Workshop #2

Learning Rust

Agenda

- Why use Rust?
- Memory safety and optimization
- Ownership
- Imports (Packages, Crates, Modules)
- Hands on!
  - HelloWorld
  - Fibonacci numbers
  - Recursive functions

Everybody 🧡 Rust!

Everybody 🧡 Rust!

But…why?

Advantages of using Rust

In a nutshell, Rust is great for its:

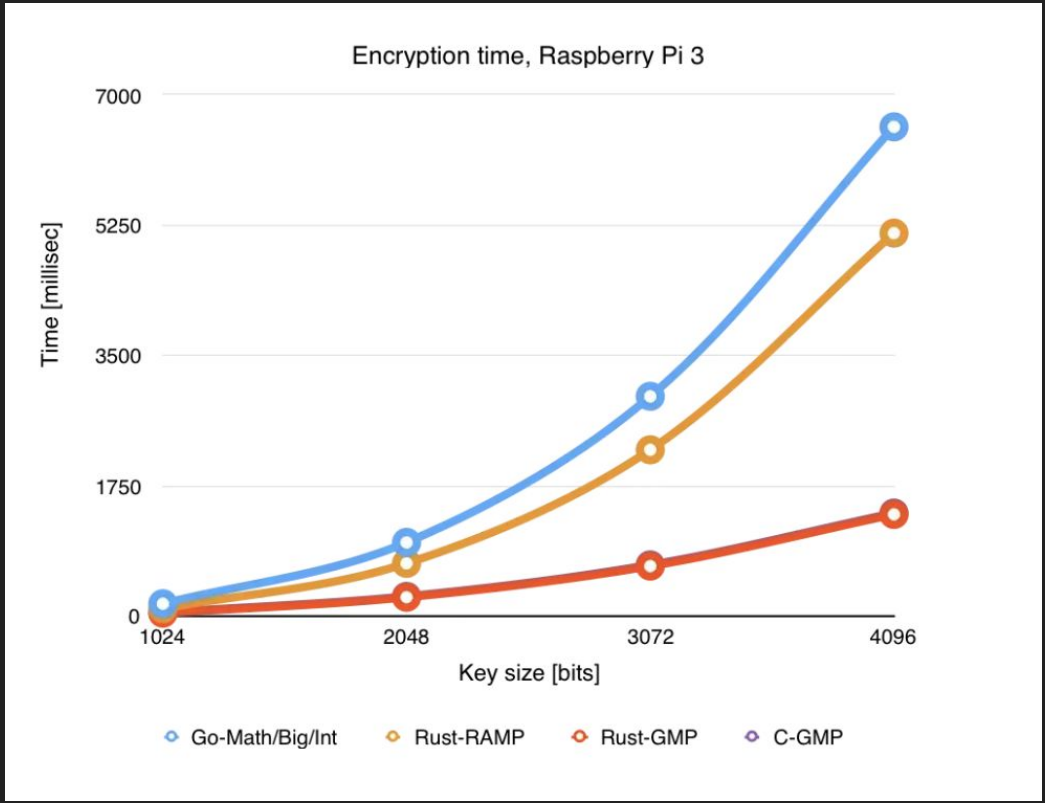
- Security
- Efficiency

Advantages of using Rust

- **You** decide how memory allocation and de-allocation work
- Direct access to hardware and memory (**no garbage collector needed**)
- The *borrow checker* helps eliminate entire classes of bugs caused by memory unsafety
- Freedom to replace pieces of code without taking memory safety risks.
- Efficient usage of memory, performant memory access and low overhead
- Bug-free approach to writing code (control over low-level details)
- High focus on Concurrency (numerous packages and libraries designed with concurrency in mind)
- Pattern on how to implement concurrency reminds Javascript (similar to “async”)
- Strong and Static typing (compile-time checks and prevents implicit conversions)

Advantages of using Rust

Just an example..



Lower is faster!

Stack and Heap

Stack:

- You can only add values to the stack with a fixed and known byte size (e.g. i8 = 8 bits)

Heap:

- Use it for unknown size variables
- Or for values that you want to easily modify in the future (i.e. adding or removing bytes)

Example

```rust
fn main() {
    let a: String = "hello".to_string();
}
```

Memory allocation is handled similarly to C++

(You store bytes in the stack and reference to values in the heap)

Ownership

Ownership is what allows Rust **not to have** a garbage collector!

3 Rules:

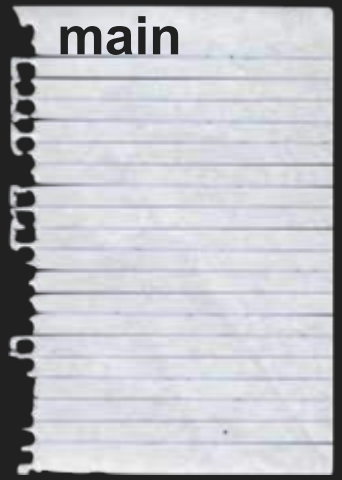
- Each value in Rust has an *owner*.
- There can only be one *owner* at a time.
- When the *owner* goes out of scope, the value will be dropped

Ownership

```rust
fn main() {
    let a: i32 = 3;
    fn_name();
    fn fn_name() {
        let a: u64 = 42;
    }
}
```

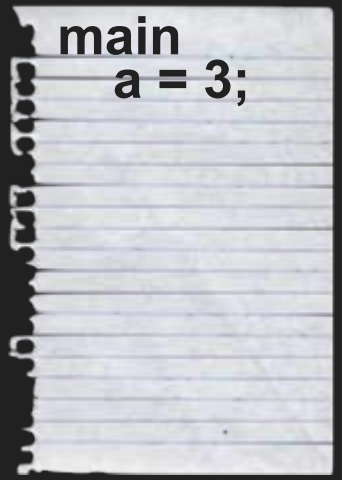
Ownership

```rust
fn main() {
    let a: i32 = 3;
    fn_name();
    fn fn_name() {
        let a: u64 = 42;
    }
}
```



Ownership

```rust
fn main() {
    let a: i32 = 3;
    fn_name();
    fn fn_name() {
        let a: u64 = 42;
    }
}
```

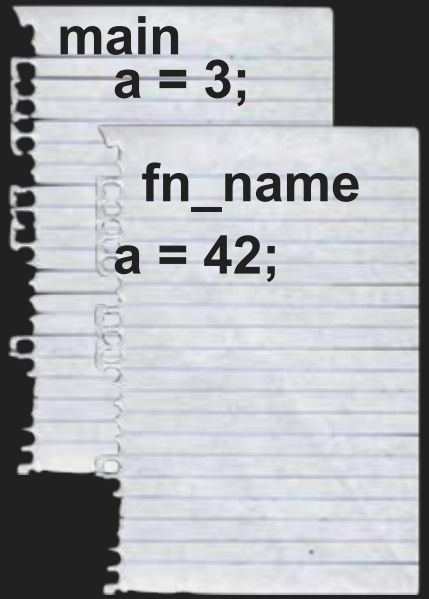


Ownership

```rust
fn main() {
    let a: i32 = 3;
    fn_name();
    fn fn_name() {
        let a: u64 = 42;
    }
}
```

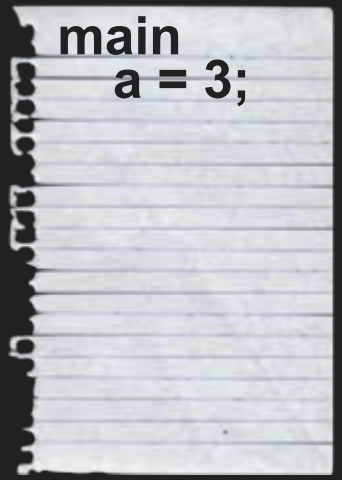
Ownership

```rust
fn main() {
    let a: i32 = 3;
    fn_name();
    fn fn_name() {
        let a: u64 = 42;
    }
}
```



Ownership

```rust
fn main() {
    let a: i32 = 3;
    fn_name();
    fn fn_name() {
        let a: u64 = 42;
    }
}
```



Ownership

```rust
fn main() {
    let a: i32 = 3;
     fn_name();
     fn fn_name() {
         let a: u64 = 42;
     }
}
```

This is valid for any closure
(e.g. also if statements and loops)

Example

```rust
fn main() {
    let x: u64 = 42;
    let a = &x;
}
```

The variable **x** owns the value **42**

We assign **x** to the variable **a, referencing** the integer **42** and making **a** the *owner*.

We end up with two **42** on the stacks, as integer is a *copy* type, which is known size, and it can be stored easily on the stack and duplicated

Borrowing / Lending

```rust
fn main() {
   let mut x: String = "hello".to_string();
   let z: &String = &x;
   let y: &String = &x;
}
```

We borrow ownership by using the **&** symbol

When the value of x is borrowed, it cannot be changed, even if mutable as **mut** promises exclusive access)

There can be infinite borrowers!

Borrowing / Lending

```rust
fn main() {
    let mut x: String = "hello".to_string();
    {
        let z: &String = &x;
        let y: &String = &x;
    }
}
```

We can also create a scope, and modify the variable there

Once z and y finish their scope, x is free to mutate itself

Example

```rust
fn main() {
    let cool_variable: String = "Area 52 is amazing".to_string();
    print_variable(cool_variable);
    print_variable(cool_variable);

    fn print_variable(cool_variable: String) {
        println!("{}", cool_variable);
    }
}
```

Example

```rust
fn main() {
    let cool_variable: String = "Area 52 is amazing".to_string();
    print_variable(cool_variable);
    print_variable(cool_variable);

    fn print_variable(cool_variable: String) {
        println!("{}", cool_variable);
    }
}
```

Example

```rust
fn main() {
    let cool_variable: String = "Area 52 is amazing".to_string();
    print_variable(cool_variable);
    print_variable(cool_variable);

    fn print_variable(cool_variable: String) {
        println!("{}", cool_variable);
    }
}
```

Example

```rust
fn main() {
    let cool_variable: String = "Area 52 is amazing".to_string();
    print_variable(cool_variable);
    print_variable(cool_variable);

    fn print_variable(cool_variable: String) {
        println!("{}", cool_variable);
    }
}
```

Example

```rust
fn main() {
    let cool_variable: String = "Area 52 is amazing".to_string();
    print_variable(cool_variable);
    print_variable(cool_variable);

    fn print_variable(cool_variable: String) {
        println!("{}", cool_variable);
    }
}
```

Example

```
fn main() {
    let cool_variable: String = "Area 52 is amazing".to_string();
    print_variable(cool_variable);
```

```
    print_variable(cool_variable);
                   -------------- value moved here
    print_variable(cool_variable);
                   ^^^^^^^^^^^^^^ value used here after move
```

```
    }
}
```

ERROR!

Example

```rust
fn main() {
    let cool_variable: String = "Area 52 is amazing".to_string();
    print_variable(&cool_variable);
    print_variable(&cool_variable);

    fn print_variable(cool_variable: &String) {
        println!("{}", cool_variable);
    }
}
```

Example

```rust
fn main() {
    let cool_variable: String = "Area 52 is amazing".to_string();
    print_variable(&cool_variable);
    print_variable(&cool_variable);

    fn print_variable(cool_variable: &String) {
        println!("{}", cool_variable);
    }
}
```

Imports

Cargo.toml

Cargo.toml

Cargo.toml

Packages

Packages are a Cargo feature that permit us build, test, and share crates.

Crates

Crates are a tree of modules that produce a library or executable.

Modules

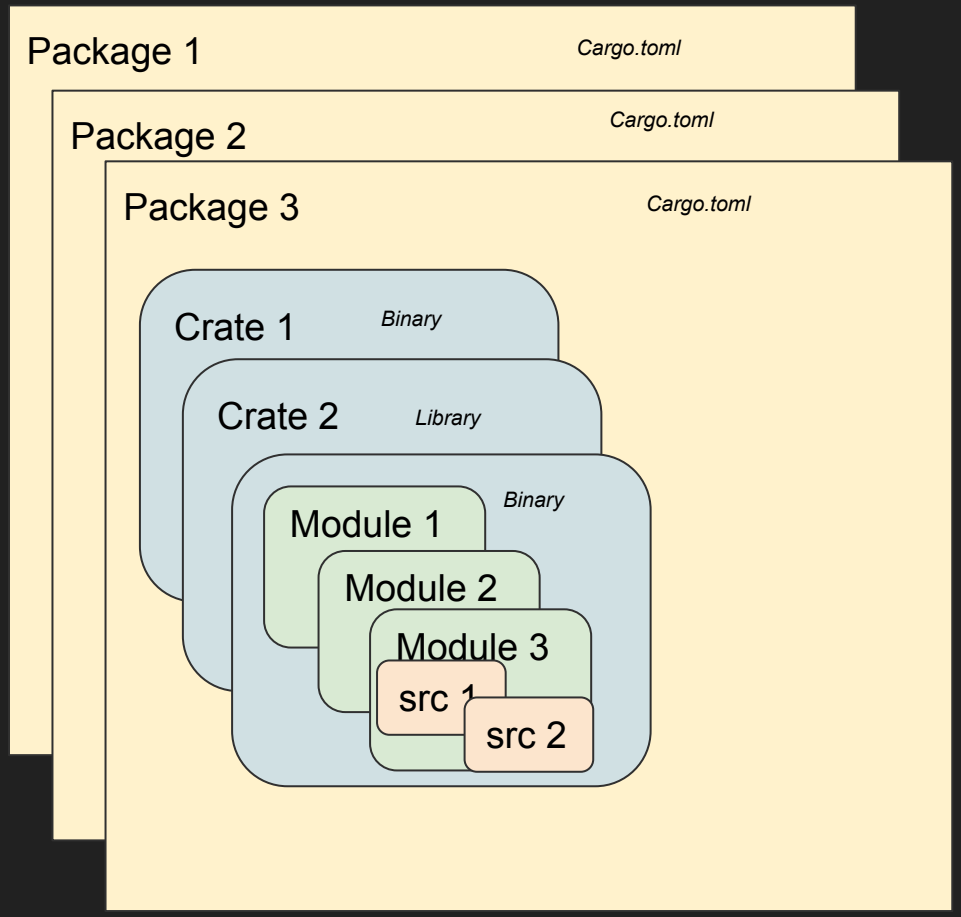
Modules allow us to control the scope and privacy of paths.

Imports

Packages

Crates

Modules



HANDS ON!

In Rust We Trust

This workshop has been provided by:

Phi Labs