Rust in Peace

Balaji Sivaraman (@balajisivaraman)

ThoughtWorks

June 28, 2018

About Me

- Primarily worked on Java/Spring/ROR stack in ThoughtWorks, writing microservices
- Pure functional programming advocate in languages like Scala/Haskell/Purescript
- Bitten by the Rust bug last year after reading a post on how it enabled Firefox's superior performance
- Currently on the way to transitioning from an applications developer to a systems programmer, thanks primarily to Rust

Agenda

- Introduction
- Ground Rules
- Systems Programming
- The Rust Programming Language
- Ownership and Borrowing
- Why Rust?
- Questions?

Ground Rules

What this talk is about?

- How Rust benefits newcomers to systems programming?
- What makes the Rust language unique?
- What benefit could I, as a web programmer, get from it?

What this talk is not about?

- Convince you to stop using your favourite PL and start using Rust

Systems Programming

Systems Programming

- What is Systems Programming?
- Why is it different from web/application programming?

What is Systems Programming?

From O'Reilly's Programming Rust [1]:

*Systems programming is **resource-constrained** programming. It is programming when every byte and every CPU cycle counts.*

What does that mean?

- Programmer can almost never trade-off on performance
- No GC
- Minimal/No Runtime
- Zero-Cost Abstractions [4]

Zero Overhead Abstractions

From Stroustrup [4]:

What you don't use, you don't pay for. And further: What you do use, you couldn't hand code any better.

The Rust Programming Language

Can you make sense of this?

```
pub fn main() {
  let v = vec!(1,2,3);
  let numbers: Vec<i32> = v.iter().map(|n| n * n).collect();
  println!("{:?}", numbers);
}
```

Rust Benefits

- High-Level Syntax (similar to Ruby or Java)
- Low-Level Performance (similar to C/C++)

Rust ❤ TDD

```rust
pub fn add_one(a: u32) -> u32 {
  a + 1
}

#[test]
fn test_add() {
  let result = add_one(1);
  assert_eq!(result, 2);
}
```

Running tests is straightforward

```
$ cargo --test sample.rs
$ ./sample
running 1 test
test test_add ... ok
test result: ok. 1 passed; 0 failed; 0 ignored;
0 measured; 0 filtered out
```

Mimic OO

Let's define a simple Trait!

```
trait Animal {
  fn walk(&self);
}
```

Mimic OO

Let's implement it!

```rust
struct Cat {
  name: String
}

impl Animal for Cat {
  fn walk(&self) {
    println!("{} walks like a cat", self.name);
  }
}
```

Mimic OO

Let's implement it again!

```
struct Dog {
  name: String
}

impl Animal for Dog {
  fn walk(&self) {
    println!("{} walks like a dog", self.name);
  }
}
```

Mimic OO

What is the output?

```
fn main() {
  let d = Dog { name: String::from("Snuggles") };
  let c = Cat { name: String::from("Puss in Boots") };
  d.walk();
  c.walk();
}
```

And the output

```
$ rustc sample.rs
$ ./sample
Snuggles walks like a dog
Puss in Boots walks like a cat
```

Key Language Features

- Functional Features
  - ENums, Pattern Matching and Algebraic Data Types
  - Lazy Iterators
  - Functions as first class values
- OO-Like Features
  - Traits and Implementations
  - Trait Bounds and Trait Objects
- Rust Lang Features
  - **Ownership and Borrowing**
  - Lifetimes
  - Unit Testing primitives as part of the core language
  - Concurrency Primitives — Threads, Channels, Atomic Values etc.

Ownership and Borrowing

A Tale of Three Programs

```
def main
a = [1, 2, 3]
puts a
end

main
```

A Tale of Three Programs

This is not representative of actual C code (:-P)

```c
#include<stdio.h>

int main() {
  int *ptr = malloc(sizeof(int) * 3);
  ptr[0] = 1;
  ptr[1] = 2;
  ptr[2] = 3;
  for(int i = 0; i < 3; i++) {
    printf("%d", ptr[i]);
  }
  free(ptr);
  return 1;
}
```

A Tale of Three Programs

```
pub fn main() {
  let v = vec!(1,2,3);
  println!("{:?}", v);
}
```

What the Rust compiler does?

```
pub fn main() {
  let v = vec!(1,2,3);  ⟶ Rust compiler allocates memory in heap for V here
  println!("{:?}", v);
} ⟶ Rust compiler drops the vector V here
```

Another Simple Program

```rust
pub fn main() {
  let v = vec!(1,2,3);
  do_something(v);
  println!("{:?}", v);
}

fn do_something(v: Vec<u64>) {
  // Do something with v
}
```

What happens here?

```
pub fn main() {
  let v = vec!(1,2,3);
  do_something(v);        ⟶ Ownership of v transferred to do_something
  println!("{:?}", v);    ⟶ Rust doesn't allow us to use v here
}

fn do_something(v: Vec<u64>) {
  // Do something with v
}                         ⟶ Rust compiler drops the vector v here
```

Returning Ownership Back

```
pub fn main() {
  let v = vec!(1,2,3);
  let v1 = do_something(v);  ⟶ Ownership of v transferred to do_something
  println!("{:?}", v1);  ⟶ We can safely use v1 here
}  ⟶ Rust compiler drops the vector v1 here

fn do_something(v: Vec<u64>) -> Vec<u64> {
  // Do something with v
  return v;  ⟶ Ownership of v returned to calling fn
}
```

Borrowing

```
pub fn main() {
  let v = vec!(1,2,3);
  do_something(&v);      ⟶ do_something borrows ownership of V
  println!("{:?}", v);   ⟶ Rust doesn't complain about V usage here
}                        ⟶ Rust compiler drops the vector V here

fn do_something(v: &Vec<u64>) {
  // Do something with v
}
```

Let's Mutate Things

```
pub fn main() {
  let v = vec!(1,2,3);
  do_something(&v);
  println!("{:?}", v);
}

fn do_something(v: &Vec<u64>) {
  v.push(4);
}
```

Uh Oh!

```
pub fn main() {
  let v = vec!(1,2,3);
  do_something(&v);
  println!("{:?}", v);
}

fn do_something(v: &Vec<u64>) {
  v.push(4); ⟶ Cannot mutate immutably borrowed V here
}
```

Everything is Mutable

```
pub fn main() {
  let mut v = vec!(1,2,3);
  do_something(&mut v);
  println!("{:?}", v);
}
```

Rust compiler drops the vector V here

```
fn do_something(v: &mut Vec<u64>) {
  v.push(4);
}
```

One Final Note [2]

```
pub fn main() {
  let mut v = vec!(1,2,3);
  let v1 = &v; //First Immutable Borrow is Fine
  let v2 = &v; //Second Immutable Borrow is Fine
  let v3 = &mut v; //Mutable and Immutable Borrows are Not Fine
  println!("{:?}", v);
}
```

Ownership and Borrowing Summary

- Ownership once transferred, cannot be regained
- There is always one owner for value, which is responsible for dropping it
- Cannot mutate immutably borrowed content
- Cannot borrow both mutably and immutably at the same time
- Can immutably borrow any number of times

Why is all this necessary?

- Eliminates common class of memory errors. For eg: Double Free Error
- Avoid data races by allowing only one mutable borrow

Why Rust - The Good Stuff

- The Rust Lang Book [3]
- Beginner Friendly Ecosystem - Rustup, Cargo, VSCode Plugin (RLS Integration) etc.
- Community that is accomodating of newcomers and is always glad to help
- Lot of scope for contributions (For eg: Rust Lang Nursery)
- Growing CLI Infrastructure powered by Rust (For eg: ripgrep, fd)

Questions?

References

[1] Jim Blandy and Jason Orendorff. "Programming Rust: Fast, Safe Systems Development". In: O'Reilly Media, 2017. Chap. Preface, p. xv. ISBN: 1491927283. URL: http://shop.oreilly.com/product/0636920040385.do.

[2] The Rust Lang Community. "The Rust Programming Language - 2nd Edition". In: 2018. Chap. 4. URL: https://doc.rust-lang.org/book/second-edition/ch04-02-references-and-borrowing.html.

[3] The Rust Lang Community. *The Rust Programming Language - 2nd Edition*. 2018. URL: https://doc.rust-lang.org/book/.

[4] Bjarne Stroustrup. "Abstraction and the C++ machine model". In: *International Conference on Embedded Software and Systems*. Springer. 2004, pp. 1–13.

Thank you!

Slides source available at: https://github.com/balajisivaraman/rust-in-peace