A Cursory Introduction to Object Ownership in Rust

By David Johnston

Recommended Version:
nightly (Mac installer)

Rust is a systems programming language that runs blazingly fast, prevents almost all crashes*, and eliminates data races.

Show me more!

Install

Other Downloads

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

```
// This code is editable and runnable!
fn main() {
    // A simple integer calculator:
    // `+` or `-` means add or subtract by 1
    // `*` or `/` means multiply or divide by 2

    let program = "+ + * - /";
    let mut accumulator = 0;

    for token in program.chars() {
        match token {
            '+' => accumulator += 1,
            '-' => accumulator -= 1,
            '*' => accumulator *= 2,
            '/' => accumulator /= 2,
            _ => { /* ignore everything else */ }
        }
    }

    println!("The program \"{}\" calculates the value {}",
             program, accumulator);
}
```

Run

More examples

* In theory. Rust is a work-in-progress and may do anything it likes up to and including eating your laundry.

```
// This code is editable and runnable!
fn main() {
    // A simple integer calculator:
    // `+` or `-` means add or subtract by 1
    // `*` or `/` means multiply or divide by 2

    let program = "+ + * - /";
    let mut accumulator = 0;

    for token in program.chars() {
        match token {
            '+' => accumulator += 1,
            '-' => accumulator -= 1,
            '*' => accumulator *= 2,
            '/' => accumulator /= 2,
            _ => { /* ignore everything else */ }
        }
    }

    println!("The program \"{}\" calculates the value {}",
              program, accumulator);
}
```

Run

Run this on play.rust-lang.org: http://is.gd/J2mxNj

Caveats

This Talk Does *Not*:

- Describe much syntax.
- Explain theory of the type system.
- Explain why type system works.

This Talk *Does*:

- Give intuition behind ownership model.
- Give intuition behind lifetimes model.
- Show moving ownership between threads.

Language Design

- Strong type system with many features.
- Rust includes theoretical and practical language design elements.
  - Theoretical: Fancy type system algorithms
  - Practical: Integrated package, testing, and documentation system
- Ideas taken from many functional and imperative languages.
  - C, C++, Cyclone
  - Scheme
  - ML, Haskell, Erlang

Language Design

- Appropriate substitute for C++, especially *systems programming*.
- An *opinionated* language, which lets the compiler gives strong safety guarantees.
- However, the programmer can always opt *into* `unsafe` operations.

The Most Interesting Feature

Safe and explicit memory management via *Ownership* and *Lifetimes*.

Traditional Problems With Explicit Memory Management (E.g. C/C++)

Some Common Memory Bugs:

- Space Leaks
- Buffer Overruns
- Dangling Pointer Dereferences (*Use After Free*)

This talk will use this last problem as our motivating example.

One Way These Problems Are Mitigated In Modern C++

```cpp
#include <vector>

int main() {
  std::vector<int> vec (3, 100);  // Stack-allocated data structure points to
                                  // three heap-allocated ints, each set to
                                  // 100.
}  // This is the end of `vec`'s "lifetime".
```

One Way These Problems Are Mitigated In Modern C++

```cpp
#include <vector>

int main() {
  std::vector<int> vec (3, 100);  // Stack-allocated data structure points to
                                  // three heap-allocated ints, each set to
                                  // 100.
}  // This is the end of `vec`'s "lifetime".
```

The object's *destructor* is called at the end of its lifetime. Any memory it owns is freed in the process.

If the data structure is recursively defined, then its children should be freed as well.

Problems Remain

```
#include <vector>
#include <foobar>

int main() {
  std::vector<int> vec (3, 100);
  foo(vec);  // What might `foo()` do with the memory which `vec` encapsulates?
}
```

Problems Remain

```cpp
#include <vector>
#include <foobar>

int main() {
  std::vector<int> vec (3, 100);
  foo(vec);  // What might `foo()` do with the memory which `vec` encapsulates?
}
```

In C++ it is hard to know whether `foo()` only “borrowed” `vec` and any resources (e.g. memory) that it encapsulates.

Rust prevents uncertainty about `foo()` with its type system. Rust uses information from the interface of `foo()` to enforce proper ownership and liveness at all times.

A Variable’s Lifetime

```
fn main() {
    let vec = vec![100, 100, 100];  // Stack-allocated data structure points to
                                    // three heap-allocated ints, each set to
                                    // 100.
}  // This is the end of `vec`'s lifetime.
```

The variable is *dropped* at the end of its lifetime. Any memory it owns is freed in the process.

If the data structure is recursively defined, then its children are freed as well.

Rust Makes *Borrowing* Explicit

```
fn foo_borrow(vec: &Vec<u32>) {  // Notice the &.
    // ...
}

fn main() {
    let vec = vec![100, 100, 100];
    foo_borrow(&vec);
}
```

The & represents:

- An address-of operation in C/C++
- A reference in C++
- A *borrow* in Rust

Because of the type signature of `foo_borrow()`, the function body is guaranteed to not stash a reference any memory encapsulated by `vec`.

The callee is guaranteed to give back all ownership of the object to the caller.

Rust Makes *Moving* Ownership Explicit

```
fn foo_move(vec: Vec<u32>) {    // Notice the lack of an &.
    // ...
}

fn main() {
    let vec = vec![100, 100, 100];
    foo_move(vec);
}
```

Because of the type signature of `foo_move()`, the caller takes ownership of `vec`.

After `foo_move(vec)` has been called, `vec` can no longer be used in the body of `main()`.

Demo!

Key Idea Behind Borrow-References

Every borrow-reference has a *statically* known type. A borrow-reference's type includes both

- The Referent's *Type*
- The Referent's *Lifetime*

Other Smart Pointer Types

- Borrow References: `&T`
- Unique “Boxed” Object Pointer: `Box<T>`
- Reference Counted Pointer: `Rc<T>`
- Asynchronous Reference Counted Pointer: `Arc<T>`

These last three are like `Vec<T>`, in the sense that they can be a stack-allocated handle to encapsulate heap-allocated memory.

Thread-Safe Communication By Moving Ownership

The simplest way to create a channel is to use the `channel` function to create a `(Sender, Receiver)` pair. In Rust parlance, a *sender* is a sending endpoint of a channel, and a *receiver* is the receiving endpoint. Consider the following example of calculating two results concurrently:

```
use std::thread::Thread;
use std::sync::mpsc;

let (tx, rx): (mpsc::Sender<u32>, mpsc::Receiver<u32>) = mpsc::channel();

Thread::spawn(move || {
    let result = some_expensive_computation();
    tx.send(result);
});

some_other_expensive_computation();
let result = rx.recv();

fn some_expensive_computation() -> u32 { 42 } // very expensive ;)
fn some_other_expensive_computation() {}      // even more so
```

Questions?