Preface

Phrase #1

In Rust, if your program compiles, it probably works."

Phrase #2

You can't segfault if you don't have null.

Phrase #3

Rust doesn't hide complexity from developers it offers them the right tools to manage all the complexity.

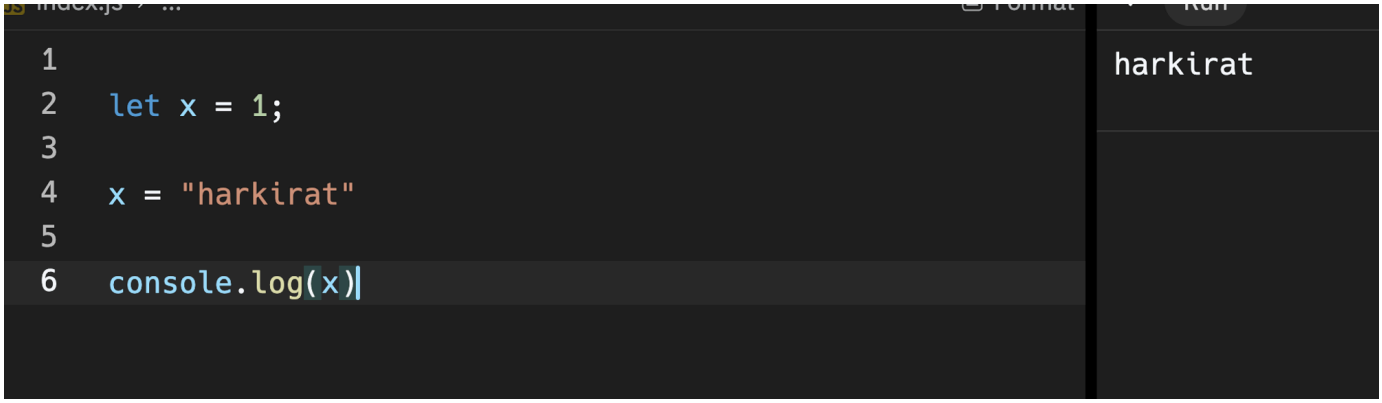
Syllabus


Easy
1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables
(number, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Optional/Result
10. Pattern matching
11. Package management
Hard
This video
1. Memory management
2. Mutability
3. Stack vs heap
4. Ownership
5. Borrowing
6. References
Next video
1. Traits
2. Generics
3. Lifetimes
4. Multithreading
5. Macros
6. Futures/async await

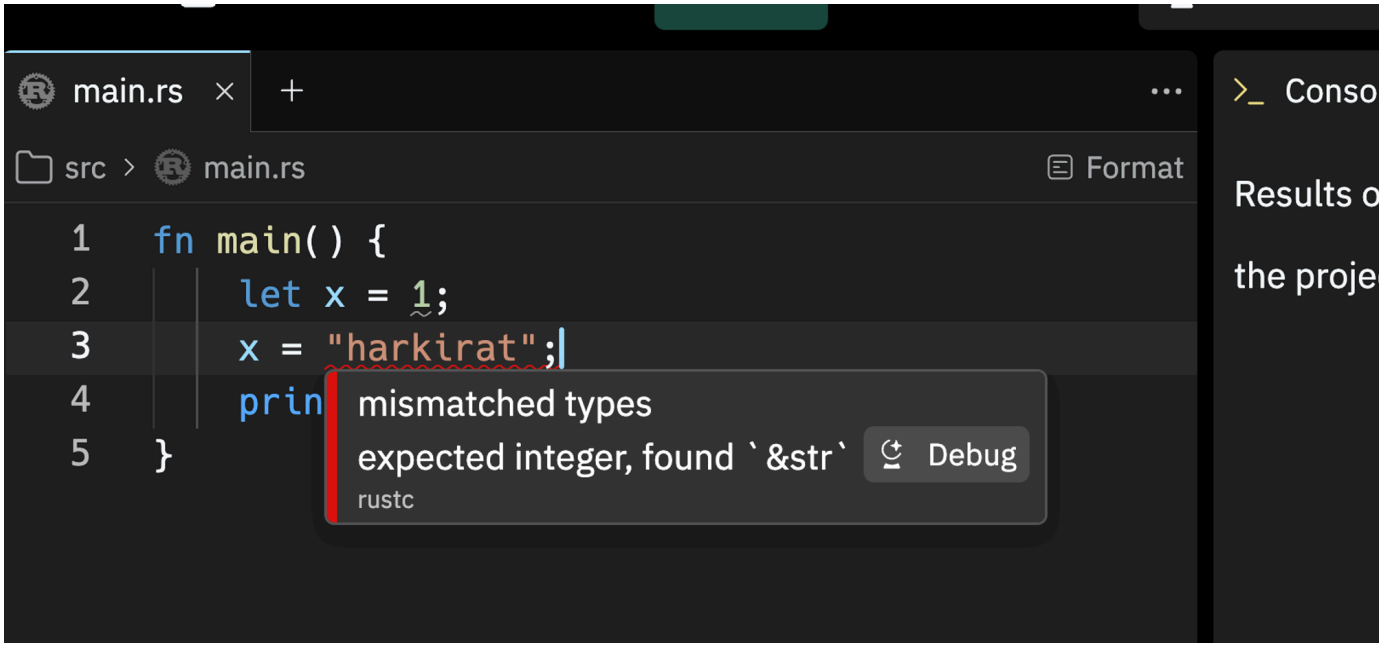
Why rust? Isn't Node.js enough?

Type safety

Javascript



Rust



💡 Typescript was introduced to get rid of some of these problems in JS

Systems language

It is intended to be used (but not restricted to) to do lower level things

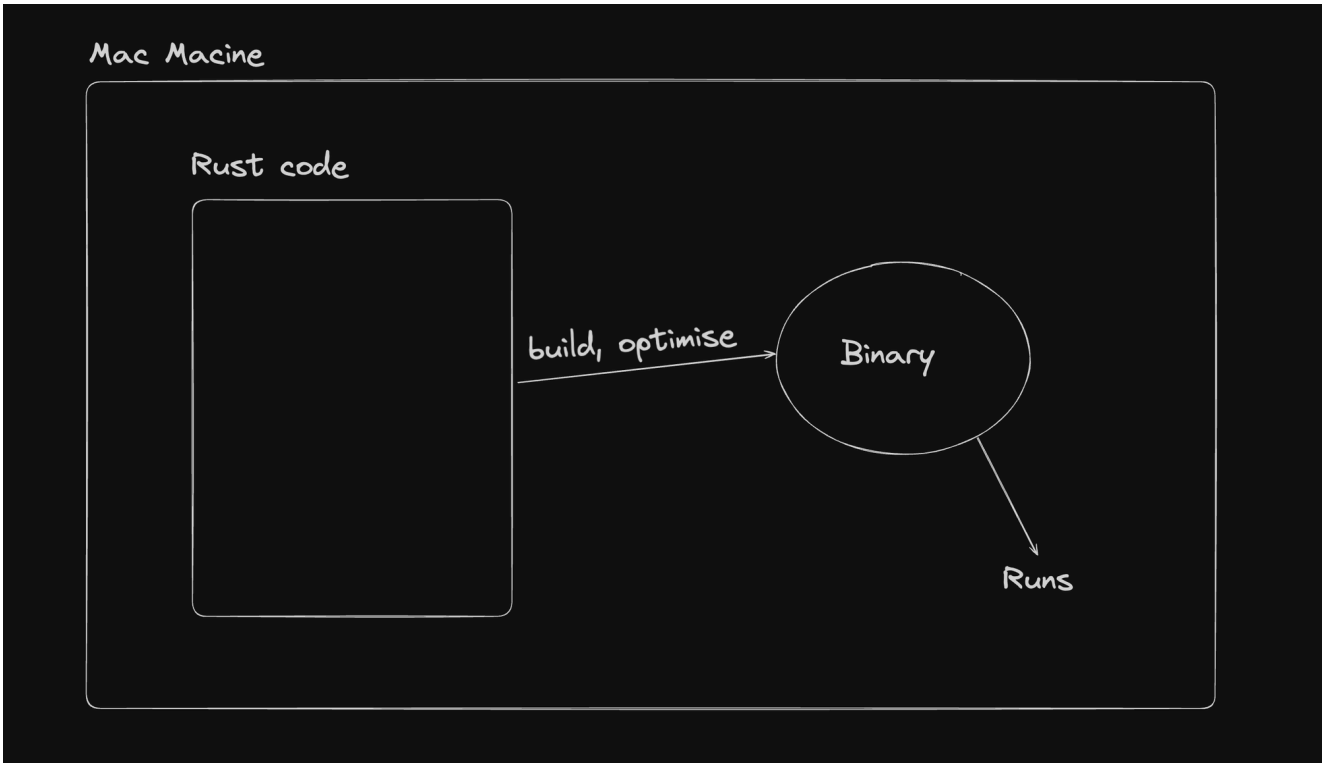
1. Building a Compiler
2. Building a browser

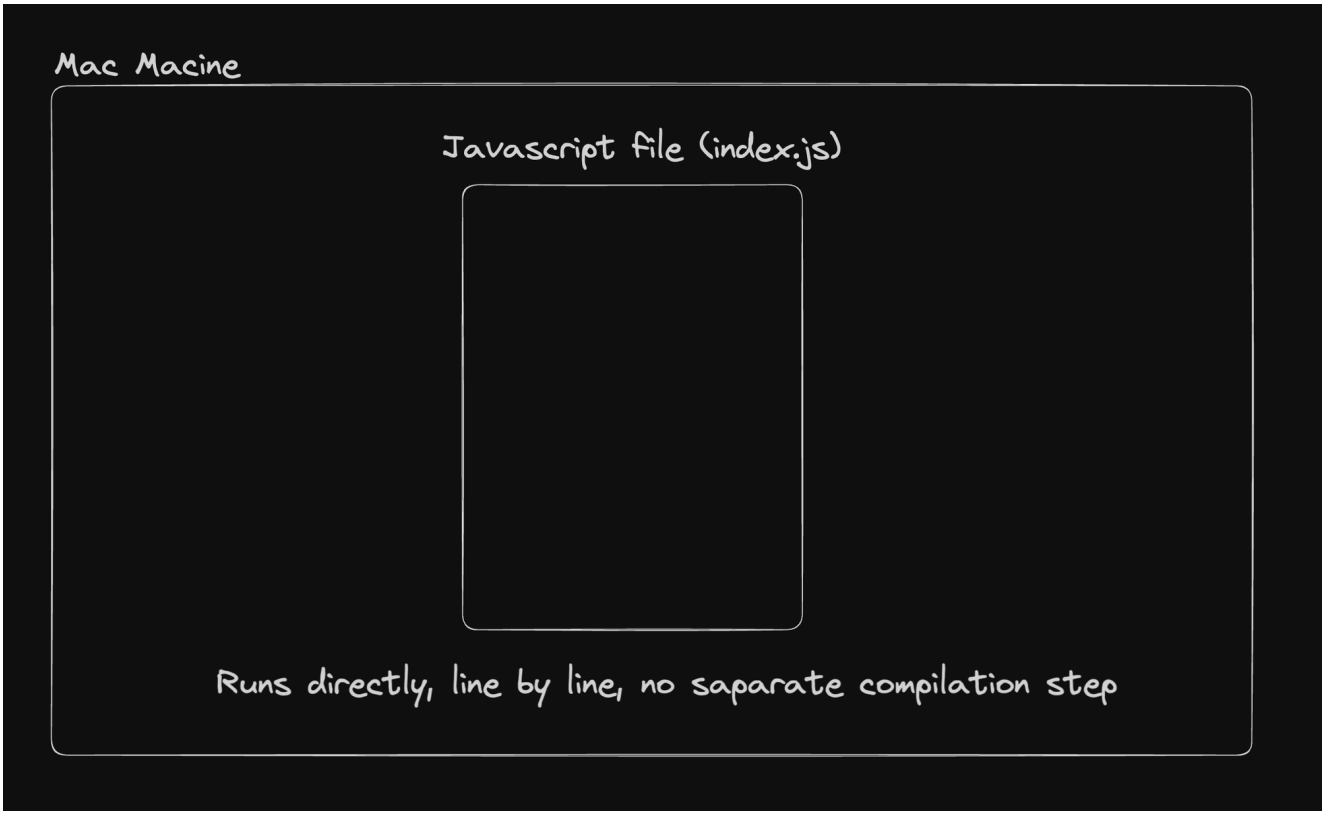
3. Working closer to the OS/kernel

Generally faster

Rust has a separate compilation step (similar to C++) that spits out an optimised binary and does a lot of static analysis at compile time.

JS does JIT compilation.





Concurrency

Rust has built-in support for concurrent programming allowing multiple threads to perform tasks simultaneously without risking data races

Javascript is single threaded generally (there are some projects that tried making it multi threaded but rarely used)

Memory safe

Rust has a concept of owners, borrowing and lifetimes that make it extremely memory safe

> 💡 Rust doesn't hide complexity from developers it offers them the right tools to manage all the complexity.

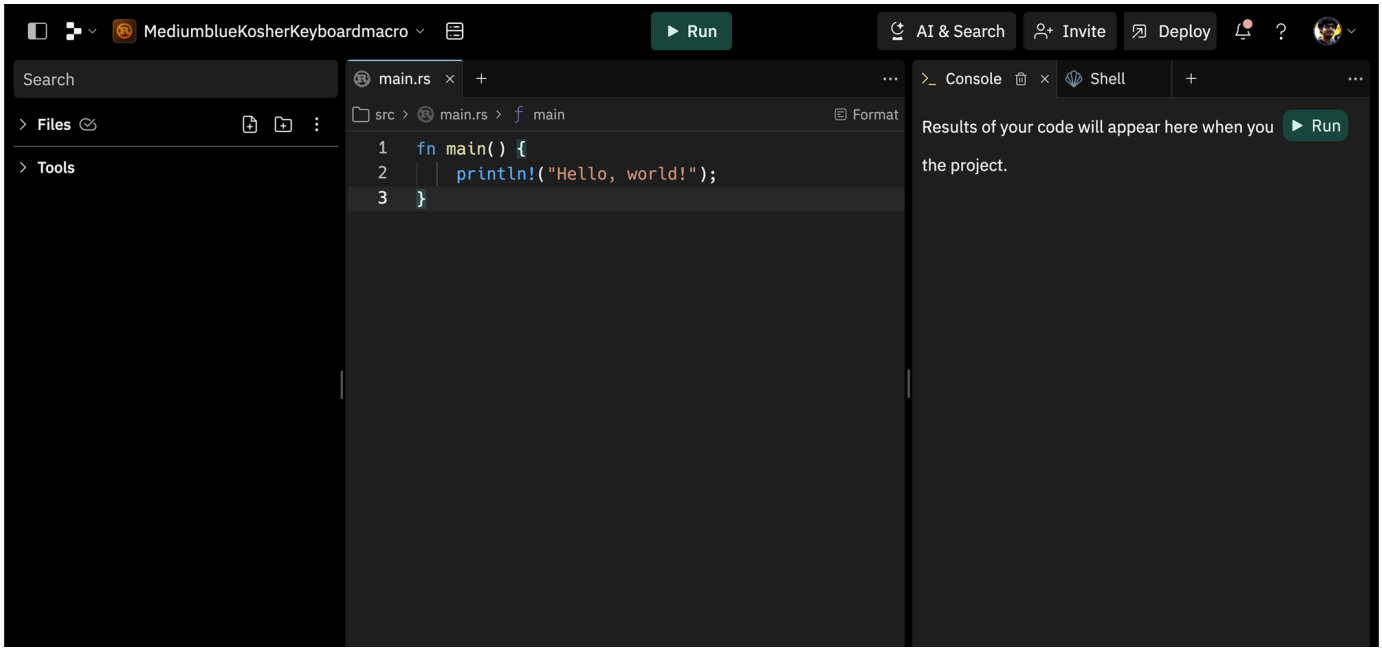
Initializing rust locally

Rust projects can be used to do a lot of things

1. Create Backend for a Full stack app
2. Create CLIs (command line interfaces)
3. Create browsers
4. Great Code Editors

For this bootcamp, we'll be getting comfortable with the syntax of Rust.

For most of the video, a repl.it playground should be good enough https://replit.com/~



But it's generally a good idea to start projects locally

Best place to find how to install it - https://www.rust-lang.org/tools/install

By the end of it, you should be able to run `cargo` in your terminal.

If not, it means you have an installation issue

```
➜  ~ cargo
Rust's package manager

Usage: cargo [+toolchain] [OPTIONS] [COMMAND]
       cargo [+toolchain] [OPTIONS] -Zscript <MANIFEST_RS> [ARGS]...

Options:
  -V, --version             Print version info and exit
      --list                List installed commands
      --explain <CODE>      Provide a detailed explanation of a rustc error message
  -v, --verbose...          Use verbose output (-vv very verbose/build.rs output)
  -q, --quiet               Do not print cargo log messages
      --color <WHEN>        Coloring: auto, always, never
  -C <DIRECTORY>            Change to DIRECTORY before doing anything (nightly-only)
      --frozen              Require Cargo.lock and cache are up to date
      --locked              Require Cargo.lock is up to date
      --offline             Run without accessing the network
      --config <KEY=VALUE>  Override a configuration value
  -Z <FLAG>                 Unstable (nightly-only) flags to Cargo, see 'cargo -Z help' for details
  -h, --help                Print help

Commands:
    build, b    Compile the current package
    check, c    Analyze the current package and report errors, but don't build object files
    clean       Remove the target directory
    doc, d      Build this package's and its dependencies' documentation
    new         Create a new cargo package
    init        Create a new cargo package in an existing directory
    add         Add dependencies to a manifest file
    remove      Remove dependencies from a manifest file
    run, r      Run a binary or example of the local package
    test, t     Run the tests
    bench       Run the benchmarks
    update      Update dependencies listed in Cargo.lock
    search      Search registry for crates
    publish     Package and upload this package to the registry
    install     Install a Rust binary. Default location is $HOME/.cargo/bin
    uninstall   Uninstall a Rust binary
    ...         See all commands with --list

See 'cargo help <command>' for more information on a specific command.
```

Cargo is similar to `npm` . It’s a package manager for rust

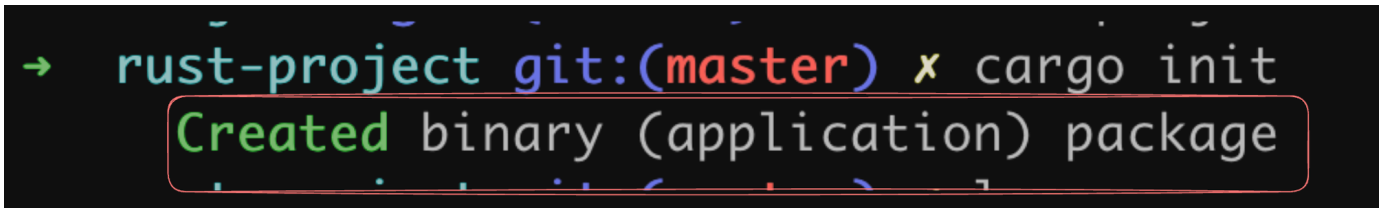
Initializing a rust project

Run the following command to bootstrap a simple Rust project

```
mkdir rust-project
cd rust-project
cargo init
```

lib vs application

Application



By default, a rust `application` gets initialised

It means an end user app that actually executes independently. Spits out a binary when compiled

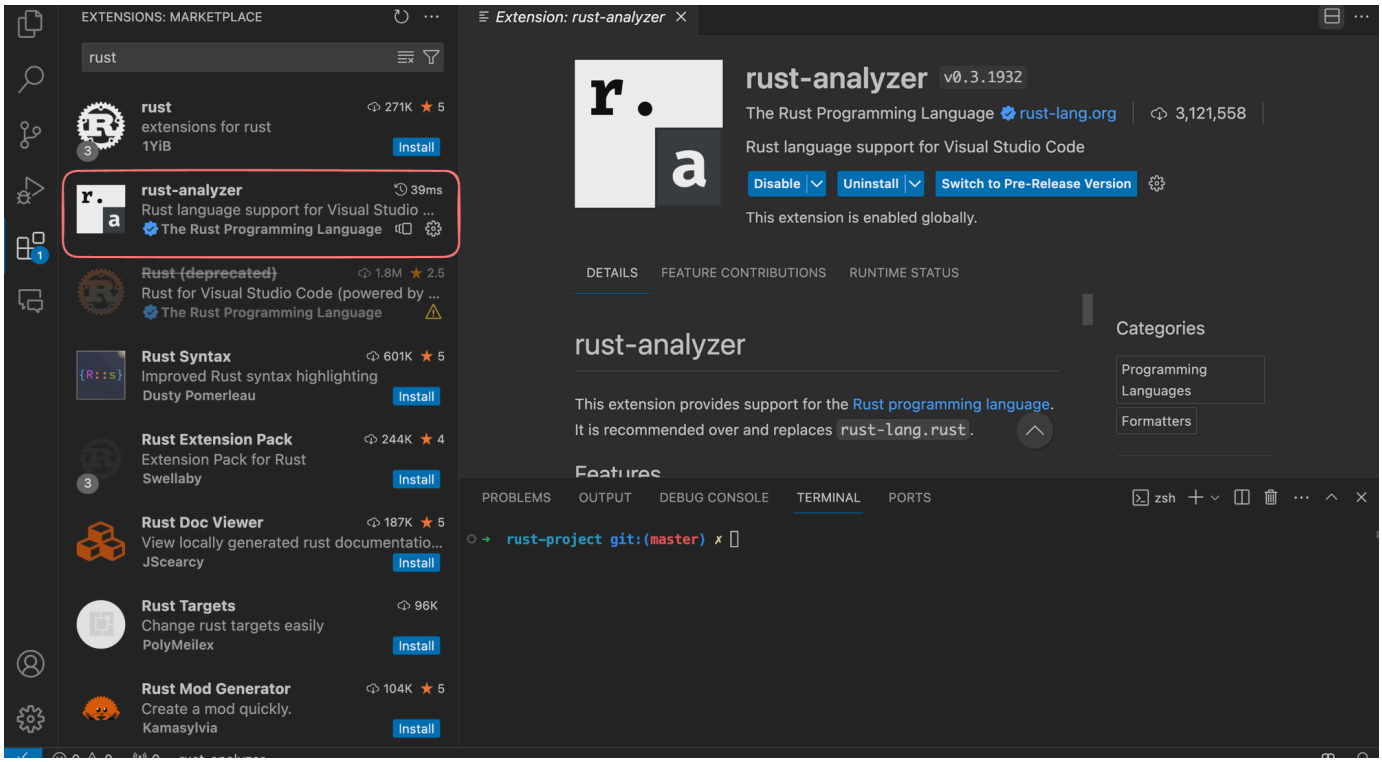
Lib

```
cargo init --lib
```

This would initialize a library that you can deploy for other people to use

VSCode setup + Hello world

1. Open the project in VSCode
2. Install the `rust-analyser` Extension



Optional extensions - CodeLLDB, toml

Hello world program

Rust code

▼ Equivalent Javascript code

```
function main() {
  console.log("Hello world")
}

main();
```

Simple Variables in rust

Before we write any more code, let's quickly discuss variables so we can understand some rust concepts before we get into the meat of the code

You can define variables using the `let` keyword (very similar to JS)

You can assign the type of the variable, or it can be inferred as well.

1. Numbers

```
fn main() {
    let x: i32 = 1;
    println!("{}", x);
}
```

Ignore how the printing is happening right now, we'll come to it later

▼ Equivalent typescript code

```
function main() {
  let x: number = 1;
  console.log(x);
}

main()
```

▼ What happens if we overflow?

```
fn main() {
    let mut num: i8 = 124;
    for i in 0..100 {
        num += 127;
    }
    print!("Number: {}", num)
}
```

2. Booleans

Bools can have two states, true or false

```
fn main() {
    let is_male = false;
    let is_above_18 = true;

    if is_male {
        println!("You are a male");

    } else {
        println!("You are not a male");
    }

    if is_male && is_above_18 {
        print!("You are a legal male");
    }
}
```

▼ Equivalent typescript code

```
function main() {
    let is_male = false;
    let is_above_18 = true;

    if (is_male) {
        console.log("You are a male");
    } else {
        console.log("You are not a male");
    }

    if (is_male && is_above_18) {
        console.log("You are a legal male");
    }
}

main();
```

3. Strings

There are two ways of doing `strings` in rust. We'll be focussing on the easier one

```
fn main() {
    let greeting = String::from("hello world");
```

```
    println!("{}", greeting);

    // print!("{}", greeting.chars().nth(1000))
}
```

▼ Equivalent typescript code

```
function main() {
  let greeting = "hello world";
  console.log(greeting);

  // console.log(greeting[1000]);
}
```

Conditionals, loops...

Conditionals

```
pub fn main() {
    let x = 99;
    let is_even = is_even(x);
    if is_even {
        print!("{} is even", x);
    } else {
        print!("{} is odd", x);
    }
}

pub fn is_even(x: i32) -> bool {
    return x % 2 == 0;
}
```

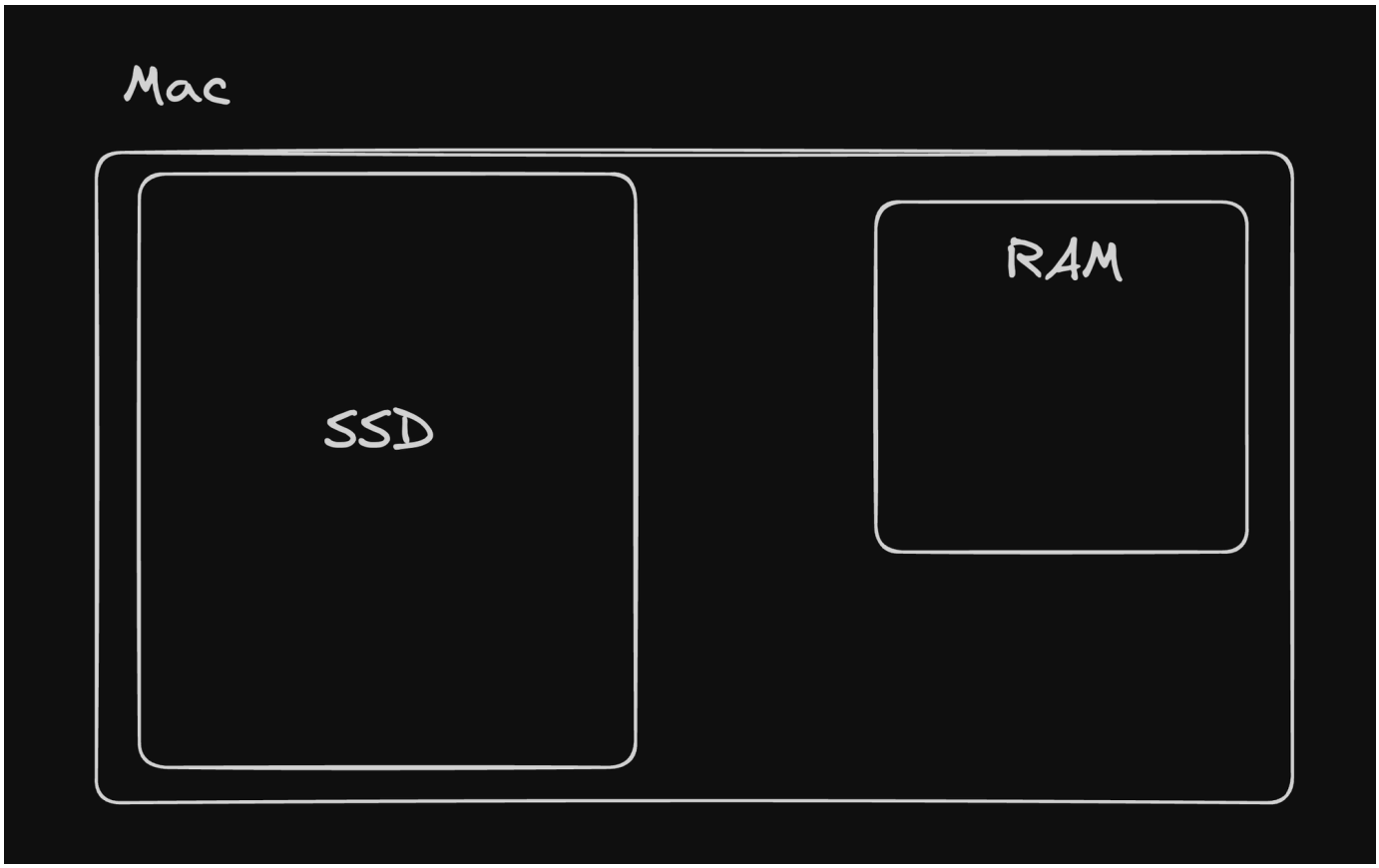
Loops

```rust
pub fn main() {
    let str = String::from("harkirat singh");
    println!("First name {}", get_first_name(str))

}

pub fn get_first_name(str: String) -> String {
    let mut first_name = String::from("");
    for c in str.chars() {
        if c == ' ' {
            break
        }
        first_name.push(c);
    }
    return first_name;
}
```

Functions

```
fn do_sum(a: i32, b: i32) -> i32 {
    return a + b;
}
```

Memory Management in Rust



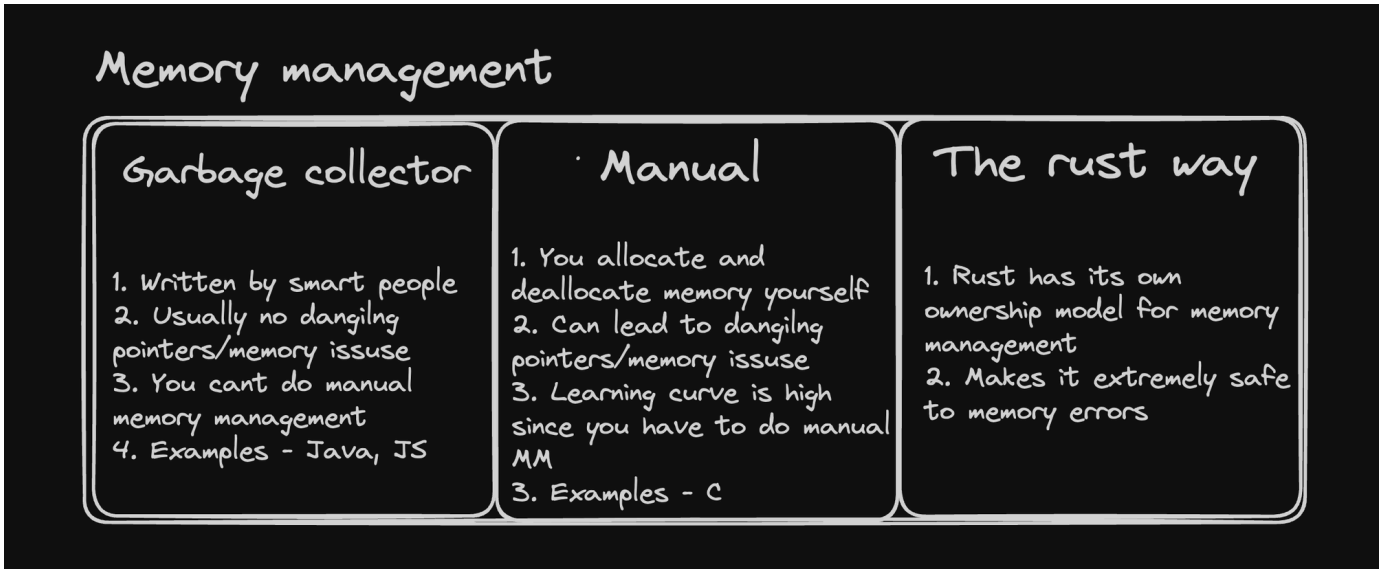
Whenever you run a program (C++, Rust, JS), it `allocates` and `deallocates` memory on the RAM.

For example, for the following JS code

```
function main() {
  runLoop();
}

function runLoop() {
  let x = [];
  for (let i = 0; i < 100000; i++) {
    x.push(1);
  }
  console.log(x);
}

main();
```

as the `runLoop` function runs, a new array is pushed to RAM, and eventually `garbage collected`

There are 3 popular ways of doing memory management



Memory management is a crucial aspect of programming in Rust, designed to ensure safety and efficiency without the need for a garbage collector.

Not having a `garbage collector` is one of the key reasons rust is so fast

It achieves this using the

1. Mutability
2. Heap and memory
3. Ownership model
4. Borrowing and references
5. Lifetimes

Jargon #0 - Mutability

Mutability

Immutable `variables` represent variables whose value cant be changed once assigned

```
fn main() {
    let x: i32 = 1;
    x = 2; // Error because x is immutable
    println!("{}", x);
}
```

By default, all variables in Rust are immutable because

1. Immutable data is inherently thread-safe because if no thread can alter the data, then no synchronization is needed when data is accessed concurrently.
2. Knowing that certain data will not change allows the compiler to optimize code better.

You can make variables mutable by using the `mut` keyword

```
fn main() {
    let mut x: i32 = 1;
    x = 2; // No error
    println!("{}", x);
}
```

💡 const in Javascript is not the same as immutable variables in rust. In JS, you can still update the contents of const `arrays` and `objects`

💡 JS also has a concept of immutability that a lot of libraries were built on in the past - https://www.npmjs.com/package/immutable

Jargon #1 - Stack vs heap

Stack vs. Heap Allocation

Rust has clear rules about stack and heap data management:

- **Stack**: Fast allocation and deallocation. Rust uses the stack for most primitive data types and for data where the size is known at compile time (eg: numbers).
- **Heap**: Used for data that can grow at runtime, such as vectors or strings.

What's stored on the stack?

Numbers - i32, i64, f64 ...

Booleans - true, false

Fixed sized arrays (we'll come to this later)

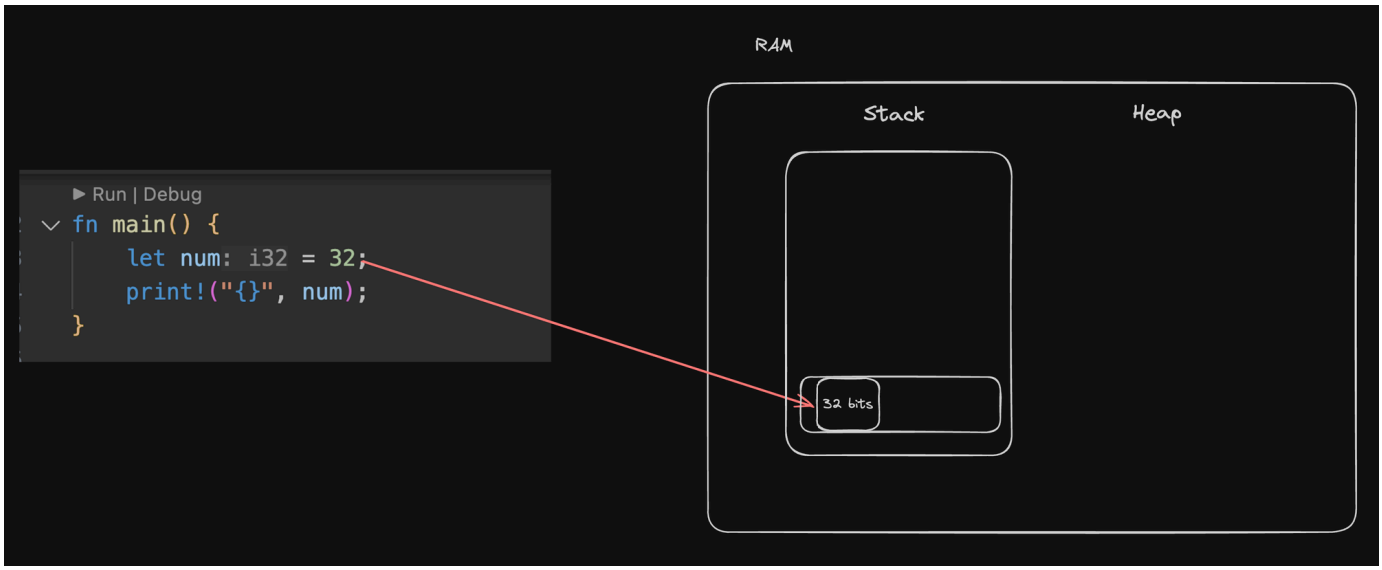
What's stored on heap?

Strings

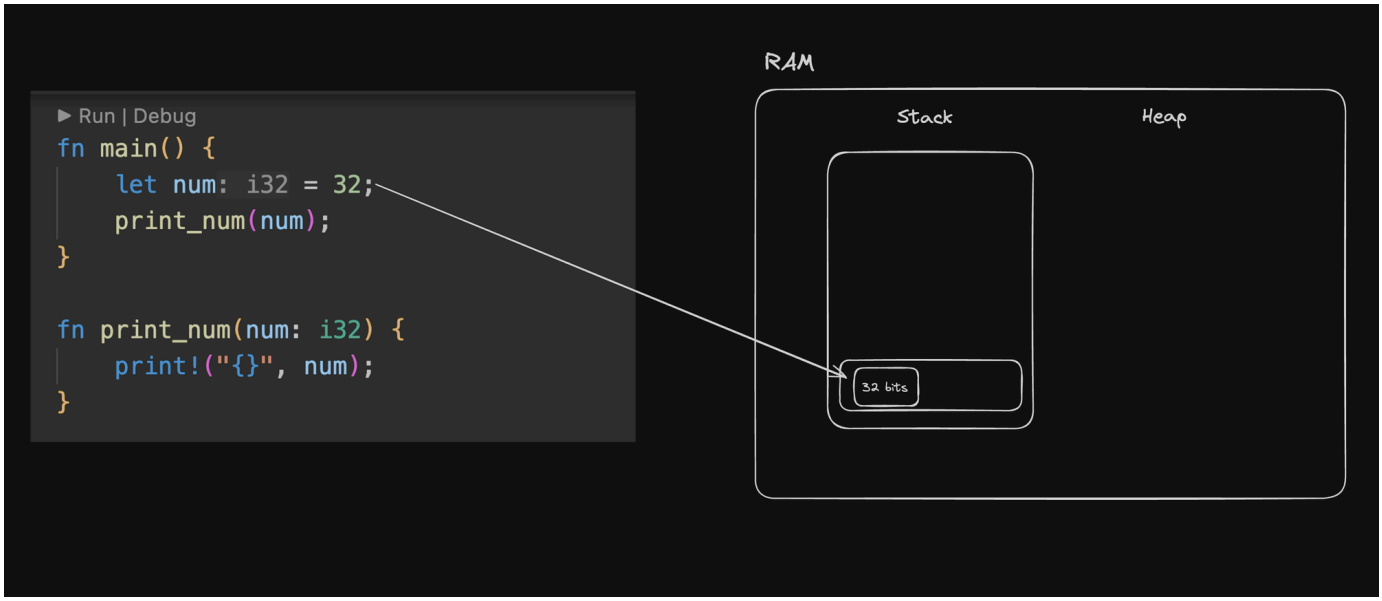
Vectors (we'll come to this later)

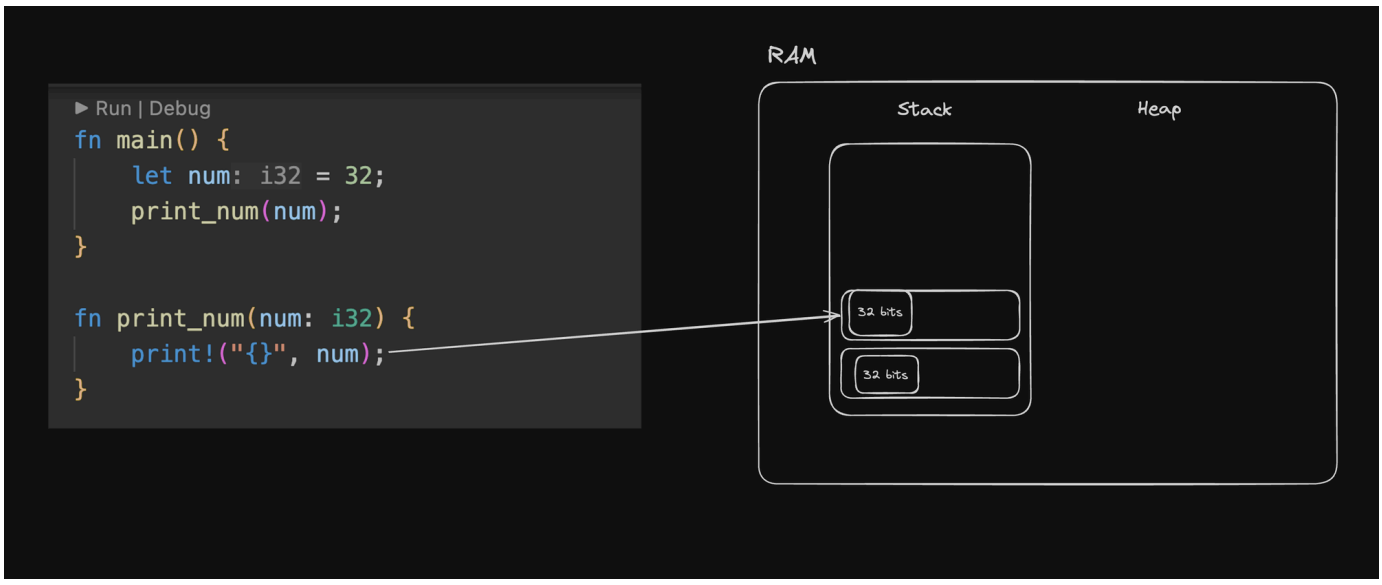
Examples

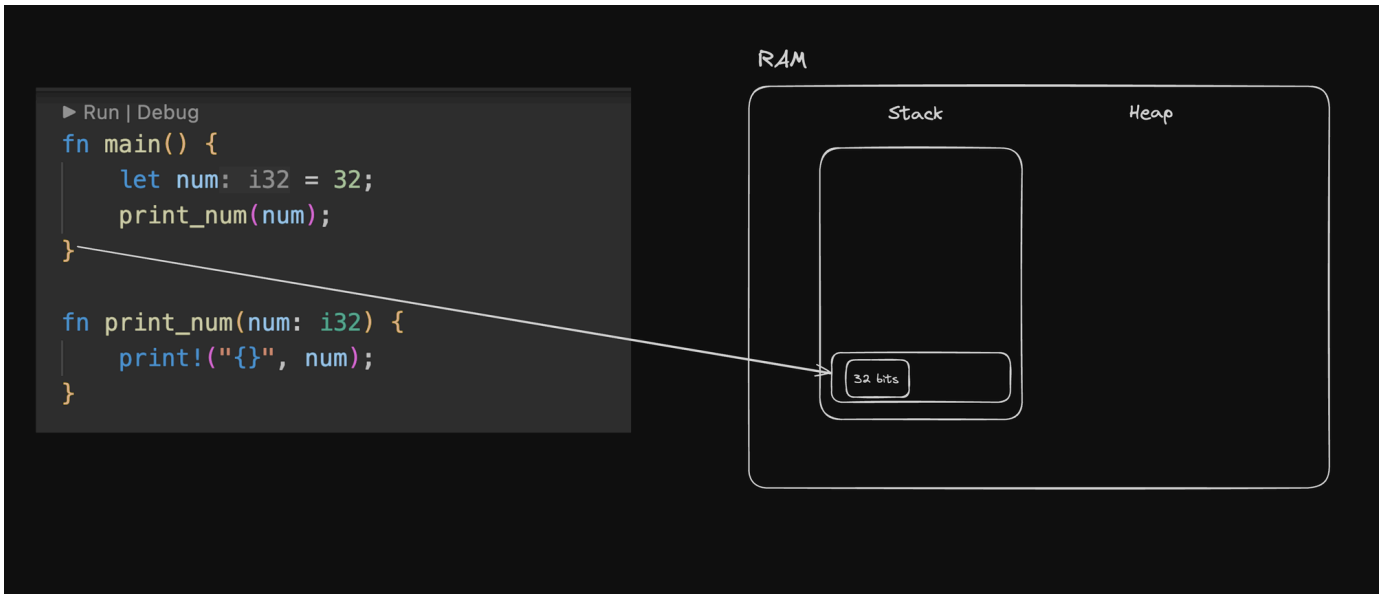
Hello world with numbers



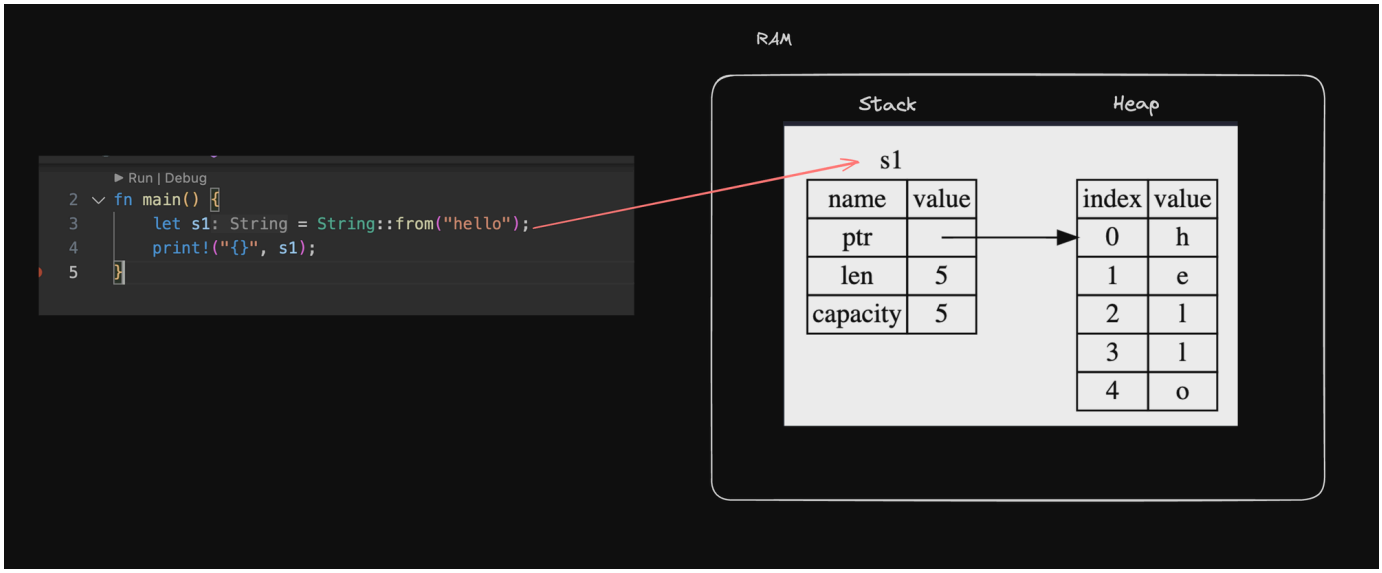
Hello world with functions







Hello world with strings



Memory in action

```
fn main() {
    stack_fn();   // Call the function that uses stack memory
    heap_fn();    // Call the function that uses heap memory
    update_string();  // Call the function that changes size of variable at runt
}

fn stack_fn() {
    // Declare a few integers on the stack
    let a = 10;
    let b = 20;
    let c = a + b;
    println!("Stack function: The sum of {} and {} is {}", a, b, c);
}
```

```rust
fn heap_fn() {
    // Create a string, which is allocated on the heap
    let s1 = String::from("Hello");
    let s2 = String::from("World");
    let combined = format!("{} {}", s1, s2);
    println!("Heap function: Combined string is '{}'", combined);
}

fn update_string() {
    // Start with a base string on the heap
    let mut s = String::from("Initial string");
    println!("Before update: {}", s);

    // Append some text to the string
    s.push_str(" and some additional text");
    println!("After update: {}", s);
}
```

Jargon #2 - Ownership

Ref - https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html

What Is Ownership?

Ownership is a set of rules that govern how a Rust program manages memory. All programs have to manage the way they use a computer's memory while running. Some languages have garbage collection that regularly looks for no-longer-used memory as the program runs; in other languages, the programmer must explicitly allocate and free the memory. Rust uses a third approach: memory is managed through a system of ownership with a set of rules that the compiler checks. If any of the rules are violated, the program won't compile. None of the features of ownership will slow down your program while it's running.

Meet Rihana

She always wants to keep a `boyfriend` (or owner) and can never remain single. She says if I ever become single (have no owners), I will die. She also can only have a single boyfriend at a time.

Stack variables

Example #1 - Passing stack Variables inside functions

```
fn main() {
        let x = 1; // crated on stack
        let y = 3; // created on stack
    println!("{}", sum(x, y));
    println!("Hello, world!");
}

fn sum(a: i32, b: i32) -> i32 {
    let c = a + b;
    return c;
}
```

This might sound trivial since if the function is popped of the stack, all variables go away with it, but check the next example

Example #2 - Scoping variables in the same fn

```
fn main() {
    let x = 1; // crated on stack
    {
        let y = 3; // created on stack
    }

    println!("{}", y); // throws error
}
```

Heap variables

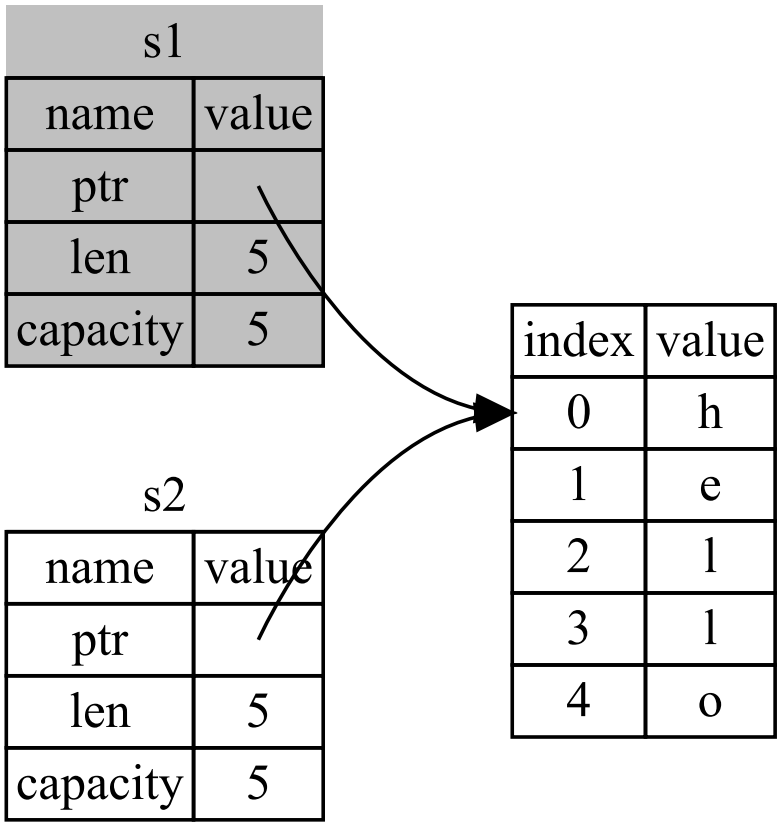
Heap variables are like Rihana. They always want to have a `single` owner, and if their owner goes out of scope, they get deallocated.

Any time the owner of a `heap variable` goes out of scope, the value is de-allocated from the heap.

Example #1 - Passing strings (heap variables) to functions as args

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;
```

```
    println!("{}", s1); // This line would cause a compile error because ownersh
}
```



Another example of the same thing -

```
fn main() {
    let my_string = String::from("hello");
    takes_ownership(my_string);
    println!("{}", my_string); // This line would cause a compile error because
}

fn takes_ownership(some_string: String) {
    println!("{}", some_string); // `some_string` now owns the data.
}
```

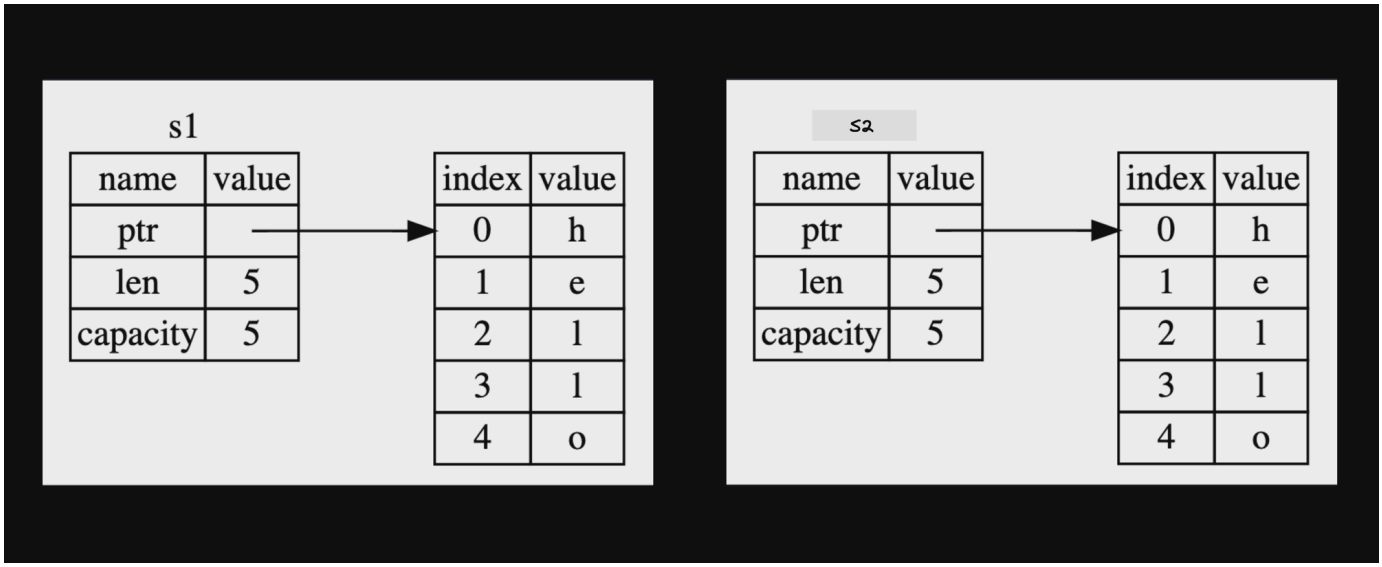
At any time, each value can have a `single` owner. This is to avoid memory issues like

1. Double free error.
2. Dangling pointers.

Fix?

Clone the string

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1.clone();
    println!("{}", s1); // Compiles now
}
```



But what if you want to pass the same string over to the function? You don't want to clone it, and you want to return back ownership to the original function?

You can either do the following -

```
fn main() {
    let s1 = String::from("hello");
    let s2 = takes_ownership(s1);
    println!("{}", s2);
}

fn takes_ownership(some_string: String) -> String {
    println!("{}", some_string);
    return some_string; // return the string ownership back to the original main
}
```

▼ You can also do this

```
fn main() {
    let mut s1 = String::from("hello");
    s1 = takes_ownership(s1);
    println!("{}", s2);
```

```
}

fn takes_ownership(some_string: String) -> String {
    println!("{}", some_string);
    return some_string; // return the string ownership back to the original ma
}
```

Is there a better way to pass strings (or generally heap related data structures) to a function without passing over the ownership?
Yes - References

Jargon #3 - Borrowing and references

Rihana upgrades

Rihana now says I'd like to be borrowed from time to time. I will still have a `single owner`, but I can still be borrowed by other variables temporarily. What rules do you think should apply to her?

1. She can be borrowed by multiple people that she's friends with but does no hanky panky
2. If she does want to do hanky panky, she can only have `1` borrower that she does it with. She cant simultaneously be with other borrowers (even with no hanky panky)

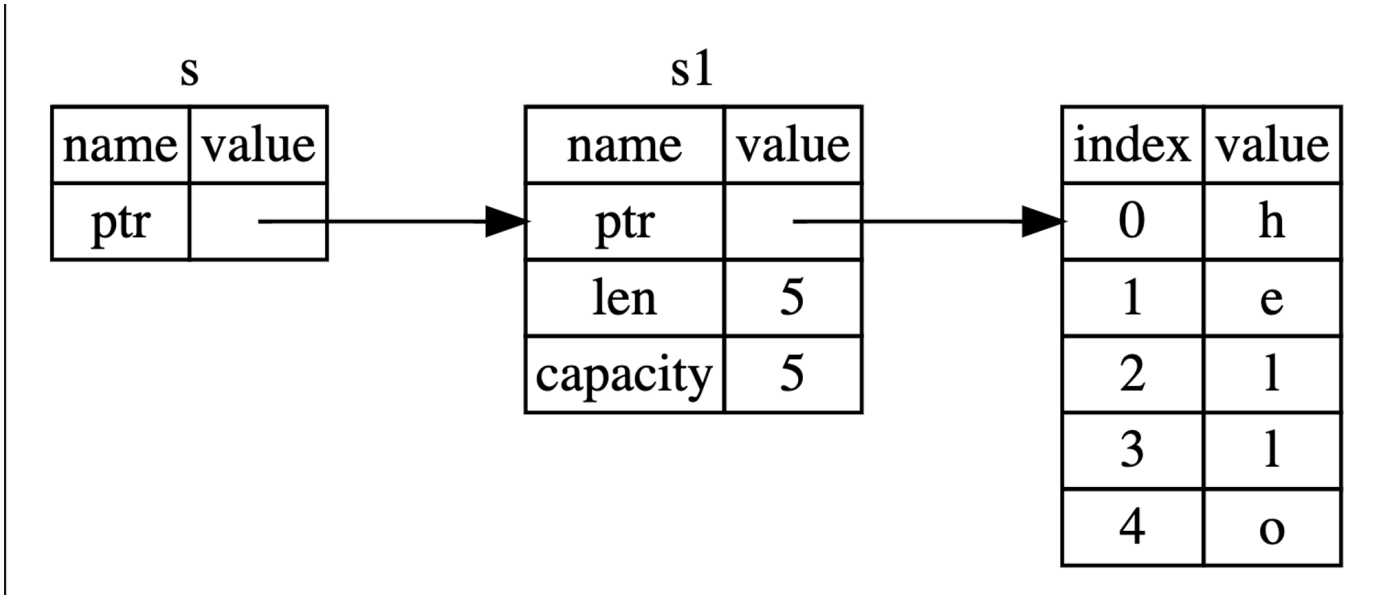
References

References mean giving the address of a string rather than the ownership of the string over to a function

For example

```
fn main() {
    let s1 = String::from("Hello");
    let s2 = &s1;

    println!("{}", s2);
```

```
    println!("{}", s1);    // This is valid, The first pointer wasn't invalidate
}
```



Borrowing

You can transferring ownership of variables to fns. By passing a reference to the string to the function `take_ownership`, the ownership of the string remains with the original variable, in the `main` function. This allows you to use `my_string` again after the function call.

```
fn main() {
    let my_string = String::from("Hello, Rust!");
    takes_ownership(&my_string);  // Pass a reference to my_string
    println!("{}", my_string);    // This is valid because ownership was not tra
}

fn takes_ownership(some_string: &String) {
    println!("{}", some_string);  // some_string is borrowed and not moved
}
```

Mutable references

What if you want a function to `update` the value of a variable?

```
fn main() {
    let mut s1 = String::from("Hello");
    update_word(&mut s1);
    println!("{}", s1);
}
```

```
fn update_word(word: &mut String) {
    word.push_str(" World");
}
```

Try having more than one mutable reference at the same time -

```
fn main() {
    let mut s1 = String::from("Hello");
    let s2 = &mut s1;
    update_word(&mut s1);
    println!("{}", s1);
    println!("{}", s2);
}

fn update_word(word: &mut String) {
    word.push_str(" World");
}
```

Rules of borrowing

- There can me many `immutable references` at the same time

```
fn main() {
    let  s1 = String::from("Hello");
    let s2 = &s1;
    let s3 = &s1;

    println!("{}", s1);
    println!("{}", s2);
    println!("{}", s3);
}
// No errors
```

- There can be only one `mutable reference` at a time

```
fn main() {
    let mut s1 = String::from("Hello");
    let s2 = &mut s1;
    let s3 = update_word(&mut s1);
```

```
    println!("{}", s1);
    println!("{}", s2);
}

fn update_word(word: &mut String) {
    word.push_str(" World");
}
// Error
```

- If there is a `mutable reference` , you can't have another immutable reference either.

```
fn main() {
    let mut s1 = String::from("Hello");
    let s2 = &mut s1;
    let s3 = &s1;

    println!("{}", s1);
    println!("{}", s2);
}

fn update_word(word: &mut String) {
    word.push_str(" World");
}
```

This to avoid any data races/inconsistent behaviour

If someone makes an `immutable reference` , they don't expect the value to change suddenly

If more than one `mutable references` happen, there is a possibility of a data race and synchronization issues

> 💡 Two good things to discuss at this point should be **but we're going to ignore it for now**
> 1. `Lifetimes`
> 2. **String slices (&str)**

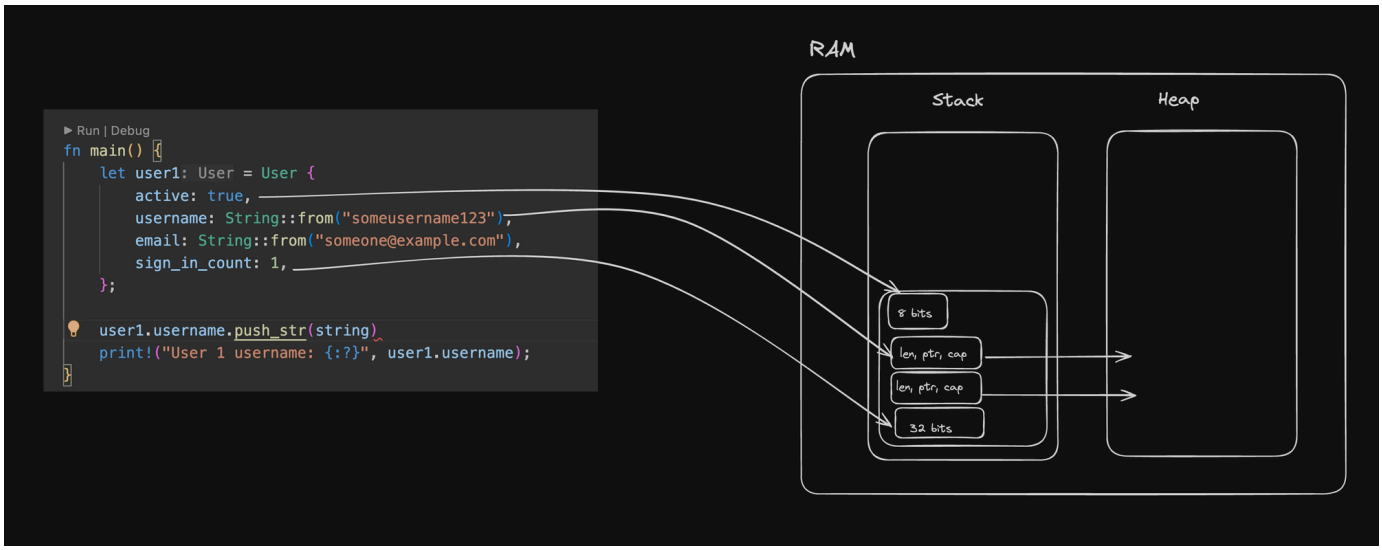
Ref - https://doc.rust-lang.org/book/ch04-03-slices.html

Structs

Structs in rust let you structure data together. Similar to `objects` in javascript

```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}

fn main() {
    let user1 = User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    };
    print!("User 1 username: {:?}", user1.username);
}
```

Can you guess if they are stored in `stack` or `heap` ?



Side quest - Learning about traits

Try running experiments around

1. `mutable` and `immutable` references

2. Ownership transfer

for structs

Code 1 - Only stack types in the struct

```
struct User {
    active: bool,
    sign_in_count: u64,
}

fn main() {
    let mut user1 = User {
        active: true,
        sign_in_count: 1,
    };

    print_name(user1);
    print!("User 1 username: {}", user1.active); // Error - can not use borrowed
}

fn print_name(user1: User) {
    print!("User 1 username: {}", user1.active);
}
```

Add the `copy trait`

```
#[derive(Copy, Clone)]
struct User {
    active: bool,
    sign_in_count: u64,
}

fn main() {
    let mut user1 = User {
        active: true,
        sign_in_count: 1,
    };

    print_name(user1);
    print!("User 1 username: {}", user1.active); // Error goes away because user
}
```

```
fn print_name(user1: User) {
    print!("User 1 username: {}", user1.active);
}
```

Code 2 - Add strings

```
struct User {
    active: bool,
    sign_in_count: u64,
    username: String,
}

fn main() {
    let mut user1 = User {
        active: true,
        sign_in_count: 1,
        username: "harkirat".to_string()
    };

    change_name(user1);
    print!("User 1 username: {}", user1.active); // Error - can not use borrowed
}

fn change_name(user1: User) {
    print!("User 1 username: {:?}", user1.active);
}
```

Try adding the `Copy trait` (you wont be able to because strings dont implement them, use `clone` trait instead)

```
#[derive(Clone)]
struct User {
    active: bool,
    sign_in_count: u64,
    username: String,
}

fn main() {
    let mut user1 = User {
        active: true,
```

```
        sign_in_count: 1,
        username: "harkirat".to_string()
    };

    change_name(user1.clone());
    print!("User 1 username: {}", user1.active); // Error - can not use borrowed
}

fn change_name(user1: User) {
    print!("User 1 username: {:?}", user1.active);
}
```

Implementing structs

You can also `implement structs` , which means you can attach functions to instances of structs

(Very similar to classes in TS)

```
struct Rect {
   width: u32,
   height: u32,
}

impl Rect {
    fn area(&self) -> u32 {
         self.width * self.height
    }
}

fn main() {
    let rect = Rect {
        width: 30,
        height: 50,
    };
    print!("The area of the rectangle is {}", rect.area());
}
```

Enums

Enums in rust are similar to enums in Typescript. They allow you to define a type by enumerating its possible *variants*.

Ref - https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html

```
enum Direction {
    North,
    East,
    South,
    West,
}

fn main() {
    let my_direction = Direction::North;
    let new_direction = my_direction; // No error, because Direction is Copy
    move_around(new_direction);
}

fn move_around(direction: Direction) {
    // implements logic to move a character around
}
```

Why not simply do the following -

```
fn main() {
    move_around("north".to_string());
}

fn move_around(direction: String) {
    if direction == "north" {
        println!("Moving North");
    }
}
```

Because we don't enforce the `4` variants of directions. So this is much looser than strictly allowing only 4 variants for direction

Enums with values

```
// Define an enum called Shape
enum Shape {
    Circle(f64),  // Variant with associated data (radius)
    Square(f64),  // Variant with associated data (side length)
    Rectangle(f64, f64),  // Variant with associated data (width, height)
}

// Function to calculate area based on the shape
fn calculate_area(shape: Shape) -> f64 {
    // calculates the area of the shape
    return 0
}

fn main() {
    // Create instances of different shapes
    let circle = Shape::Circle(5.0);
    let square = Shape::Square(4.0);
    let rectangle = Shape::Rectangle(3.0, 6.0);

}
```

We will be implementing the `calcuate_area` function in the `pattern matching` section

Pattern matching

Let you pattern match across various variants of an enum and run some logic

```
// Define an enum called Shape
enum Shape {
    Circle(f64),  // Variant with associated data (radius)
    Square(f64),  // Variant with associated data (side length)
    Rectangle(f64, f64),  // Variant with associated data (width, height)
}

// Function to calculate area based on the shape
fn calculate_area(shape: Shape) -> f64 {
    match shape {
        Shape::Circle(radius) => std::f64::consts::PI * radius * radius,
        Shape::Square(side_length) => side_length * side_length,
        Shape::Rectangle(width, height) => width * height,
    }
}

fn main() {
    // Create instances of different shapes
    let circle = Shape::Circle(5.0);
    let square = Shape::Square(4.0);
    let rectangle = Shape::Rectangle(3.0, 6.0);

    // Calculate and print the areas
    println!("Area of circle: {}", calculate_area(circle));
    println!("Area of square: {}", calculate_area(square));
    println!("Area of rectangle: {}", calculate_area(rectangle));
}
```

Error handling

Different languages have different ways to handle errors.
Javascript, for example, has the concept of try catch blocks

```
try {
    const data = fs.readFileSync('example.txt', 'utf8');
    console.log("File content:", data);
} catch (err) {
    console.error("Error reading the file:", err);
}
```

The reason we put the code inside a try catch block is that `reading a file` is `unpredictable`. The file might not exist, the file might be locked by another process, and hence there is a possibility of this code `throwing an error`

The same is true for a rust program trying to access a file. But the way rust does error handling is slightly different

Result Enum

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

If you look at the code above, it is an enum (with generic types)

This enum is what a function can return/returns when it has a possibility of throwing an error

For example

```
use std::fs::File;

fn main() {
    let greeting_file_result = fs::read_to_string("hello.txt");
}
```

Notice the type of `greeting_file_result` in VSCode

It returns an enum that looks as follows. It's an enum with the `Ok` variant having a string value and `Err` variant having an Error value

```
enum Result{
    Ok(String),
    Err(Error),
}
```

Complete code

```
use std::fs;

fn main() {
    let greeting_file_result = fs::read_to_string("hello.txt");

    match greeting_file_result {
        Ok(file_content) => {
            println!("File read successfully: {:?}", file_content);
        },
        Err(error) => {
            println!("Failed to read file: {:?}", error);
        }
    }
}
```

Incase you write a function yourself, you can also return a `Result` from it. As the name suggests, `Result` holds the `result` of a function call that might lead to an error.

Unwraps

Incase you are ok with runtime errors (crashing the process while it runs if an error happens), then you can `unwrap` a `Result`

```
use std::fs;

fn main() {
    let greeting_file_result = fs::read_to_string("hello.txt");
```

```
    print!("{}", greeting_file_result.unwrap());
}
```

Returning a custom error

```
use core::fmt;
use std::{fmt::{Debug, Formatter}, fs};

pub struct FileReadError {

}

fn main() {
    let contents = read_file("hello.txt".to_string());
    match contents {
        Ok(file_content) => {
            println!("File content: {}", file_content);
        },
        Err(error) => {
            println!("Error reading file: {:?}", error);
        }
    }
}

fn read_file(file_path: String) -> Result<String, FileReadError> {
    let greeting_file_result = fs::read_to_string("hello.txt");
    match greeting_file_result {
        Ok(file_content) => {
            Ok(file_content)
        },
        Err(error) => {
            let err = FileReadError {};
            Err(err)
        }
    }
}
```

Option enum

Ref - https://viblo.asia/p/billion-dollar-mistake-RQqKLopr57z

The Option enum was introduced in Rust to handle the concept of nullability in a safe and expressive way. Unlike many programming languages that use a null or similar keyword to represent the absence of a value, Rust doesn't have null.

```
pub enum Option<T> {
    None,
    Some(T),
}
```

If you ever have a function that should return null, return an `Option` instead.

For example

```
fn find_first_a(s: String) -> Option<i32> {
    for (index, character) in s.chars().enumerate() {
        if character == 'a' {
            return Some(index as i32);
        }
    }
    return None;
}

fn main() {
    let my_string = String::from("raman");
    match find_first_a(my_string) {
        Some(index) => println!("The letter 'a' is found at index: {}", index),
        None => println!("The letter 'a' is not found in the string."),
    }
}
```

Collections

https://doc.rust-lang.org/book/ch08-00-common-collections.html

Cargo, packages and external deps

Just like the `nodejs` ecosystem has `npm` , the rust ecosystem has `cargo`

Cargo is a `package manager` for rust, which means we can use it to bring packages (crates in case of rust) to our projects

Generate a random number

Use crate - https://crates.io/crates/rand

Run `cargo add rand`

```
use rand::{Rng, thread_rng};

fn main() {
    let mut rng = thread_rng();
    let n: u32 = rng.gen();
    println!("Random number: {}", n);
}
```

Store time in a DB/as a variable

Use crate https://crates.io/crates/chrono

Run `cargo add chrono`

```
use chrono::{Local, Utc};

fn main() {
    // Get the current date and time in UTC
    let now = Utc::now();
    println!("Current date and time in UTC: {}", now);

    // Format the date and time
    let formatted = now.format("%Y-%m-%d %H:%M:%S");
    println!("Formatted date and time: {}", formatted);

    // Get local time
    let local = Local::now();
```

```
    println!("Current date and time in local: {}", local);
}
```

What all libraries does rust have?

A lot of them

1. https://actix.rs/ - Extremely fast http server
2. https://serde.rs/ - Serializing and deserialising data in rust
3. https://tokio.rs/ - Asynchronous runtime in rust
4. https://docs.rs/reqwest/latest/reqwest/ - Send HTTP requests
5. https://docs.rs/sqlx/latest/sqlx/ - Connect to sql database

Leftovers - Traits, Generics and Lifetimes, Multithreading, macros, async ops (Futures)

https://doc.rust-lang.org/book/ch10-00-generics.html

https://doc.rust-lang.org/book/ch19-00-advanced-features.html

https://doc.rust-lang.org/book/ch20-00-final-project-a-web-server.html

https://doc.rust-lang.org/std/future/trait.Future.html