

NOTES

GET_NEXT_LINE:

- Understanding what needs to be done.

You are now starting to understand that it will get tricky to read data from a file descriptor if you don't know its size beforehand. What size should your buffer be? How many times do you need to read the file descriptor to retrieve the data ?


It is perfectly normal and natural that, as a programmer, you would want to read a "line" that ends with a line break from a file descriptor. For example each command that you type in your shell or each line read from a flat file.

Thanks to the project `get_next_line`, you will finally be able to write a function that will allow you to read a line ending with a newline character from a file descriptor. You'll be able to add this function to your `libft` if you feel like it and most importantly, use it in all the future projects that will require it.

- CORE CONCEPTS COVERED:
 1. Understanding memory allocations: heap vs stack
 2. Static variables
 3. Static functions
 4. Manipulation and life cycle of a buffer
 5. Buffer overflow
 6. Pointers


MEMORY

Stack Allocation : The allocation happens on contiguous blocks of memory. We call it stack memory allocation because the allocation happens in function call stack. The size of memory to be allocated is known to compiler and whenever a function is called, its variables get memory allocated on the stack. And whenever the function call is over, the memory for the variables is deallocated. This all happens using some predefined routines in compiler. Programmer does not have to worry about memory allocation and deallocation of stack variables.



```
int main()
{
    // All these variables get memory
    // allocated on stack
    int a;
    int b[10];
    int n = 20;
    int c[n];
}
```

Heap Allocation : The memory is allocated during execution of instructions written by programmers. Note that the name heap has nothing to do with heap data structure. It is called heap because it is a pile of memory space available to programmers to allocate and de-allocate. If a programmer does not handle this memory well, **memory leak** can happen in the program.



```
int main()
{
    // This memory for 10 integers
    // is allocated on heap.
    int *ptr = new int[10];
}
```

Key Differences Between Stack and Heap Allocations

1. In a stack, the allocation and deallocation is automatically done by whereas, in heap, it needs to be done by the programmer manually.
2. Handling of Heap frame is costlier than handling of stack frame.
3. Memory shortage problem is more likely to happen in stack whereas the main issue in heap memory is fragmentation.
4. Stack frame access is easier than the heap frame as the stack have small region of memory and is cache friendly, but in case of heap frames which are dispersed throughout the memory so it cause more cache misses.
5. Stack is not flexible, the memory size allotted cannot be changed whereas a heap is flexible, and the allotted memory can be altered.
6. Accessing time of heap takes is more than a stack.

Comparison Chart:

PARAMETER	STACK	HEAP
Basic	Memory is allocated in a contiguous block.	Memory is allocated in any random order.
Allocation and Deallocation	Automatic by compiler instructions.	Manual by programmer.
Cost	Less	More
Implementation	Hard	Easy
Access time	Faster	Slower
Main Issue	Shortage of memory	Memory fragmentation
Locality of reference	Excellent	Adequate
Flexibility	Fixed size	Resizing is possible


STATIC VARIABLES

Static Variables in C

Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve their previous value in their previous scope and are not initialized again in the new scope.

Syntax:

```
static data_type var_name = var_value;
```



```
#include<stdio.h>
int fun()
{
    static int count = 0;
    count++;
    return count;
}

int main()
{
    printf("%d ", fun());
    printf("%d ", fun());
    return 0;
}
```

Output:


```
1 2
```

2) Static variables are allocated memory in data segment, not stack segment. See [memory layout of C programs](#) for details.

3) Static variables (like global variables) are initialized as 0 if not initialized explicitly. For example in the below program, value of x is printed as 0, while value of y is something garbage. See [this](#) for more details.

STATIC FUNCTIONS


In C, functions are global by default. The *"static"* keyword before a function name makes it static. For example, below function *fun()* is static.



```
static int fun(void)
{
    printf("I am a static function ");
}
```


Unlike global functions in C, access to static functions is restricted to the file where they are declared. Therefore, when we want to restrict access to functions, we make them static. Another reason for making functions static can be reuse of the same function name in other files.

For example, if we store following program in one file *file1.c*



```
/* Inside file1.c */
static void fun1(void)
{
    puts("fun1 called");
}
```

And store following program in another file *file2.c*



```
/* Inside file2.c */
int main(void)
{
    fun1();
    getchar();
    return 0;
}
```

Now, if we compile the above code with command *"gcc file2.c file1.c"*, we get the error *"undefined reference to 'fun1'"*. This is because *fun1()* is declared *static* in *file1.c* and cannot be used in *file2.c*.

Please write comments if you find anything incorrect in the above article, or want to share more information about static functions in C.

BUFFERS

What is a buffer?

A temporary storage area is called buffer. All standard input and output devices contain an input and output buffer. In standard C/C++, streams are buffered, for example in the case of standard input, when we press the key on keyboard, it isn't sent to your program, rather it is buffered by operating system till the time is allotted to that program.

How does it effect Programming?

On various occasions you may need to clear the unwanted buffer so as to get the next input in the desired container and not in the buffer of previous variable. For example, in case of C after encountering "scanf()", if we need to input a character array or character, and in case of C++, after encountering "cin" statement, we require to input a character array or a string, we require to clear the input buffer or else the desired input is occupied by buffer of previous variable, not by the desired container. On pressing "Enter" (carriage return) on output screen after the first input, as the buffer of previous variable was the space for new container (as we didn't clear it), the program skips the following input of container.

BUFFER OVERFLOW

<https://www.youtube.com/watch?v=CQ6pGrXY1Us>

2D ARRAY

```
int x[3][4] = {{0,1,2,3}, {4,5,6,7}, {8,9,10,11}};
```

This type of initialization make use of nested braces. Each set of inner braces represents one row. In the above example there are total three rows so there are three sets of inner braces.

Accessing Elements of Two-Dimensional Arrays: Elements in Two-Dimensional arrays are accessed using the row indexes and column indexes.

POINTERS

& - to access the address of a variable to a pointer.

* - used to declare pointer

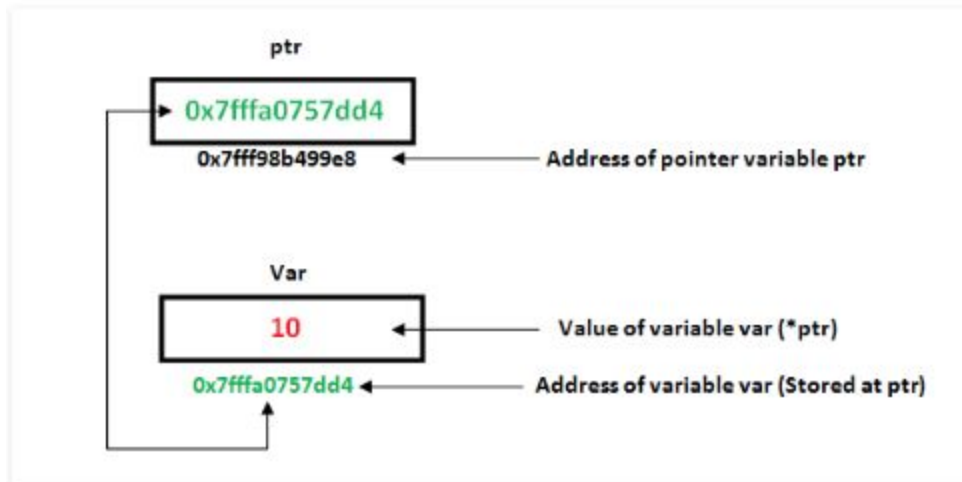
- To access the value stored in the address

Example

Int `var` = 10;

Int *`ptr` = &`var` (Holds the address of the variable `var`)

Below is pictorial representation of above program:



POINTER ARITHMETIC

A pointer may be:

- incremented (++)
- decremented (--)
- an integer may be added to a pointer (+ or +=)
- an integer may be subtracted from a pointer (- or -=)

Pointer arithmetic is meaningless unless performed on an array.

Note : Pointers contain addresses. Adding two addresses makes no sense, because there is no idea what it would point to. Subtracting two addresses lets you compute the offset between these two addresses.

```

// C++ program to illustrate Pointer Arithmetic
// in C/C++
#include <bits/stdc++.h>

// Driver program
int main()
{
    // Declare an array
    int v[3] = {10, 100, 200};

    // Declare pointer variable
    int *ptr;

    // Assign the address of v[0] to ptr
    ptr = v;

    for (int i = 0; i < 3; i++)
    {
        printf("Value of *ptr = %d\n", *ptr);
        printf("Value of ptr = %p\n\n", ptr);

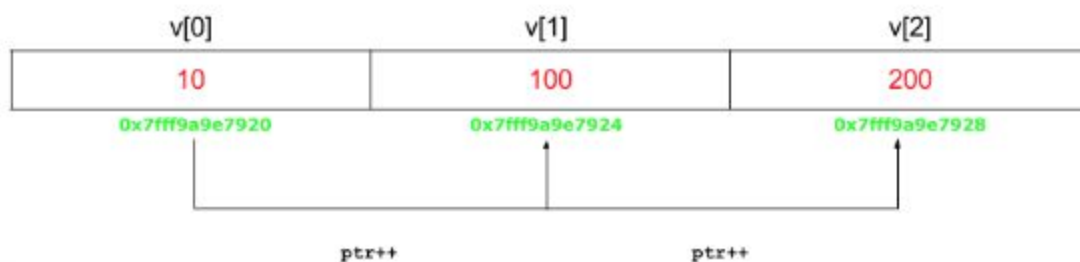
        // Increment pointer ptr by 1
        ptr++;
    }
}

```

Output: Value of *ptr = 10
Value of ptr = 0x7ffcae30c710

Value of *ptr = 100
Value of ptr = 0x7ffcae30c714

Value of *ptr = 200
Value of ptr = 0x7ffcae30c718



DOUBLE POINTERS

We already know that a pointer points to a location in memory and thus used to store address of variables. So, when we define a pointer to pointer. The first pointer is used to store the address of second pointer. That is why they are also known as double pointers.

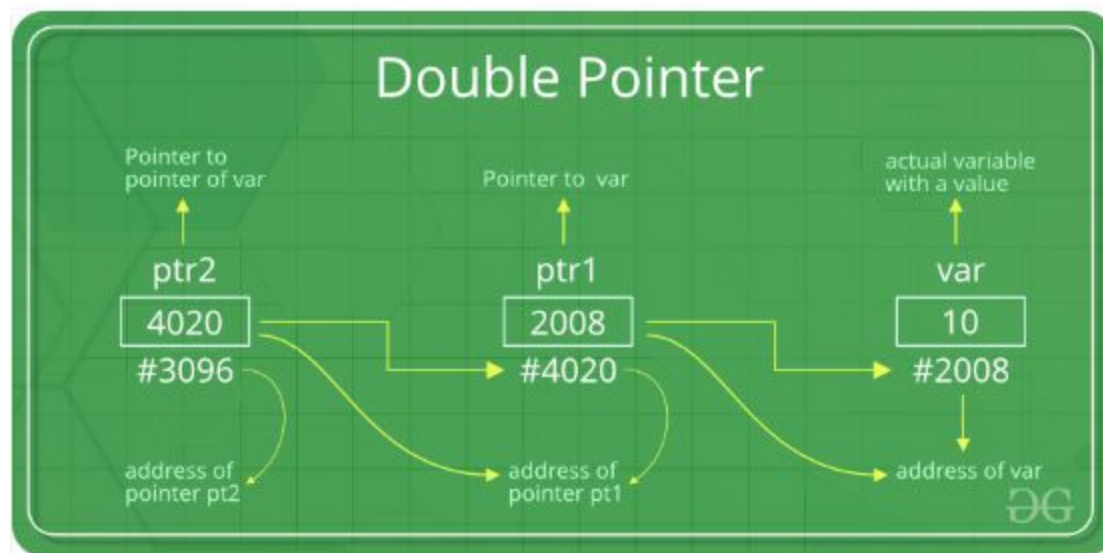
How to declare a pointer to pointer in C?

Declaring Pointer to Pointer is similar to declaring pointer in C. The difference is we have to place an additional '*' before the name of pointer.

Syntax:

```
int **ptr;    // declaring double pointers
```

Below diagram explains the concept of Double Pointers:



The above diagram shows the memory representation of a pointer to pointer. The first pointer ptr1 stores the address of the second pointer ptr2 and the second pointer ptr2 stores the address of the variable.

FD

How does file descriptor work?



File descriptors are an index into a **file descriptor** table stored by the kernel. 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.

Why is fd =3 by default when using open?

Nothing: there are three standard file descriptions, STDIN, STDOUT, and STDERR. They are assigned to 0, 1, and 2 respectively.

What you are seeing there is an artifact of the way `ls(1)` works: in order to read the content of the `/proc/self/fd` directory and display it, it needs to open that directory.

That means it gets a file handle, typically the first available ... thus, 3.

If you were to run, say, `cat` on a separate console and inspect `/proc/${pid}/fd` for it you would find that only the first three were assigned.

In the `ls` example, I would imagine descriptor 3 is the one being used to read the filesystem. Some C commands (eg, `open()`), which underpin the generation of file descriptors, guarantee the return of *"the lowest numbered unused file descriptor"* (POSIX -- note that low level `open()` is actually not part of Standard C). So they are recycled after being closed (if you open and close different files repeatedly, you will get 3 as an fd over and over again).

For more info on file descriptors..

<https://www.geeksforgeeks.org/input-output-system-calls-c-create-open-close-read-write/>