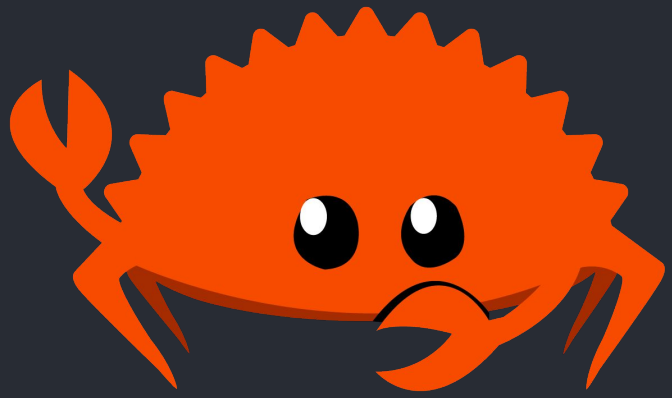
an intro to rust

jacob hempel



Ferris, the rustacean

- topics:
  - what is rust?
  - syntax basics
  - rust goodies
  - ownership
  - spooky stuff

- resources:
  - https://doc.rust-lang.org/book/
  - https://doc.rust-lang.org/rust-by-example/

if you want to follow along:

- Linux/Mac:
  - Go to https://www.rust-lang.org/tools/install and copy the command into your terminal
  - Follow on screen instructions
  - $ source $HOME/.cargo/env
- Windows: (oh boy)
  - You'll need Visual Studio installed (or MinGW, if that's your thing)
  - Go to https://www.rust-lang.org/tools/install and follow the instructions
  - Cargo commands should work in Command Prompt or Powershell, you might need an editor like Atom or VSCode to write code

what is rust?

- rust is a compiled language made by the Mozilla Foundation
  - fast: faster than everything except raw Assembly and sometimes C
    - no runtime and no garbage collection
    - rust does heavy optimizations at compile time to build some seriously fast executables
    - zero-cost abstractions: rust libraries provide high level abstractions that run just as fast as the “manual” versions
  - safe: rust guarantees that your program cannot have undefined behavior - this is great for security and stability
    - more on this soon!
  - modern: rust’s toolchain is powerful and easy to use - and it’s the default!
    - cargo is a compiler and package manager all in one!
    - smart, helpful compiler messages

who’s using rust?

- Firefox’s new Servo rendering engine
- Cloudflare
- Dropbox
- Chucklefish (maker of Starbound & publisher of Stardew Valley)
- Ceph (enterprise storage)
- Canonical (Ubuntu)

quick intro to cargo

- Make a new directory/project with:
  - `cargo new --bin <projectName>`
- Go into the directory
  - `cd <projectName>`
- Edit the files in the src/ directory
  - `<editor> src/main.rs`
- Run your project!
  - `cargo run`

```
rust types:
    - integers: i8, i16, i32, i64, i128, isize
    - unsigned: u8, u16, u32, u64, u128, usize
    - boolean: bool
    - floats: f32, f64
    - strings: str, String
```

```rust
fn mutability() {
    // rust gets its safety by enforcing mutability
    //      and ownership
    // variables are immutable by default,
    //      but you can add the mut keyword to
    //      make them mutable

    let x = 1;          // type is inferred
    let y :i32 = 2; // explicit typing

    x = x + 1;          // what's wrong here??

}
```

```
error[E0384]: cannot assign twice to immutable variable `x`
  --> src/main.rs:42:5
   |
39 |     let x = 1;       // type is inferred
   |         -
   |         |
   |         first assignment to `x`
   |         help: make this binding mutable: `mut x`
...
42 |     x = x + 1;       // what's wrong here??
   |     ^^^^^^^^^ cannot assign twice to immutable variable
```

```
fn mutability() {
    // fix the error just like the compiler says to
    // and it works!

    let mut x = 1;
    let y :i32 = 2;

    x = x + 1;          // no more red squiggles!

}
```

```
// more syntax

// it's a lot like C/C++
// see, comments are like this

/* you can also do these ones */

fn main() {
    for i in 0..10 {
        println!("{}", i);
    }
}
```

```rust
// functions look like this
// fn name(arg: type) -> return type
fn do_stuff(mut x: i32) -> i32 {
    x += 1;
    x          // implicit return
}
fn other_stuff() {
    let mut x = 0;
    while x < 10 {
        x = do_stuff(x);
    }
}
```

```rust
#[derive(Debug)]
struct Point {
    x: f64,
    y: f64,
}
impl Point {
    pub fn new(nx: f64, ny: f64) -> Point {
        let p = Point {
            x: nx,
            y: ny,
        };
        p
    }
    pub fn mv(&mut self, dx: f64, dy: f64) {
        self.x += dx;
        self.y += dy;
    }
}
```

```rust
fn point_stuff() {
    let mut p1 = Point::new(1.0, 1.0);
    println!("p1 starts @ {:?}", p1);
    // PRINTS: p1 starts @ Point {x: 1.0, y: 1.0}

    p1.mv(3.1, -2.6);
    println!("p1 mvd to: {:?}", p1);
    // PRINTS: p1 mvd to: Point {x: 4.1, y: -1.6}

    let mut p2 = Point::new(2.0, 2.0);
    p2.mv(2.0, -2.0);
}
```

```
// imagine we're making a networking library
enum IpAddrKind {
    V4,
    V6,
}
struct IpAddr {
    kind: IpAddrKind,
    address: String,
}
fn enums() {
    let ipv4 = IpAddrKind::V4;
    let ipv6 = IpAddrKind::V6;
    let loopback = IpAddr{
        kind: ipv4,
        address: String::from("127.0.0.1"),
    };
}
```

```rust
// rust will allow us to associate data with
// enum variants
#[derive(Debug)]
enum IpAddr {
    V4(i8, i8, i8, i8),
    V6(String),
}
fn enums() {
    let loopback = IpAddr::V4(127, 0, 0, 1);
    let someV6addr = IpAddr::V6(String::from("ff::01"));
    println!("loopback is: {:?}", loopback);
    println!("some addr is: {:?}", someV6addr);
}
```

```rust
// rust has a few special enums
// Option is an especially useful one
enum Option<T> {
    Some(T),
    None,
}

// you can use unwrap() to unpack an option
let my_option = unreliable_funtion();
let result = my_option.unwrap();

// unwrap will return a value of type T if Option is Some
// but it will panic! if the Option is None
// panic!ing is a lot like throwing an exception in C++
// and it will crash the program unless handled
```

```rust
fn divide(x: f64, y: f64) -> Option<f64> {
    if y != 0.0 {
        Some(x / y)
    } else {
        None
    }
}
fn unwrap_math() {
    // dont have to add explicit typing here
    let val1: f64 = divide(10.0, 1.0).unwrap();
    let val2: f64 = divide(10.0, 0.0).unwrap();
    println!("val1: {}, val2: {}", val1, val2);
}
```

```rust
fn divide(x: f64, y: f64) -> Option<f64> {
    if y != 0.0 {
        Some(x / y)
    } else {
        None
    }
}
fn match_math() {
    let opt1 = divide(10.0, 1.0);
    match opt1 {
        Some(x) => println!("result of division is: {}", x),
        None => println!("ya can't divide by zero!!"),
    }
    let opt2 = divide(10.0, 0.0);
    match opt2 {
        Some(x) => println!("result of division is: {}", x),
        None => println!("ya can't divide by zero!!"),
    }
}
```

safety at compile time

- rust uses an ownership paradigm to protect your program's data - in short, rust data must follow these three rules:
  - each value in rust has a variable that's called its owner
  - there can only be one owner at a time
  - when the owner goes out of scope, the value will be dropped

ownership

to talk about ownership, we have to briefly talk about memory - data for your program can be in two places, the stack and the heap

the stack:

- function calls and their local data are placed on the stack
- accessed directly
- types with fixed size, like integers, bools, and chars

the heap:

- used for large blocks of allocated memory
- accessed via pointers
- types with variable size, like strings and vectors

```rust
fn ownership_1() {
    let mut x = 3;   // x is a "binding" for the value (which is 3)
    let mut y = x;   // what happens here? is x copied or moved?
    x += 3;
    y += 2;
    println!("x is: {} \ny is: {}", x, y);

    let mut v1 = Vec::new();
    v1.push(1);
    v1.push(2);
    let v2 = v1;
    println!("v1 is: {:?} \ny is: {:?}", v1, v2);
}

```

let's see what the compiler has to say about it...

```
→ learnrust git:(master) ✗ cargo check
    Checking learnrust v0.1.0 (/home/jacob/Dropbox/Code/rust/learnrust)
error[E0382]: borrow of moved value: `v1`
  --> src/main.rs:62:42
   |
58 |     let mut v1 = Vec::new();
   |         ------ move occurs because `v1` has type `std::vec::Vec<i32>`, which does not implement the `Copy` trait
...
61 |     let v2 = v1;
   |              -- value moved here
62 |     println!("v1 is: {:?} \ny is: {:?}", v1, v2);
   |                                          ^^ value borrowed here after move

error: aborting due to previous error
```

okay, what's all this mean?

because a vector can expand and contract indefinitely, it can't live on the stack, because the stack's structure must be known at compile time

therefore, a "vec" object in rust is really just a pointer to a heap allocated block of memory

so, if you take a shallow copy of a vec object (like you do for integers) then the line: "`let mut v2 = v1;`" would break the second rule of ownership - there would be two owners of the heap allocated vector

in rust, we say that v1 has been moved into v2 - v2 now owns the vector, so v1 is essentially dead

```
→ learnrust git:(master) ✗ cargo check
    Checking learnrust v0.1.0 (/home/jacob/Dropbox/Code/rust/learnrust)
error[E0382]: borrow of moved value: `v1`
  --> src/main.rs:62:42
   |
58 |     let mut v1 = Vec::new();
   |         ------ move occurs because `v1` has type `std::vec::Vec<i32>`, which does not implement the `Copy` trait
...
61 |     let v2 = v1;
   |              -- value moved here
62 |     println!("v1 is: {:?} \ny is: {:?}", v1, v2);
   |                                          ^^ value borrowed here after move

error: aborting due to previous error
```

so v1 has been moved into v2 - “borrow of moved value: v1”

but what the hell is a borrow? And what’s a ‘Copy’ trait

references and borrowing

references in rust work a lot like smart pointers in C++, but naturally they have a rust-ic style

references are variables that refer to another object (much like a pointer in C/C++)

a reference does not take ownership of the thing it references, but references have their own rules:

- you can have one mutable reference to a value
- OR you can have any number of immutable references to a value
- but not both!!

references and borrowing (continued)

rules like this might seem annoying (and they can be, trust me), but these rules allow rust to prevent undefined behavior

one of the biggest problems with parallel programming is something called a data race, where two threads try and access the same data value, at the same time, and end up messing up the other thread

rust's reference rules make data races impossible

```rust
fn references_1() {
    let mut v = vec![1, 2, 3];
    let r1 = &mut v;
    let r2 = &mut v;
    println!("{:?}, {:?}", r1, r2);
}
fn references_2() {
    let mut v = vec![1, 2, 3];
    let r1 = &v;
    let r2 = &v;
    let r3 = &mut v;
    println!("{:?}, {:?}, {:?}", r1, r2, r3);
}
```

references and borrowing (continued)

when a reference is used to access or mutate a value, the reference is said to be “borrowing” the value

```
→ learnrust git:(master) ✗ cargo check
    Checking learnrust v0.1.0 (/home/jacob/Dropbox/Code/rust/learnrust)
error[E0382]: borrow of moved value: `v1`
  --> src/main.rs:62:42
   |
58 |     let mut v1 = Vec::new();
   |         ------ move occurs because `v1` has type `std::vec::Vec<i32>`, which does not implement the `Copy` trait
...
61 |     let v2 = v1;
   |              -- value moved here
62 |     println!("v1 is: {:?} \ny is: {:?}", v1, v2);
   |                                          ^^ value borrowed here after move

error: aborting due to previous error
```

so now we're (mostly) good with borrows and moves, but what's this bit about Copy traits?

traits

traits are attributes you can attach to a type that gives the type implicit and/or explicit behavior

there are numerous built-in traits:

- `Display` and `Debug`
- `Add`, `Sub`, `Neg`, `Mul`, `Div`, `Rem`
- `Not`, `BitAnd`, `BitOr`, `BitXor`, `Shl`, `Shr`
- `Copy` and `Clone`
- and many more

traits (continued)

- many of those traits provide implicit behavior
  - if a type has Display, you can print it with “{}” in a println!() macro
  - if a type has Debug, you can print it with “{:?}” in a println!() macro
  - if two types both have Add, then they can be added together with +
- this allows rust to imitate C++’s operator overloading
  - you could design a BigInt struct that uses vectors to store very large numbers
  - implement the Add trait for it, and then add your BigInts together!
    - let sum = big1 + big2;
- generics can also require that the type T has certain traits
  - for example, if you are making a data structure that requires that you add up your member data, your data structure could require that T has the Add trait

Copy & Clone

Copy:

- If a variable is on the right side of an assignment, and the variable's type has Copy, then the value will be copied to a new variable - otherwise, a Move will occur
- Most simple types (numbers/chars) have Copy, but more complex types (structs, vectors, Strings) do not

Clone:

- If a type has the Clone trait, a call to .clone() will produce a deep copy of the object
- Most types have Clone implemented (or it can be #derived

```rust
fn ownership() {
    let mut x = 3;        // since x is an integer, and integers have the Copy trait
    let mut y = x;        // this line will cause y to be a distinct copy of x
    x += 3;
    y += 2;
    println!("x is: {} \ny is: {}", x, y);

    let mut v1 = Vec::new();
    v1.push(1);
    v1.push(2);
    let v2 = v1.clone(); // however, vectors dont have the Copy trait, so you have to
                         // use .clone() to create a copy of one
    println!("v1 is: {:?} \ny is: {:?}", v1, v2);
}
```

spooky stuff

rust has so many interesting, sometimes unsafe features and keywords that there used to be a second book, the “rustnomicon,” which covered some of the more fringe stuff, like:

- the `unsafe` keyword, which turns off some borrow-checking
- memory tools (`Box<T>`, `Rc<T>`, `Cell<T>`, `RefCell<T>`, `Arc<T>`, etc)
- closures, threading, shared-state, pipes, etc
- lifetimes
- macros

most of this can now be found in the later chapters of the regular book, which I encourage you to read if you’re curious!!

thanks so much!!

if we have any time left, you can try some of these problems in rust:

- Tic tac toe
- See if a string is a palindrome (hard more - a number)