Rust bootcamp

Beginner to Advance

Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

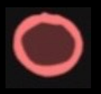
1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Part 2

1. Collections, vectors
2. Iterators
3. Hashmaps
4. Strings, &str and slices
3. Generics
4. Traits
5. Multithreading
6. Macros
8. Futures
9. Async/await and tokio
10. Lifetimes

Easy

Medium



Hard

Before we start, lets recap Part 1

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Recapping Part 1

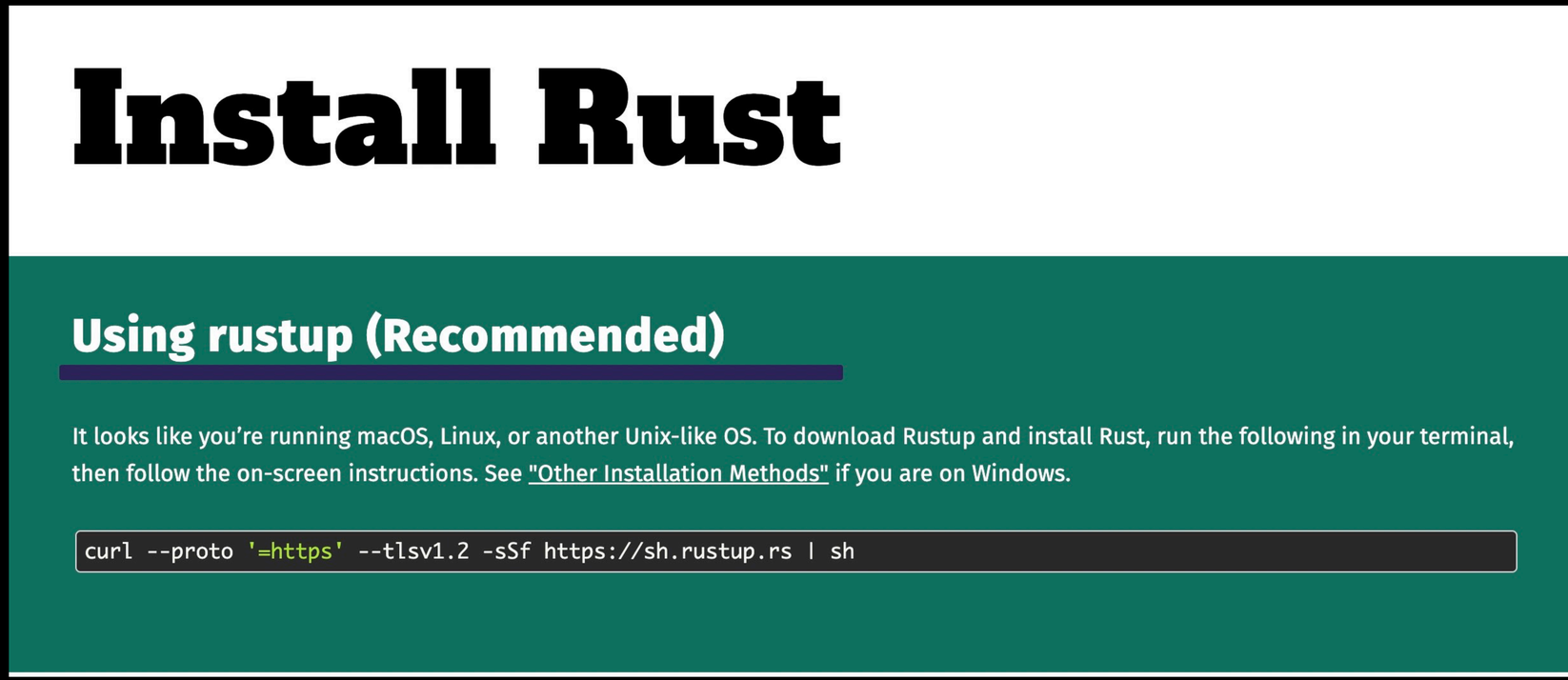
Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

1. Steps to install - https://www.rust-lang.org/tools/install



Recapping Part 1

Easy

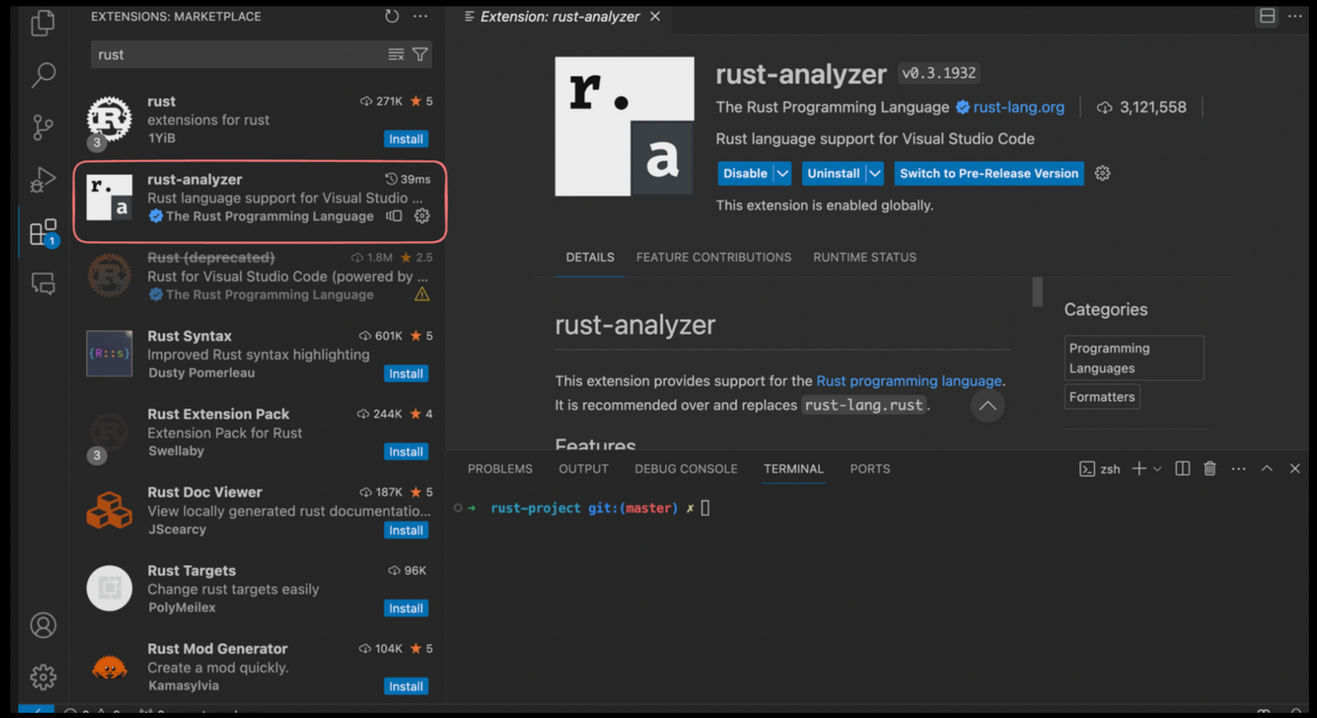
1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

1. Steps to install - https://www.rust-lang.org/tools/install

2. Install VSCode Extensions - rust-analyser , CodeLLDB, toml



Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

1. Start a project (binary)

```
→  rust-project git:(master) ✗ cargo init
     Created binary (application) package
```

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

1. Start a project (binary)

```
→  rust-project git:(master) ✗ cargo init
     Created binary (application) package
```

2. Start a project (library)

```
→  rust-project git:(master) ✗ cargo init --lib
     Created library package
```

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Q. Write a function is_even that takes a number as an input and returns true if it is even

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

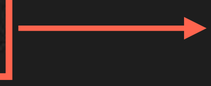
Q. Write a function is_even that takes a number as an input and returns true if it is even

```rust
fn main() {
    println!("{}", is_even(20));
}

fn is_even(num: i32) -> bool {
    if num % 2 == 0 {
        return true;
    }
    return false;
}
```

```rust
fn main() {
    println!("{}", is_even(20));
}

fn is_even(num: i32) -> bool {
    if num % 2 == 0 {
        return true;
    }
    return false;
}
```

Defining the main function

```rust
fn main() {
    println!("{}", is_even(20));
}

fn is_even(num: i32) -> bool {
    if num % 2 == 0 {
        return true;
    }
    return false;
}
```



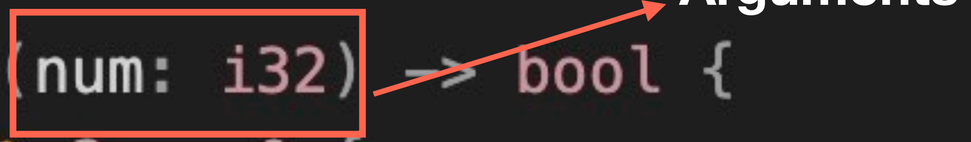
Calling a function

```rust
fn main() {
    println!("{}", is_even(20));
}

fn is_even(num: i32) -> bool {
    if num % 2 == 0 {
        return true;
    }
    return false;
}
```

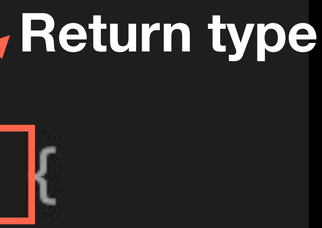
Defining a function

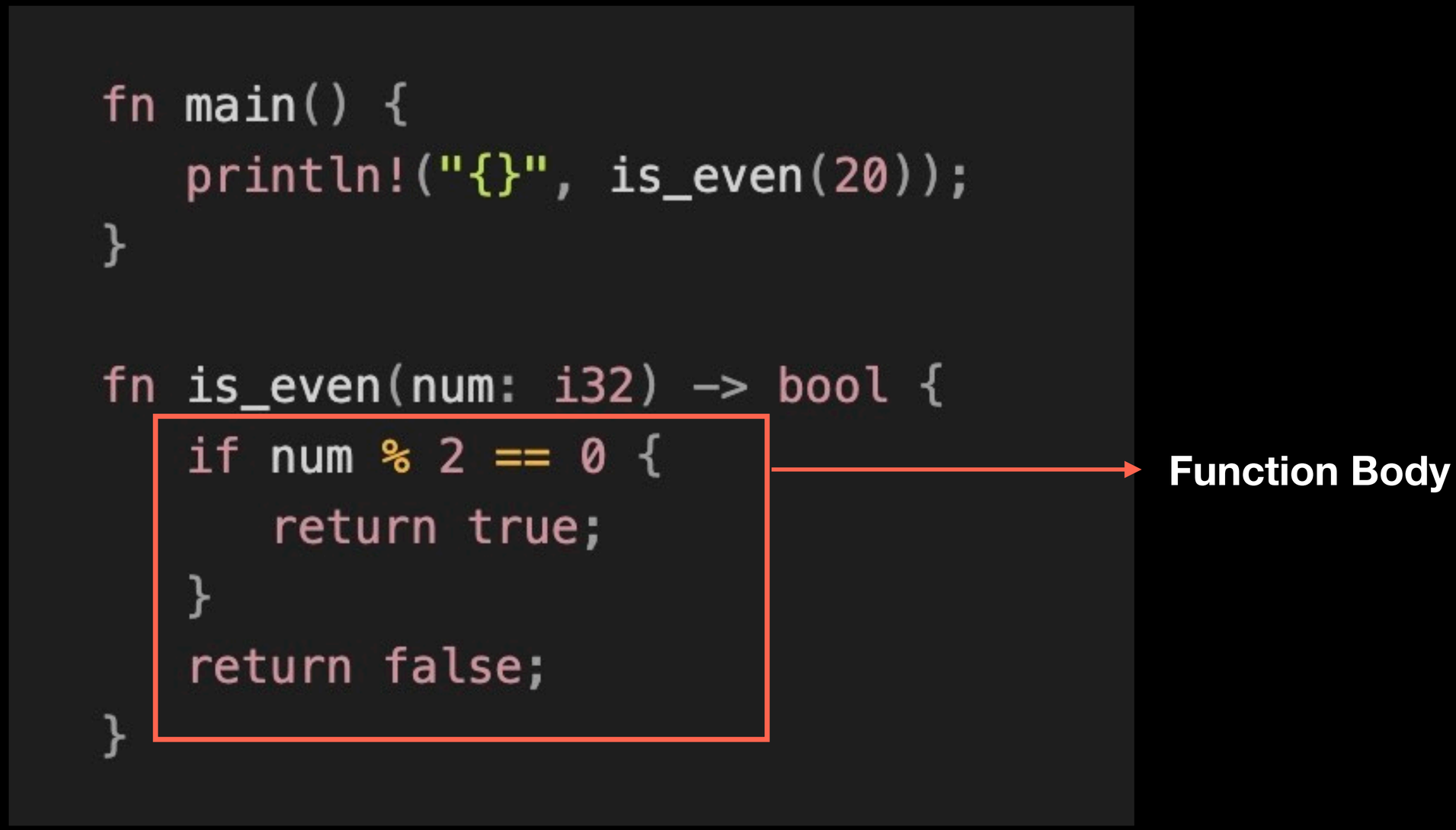
```rust
fn main() {
    println!("{}", is_even(20));
}

fn is_even(num: i32) -> bool {
    if num % 2 == 0 {
        return true;
    }
    return false;
}
```

```rust
fn main() {
    println!("{}", is_even(20));
}

fn is_even(num: i32) -> bool {
    if num % 2 == 0 {
        return true;
    }
    return false;
}
```



```rust
fn main() {
    println!("{}", is_even(20));
}

fn is_even(num: i32) -> bool {
    if num % 2 == 0 {
        return true;
    }
    return false;
}
```




fn main() {
println!("{}", is_even(20));
}
fn is_even(num: i32) -> bool {
if num % 2 == 0 {
return true;
}
return false;
}
Function Body

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Q. Write a function is_even that takes a number as an input and returns true if it is even

```rust
fn main() {
    println!("{}", is_even(20));
}

fn is_even(num: i32) -> bool {
    if num % 2 == 0 {
        return true;
    }
    return false;
}
```

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Q. Write a function fib that finds the fibbonacci of a number it takes as input

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Q. Write a function fib that finds the fibbonacci of a number it takes as input

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

fn fib(num: i32) -> i32 {
    let mut first = 0;
    let mut second = 1;
    if (num == 0) {
        return first;
    }
    if (num == 1) {
        return 1;
    }

    for i in 1..num - 2 {
        let temp = second;
        second = second + first;
        first = temp;
    }
    return second;
}
```

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

fn fib(num: i32) -> i32 {
    let mut first = 0;
    let mut second = 1;
    if (num == 0) {
        return first;
    }
    if (num == 1) {
        return 1;
    }

    for i in 1..num - 2 {
        let temp = second;
        second = second + first;
        first = temp;
    }
    return second;
}
```

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

fn fib(num: i32) -> i32 {
    let mut first = 0;
    let mut second = 1;
    if (num == 0) {
        return first;
    }
    if (num == 1) {
        return 1;
    }

    for i in 1..num - 2 {
        let temp = second;
        second = second + first;
        first = temp;
    }
    return second;
}
```

Mutability

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

fn fib(num: i32) -> i32 {
    let mut first = 0;
    let mut second = 1;
    if (num == 0) {
        return first;
    }
    if (num == 1) {
        return 1;
    }

    for i in 1..num - 2 {
        let temp = second;
        second = second + first;
        first = temp;
    }
    return second;
}
```

If statement

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

fn fib(num: i32) -> i32 {
    let mut first = 0;
    let mut second = 1;
    if (num == 0) {
        return first;
    }
    if (num == 1) {
        return 1;
    }

    for i in 1..num - 2 {
        let temp = second;
        second = second + first;
        first = temp;
    }
    return second;
}
```

For loops

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Q. Write a function get_string_length that takes a string as an input and returns its length

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Q. Write a function get_string_length that takes a string as an input and returns its length

```rust
fn get_string_length(s: &str) -> usize {
    s.chars().count()
}

fn main() {
    let my_string = String::from("Hello, world!");
    let length = get_string_length_chars(&my_string);
    println!("The number of characters in the string is: {}", length);
}
```

```rust
fn get_string_length(s: &str) -> usize {
    s.chars().count()
}

fn main() {
    let my_string = String::from("Hello, world!");
    let length = get_string_length_chars(&my_string);
    println!("The number of characters in the string is: {}", length);
}
```

Defining a string

```rust
fn get_string_length(s: &str) -> usize {
    s.chars().count()
}

fn main() {
    let my_string = String::from("Hello, world!");
    let length = get_string_length_chars(&my_string);
    println!("The number of characters in the string is: {}", length);
}
```

Implicit Return

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Q. Write a function get_string_length that takes a string as an input and returns its length

```rust
fn get_string_length(s: &str) -> usize {
    s.chars().count()
}

fn main() {
    let my_string = String::from("Hello, world!");
    let length = get_string_length_chars(&my_string);
    println!("The number of characters in the string is: {}", length);
}
```

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Structs let you structure data together

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Structs let you structure data together

```
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);
}
```

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Implementing structs

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Implementing structs

```
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());
}
```

```rust
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());
}
```

Defining the struct

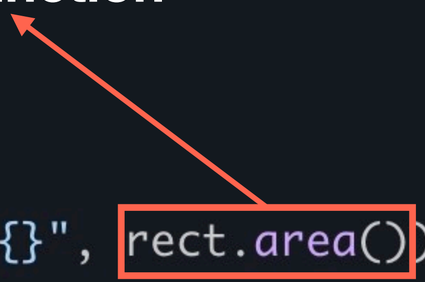
```rust
struct Rect {
    width: u32,
    height: u32,
}

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());
}
```

Implementing Functions on the struct

```rust
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());
}
```

Declaring a variable

```
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());
}
```



Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Structs let you structure data together

```
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);
}
```

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Enums let you enumerate over various types of a value

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Enums let you enumerate over various types of a value

```
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
}
```

Recapping Part 1

```rust
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
}
```

Defining the enum

Recapping Part 1

```rust
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
}
```

Declaring the enum

Recapping Part 1

```
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
}
```

Enum as an argument

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Enums let you enumerate over various types of a value

```
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
}
```

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

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);

}
```

Recapping Part 1

Has an associated radius

```
// 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);

}
```

Recapping Part 1

Has an associated side

```
// 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);

}
```

Recapping Part 1

Has an associated width and height

```
// 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);

}
```

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Enums let you enumerate over various types of a value

```
// 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);

}
```

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

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

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

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

How would you Implement this?

```
// 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);

}
```

Recapping Part 1

```
// 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,
    }
}
```

Recapping Part 1

match on the enum

```
// 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,
    }
}
```

Recapping Part 1

If type is circle

```
// 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,
    }
}
```

Recapping Part 1

Then return pi * r * r

```
// 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,
    }
}
```

Recapping Part 1

If type is square

```
// 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,
    }
}
```

Recapping Part 1

Then return side * side

```
// 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,
    }
}
```

Recapping Part 1

If shape is rectangle

```
// 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,
    }
}
```

Recapping Part 1

Then return 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,
    }
}
```

Recapping Part 1

Return the result implicitly

```
// 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,
    }
}
```

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

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

How would you Implement this?

```
// 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);

}
```

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

The Option enum lets you return either Some value or None Value

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

The Option enum lets you return either Some value or None Value

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

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Q: Write a function that returns the index of the first “a” in a string

```
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."),
    }
}
```

Recapping Part 1

```
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."),
    }
}
```

Returns an Option of type I32

Recapping Part 1

```
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."),
    }
}
```

If “a” is found

Recapping Part 1

```
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."),
    }
}
```

Returns the Some variant of the enum

Recapping Part 1

```
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."),
    }
}
```

Else it returns the None variant

Recapping Part 1

```rust
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."),
    }
}
```

We pattern match on the result

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Q: Write a function that returns the index of the first “a” in a string

```
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."),
    }
}
```

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

The Result enum lets you return either Ok value or Err Value
The result enum is how you can do error handling in Rust

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Q: Write a function that reads the contents of a file

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

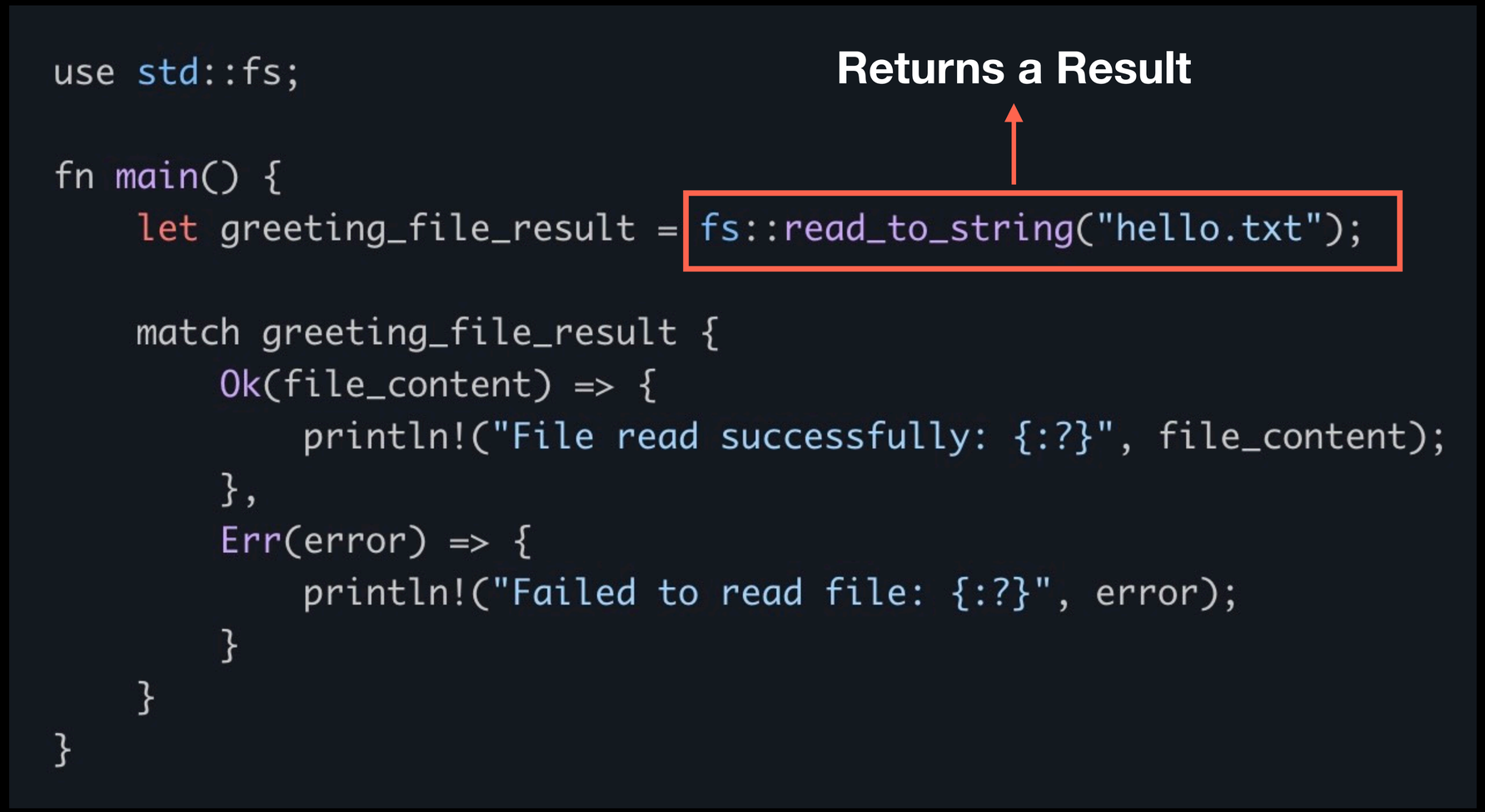
Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Q: Write a function that reads the contents of a file

```rust
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);
        }
    }
}
```

Returns a Result



```rust
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);
        }
    }
}
```

```rust
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);
        }
    }
}
```

Pattern matching on the result

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

You can add an external crate to your project by running

cargo add crate_name

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

You can add an external crate to your project by running
cargo add chrono

```rust
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);
}
```

```rust
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);
}
```

Use from external crate (Similar to import)

```rust
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);
}
```

Call a fn from an external crate

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

That covers the easy things from Part 1

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

More OS specific concepts

Recapping Part 1

Easy

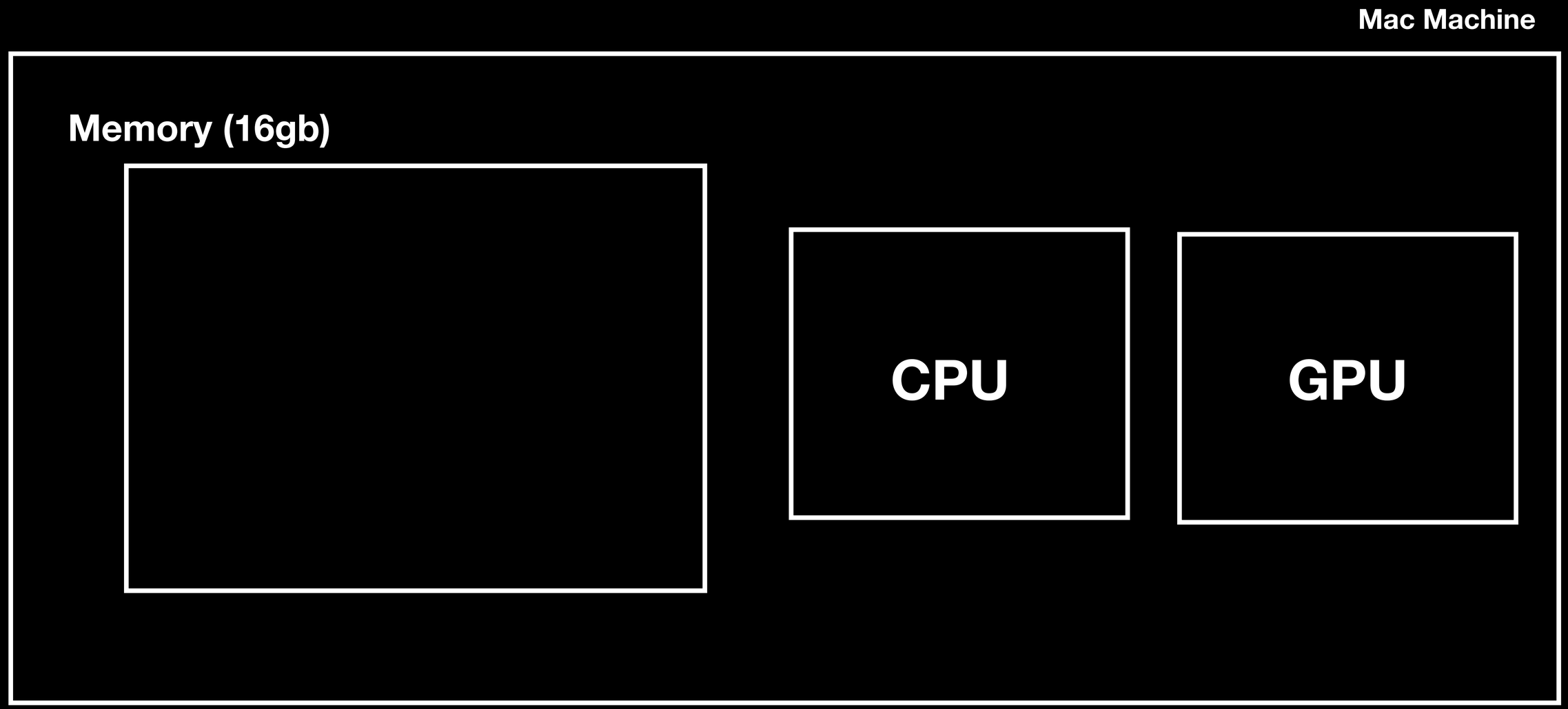
1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

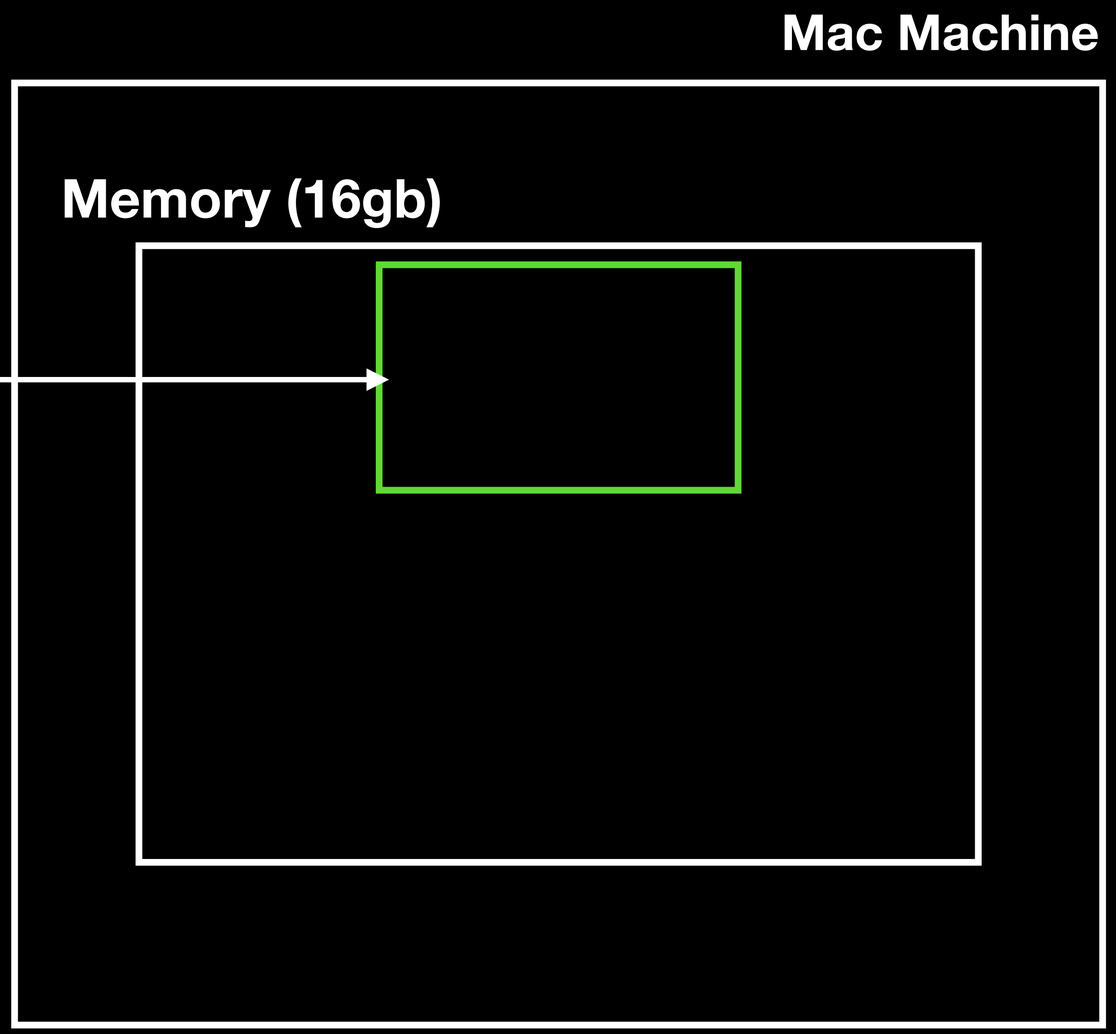
Memory management

What is memory?



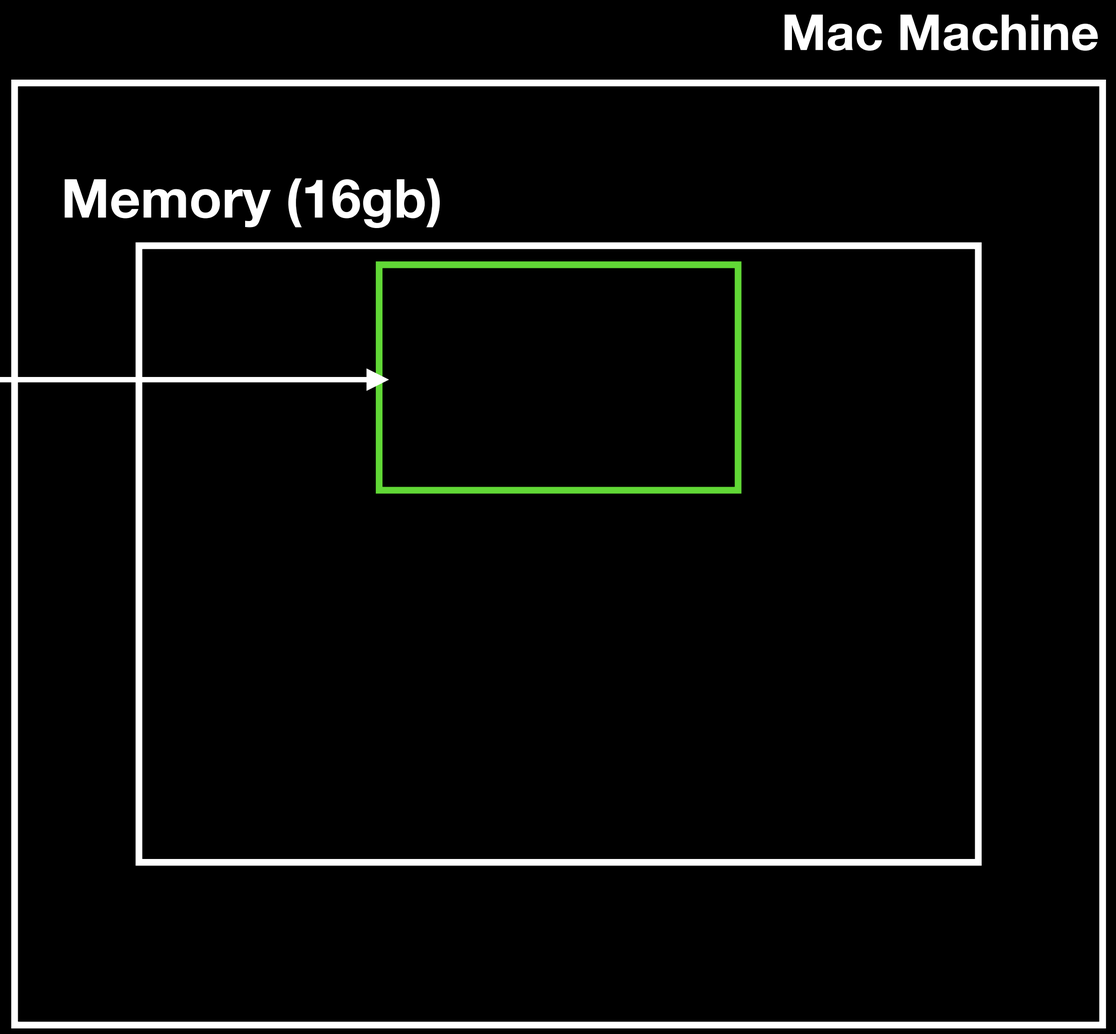
Memory management

```
➜  rust-project git:(master) ✗ cargo run
   Compiling rust-project v0.1.0 (/Users/kirat/Projects/test33/rust-project)
    Finished dev [unoptimized + debuginfo] target(s) in 9.87s
     Running `target/debug/rust-project`
Hello, world!
```



Memory management

```
➜  rust-project git:(master) ✗ cargo run
   Compiling rust-project v0.1.0 (/Users/kirat/Projects/test33/rust-project)
    Finished dev [unoptimized + debuginfo] target(s) in 9.87s
     Running `target/debug/rust-project`
Hello, world!
```



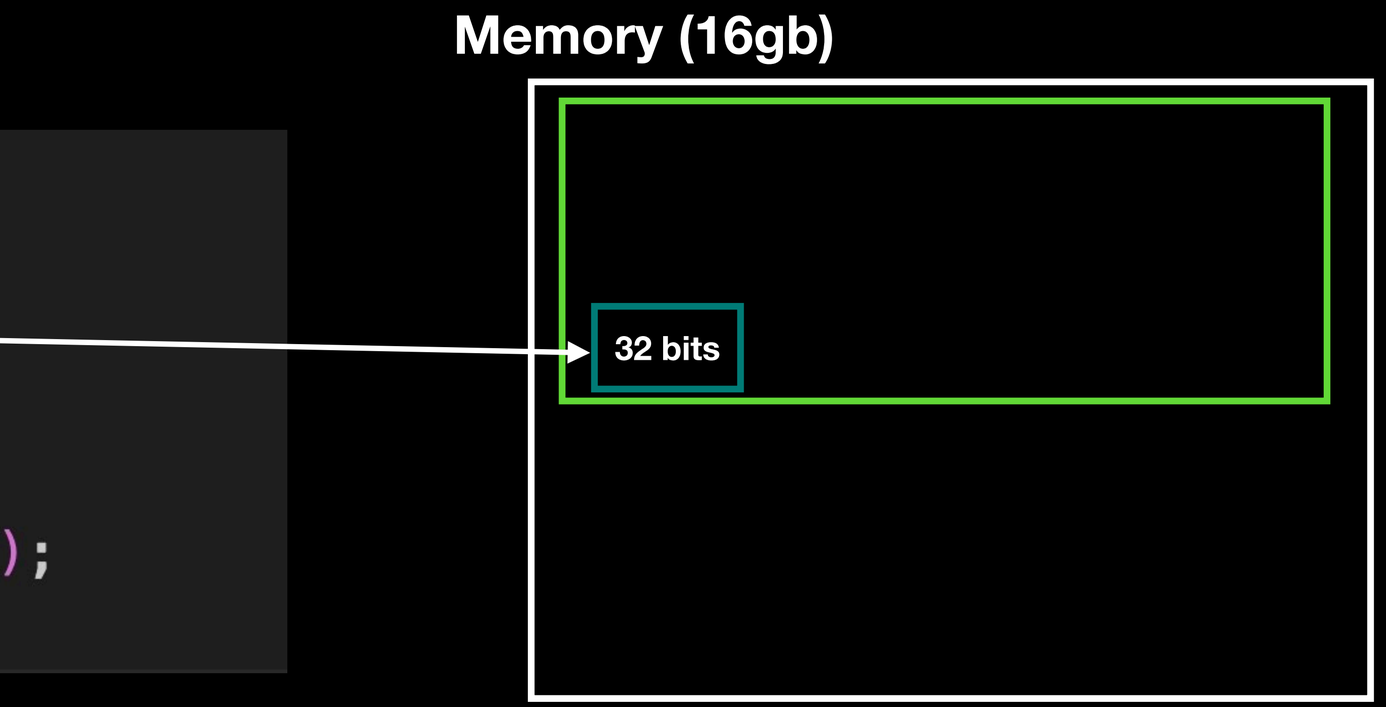
How much space do you think A program needs?

Memory management

```
▶ Run | Debug
fn main() {
    let a: i32 = 1;
    let b: i32 = 2;
    let sum: i32 = a + b;
    println!("Sum is {}", sum);
}
```

Memory management

```
▶ Run | Debug
fn main() {
    let a: i32 = 1;
    let b: i32 = 2;
    let sum: i32 = a + b;
    println!("Sum is {}", sum);
}
```

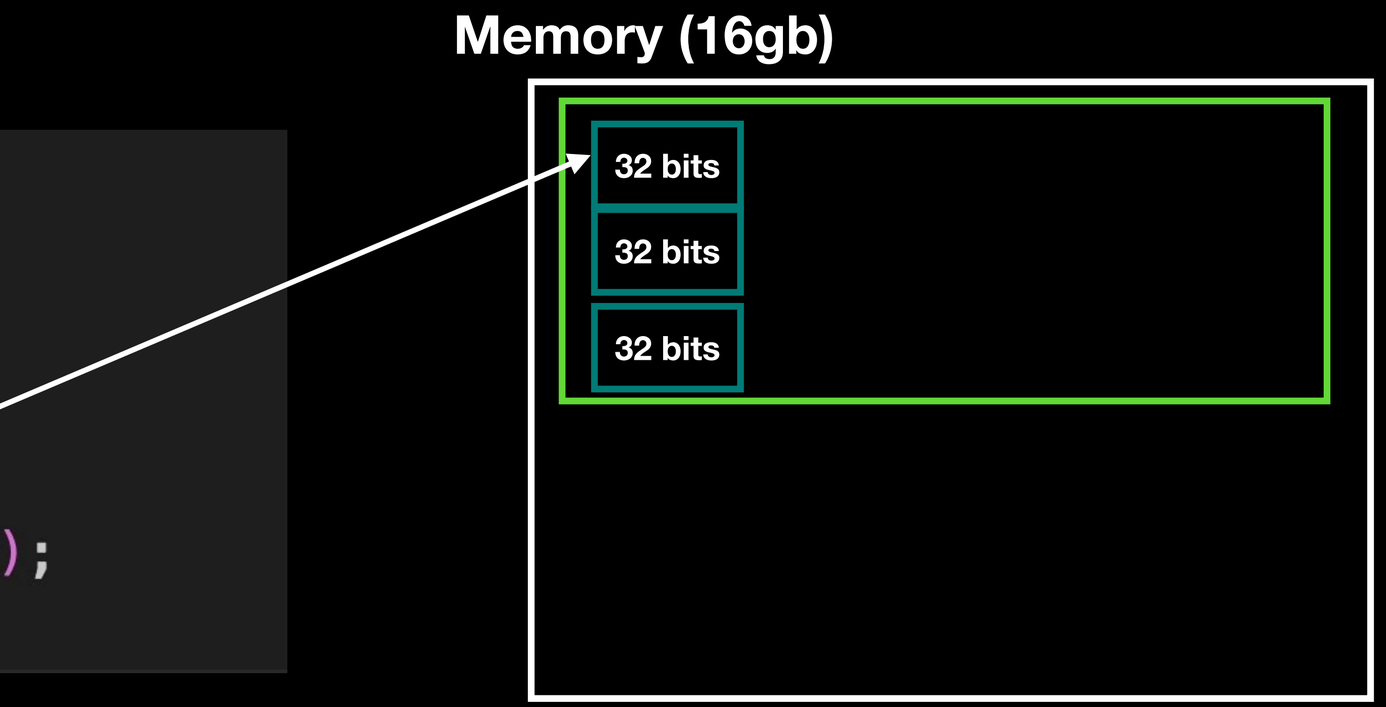


Memory management

```
▶ Run | Debug
fn main() {
    let a: i32 = 1;
    let b: i32 = 2;
    let sum: i32 = a + b;
    println!("Sum is {}", sum);
}
```

Memory management

```
▶ Run | Debug
fn main() {
    let a: i32 = 1;
    let b: i32 = 2;
    let sum: i32 = a + b;
    println!("Sum is {}", sum);
}
```

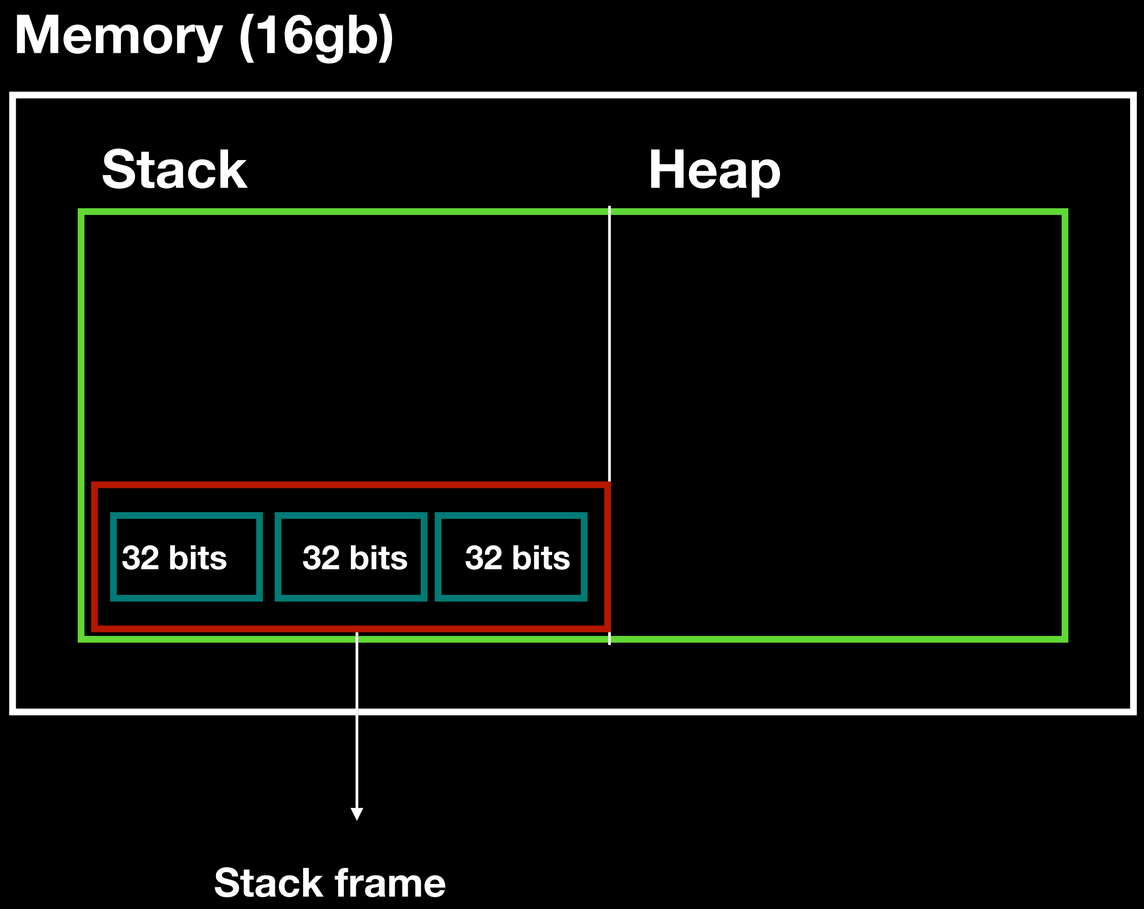


Memory management

```
▶ Run | Debug
fn main() {
    let a: i32 = 1;
    let b: i32 = 2;
    let sum: i32 = a + b;
    println!("Sum is {}", sum);
}
```

Memory management

```
▶ Run | Debug
fn main() {
    let a: i32 = 1;
    let b: i32 = 2;
    let sum: i32 = a + b;
    println!("Sum is {}", sum);
}
```



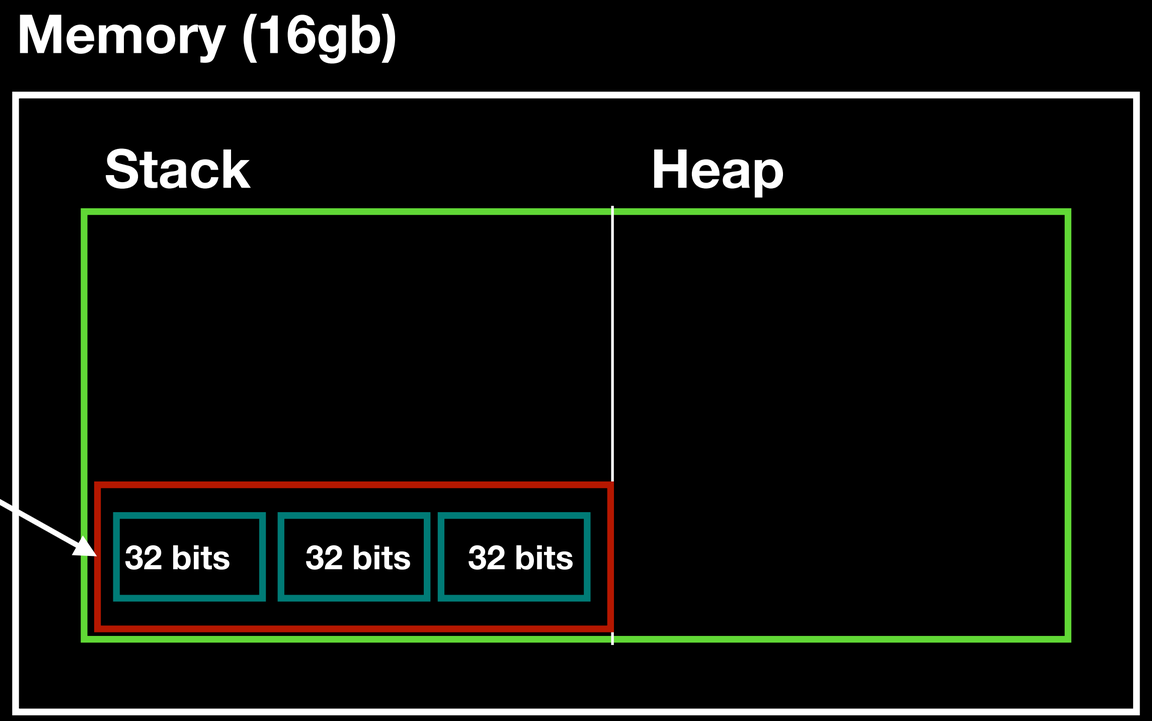
Memory management

```rust
fn main() {
    let a: i32 = 1;
    let b: i32 = 2;
    let sum: i32 = find_sum(a, b);
    println!("Sum is {}", sum);
}

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

What do you think happens here?

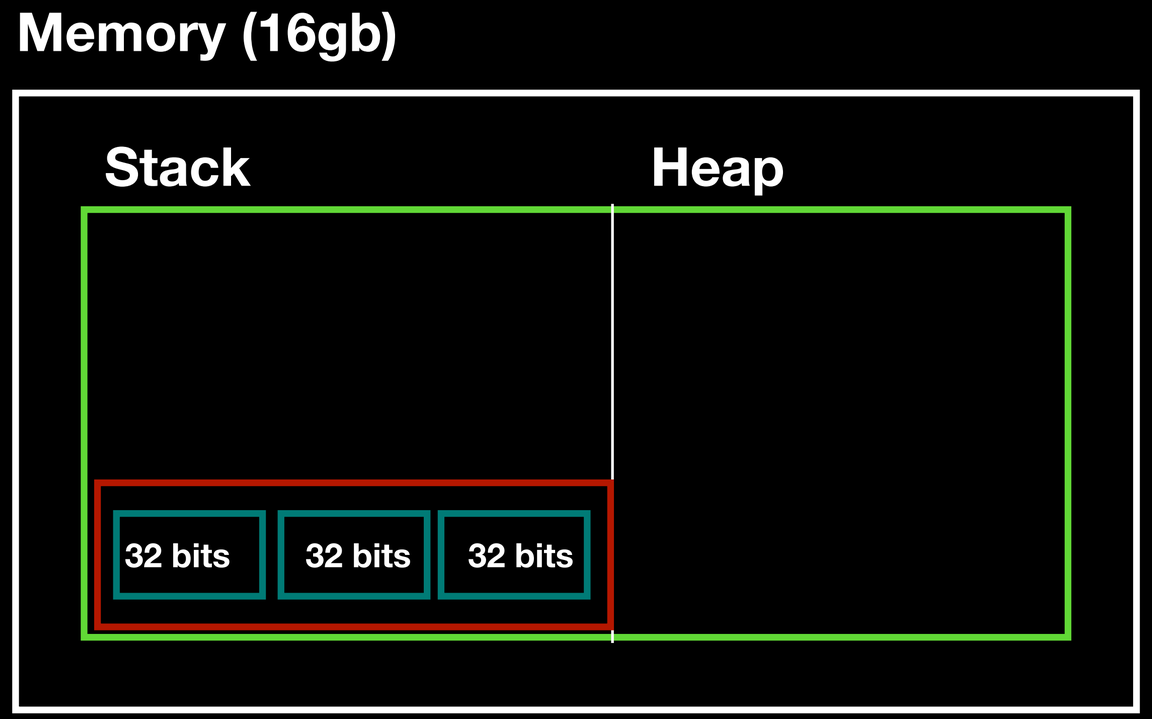
Memory management

```rust
fn main() {
    let a: i32 = 1;
    let b: i32 = 2;
    let sum: i32 = find_sum(a, b);
    println!("Sum is {}", sum);
}

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



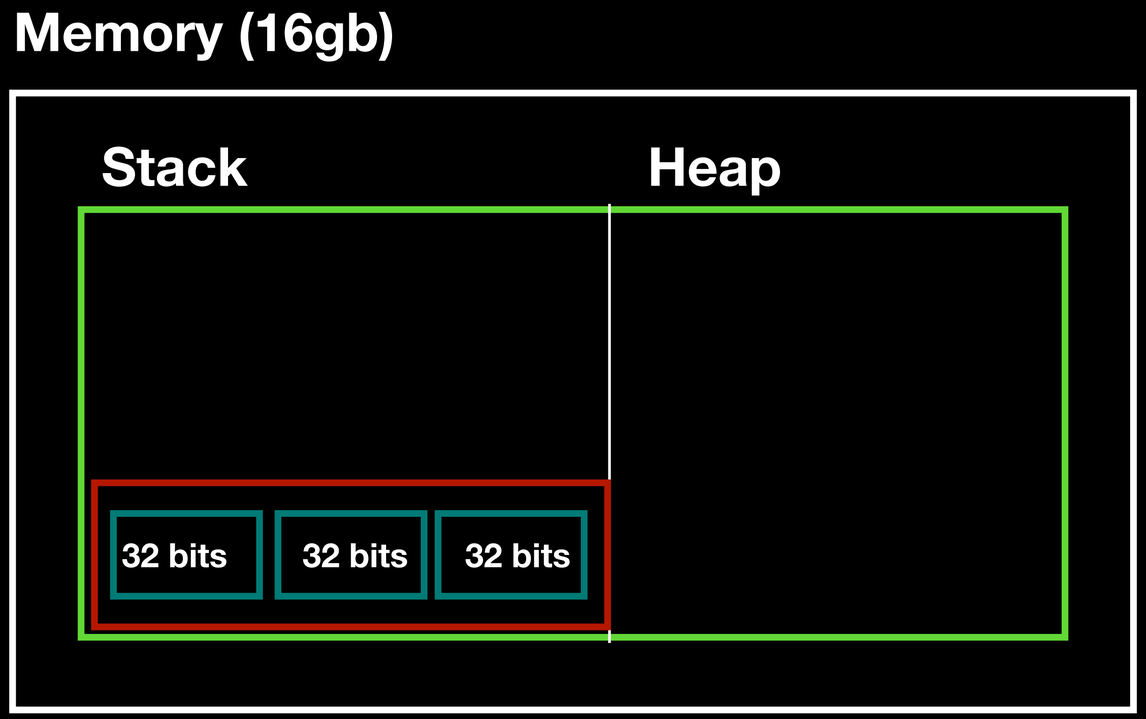
Memory management

```rust
fn main() {
    let a: i32 = 1;
    let b: i32 = 2;
    let sum: i32 = find_sum(a, b);
    println!("Sum is {}", sum);
}

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



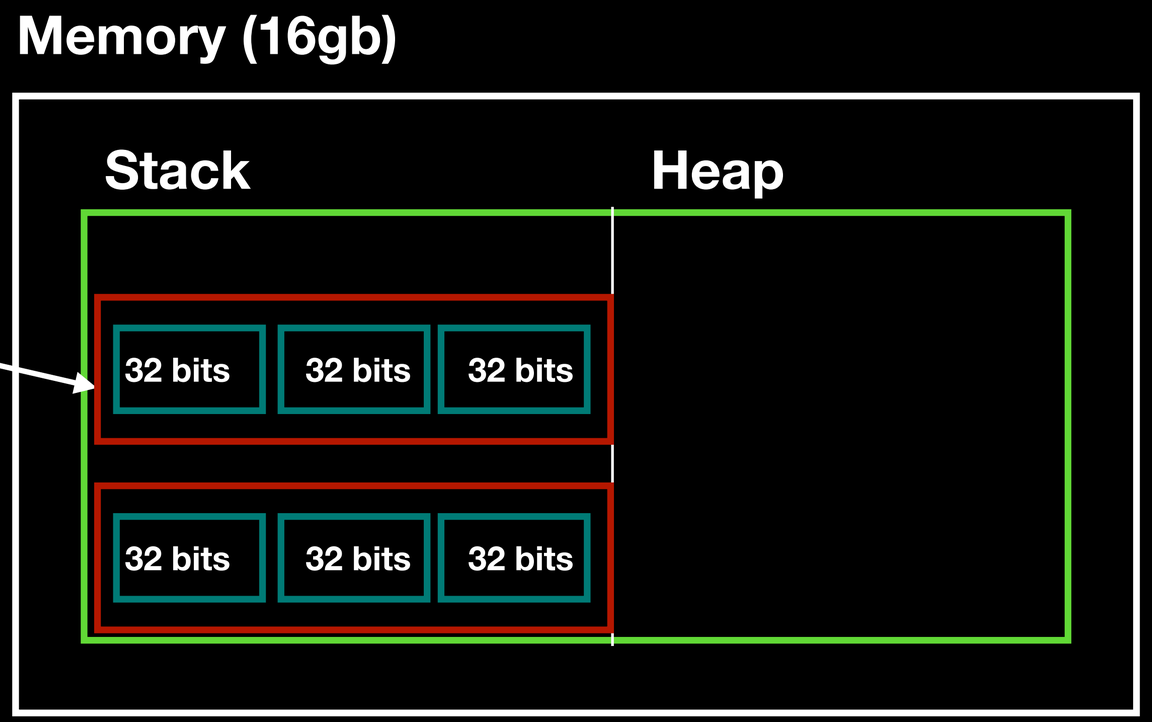
Memory management

```rust
fn main() {
    let a: i32 = 1;
    let b: i32 = 2;
    let sum: i32 = find_sum(a, b);
    println!("Sum is {}", sum);
}

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



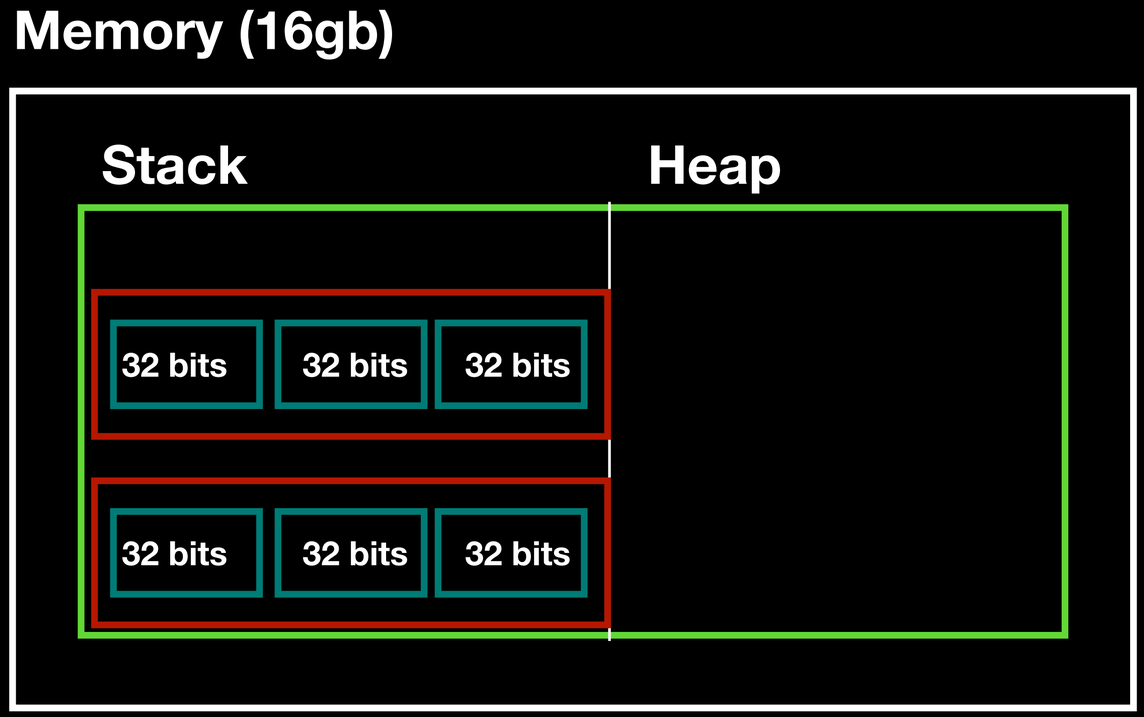
Memory management

```rust
fn main() {
    let a: i32 = 1;
    let b: i32 = 2;
    let sum: i32 = find_sum(a, b);
    println!("Sum is {}", sum);
}

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



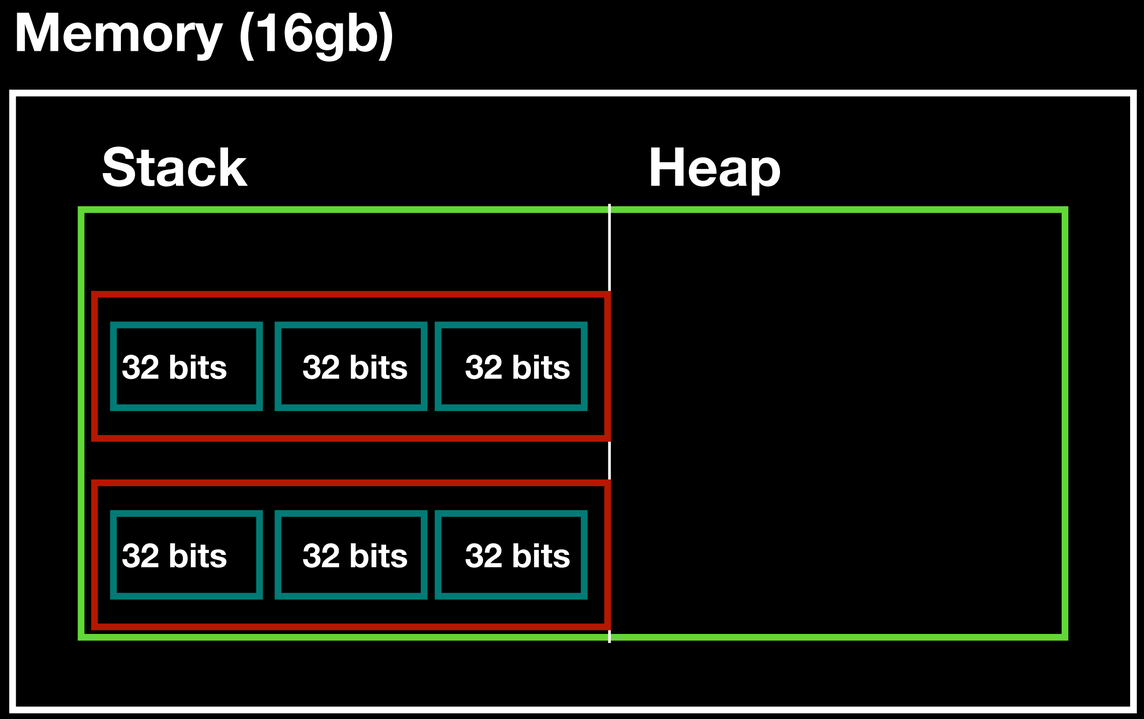
Memory management

```rust
fn main() {
    let a: i32 = 1;
    let b: i32 = 2;
    let sum: i32 = find_sum(a, b);
    println!("Sum is {}", sum);
}

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



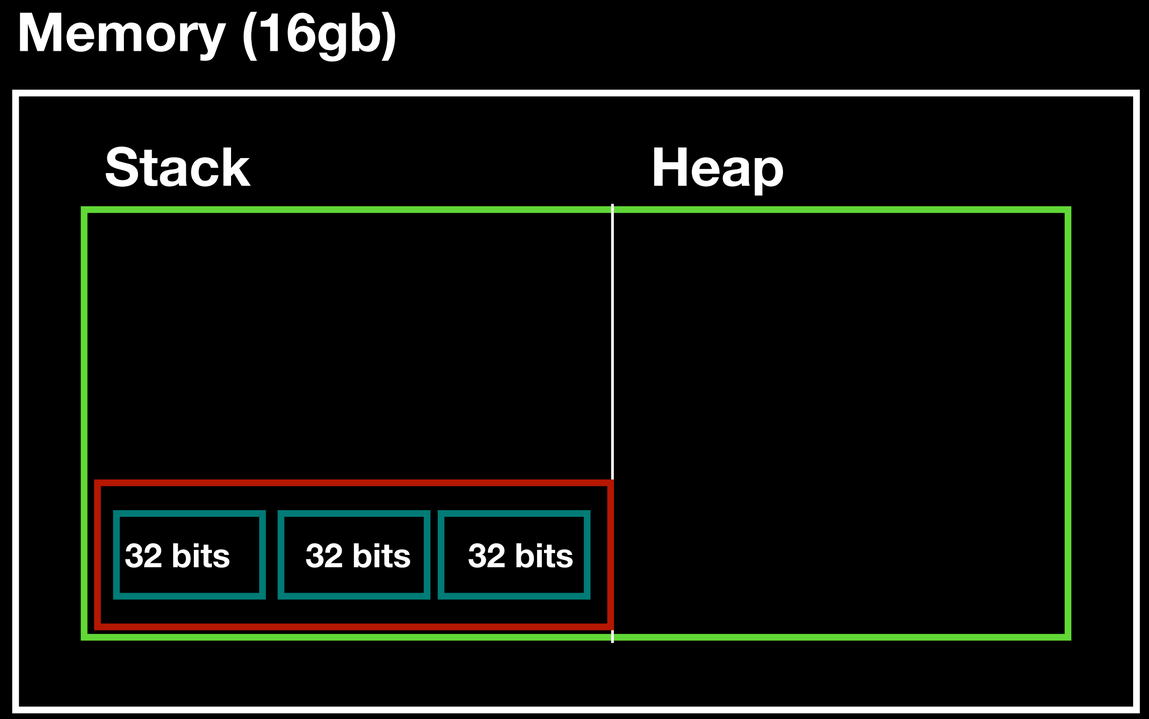
Memory management

```rust
fn main() {
    let a: i32 = 1;
    let b: i32 = 2;
    let sum: i32 = find_sum(a, b);
    println!("Sum is {}", sum);
}

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



Memory management

```rust
fn main() {
    let a: i32 = 1;
    let b: i32 = 2;
    let sum: i32 = find_sum(a, b);
    println!("Sum is {}", sum);
}

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



Memory management

Until now, we've only stored things on the stack
Lets try to store things on the heap next

Memory management

Stack	Heap
Dynamic, allocated at runtime	Static, allocated at compile time
Much larger in size	Smaller in size
Slower due to dynamic allocation and deallocation	Faster
Used for Dynamic and large data structures (e.g., Vec, HashMap, Box)	Used for Small, fixed-size variables and function call information

Memory management

Stored on the stack

1. Numbers - i32, u64, f32
2. Booleans - true, false
3. Fixed sized arrays - [i32; 4]
4. Structs - { x: i32, y: i32 }
5. References (later)

Stored on the heap

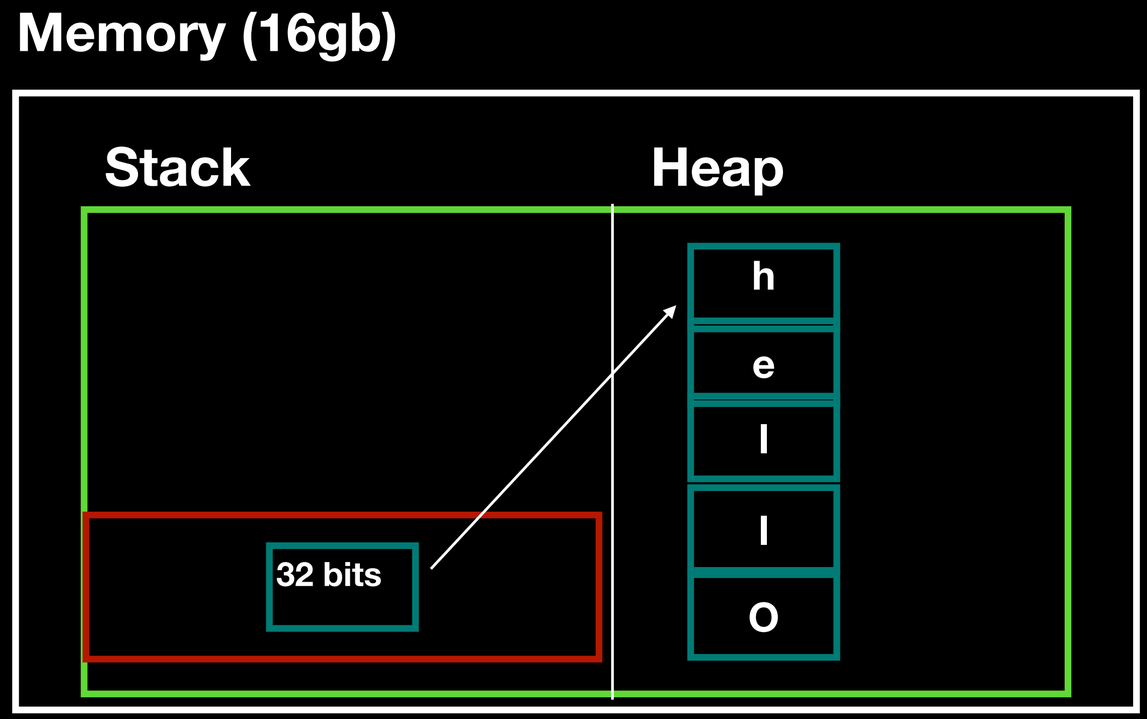
1. Strings
2. Vectors
3. HashMap
4. Large Arrays/Structs that can’t fit in the Stack

Memory management

```rust
fn main() {
    let name: String = String::from("Hello");
    println!("Name is {}", name);
}
```

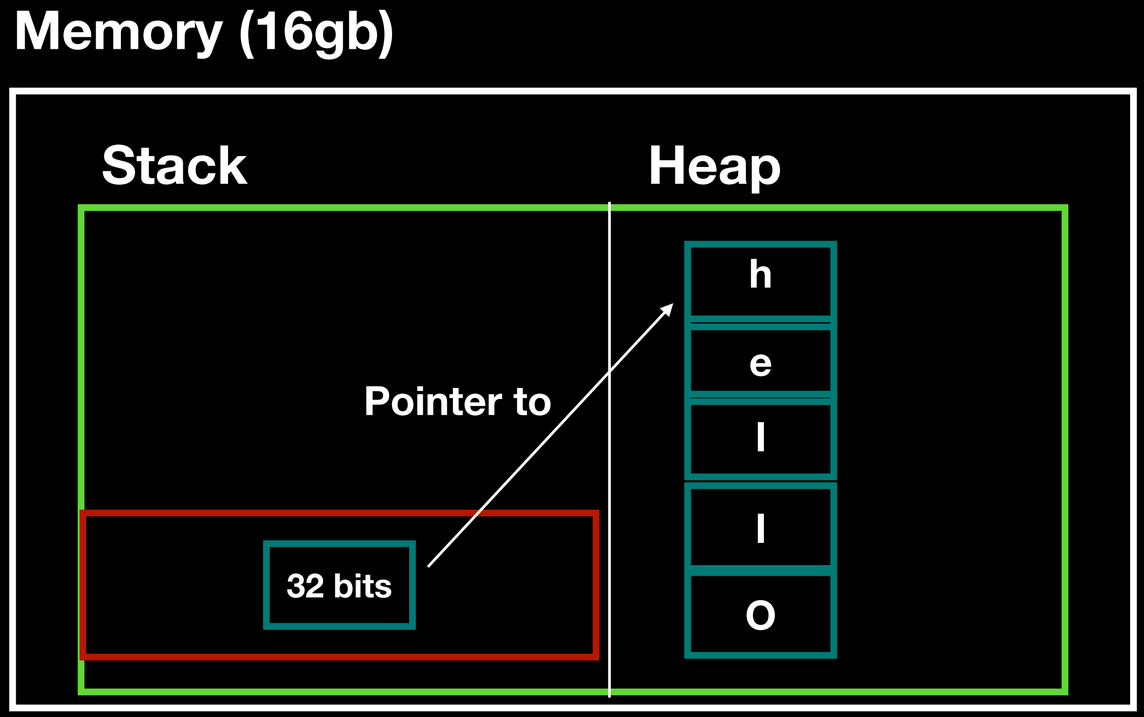
Memory management

```rust
fn main() {
    let name: String = String::from("Hello");
    println!("Name is {}", name);
}
```



Memory management

```rust
fn main() {
    let name: String = String::from("Hello");
    println!("Name is {}", name);
}
```



Why are strings stored on the heap?

1. **They are large**
2. **Their size can change at runtime, and the size of a stack frame needs to be fixed**

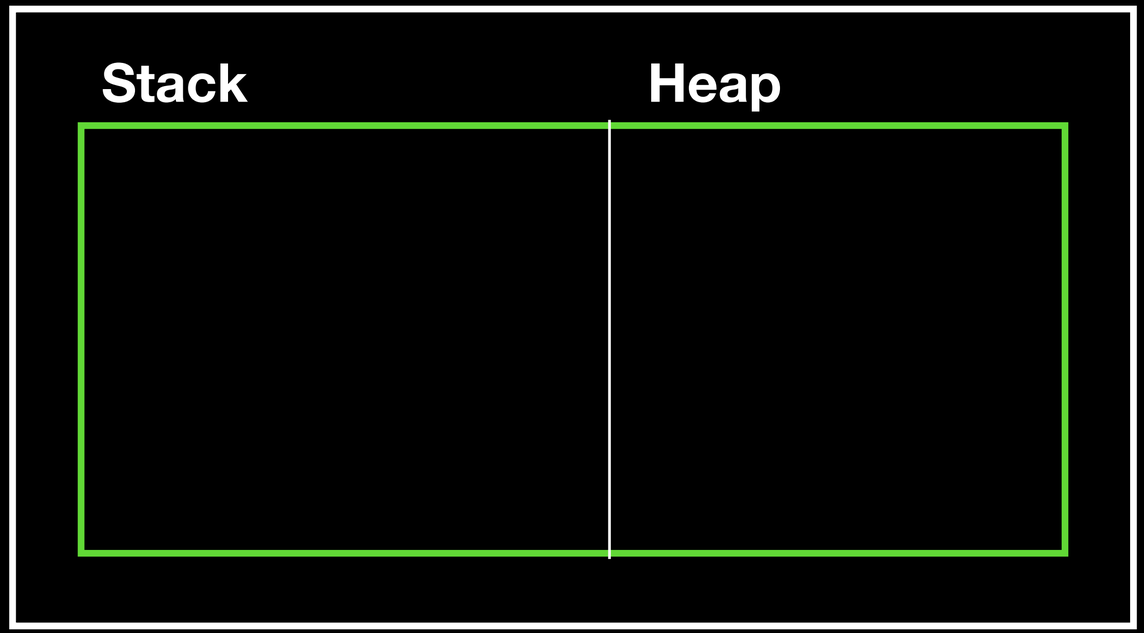
Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References



Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

In rust, all variables are immutable by default
You have to explicitly set them as mut

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

In rust, all variables are immutable by default
You have to explicitly set them as mut

Immutable by default

```
fn main() {
    let name: String = String::from("Hello");
    name.push_str(string: "World");
}
```

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

In rust, all variables are immutable by default
You have to explicitly set them as mut

```
▶ Run | Debug
fn main() {
    let mut name: String = String::from("Hello");
    name.push_str(string: "World");
}
```

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

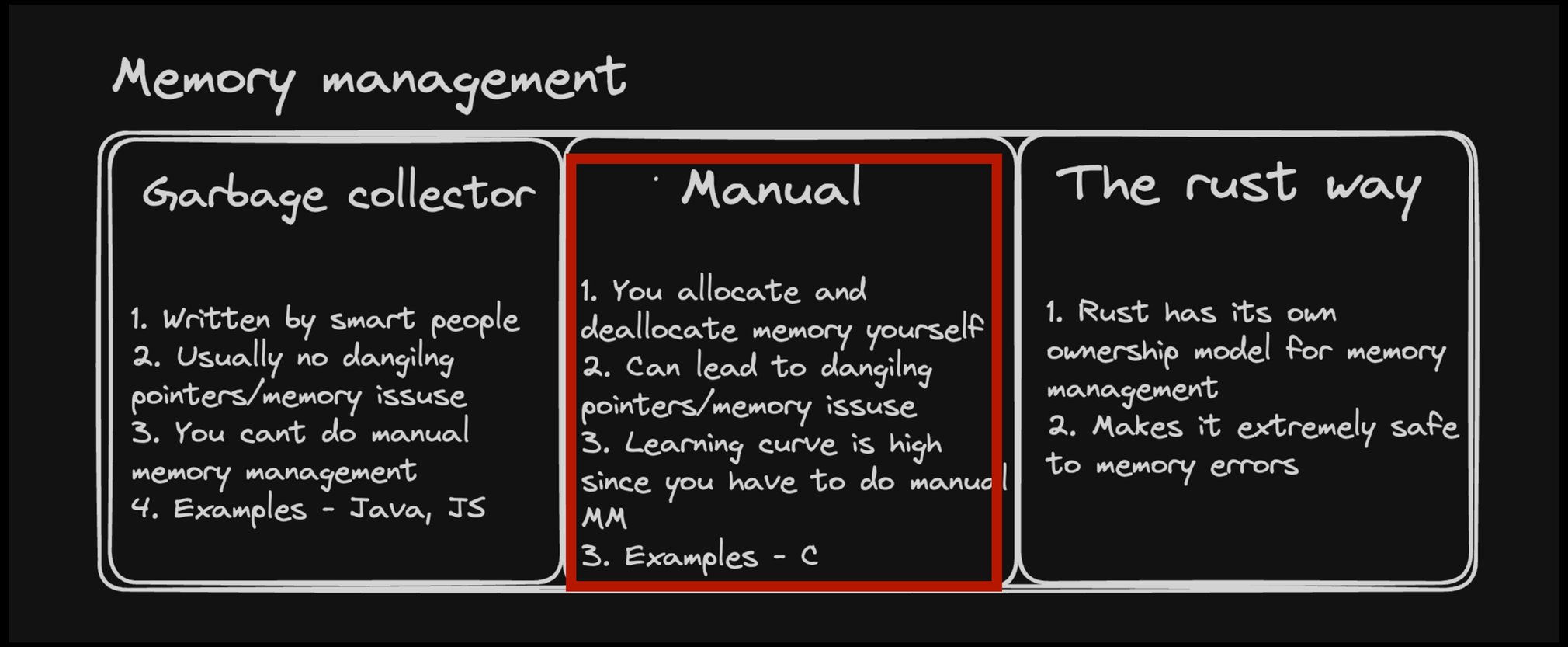
Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

We need to understand 4 jargon

1. Ownership
2. Moving
3. Borrowing
4. References

Three ways of doing Memory management



This C++ code has a memory leak

src > a.cpp

```cpp
#include <iostream>
#include <cstring> // For strcpy

// Function that allocates memory on the heap and prints the string
void createString() {
    // Allocate memory for a string on the heap
    char* str = new char[30];

    // Copy a string into the allocated memory
    strcpy(str, "Hello, heap memory leak!");

    // Print the string
    std::cout << str << std::endl;

    // Free the allocated memory
    // delete[] str;
}

int main() {
    // Call the function
    createString();

    return 0;
}

```

This C++ code has a memory leak

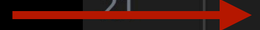
src > a.cpp

```cpp
#include <iostream>
#include <cstring> // For strcpy

// Function that allocates memory on the heap and prints the string
void createString() {
    // Allocate memory for a string on the heap
    char* str = new char[30];

    // Copy a string into the allocated memory
    strcpy(str, "Hello, heap memory leak!");

    // Print the string
    std::cout << str << std::endl;

    // Free the allocated memory
    // delete[] str;
}

int main() {
    // Call the function
    createString();

    return 0;
}
```

This C++ code has a memory leak

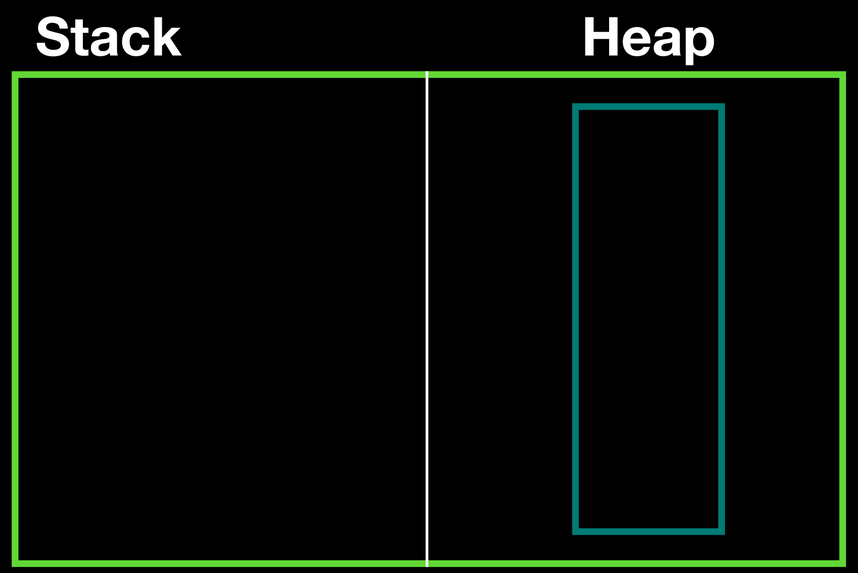
src > a.cpp

```cpp
#include <iostream>
#include <cstring> // For strcpy

// Function that allocates memory on the heap and prints the string
void createString() {
    // Allocate memory for a string on the heap
    char* str = new char[30];

    // Copy a string into the allocated memory
    strcpy(str, "Hello, heap memory leak!");

    // Print the string
    std::cout << str << std::endl;

    // Free the allocated memory
    // delete[] str;
}

int main() {
    // Call the function
    createString();

    return 0;
}
```



This C++ code has a memory leak

Allocates data on the heap

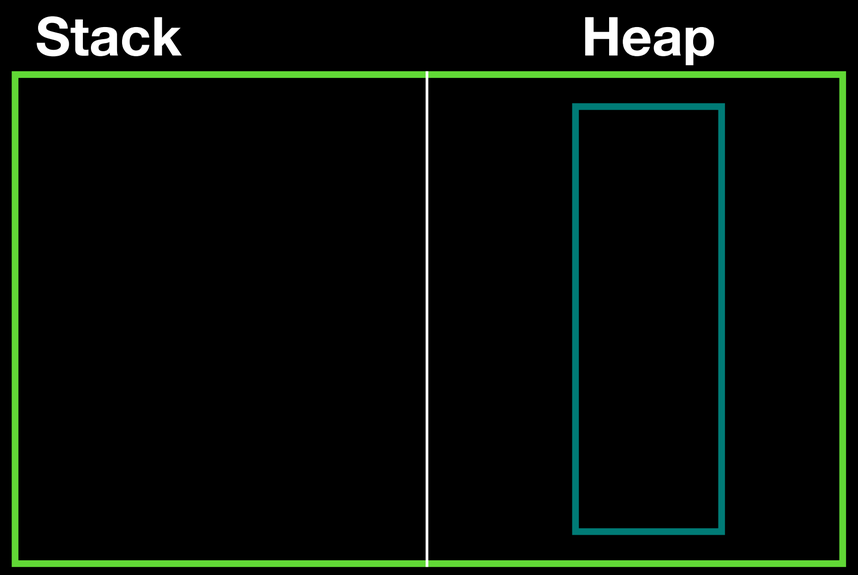


src > a.cpp

```cpp
#include <iostream>
#include <cstring> // For strcpy

// Function that allocates memory on the heap and prints the string
void createString() {
    // Allocate memory for a string on the heap
    char* str = new char[30];

    // Copy a string into the allocated memory
    strcpy(str, "Hello, heap memory leak!");

    // Print the string
    std::cout << str << std::endl;

    // Free the allocated memory
    // delete[] str;
}

int main() {
    // Call the function
    createString();

    return 0;
}

```

This C++ code has a memory leak

Copies over data to the heap

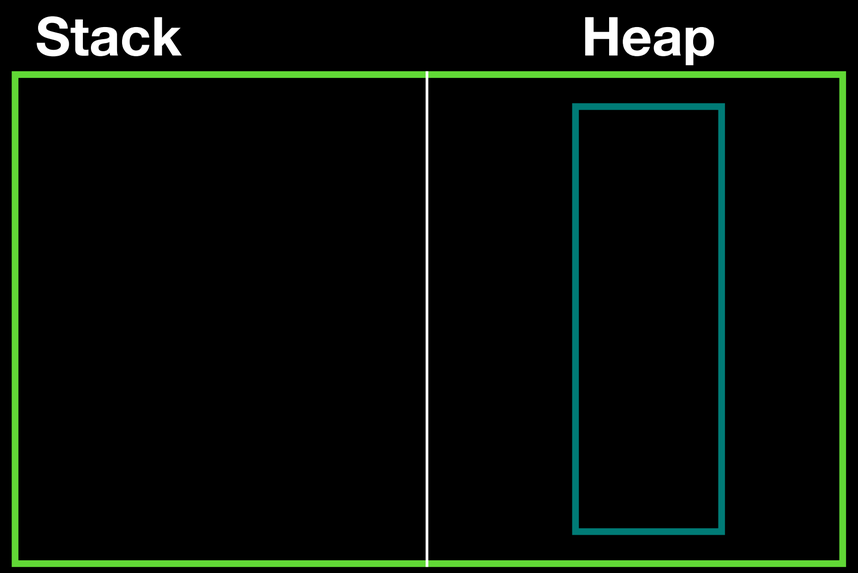


src > a.cpp

```cpp
#include <iostream>
#include <cstring> // For strcpy

// Function that allocates memory on the heap and prints the string
void createString() {
    // Allocate memory for a string on the heap
    char* str = new char[30];

    // Copy a string into the allocated memory
    strcpy(str, "Hello, heap memory leak!");

    // Print the string
    std::cout << str << std::endl;

    // Free the allocated memory
    // delete[] str;
}

int main() {
    // Call the function
    createString();

    return 0;
}

```

This C++ code has a memory leak

Prints the data on screen

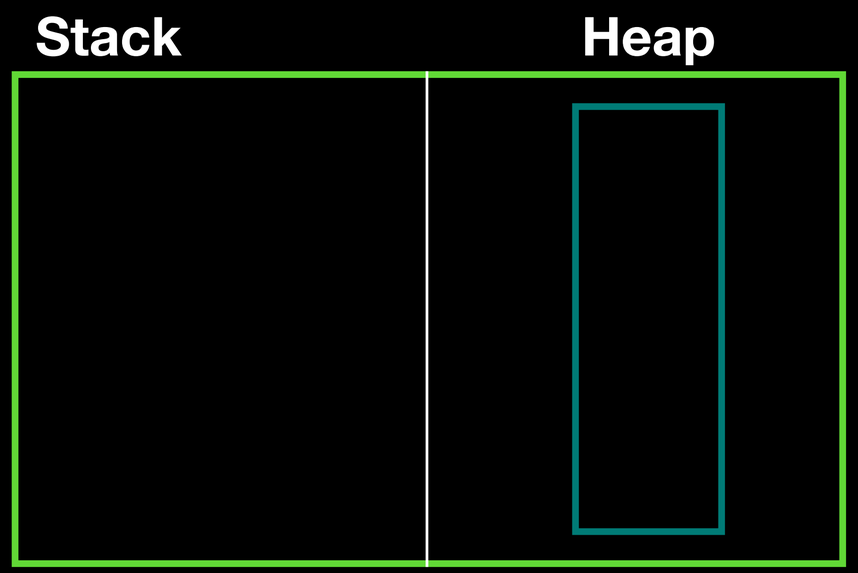


src > a.cpp

```cpp
#include <iostream>
#include <cstring> // For strcpy

// Function that allocates memory on the heap and prints the string
void createString() {
    // Allocate memory for a string on the heap
    char* str = new char[30];

    // Copy a string into the allocated memory
    strcpy(str, "Hello, heap memory leak!");

    // Print the string
    std::cout << str << std::endl;

    // Free the allocated memory
    // delete[] str;
}

int main() {
    // Call the function
    createString();

    return 0;
}

```

This C++ code has a memory leak

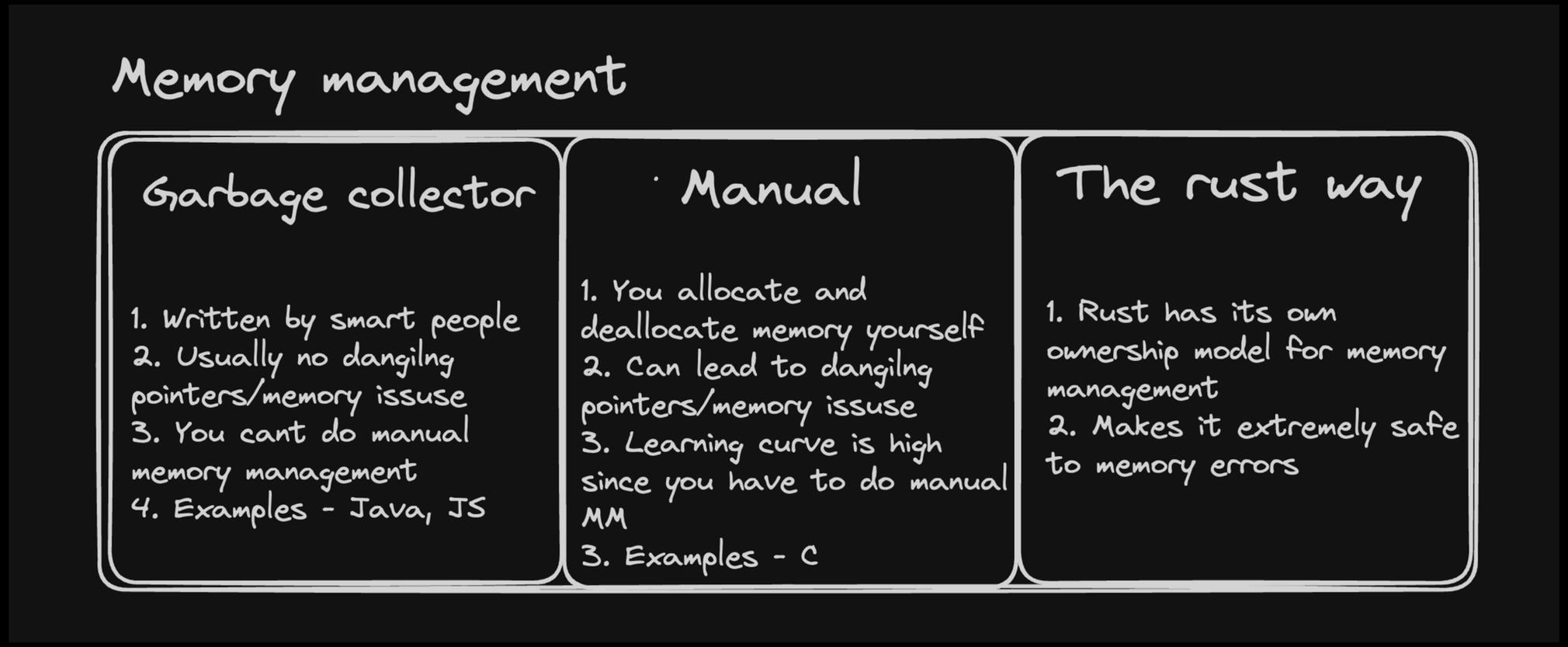
Data is still alive on the heap
Even though there is no reference to the string
(Memory leak)



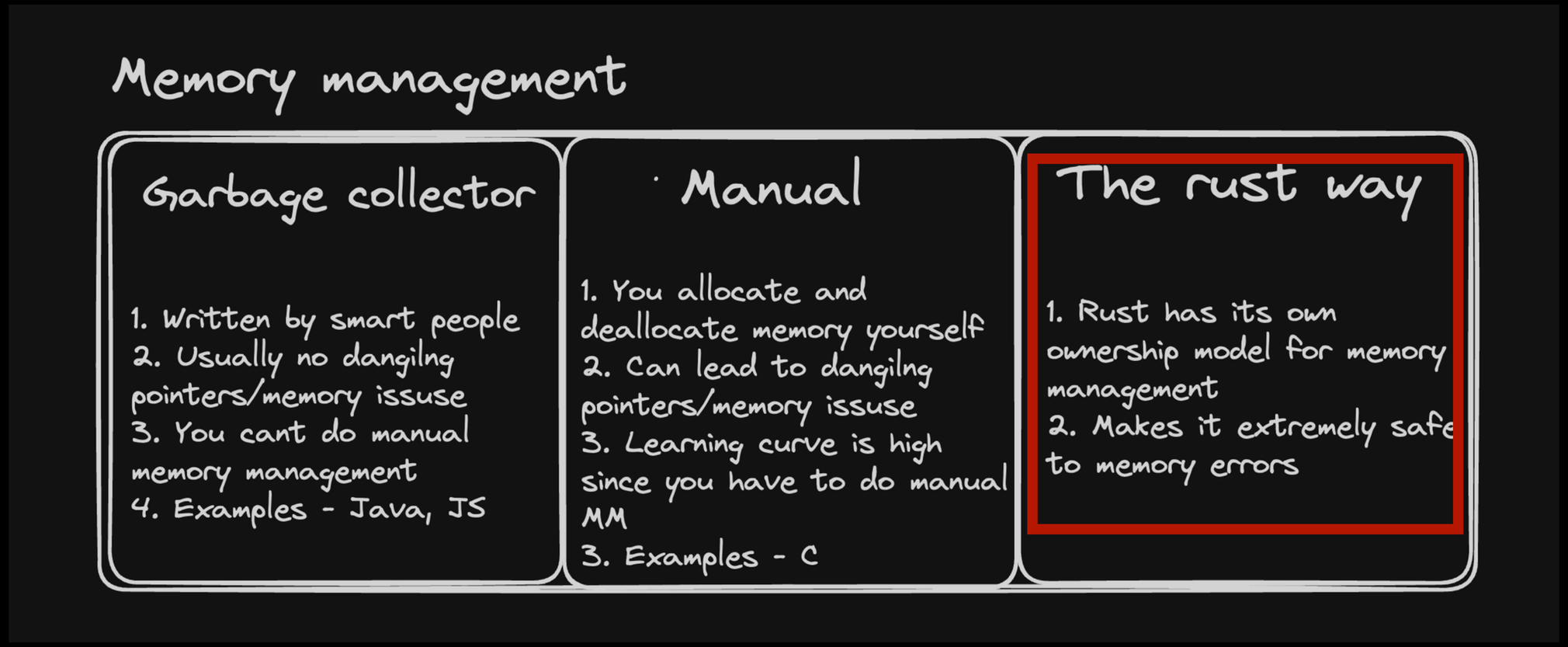
src > a.cpp

```cpp
#include <iostream>
#include <cstring> // For strcpy

// Function that allocates memory on the heap and prints the string
void createString() {
    // Allocate memory for a string on the heap
    char* str = new char[30];

    // Copy a string into the allocated memory
    strcpy(str, "Hello, heap memory leak!");

    // Print the string
    std::cout << str << std::endl;

    // Free the allocated memory
    // delete[] str;
}

int main() {
    // Call the function
    createString();

    return 0;
}

```

Three ways of doing Memory management



Three ways of doing Memory management

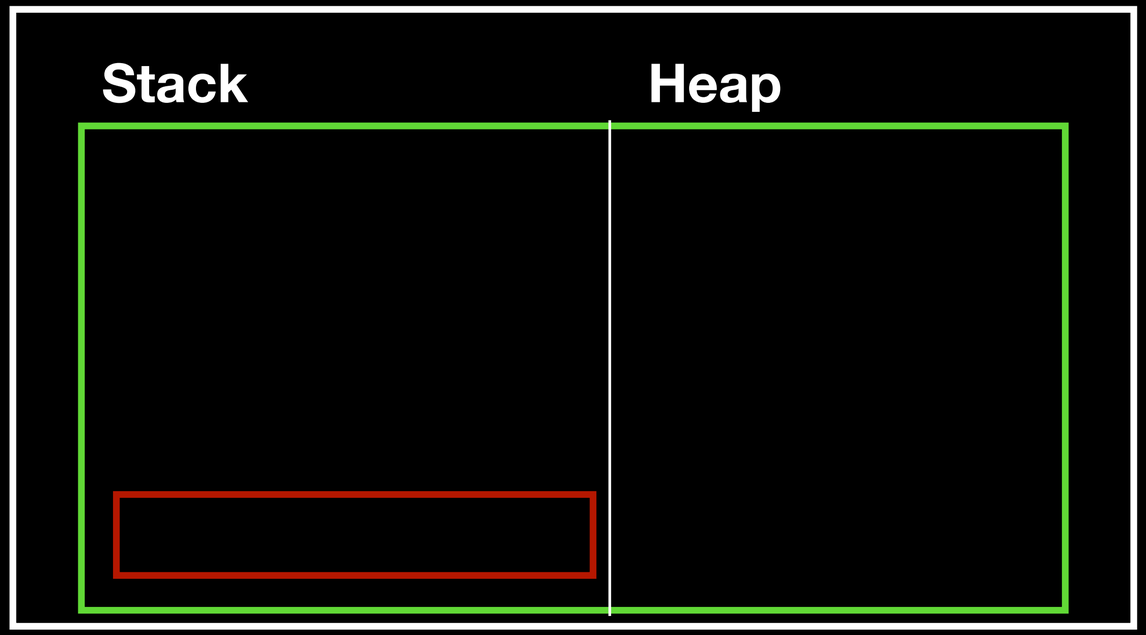


Is there a memory leak?

```rust
fn create_string() {
    // Allocate a String on the heap
    let s: String = String::from("Hello, heap memory leak!");

    // Print the string
    println!("{}", s);
}

▶ Run | Debug
fn main() {
    // Call the function
    create_string();
}
```

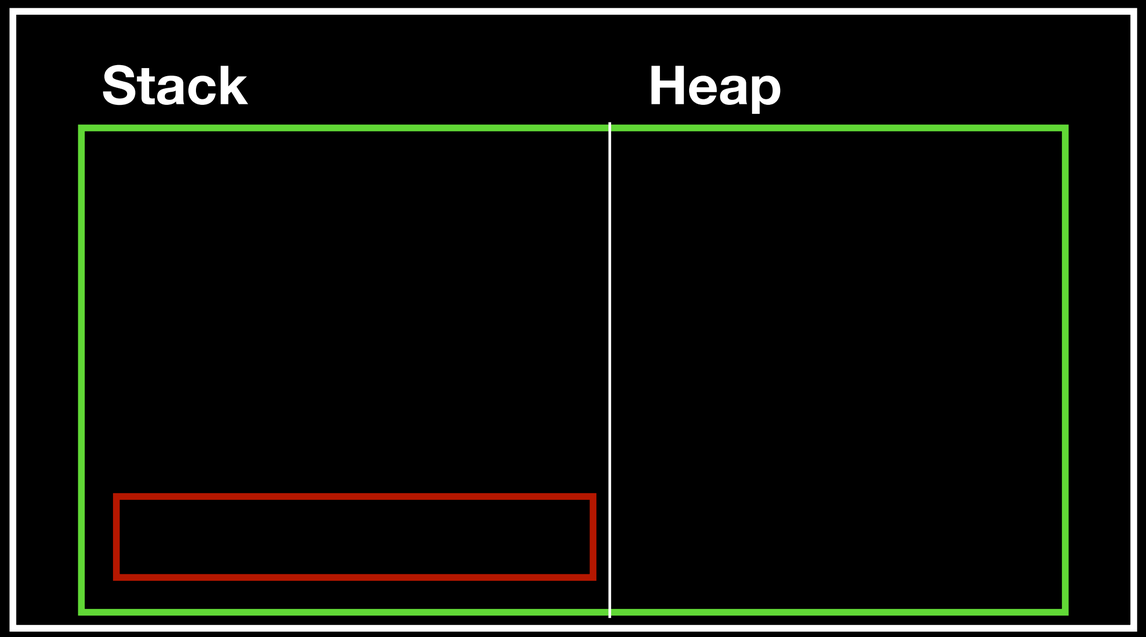
Is there a memory leak?

```rust
fn create_string() {
    // Allocate a String on the heap
    let s: String = String::from("Hello, heap memory leak!");

    // Print the string
    println!("{}", s);
}

▶ Run | Debug
fn main() {
    // Call the function
    create_string();
}
```



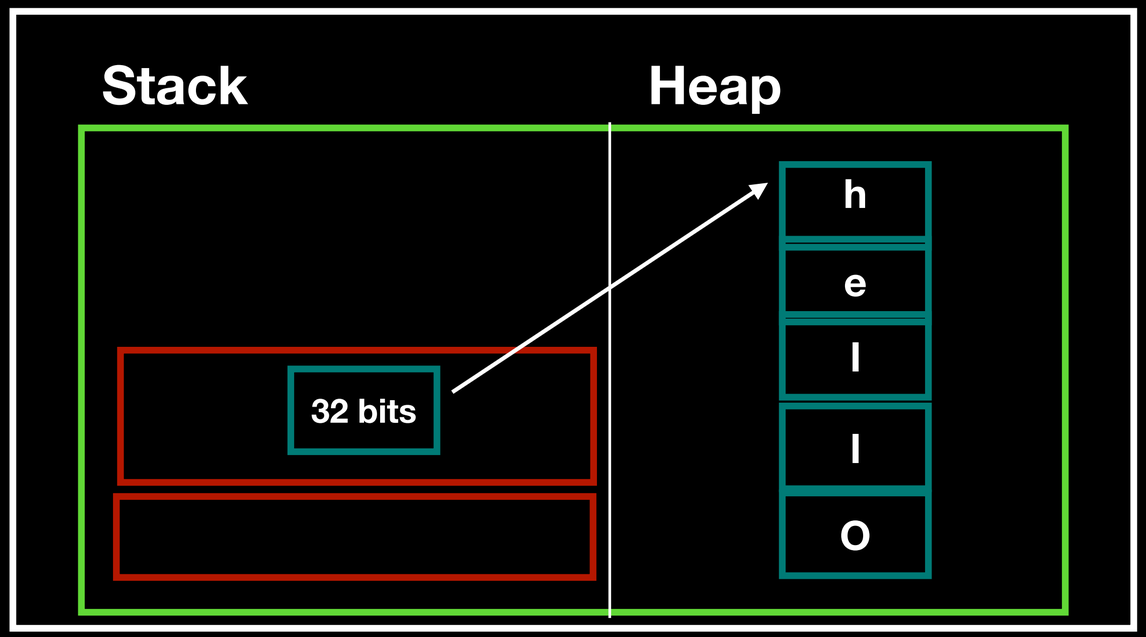
Is there a memory leak?

```rust
fn create_string() {
    // Allocate a String on the heap
    let s: String = String::from("Hello, heap memory leak!");

    // Print the string
    println!("{}", s);
}

▶ Run | Debug
fn main() {
    // Call the function
    create_string();
}
```



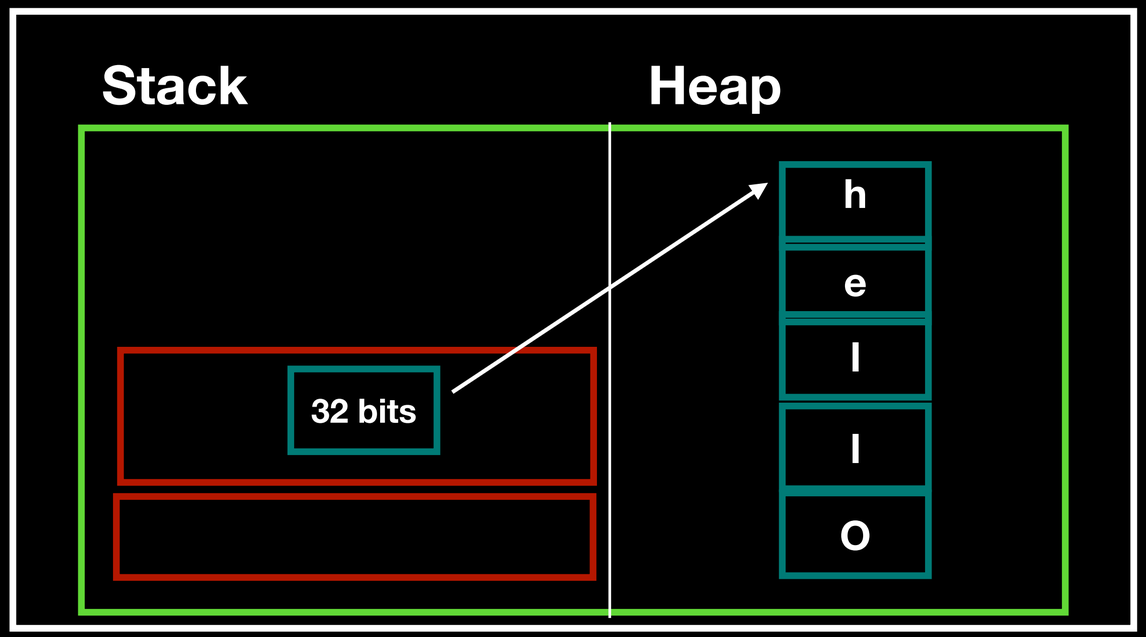
Is there a memory leak?

```rust
fn create_string() {
    // Allocate a String on the heap
    let s: String = String::from("Hello, heap memory leak!");

    // Print the string
    println!("{}", s);
}

▶ Run | Debug
fn main() {
    // Call the function
    create_string();
}
```



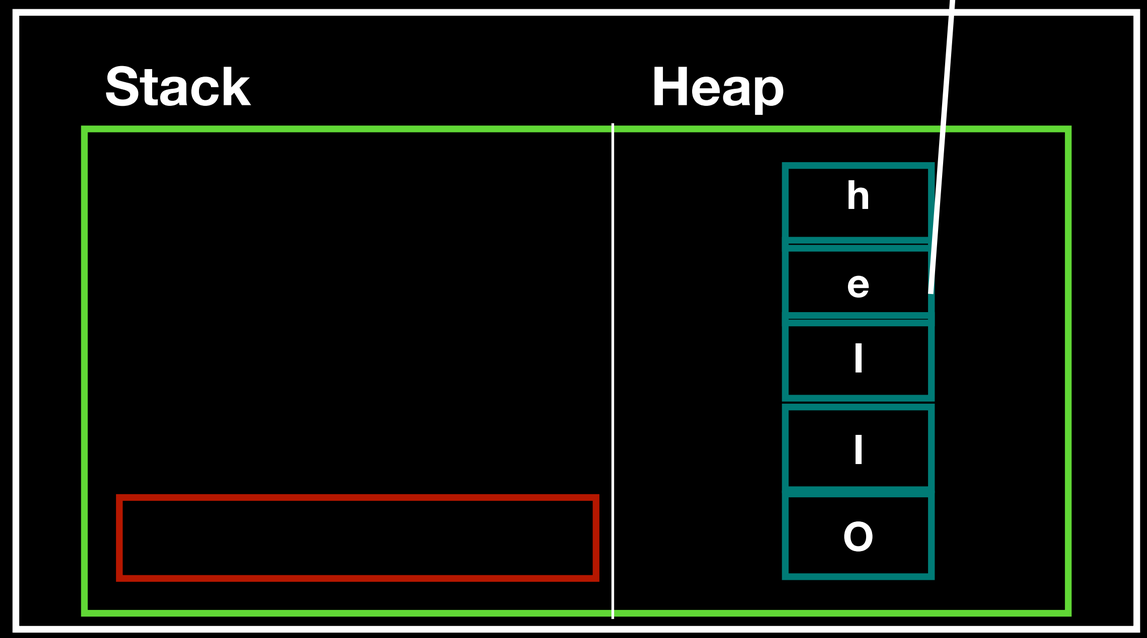
Is there a memory leak?

```rust
fn create_string() {
    // Allocate a String on the heap
    let s: String = String::from("Hello, heap memory leak!");

    // Print the string
    println!("{}", s);
}

▶ Run | Debug
fn main() {
    // Call the function
    create_string();
}
```



Is there a memory leak?

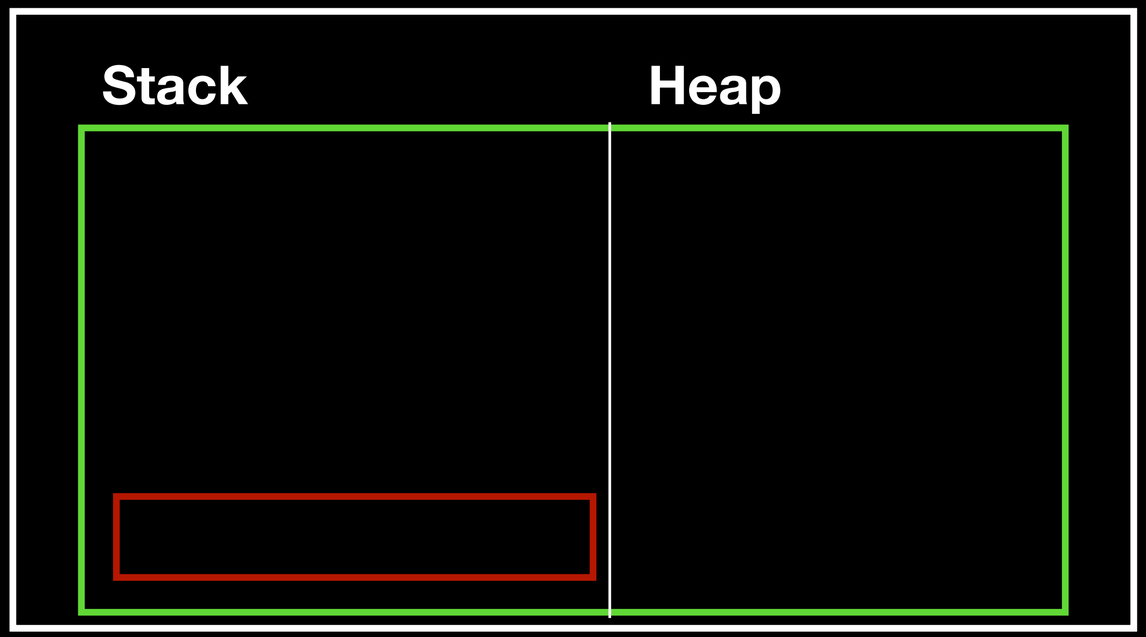
```rust
fn create_string() {
    // Allocate a String on the heap
    let s: String = String::from("Hello, heap memory leak!");

    // Print the string
    println!("{}", s);
}

▶ Run | Debug
fn main() {
    // Call the function
    create_string();
}
```



Is there a memory leak?

Does the heap get cleaned up?
Yes

```rust
fn create_string() {
    // Allocate a String on the heap
    let s: String = String::from("Hello, heap memory leak!");

    // Print the string
    println!("{}", s);
}

▶ Run | Debug
fn main() {
    // Call the function
    create_string();
}
```



Ownership rules

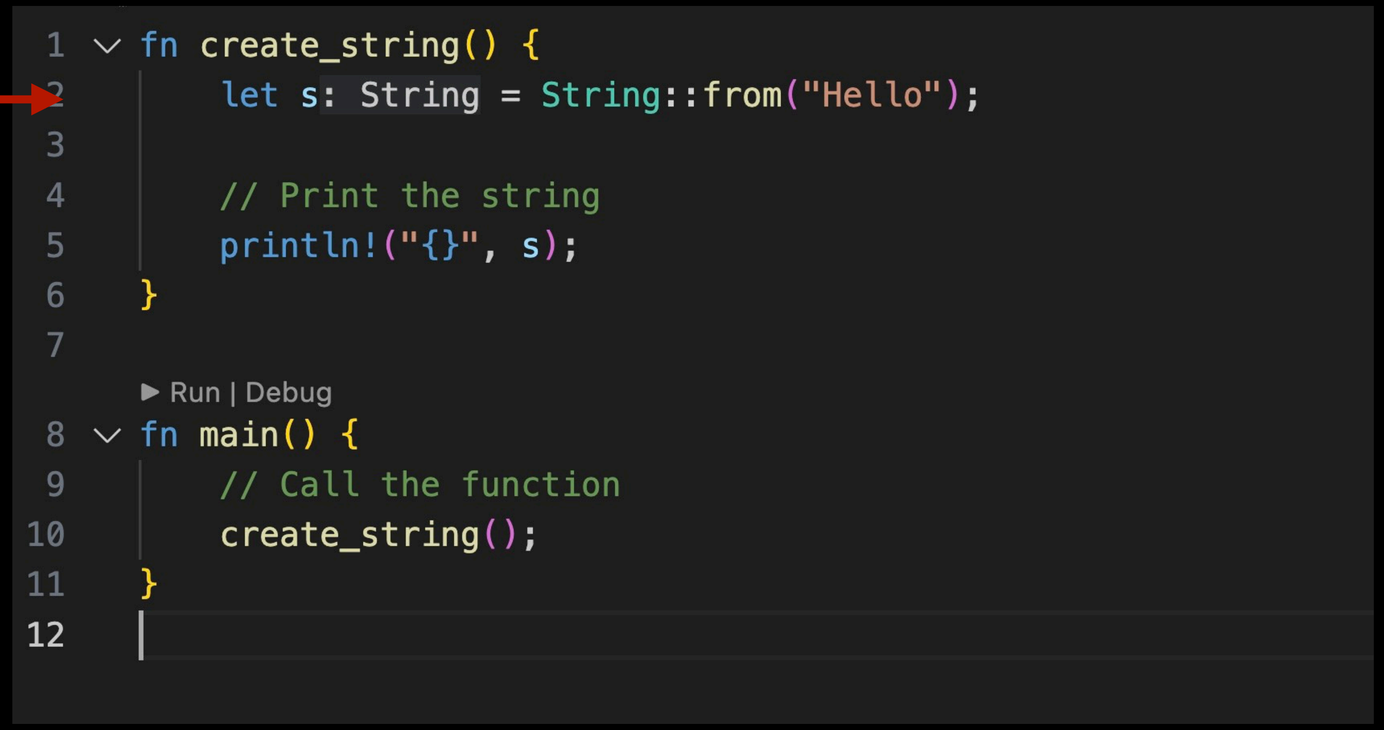
Ownership Rules

First, let's take a look at the ownership rules. Keep these rules in mind as we work through the examples that illustrate them:

- Each value in Rust has an *owner*.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

Ownership

S is the owner of the value “hello”



```rust
fn create_string() {
    let s: String = String::from("Hello");

    // Print the string
    println!("{}", s);
}

▶ Run | Debug
fn main() {
    // Call the function
    create_string();
}
```

Ownership

S goes out of scope,
value gets removed from the heap

```rust
fn create_string() {
    let s: String = String::from("Hello");

    // Print the string
    println!("{}", s);
}

▶ Run | Debug
fn main() {
    // Call the function
    create_string();
}
```

We need to understand 4 jargon

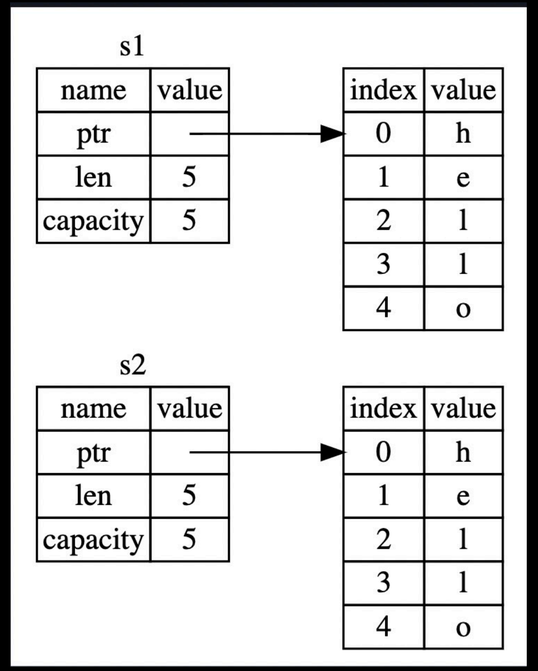
1. Ownership
2. Moving
3. Borrowing
4. References

Move

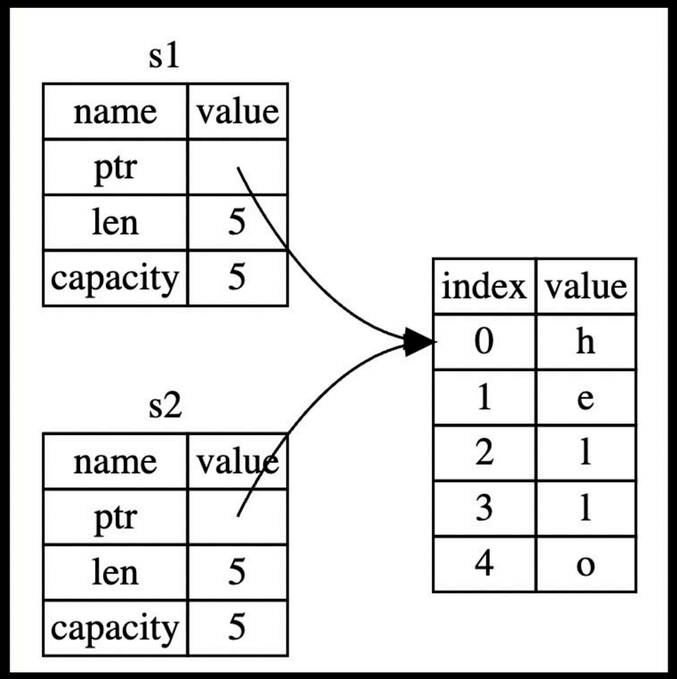
```rust
fn create_string() {
    let s1: String = String::from("Hello");
    let s2: String = s1;
    // Print the string
    println!("{}", s1);
}

▶ Run | Debug
fn main() {
    // Call the function
    create_string();
}
```

Move

```rust
fn create_string() {
    let s1: String = String::from("Hello");
    let s2: String = s1;
    // Print the string
    println!("{}", s1);
}

▶ Run | Debug
fn main() {
    // Call the function
    create_string();
}
```



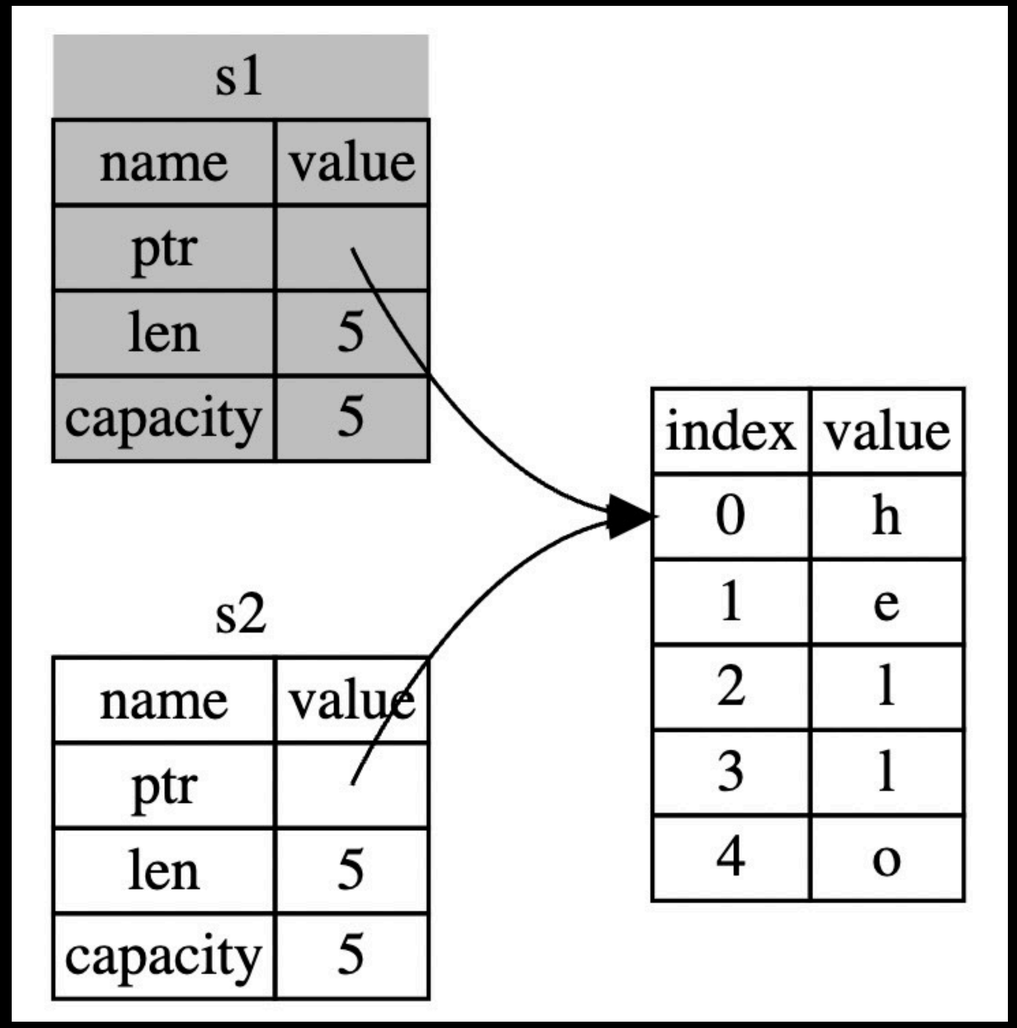
1



2

Move

```rust
fn create_string() {
    let s1: String = String::from("Hello");
    let s2: String = s1;
    // Print the string
    println!("{}", s1);
}

▶ Run | Debug
fn main() {
    // Call the function
    create_string();
}
```



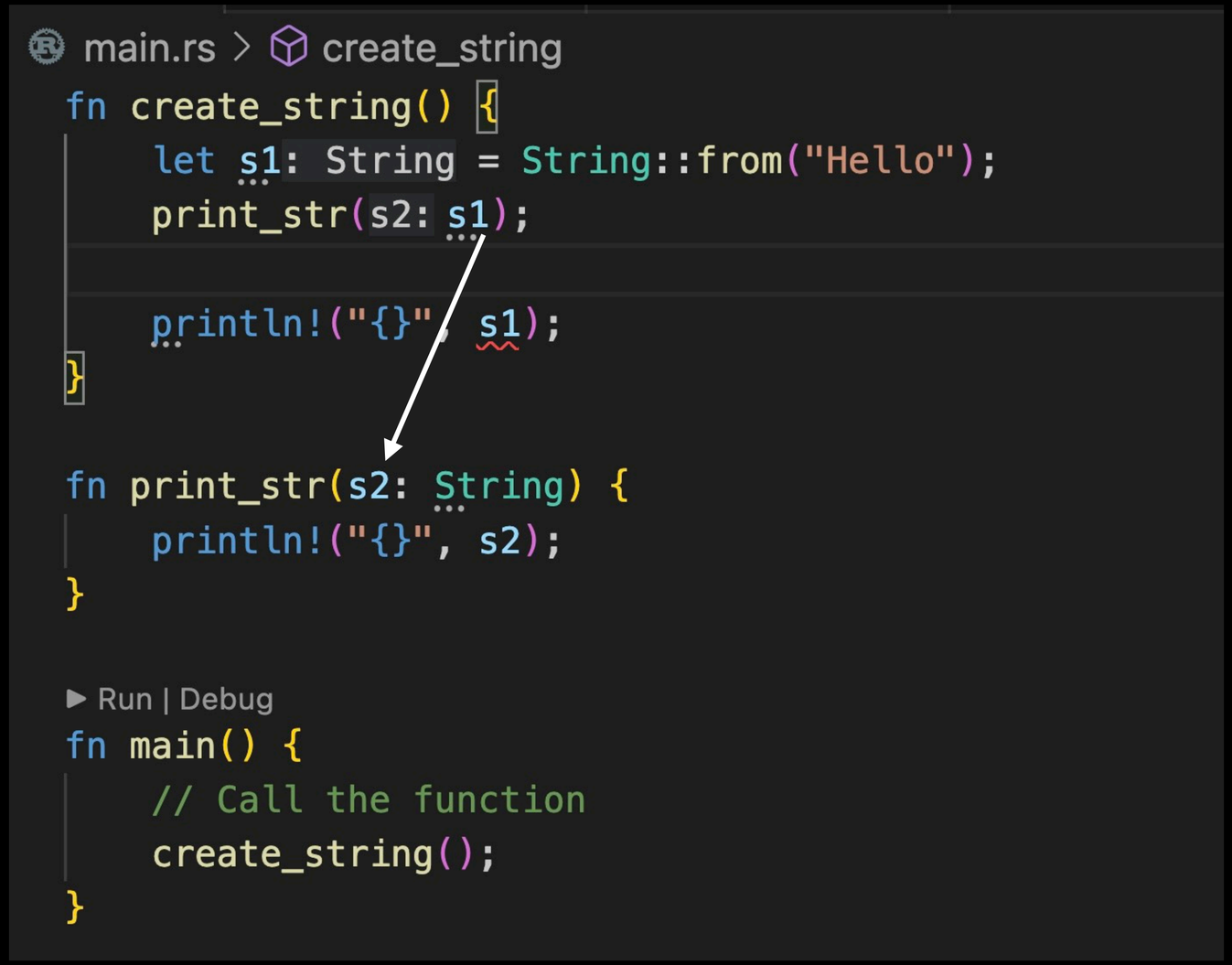
✅

Move

```rust
fn create_string() {
    let s1: String = String::from("Hello");
    let s2: String = s1;
    // Print the string
    println!("{}", s1);
}

▶ Run | Debug
fn main() {
    // Call the function
    create_string();
}
```

```
borrow of moved value: `s1`
value borrowed here after move rustc(Click for full compiler diagnostic)
```

Move


main.rs > create_string
fn create_string() {
let s1: String = String::from("Hello");
print_str(s2: s1);
println!("{}", s1);
}
fn print_str(s2: String) {
println!("{}", s2);
}
▶ Run | Debug
fn main() {
// Call the function
create_string();
}

Move

You can return the value from a function
Back to the function calling it

```rust
fn create_string() {
    let mut s1: String = String::from("Hello");
    s1 = print_str(s2: s1);

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

fn print_str(s2: String) -> String {
    println!("{}", s2);
    return s2;
}

▶ Run | Debug
fn main() {
    // Call the function
    create_string();
}
```

We need to understand 4 jargon

1. Ownership
2. Moving
3. Borrowing
4. References

Borrowing

This code is ugly
(you have to return the value back)

```rust
fn create_string() {
    let mut s1: String = String::from("Hello");
    s1 = print_str(s2: s1);

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

fn print_str(s2: String) -> String {
    println!("{}", s2);
    return s2;
}

▶ Run | Debug
fn main() {
    // Call the function
    create_string();
}
```

Borrowing

We pass in a reference to s1
s1 is borrowed by print_str
s1 is still owned by create_string

```rust
fn create_string() {
    let mut s1: String = String::from("Hello");
    print_str(s2: &s1);

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

fn print_str(s2: &String) {
    println!("{}", s2);
}

▶ Run | Debug
fn main() {
    // Call the function
    create_string();
}
```

Borrowing

No Borrowing - Error ❌

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

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

Borrowing - No error ✅

```
▶ Run | Debug
fn main() {
    let s1: String = String::from("hello");
    let s2: &String = &s1;

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

Borrowing

What if you want the borrow to mutate the value?

Borrowing

What if you want the borrow to mutate the value?

```rust
fn create_string() {
    let mut s1: String = String::from("Hello");
    print_str(s2: &mut s1);

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

fn print_str(s2: &mut String) {
    s2.push_str(string: " World");
}

▶ Run | Debug
fn main() {
    // Call the function
    create_string();
}
```

Borrowing

The Rules of References

Let's recap what we've discussed about references:

- At any given time, you can have *either* one mutable reference *or* any number of immutable references.
- References must always be valid.

Borrowing

```rust
fn print_str(s2: &mut String) {
    s2.push_str(string: " World");
}

▶ Run | Debug
fn main() {
    let mut s1: String = String::from("Hello");
    let mut s2: &mut String = &mut s1;
    print_str(s2: &mut s1);
    println!("{}", s2);
}
```

Define variables s1

Borrowing

```rust
fn print_str(s2: &mut String) {
    s2.push_str(string: " World");
}

▶ Run | Debug
fn main() {
    let mut s1: String = String::from("Hello");
    let mut s2: &mut String = &mut s1;
    print_str(s2: &mut s1);
    println!("{}", s2);
}
```

Take mutable reference #1

Borrowing

```rust
fn print_str(s2: &mut String) {
    s2.push_str(string: " World");
}

▶ Run | Debug
fn main() {
    let mut s1: String = String::from("Hello");
    let mut s2: &mut String = &mut s1;
    print_str(s2: &mut s1);
    println!("{}", s2);
}
```

Take mutable reference #2

Borrowing

```rust
fn print_str(s2: &mut String) {
    s2.push_str(string: " World");
}

▶ Run | Debug
fn main() {
    let mut s1: String = String::from("Hello");
    let mut s2: &mut String = &mut s1;
    print_str(s2: &mut s1);
    println!("{}", s2);
}
```

Use mutable reference #1

Borrowing

```
error[E0499]: cannot borrow `s1` as mutable more than once at a time
 --> src/main.rs:8:15
  |
7 |     let mut s2 = &mut s1;
  |                  ------- first mutable borrow occurs here
8 |     print_str(&mut s1);
  |               ^^^^^^^ second mutable borrow occurs here
9 |     println!("{}", s2);
  |                    -- first borrow later used here
```

We need to understand 4 jargon

1. Ownership
2. Moving
3. Borrowing
4. References

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Recapping Part 1

Easy

1. Installing rust
2. IDE Setup
3. Initializing a project locally
4. Variables (nums, strings and bools)
5. Conditionals, loops
6. Functions
7. Structs
8. Enums
9. Options/Result
10. Pattern Matching
11. Package Management

Hard

1. Memory management
2. Mutability
3. heap vs stack
4. Ownership
5. Borrowing
6. References

Lets get into Part #2

1. Collections, vectors
2. Iterators
3. Hashmaps
4. Strings, &str and slices
3. Generics
4. Traits
5. Multithreading
6. Macros
8. Futures
9. Async/await and tokio
10. Lifetimes