Lecture 35 — Rust

Patrick Lam & Jeff Zarnett
patrick.lam@uwaterloo.ca jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

October 15, 2020

Let's get started...

```
fn main() {
    let x = 42; // NB: Rust infers type "s32" for x.
    println!("x is {}", x);
}
```

By default, Rust variables are immutable.

```
fn main() {
    let x = 42; // NB: Rust infers type "s32" for x.
    x = 17; // compile-time error!
}
```

Shadowing

```
let x = 1729;
let x = 88;
println!("shadowed x is {}", x);
```

```
let mut x = 33; // mutable
x = 5;
println!("mutated x is {}", x);
```

More Realistic Shadowing

```
let mut guess = String::new();

io::stdin().read_line(&mut guess)
    .expect("Failed to read line");

let guess: u32 = guess.trim().parse()
     .expect("Please type a number!");
```

We like this because we can re-use the variable name...

```
let war = war();
```

Immutability Rules

By default, a variable in Rust is immutable.

You can make it mutable if you choose, explicitly by declaring it as mutable.

Lots of concurrency issues involve the internal state of objects that are accessed by different threads.

Structs or tuples are either all mutable or all immutable.

Constants

Rust obviously has compile-time constants and they are truly unmodifiable. These have to be known at compile time, and are truly a fixed value.

This is different from an immutable type which is determined at runtime but cannot be changed once it has been assigned.

In C, you can cast away `const`-ness; not so in Rust.

Why Immutable?

The best way to avoid having to use locks (even read/write locks still require writes to acquire the read lock): have no writes.

However, there's a tradeoff.

If your data structure is immutable but you want to update it, you need to copy the data structure, at least partially.

Rust defines the behaviour of going beyond the end of an array: it is a runtime exception (“panic”), unlike C/C++, where it is undefined behaviour (anything can happen).

```
let a = [1,2,3,4,5];
let index = 10;
println!("error!␣{}", a[index]); // panics here.
```

Why Rust?

- harder to write unsafe code: compiler + runtime ensure safety. No arrays-out-of-bounds accesses, null pointers (at all), wild pointers;
- yet can still write low-level code;
- supports zero-cost abstractions (like C++);
- designed with ergonomics in mind;
- type system obviates need for either garbage collection or manual memory management (which you will get wrong)
- type system prevents race conditions;
- dependency management using crates.

Ownership

Rust uses ownership types to manage its heap.

- Each value in Rust has a variable that owns it.
- This variable is unique.
- When the owner goes out of scope, the value will be dropped (aka freed).


This one sparks joy.
THIS MEMORY HAS AN OWNER THAT IS IN SCOPE
This one does not spark joy.
THIS MEMORY HAS NO OWNER THAT IS IN SCOPE
imgflip.com

Variable scopes are fairly standard.

```
fn main() {
    println!("start");
    { // no s
        let s = "I am s";
        println!("s is {}", s);
    } // s now out of scope
}
```

String objects contain a heap component, which may be allocated and freed.

What can go wrong with heap allocation?

It's Yours!

If you own something, you can change it.

```
fn main() {
    let s = String::from("hello"); // immutable String
    let mut s2 = String::from("459 assignments"); // mutable String
    s2.push_str(", maybe?");
    println!("got string {}", s);
}
```

Time to Die?

Rust uses rule #3: if something goes out of scope, then drop (free) it. This is quite like C++ RAII (Resource Acquisition is Initialization).

Still, we need a solution for objects that live beyond their original scope, e.g. return values.

```
fn return_a_string() -> String {
  let s = String::from("longevity");
  return s; // transfers ownership (moves) to caller
}

fn main() {
  let returned_string = return_a_string();
  println!("string␣{}", returned_string);
}
```

Ownership is transferred...

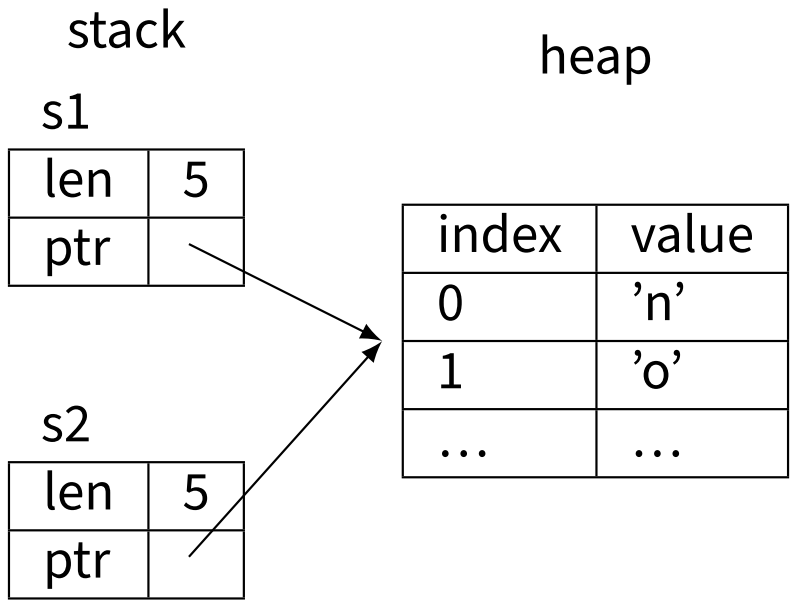
Transferring Ownership

```
let s1 = String::from("no surprise");
println!("can print {}", s1);
let s2 = s1;
println!("can still print {}", s2);
println!("this line won't compile! {}", s1); // no longer owns
```

(Stack variables are copied.)

Well, you see...

Rust strings are a hybrid, containing both a stack part and a heap part.



The assignment `let s2 = s1` carries out a shallow copy of the stack part.
Rust does not automatically copy the heap part.

What should get freed?

It's Yours Now

After the move, `s1` is no longer valid.

Ownership of the heap part is moved from `s1` to `s2` by the assignment.

Only when `s2` goes out of scope do we free the heap object.

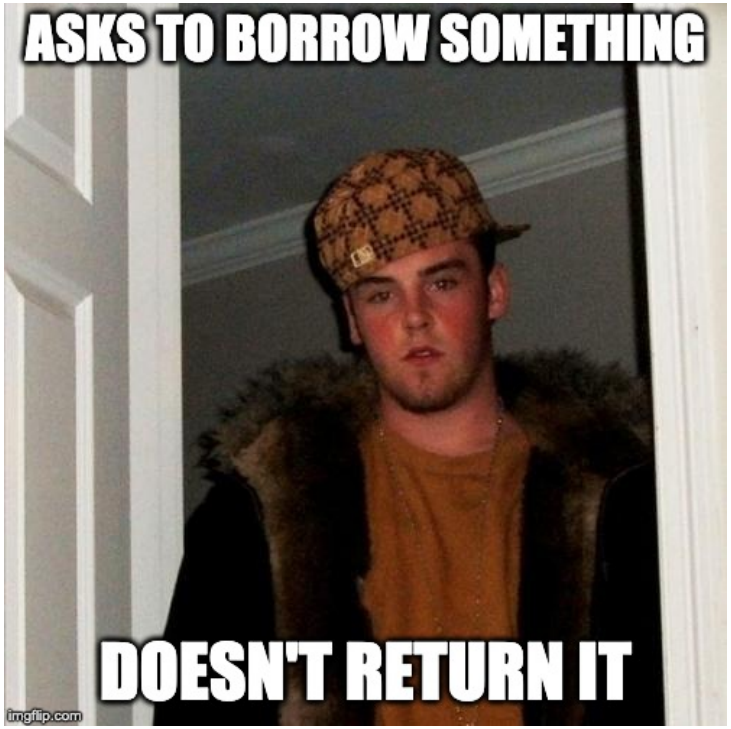
And because the heap object only has one owner, it is only freed once.

Deep copy is possible with "clone", if used explicitly.

Move semantics also applies to function calls and return values, e.g.

```
fn main() {
    let s = String::from("moving to callee");
    callee(s); // afterwards, s is invalid
}
fn callee(param:String) {
    println!("got {}, param");
} // param goes out of scope, object dropped
```

If you return something, then the ownership passes back to the caller.




```
fn main() {
    let s = String::from("459");
    let len = calculate_length(s);
    println!("string␣{}␣has␣length␣{}", s /* we still have it! */,
             len);
}
fn calculate_length(s:&String) -> usize { // note the & for borrow
    s.len() // last expr is return value
} // s is ref so nothing goes out of scope
```

Like other variables, references are immutable by default. We can have mutable references, though.

```
fn change(s:&mut String) {
  s.push_str("more");
}
fn main() {
  let mut main_str = String::from("some");
  change(&mut main_str); // create mutable ref to main_str
}
```

The following code won't compile:

```
let mut s = String::from("one");
let r1 = &mut s;
let r2 = &mut s; // rustc complains!
```

In fact, while `r1` is in scope and a future use of `r1` is to execute, you can't do anything with the original `s`.

The only way to access the string is through `r1`.

After the last use of `r1`, you can create `r2`.

Since there is only one way to access `r1`, then there will be no race conditions.

This is OK:

```
let mut s = String::from("one");
let r1 = &s;
let r2 = &s; // no problem!
```

But you can't then do `let r3 = &mut s;`.

You can have as many outstanding immutable refs as you want.

If there are any immutable refs, you can't have any mutable refs.

The immutable refs must go out of scope first.

You also can't commit use-after-free errors: you can't return a ref that outlives its value.

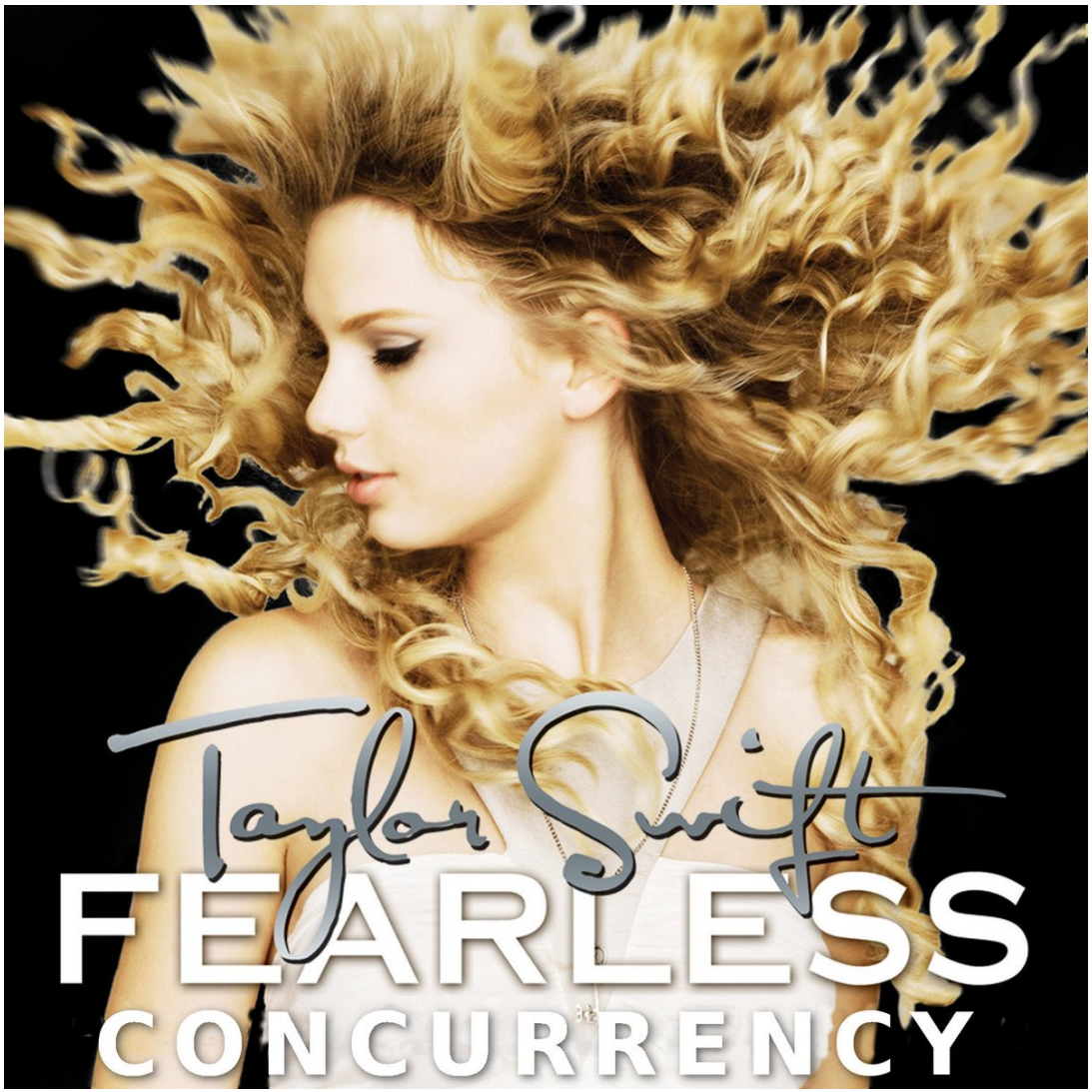
```
fn dangle() -> &String {
  let s = String::from("hello");
  &s // rustc complains: s goes out of scope with active refs
}
```

Rust also has smart pointers, which may be reference counted.

As with many other aspects of Rust, we trade compiler errors for runtime errors; in this case, runtime concurrency errors like race conditions.

That is, the type system ensures concurrency safety!

Fearless Concurrency



Source: Twitter, @ManishEarth:
https://twitter.com/ManishEarth/status/1025423234576076800

We know what to do

Rust uses a fork/join model like pthreads. It delegates to the operating system's threads support and hence implements 1:1 threads.

```
let handle = thread::spawn(|| { // closure (can put args between ||)
  // thread code goes here
});
// main thread continues here
handle.join.unwrap(); // unwrap: panic in case of error
```

This is not too different from C++.

OK, how do we share data between threads? We can move it from main to thread:

```
let v = vec![1,2,3];
let handle = thread::spawn(move || { // move: everything accessed inside
     closure is moved
  println!("vector: {:?}", v);
}); // no longer have access to v in main
handle.join().unwrap();
```

Rust is saving you from being able to concurrently access v in main and thread.

In this case, each value still only has one owner.

We use channels.

The ownership passes from the sender, through the channel, to the receiver.

```
use std::thread;
use std::sync::mpsc; // multi producer, single consumer

fn main() {
    let (tx, rx) = mpsc::channel(); // tx is cloneable
    thread::spawn(move || { // here, tx goes to closure
        let val = String::from("april");
        tx.send(val).unwrap(); // val moved from sender
    });
    let received = rx.recv().unwrap();
    println!("got: {}", received);
}
```

Note the send/receive pair. There is also `try_recv` to do nonblocking receives.

We'll need to talk about multiple ownership. But let's talk about mutexes first.

```
use std::sync::Mutex;
fn main() {
    let m = Mutex::new(5); // mutex guards access to an i32
    {
        let mut num = m.lock().unwrap();
        // unwrap: maybe some other thread panicked while holding lock;
        //         then we panic too.
        *num = 6; // "deref" the mutex (is actually a smart pointer)
    } // release lock when num goes out of scope
    println!("m={:?}", m);
}
```

Well, that's fine, but it's just one thread. We really do need multiple ownership to share data.

Count... Again?

The shared data needs to be owned by all threads, and a naive solution will get rejected by the borrow checker.

Instead, we have to use reference counted cells, implemented by `Arc`.

```
use std::sync::{Mutex, Arc};
use std::thread;
fn main() {
  let counter = Arc::new(Mutex::new(0)); // atomic reference cell
  let mut handles = vec![];

  for _ in 0..10 {
    let counter = Arc::clone(&counter); // clone the Arc
    let handle = thread::spawn(move || {
      let mut num = counter.lock().unwrap();
      *num += 1;
    });
    handles.push(handle);
  }
  for handle in handles {
    handle.join().unwrap();
  }
  println!("result: {}", *counter.lock().unwrap());
}
```

Lock Ownership

Rust guarantees that you have the appropriate lock, using ownership (possibly multiple ownership).

Rust does not guarantee lack of deadlocks.