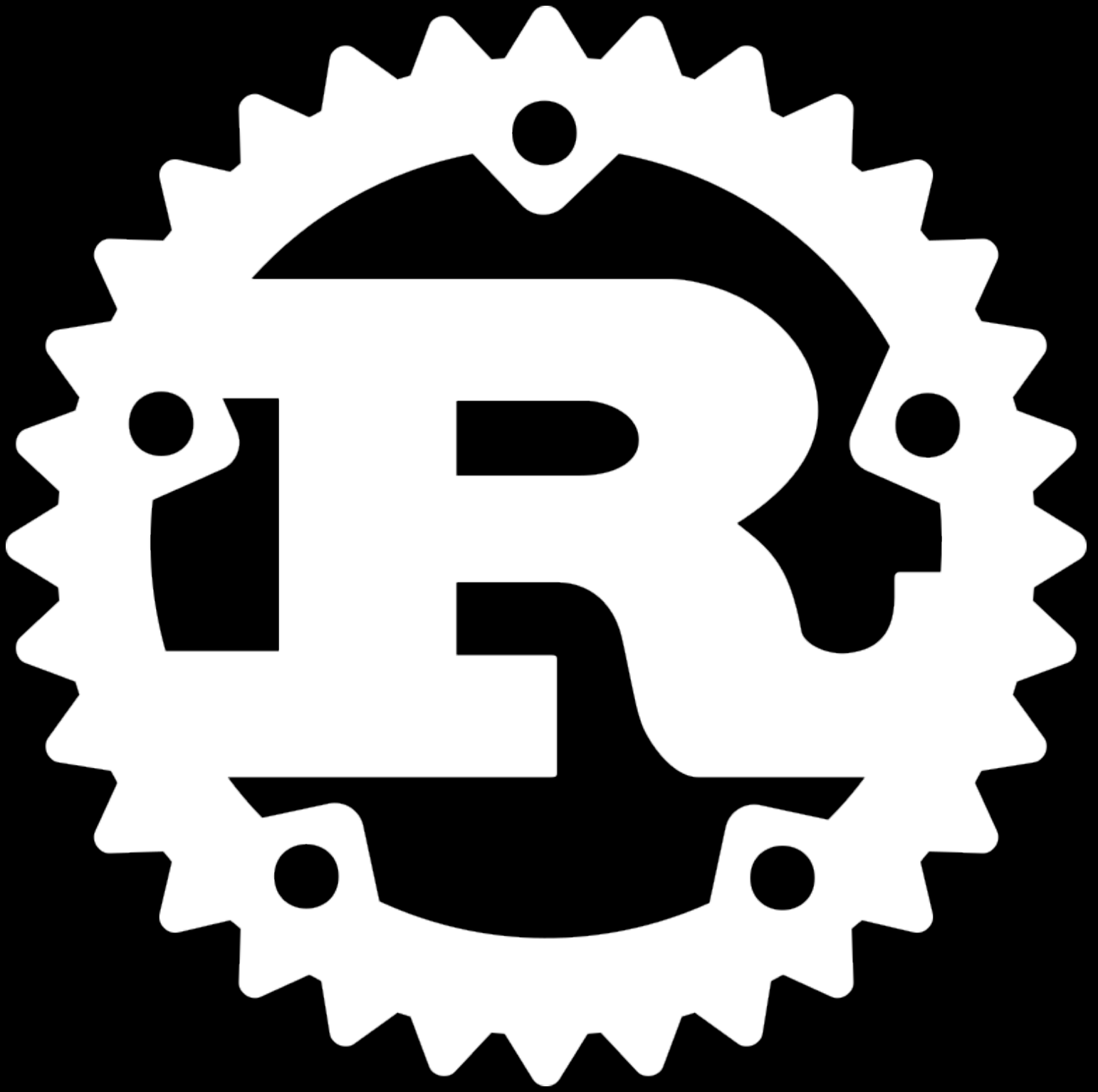

The Rust Language

Memory, Ownership and Lifetimes

- Jeff Walker "Code Ranger"
- .NET Web Developer (C#)
- Jeff@WalkerCodeRanger.com
- @WalkerCodeRangr
- WalkerCodeRanger.com
- Interested in languages
- Worked on design of own programming language
- Attended Columbus Rust Society meet up since Aug 2015

Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety.

Featuring

- zero-cost abstractions
- move semantics
- guaranteed memory safety
- threads without data races
- trait-based generics
- pattern matching
- type inference
- minimal runtime
- efficient C bindings

Challenges

- Garbage collection not acceptable
- Manual memory mangement unsafe
- Smart pointers help, but still allow programming errors
- Want control over stack vs heap

Ownership

- Enforced statically at compile time
- Memory freed at end of scope

```
struct Point
{
    x: i32,
    y: i32
}

fn main()
{
    let p = Point {x: 2, y: 4};
}
```

Move Semantics

```
let p = Point {x: 2, y: 4};
let q = p;
println!("p.x: {}", p.x);
```

```
error[E0382]: use of moved value: `p.x`
  --> <anon>:11:25
   |
10 |     let q = p;
   |         - value moved here
11 |     println!("p.x: {}", p.x);
   |                         ^^^ value used here after move
   |
   = note: move occurs because `p` has type `Point`, which
does not implement the `Copy` trait
```

Move into Function

```
fn take(p: Point)
{
    // not important
}

let p = Point {x: 2, y: 4};
take(p);
println!("p.x: {}", p.x);
```

```
error[E0382]: use of moved value: `p.x`
```

`Copy` Types

```
let x = 1;
let x2 = x;
println!("x is: {}", x);
```

Prints:

```
x is: 1
```

Limitations of Ownership

```
fn foo(p: Point) -> Point
{
    // do stuff with p

    // hand back ownership
    p
}
```

Borrowing References

```
fn distance_from_origin(p: &Point) ->
f64
{
    ((p.x*p.x+p.y*p.y) as f64).sqrt()
}

let p = Point {x: 2, y: 4};
let d = distance_from_origin(&p);
println!("d: {}", d);
println!("p.x: {}", p.x);
```

```
d: 4.47213595499958
p.x: 2
```

Immutability is the Default

```
let p = Point {x: 2, y: 4};
p.x = 5;
```

```
error: cannot assign to immutable
field `p.x`
   --> <anon>:14:5
    |
14  |     p.x = 5;
    |     ^^^^^^^
```

Immutability is the Default

```
let mut p = Point {x: 2, y: 4};
p.x = 5;
```

Mutable References

```
fn double(p: &Point)
{
    p.x *= 2;
    p.y *= 2;
}
```

```
error: cannot assign to immutable field
`p.x`

error: cannot assign to immutable field
`p.y`
```

Mutable References

```
fn double(p: &mut Point)
{
    p.x *= 2;
    p.y *= 2;
}

let mut p = Point {x: 2, y: 4};
double(&mut p);
```

References But...

```
let mut x = 5;
let y = &mut x;
*y += 1;
println!("{}", x);
```

```
error[E0502]: cannot borrow `x` as immutable because it is also borrowed as mutable
  |
4 |     let y = &mut x;
  |                 - mutable borrow occurs here
5 |     *y += 1;
6 |     println!("{}", x);
  |                     ^ immutable borrow occurs here
7 | }
  | - mutable borrow ends here
```

Rules

- Only one owner at a time
- Owner is only one that can access data (unless they lend it out)
- Ownership can be transferred (move semantics)
- Any borrow must last for a scope no greater than that of the owner
- Have one or the other of these two kinds of borrows, but not both at the same time:
  - one or more references (&T) to a resource
  - exactly one mutable reference (&mut T)
- Owner limited while borrowed

Scopes

```
fn main()
{
   let mut x = 5;
   let y = &mut x;      // + &mut borrow
   *y += 1;             // |
   println!("{}", x); // | try to borrow
                        // + borrow ends
}
```

```
error[E0502]: cannot borrow `x` as
immutable because it is also
borrowed as mutable
```

Scopes (fixed)

```
fn main()
{
  let mut x = 5;
  {
    let y = &mut x; // + &mut borrow
    *y += 1;        // |
  }                 // + borrow ends

  println!("{}", x); // borrow
}
```

Why?

- Safe data structures
- Iterator invalidation
- Use after free
- Resource mangement

Safe Data Structures

```
fn main()
{
    let v = vec![1, 2, 3];
    foo(v);
    // Wouldn't be safe to use v here,
    // stack data invalid
}

fn foo(mut v: Vec<i32>)
{
    v.truncate(2);
    // store the vector somewhere
}
```

Iterator Invalidation

```
let mut v = vec![1, 2, 3];

for i in &v { println!("{}", i); }

for i in &v
{
    println!("{}", i);
    v.push(34);
}
```

```
error[E0502]: cannot borrow `v` as
mutable because it is also
borrowed as immutable
```

Use After Free

```
let y: &i32;
{
    let x = 5;
    y = &x;
}
println!("{}", y);
```

```
error: `x` does not live long enough
 --> <anon>:5:10
  |
5 |     y = &x;
  |          ^ does not live long enough
6 | }
  | - borrowed value only lives until here
...
9 | }
  | - borrowed value needs to live until here
```

Resources

- Files
- Networking
- Instead of disposible pattern, use Resource Acquisition is Initialization (RAII) pattern
- Implement `Drop` trait
- Don't need something like C# `using` statement

Lifetimes

```
fn skip_prefix(line: &str, prefix: &str) -> &str
{
    // ...
    line
}
fn main()
{
    let line = "lang:en=Hello World!";
    let v = skip_prefix(line, "lang:en=");
    println!("{}", v);
}
```

```
error[E0106]: missing lifetime specifier
  |
1 | fn skip_prefix(line: &str, prefix: &str) -> &str
  |                                             ^ expected lifetime
parameter
  |
  = help: this function's return type contains a borrowed
Value, but the signature does not say whether it is borrowed
From line` or `prefix`
```

Lifetimes

```
fn skip_prefix<'a, 'b>
    (line: &'a str, prefix: &'b str)
    -> &'a str
{
    // ...
    line
}
```

Lifetimes in Structs

```
struct Foo<'a>
{
    x: &'a i32,
}

fn main()
{
    let y = &5;
    let f = Foo { x: y };

    println!("{}", f.x);
}
```

Lifetime Elision

- The reason we didn't have to have lifetimes in every example
- Each elided lifetime in a function’s arguments becomes a distinct lifetime parameter
- If there is exactly one input lifetime, elided or not, that lifetime is assigned to all elided lifetimes in the return values of that function.
- If there are multiple input lifetimes, but one of them is `&self` or `&mut self`, the lifetime of self is assigned to all elided output lifetimes.

Elision Examples

```
fn print(s: &str);
fn print<'a>(s: &'a str);

fn debug(lvl: u32, s: &str);
fn debug<'a>(lvl: u32, s: &'a str);

fn substr(s: &str, until: u32) -> &str;
fn substr<'a>(s: &'a str, until: u32) -> &'a str;

fn get_str() -> &str; // ILLEGAL, no inputs

// ILLEGAL, two inputs
fn frob(s: &str, t: &str) -> &str;
fn frob<'a, 'b>(s: &'a str, t: &'b str) -> &str;
```

Elision Examples

```
fn get_mut(&mut self) -> &mut T;
fn get_mut<'a>(&'a mut self) -> &'a mut T;

fn args<T: ToCStr>(&mut self, args: &[T])
    -> &mut Command;
fn args<'a, 'b, T: ToCStr>(&'a mut self, args: &'b
[T]) -> &'a mut Command;

fn new(buf: &mut [u8]) -> BufWriter;
fn new<'a>(buf: &'a mut [u8]) -> BufWriter<'a>;
```

Summary

- Use Rust for low level programming (instead of C/C++)
- Borrow Checker provides safe memory and resource management
- The rules of the borrow checker make sense in other languages

Dyon

- Dynamically typed scripting language, designed for game engines and interactive applications

```
// `a` outlives `b`
fn put(a: 'b, mut b) {
    b[0] = a
}

fn main() {
    a := [2, 3]  // - lifetime of `a`
                 // |
    b := [[]]    // |  - lifetime of `b`
                 // |  |
    put(a, mut b)// |  |
}
```

Adamant

```
public async Main
       (console: mut Console, args: string[]) -> Promise
{
  let results = mut new List<~own Promise<int>>();

  foreach(let file in args)
  {
    console.WriteLine("Begin processing file: "
                                              + file);
    results.Add(ProcessFile(file));
  }

  let lengths = await Promise.WhenAll(results);
  Console.WriteLine("Total Length of File(s): "
                                        + lengths.Sum());
}
```

Adamant

```
public async ProcessFile
             (fileName: string) -> Promise<int>
{
  let file =
            mut new FileReader.Open(fileName);
  let contents = await file.ReadToEnd();

  // simulate work
  await Promise.Wait(
                       new TimeSpan.Seconds(5));
  return contents.Length;
}
```

Questions?

Resources

- rust-lang.org
- doc.rust-lang.org/book
- rustbyexample.com
- www.piston.rs/dyon-tutorial
- adamant-lang.org

Contact Me

- Jeff Walker "Code Ranger"
- Jeff@WalkerCodeRanger.com
- @WalkerCodeRangr
- WalkerCodeRanger.com