Ownership, Borrowing, and Lifetimes In Rust

The Rust Book: Presented by Shoshana Malfatto And Kevin Blissett
Introduction

- Rust is a new, multi-paradigm, open-source language released in 2010 and sponsored by the Mozilla foundation
- Designed for high performance and safety
  - A good fit for web browsers, operating systems, device drivers, etc.
- Filled with cool features like macros, pattern matching, algebraic data types, and a very unique approach to memory management...
Motivation

- Memory management is one of the critical features of any programming language
- In many programming languages we have two bad options: Garbage Collection or manual memory management
  - Manual memory management is tedious and extremely error prone
  - Garbage collection is heavy and slow
- ... but what if there was a better way?
Ownership

The wonders of memory safety checks at compile time
Review (from every CS class ever)

- **Stack**
  - Data stored on stack must have a known, fixed size which is known at compile time.

- **Heap**
  - Data with unknown size can be stored here.
  - When you ask for space, the OS finds an empty spot on the heap, marks it as used, and returns a pointer. This is called allocating memory.
  - Accessing data on the heap is slower than accessing data on the stack. We want fewer allocations and fewer jumps in memory.
  - Without a garbage collector, the programmer has to remember to explicitly return memory that isn’t needed anymore.

- **Ownership helps with managing heap data!**
Ownership

- Each value in Rust has a variable called its owner.
  - There can only be one owner at a time.
  - When the owner goes out of scope, the value will be dropped.
- Rust strings
  - String literals are immutable.
  - Strings with `String` type, which are allocated on the heap, are mutable.
- Memory is automatically freed when it goes out of scope.
- Every `allocate` should be paired with exactly one `free`.
  - What if you make an alias? What happens when one variable goes out of scope before the other?
  - Move - first variable is invalidated by second variable.
Ownership

```rust
let s1 = String::from("hello");
let s2 = s1;
```
Ownership

```rust
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);
```

error[E0382]: use of moved value: `s1`
  --> src/main.rs:5:28
   |
   |    -- value moved here
   | 3   |
   |     let s2 = s1;
   |     -- value moved here
   | 4   |
   |      println!("{}, world!", s1);
   |      ^^^ value used here after move
   | 5   |
   |      = note: move occurs because `s1` has type `std::string::String`, which does
   |      not implement the `Copy` trait
Ownership

- Doesn’t apply to stack data - `Copy` trait
- Clone - deep copy, expensive
- Function parameters
  - Similar to variable assignment
  - Function can take ownership of value
- Return values
References and Borrowing

How to get around some of those compiler errors (and how to create more)
References and Borrowing

- Creating a reference as a parameter instead of actual value means function never needs to return value since it does not have ownership.
  - Called *borrowing*
  - References are immutable by default
  - You can get around this with mutable references!
- You can only have one mutable reference to a particular piece of data in a particular scope.
  - Breaking this rule will cause a compiler error.
  - You can’t have data races in Rust!
References and Borrowing

```rust
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("The length of "s1" is {}", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

<table>
<thead>
<tr>
<th>s</th>
<th>s1</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>value</td>
</tr>
<tr>
<td>ptr</td>
<td>—</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>—</td>
</tr>
<tr>
<td>len</td>
<td>5</td>
</tr>
<tr>
<td>capacity</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>h</td>
</tr>
<tr>
<td>1</td>
<td>e</td>
</tr>
<tr>
<td>2</td>
<td>l</td>
</tr>
<tr>
<td>3</td>
<td>l</td>
</tr>
<tr>
<td>4</td>
<td>o</td>
</tr>
</tbody>
</table>
References and Borrowing

```rust
def main() {
    let s = String::from("hello");
    change(&s);
}

def change(some_string: &String) {
    some_string.push_str("", world");
}
```

```rust
def main() {
    let mut s = String::from("hello");
    change(&mut s);
}

def change(some_string: &mut String) {
    some_string.push_str("", world");
}
```
References and Borrowing

```rust
define_string_and_mut
```
References and Borrowing

- References will never be dangling references
  - If you have a reference to some data, the data will not go out of scope before the reference does
  - Ex: You can’t have a function return a reference to a String created within the function. The String will be deallocated when the function exits.

- Slices
  - References to part of a String or array

- Concepts of ownership and borrowing ensure memory safety in Rust programs at compile time, while still giving the programmer control over memory usage.
Lifetimes

How long will your variables live?
Lifetimes: Motivation

- Tracking the lifetimes of variables is a key part of the Rust ownership and borrowing system.
  - Handled by a (static analysis) technique called the “borrow checker”
- The borrow checker’s job is to ensure that it’s impossible to use a reference that isn’t valid (enforced at compile time!)
- But how do we track the lifetimes of variables when we’re inside a function
  - We don’t know anything about the scopes of our inputs
  - We don’t know what information to pass back up about how long our return values will remain valid
- Let’s look at the code...
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}
The borrow checker isn’t able to keep track of lifetimes without additional annotation.
Lifetimes

```rust
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

We use `a` to annotate lifetimes. This annotation doesn’t change the lifetime of the variables, but just gives additional information to the compiler.
```rust
fn main() {
    let string1 = String::from("long string is long");

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(), string2.as_str());
        println!("The longest string is {}", result);
    }
}

fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {}", result);
}
```
Fearless Concurrency

Ownership is about more than taking out the trash
Fearless Concurrency

“The Rust project was initiated to solve two thorny problems:

How do you do safe systems programming?

How do you make concurrency painless?

Initially these problems seemed orthogonal, but to our amazement, the solution turned out to be identical: the same tools that make Rust safe also help you tackle concurrency head-on.”

This and following slides adapted from the Rust blog: https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html
Fearless Concurrency

“The compiler prevents all data races.”

(“A data race is any unsynchronized, concurrent access to data involving a write.”)
Fearless Concurrency

- “Memory safety bugs and concurrency bugs often come down to code accessing data when it shouldn't. Rust's secret weapon is *ownership*, a discipline for access control that systems programmers try to follow, but that Rust's compiler checks statically for you.”
- And because ownership is so fundamental to the language, it works for all the flavors of concurrency the language provides. Take your pick!
  - Channels
  - Locks
  - Shared stack frames (!)
  - … and whatever else you like!
Fearless Concurrency: Channels

Go, a language built from the ground up for concurrency, has some sound advice:

Do not communicate by sharing memory; instead, share memory by communicating.

--Effective Go

Rust goes farther than advice and enforces this pattern at the compiler level!
Fearless Concurrency: Channels

```rust
fn send<T: Send>(chan: &Channel<T>, t: T);
fn recv<T: Send>(chan: &Channel<T>) -> T;
```

// Suppose chan: Channel<Vec<i32>>

```rust
let mut vec = Vec::new();
// do some computation
send(&chan, vec);
print_vec(&vec);
```

Ownership prevents this race condition!
“In Rust, threads are ‘isolated’ from each other automatically, due to ownership. Writes can only happen when the thread has mutable access, either by owning the data, or by having a mutable borrow of it. *Either way, the thread is guaranteed to be the only one with access at the time.*”
Fearless Concurrency: Mutexes

// create a new mutex
fn mutex<T: Send>(t: T) -> Mutex<T>;

// acquire the lock
fn lock<T: Send>(mutex: &Mutex<T>) -> MutexGuard<T>;

// access the data protected by the lock
fn access<T: Send>(guard: &mut MutexGuard<T>) -> &mut T;
fn use_lock(mutex: &Mutex<Vec<i32>>) {
    // acquire the lock, taking ownership of a guard;
    // the lock is held for the rest of the scope
    let mut guard = lock(mutex);

    // access the data by mutably borrowing the guard
    let vec = access(&mut guard);

    // vec has type `&mut Vec<i32>`
    vec.push(3);

    // lock automatically released here, when `guard` is destroyed
}
Fearless Concurrency: Mutexes

“There are two key ingredients here:

- The mutable reference returned by `access` cannot outlive the `MutexGuard` it is borrowing from.
- The lock is only released when the `MutexGuard` is destroyed.

“The result is that Rust enforces locking discipline: it will not let you access lock-protected data except when holding the lock. Any attempt to do otherwise will generate a compiler error.”

```rust
fn use_lock(mutex: &Mutex<Vec<i32>>) {
    // acquire the lock, taking ownership of a guard;
    // the lock is held for the rest of the scope
    let mut guard = lock(mutex);

    // access the data by mutably borrowing the guard
    let vec = access(&mut guard);

    // vec has type `&mut Vec<i32>`
    vec.push(3);

    // lock automatically released here, when `guard` is destroyed
}
```
Critique

● The Rust Book is well written and Rust itself is awesome.
● Despite thinking Rust is The Coolest™, I don’t actually use it…
  ○ Rust is decidedly not a language that wants to do everything (like, e.g. Swift). Rust wants to do some things extremely well.
  ○ Programming in Rust just asks for a lot up front in order to get a lot of safety in the end. That means programmer ergonomics are not always the best.
● Level of detail in Rust Book is not always ideal
  ○ Some areas are hard to follow because of the level of detail
  ○ Others seem to glaze over important points
  ○ But overall, the Rust Book is great.
Questions?