Minishell: Building a mini-bash (a @42 project)

MannBell · Follow
11 min read · Jul 29, 2023

Listen Share

This project was one of the most challenging yet indulging projects of the 42 curriculum that I have faced so far, when you have to deal with many test cases and when you get to learn many things, from software architecture, system calls, file descriptors… to team coordination, management, and work distribution.

Building a shell that mimics the bash was an intimidating thing for me at first, because the only thing that I knew back then about shells was how to use them, in a basic way to accomplish basic tasks, but I was not really aware of how they internally work or function, and building a shell from scratch was really a challenging task for me, though this is one of the main things that I do really like about the 42 pedagogy, is that I always keep finding myself at a challenge, always trying to learn something new, from the early days of the Piscine (42 selection phase) up to writing this article.

Minishell is somewhat of a big project –especially for a beginner–, and a complex one too, and starting it definitely needed some planning and research, to avoid overhauling the whole design later — even though I ended up doing something similar.

Since this is a team project I had to form a team of two members, and that what I did. I went searching for a team member, formed a team, and started coordinating tasks.

The shell basically has two parts, the parsing (where you treat user input), and the execution (where you execute what have been parsed).

I took the parsing part, while the execution part was taken by the other team member, but I ended up doing almost both after finding out that the execution part was poorly done, with many bad practices (needless global variable use, needless complication and bad code management), the code was not fully functional, with many bugs that were hard to trace (because of bad practices).

But the main reason that pushed me to redo most of the execution part is that the other team member refused to fix their own code and instead they just threw their responsibility of fixing it over me, which led to breaking up the team later — consequently going through the hassle of finding a new team member again.

I paused for a minute, and I though thoroughly about the situation… I haven't got much time left, if I make no progress, nobody is going to do it for me, so I had no choice but to tackle the execution part alone, yes it was hard and too much stressing to do almost everything, but it was rewarding later, and I am very grateful that I went over the execution part because else I wouldn't have learned about many things and details.

Anyway, let's have a project overview…

We are required to build a mini shell (command-line interpreter) that mimics the bash, hence the name it wouldn't be doing all the work that bash does, but the basic functionality:

- The shell will work only in interactive mode (no scripts, i.e. the executable takes no arguments)
- Run simple commands with absolute, relative path (e.g. `/bin/ls`, `../bin/ls`)
- Run simple commands without a path (e.g. `ls`, `cat`, `grep`, etc…)
- Have a working history (you can navigate through commands with up/down arrows)
- Implement pipes (`|`)
- Implement redirections (`<`, `>`, `>>`)

- Implement the here-doc (`<<`)
- Handle double quotes (`""`) and single quotes (`''`), which should escape special characters, beside `$` for double quotes.
- Handle environment variables (`$` followed by a sequence of characters).
- Handle signals like in bash (`ctrl + C` , `ctrl + \` , `ctrl + D`).
- Implement the following built-ins:
  — echo (option `-n` only)
  — exit
  — env (with no options or arguments)
  — export (with no options)
  — unset (with no options)
  — cd
  — pwd
- **And for the bonus part (optional, but i did it, because it's cool!)**
  — handle `&&` and `||` with the parenthesis `()` for priority.
  — handle `*` wildcards for the current working directory.

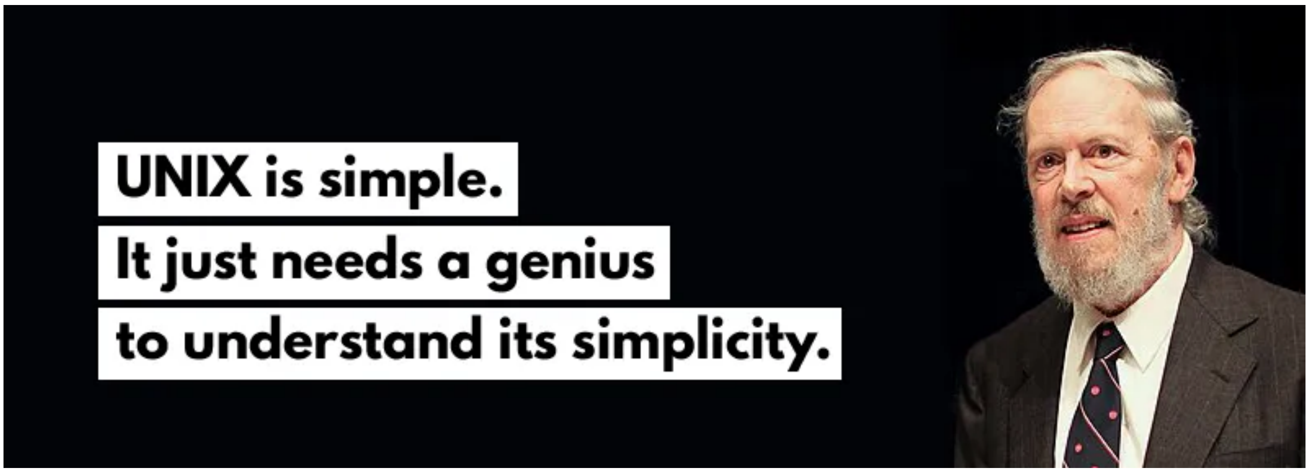
OMG! Where to start?! — That was my reaction when I did read the assignment for the first time.

After doing my research on bash, and how it is implemented, I found out that its implementation is split into two basic parts; we can call them the front-end and the back-end.

The front-end is the part that deals with user input and user interaction, like commands and signals, while the back-end is where the internal work is done (the execution).

We have mainly two things to take care of in the front-end, a command (user input as a line/string) or a signal (`ctrl + C` , etc...). I chose to postpone signals until I finish the execution, as I found implementing signals from the start will just add unnecessary complication.
That means that the first thing that I started with was treating user input.

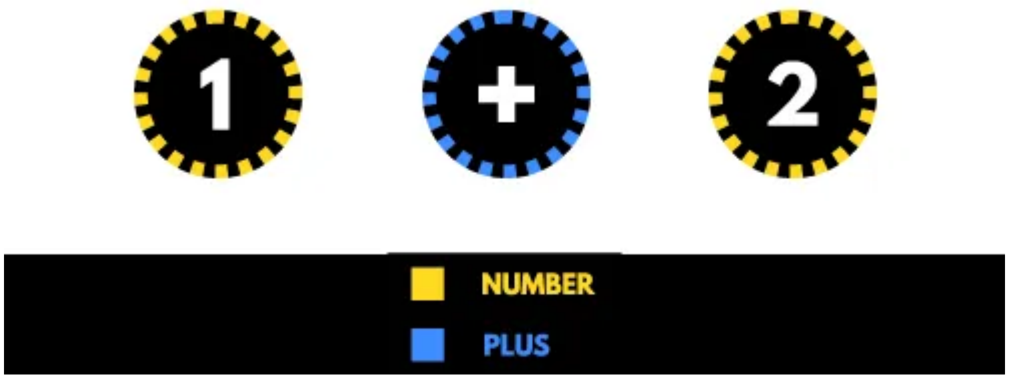
The first time that I thought about user-input and how the shell treats it I realized that there must be a systematic/algorithmic way to treat those lines, though many of my peers/colleagues went to “hard-code” parsing and naively treating cases. Not going to lie, that was tempting for me too at first, but I realized that I should get out of my comfort zone, and learn something new.



Quote by Dennis Ritchie

At first it was a really pain in the head to get myself into parsing, I didn’t know how or where to start, and at the same time I didn’t want to start ignorant with a naive approach, and then need to rework the entire thing, that’s why I looked up the way bash parses commands, and it turned out that parsing a command goes through two phases, the lexical analysis (lexing) which produces “lexems” and then the syntax analysis (parsing the lexems).

Lexical analysis / tokenization: taking the input from the user and processing it char by char into “tokens”.
For example, let’s say we are building a calculator program and we have `1 + 2` as input, we can convert this input to a stream of tokens similar to the following:



A graphical representation of tokens

As you can see, the tokens have a type and value, for example, the first token from the left is of type `NUMBER` and has a value of `1` .
So why do we that? It is to have some kind of abstraction, because while we are parsing using a specific grammar, the most important thing to us is the type of the lexems/tokens — not the values– and the order they came in.

Syntax analysis / Parsing: scanning the stream of tokens according to a specific grammar and then deciding what to do with them (e.g. generating an AST — Abstract Syntax Tree).

Lexing was kind of straightforward for me, as you just split the input into tokens as you please and see useful for the parsing phase.

Parsing though was challenging, and I realized that I needed guidance, so I looked up the way bash does that and I found that it does so by using a combination of techniques and mainly The Recursive Descent, I watched a tutorial about it, and it said that you must first learn The Theory of Computation, but after looking up what this theory of computation is, I realized that there is no way I am going to learn all that in the sheer amount of time that I got, so I went into the practical mode, what does that mean? It means that I watch the tutorial nevertheless, and whenever I find something that I don't understand, I go and I learn it, rather than going through the whole theory of computation which may contain some things that I won't really need for this project — though, I found the theory of computation very interesting.

Before I got a bit familiar with the Recursive Descent, I had to understand first

what a grammar is and what a language is, etc…
So what is a grammar?

There are several kinds of grammars, but the type of grammar that is used for this project is called CFG (Context Free Grammar).

A grammar is basically a way of telling how a language can be made, back to the example above… We can create a grammar that tells us how an arithmetic expression can be formed, for example:

```
//The following grammar is written in Backus-nauer form

<Expression>  :=  <Number>
              |   <Number> ‘+’ <Expression>

<Number>      :=  One or more Digits
```

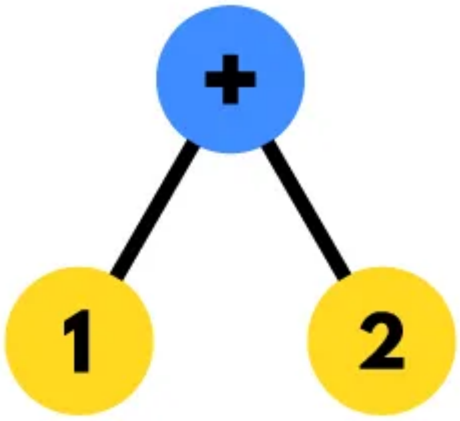
We are saying in this grammar, that an Expression can be a Number or a Number followed by a `+` sign that is followed by another Expression, and a Number can be one or more digits.

> *> Note: not all grammars work with the Recursive Descent, the chosen grammar shall not have ambiguity or left-recursion.*

It took me about two weeks in total to learn the basic use of this Recursive Descent, still I ended up not using it, because my main goal was how to build a simple easy to use AST, but this technique may not give an AST directly, as you will have to build a “parse tree” first and then convert it into an AST, I tried to do that, but I got stuck with the grammar and the way I would convert that “parse tree” into an AST, and also the operator precedence was an ambiguous thing for me back then, and things got over complicated quickly, that’s when I realized that I am in need of a more efficient yet simple technique to switch to…
After doing some research I found another technique that helped me to get my parser together, and it is like a newer improved version of the Recursive Descent, it is called Precedence Climbing , which you can read more about in **this article** that helped me a lot in understanding it.

Before I chose the Precedence Climbing technique, I found another technique called the shunting-yard algorithm which deals with operator precedence too, but I preferred the Precedence Climbing technique as I found it simpler and more compact.

By using the Precedence Climbing parsing on the stream of tokens from the example above we could end up with an AST that looks like the following:



Graphical representation of an AST

> *> I try to keep things as simple as possible, that's why I am using a simple operation instead of a complex command line, as the process is almost identical in both.*

After building the AST, we initialize it. This part is a complex one as it contains many tasks:

The expansion of environment variables:

I don't know the exact way bash expands environment variables, but I came up with my own way, and it is composed of few steps illustrated with an example.

Let's set a variable `A` with the value `1`.

```
export A="1"
```

Let's suppose we get the following user input:

```
""$A B "CD"
```

The first thing with do, is to replace `$A` with its value, so it becomes:

`""1 B "CD"`

Then we get rid of empty strings, so we get:

`1 B "CD"`

Then we split the string by spaces (depending on the quotes), so we get three strings:

`1` , `B` , `"CD"`

Then we check if there is any globbing (`*` wildcard), if there is we replace the wildcard by the appropriate strings — the globbing algorithm wasn't mine, I looked it up on the internet, it's a backtracking algorithm, you can learn more about it here.

Finally, we strip out any quotes:

`1` , `B` , `CD`

> *> I left quotes intentionally all the way to the end, because they are useful in the globbing part.*

Here-doc initialization:

Here-doc initialization is where here-doc gets initialized by storing the input in a pipe, which will be used later by the execution — ***though we have to make sure to close the corresponding file descriptors, else the program may freeze somewhere.***

After finishing initializing the AST, comes the time for the execution.

The execution.

What we are basically given in the execution part is an AST, and the way we will execute it is by traversing it recursively, here is a pseudo-code that traverses the arithmetic AST built earlier in the parsing phase.

```
int eval(t_node node)
{
 if (node.type == PLUS)
   return (node.left + node.right);
 else
   return (node.value)
}
```

In the case of a shell, we would have something as `|`, `||` or `&&` instead of `+`.
The pseudo-code would look something similar to:

```
int execute(t_node node)
{
 if (node.type == PIPE)
   return (execute_pipe(node.left, node.right));
 else
   return (execute_simple_command(node.value))
}
```

I learned many things in the execution part, most importantly, is how to manage file descriptors and forking processes.
For example, in the function `execute_pipe` we would have something similar to:

```
int execute_pipe()
{
   pipe(pipe_fds);
   left_pid = fork();
   if (pid == 0) // inside the left child
   {
      // do stuff
      close(pipe_fds[0]); //Accordingly
      close(pipe_fds[1]); //Accordingly
   }
   else // back to the parent
   {
      //do the same for the right child
      // Make sure to close the pipe_fds in the parent too
   }
}
```

That's about the core of the execution part, beside some work on redirections, where we basically open a file and then redirect (using dup2) the standard input or the standard output to that file, and in case of the here-doc we redirect the standard input to a pipe.

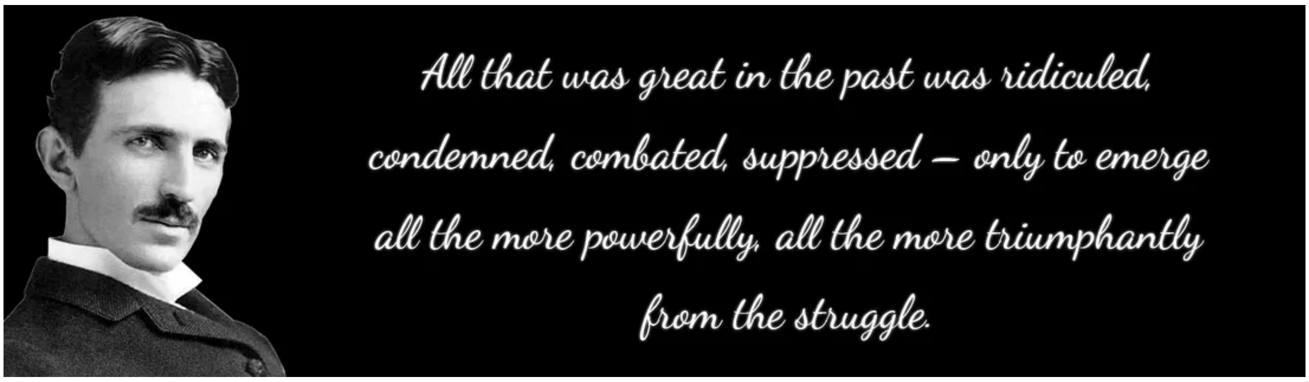
The built-ins are a set of useful functions that are needed in the Minishell, they differ in complexity, from the easy ones like `echo` to complex ones, the built-ins were taken care of by my final teammate, to whom i take the chance to acknowledge their effort and willingness to work.

After finishing the execution, we can implement the signals (`ctrl + C`, `ctrl + \`, `ctrl + D`).
For `ctrl + C`, `ctrl + \`, we can catch `SIGINT` and `SIGQUIT` respectively, but in the case of `ctrl + D` we have to check the `EOF` or the end of string (i.e. `NULL` character).

That was it, I wanted to keep this article as simple as possible by not putting too many details and intricacy into it, the code is written in C, you can always check it at my GitHub account: m4nnb3ll

I wish that this article would be helpful to at least somebody… I wanted to share my experiences and the difficulties that I went through and that it won't always work as you would expect, and difficulties come along the way, it is normal, it is the way it is, if you find yourself in a hardship, don't just give up, but instead start looking at what you have, what you can do, and the possible solutions… if you can't get along with somebody, it's okay, move on, look for another partner who you would get along with and so on…

Finally, I would like to finish with a quote of the brilliant Serbian Physicist and Inventor **Nikola Tesla** that I do really like:


All that was great in the past was ridiculed, condemned, combated, suppressed — only to emerge all the more powerfully, all the more triumphantly from the struggle.

Quote by Nikola Tesla

Unix Linux Shell Bash Minishell

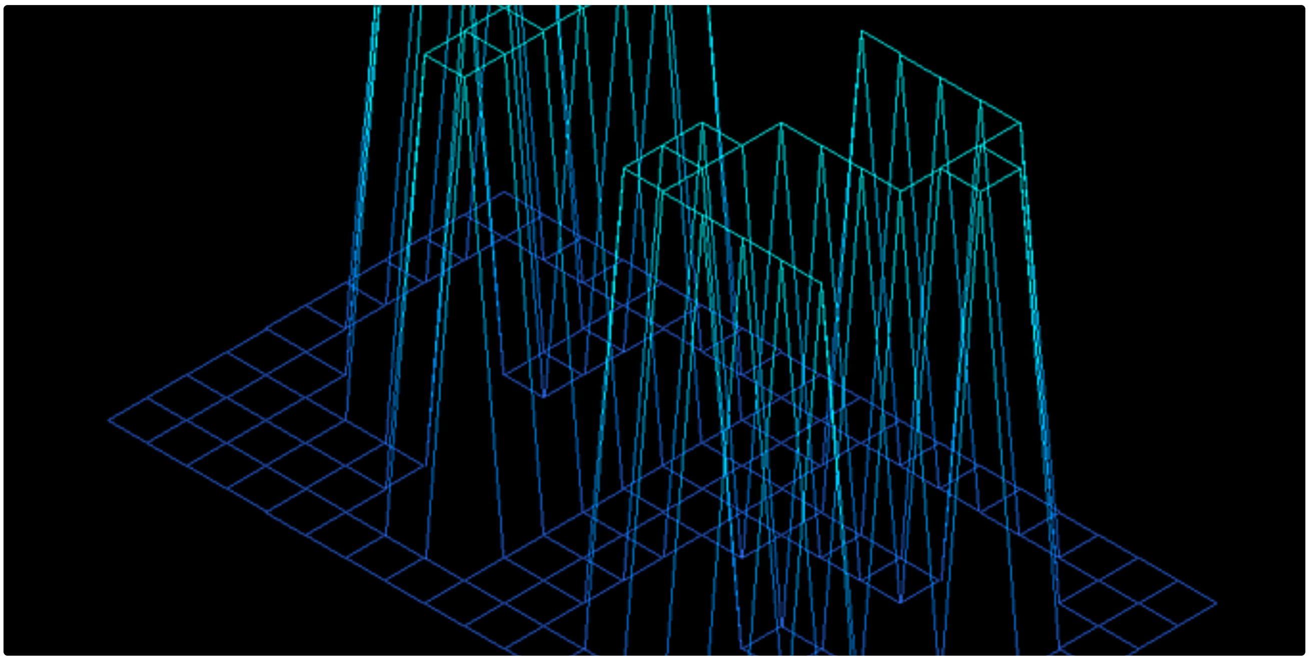
Follow

Written by MannBell

49 Followers

Student at 1337 Med (42 Network)

More from MannBell



MannBell

Fil de Fer(Fdf), the first graphical project at 42 The Network

The name of the porject is in French, which literally means in English “Iron Wire”, as the name suggests the project is about “Wireframe...

5 min read · Jun 25, 2023

6

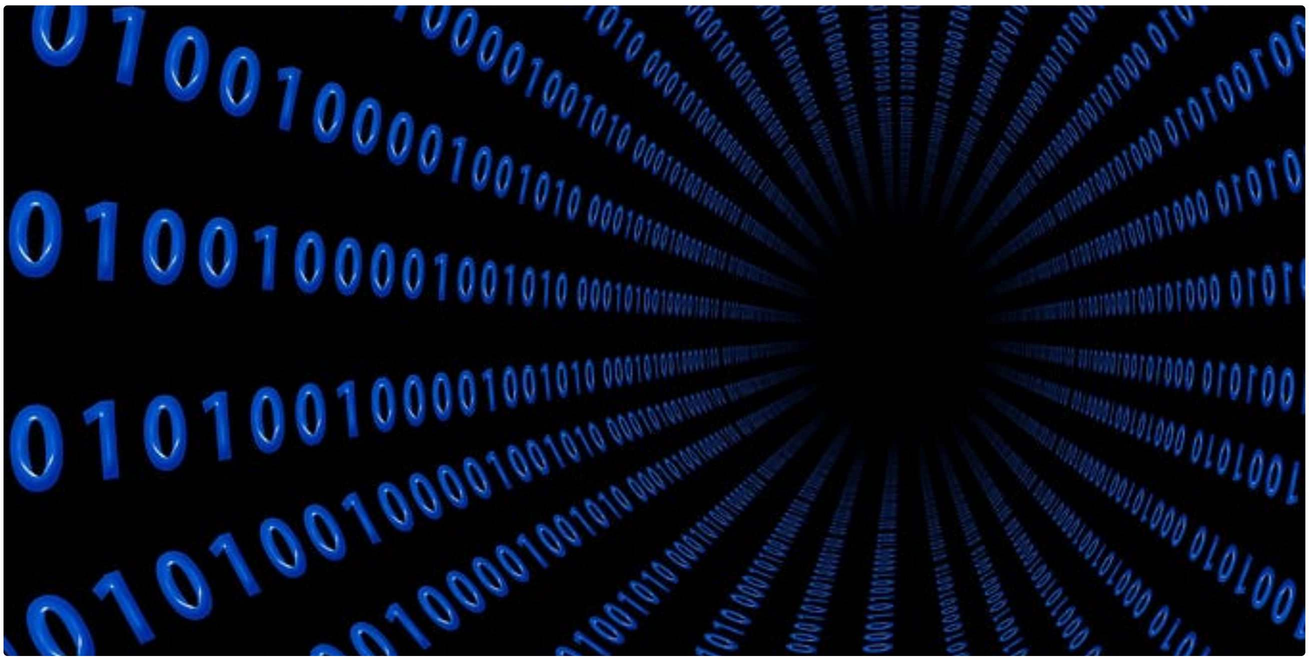
MannBell

webserv: Building a Non-Blocking Web Server in C++98 (A 42 project)

In the vast realm of internet protocols, the Hypertext Transfer Protocol (HTTP) stands as the backbone of communication for the World Wide...

5 min read · Jan 14, 2024

40



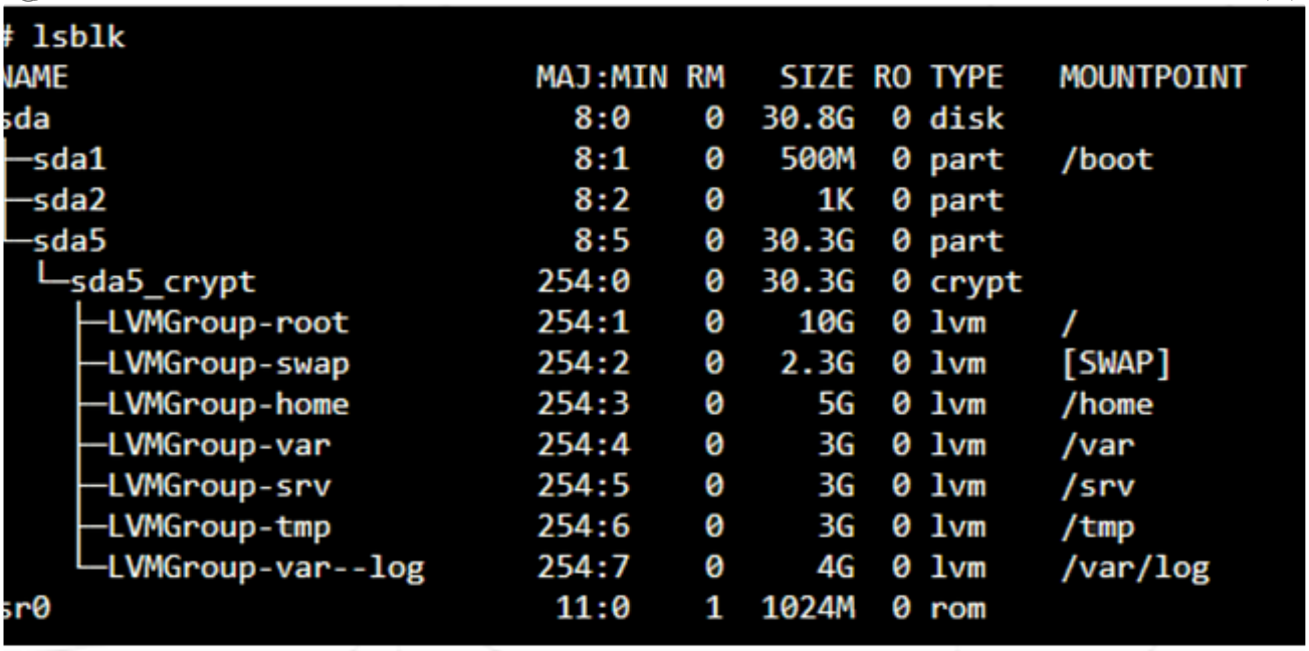
MannBell

First time to work with signals, Minitalk(a 42 The Network project)

Before i started working with signals, i had already encountered them but without realizing what they really were, for example when a...

4 min read · Jun 23, 2023

1



```
# lsblk
NAME                      MAJ:MIN RM   SIZE RO TYPE   MOUNTPOINT
sda                         8:0    0  30.8G  0 disk
├─sda1                      8:1    0   500M  0 part   /boot
├─sda2                      8:2    0     1K  0 part
└─sda5                      8:5    0  30.3G  0 part
  └─sda5_crypt            254:0    0  30.3G  0 crypt
    ├─LVMGroup-root       254:1    0    10G  0 lvm    /
    ├─LVMGroup-swap       254:2    0   2.3G  0 lvm    [SWAP]
    ├─LVMGroup-home       254:3    0     5G  0 lvm    /home
    ├─LVMGroup-var        254:4    0     3G  0 lvm    /var
    ├─LVMGroup-srv        254:5    0     3G  0 lvm    /srv
    ├─LVMGroup-tmp        254:6    0     3G  0 lvm    /tmp
    └─LVMGroup-var--log   254:7    0     4G  0 lvm    /var/log
sr0                        11:0    1  1024M  0 rom
```

MannBell

Setting up a WordPress server on Debian (Born2beroot bonus) 42 The Network

I- Setting up partitions using LVM

6 min read · Jan 29, 2023

60

See all from MannBell

Recommended from Medium

MannBell

webserv: Building a Non-Blocking Web Server in C++98 (A 42 project)

In the vast realm of internet protocols, the Hypertext Transfer Protocol (HTTP) stands as the backbone of communication for the World Wide...

5 min read · Jan 14, 2024

40

Deepak Ranolia

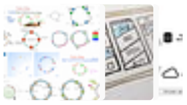
C++ Cheatsheet: Expert Level

Welcome to the C++ Cheatsheet designed to enhance your proficiency in C++ programming. This is structured into four levels: Basic...

7 min read · Dec 5, 2023

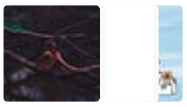
116

Lists



General Coding Knowledge

20 stories · 1072 saves



Staff Picks

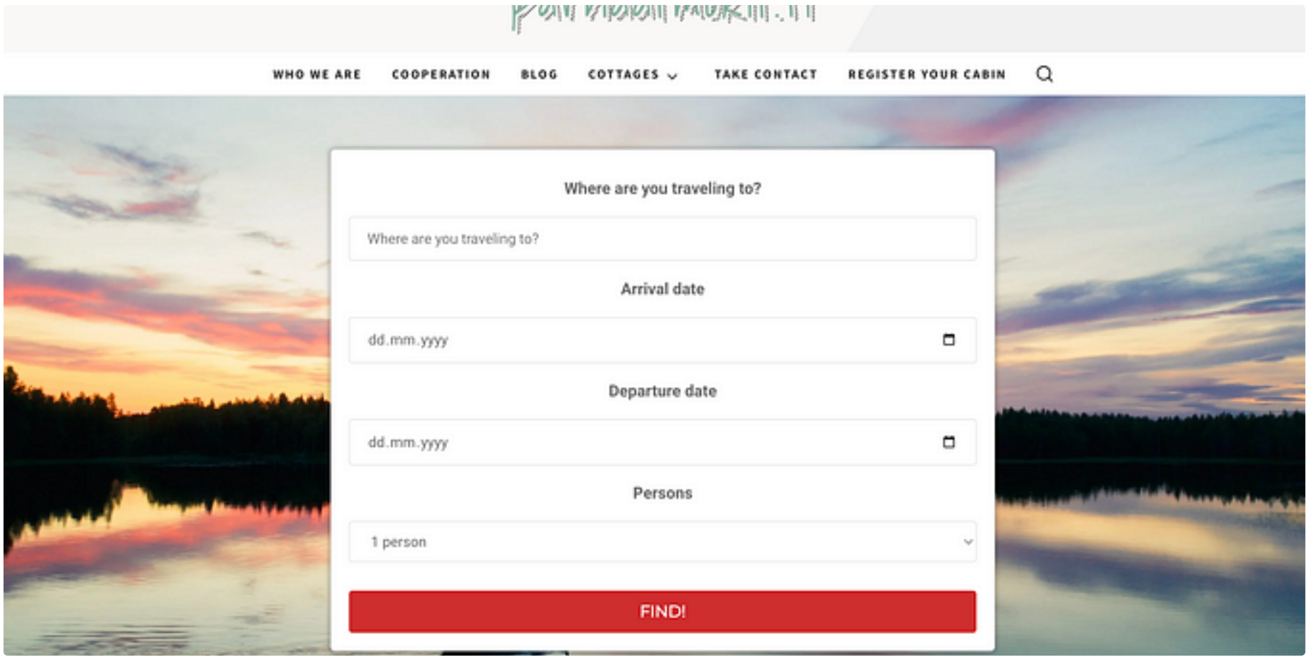
609 stories · 873 saves

AHMED EZ-ZOUINE

1337 | ft_get_next_line | File I/O

The goal of this project is to learn about static variables and improving your memory management skills (allocating and freeing)

5 min read · Nov 28, 2023



Artturi Jalli

I Built an App in 6 Hours that Makes $1,500/Mo

Copy my strategy!

· 3 min read · Jan 23, 2024

15.6K 181

Agnathavasi

ICC2 —Usefull commands

1. Get the count of Clock buffers? report_device_group -detailed clock_network

4 min read · Nov 20, 2023

7

ANC

Creating a Simple Custom List in C++

#include <iostream> class CustomList { struct Node { Node *next; int data; Node(int item): data(item)...

2 min read · Nov 5, 2023

2 1

See more recommendations