

Workshop #2

Learning Rust



Agenda

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



Why Rust?

Why Rust?

Because...

Everybody  Rust!

Everybody  Rust!

But..Why?

Advantages of using Rust

In a nutshell, Rust is great for its:

- Security
- Efficiency

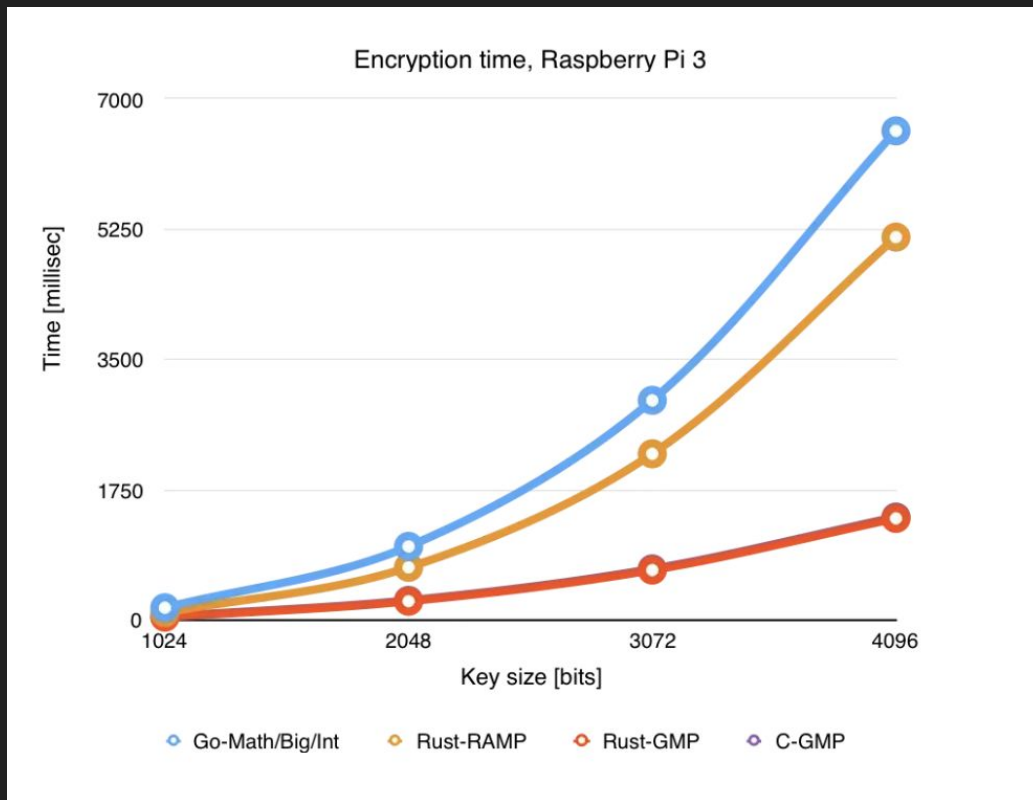
Advantages of using Rust



- **You** decide how memory allocation and de-allocation work
- Direct access to hardware and memory (**no garbage collector needed**)
- 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”)
- The *borrow checker* helps eliminate entire classes of bugs caused by memory unsafety
- 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 things 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

```
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

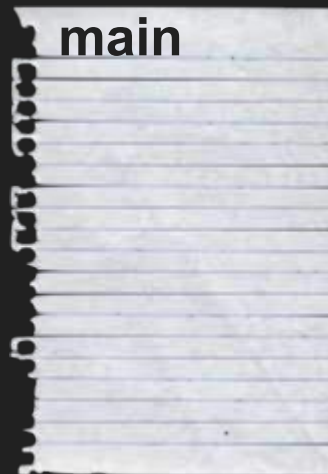


```
fn main() {  
    let a: i32 = 3;  
  
    fn_name();  
  
    fn fn_name() {  
        let a: u64 = 42;  
    }  
}
```


Ownership



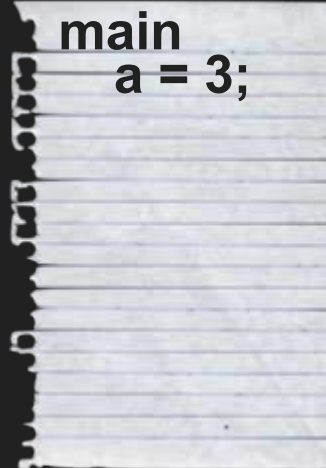
```
fn main() {  
    let a: i32 = 3;  
    fn_name();  
    fn fn_name() {  
        let a: u64 = 42;  
    }  
}
```



Ownership




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

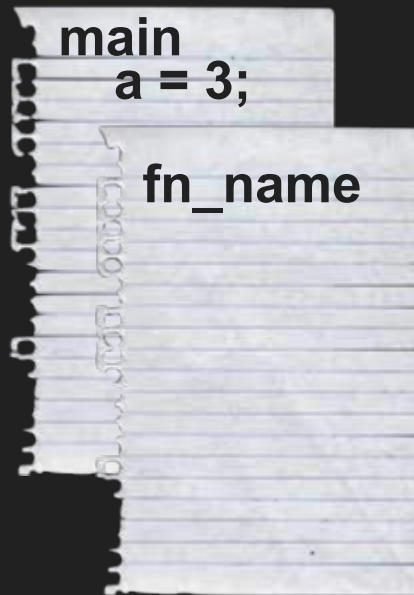


main
a = 3;


Ownership



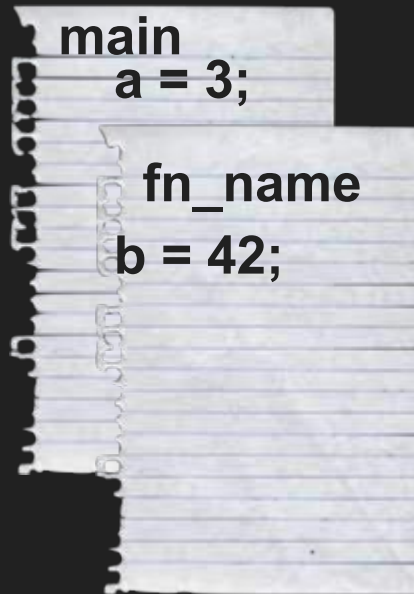
```
fn main() {  
    let a: i32 = 3;  
    fn_name();  
    fn fn_name() {  
        let a: u64 = 42;  
    }  
}
```



Ownership

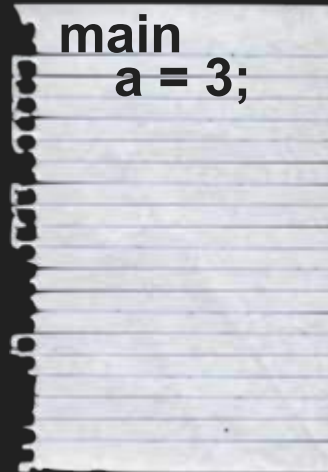


```
fn main() {  
    let a: i32 = 3;  
    fn_name();  
    fn fn_name() {  
        let a: u64 = 42;  
    }  
}
```



Ownership

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



Ownership

```
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

```
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*.

Borrowing / Lending

```
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)

Borrowing / Lending

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

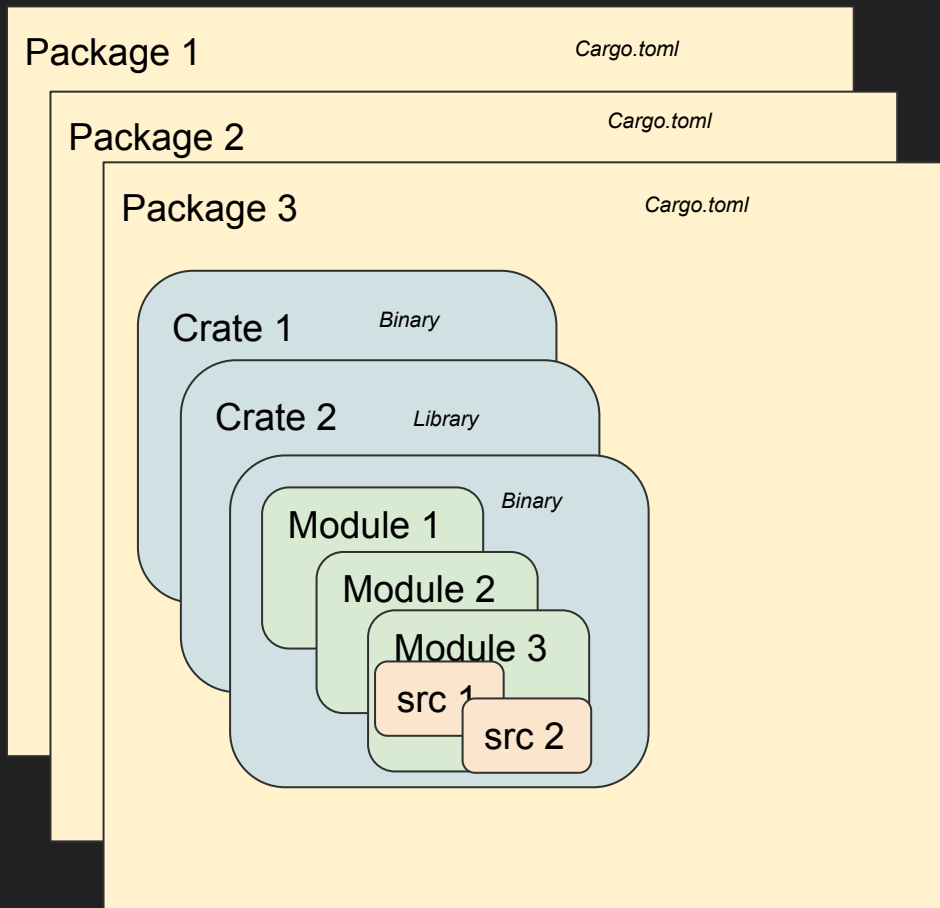
We can also create a scope, and modify the variable there

Imports

Packages

Crates

Modules



HANDS ON!



BUILD

Y U NO BUILD

memegenerator.net

In Rust We Trust



This workshop has been
provided by:

