

# Advances in Programming Languages

## Lecture 17: Traits and References in Rust

Ian Stark

School of Informatics  
The University of Edinburgh

Thursday 15 November 2018  
Semester 1 Week 9



# Outline

- 1 Opening
- 2 Review
- 3 Traits
- 4 Ownership
- 5 Closing

# Outline

- 1 Opening
- 2 Review
- 3 Traits
- 4 Ownership
- 5 Closing

# Topic: Programming for Memory Safety

The final block of lectures covers some distinctive features of the **Rust** programming language.



- Introduction: Zero-Cost Abstractions (and their cost)
- Control of Memory: Deconstructed Objects, Ownership and Borrowing
- Concurrency: Shared Memory without Data Races

Rust is a fairly new language (1.0 in 2015) that aims to support safe and efficient systems programming. In support of that aim, much of its design builds on the kinds of advances in programming languages that have appeared in this course.



# Homework from Monday

## 1. Read this



The Rust Project Developers

Chucklefish Taps Rust to Bring Safe Concurrency to Video Games

Rust Case Study, 2018

[https://is.gd/rust\\_chuckle](https://is.gd/rust_chuckle)

## 2. Watch this



Byron Cook

Formal Reasoning about the Security of Amazon Web Services

Plenary talk at the Federated Logic Conference 2018.

[https://is.gd/cook\\_floc2018](https://is.gd/cook_floc2018)



# Outline

1 Opening

2 Review

3 Traits

4 Ownership

5 Closing

# The Rust Programming Language

The **Rust** language is intended as a tool for *safe systems programming*. Three key objectives contribute to this.

- Zero-cost abstractions
- Memory safety
- Safe concurrency

## Basic References

<https://www.rust-lang.org>

<https://blog.rust-lang.org>

The “systems programming” motivation resonates with that for imperative **C/C++**. The “safe” draws extensively on techniques developed for functional **Haskell** and **OCaml**. Sometimes these align more closely than you might expect, often through overlap between two aims:

- Precise control for the programmer;
- Precise information for the compiler.

# Review

Some basic Rust constructions.

- Rust *bindings* like `let x = 10;` are immutable by default.
- Mutability must be explicitly declared `let mut y = true;`
- Rust has conditionals, loops, and first-class functions.
- Values can be arranged in *tuples*, *structs*, *tuple structs* and labelled *enumerations*.
- Values can be decomposed with pattern matching and a discriminating `match` statement.
- Parametric polymorphism is available through generic structs, enumerations, and functions.

# Zero-Cost Abstraction

All of these language features — data structures, control structures, generics — provide *abstractions* that help empower a programmer.

However, it's an important principle of Rust (and C++ before it) that all of these can readily be compiled down to simple executable code with no overhead to maintaining the abstraction.

Several of the constraints in the language are there to help with this: default immutability, strict type-checking, checked pattern-matching, restricted `for`, monomorphisation of generics, . . .

These constraints also mean that Rust is not C. It's much more strict on the programmer — which might be judged a cost — but with the benefit of certain behavioural guarantees and potentially more aggressive optimisation from an informed compiler.

# Outline

1 Opening

2 Review

**3 Traits**

4 Ownership

5 Closing

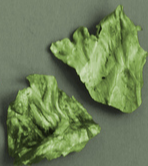
# Rust's Object Model

The meaning of “objects” and “classes” can differ quite substantially between different programming languages. Generally, concepts of “object” bring together in a single construction a range of helpful abstractions. These might include, for example: data, operations, state, identity, references, namespaces, aggregation, encapsulation, inheritance, polymorphism, . . .

Rust doesn't really have an object model. Instead, it provides many of these abstractions as individual components, from which you might pick and choose to build your own abstractions.

This aligns with the zero-cost principle: you need not pay any potential cost of the features you don't use.

*A deconstructed object model*





# Structs

The basic Rust `struct` datatype provides a named record structure with selectors, as with the fields of an object.

```
struct Point {  
    x: f64,  
    y: f64,  
}
```

```
let p = Point { x: 1.0, y: -2.5 };  
let (a,b) = p;
```

```
let mut q = Point { x: 0.0, y: 0.0 };  
q.x = q.x + 3.4;
```

# Method Call Syntax

An `impl` declaration gives methods for values of an associated `struct` type.

```
struct Point { x: f64, y: f64 }

impl Point {
    fn origin_distance (self) -> f64 {
        ( self.x * self.x + self.y * self.y ).sqrt()
    }

    fn flip (self) -> Point {
        Point { x: self.y, y: self.x }
    }
}

let p = Point { x: 2.4, y: 3.5 };
println!("{}", p.flip ().origin_distance()); // Prints 4.2438...
```

# Static Methods

Methods don't have to be attached to a value.

```
struct Point { x: f64, y: f64 }

impl Point {
    fn x_axis (v: f64) -> Point {
        Point { x: v, y: 0.0 }
    }

    fn y_axis (v: f64) -> Point {
        Point { x: 0.0, y: v }
    }
}

let p = Point::x_axis(1.2);
println!("{}", p.flip().origin_distance()); // Prints 1.2
```

# Traits

With a **trait** we can declare a method suite to be **implemented**.

```
trait HasDistance {
    fn origin_distance (self) -> f64;
    fn xy_axis_distance (self) -> (f64,f64);
}

impl HasDistance for Point {
    fn origin_distance (self) -> f64 {
        ( self.x * self.x + self.y * self.y ).sqrt()
    }

    fn xy_axis_distance (self) -> (f64,f64) {
        ( self.y, self.x )
    }
}
```

# Trait Inheritance

With `trait inheritance` we declare that a trait extends one or more others.

```
trait HasDistance {
    fn origin_distance (self) -> f64;
    fn xy_axis_distance (self) -> (f64,f64);
}

trait Has3Distance : HasDistance {
    fn z_axis_distance (self) -> f64;
    fn xyz_axis_distance (self) -> (f64,f64,f64);
}

trait SpaceTime : HasDistance + HasTime {
    fn spacelike (self) -> bool;
    fn timelike (self) -> bool;
}
```



# Standard Traits

Some traits common to many data or numerical types.

**Eq** Can be tested for equality with `==`.

**Ord** Can be tested for order with `<`, `>`, `<=` and `>=`.

**Default** Has some given default value.

**Hash** Provides a `hash` function.

Typically these also assume properties which are not checked by the compiler — for example, that `==` is an equivalence relation.

The `#[derive(...)]` *attribute* prompts the compiler to automatically generate implementations for standard traits like these.

## Traits and Generics

Where a type parameter `<T>` appears in a declaration, it can usually be given a *trait bound* in the form `<T:Trait>`.

```
fn is_in_positive_quadrant<T:HasDistance> (p:T) -> bool {  
    let (x,y) = p.xy_axis_distance();  
    x >= 0.0 && y >= 0.0  
}
```

```
struct HashTable <K:Eq+Hash,V> { ... }
```

```
impl <K:Eq+Hash,V> HashTable <K,V> {  
    fn new () -> HashTable<K,V> { ... }  
}
```

```
let ht : HashTable<(i32,i32),Point> = HashTable::new();
```



## More About Traits

Methods in traits are *statically dispatched* — which code is executed is fixed by the static type of the value for which the method is invoked.

It is possible to arrange for dynamic dispatch, where the code chosen depends on the runtime type of a value, but this is not the default.

Putting trait bounds on generic functions doesn't change the fact that during compilation these are replaced by multiple monomorphic versions.

A *marker trait* is one that doesn't require any methods at all, but serves to record some useful property of a type. These might be recognized specially by the compiler, or describe something that the programmer wishes to indicate but cannot directly express in the language.

For example, the `Sized` trait indicates types whose size in memory is fixed and known at compile time.

# Outline

- 1 Opening
- 2 Review
- 3 Traits
- 4 Ownership**
- 5 Closing



# Passing Structures

```
fn difference (a:Point,b:Point) -> Point {  
    Point { x: (b.x-a.x), y: (b.y-a.y) }  
}
```

```
let p = Point { x:1.2, y:5.0 };  
let q = Point { x:8.3, y:3.7 };
```

```
let r = difference(p,q);           // r is now Point { x:7.1, y:-1.3 }  
let s = r;                         // s is now Point { x:7.1, y:-1.3 }
```

# Boxing Structures

```
fn difference (a:Box<Point>,b:Box<Point>) -> Box<Point> {  
    Box::new (Point { x: (b.x-a.x), y: (b.y-a.y) })  
}  
  
let p = Box::new(Point { x:1.2, y:5.0 });  
let q = Box::new(Point { x:8.3, y:3.7 });  
  
let r = difference(p,q);  
let s = *r;           // s is now Point { x:7.1, y:-1.3 }
```

# Vectors on the Heap

```
fn total (v:Vec<i32>) -> i32 {  
    let mut accum = 0;  
    for x in v { accum = accum + x };  
    accum  
}  
  
let a = vec![1,5,8,3];           // a : Vec<i32>  
  
let t = total(a);                // t is now 17
```

# When to Deallocate?

## In Most Languages

Values explicitly passed around, whether small or large, have a lifetime exactly as long as their bindings stay in scope. They can be placed in stack-allocated memory which is released when they go out of scope. If they are large, though, it may be costly to pass them around.

Values allocated on the heap are cheaper to pass by reference in and out of functions. However, when can the heap space be released?

In **C**, the user has to manage this explicitly. In **Java** or **OCaml** the runtime system has a garbage collector.

Getting this right is important: not just to avoid wasting space, but also for *memory safety* — not reading or writing heap areas after deallocation.

# When to Deallocate?

## In Rust

**Rust** uses *move semantics*: the *ownership* of values passes from one binding to the next; through assignment, function call and return.

The lifetime of a value, whether on the stack or the heap, can be tracked precisely through the lifetimes of its bindings.

The compiler does this statically, guaranteeing memory safety without programmer intervention or a runtime garbage collector.



# Move Semantics

```
fn difference (a:Point,b:Point) -> Point {  
    Point { x: (b.x-a.x), y: (b.y-a.y) }  
}
```

```
let p = Point { x:1, y:5 };
```

```
let q = Point { x:8, y:3 };
```

```
let r = difference(p,q);           // r is now Point { x:7, y:-2 }  
                                   // p and q are no more
```

```
let s = difference(q,p);           // Error: use of moved value
```

```
let t = difference(r,r);           // Error: use of moved value
```

# Move Semantics

```
fn difference (a:Box<Point>,b:Box<Point>) -> Box<Point> {  
    Box::new (Point { x: (b.x-a.x), y: (b.y-a.y) })  
}
```

```
let p = Box::new(Point { x:1, y: 5 });  
let q = Box::new(Point { x:8, y: 3 });
```

```
let r = difference(p,q);  
let s = *r;                // s is now Point { x:7, y:-2 }  
                          // p, q and r are no more
```

```
let t = difference(q,p);   // Error: use of moved value  
let u = *r                 // Error: use of moved value
```

# Move Semantics

```
fn imean (v:Vec<i32>) -> i32 {  
    let mut accum = 0;  
  
    for x in v { accum = accum + x };  
  
    accum / ( v.len() as i32 ) // Error: use of moved value  
}  
  
let a = vec![1,5,8,3]; // a : Vec<i32>  
  
let m = imean(a); // Hoping for 4 here
```

## Clone and Copy Traits

```
fn imean (v:Vec<i32>) -> i32 {  
    let mut accum = 0;  
  
    for x in v.clone() { accum = accum + x };  
  
    accum / ( v.len() as i32 )    // This now works  
}  
  
let a = vec![1,5,8,3];           // a : Vec<i32>  
  
let m = imean(a);               // We do get 4 here
```

## Clone and Copy Traits

The `Clone` trait provides an explicit `clone()` method to duplicate a value. A value and its clone have separate ownership.

Types with the marker trait `Copy` are always assigned by cloning; `Clone` is a supertrait of `Copy`, so all types marked `Copy` also implement `Clone`.

Basic types like `bool` and `i32` implement `Copy`; they have *copy semantics*.

```
let v = 2;
let w = v+v;
let u = 2+2;

fn maxmost (a:Point,b:Point) -> Point {
    Point { x: if a.x > b.x { a.x } else { b.x },
           y: if a.y > b.y { a.y } else { b.y } }
}
```

# Complete Deconstruction

We've now taken apart almost everything that makes an object.

We can put structures on the heap and have the compiler track exactly when to allocate memory and when to free it, with no runtime cost.

We can pass arguments and received results, implement traits and invoke methods, move values and clone them. All runtime computation is clear and visible.

The cost, though, is that now every assignment is a transfer of ownership, and nothing can be used more than once without laboriously making a clone.

This is similar to the world of *linear type systems*



# References and Borrowing

```
fn difference (a: &Point,b: &Point) -> Point {  
    Point { x: (b.x-a.x), y: (b.y-a.y) }  
}
```

```
let p = Point { x:1, y:5 };  
let q = Point { x:8, y:3 };
```

```
let r = difference(&p,&q);    // r is now Point { x:7, y:-2 }  
                           // p and q are still available
```

```
let s = difference(&q,&p);    // Borrow references again
```

```
let t = difference(&r,&r);    // Borrow two references to r
```



## References and Borrowing

```
fn imean (v: &Vec<i32>) -> i32 {  
    let mut accum = 0;  
  
    for x in v { accum = accum + x };    // Just borrowing  
  
    accum / ( v.len() as i32 )    // Works fine  
}  
  
let a = vec![1,5,8,3];                // a : Vec<i32>  
  
let m = imean(&a);                    // We get 4 here
```

# Multiple Reader Single Writer

```
fn swap (x: &mut i32, y: &mut i32) {  
    let (xv, yv) = (*x, *y);  
  
    *x = yv; *y = xv;  
}
```

```
let mut a = 1;  
let mut b = 5;
```

```
swap (&mut a, &mut b);    // Now a = 5 and b = 1
```

```
swap (&mut a, &mut b);    // Back to a = 1 and b = 5
```

```
swap (&mut a, &mut a);    // Borrowing error
```

# Safe Systems Programming

Move semantics lets the compiler statically check the lifetime of structured values. This guarantees memory safety without runtime overhead.

Borrowing references makes it possible to live with move semantics. Borrowing mutable references, with multiple-read single-writer, makes for C-like pointer manipulation and precise control of memory.

That's the core of *safe systems programming* in **Rust**. So all is good, yes?

## The Borrow Checker

“Many new users to Rust experience something we like to call ‘fighting with the borrow checker’, where the Rust compiler refuses to compile a program that the author thinks is valid.”

*The Rust Programming Language*

# Outline

- 1 Opening
- 2 Review
- 3 Traits
- 4 Ownership
- 5 Closing

# Was it all worth it?

Rust certainly offers a lot with its aim of zero-cost abstractions and precise control of memory through ownership and borrowing.

Not everyone is convinced it's worth the cost, though.

It's also notable that **Rust** has an **unsafe** keyword, essential for providing its standard libraries, and **Safe Rust** sits within a larger **Unsafe Rust** language.

So is **Rust** code really safe? What if it uses **unsafe** parts? What has been proved about any of this?

## ERC Project "RustBelt"

### Announcement

We are very pleased to announce the awarding of a 2015 ERC Consolidator Grant for the project **"RustBelt: Logical Foundations for the Future of Safe Systems Programming"**. The project concerns the development of rigorous formal foundations for the Rust programming language (see [project summary](#) below).

The project is 5 years long and will include funding for several postdoc and PhD student positions supervised by [Derek Dreyer](#) at the [Max Planck Institute for Software Systems \(MPI-SWS\)](#) in Saarbruecken, Germany.



# Homework

## 1. Do This

Work through this extremely short introduction to using Rust.



Getting Started With Rust

lil\_firelord, CodinGame

**<https://tech.io/playgrounds/365/getting-started-with-rust>**

If you want to know more then I recommend: <https://stevedonovan.github.io/rust-gentle-intro> and then <https://doc.rust-lang.org/rust-by-example> and then <https://doc.rust-lang.org/book>

## 2. Watch this



RustBelt: Securing the Foundations of the Rust Programming Language

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, Derek Dreyer.

POPL 2018: 4th ACM SIGPLAN Symposium on Principles of Programming Languages

Video: **<https://is.gd/rustbeltpopl2018>**

Slides and more: <https://plv.mpi-sws.org/rustbelt/popl18/>

# Image Credits



## Deconstructed Burger

Things Organized Neatly, Austin Radcliffe on Tumblr



## Deconstructed Coffee

Jamila Rizvi on Facebook and Instagram



## Deconstructed Pizza

The Foodie Collective



## Deconstructed Lucky Charms

Hello Cereal Lovers on Things Organized Neatly.



