Rust for the Rest of Us

LOPSA ETENN - 12/6/16

Nolan Davidson

Chef Software

Goals

What is Rust?

Talk about things that are awesome

Talk about challenges

Look at some tooling

Look at ways to dive in

~~Learn Rust~~

From the website…

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

www.rust-lang.org

Disclaimer

The following properties apply to pure, safe Rust.

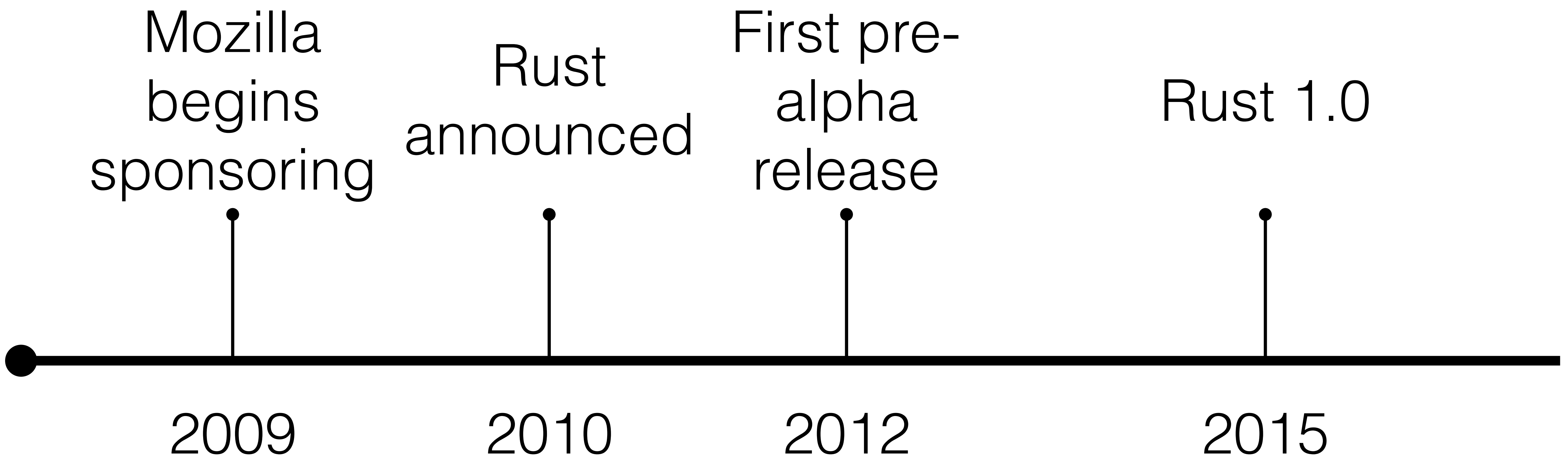
You can write *unsafe* Rust, which voids these guarantees.

Currently, many libraries are bindings to C libraries.

…so, what is it?

* Compiled
* Memory safe
* Not garbage collected
* Thread safe
* Multi-paradigm
* Focused on zero cost abstractions

History of Rust



Zero cost abstractions

Refers to run-time cost.

Allows us to have nice, higher-level features like traits and type inference in a low level language.

Memory management compiles down to code that looks like C, with memory allocations and deallocations.

Ownership

Rust is memory safe without garbage collection.

This is an example of a zero-cost abstraction.

The Rust compiler will enforce ownership at compile time, the run time performance will be as if we had allocated and deallocated every thing manually.

These rules also apply to threaded operations.

Ownership

Variable bindings have “ownership” of the thing they are bound to.

```
let name = “Nolan".to_string(); // Binds value to “name”
```

The ownership system requires that we have only one binding to any given resource.

```
let my_name = name; // Binds value of “name” to “my_name”.
```

Accessing “name” after this point will result in an error at compile time.

Ownership

```rust
fn main() {
    let string = "Hello world!".to_string();
    let big_string = make_it_big(string);
    println!("Original string: {}", string);
    println!("Uppercase string: {}", big_string);
}

fn make_it_big(s: String) -> String {
    s.to_uppercase()
}
```

Looks good, right?

```
cargo run
   Compiling rust-examples v0.1.0 (file:///Users/nolan/source/rust-talk/examples)
error[E0382]: use of moved value: `string`
 --> src/main.rs:4:37
  |
3 |     let big_string = make_it_big(string);
  |                                  ------ value moved here
4 |     println!("Original string: {}", string);
  |                                     ^^^^^^ value used here after move
  |
  = note: move occurs because `string` has type `std::string::String`, which does not implement the `Copy` trait

error: aborting due to previous error
```

References

If we use references, we can borrow the values rather than changing ownership.

```
fn main() {
    let string = "Hello world!".to_string();
    let big_string = make_it_big(&string);
    println!("Original string: {}", string);
    println!("Uppercase string: {}", big_string);
}

fn make_it_big(s: &String) -> String {
    s.to_uppercase()
}
```

The & operator denotes that we are passing and consuming a reference.

Copy types

Some types implement the Copy trait. Copy types will create a copy of the data instead of passing ownership.

Integers implement the Copy trait.

```
fn main() {
    let i: i32 = 10;
    let double_i = double_it(i);
    println!("i is {}", i);
    println!("i doubled is {}", double_i);
}

fn double_it(i: i32) -> i32 {
    i * 2
}
```

Immutability is Standard

Variable bindings are immutable by default.

```
fn main() {
    //  Error
     let i = 10;
     i = 20;

    // Correct
    let mut i = 10;
    i = 20;
}
```

This is great for keeping track of where you are mutating state.

Is it OO?

Rust does not currently implement a true object oriented system.

We use structs combined with traits to achieve some OO objectives.

This is system is focused on composition, rather than inheritance. There is an open RFC about adding inheritance, so stay tuned.

Structs and Impls

```rust
struct Person {
    name: String,
    age: i16,
}

impl Person {
    fn new(name: String, age: i16) -> Person {
        Person {
            name: name,
            age: age,
        }
    }

    fn greet(&self) { println!("{} says hello!", self.name); }
}

fn main() {
    let mut p = Person::new("Eddie".to_string(), 42);
    println!("{} is {} years old.", p.name, p.age);
    p.greet();
}
```

Traits

```rust
struct Circle { radius: f64 }
struct Square { side: f64 }

trait Shape {
    fn area(&self) -> f64;
    fn perimter(&self) -> f64;
}

impl Shape for Circle {
    fn area(&self) -> f64 { std::f64::consts::PI * (self.radius * self.radius) }
    fn perimter(&self) -> f64 { 2.0 * std::f64::consts::PI * self.radius  }
}

impl Shape for Square {
    fn area(&self) -> f64 { self.side * self.side }
    fn perimter(&self) -> f64 { self.side * 4.0 }
}

fn print_shape<T: Shape>(shape: T) {
    println!("Area is {}", shape.area());
}

fn main() {
    let c = Circle { radius: 1.0 };
    let s = Square { side: 2.0 };
    print_shape(c);
    print_shape(s);
}
```

Error handling

The preference is to handle errors by return values, rather than exceptions.

The Rust standard library provides two built in types for doing this: Option and Result.

Option returns either Some() or None(), to allow for possibility of no value. Result expands on this by returning either Ok() or Err(), taking into account error conditions.

Error handling

```rust
use std::fs::File;
use std::path::Path;
use std::io::prelude::*;

fn main() {
    let path = Path::new("myfile.txt");

    let mut file = match File::open(&path) {
        Err(e) => panic!("Could not open file!"),
        Ok(file) => file,
    };

    let mut s = String::new();
    match file.read_to_string(&mut s) {
        Err(e) => println!("couldn't read the file!"),
        Ok(_) => println!("file contains \n{}", s),
    };

}
```

Testing batteries included!

Tests can be embedded along side the code being tested, or in a separate directory.

Embedded tests function as unit-style tests. Great for testing small pieces of functionality.

Tests placed under the tests/ directory are integration-style tests. These are often used to test library code, as they allow you to consume your crate as anyone else would.

Tooling

crates.io

Repository for Rust packages (crates)

Crates are shared as source.

Used by the cargo tool.

Cargo

Package manager

Build tool

Cargo

cargo new - generates a new project
cargo build - compiles the current project
cargo run - compiles and run the current project
cargo publish - publish crate to crates.io
cargo test - run tests

Cargo

Cargo.toml

```
[package]
name = "my_app"
version = "0.1.0"
authors = ["Nolan Davidson <ndavidson@chef.io>"]

[dependencies]
rand = 0.3.0
```

Rustup

Tool for managing multiple versions of Rust.

Great for testing against multiple versions and experimenting with nightly build features.

Other tools

rustfmt - Apply Rust style
racer - Code completion
clippy - Linting
Various editor plugins.

Who’s using it?

Mozilla - Servo browser engine
Chef - Habitat, command line tools
Dropbox - Parts of their file storage system

Where to learn more

https://www.rust-lang.org
https://doc.rust-lang.org/book/ (the book)
Programming Rust (O'Reilly book)
http://rustbyexample.com/