

    *line = ft_substr(archive, 0, (contains_line));
    archive = ft_get_the_rest(archive);
  }
}
return (r);
```

it's seems to be working properly,
except the last line is repeated twice.

1THIS IS THE first LINE

1- 2THIS IS THE second LINE
1- 3THIS IS THE third LINE
1- 4THIS IS EOF
0- 4THIS IS EOF

To fix this, consider writing designated lines of code for this edge case. compare the archive len with the buffer len && if there is a newline '\n', make the last line, free and return 0. Combined this should give you the desire result.

Now that finally I have been able to complete the task, (with a lot of help: note to self, consider most cases, if you don't remember one just ask the masters of code)
Let's talk about how to make multiple function calls:

1. **Assign the value of a variable to the result of a function:**
   `*line = ft_substr(archive, 0, (contains_line));`
2. **or in the return after an if statement:**
   ```
   if (!ft_strlen(arch[fd]) && !v.i && !v.r)
   return (ft_delete(&arch[fd]));
   ```
3. **Inside an if statement to break a loop:**
   `if (ft_strchr(arch[fd], '\n')) break;`
4. **As a parameter to another function:**
   `arch[fd] = ft_no_leaks(arch[fd], ft_strdup(arch[fd] + v.i + 1));`

Here is a video about "power transfers" between multiple function calls.
https://www.youtube.com/watch?v=G9QoX51IDzI

This is an example of what you don't want to do. It's hard to debug and has leaks. Instead opt for a modular approach. This version also doesn't work with mandatory and bonus. and fails all testers.

```
void debug_number(int i);
void debug(const char *s, char *name);

int get_next_line(int fd, char **line)
{
 char buf[BUFF_SIZE + 1];
 static char *archive = NULL;
 char *ret = NULL;
 char *tmp;

 ssize_t r = 1;
 int contains_line = ft_find_line_len(archive, '\n');

 if (archive)
 {
   if (contains_line != -1)
   {
     *line = ft_substr(archive, 0, (contains_line));
     archive = ft_get_the_rest(archive, 1);
     return (LINE);
   }
   else
   {
     ret = ft_strdup(archive);
     free(archive);
     archive = NULL;
   }
 }

 while ((r = read(fd, buf, BUFF_SIZE)) && r > 0)
 {
   buf[r] = 0;
   if (r == -1)
     return (-1);
   int contains_line = ft_find_line_len(buf, '\n');

   if (contains_line != -1)
   {
     if (ret)
     {
       tmp = ft_substr(buf, 0, (contains_line));
       *line = ft_strjoin(ret, tmp);
       free(tmp);
       free(ret);
     }
     else
       *line = ft_substr(buf, 0, (contains_line));
     archive = ft_get_the_rest(buf, 0);

     if (r < BUFF_SIZE && !ft_strlen(archive))
       return (0);
     else
       return (LINE);
   }
   else
   {
     if (ret)
     {
       tmp = ret;
       ret = ft_strjoin(ret, buf);
       free(tmp);
     }
     else
       ret = ft_strdup(buf);
   }
 }
 if (ret)
 {
   *line = ft_strdup(ret);
   return (0);
 }
 return (r);
}
```

This is what Moulinette asks us to do when correcting. I found this useful to build my main and test edge cases.

Mandatory Part

Error Management

Carry out AT LEAST the following tests to try to stress the error

management

- Pass an arbitrary file descriptor to the get_next_line function on

which it is not possible to read, for example 42. The function must

return -1.

- Check the error returns for read and malloc.

If there is an error, you must stop the evaluation.

Yes No

Testing

As the evaluator, you are expected to provide a main which will always check:

- The return value of the get_next_line.

- The line read is the line sent to the input, without the \n.

Test all the possible combinations of the following rules:

- Large BUFFER_SIZE (>1024)

- Small BUFFER_SIZE (< 8, and 1)

- BUFFER_SIZE exactly the length of the line to read

- 1 byte variant (+/-) between the line and the BUFFER_SIZE

- Read on stdin

- Read from a file

- (Multiple/Single) Long line (2k+ characters)

- (Multiple/Single) Short line (< 4 characters, even 1)

- (Multiple/Single) Empty line

These tests should allow you to verify the strength of the student's

get_next_line. If any fails, grade 0 on the project.

Yes No

Bonus

We will look at your bonuses if and only if your mandatory part is excellent. This means that you must complete the mandatory part, beginning to end, and your error management must be flawless, even in cases of twisted or bad usage. So if you didn't score all the points on the mandatory part during this defence bonuses will be totally ignored.

Multiple fd reading

Perform the same tests as you did before, this time launch multiple instances of get_next_line with a

different file descriptor on each. Make sure that each get_next_line is returning the correct line,

combine with a non-existing fd to check for errors. Yes No

Single static variable Check the code and verify if there is indeed a single static variable. Give the points if that's the case.

Testers:

https://github.com/Tripouille/gnlTester

How to check for leaks?
There are several options available, I'll name three:

1. Make an infinite loop on your main, run the program: in a new terminal window write 'ps' and enter. The Id of your program or PID will appear. So after just type 'leaks [your pid]' and all the unreachable leaks will appear. this method doesn't check reachable leaks.
2. Clone this repo and follow the instructions: https://github.com/shenyuflying/memwatch don't forget to star him ⭐
3. Are you at the school? In your software manager, instal Virtual Box, and then Unbuntu. In a shared folder open the terminal and run your program and then valgrind. Come meet me in the Cluster and I'll help you.

Good Luck, ping me on slack @mvaldeta 💥
(https://github.com/m4r11/42Cursus/tree/main/lvl1.1_get_next_line)