Rust

David Inyangson

Lesson Objectives

- Become familiar with Rust basic syntax and types
- Apply Rust language features when writing code
- Intuitively understand Rust’s borrowing and ownership semantics

$WHOAMI

- 3rd year PhD student advised by Avi Rubin & Tushar Jois
- Research focus of applied cryptography in anonymous and privacy preserving communication systems and tools
- CMU & Hopkins

Origin Story

- Grayden Hoare
  - Mozilla Software Developer, 2006
- Inspiration: Broken Elevator
- Could you design a program language both
  - Compact
  - Memory error free
- Officially open source
  - 2010

Why so many Rustaceans?

- Fast
  - Can offer performance comparable to C
  - Typically faster than languages like Python, Go
- Safety
  - Memory Safe
  - Type Safe
- Package Manager
- No garbage collection
- No null pointers!

When should you use Rust?

- High performance systems programming
- Parallelism
  - Take advantage of multiple CPUs
- Concurrency
  - Multiple threads
- For smaller projects…
  - Learning curve of Rust is pretty steep
  - May be able to develop less complicated programs quicker in other languages

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

Programming Concepts

Variables & Mutability

- Variables by default are immutable
- Compiler checks for this
- But what is the purpose?
  - Safety?
  - Concurrency?
  - Optimizations?

```
fn main() {
    let x = 5;
    println!("The value of x is: {x}");
    x = 6;
    println!("The value of x is: {x}");
}
```

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:4:5
  |
2 |     let x = 5;
  |         -
  |         |
  |         first assignment to `x`
```

Variables & Mutability

- Having mutable variables can be convenient and essential
- You can intentionally set variables as mutable

```
fn main() {
    let mut x = 5;
    println!("The value of x is: {x}");
    x = 6;
    println!("The value of x is: {x}");
}
```

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/variables`
The value of x is: 5
The value of x is: 6
```

Variables & Mutability

- Constants are always immutable
- To declare
  - Const keyword
  - Include the type
- Variables can be shadowed
  - New variable with same name as previous

```
fn main() {
    let x = 5;

    let x = x + 1;

    {
        let x = x * 2;
        println!("The value of x in the inner scope is: {x}");
    }

    println!("The value of x is: {x}");
}
```

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/variables`
The value of x in the inner scope is: 12
The value of x is: 6
```

Data Types

```
let guess: u32 = "42".parse().expect("Not a number!");
```

- Rust is statically typed
- Every value in Rust is of a specific data type
- Compiler can usually infer but not always…

```
$ cargo build
   Compiling no_type_annotations v0.1.0 (file:///projects/no_type_annotations)
error[E0284]: type annotations needed
 --> src/main.rs:2:9
  |
2 |     let guess = "42".parse().expect("Not a number!");
  |         ^^^^^        ----- type must be known at this point
  |
  = note: cannot satisfy `<_ as FromStr>::Err == _`
help: consider giving `guess` an explicit type
  |
2 |     let guess: /* Type */ = "42".parse().expect("Not a number!");
  |              ++++++++++++

For more information about this error, try `rustc --explain E0284`.
error: could not compile `no_type_annotations` (bin "no_type_annotations") due to 1 prev
```

Data Types

- Scalar Types
  - Integer
  - Floating point number
  - Boolean
  - Character

Length	Signed	Unsigned
8-bit	`i8`	`u8`
16-bit	`i16`	`u16`
32-bit	`i32`	`u32`
64-bit	`i64`	`u64`
128-bit	`i128`	`u128`
arch	`isize`	`usize`

Data Types

- Compound Types
  - Multiple values as one
- Tuples
  - Comma separated list of values
  - Within parenthesis
- Arrays
  - Single chunk of memory
  - Known fixed size
  - Allocated on stack

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
}
```

```
fn main() {
    let a = [1, 2, 3, 4, 5];

    let first = a[0];
    let second = a[1];
}
```

Data types

- Two types of Rust strings: `String` and `&str`.
- `String` is a heap-allocated, growable vector of characters.
- `&str` is a type[1] that's used to slice into `String`.
- String literals like `"foo"` are of type `&str`.

```
let s: &str = "galaxy";
let s2: String = "galaxy".to_string();
let s3: String = String::from("galaxy");
let s4: &str = &s3;
```

[1] `str` is an unsized type, which doesn't have a compile-time known size, and therefore cannot exist by itself.

Data Types

Vec<T>

- A standard library type: you don't need to import anything.
- A `Vec` (read "vector") is a heap-allocated growable array.
  - (cf. Java's `ArrayList`, C++'s `std::vector`, etc.)
- `<T>` denotes a generic type.
  - The type of a `Vec` of `i32`s is `Vec<i32>`.
- Create `Vec`s with `Vec::new()` or the `vec!` macro.
  - `Vec::new()` is an example of namespacing. `new` is a function defined for the `Vec` struct.

Data types

```rust
// Explicit typing
let v0: Vec<i32> = Vec::new();

// v1 and v2 are equal
let mut v1 = Vec::new();
v1.push(1);
v1.push(2);
v1.push(3);

let v2 = vec![1, 2, 3];

// v3 and v4 are equal
let v3 = vec![0; 4];
let v4 = vec![0, 0, 0, 0];
```

Functions

- main()
  - Entrypoint for most programs
- fn keyword
  - Declare new functions
- snake_case

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

    another_function();
}

fn another_function() {
    println!("Another function.");
}
```

Functions

- Statements
  - Instructions that perform some action and do not return a value
- Expression
  - Evaluate to a resultant value
- Rust is an expression based language
- Can end a function with an expression

```
fn main() {
    let x = (let y = 6);
}
```

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
error: expected expression, found `let` statement
 --> src/main.rs:2:14
  |
2 |     let x = (let y = 6);
  |              ^^^
  |
  = note: only supported directly in conditions of `if` and `while` expressions

warning: unnecessary parentheses around assigned value
 --> src/main.rs:2:13
  |
2 |     let x = (let y = 6);
  |             ^         ^
  |
  = note: `#[warn(unused_parens)]` on by default
help: remove these parentheses
  |
2 -     let x = (let y = 6);
2 +     let x = let y = 6;
  |
```

Casting

- Cast between types with `as`:

```
let x: i32 = 100;
let y: u32 = x as u32;
```

- Naturally, you can only cast between types that are safe to cast between.
  - No casting `[i16; 4]` to `char`! (This is called a “non-scalar” cast)
  - There are unsafe mechanisms to overcome this, if you know what you’re doing.

Comments

```
/// Triple-slash comments are docstring comments.
///
/// `rustdoc` uses docstring comments to generate
/// documentation, and supports **Markdown** formatting.
fn foo() {
    // Double-slash comments are normal.

    /* Block comments
     * also exist /* and can be nested! */
     */
}
```

Control Flow

Control Flow

- if expression
  - Branch code depending on conditions
- Can optionally include an ‘else’ expression
- If condition must be a bool
  - It’s either true or false

```
fn main() {
    let number = 6;

    if number % 4 == 0 {
        println!("number is divisible by 4");
    } else if number % 3 == 0 {
        println!("number is divisible by 3");
    } else if number % 2 == 0 {
        println!("number is divisible by 2");
    } else {
        println!("number is not divisible by 4, 3, or 2");
    }
}
```

Control Flow

- Three kinds of loops
  - loop
    - Run until you explicitly stop it
  - while
    - Runs while condition is true
  - for
    - loop through a collection

```
fn main() {
    let mut count = 0;
    'counting_up: loop {
        println!("count = {count}");
        let mut remaining = 10;

        loop {
            println!("remaining = {remaining}");
            if remaining == 9 {
                break;
            }
            if count == 2 {
                break 'counting_up;
            }
            remaining -= 1;
        }

        count += 1;
    }
    println!("End count = {count}");
}
```

Control Flow

- `for` is the most different from most C-like languages
  - `for` loops use an *iterator expression*:
  - `n..m` creates an iterator from n to m (exclusive).
  - Some data structures can be used as iterators, like arrays and `Vec`s.

```
// Loops from 0 to 9.
for x in 0..10 {
    println!("{}", x);
}

let xs = [0, 1, 2, 3, 4];
// Loop through elements in a slice of `xs`.
for x in &xs {
    println!("{}", x);
}
```

Ownership

Ownership

Memory Management

- All programs have to manage how they use a computer's memory while running
- Garbage Collection
  - Regularly look for no longer used memory
- Manual Intervention
  - Programmer must explicitly allocate and free memory
- Rust approach
  - Memory is managed through an ownership system
    - Compiler checks rules
    - If rules are violated, program won’t compile
    - Main purpose is to manage heap data

Ownership

Scope

- Range where an item is valid
- Instead of having to manually deallocate heap data, memory is automatically returned when a variable goes out of scope
- drop()

```
fn foo() {
    // Creates a Vec object.
    // Gives ownership of the Vec object to v1.
    let mut v1 = vec![1, 2, 3];

    v1.pop();
    v1.push(4);

    // At the end of the scope, v1 goes out of
scope.
    // v1 still owns the Vec object, so it can be
cleaned up.
}
```

Ownership

Move

- Data on the heap is not copied, but rather the first variable goes out of scope
- To avoid double free errors, Rust has a ‘move’ operation
- More akin to a shallow copy

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

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

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:15
  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `String`, which does not implement the
3 |     let s2 = s1;
  |              -- value moved here
4 |
5 |     println!("{s1}, world!");
  |               ^^^^ value borrowed here after move
  |
  = note: this error originates in the macro `$crate::format_args_nl` which comes from t
help: consider cloning the value if the performance cost is acceptable
  |
3 |     let s2 = s1.clone();
  |                ++++++++

For more information about this error, try `rustc --explain E0382`.
error: could not compile `ownership` (bin "ownership") due to 1 previous error
```

Ownership

Clone

- If we want a deep copy, we can clone data
- Types where size is known at compile time and stored on the stack implement the Copy trait

```
let x = 5;
let y = x;

println!("x = {x}, y = {y}");
```

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

println!("s1 = {s1}, s2 = {s2}");
```

Ownership

- Passing a variable to a function will move or copy it

```
fn main() {
    let s = String::from("hello");  // s comes into scope

    takes_ownership(s);             // s's value moves into the function...
                                    // ... and so is no longer valid here

    let x = 5;                      // x comes into scope

    makes_copy(x);                  // x would move into the function,
                                    // but i32 is Copy, so it's okay to still
                                    // use x afterward

} // Here, x goes out of scope, then s. But because s's value was moved, nothing
  // special happens.

fn takes_ownership(some_string: String) { // some_string comes into scope
    println!("{some_string}");
} // Here, some_string goes out of scope and `drop` is called. The backing
  // memory is freed.

fn makes_copy(some_integer: i32) { // some_integer comes into scope
    println!("{some_integer}");
} // Here, some_integer goes out of scope. Nothing special happens.
```

Ownership

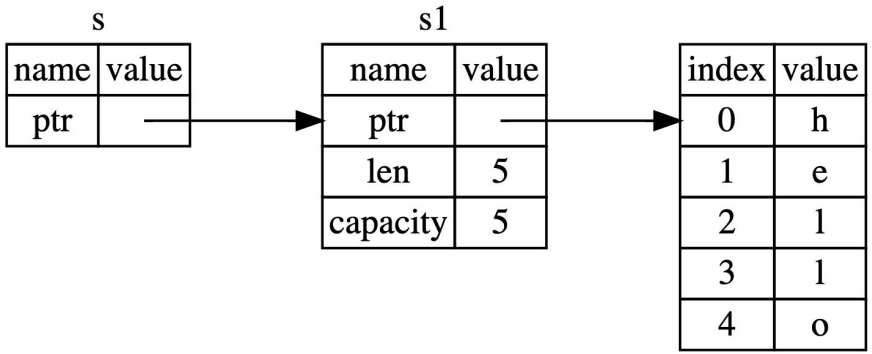
- Taking ownership and then returning ownership with every function is a bit tedious
- Reference
  - Like a pointer
  - Allows us to access data owned by another variable
  - Value reference points to won't be dropped when reference is out of scope
    - No double free error
  - Can't modify something you're borrowing

```rust
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{s1}' is {len}.");
}

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



Ownership

What if we need to change the value of a reference

- Mutable references
- One restriction
  - Can not borrow a variable as mutable more than once at a time
- Why?
  - Data races
    - Two or more pointers access data at the same time
    - May try to write to it

```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

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

```
let mut s = String::from("hello");

{
    let r1 = &mut s;
} // r1 goes out of scope here, so

let r2 = &mut s;
```

Lab 2

CARGO

- Rust’s package manager & build tool
- Create a new project:
  - `cargo new project_name` (library)
  - `cargo new project_name --bin` (executable)
- Build your project: `cargo build`
- Run your tests: `cargo test`
- Magic, right? How does this work?

Cargo

- Cargo uses the `Cargo.toml` file to declare and manage dependencies and project metadata.
  - TOML is a simple format similar to INI.

```
[package]
name = "Rust"
version = "0.1.0"
authors = ["Ferris <ferris@rust-lang.org>" ]

[dependencies]
uuid = "0.1"
rand = "0.3"

[profile.release]
opt-level = 3
debug = false
```

Lab 2

- Lab 2
- Install Rust
- Set up a new project

Homework

- Complete the Lab
- Complete the assigned reading

BEFORE CLASS ENDS: COMPLETE EXIT TICKET!

- https://forms.gle/YFBZaWemezgUqjQDA