Why Rust?

2024 Fall ECE454: Computer Systems Programming

Lecture 20

Jon Eyolfson

1.0.0

A Next-Gen Systems Programming Language

Only Compile Correct Programs

This compiler has lots of checks to ensure your program is correct

There are 3 main benefits of using Rust

- Ownership: prevents memory issues without a GC
- Borrow checker: prevents concurrency issues (data races)
- Lifetimes: ensures that references are valid

Hello, World! in Rust

```
fn main() {
  println!("Hello, world!");
}
```

`println!` is a macro, macros always end with !
- Unlike C preprocessor macros that are simple text replacement,
  - Rust macros work with the compiler

Rust is Statically Typed

However, it does type inference by default
- You can omit types and let the compiler figure it out
  - (types are always required for function arguments)

The primitive types are:
- Signed integers: `i8`, `i16`, `i32`, `i64`, `i128` and `isize` (pointer size)
- Unsigned integers: `u8`, `u16`, `u32`, `u64`, `u128` and `usize` (pointer size)
- Floating point: `f32`, `f64`
- `char` Unicode scalar values like `'a'`, `'α'` and `'∞'` (4 bytes each)
- `bool` either `true` or `false`
- The unit type (), whose only possible value is an empty tuple: ()

You can add a suffix to a literal if you want it to be a certain type, e.g. `5u16`

Variables are Immutable by Default

You define a variable using a let statement:

```rust
fn main() {
  let x = 42; /* The default type of integers is i32, you can also write:
                 let x: i32 = 42; */
  println!("x = {}", x);
  x = 0;
}
```

```
 error[E0384]: cannot assign twice to immutable variable `x`
 --> src/bin/hello.rs:4:5
  |
2 |     let x = 42;
  |         - first assignment to `x`
5 |     x = 0;
  |     ^^^^^ cannot assign twice to immutable variable
  |
help: consider making this binding mutable
  |
2 |     let mut x = 42;
  |         +++

For more information about this error, try `rustc --explain E0384`.
```

You Can Declare Variable Mutable, or Shadow It

Unlike other languages, you can shadow the same name in the same scope:

```
fn main() {
  let x = 42;
  println!("x = {}", x);
  let x = 0;
  println!("x = {}", x);
}
```

This is especially useful if you want to change the type of a variable

```
fn main() {
  let x = 42;
  println!("x = {}", x);
  let x: String = x.to_string();
  println!("x = {}", x);
}
```

Rust is an Expression-Based Language

Functions can optionally end with an expression instead of `return`:

```rust
fn main() {
  let x = get_val();
  println!("x = {}", x);
}

fn get_val() -> i32 {
  42
}
```

Unlike C/C++, you don’t need a declaration before you use a function

You Can Define a `struct` (like C/C++)

```rust
struct Point {
  x: i32,
  y: i32,
}
```

You can implement associated functions using:

```rust
impl Point {
  fn distance(&self) -> f64 {
    (self.x.powi(2) + self.y.powi(2)).sqrt()
  }
}
```

Note: `&self` is short for `self: &Self`

Associated Functions with `self` Arguments are Methods

We can create a Pointer and call the method like:

```
fn main() {
  let point = Point {x: 3.0, y: 4.0};
  println!("Distance: {:.1}", point.distance());
}
```

Note: normally you create an associated function called new to construct:

```
impl Point {
  fn new(x: f64, y: f64) -> Self {
    Point { x: x, y: y }
  }
  /* distance */
}
```

Then you can do:

```
fn main() {
  let point = Point::new(3.0, 4.0);
}
```

References are Borrows

In the distance function `&self` means an immutable borrow

You can have as many immutable borrows as you'd like (no data races!)

You can only have one mutable borrow at a time (again, no data races!)

If We Don’t Use References, We’re Claiming Ownership

```rust
impl Point {
  fn destroy(self) {
    println!("Destroying Point {{ x: {}, y: {} }}", self.x, self.y)
  }
}
```

This method claims ownership of the object, when it’s out of scope it’s gone (the Rust term is “dropped”)

By default, the object is moved (if it can be copied, then it’s copied)

The Compiler Will Ensure We Can't Use a Dropped Object

```rust
fn main() {
  let point = Point::new(3.0, 4.0);
  point.destroy();
  point.destroy();
}
```

```
error[E0382]: use of moved value: `point`
   |
21 |     let point = Point::new(3.0, 4.0);
   |         ----- move occurs because `point` has type `Point`,
                             which does not implement the `Copy` trait
...
24 |     point.destroy();
   |           --------- `point` moved due to this method call
25 |     point.destroy();
   |     ^^^^^ value used here after move
   |
note: `Point::destroy` takes ownership of the receiver `self`, which moves `point`
  --> src/bin/point.rs:14:16
   |
14 |     fn destroy(self) {
   |                ^^^^

For more information about this error, try `rustc --explain E0382`
```

Traits Are More Flexible Interfaces

Unlike interfaces, different traits can use the same function names

We could try to implement the Copy trait, but it requires the Clone trait:

```
impl Clone for Point {
  fn clone(&self) -> Self {
    Point { x: self.x, y: self.y }
  }
}
```

If we implement the copy trait, it means we can do `let y = x;` (it uses `clone`)

```
impl Copy for Point {}
```

Now we're destroying a copy every method call, so the compiler lets us

You Can Also Just Let the Compiler Write These For You

```
#[derive(Clone, Copy)]
struct Point {
  x: f64,
  y: f64,
}
```

Another useful one:

```
#[derive(Debug)]
struct Point {
  x: f64,
  y: f64,
}

fn main() {
  let point = Point::new(3.0, 4.0);
  dbg!(point);
}
```

There's Generics Instead of Templates

You can try to implement something like:

```
fn max<T>(v1: T, v2: T) -> T {
      if v1 > v2 {
          v1
      }
      else {
          v2
      }
  }
```

This doesn't work because the compiler doesn't know you're allowed to do >

You Need to Specify Which Trait the Generic Needs

For example:

```
fn max<T: std::cmp::PartialOrd>(v1: T, v2: T) -> T {
    if v1 > v2 {
        v1
    }
    else {
        v2
    }
}
```

You can also specify multiple traits using +

You Can Create Traits

```rust
trait Speak {
  fn speak(&self);
}

struct Cat;
impl Speak for Cat {
    fn speak(&self) {
        println!("Meow!");
    }
}

struct Dog;
impl Speak for Dog {
    fn speak(&self) {
        println!("Woof!");
    }
}
```

Note: `Cat` and `Dog` are zero-sized types

You Can Create Functions Requiring the Traits

```rust
/* Static dispatch, uses generics (multiple versions of function). */
fn call_speak_static(s: &impl Speak) {
    s.speak();
}

/* Dynamic dispatch, uses fat pointers (one copy, uses vtables). */
fn call_speak_dynamic(s: &dyn Speak) {
    s.speak();
}
```

Using static or dynamic is a trade-off, Rust has zero-cost abstractions
(that’s why you need to opt-in to dynamic dispatch)

`enum`s are Tagged and Strongly Typed

They're also called variants, since they could represent multiple things:

```rust
enum Owned {
  Valid(String),
  Invalid,
}

fn main () {
    let x = Owned::Valid(String::from("Success"));
    let y = Owned::Invalid;

    if let Owned::Valid(s) = x {
        println!("x is Valid: {}", s);
    }
    if let Owned::Valid(s) = y {
        println!("y is Valid: {}", s);
    }
}
```

`enum`s are Tagged and Strongly Typed

They’re also called variants, since they could represent multiple things:

```rust
enum Owned {
  Valid(String),
  Invalid,
}

fn main () {
    let x = Owned::Valid(String::from("Success"));
    let y = Owned::Invalid;

    if let Owned::Valid(s) = x {
        println!("x is Valid: {}", s);
    }
    if let Owned::Valid(s) = y {
        println!("y is Valid: {}", s);
    }
}
```

The Standard Library Makes Use of `enum`

```rust
pub enum Option<T> {
  None,
  Some(T),
}

pub enum Result<T, E> {
  Ok(T),
  Err(E),
}
```

If you want to get the value and panic if the value is None, you can use:
- `option.unwrap()` returns `T` or panics

Functions that may fail usually return a `Result`
- There's also syntax sugar like ? to return if there's an `Err`

You Can Also Use Match Expressions

```
match option {
    None => (),
    Some(val) => println!("Got {}", val),
}
```

The compiler ensures you always check every variant
(you can use _ as wildcard)

A `Box` is a Unique Pointer

```rust
fn main() {
    let six = Box::new(6);
    println!("Six = {}", six);
    drop(six);
    println!("Six = {}", six); /* Compile-time error! */
}
```

It’s freed when it goes out of scope (you can also drop it early)

Moving Will Transfer Ownership

```
fn foo(val: Box<i32>) {
    println!("Val = {}", val);
}

fn main() {
    let six = Box::new(6);
    println!("Six = {}", six);
    foo(six);
    println!("Six = {}", six); /* Compile-time error! */
}
```

We Can Transfer to Threads

```
fn main() {
    let four = Box::new(4);
    println!("Main, got: {}", four);
    let t = std::thread::spawn(move || {
        println!("Thread, got: {}", four);
    });
    // println!("Main, still: {}", four); /* Compile-time error. */
    t.join().unwrap();
}
```

C++ Has Move Semantics, But You're on Your Own

```
#include <iostream>
#include <memory>

using namespace std;

void test(unique_ptr<int> ptr) {
  cout << "ptr = " << *ptr << endl;
}

int main(void) {
  unique_ptr<int> six = make_unique<int>(6);
  test(std::move(six));
  cout << "six = " << *six << endl;
  return 0;
}
```

This code segfaults

C++ Also Does Not Care About Lifetimes

```cpp
#include <cstdio>

using namespace std;

int* get_stack() {
    int x = 1;
    return &x;
}

int main(void) {
    int* p = get_stack();
    cout << "*p = " << *p << endl;
    return 0;
}
```

This code segfaults

You Need to Specify Lifetimes, and This Will Not Compile

```rust
fn get_stack<'a>() -> &'a i32 {
    let x: i32 = 42;
    &x
}

fn main() {
    let y: &i32 = get_stack();
    println!("Got {:p}", y);
}
```

```
 error[E0515]: cannot return reference to local variable `x`
 --> src/bin/stack.rs:3:5
  |
3 |     &x
  |     ^^ returns a reference to data owned by the current function

For more information about this error, try `rustc --explain E0515`.
error: could not compile `demos` (bin "stack") due to 1 previous error
```

Let's Create a Vertex Type for a Graph

```rust
#[derive(Debug)]
struct Vertex<'a> {
    value: i32,
    edges: Vec<&'a Vertex<'a>>,
}

fn main() {
    let a = Vertex {
        value: 1,
        edges: vec![]
    };
    let b = Vertex {
        value: 2,
        edges: vec![&a],
    };
    let c = Vertex {
        value: 3,
        edges: vec![&a],
    };
    dbg!(&a);
    dbg!(&b);
    dbg!(&c);
}
```

We Can Try to Increment All Values

```
fn increment_all(vertex: &Vertex) {
    vertex.value = vertex.value + 1; /* Compile-error. */
    for edge in &vertex.edges {
        increment_all(&edge)
    }
}
```

Note, `&mut Vertex` will not work either since we're iterating

There's 4 Main Ways to Get Interior Mutability

`Cell`

`RefCell`

`Mutex`

`RwLock`

We Can Use a Cell Initially

```rust
struct Vertex<'a> {
    value: Cell<i32>,
    edges: Vec<&'a Vertex<'a>>,
}

fn increment_all(vertex: &Vertex) {
    let current_value = vertex.value.get();
    vertex.value.set(current_value + 1);
    for edge in &vertex.edges {
        increment_all(&edge)
    }
}
```

For Non-Copyable Types, We Can Use a RefCell

```
struct Vertex<'a> {
    value: RefCell<String>,
    edges: Vec<&'a Vertex<'a>>,
}

fn uppercase_all(vertex: &Vertex) {
    {
        let mut value = vertex.value.borrow_mut();
        value.make_ascii_uppercase();
    }
    for edge in &vertex.edges {
        uppercase_all(&edge)
    }
}
```

Some Important Threading Traits

`Send` means you can transfer ownership between threads

`Sync` means the object is thread-safe

With Threads We Can’t Guarantee Lifetimes

```rust
struct Vertex {
    value: Mutex<String>,
    edges: Vec<Arc<Vertex>>,
}

fn uppercase_all(vertex: &Vertex) {
    {
        let mut value = vertex.value.lock().unwrap();
        value.make_ascii_uppercase();
        /* Implicit unlock here through drop. */
    }
    for edge in &vertex.edges {
        uppercase_all(&edge)
    }
}
```

We can use a smart pointer, `Arc` (atomic reference counting)

Our Program Has No Data Races, or Lifetime Issues

```
let a = Arc::new(Vertex {
    value: Mutex::new(String::from("a")),
    edges: vec![]
});
let b = Arc::new(Vertex {
    value: Mutex::new(String::from("b")),
    edges: vec![a.clone()],
});
let c = Arc::new(Vertex {
    value: Mutex::new(String::from("c")),
    edges: vec![a.clone()],
});

let t1_c = c.clone();
let t1 = std::thread::spawn(move || {
    uppercase_all(&t1_c);
});
let t2_b = b.clone();
let t2 = std::thread::spawn(move || {
    uppercase_all(&t2_b);
});

t1.join().expect("t1 join not ok");
t2.join().expect("t2 join not ok");
```

Summary of 4 Main Ways to Get Interior Mutability

`Cell`
  You can `get` and `set`, the type must be copyable

`RefCell`
  Dynamic borrow checking, will panic if there’s two mutable borrows

`Mutex`
  Same as `pthread_mutex_t`, safe to use with threads

`RwLock`
  Same as `pthread_rwlock_t`, safe to use with threads

Rust Ensures Your Program is Memory-Safe

The analysis can't prove every valid memory-safe program, only a subset
(this may be initially frustrating coming from C)

However, you may a TON of assumptions in C/C++,
and they may be invalid as code evolves (have fun debugging)

In Rust, if you know better than the compiler, you can always use `unsafe`
(likely you just need to focus on `unsafe` code for debugging)