



# Rust Fundamentals

Bosch Cybersecurity University



# Topics

- Basics of Rust
- Data types
- Functions
- Strings
- Ownership
- Traits
- Memory Management
- Control Flow
- Error Handling
- Concurrency
- Async and Await



# Basics of Rust





# Getting Started

1. How to install rust.

<https://www.rust-lang.org/learn/get-started>

2. Rust tools.

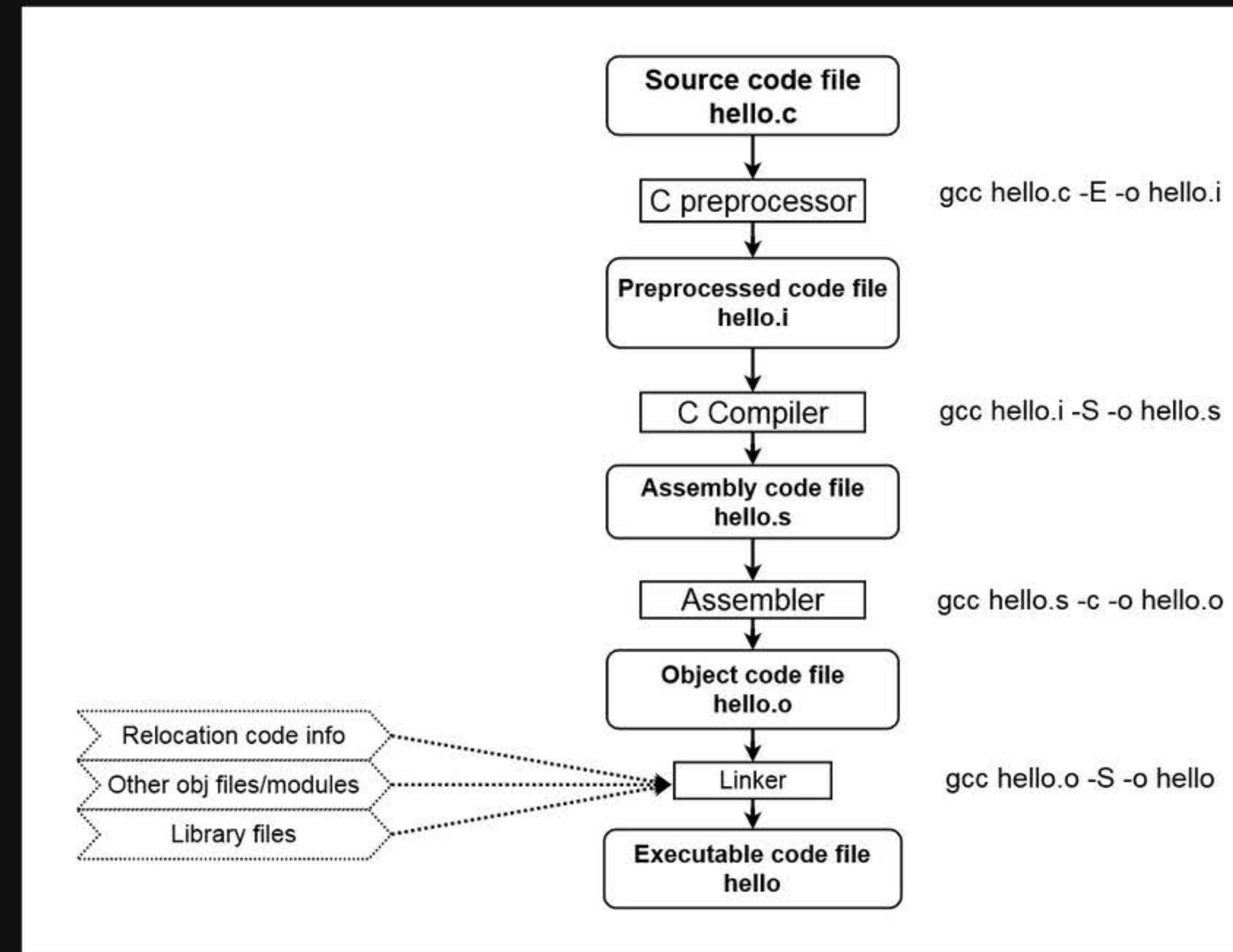
- rustup
- cargo
- rustc
- crates.io

3. Rust Project Structure





# Compilation Process in C/C++



Makefile



# Compilation Process in Rust

A screenshot of a terminal window titled "northwood@NORTHWOOD: ~/test\_project\$". The terminal shows the following sequence of commands and output:

```
cargo build --release
Compiling test_project v0.1.0 (/home/northwood/test_project)
  Finished release [optimized] target(s) in 1.08s
tree
Cargo.lock
Cargo.toml
src
└── main.rs
target
└── CACHEDIR.TAG
    └── release
        └── build
            └── deps
                └── test_project-74a751915a4c2880
                    └── test_project-74a751915a4c2880.d
            └── examples
            └── incremental
            └── test_project
                └── test_project.d
7 directories, 8 files
./target/release/test_project
Hello, world!
```

TOML file





# Data Types





# Variables and Mutability

Variables are immutable by default

```
1 fn main() {  
2     let x = 5;  
3     println!("The value of x is: {x}");  
4     x = 6;  
5     println!("The value of x is: {x}");  
6 }
```





# Variables and Mutability

Error message displayed:

```
1 $ cargo run
2 Compiling variables v0.1.0 (file:///projects/variables)
3 error[E0384]: cannot assign twice to immutable variable `x`
4 --> src/main.rs:4:5
5 2   |     let x = 5;
6 3   |
7 4   |     - first assignment to `x`
8 5   |     |
9 6   |     help: consider making this binding mutable: `mut x`
10 7   | println!("The value of x is: {}", x);
11 8   |     x = 6;
12 9   |     ^^^^^ cannot assign twice to immutable variable
13 10  | For more information about this error, try `rustc --explain E0384` .
14 11  error: could not compile `variables` due to previous error
```





# Mutability

To fix the error add `mut` keyword before variable name.

```
1 fn main() {  
2     let mut x = 5;  
3     println!("The value of x is: {}", x);  
4     x = 6;  
5     println!("The value of x is: {}", x);  
6 }
```

Output:

```
1 $ cargo run  
2 Compiling variables v0.1.0 (file:///projects/variables)  
3 Finished dev [unoptimized + debuginfo] target(s) in 0.30s  
4 Running `target/debug/variables`  
5 The value of x is: 5  
6 The value of x is: 6
```





# Constants

- Constants are values that are bound to a name and are not allowed to change
- `const` keyword is used to declare constants instead of the `let` keyword, and the type of the value must be annotated
- `mut` is not used with `const`





# Constants

- Constants can be declared in any scope, including the global scope
- Constants may be set only to a constant expression, not the result of a value that could only be computed at runtime

```
1 struct Circle {  
2     radius: f64,  
3 }  
4  
5 impl Circle {  
6     const PI: f64 = 3.14159;  
7  
8     fn new(radius: f64) -> Circle {  
9         Circle { radius }  
10    }  
11    fn calculate_area(&self) -> f64 {  
12        Circle::PI * self.radius * self.radius  
13    }  
14 }
```





# Shadowing

- You can declare a new variable with the same name as a previous variable
- The first variable is shadowed by the second

```
1 fn main() {  
2     let x = 5;  
3     let x = x + 1;  
4  
5     {  
6         let x = x * 2;  
7         println!("The value of x in the inner scope is: {x}");  
8     }  
9  
10    println!("The value of x is: {x}");  
11 }
```





# Data Types

- Data type subsets: *scalar* and *compound*
- Rust is a **statically typed** language, which means that it **must know the types** of all variables at **compile time**
- Data Structures in Rust



# Scalar Types

A scalar type represents a single value

- Integers
- Floating-point numbers
- Booleans
- Characters



# Integer Types

An integer is a number without a fractional component

## Number literals Example

Decimal	98_222
Hex	0xff
Octal	0o77
Binary	0b1111_0000
Byte (u8 only)	b'A'



# Floating-Point Types

- Numbers with decimal points
- Rust's floating-point types are `f32` and `f64`

```
1 fn main() {  
2     let x = 2.0;          // f64 (default)  
3     let y: f32 = 3.0;    // f32  
4 }
```





# Numeric Operations

```
1 fn main() {  
2  
3     // addition  
4     let sum = 5 + 10;  
5  
6     // subtraction  
7     let difference = 95.5 - 4.3;  
8  
9     // multiplication  
10    let product = 4 * 30;  
11  
12    // division  
13    let quotient = 56.7 / 32.2;  
14    let truncated = -5 / 3;          // Results in -1  
15  
16    // remainder  
17    let remainder = 43 % 5;  
18 }
```





# The Boolean Type

- Rust has two possible values: `true` and `false`
- Booleans are one byte in size
- It is specified using `'bool'`

```
1 fn main() {  
2  
3     let t = true;  
4  
5     let f: bool = false; // with explicit type annotation  
6 }
```





# The Character Type

- Rust's `char` type is the language's most primitive alphabetic type
- Specify `char` literals with single quotes ''
- Rust's `char` type is four bytes in size and represents a unicode Scalar Value, which means it can represent a lot more than just ASCII

```
1 fn main() {  
2     let c = 'z';  
3     let z: char = 'Z';          // with explicit type annotation  
4     let heart_eyed_cat = '😻'; //emoji  
5 }
```



# Compound Types

- Compound types can group multiple values into one type
- Rust has two primitive compound types: **tuples** and **arrays**





# The Tuple Type

- A tuple is a general way of grouping together a different types into one compound type
  - Fixed length
  - Known at Compile time
  - Heterogeneous

```
1 fn main() {  
2     // creating tuple  
3     let tup: (i32, f64, u8) = (500, 6.4, 1);  
4     let gfg: (&str, &str, &str) = ("Apple", "For", "Bananas");  
5  
6     // accessing tuple data using positional argument  
7     println!("{} {} {}", gfg.0, gfg.1, gfg.2);  
8  
9     // creating another tuple  
10    let article = ("abc", "xyz", 14,12,2020);  
11    let (a,b,c,d,e) = article;  
12  
13    // accessing tuple using variables  
14    println!("This written by {} at {} on {}/{}/{}/{}", b,a,c,d,e);  
15 }
```





# Destructuring Tuple

```
1 fn main() {  
2  
3     let tup = (500, 6.4, 1);  
4  
5     let (x, y, z) = tup;  
6  
7     println!("The value of y is: {y}");  
8 }
```





# The Array Type

- Collection of multiple values is with an array
- Every element of an array must have the same type
- Arrays in Rust have a fixed length

```
1 fn main() {  
2     let a = [1, 2, 3, 4, 5];  
3 }
```

## Accessing Array Elements

```
1 fn main() {  
2     let a = [1, 2, 3, 4, 5];  
3     let first = a[0];  
4     let second = a[1];  
5 }
```





# Compound Types

## Exercise-1

(clue: all the array elements should be of same type)

```
1 fn main() {  
2  
3     // Fix the error  
4     let _arr = [1, 2, '3'];  
5  
6     println!("Success!");  
7 }
```



# Compound Types

## Exercise-2

(clue: indexing should start from 0)

```
1 fn main() {  
2     let arr = ['a', 'b', 'c'];  
3     let ele = arr[1]; // Only modify this line to make the code work!  
4     assert!(ele == 'a');  
5     println!("Success!");  
6 }
```





# Structure

- Structs hold multiple related values
- In a struct you'll name each piece of data so it's clear what the values mean
- Order of the data to specify or access the values of an instance need not be same





# Defining a Struct

- The `struct` keyword is used to declare a structure followed by the name
- Inside curly brackets, we define the names and types of the pieces of data, which we call 'fields'

```
1 struct User {  
2     active: bool,  
3     username: String,  
4     email: String,  
5     sign_in_count: u64,  
6 }  
7  
8 fn main() {}
```





# Instantiating a Struct

Create instance by stating the name of the `struct` and then add curly brackets {} containing **'key:value'** pairs

- keys are the names of the fields
- values are the data stored in those fields

```
1 fn main() {  
2     let user1 = User {  
3         active : true,  
4         username: String::from("someusername123"),  
5         email   : String::from("someone@example.com"),  
6         count   : 1,  
7     };  
8 }
```





# Enums

- The `enum` keyword allows the creation of a type which may be one of a few different variants.
- Any variant which is valid as a struct is also valid as an enum

```
1 enum Work {
2     Civilian,
3     Soldier,
4 }
5 fn main() {
6     use crate::Work::{Civilian, Soldier};
7     // Equivalent to `Work::Civilian`.
8     let work = Civilian;
9     match work {
10         // Note the lack of scoping because of the explicit `use` above
11         Civilian => println!("Civilians work!"),
12         Soldier   => println!("Soldiers fight!"),
13     }
14 }
```





# Functions





# What we are covering

- Define a Function in Rust
- Calling a Function in Rust
- Function Parameters
- Function with Return Value
- Memory Management
- Generic Functions
- Associated Functions
- Method that Access Data
- Generic Types
- Closures





# Define a Function in Rust

Function is a reusable block of code, in Rust `fn` keyword is used to define a function.

```
1 //define a function
2 fn function_name(arguments) -> return_val {
3     // code
4 }
```

Example:

```
1 // greet function
2 fn greet() {
3     println!("Hello world!");
4 }
5
6 // main function
7 fn main() {
8 }
```



# Calling a Function in Rust

Function name followed by parentheses "()" is used to call a function

```
1 fn greet() {           <-----  
2  
3     println!("Hello world!"); | Execution flow is  
4 }                           | transferred to  
5  
6 fn main() {  
7     greet(); -----^ called function  
8 }
```



# Function Parameters

```
1 fn multiply(num1: f64, num2: i64) {  
2  
3     let result = num1 * num2 as f64;  
4  
5     println!("Result: {}", result);  
6 }
```

In this example function **multiply** accepts two parameters:

1. 'num1' - which has type f64 i.e. float 64
2. 'num2' - which has type i64 i.e. integer 64



# Function with Return Value

- `->` syntax is used to specify the return type
- possible to return a without explicitly using the `return` keyword
- the last expression is returned as the result of function

```
1 fn return_nothing() {  
2     // code  
3 }  
4  
5 fn return_bool() -> bool {  
6     // code  
7     return true;  
8 }  
9  
10 fn multiply(float: f64, integer: i64) -> f64 {  
11     let result = float * integer as f64;  
12     result  
13 }
```





# Generic Functions

Generic parameters allow a function, struct, or enum to work with multiple types, instead of just one specific type. You can specify generic parameters in a function signature by enclosing them in angle brackets (< >) after the function name.

Example using `PartialOrd` trait:

```
1 fn min< T:PartialOrd > (a: T, b: T) -> T {
2   if a < b {
3     a
4   } else {
5     b
6   }
7 }
8
9 fn main(){
10   let x = min(5, 10);           // integer
11   let y = min(5.0, 10.0);      // float
12   let z = min("hello", "world"); // string
13   println!("x: {x}");
14   println!("y: {y}");
15   println!("z: {z}");
16 }
```





# Associated Functions

- function that belongs to a struct or enum, rather than an instance of that struct or enum
- Associated functions are called using the name of the struct or enum, followed by the double colon (::) operator

Example of an associated function `new` that acts as a constructor

```
1 struct Point {  
2     x: i32,  
3     y: i32,  
4 }  
5  
6 impl Point {  
7     fn new(x: i32, y: i32) -> Point {  
8         Point { x, y }  
9     }  
10 }  
11 fn main(){  
12     let p = Point::new(1, 2);  
13 }
```





# Methods that access data

- 'new': a constructor that creates a new instance of the Rectangle struct.
- 'area': a method that calculates the area of the Rectangle and uses `&self` to access the data

```
1 struct Rectangle {  
2     width: i32,  
3     height: i32,  
4 }  
5  
6 impl Rectangle {  
7     fn new(width: i32, height: i32) -> Rectangle {  
8         Rectangle { width, height }  
9     }  
10    fn area(&self) -> i32 {  
11        self.width * self.height  
12    }  
13 }
```





# Closures

- Closure is an anonymous function that can capture values from the enclosing environment and defined using the `||` characters.
  - Optional body delimitation `{ }` for a single expression (mandatory otherwise).

```
1 let print_text = ||| println!("Hello, World!");  
2 ~~~ ~~~~~~  
3 ^ ^  
4 | |  
5 start of closure Body of closure
```



# Closures

## Examples:

```
1 fn call_function< F: Fn(i32) -> i32>(f: F, x: i32) -> i32 {
2     f(x)
3 }
4
5 fn return_function() -> Box < dyn Fn(i32) -> i32> {
6     Box::new(|x: i32| -> i32 { x * x })
7 }
8
9 fn main(){
10    // Closure
11    let square = |x: i32| -> i32 { x * x };
12    println!("The square of 2 is {}", square(2));
13
14    // Closure can be passed as an argument to a function
15    println!("The square of 2 is {}", call_function(square, 2));
16
17    // Closure can be returned as a result from a function:
18    let square = return_function();
19 }
```





# Strings



# Strings

There are two types of strings in Rust

- `String`
- `&str`



# String

- Implemented as a smart pointer, specifically a Vec.
- Heap allocated, growable
- Can be mutated
- Always be a valid UTF-8, not null terminated sequence.



# Deref and String

String implements Deref trait, which means that you can pass a &String to something expecting a &str, and it will Just Work:

```
1 fn accepts_str(s: &str) {  
2     // code ..  
3 }  
4  
5 fn main(){  
6     let s = String::from("hello");  
7     accepts_str(&s);  
8 }
```



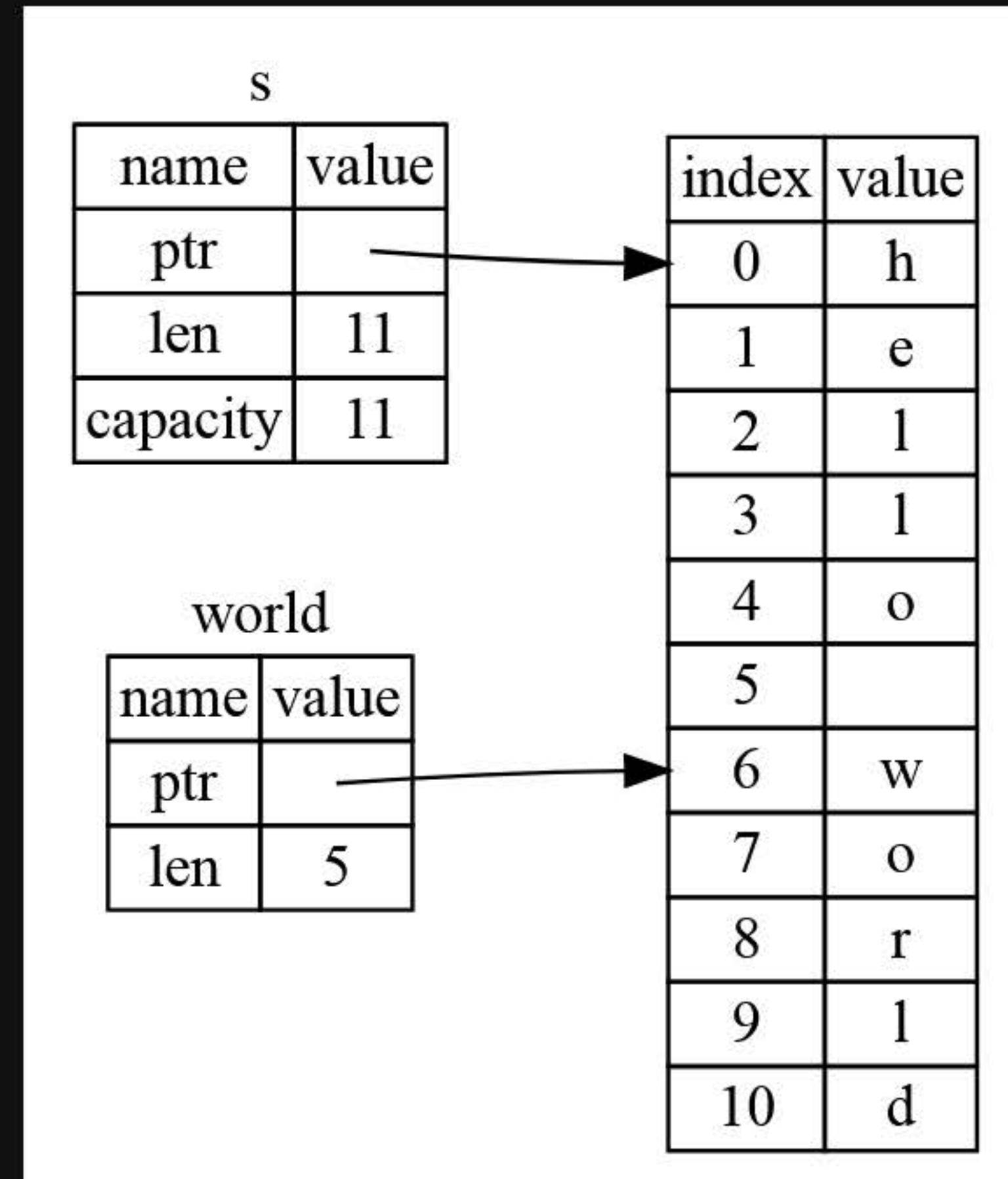


# &str

- An immutable reference to a valid UTF-8 sequence.
- May be anywhere, on the heap, stack, or in program memory.
- Only seen as a borrowed value.
- Has a fixed size, the value is known at run time



```
1 let s: String = String::from("hello world");
2
3 let world: &str = &s[6..11];
```





# Initialization

## String

```
1 // creating a String using the String::new
2 let s = String::new();
3 //creating a String using the String::from function
4 let s = String::from("hello");
5 //converting a `&str` to a `String` using the to_string method
6 let s = "hello".to_string();
```

Use `String` when you need to modify or own a string

## &str

```
1 //creating an &str from a string literal
2 let s = "hello";
3 //creating an &str from a String:
4 let s = String::from("hello");
5 let str_slice = &s[0..2];
```

Use `&str` when you just want to borrow a reference to a string





# Operations on Strings

concatenation | appending | slicing | splitting | replacing | iterating | checking starts  
prefix | checkin ends suffix

```
1 let s1 = String::from("hello world");
2 let mut s2 = String::from("rust is awesome");
3
4 let s3 = s1 + &s2; // Concat using the `+` operator
5 let s4 = format!("{}{}", s1, s2); // Concat using the format! macro
6 s2.push_str(", & memory safe"); // Appending using the push_str
7 let slice = &s[0..2]; // Slicing using string indices
8
9 for c in "hello".chars() { // Iterating over the string
10    println!("{}", c);
11 }
12 println!("Length:{}", s.len()); // Finding the length of a string
13 if(s1.starts_with("he")){} // Check if starts with a prefix
14 if(s1.ends_with("ld")){} // Check if ends with a suffix
```





# Ownership & Borrowing



# Ownership

- Every resource has a unique **owner**.
- The **owner** can change the owning value according to mutability.
- Ownership **can be transferred** to an other variable.
- Owner **cannot free or mutate** its resource **while it is borrowed**.
- Ownership model guarantees **safety**.



# Ownership Rules

Each value in Rust has only **one owner**

```
1 fn main () {  
2     let vector1 = vec![1, 2, 3];  
3 }
```

here 'vector1' is **owner**

```
1 fn main () {  
2     let vector1 = vec![1, 2, 3];  
3     ~~~~~ | <---  
4         |  
5         This memory  
6         gets `deallocated`  
7 }   <----- here `vector1` goes out of scope
```

Value is **dropped**,  
when owner goes out of scope.





# Ownership Rules

```
1 fn main () {  
2  
3     let vector1 = vec![1, 2, 3]; // `vector1` is invalid variable  
4     let vector2 = vector1; // because  
5     ~~~~~ // `vector2` is New Owner  
6  
7     println!("{}:?}", vector1);  
8 }
```

Error:

```
1 error[E0382]: borrow of moved value: `vector1`  
2     --> src/main.rs:4:19  
3  
4 2  |     let vector1 = vec![1, 2, 3];  
5  |     ----- move occurs because `vector1` has  
6  |     type `Vec< i32>`, which does not implement  
7  |     the `Copy` trait  
8 3  |     let vector2 = vector1;  
9  |     ----- value moved here  
10 4 |     println!("{}:?", vector1);  
11  |     ^^^^^^^ value borrowed here after move
```





# Borrowing

```
1 let vector2 = &vector1;  
2           ^ -----  
3 someFunction(&vector1);      |---> Reference  
4           ^ -----
```



# Borrowing Rules

All references are *immutable*  
by default.

```
1 fn main () {  
2     let mut vector1 = vec![1, 2, 3];  
3     ^^^ -----  
4     let vector2 = &mut vector1;  |---> Explicitly mentioned  
5             ^^^ -----      as `mutable`  
6                             using `mut` keyword  
7 }
```





# Borrowing Rules

Not **more than one**

'mutable reference' is allowed in *a scope*

```
1 fn main () {  
2     let mut _vector1 = vec![1, 2, 3];  
3     let _vector2 = &mut _vector1;  
4     let _vector3 = &mut _vector1;  
5     ^^^^^^^^^^^^^^^^^^ ---> NOT ALLOWED  
6 }
```

'mutable' and 'immutable' reference **can not go hand in hand** *within a scope*



# Traits



# Traits

- '*Traits*' are the abstract mechanism for adding functionality to types or it tells *Rust compiler* about functionality a type **must** provide.
- '*Traits*' are a way to group methods to define a set of **behavior** necessary to accomplish some purpose.



# Trait Definition

- Traits are defined by the `trait` keyword followed by the name of the trait
- While defining any trait we have to provide method signatures (method declaration)

```
1 pub trait Calculator {  
2     fn add(&self) -> i32;  
3     fn sub(&self) -> i32;  
4     fn div(&self) -> i32;  
5     fn mul(&self) -> i32;  
6 }
```





# Trait implementation

- `impl` keyword used to implement a trait
- `for` is used when implementing traits as in impl Trait for Type

```
1 impl trait_name for type_name {  
2  
3     //method definitions  
4 }
```



# Example

```
1 struct Data {
2     first_num: i32,
3     second_num: i32
4 }
5 impl Calculator for Data {
6     fn add(&self) -> i32 {
7         self.first_num + self.second_num
8     }
9     fn sub(&self) -> i32 {
10        self.first_num - self.second_num
11    }
12    fn div(&self) -> i32 {
13        self.first_num / self.second_num
14    }
15    fn mul(&self) -> i32 {
16        self.first_num * self.second_num
17    }
18 }
19 fn main() {
20     println!("Output of Add: {}", Data {first_num:2, second_num: 2}.add());
21     println!("Output of Sub: {}", Data {first_num:4, second_num: 2}.sub());
22     println!("Output of Div: {}", Data {first_num:10, second_num: 2}.div());
23     println!("Output of Mul: {}", Data {first_num:2, second_num: 2}.mul());
24 }
```

Output:

```
1 Compiling playground v0.0.1 (/playground)
2 Finished dev [unoptimized + debuginfo] target(s) in 0.59s
3     Running `target/debug/playground`
4
5 Output of Add: 4
6 Output of Sub: 2
7 Output of Div: 5
8 Output of Mul: 4
```



# Default implementation of trait

- Rust allows you to provide a default implementation of the Trait's methods
- A Type can keep the implementation or can override also
- In the same Trait, default implementation can call another method also





# Example

```
1 pub trait Calculator {
2     fn add(&self) -> i32;
3     fn sub(&self) -> i32;
4     fn get_result(&self) {
5         println!("The result of Addition is {}", self.add());
6     }
7 }
8
9 struct Data {
10    first_num: i32,
11    second_num: i32
12 }
13
14 impl Calculator for Data {
15     fn add(&self) -> i32 {
16         self.first_num + self.second_num
17     }
18
19     fn sub(&self) -> i32 {
20         self.first_num - self.second_num
21     }
22 }
23
24 fn main() {
25     Data {first_num:2, second_num: 2}.get_result();
26     println!("Output of Sub: {}", Data {first_num:4, second_num: 2}.sub());
27 }
```





# Implement traits for existing types

- Unlike *interfaces* in languages like Java, new traits can be implemented for existing types
- For example, we can implement **trait** for existing types like bool, f32, i32, etc.

```
1 trait PrintInfo {           //Define a trait called `PrintInfo`  
2     fn print_info(&self);  
3 }  
4  
5 impl PrintInfo for i32 {    // Implement `PrintInfo` for the built-in type `i32`  
6     fn print_info(&self) {  
7         println!("This is an integer: {}", self);  
8     }  
9 }  
10  
11 fn main() {  
12     let num: i32 = 42;  
13     num.print_info();        // Output: "This is an integer: 42"  
14 }
```





# Access Methods from the Same Trait

We can access other methods declared in the same trait using `self`

```
1 trait Calculator {  
2     fn add(&self) -> u32;  
3     // We can provide default method definitions.  
4     fn get_result(&self) {  
5         println!("Result of Add() is {}", self.add());  
6     }  
7 }
```





# Passing Trait as a Function's Parameters

Passing *trait* into a function's parameter is a quite interesting concept in the traits environment so, with the help of this user can put the restriction into his functionality like only limit functions can able to use it.

```
1 pub fn calculate(item: impl Calculator) {  
2     println!("Addition {}", item.add());  
3 }
```





# Trait Bound Concept

Trait Bound Concept is quite similar to the Passing Trait as a Function's Parameters but with some syntactical differences.

```
1 pub fn calculate< T: Calculator> (item: T) {  
2     println!("Addition {}", item.add());  
3 }
```



# Lifetimes

- It represents scope in which a reference is valid.
- It ensures that references do not outlive the data they refer to.
- By analyzing the lifetimes, the compiler can enforce memory safety.
- Lifetimes are denoted by apostrophes ('), also known as "ticks."
- Often used in function signatures, struct definitions, and generic type parameters to specify the relationship between references.

```
1 fn longest_string<'a>(s1: &'a str, s2: &'a str) -> &'a str {
2     if s1.len() > s2.len() {
3         s1
4     } else {
5         s2
6     }
7 }
8
9 fn main() {
10    let s1 = "Hello";
11    let s2 = "World";
12    let result = longest_string(s1, s2);
13    println!("Longest string: {}", result);
14 }
```





# Visibility Modifiers

- It controls the accessibility of structs, enums, functions, methods, and modules
- Rust provides three visibility modifiers:
  - `pub`: accessible from other modules or crates
  - `pub(crate)`: accessible only within the same crate
  - `pub(in path)`: accessible only within the specified module path and its submodules.

```
1 mod my_module {  
2     pub struct PublicStruct {}  
3     struct PrivateStruct {}  
4     pub fn public_function() {}  
5     fn private_function() {}  
6     pub(crate) fn crate_function() {}  
7     pub(in crate::my_module::nested) fn restricted_function() {}  
8 }  
9  
10 fn main() {  
11     let public_struct = my_module::PublicStruct {};  
12     my_module::public_function();  
13     // let private_struct = my_module::PrivateStruct {};// Error: not accessible here  
14     // my_module::private_function();// Error: not accessible here  
15 }
```



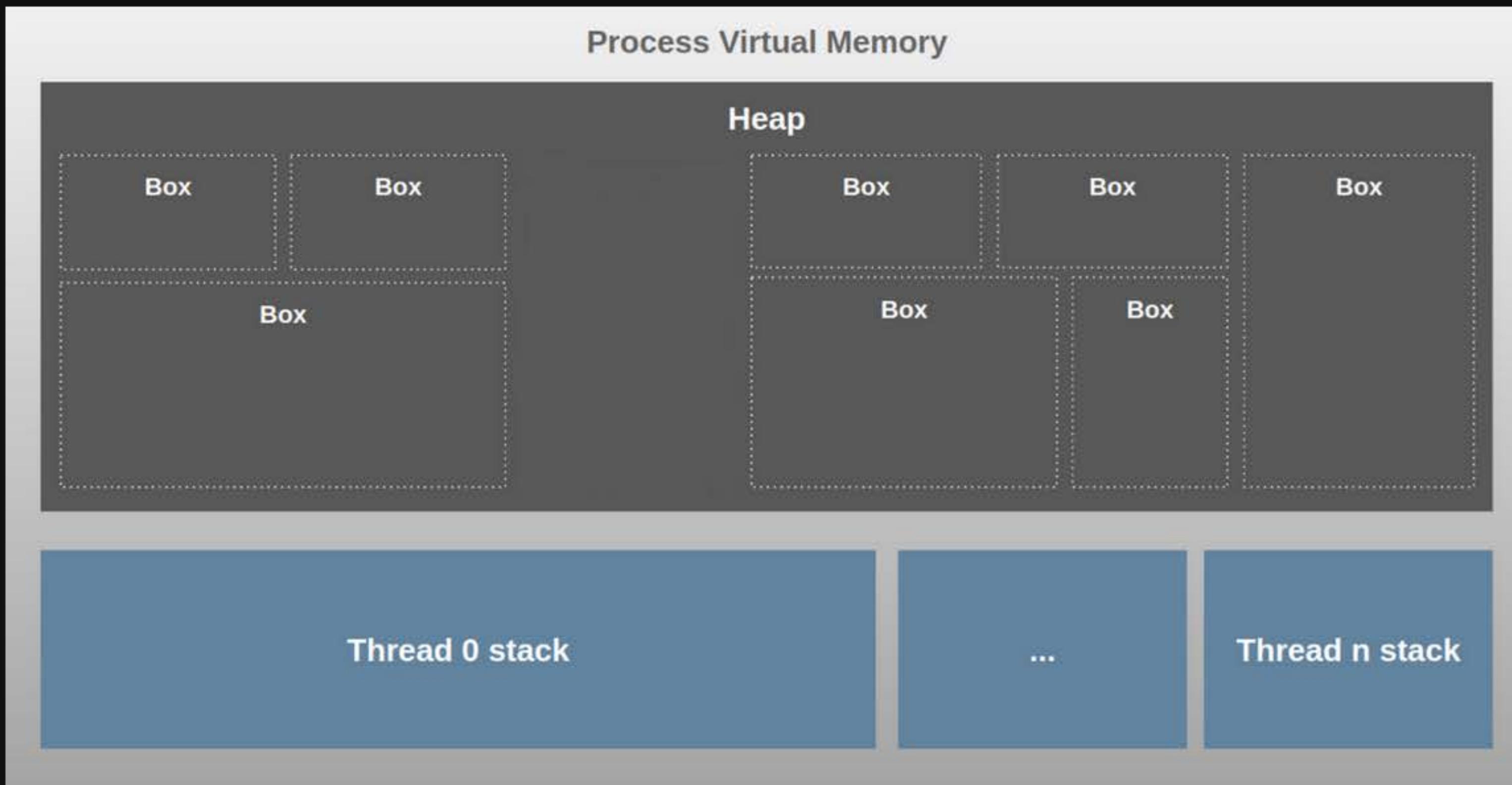


# Memory Management



# Memory Management

Rust program process is allocated some virtual memory by the Operating System(OS), this is the total memory that the process has access.





# Heap Memory

- This is where all **dynamic data**(any data for which size cannot be calculated at compile time) is stored.
- This is the biggest block of memory and the part managed by Rust's Ownership model.
- **Box:** The **Box** type is an abstraction for a heap-allocated value in Rust.
- Heap memory is allocated when `Box::new` is called.
- A `Box < T >` holds the **smart-pointer** to the heap memory, allocated for type `T` and the reference is saved on the Stack.



# Stack Memory

- This is the Stack memory area and there is one stack per thread.
- This is where static values are allocated by default.
- Static data(data size known at compile time) includes function frames, primitive values, structs and pointers to dynamic data in Heap



# Memory Usage

All values in Rust are allocated on the Stack by default. There are two **exceptions** to this

1. When size of the value is dynamic like **Strings/Vectors**
2. When you manually create a **Box < T >** value which is allocated on heap

In both exception cases, the value will be allocated on Heap and its pointer will live on the Stack.



## Rust Memory usage

```
struct Employee<'a> {
    // The `a defines the lifetime of dynamic data
    name: &'a str,
    salary: i32,
    sales: i32,
    bonus: i32,
}

const BONUS_PERCENTAGE: i32 = 10;

// salary is borrowed
fn get_bonus_percentage(salary: &i32) -> i32 {
    let percentage = (salary * BONUS_PERCENTAGE) / 100;
    return percentage;
}

// salary is borrowed
fn find_employee_bonus(salary: &i32, no_of_sales: i32) -> i32 {
    let bonus_percentage = get_bonus_percentage(salary);
    let bonus = bonus_percentage * no_of_sales;
    return bonus;
}

fn main() {
    // variable is declared as mutable
    let mut john = Employee {
        name: &format!("{}","John"), // explicitly making the value dynamic
        salary: 5000,
        sales: 5,
        bonus: 0,
    };

    // salary is borrowed while sales is cloned since i32 is a primitive
    john.bonus = find_employee_bonus(&john.salary, john.sales);
    println!("Bonus for {} is {}", john.name, john.bonus);
}
```

Thread stack

main frame

Heap





## Rust Memory usage

```
struct Employee<'a> {
    // The 'a defines the lifetime of dynamic data
    name: &'a str,
    salary: i32,
    sales: i32,
    bonus: i32,
}

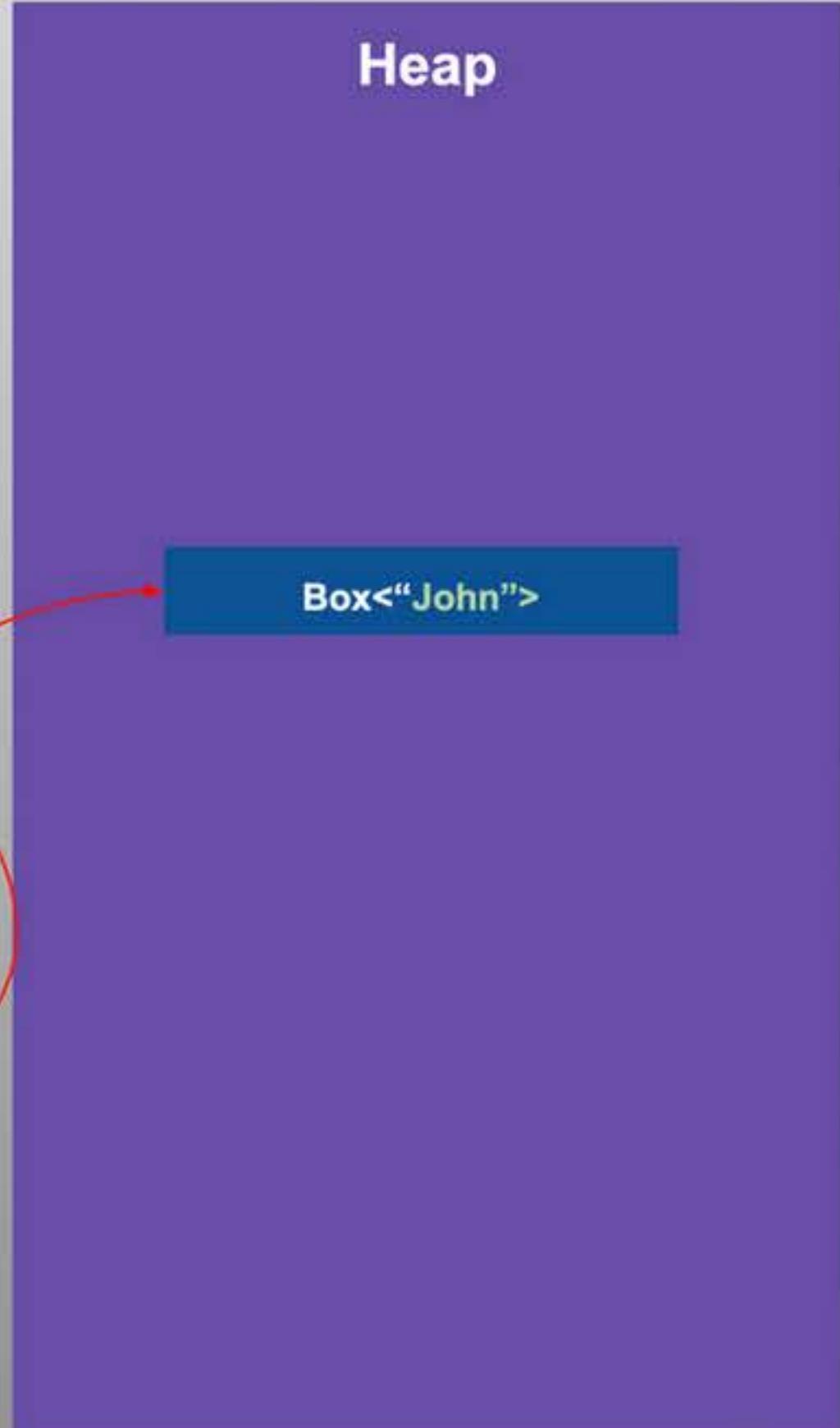
const BONUS_PERCENTAGE: i32 = 10;

// salary is borrowed
fn get_bonus_percentage(salary: &i32) -> i32 {
    let percentage = (salary * BONUS_PERCENTAGE) / 100;
    return percentage;
}

// salary is borrowed
fn find_employee_bonus(salary: &i32, no_of_sales: i32) -> i32 {
    let bonus_percentage = get_bonus_percentage(salary);
    let bonus = bonus_percentage * no_of_sales;
    return bonus;
}

fn main() {
    // variable is declared as mutable
    let mut john = Employee {
        name: &format!("{}","John"), // explicitly making the value dynamic
        salary: 5000,
        sales: 5,
        bonus: 0,
    };

    // salary is borrowed while sales is cloned since i32 is a primitive
    john.bonus = find_employee_bonus(&john.salary, john.sales);
    println!("Bonus for {} is {}", john.name, john.bonus);
}
```





## Rust Memory usage

```
struct Employee<'a> {
    // The 'a defines the lifetime of dynamic data
    name: &'a str,
    salary: i32,
    sales: i32,
    bonus: i32,
}

const BONUS_PERCENTAGE: i32 = 10;

// salary is borrowed
fn get_bonus_percentage(salary: &i32) -> i32 {
    let percentage = (salary * BONUS_PERCENTAGE) / 100;
    return percentage;
}

// salary is borrowed
fn find_employee_bonus(salary: &i32, no_of_sales: i32) -> i32 {
    let bonus_percentage = get_bonus_percentage(salary);
    let bonus = bonus_percentage * no_of_sales;
    return bonus;
}

fn main() {
    // variable is declared as mutable
    let mut john = Employee {
        name: &format!("{}","John"), // explicitly making the value dynamic
        salary: 5000,
        sales: 5,
        bonus: 0,
    };

    // salary is borrowed while sales is cloned since it is a primitive
    john.bonus = find_employee_bonus(&john.salary, john.sales);
    println!("Bonus for {} is {}", john.name, john.bonus);
}
```

Thread stack

main frame	
john	
name	
salary	5000
sales	5
bonus	0

Heap

Box<"John">





## Rust Memory usage

```
struct Employee<'a> {
    // The 'a defines the lifetime of dynamic data
    name: &'a str,
    salary: i32,
    sales: i32,
    bonus: i32,
}

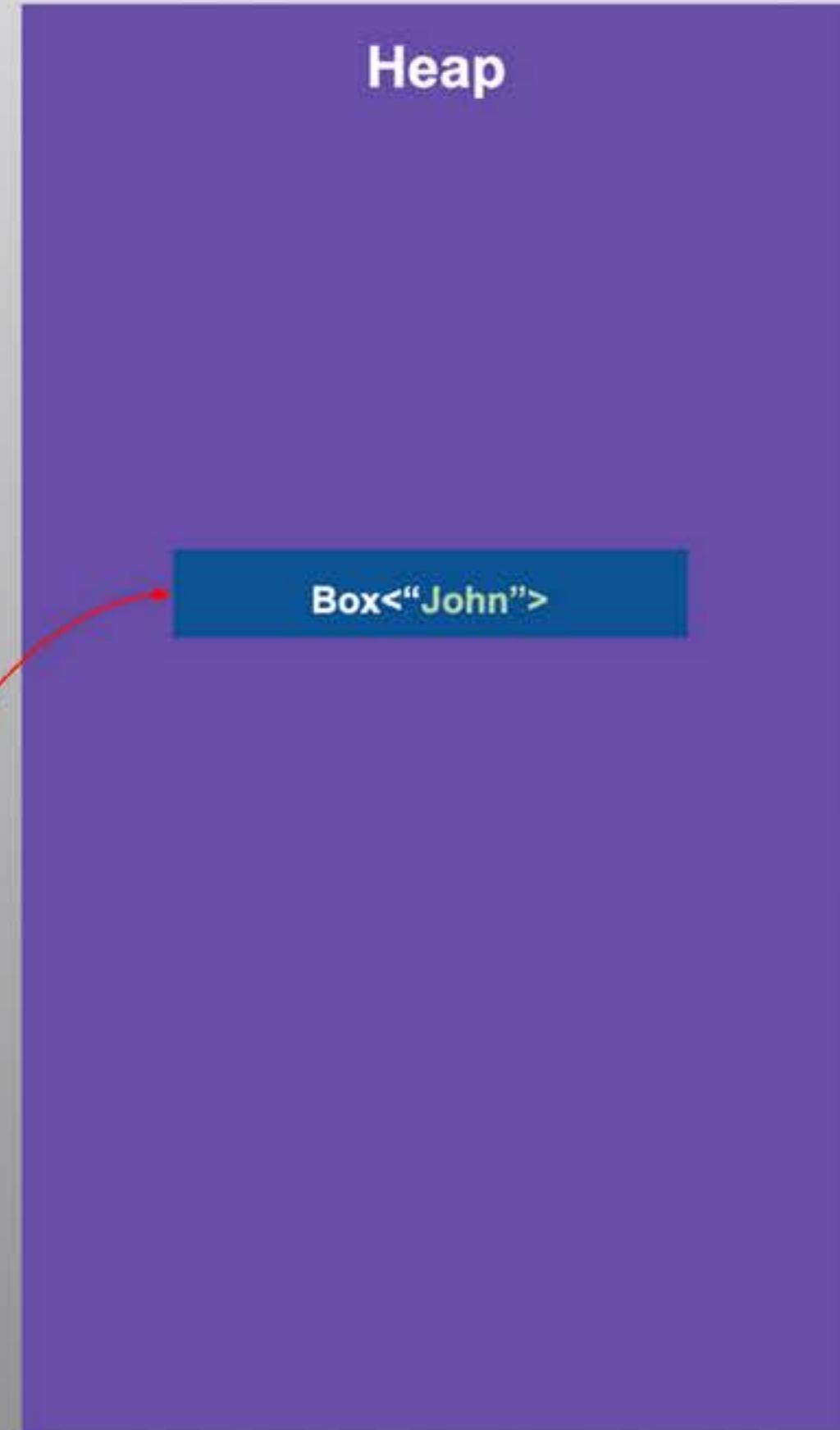
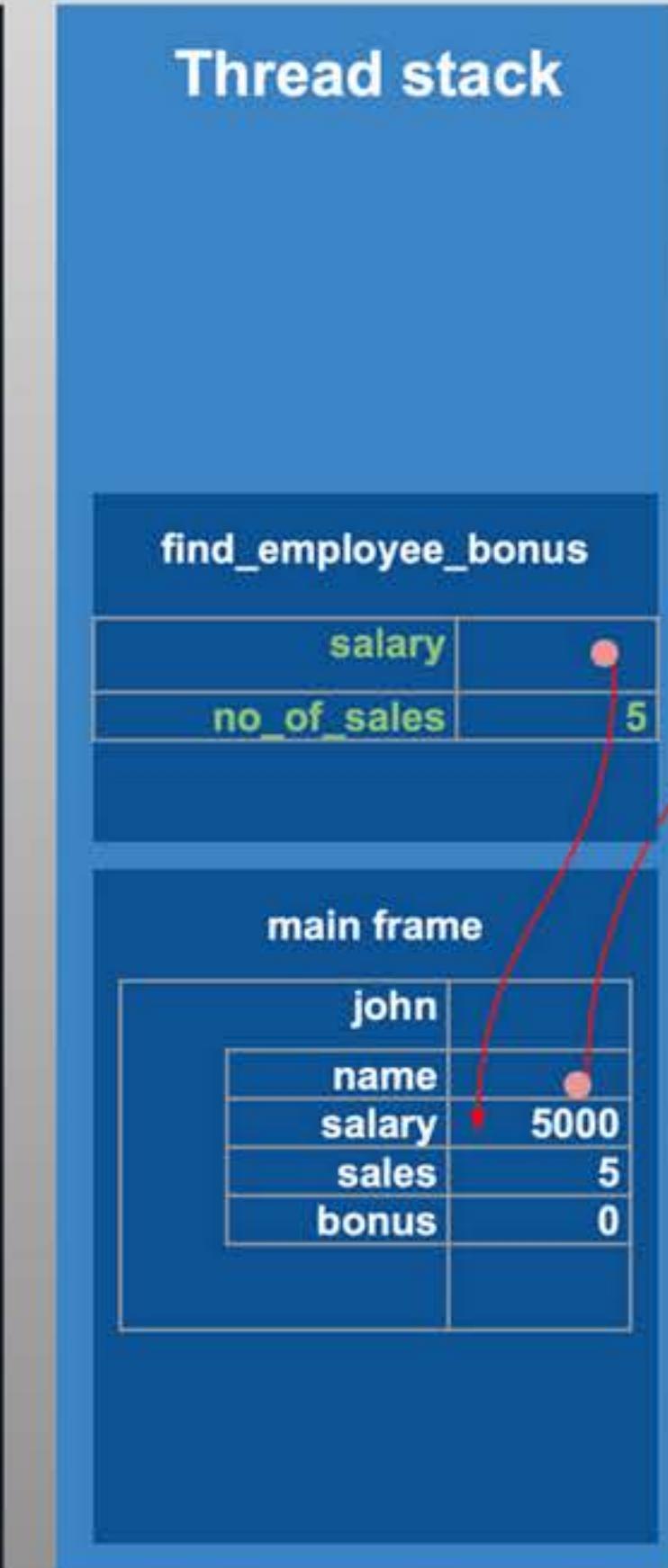
const BONUS_PERCENTAGE: i32 = 10;

// salary is borrowed
fn get_bonus_percentage(salary: &i32) -> i32 {
    let percentage = (salary * BONUS_PERCENTAGE) / 100;
    return percentage;
}

// salary is borrowed
fn find_employee_bonus(salary: &i32, no_of_sales: i32) -> i32 {
    let bonus_percentage = get_bonus_percentage(salary);
    let bonus = bonus_percentage * no_of_sales;
    return bonus;
}

fn main() {
    // variable is declared as mutable
    let mut john = Employee {
        name: &format!("{}","John"), // explicitly making the value dynamic
        salary: 5000,
        sales: 5,
        bonus: 0,
    };

    // salary is borrowed while sales is cloned since it's a primitive
    john.bonus = find_employee_bonus(&john.salary, john.sales);
    println!("Bonus for {} is {}", john.name, john.bonus);
}
```





## Rust Memory usage

```
struct Employee<'a> {
    // The 'a defines the lifetime of dynamic data
    name: &'a str,
    salary: i32,
    sales: i32,
    bonus: i32,
}

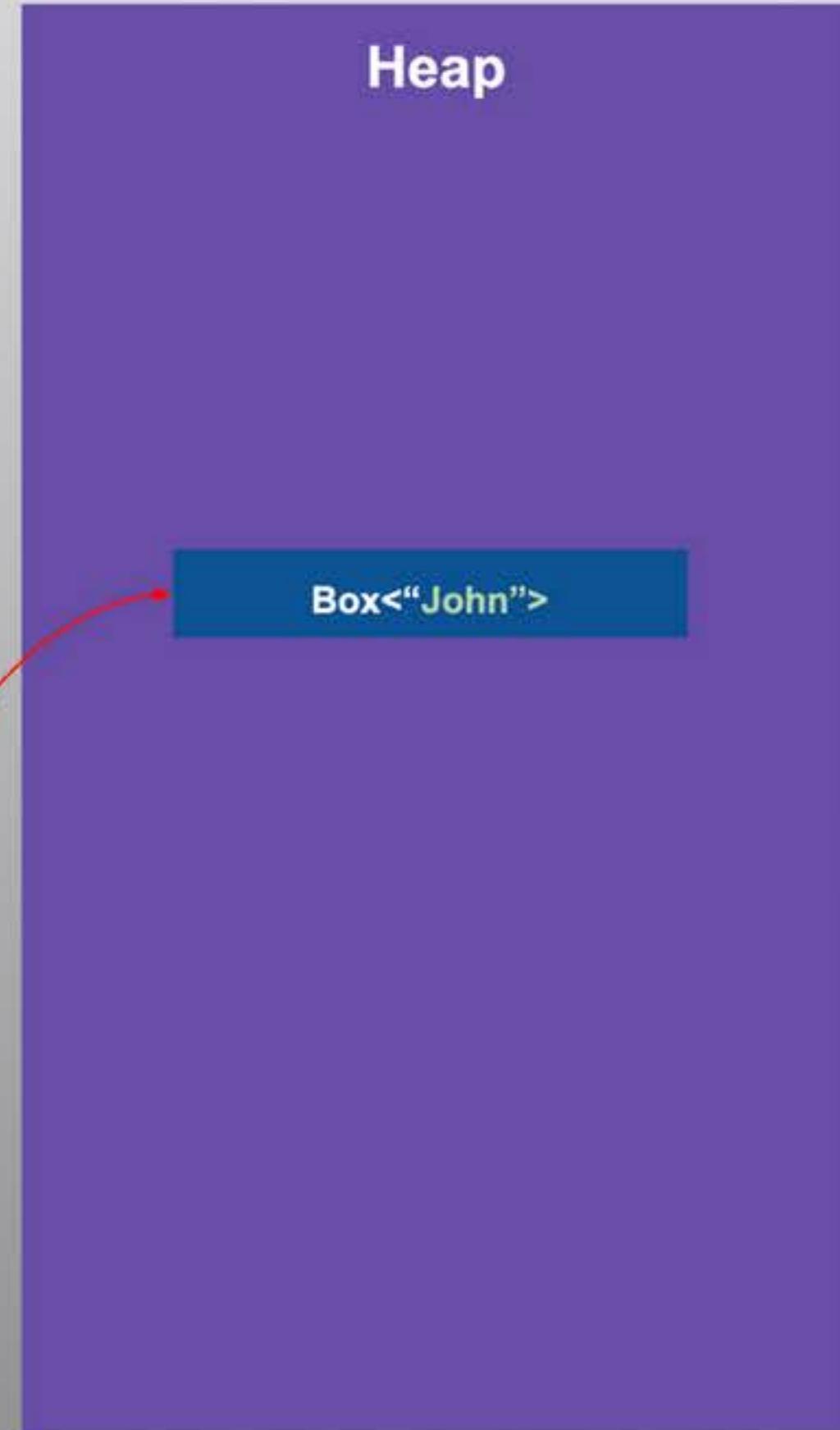
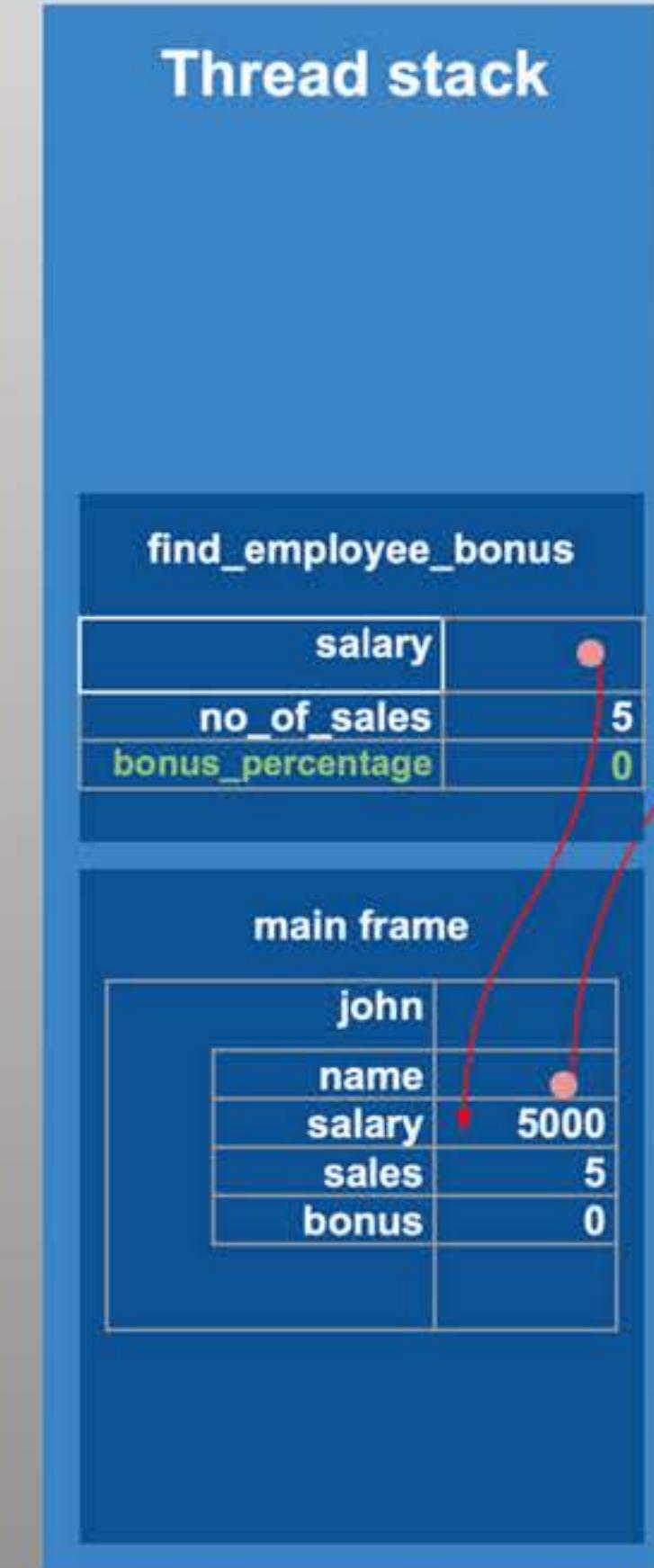
const BONUS_PERCENTAGE: i32 = 10;

// salary is borrowed
fn get_bonus_percentage(salary: &i32) -> i32 {
    let percentage = (salary * BONUS_PERCENTAGE) / 100;
    return percentage;
}

// salary is borrowed
fn find_employee_bonus(salary: &i32, no_of_sales: i32) -> i32 {
    let bonus_percentage = get_bonus_percentage(salary);
    let bonus = bonus_percentage * no_of_sales;
    return bonus;
}

fn main() {
    // variable is declared as mutable
    let mut john = Employee {
        name: &format!("{}","John"), // explicitly making the value dynamic
        salary: 5000,
        sales: 5,
        bonus: 0,
    };

    // salary is borrowed while sales is cloned since it's a primitive
    john.bonus = find_employee_bonus(&john.salary, john.sales);
    println!("Bonus for {} is {}", john.name, john.bonus);
}
```





## Rust Memory usage

```
struct Employee<'a> {
    // The 'a defines the lifetime of dynamic data
    name: &'a str,
    salary: i32,
    sales: i32,
    bonus: i32,
}

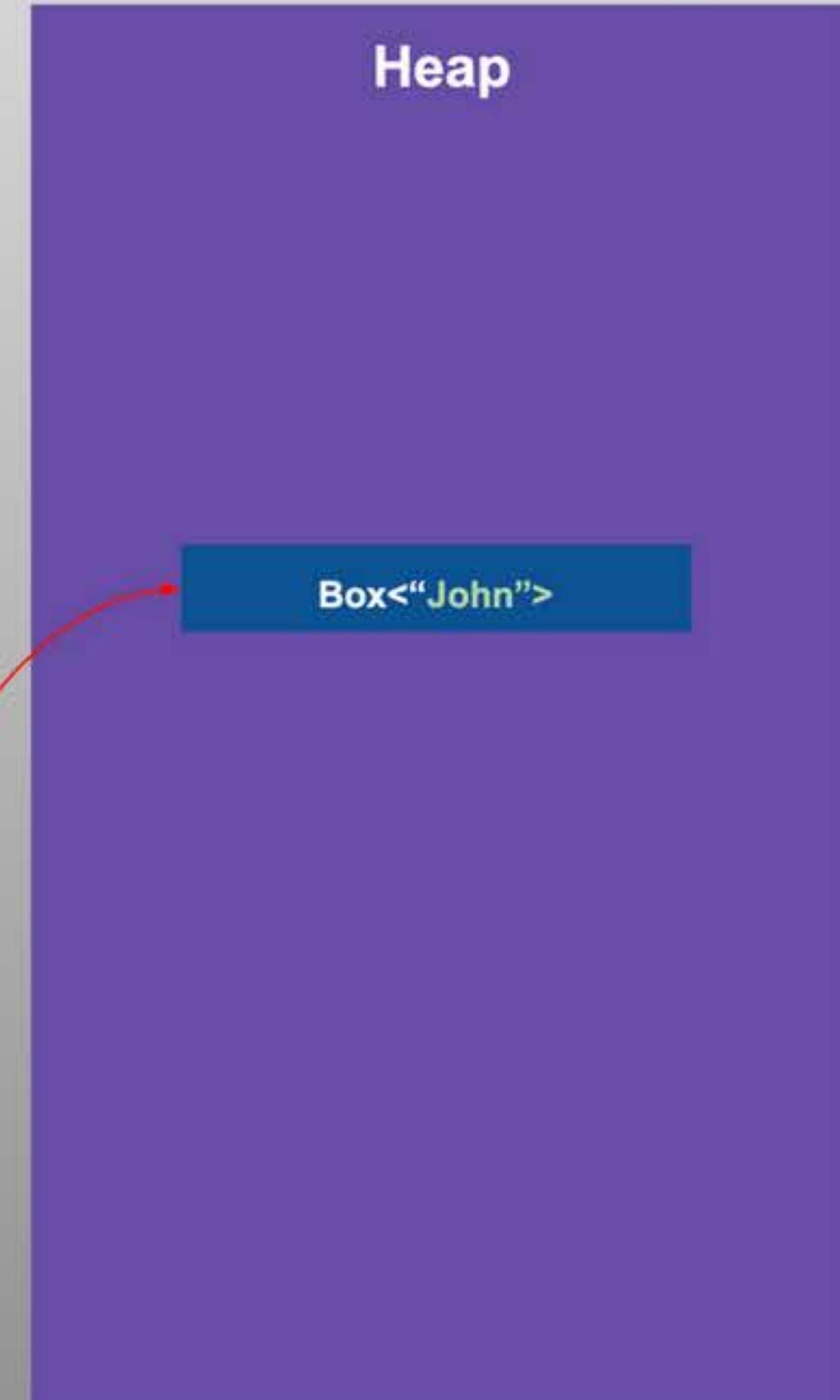
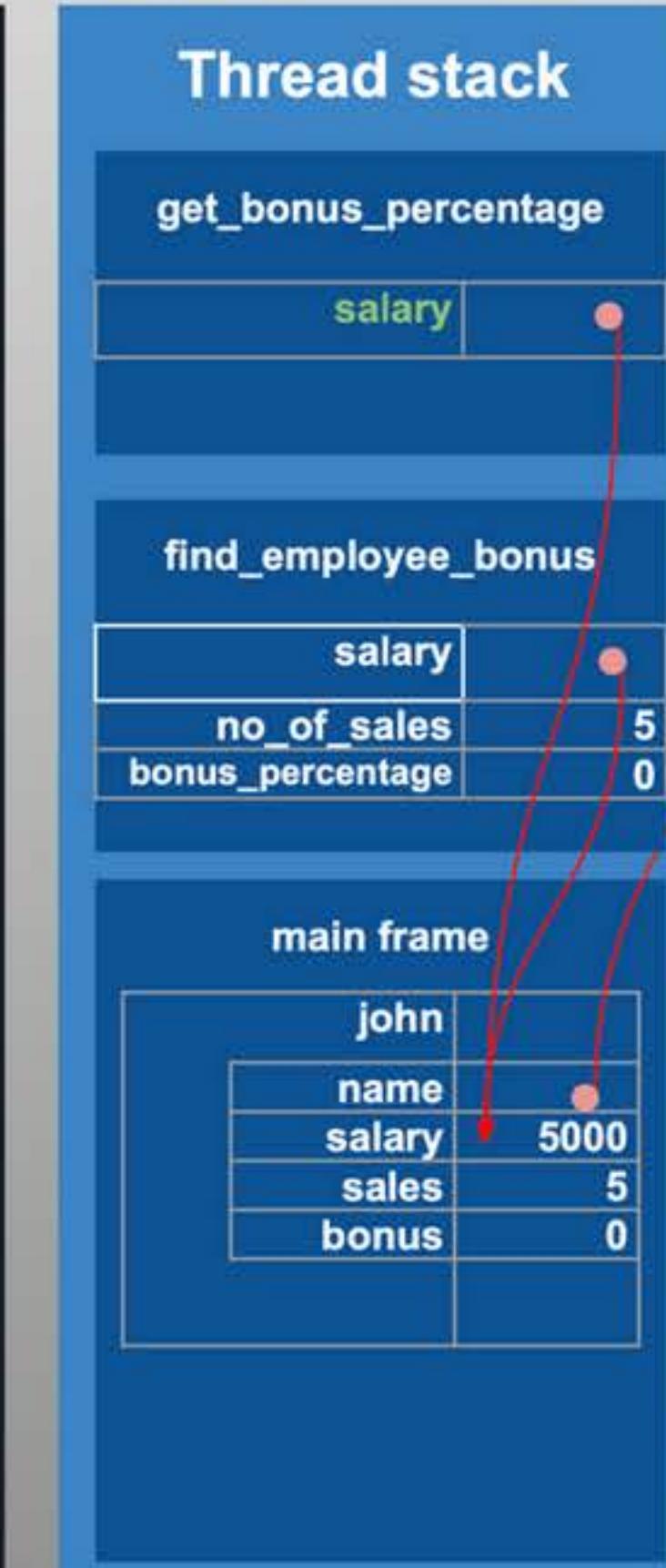
const BONUS_PERCENTAGE: i32 = 10;

// salary is borrowed
fn get_bonus_percentage(salary: &i32) -> i32 {
    let percentage = (salary * BONUS_PERCENTAGE) / 100;
    return percentage;
}

// salary is borrowed
fn find_employee_bonus(salary: &i32, no_of_sales: i32) -> i32 {
    let bonus_percentage = get_bonus_percentage(salary);
    let bonus = bonus_percentage * no_of_sales;
    return bonus;
}

fn main() {
    // variable is declared as mutable
    let mut john = Employee {
        name: &format!("{}","John"), // explicitly making the value dynamic
        salary: 5000,
        sales: 5,
        bonus: 0,
    };

    // salary is borrowed while sales is cloned since i32 is a primitive
    john.bonus = find_employee_bonus(&john.salary, john.sales);
    println!("Bonus for {} is {}", john.name, john.bonus);
}
```





## Rust Memory usage

```
struct Employee<'a> {
    // The 'a defines the lifetime of dynamic data
    name: &'a str,
    salary: i32,
    sales: i32,
    bonus: i32,
}

const BONUS_PERCENTAGE: i32 = 10;

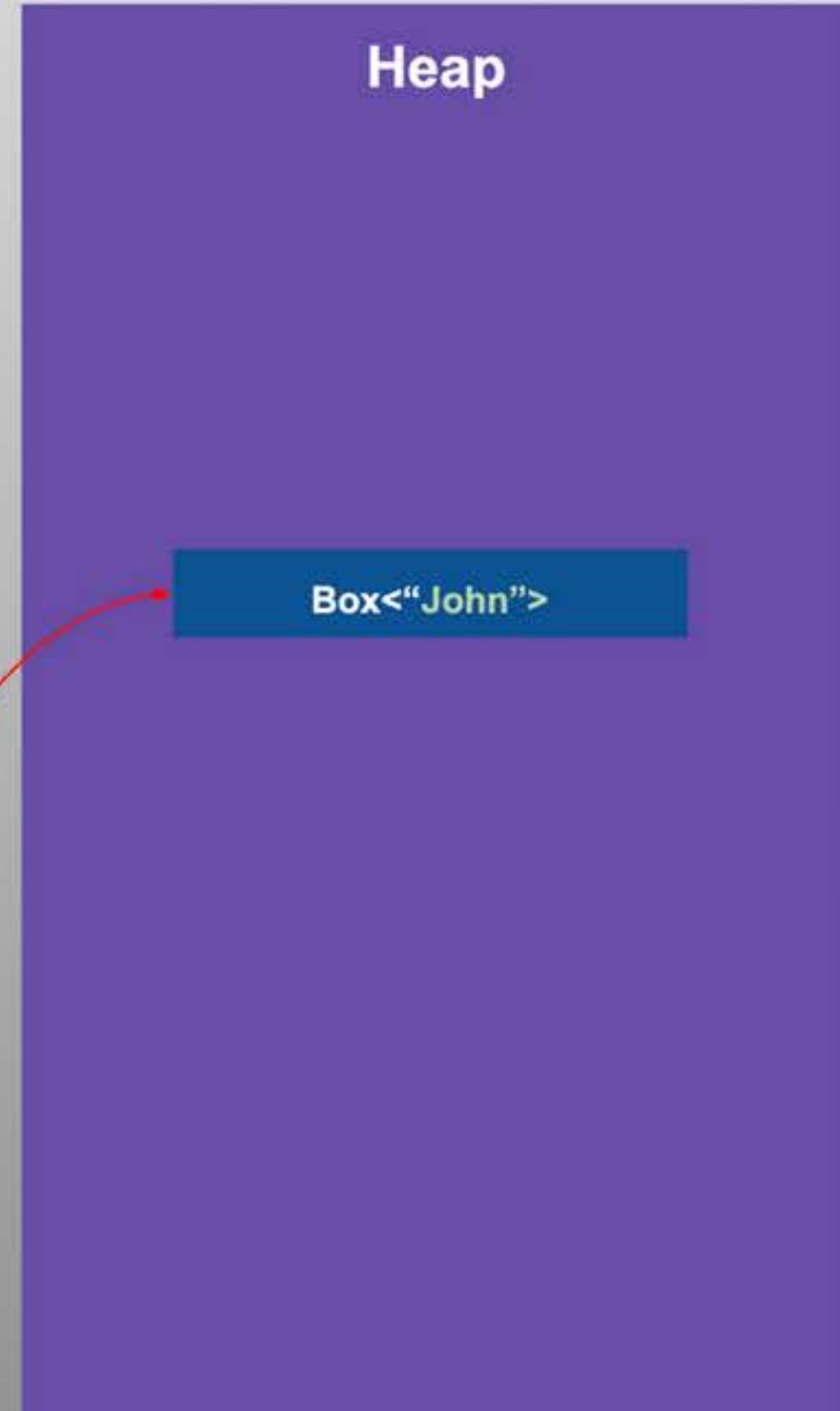
// salary is borrowed
fn get_bonus_percentage(salary: &i32) -> i32 {
    let percentage = (salary * BONUS_PERCENTAGE) / 100;
    return percentage;
}

// salary is borrowed
fn find_employee_bonus(salary: &i32, no_of_sales: i32) -> i32 {
    let bonus_percentage = get_bonus_percentage(salary);
    let bonus = bonus_percentage * no_of_sales;
    return bonus;
}

fn main() {
    // variable is declared as mutable
    let mut john = Employee {
        name: &format!("{}","John"), // explicitly making the value dynamic
        salary: 5000,
        sales: 5,
        bonus: 0,
    };

    // salary is borrowed while sales is cloned since it is a primitive
    john.bonus = find_employee_bonus(&john.salary, john.sales);
    println!("Bonus for {} is {}", john.name, john.bonus);
}
```

Thread stack	
get_bonus_percentage	
salary	●
percentage	500
find_employee_bonus	
salary	●
no_of_sales	5
bonus_percentage	0
main frame	
john	●
name	
salary	5000
sales	5
bonus	0





## Rust Memory usage

```
struct Employee<'a> {
    // the 'a defines the lifetime of dynamic data
    name: &'a str,
    salary: i32,
    sales: i32,
    bonus: i32,
}

const BONUS_PERCENTAGE: i32 = 10;

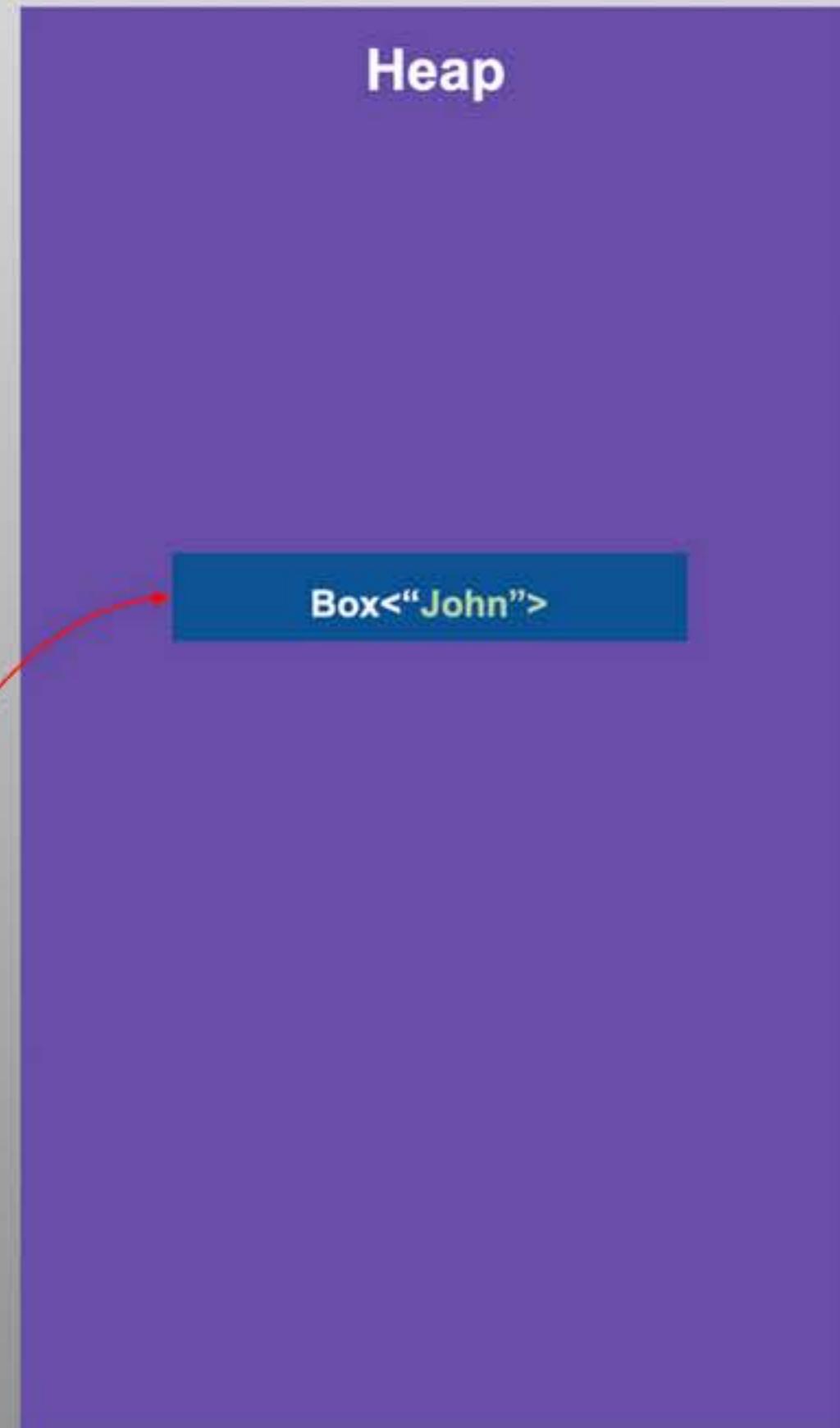
// salary is borrowed
fn get_bonus_percentage(salary: &i32) -> i32 {
    let percentage = (salary * BONUS_PERCENTAGE) / 100;
    return percentage;
}

// salary is borrowed
fn find_employee_bonus(salary: &i32, no_of_sales: i32) -> i32 {
    let bonus_percentage = get_bonus_percentage(salary);
    let bonus = bonus_percentage * no_of_sales;
    return bonus;
}

fn main() {
    // variable is declared as mutable
    let mut john = Employee {
        name: &format!("{}","John"), // explicitly making the value dynamic
        salary: 5000,
        sales: 5,
        bonus: 0,
    };

    // salary is borrowed while sales is cloned since i32 is a primitive
    john.bonus = find_employee_bonus(&john.salary, john.sales);
    println!("Bonus for {} is {}", john.name, john.bonus);
}
```

Thread stack	
<code>get_bonus_percentage</code>	
salary	●
percentage	500
return	500
<code>find_employee_bonus</code>	
salary	●
no_of_sales	5
bonus_percentage	0
<code>main frame</code>	
john	●
name	
salary	5000
sales	5
bonus	0





## Rust Memory usage

```
struct Employee<'a> {
    // The 'a defines the lifetime of dynamic data
    name: &'a str,
    salary: i32,
    sales: i32,
    bonus: i32,
}

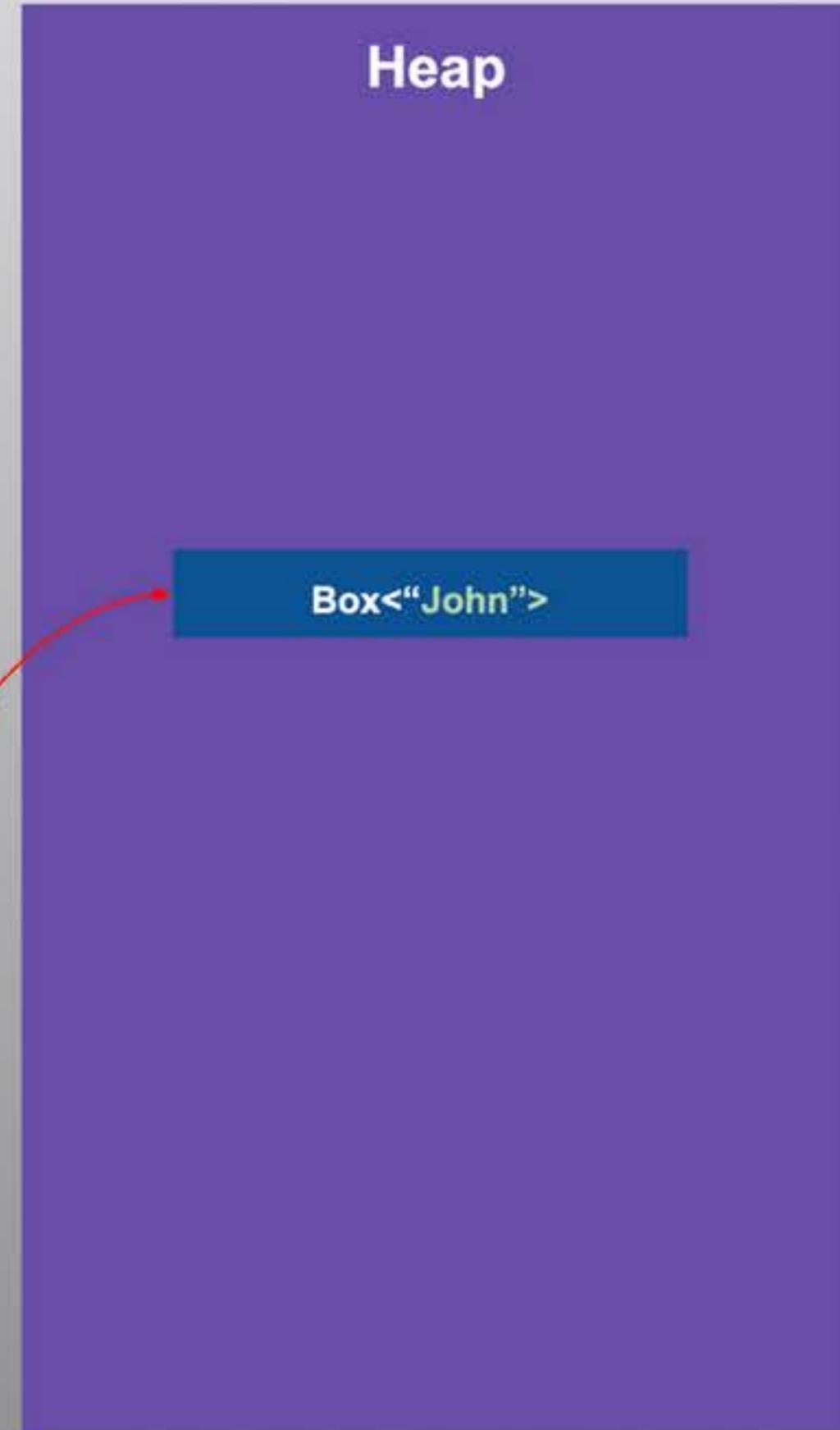
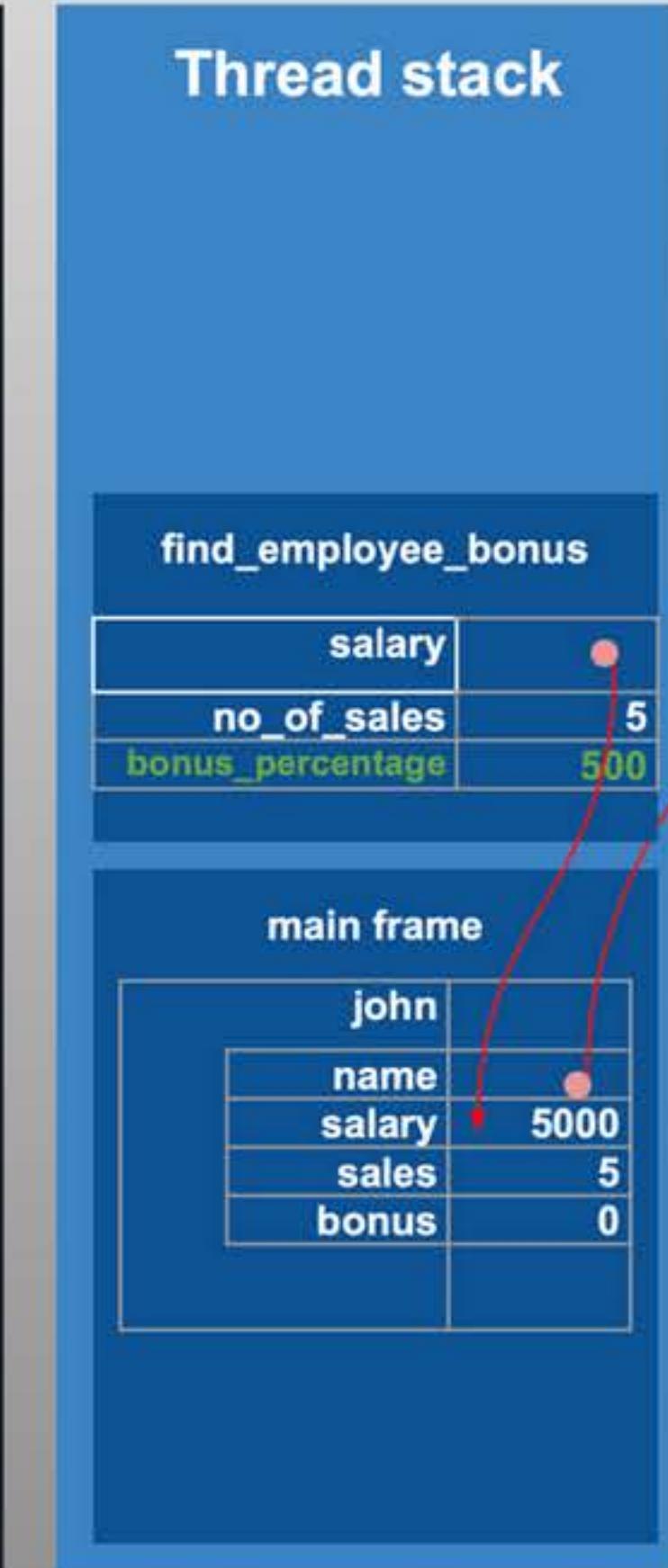
const BONUS_PERCENTAGE: i32 = 10;

// salary is borrowed
fn get_bonus_percentage(salary: &i32) -> i32 {
    let percentage = (salary * BONUS_PERCENTAGE) / 100;
    return percentage;
}

// salary is borrowed
fn find_employee_bonus(salary: &i32, no_of_sales: i32) -> i32 {
    let bonus_percentage = get_bonus_percentage(salary);
    let bonus = bonus_percentage * no_of_sales;
    return bonus;
}

fn main() {
    // variable is declared as mutable
    let mut john = Employee {
        name: &format!("{}","John"), // explicitly making the value dynamic
        salary: 5000,
        sales: 5,
        bonus: 0,
    };

    // salary is borrowed while sales is cloned since it's a primitive
    john.bonus = find_employee_bonus(&john.salary, john.sales);
    println!("Bonus for {} is {}", john.name, john.bonus);
}
```





## Rust Memory usage

```
struct Employee<'a> {
    // The 'a denotes the lifetime of dynamic data
    name: &'a str,
    salary: i32,
    sales: i32,
    bonus: i32,
}

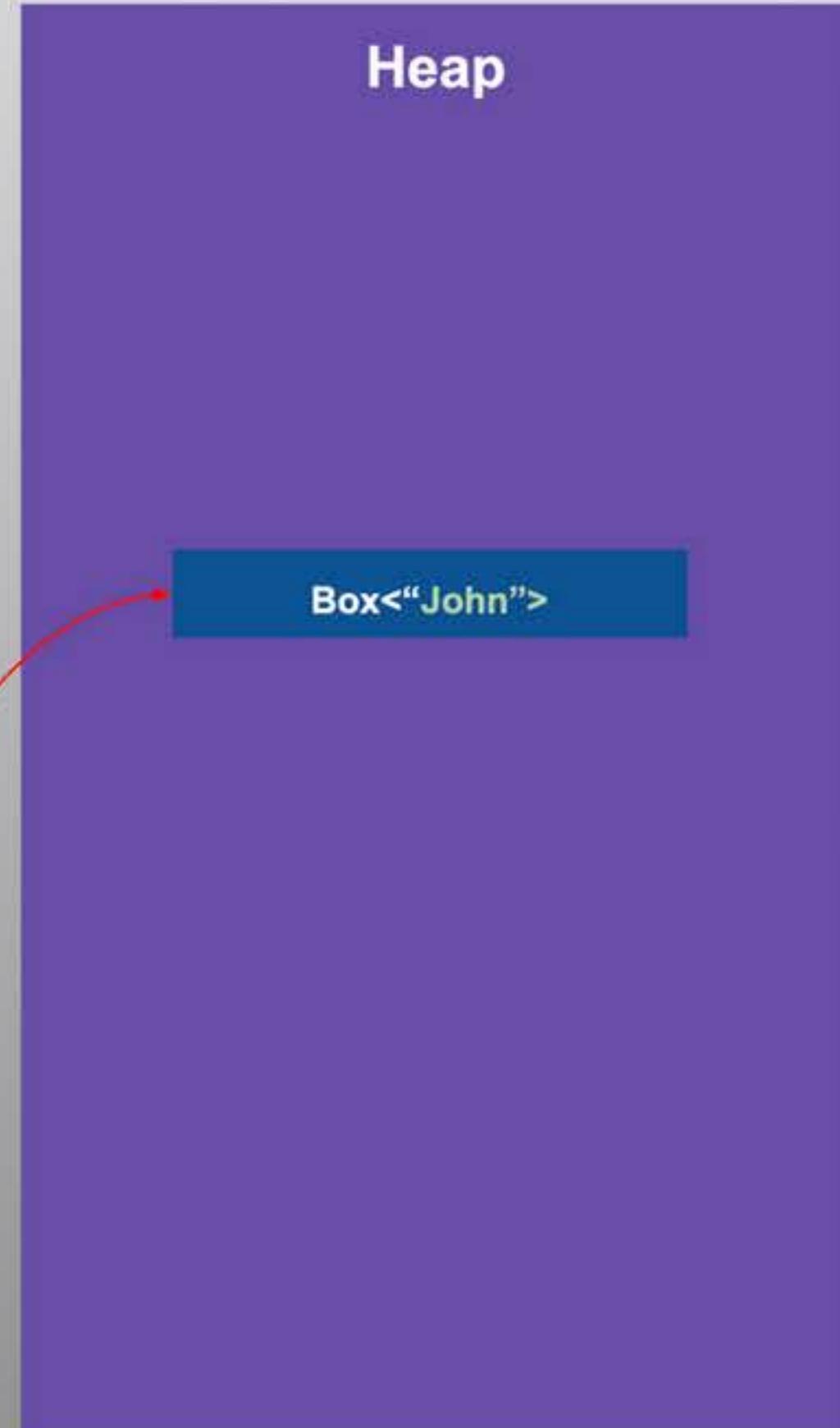
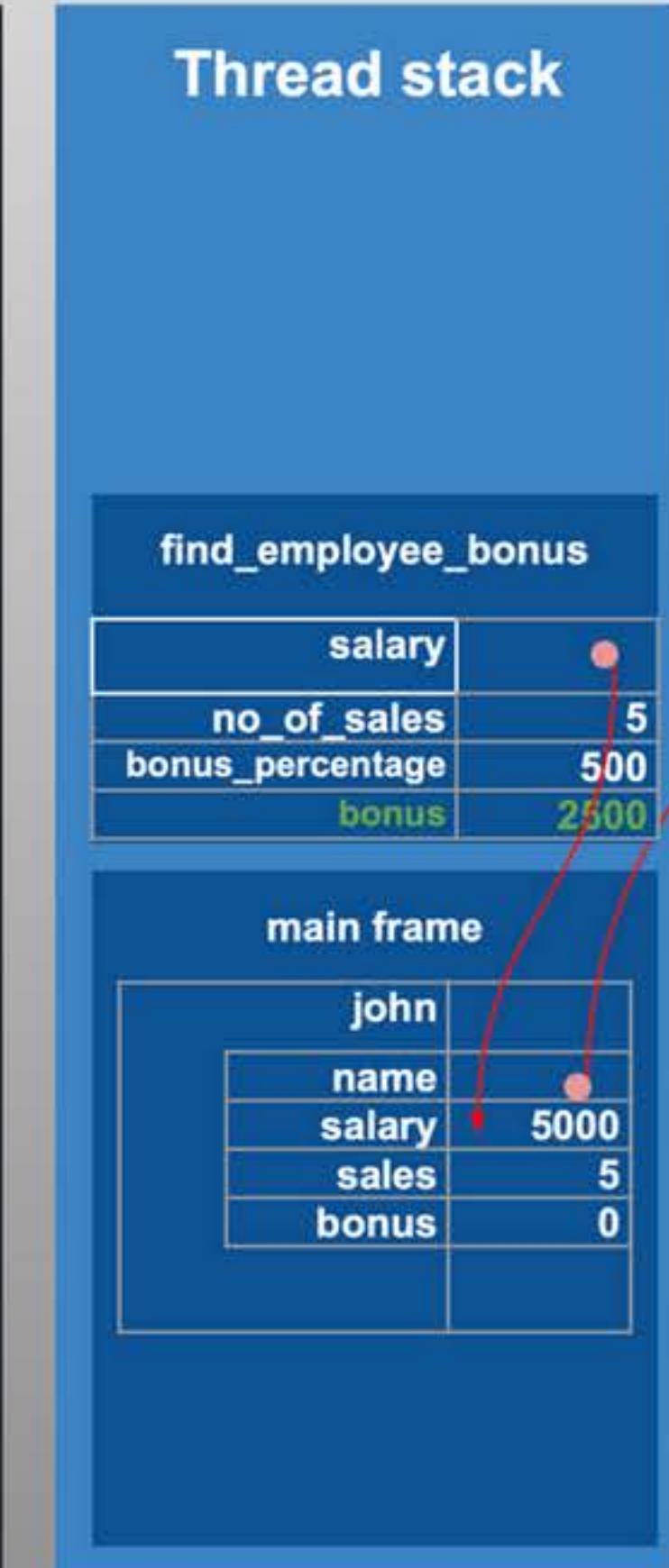
const BONUS_PERCENTAGE: i32 = 10;

// salary is borrowed
fn get_bonus_percentage(salary: &i32) -> i32 {
    let percentage = (salary * BONUS_PERCENTAGE) / 100;
    return percentage;
}

// salary is borrowed
fn find_employee_bonus(salary: &i32, no_of_sales: i32) -> i32 {
    let bonus_percentage = get_bonus_percentage(salary);
    let bonus = bonus_percentage * no_of_sales;
    return bonus;
}

fn main() {
    // variable is declared as mutable
    let mut john = Employee {
        name: &format!("{}","John"), // explicitly making the value dynamic
        salary: 5000,
        sales: 5,
        bonus: 0,
    };

    // salary is borrowed while sales is cloned since i32 is a primitive
    john.bonus = find_employee_bonus(&john.salary, john.sales);
    println!("Bonus for {} is {}", john.name, john.bonus);
}
```





## Rust Memory usage

```
struct Employee<'a> {
    // The 'a' denotes the lifetime of dynamic data
    name: &'a str,
    salary: i32,
    sales: i32,
    bonus: i32,
}

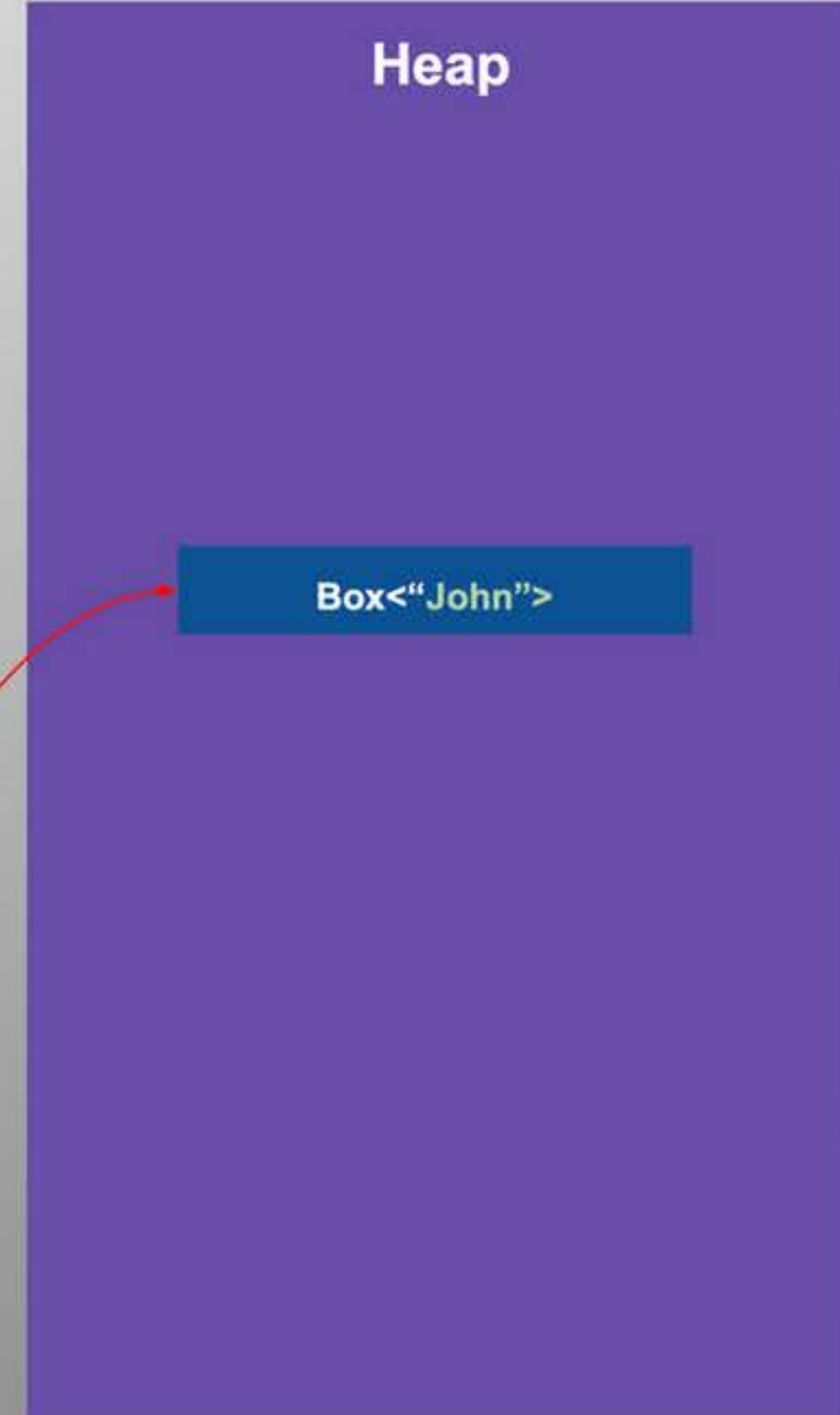
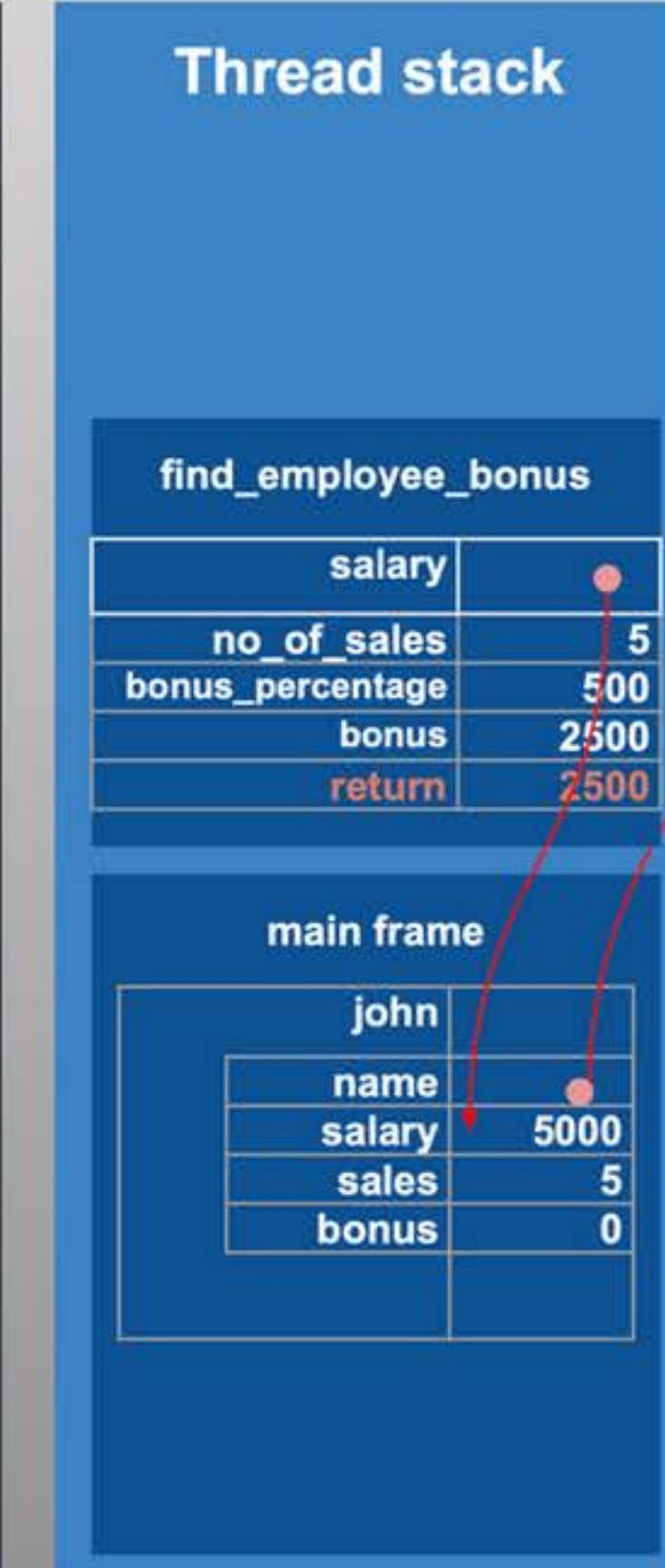
const BONUS_PERCENTAGE: i32 = 10;

// salary is borrowed
fn get_bonus_percentage(salary: &i32) -> i32 {
    let percentage = (salary * BONUS_PERCENTAGE) / 100;
    return percentage;
}

// salary is borrowed
fn find_employee_bonus(salary: &i32, no_of_sales: i32) -> i32 {
    let bonus_percentage = get_bonus_percentage(salary);
    let bonus = bonus_percentage * no_of_sales;
    return bonus;
}

fn main() {
    // variable is declared as mutable
    let mut john = Employee {
        name: &format!("{}","John"), // explicitly making the value dynamic
        salary: 5000,
        sales: 5,
        bonus: 0,
    };

    // salary is borrowed while sales is cloned since i32 is a primitive
    john.bonus = find_employee_bonus(&john.salary, john.sales);
    println!("Bonus for {} is {}", john.name, john.bonus);
}
```





## Rust Memory usage

```
struct Employee<'a> {
    // the 'a' defines the lifetime of dynamic data
    name: &'a str,
    salary: i32,
    sales: i32,
    bonus: i32,
}

const BONUS_PERCENTAGE: i32 = 10;

// salary is borrowed
fn get_bonus_percentage(salary: &i32) -> i32 {
    let percentage = (salary * BONUS_PERCENTAGE) / 100;
    return percentage;
}

// salary is borrowed
fn find_employee_bonus(salary: &i32, no_of_sales: i32) -> i32 {
    let bonus_percentage = get_bonus_percentage(salary);
    let bonus = bonus_percentage * no_of_sales;
    return bonus;
}

fn main() {
    // variable is declared as mutable
    let mut john = Employee {
        name: &format!("{}","John"), // explicitly making the value dynamic
        salary: 5000,
        sales: 5,
        bonus: 0,
    };

    // salary is borrowed while sales is cloned since i32 is a primitive
    john.bonus = find_employee_bonus(&john.salary, john.sales);
    println!("Bonus for {} is {}", john.name, john.bonus);
}
```

### Thread stack

main frame	
john	
name	John
salary	5000
sales	5
bonus	2500

### Heap

Box<"John">





## Rust Memory usage

```
struct Employee<'a> {
    // The 'a defines the lifetime of dynamic data
    name: &'a str,
    salary: i32,
    sales: i32,
    bonus: i32,
}

const BONUS_PERCENTAGE: i32 = 10;

// salary is borrowed
fn get_bonus_percentage(salary: &i32) -> i32 {
    let percentage = (salary * BONUS_PERCENTAGE) / 100;
    return percentage;
}

// salary is borrowed
fn find_employee_bonus(salary: &i32, no_of_sales: i32) -> i32 {
    let bonus_percentage = get_bonus_percentage(salary);
    let bonus = bonus_percentage * no_of_sales;
    return bonus;
}

fn main() {
    // variable is declared as mutable
    let mut john = Employee{
        name: &format!("{}","John"), // explicitly making the value dynamic
        salary: 5000,
        sales: 5,
        bonus: 0,
    };

    // salary is borrowed while sales is cloned since i32 is a primitive
    john.bonus = find_employee_bonus(&john.salary, john.sales);
    println!("Bonus for {} is {}", john.name, john.bonus);
}
```

### Thread stack

### Heap





# Smart Pointers

- A **pointer** is a general concept for a **variable** that contains an address in **memory**.
- Smart pointers implement the ``Deref`` and ``Drop`` traits.
  - The **Deref trait** allows an instance of the smart pointer struct to behave like a reference so you can write your code to work with either references or smart pointers.
  - The **Drop trait** allows you to customize the code that's run when an instance of the smart pointer goes out of scope.





# Smart pointers in the standard library

- `Box< T >` for allocating values on the heap
- `Rc< T >`, a reference counting type that enables multiple ownership
- `Ref< T >` and `RefMut< T >`, accessed through `RefCell< T >`, a type that enforces the borrowing rules at runtime instead of compile time



# Box< T >

- Boxes allow you to store data on the heap rather than the stack. What remains on the stack is the pointer to the heap data.
- Box< T > allows immutable or mutable borrows checked at compile time

```
1 fn main() {  
2     let b = Box::new(5);  
3     println!("b = {}", b);  
4 }
```



# Rc< T >

- To enable multiple ownership explicitly by using the Rust type Rc, which is an abbreviation for reference counting.
- The Rc type keeps track of the number of references to a value to determine whether or not the value is still in use
- If there are zero references to a value, the value can be cleaned up without any references becoming invalid.
- Rc allows only immutable borrows checked at compile time



# RefCell< T >

- Rc< T >, the RefCell< T > type represents single ownership over the data it holds.
- At any given time, you can have either (but not both) one mutable reference or any number of immutable references.
- Rc< T > enables multiple owners of the same data; Box< T > and RefCell< T > have single owners.
- RefCell< T > allows immutable or mutable borrows checked at runtime.



# Control Flow



# if expression

- The most common way to introduce control flow and branch code.
- Provide a condition and then execute the block of code if the condition is met.

```
1 fn main() {  
2     let price: i32 = 10;  
3     if price > 0 {  
4         println!("true");  
5     }  
6 }
```



# if-else expression

- An else expression can be added optionally.
- If no else is provided the program will skip the if block if the condition is false.

```
1 fn main(){
2     let price: i32 = 10;
3     if price > 0 {
4         println!("true");
5     }
6     else {
7         println!("false");
8     }
9 }
```





# else-if expression

- If you have more than two condition to check, if and else can be combined in else if expression.
- In the case all if and else if conditions evaluates to false, then the else block is executed.

```
1 fn main(){
2     let price: i32 = 10;
3     if price == 1 {
4         println!("price is 1.");
5     }
6     else if price == 2 {
7         println!("price is 2.");
8     }
9     else {
10        println!("other price");
11    }
12 }
```





# match

- Rust provides pattern matching with the `match` keyword
- A scrutinee expression is provided to compare to the patterns.
- Arms are evaluated and compared with the scrutineer expression

```
1 fn main(){
2     let x = 1;
3     match x {
4         1 => println!("one"),
5         2 => println!("two"),
6         3 => println!("three"),
7         4 => println!("four"),
8         5 => println!("five"),
9         _ => println!("something else"),
10    }
11 }
```





# match

- Here the scrutinee expression is 'x'
- Each arm has a pattern and some code. The `=>` operator separates the pattern and the code to run.
- The first arm with the matching pattern is chosen as the branch target of the match.

```
1 fn main(){
2   let x = 1;
3   match x {
4     1 => println!("one"),
5     2 => println!("two"),
6     3 => println!("three"),
7     4 => println!("four"),
8     5 => println!("five"),
9     _ => println!("something else"),
10  }
11 }
```





# loop

- Used to execute over block of code **forever**, Or until it is stopped, or the program quits.
- Instead of having this `loop` infinitely the **break** keyword can be used.

```
1 fn main() {  
2  
3     let mut i = 0;  
4  
5     loop {  
6         i += 1;  
7  
8         if i > 100 { break; }  
9     }  
10 }
```





# while

- Conditional loops
- Run until the condition is met or become false

```
1 fn main(){
2
3     let mut num = 3;
4
5     while num !=0 {
6
7         println!("{}", num);
8         num -= 1;
9
10    }
11    println!("LIFTOFF!!");
12 }
```





# for loop

- Iterate over a element in a collection.
- Every iteration of loop extracts values.

```
1 fn main(){
2
3     let a = [10, 20, 30, 40, 50];
4
5     for element in a.iter() {
6
7         println!("the value is {}", element);
8
9     }
10 }
```





# iterators

- In Rust, iterators help us achieve the process of looping.
- Using the ***iter()*** function, we tell Rust that the given array can be used with a loop.

```
1 fn main(){
2     let a = [10, 20, 30, 40, 50];
3     for element in a.iter() {
4         println!("the value is {}", element);
5     }
6 }
```



# iterator trait

- In Rust, all iterators implement a trait named 'iterator' that has a method called **next()**
- Using the **iter()** function, we tell Rust that the given array can be used with a loop.

```
1 fn main(){
2     let a = [10, 20, 30, 40, 50];
3     for element in a.iter() {
4         println!("the value is {}", element);
5     }
6 }
```

Any data type which implements **next()** method can be iterator.



# Difference in `into_iter`, `iter`, `iter_mut`

- **`into_iter`:** Consumes the collection, once the collection has been consumed, it is no longer available for reuse.
- **`iter`:** This borrows each element of the collection through each iteration, thus leaving the collection untouched and available for reuse after the loop
- **`iter_mut`:** This mutably borrows each element of the collection, allowing for the collection to be modified in place.





# Creating our own iterator

Example:

```
1 struct Counter {
2     count: u32,
3 }
4 impl Counter {
5     fn new() -> Counter {
6         Counter { count: 0 }
7     }
8 }
9 impl Iterator for Counter {
10    type Item = u32;
11    fn next(&mut self) -> Option<Self::Item> {
12        if self.count < 5 {
13            self.count += 1;
14            Some(self.count)
15        } else {
16            None
17        }
18    }
19 }
20 fn main() {
21     let mut counter = Counter::new();
22     assert_eq!(counter.next(), Some(1));
23     assert_eq!(counter.next(), Some(2));
24     assert_eq!(counter.next(), Some(3));
25     assert_eq!(counter.next(), Some(4));
26     assert_eq!(counter.next(), Some(5));
27     assert_eq!(counter.next(), None);
28 }
```





# Excercise:

```
1 struct Fibonacci {
2     curr: u32,
3     next: u32,
4 }
5
6 // Implement `Iterator` for `Fibonacci`.
7 // The `Iterator` trait only requires a method to be defined for the `next` element.
8 impl Iterator for Fibonacci {
9     // We can refer to this type using Self::Item
10    type Item = u32;
11
12    /* Implement next method */
13    fn next(&mut self)
14 }
15
16 // Returns a Fibonacci sequence generator
17 fn fibonacci() -> Fibonacci {
18     Fibonacci { curr: 0, next: 1 }
19 }
20
21 fn main() {
22     let mut fib = fibonacci();
23     assert_eq!(fib.next(), Some(1));
24     assert_eq!(fib.next(), Some(1));
25     assert_eq!(fib.next(), Some(2));
26     assert_eq!(fib.next(), Some(3));
27     assert_eq!(fib.next(), Some(5));
28 }
```





# + Operator & where Clause

+ operator is used to add more trait bound.

```
1 pub fn calculate< T: Calculator + other_Trait> (item: T) -> u32 {  
2     /// function's stuff  
3  
4 }
```

`where` clause is used to simplify the concept of Trait Bound like

```
1 /// without where clause  
2 pub fn calculate< T:Calculator, U:Display> (item: T, data: Display) -> u32 {  
3     /// function's stuff  
4 }  
5  
6 ///with where clause  
7 pub fn calculate< T, U> (item: T, data: Display) -> u32  
8     where T: Calculator,  
9         U: Display  
10 {  
11     /// function's stuff  
12 }
```





# Error Handling



# Types of Error Handling

- Error handling is process of anticipating and working with the possibility of failure
- Learning how to handle errors in rust most effective and time saving
- There are two types of Errors in rust
  - Unrecoverable Errors
  - Recoverable Errors



# Unrecoverable Errors

Unrecoverable errors are symptoms of bugs

**panic!** macro is a simplest way of handling errors

```
1 fn main() {  
2     panic!("crash and burn");  
3 }
```





# What happens when panic is encountered

- Failure message is printed
- Program winds up the stack
- Then it quits

## When to use panic ?

- Panic only have to be used when a program comes to an unrecoverable state.





# Recoverable Errors

Use of **Option** enum

Most errors aren't serious enough that require a program to stop entirely

**Option** < T > enum is useful to handle non-existing values

```
1 fn main(){
2     enum Option< T > {
3         None,
4         Some(T),
5     }
6 }
```



# Recoverable Errors

Use of **Result** enum

Result type enum is used for most common errors, when a operation is expected to work but does not.

**Result < T, E >** enum is well suited when problems are expected.

```
1 fn main(){
2     enum Result< T, E>{
3         Ok(T),
4         Err(E),
5     }
6 }
```





# Helper Functions of Result Type

`unwrap`

```
1 use std::fs::File;
2 fn main(){
3     let f = File::open("hello.txt").unwrap();
4 }
```

`expect`

```
1 use std::fs::File;
2 fn main(){
3     let f = File::open("hello.txt")
4             .expect("Failed to open hello.txt");
5 }
```



# The ? operator

This is similar to match statement

```
1 fn question() -> Result <(), Error> {  
2   // let x = ...  
3   match ultimate_answer(x) {  
4     Ok(_) => {},  
5     Err(err) => return Err(err.into()),  
6   };  
7   // code  
8 }
```

The above code can be simplified using this ?

```
1 fn question() -> Result <(), Error> {  
2   // let x = // ...  
3   ultimate_answer(x)?; // if `ultimate_answer` returns an error,  
4                         // `question` stops here  
5                         // and returns the error.  
6 }
```



# Error Propagation

- Code must propagate error information back to caller function after detecting it.
- One of the benefits is that your code will look cleaner by simply propagating error information back to the caller that can handle the error.
- Other benefit is that your function doesn't need extra code to propagate both the successful and unsuccessful cases.

```
1 use std::fs::File;
2 fn main() {
3     let f = File::open("hello.txt");
4     let f = match f {
5         Ok(file) => file,
6         Err(error) => panic!("Can't open the file {:?}", error),
7     };
8 }
```





# Concurrency



# Fearless Concurrency

- Message-passing concurrency, where '*channels*' send messages between **threads**
- Shared-state concurrency, where multiple between **threads** have access to same piece of data
- The *Sync* and *Send* traits, which extend Rust's concurrency guarantees to user-defined types as well as types provided by the standard library
  - To create threads to run multiple pieces of code at the same time





# Creating a New Thread

To create a new thread, we call the **thread::spawn** function and pass it a **closure** containing the code we want to run in the new thread.

```
1 use std::thread;
2 use std::time::Duration;
3
4 fn main() {
5
6     thread::spawn(|| {
7         for i in 1..10 {
8             println!("hi number {} from the spawned thread!", i);
9             thread::sleep(Duration::from_millis(1));
10        }
11    });
12
13    for i in 1..5 {
14        println!("hi number {} from the main thread!", i);
15        thread::sleep(Duration::from_millis(1));
16    }
17 }
```





# Creating a New Channel

- **Channel** has two halves: **transmitter** and **receiver**
- One thread uses transmitter to **send** data on the channel.
- Other thread uses receiver to **receive** data from the channel
- Channel is **closed** if either transmitter or receiver is **dropped**

```
1 use std::sync::mpsc;
2 use std::thread;
3
4 fn main() {
5
6     let (tx, rx) = mpsc::channel();
7
8     thread::spawn(move || {
9         let val = String::from("hi");
10        tx.send(val).unwrap();
11    });
12
13    let received = rx.recv().unwrap();
14    println!("Got: {}", received);
15 }
```





# Shared-State Concurrency

- Shared memory concurrency is like **multiple ownership**.
- Multiple threads can access same data at the same time.
- `Mutex` ensures access to shared data to only thread at a time.

```
1 use std::sync::{Arc, Mutex};  
2 use std::thread;  
3  
4 fn main() {  
5     let counter = Arc::new(Mutex::new(0));  
6     let mut handles = vec![];  
7  
8     for _ in 0..10 {  
9         let counter = Arc::clone(&counter);  
10        let handle = thread::spawn(move || {  
11            let mut num = counter.lock().unwrap();  
12            *num += 1;  
13        });  
14        handles.push(handle);  
15    }  
16    for handle in handles {  
17        handle.join().unwrap();  
18    }  
19    println!("Result: {}", *counter.lock().unwrap());  
20 }
```





# Send and Sync Traits

- The **`Send` marker** trait indicates that **ownership** of values of the type can be **transferred between** threads.
- The **`Sync` marker** trait indicates that it is safe to be **shared** between threads.
- Any type `<T>` is **Sync** if `&T` is **Send**, meaning the immutable reference can be sent safely to another thread.
- Similar to Send, primitive types are Sync, and types composed entirely of types that are Sync are also Sync.



# Send Trait

If a type implements `Send`, it is guaranteed to be safe to `move` between threads. It means that multiple threads can use a value of a type

```
1 use std::thread;
2
3 fn main(){
4
5     let data = vec![1, 2, 3];
6     thread::spawn(move || {
7         println!("{:?}", data);
8     });
9 }
```





# Sync Trait

If a type implements `Sync`, it is guaranteed to be safe to share between threads and a value of a type can be accessed in multiple threads at the same time

```
1 use std::sync::Arc;
2 use std::thread;
3
4 fn main(){
5     let data = Arc::new(vec![1, 2, 3]);
6     let data_clone = data.clone();
7
8     thread::spawn(move || {
9         println!("{:?}", data_clone);
10    });
11
12    println!("{:?}", data);
13 }
```

Use `Arc` instead of a raw pointer, it provides thread-safety and prevent data race conditions.



# Async/ Await



# What is async programming ?

A concurrent programming model



# Multitasking

- One of the fundamental features of most operating systems is multitasking, which is the ability to execute multiple tasks concurrently.

- Preemptive Multitasking

Preemptive multitasking is that the operating system controls when to switch tasks.

- Cooperative Multitasking

Cooperative multitasking lets each task run until it voluntarily gives up control of the CPU.



# Async / Await

- Built-in tool in rust programming for writing asynchronous functions.
  - **async**  
Transforms a block of code into a state machine that implements a trait called '**future**'.
  - **await**
    - Mechanism to run a future
    - It asynchronously waits for the future to complete.



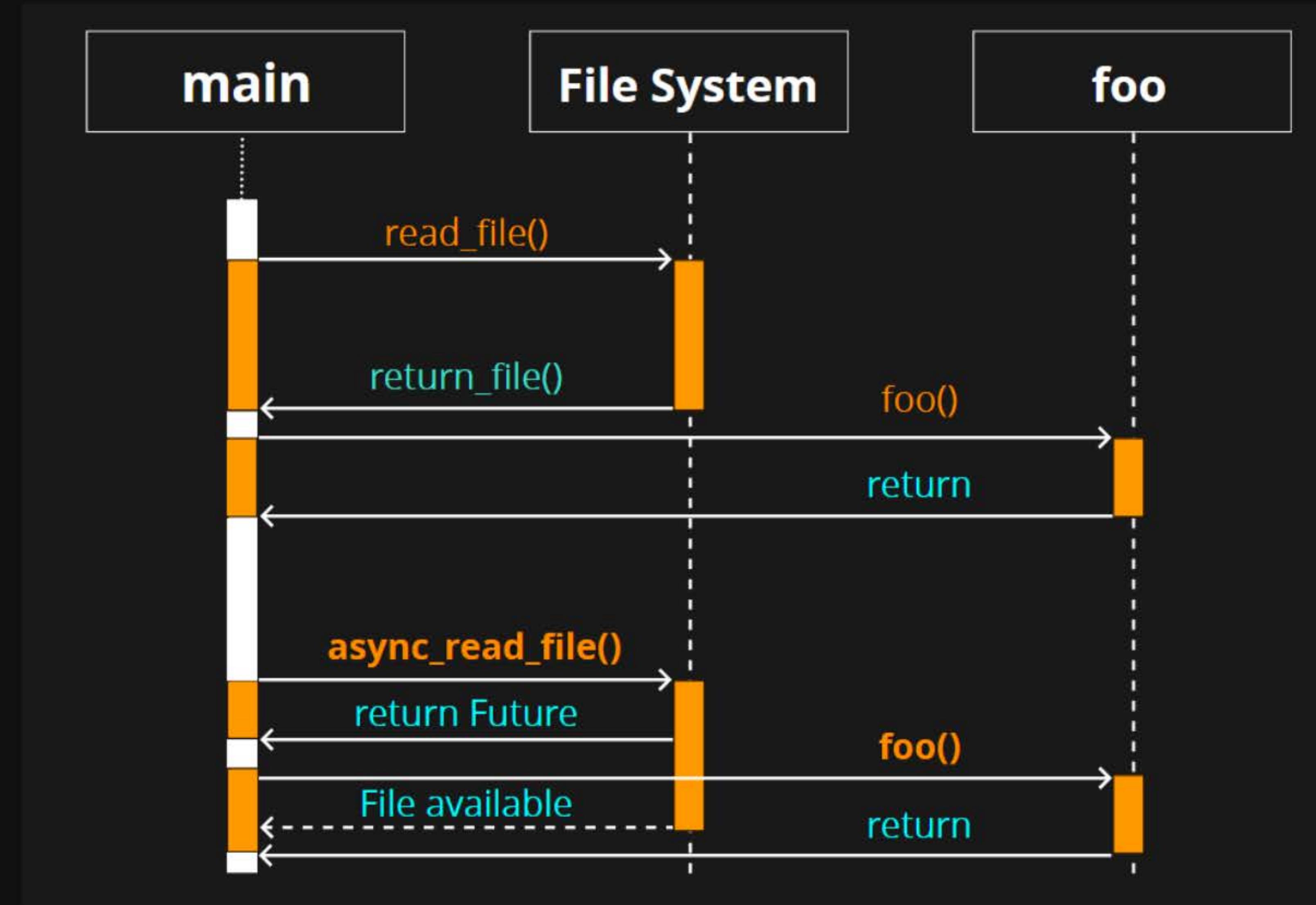
# Futures

- A future represents a value that might not be available yet.
- Futures make it possible to continue execution until the value is available.

```
1 let future = async_read_file("foo.txt");
2
3 let file_content = loop {
4
5     match future.poll(...) {
6
7         Poll::Ready(value) => break value,
8         Poll::Pending => {}, // do nothing
9
10    }
11 }
```

```
1 pub enum Poll< T > {
2     Ready(T),
3     Pending,
4 }
```

# Flow Diagram for `async`





# The `async/await` Pattern

`async/await` pattern let the programmer write code that looks like normal synchronous code, but the compiler turns it into asynchronous at compile time.

```
1 async fn foo() -> u32 {  
2     0  
3 }
```

Above code is roughly translated by the compiler to:

```
1 fn foo() -> impl Future < Output = u32 > {  
2     future::ready(0);  
3 }
```

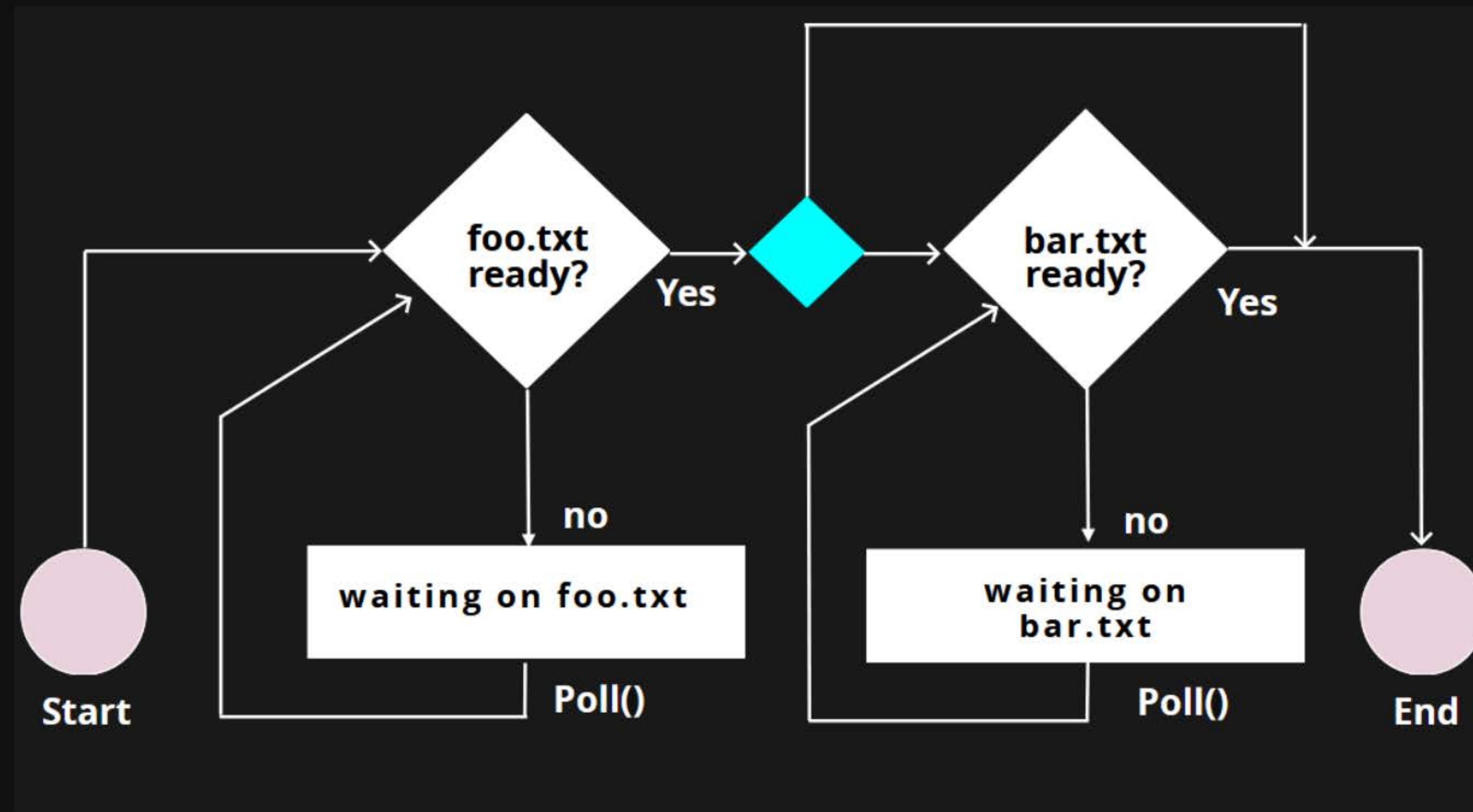


# State Machine Transformation

```
1 async fn example(min_len: usize) -> String {  
2     let content = async_read_file("foo.txt").await;  
3     if content.len() < min_len {  
4         content + &async_read_file("bar.txt").await  
5     } else {  
6         content  
7     }  
8 }
```



# State Machine Transformation





# Executors and Wakers

- Executors
  - Allow spawning futures as independent tasks, typically through some sort of 'spawn' method.
  - Responsible for polling all futures until they are completed.
  - Can Switch to a different future whenever a future returns 'Poll::Pending'.
- Wakers
  - Wakers are objects that are used to wake up a task that is blocked waiting for some event to occur.
  - Wakers are typically created by executors and passed to tasks, allowing tasks to be woken up when the relevant event occurs.



# Thank You

