RUST PROGRAMMING COURSE NOTES

DAVID YANG

1. Rust Basics

```rust
// fields go in structs
struct Dog {
    breed: String,
    age: u32, // unsigned
}

// methods/functions go in "impl" block
impl Dog {
    fn bark(&self) {
        println!("bark!");
    }
}

// add and max functions
fn add(a: i32, b: i32) -> i32{
    a + b // or "return a + b;"
}

fn max(a: i32, b: i32) -> i32{
    if a > b {
        a
    }

    else {
        b
    }
}

fn order(a: i32, b: i32) -> (i32, i32) {
    if a > b { (b, a) } else { (a, b) }
}

fn my_name() -> String {
    "David".to_string()
}

fn main() {
    // print statement
```

```rust
    println!("Hello, world!");

    // for loop
    for i in 0..10 {
        println!("i is: {}", i);
    }

    // defining immutable int
    let x: i32 = 0;
    println!("{x}");

    // array
    let arr = [1,2,3,4];


    // using classes
    let sparky = Dog {
        breed: "Chihuahua".to_string(),
        age: 4,
    };

    sparky.bark();

    // testing functions
    println!("{}", add(4, 5));

    let x: i32 = -5;

    // no parentheses around conditionals
    let mut abs: i32 = if x > 0 {
        x
    } else {
        -x
    };

    abs += 1;


}
```

2. Memory Management in Rust

2.1. Ownership.

Definition 1. *Rust uses an* ***ownership model*** *to manage memory, with three fundamental rules.*

1. *Every value has an owner (variable/struct).*
2. *Every owner is unique.*
3. *When the owner goes out of scope, the value is dropped (freed).*

```
fn main() {
    // allocated on the heap
    let string = "Hello".to_string();
    // dropped
} // 'string' goes out of scope, so "Hello" freed

fn say_string(string: String) {
    println!("{}", string);
}

fn main() {
    let string = "Hello".to_string();
    say_string(string);
    // running say_string(string) again would give an error, as the value
    // is used after the move.
}
```

Ownership is checked at compile time. One approach would be to clone the string with string.clone and to then call the function on it.

Note that this ownership model <u>does not</u> impact "cheap types" like integers, booleans, and floats.

2.2. **References.** We can use '&' for references. For example, we could pass a reference to an initial string and call say_string on it just fine.

```
fn say_string(string: &String) {
    println!("{}", string);
}

fn main() {
    let string = "Hello".to_string();
    say_string(&string);
    say_string(&string);
}
```

Another example:

```
fn main() { // this is okay, as a loses ownership to b.
    let b;
    {
        let a = "Hello".to_string();
        say_string(&a);
        b = a;
    }
    say_string(&b);
}
```

Let's now try to implement a counter.

```
struct Counter {
    count: u32,
}

impl Counter {
    fn get_count(&self) -> u32 {
        self.count
    }

    fn increment(&mut self) {
        self.count += 1
    }
}

fn main() {
    let mut counter = Counter {count : 0};
    // mutable reference needs to have mutable ‘‘var’’ owner
}
```

You can take one mutable reference or as many immutable references as you want at a time (but not both).

```
let mut vec: vec![1, 2, 3];
let first = &vec[0];
vec.clear(); // mutable reference

println!("{}", first) // first is an immutable borrow, and we cannot have both
    at the same time
```

The main takeaways can be summarized as follows:

- Every piece of data has a unique owner.
- When that owner goes out of scope, the data is dropped.
- Ownership can be transferred by moving or copying the data.
- Data can also be borrowed via references to avoid unnecessary copying/moving.
- References are guaranteed at compile time to always be valid.
- Slices are references to contiguous chunks of memory.

- You can't borrow something if it is already mutably borrowed, guaranteeing immutability.

3. Enums

Definition 2. *In Rust, enums can be used to define a type that can take on one of multiple possible variants.*

```
enum Shape {
    Circle,
    Rectangle,
}

let what_shape: String = match circle {
    Shape::Circle => "circle".to_string(),
    _ => "not a circle".to_string(), // _ is a "wildcard" type
};

fn main() {
    let circle = Shape::Circle;

    match circle {
        Shape::Circle => println!("Circle"),
        Shape::Rectangle => println!("Rectangle"),
    }
}
```

We can also add data to each variant, either by making it a tuple variant or a struct variant.

Definition 3. *Tuple variants have unnamed fields, while struct variants have named fields.*

```
enum Shape {
    Circle(f64),
    Rectangle {
        width: f64,
        height: f64,
    }
}

fn main() {
    let circle = Shape::Circle(5.0);

    match circle {
        Shape::Circle(radius) => {
            println!("Circle with radius {radius}");
        }
        Shape::Rectangle { width: w, height: h } => {
            println!("Rectangle that is {w} by {h}");
        }
```

```
    }
}
```

Note that enums are also types, so we can pass in different variants of a given structure to a function expecting that structure:

```
impl Shape {
    fn area(&self) -> f64 {
        match self {
            Shape::Rectangle { width, height } => {
                width * height
            }
            Shape::Circle(radius) => {
                3.141 * radius * radius
            }
        }
    }
}

let circle: Shape = Shape::Circle(5.0);
let area = circle.area();
println!("Area: {}", area);
```

We can use enums to solve some major problems, including nullability and error handling.

For example, to handle "divide by zero" errors, we can use the `Option` enum.

```
// The 'T' is a generic type, ignore for now.
enum Option<T> {
    None,
    Some(T),
}

fn divide(numerator: f32, denominator: f32) -> Option<f32> {
    // Check for div by zero
    if denominator == 0.0 {
        // We can't divide by zero, no float can be returned
        None
    } else {
        // denominator is nonzero, we can do the operation
        let quotient: f32 = numerator / denominator;

        // Can't just return 'quotient' because it's 'f32'
        Some(quotient)
    }
}
```

```
let quotient: Option<f32> = divide(10.0, 2.0);
println!("{:?}", quotient);

let zero_div: Option<f32> = divide(10.0, 0.0);
println!("{:?}", zero_div);
```

We can similarly account for null pointer names for greet(), which we coded in Lab 1.

```
fn greet(maybe_name: Option<&str>) {
    match maybe_name {
        Some(name) => println!("Hello, {}", name),
        None => println!("Who's there?"),
    }
}

greet(Some("William"));
greet(None);
```

The other big problem that can be handled by enums is error handling. To handle this sort of error, we can use the Result enum.

```
enum Result<T, E> {
    // success
    Ok(T),
    // failure
    Err(E),
}
```

We can use the `?` operator to allow for error propagation:

```
fn read_to_string(path: &str) -> Result<String, io::Error>

use std::{fs, io};

fn main() -> Result<(), io::Error> {
    let string: String = match fs::read_to_string("names.txt") {
        Ok(string) => string,
        Err(err) => return Err(err),
    };

    println!("{}", string);
    Ok(())
}
```

can be transformed to the following code:

```
use std::{fs, io};
```

```rust
// the `Ok` value is `()`, the unit type.
fn main() -> Result<(), io::Error> {
    let string: String = fs::read_to_string("names.txt")?; // <-- here

    println!("{}", string);
    Ok(())
}
```

This can also work for functions that return option types, as follows:

```rust
fn sum_first_two(vec: &[i32]) -> Option<i32> {
    // `.get(x)` might return `None` if the vec is empty.
    // If either of these `.get(x)`s fail, the function will
    // short circuit and return `None`.
    Some(vec.get(0)? + vec.get(1)?)
}
```

We can even write expressions with `?` , which is itself an expression! For example, we can do the following, which takes the result of a division and add 5 to it.

```rust
fn compute(a: f32, b: f32) -> Result<f32, String> {
    Ok(divide(n: a, d: b)? + 5.0)
}
```

An important caveat of the "?" syntax is that it only works with `Option` and `Result` types.

To summarize, we learned the following:

- Rust enums are types that can be one of several variants which may contain different types.
- We use match statements to determine which variant an enum is.
- The problem of null pointers and references can be solved with enums like `Option<T>`
- Different languages have their own ways to mark a function as "fallible", Rust has the `Result<T, E> enum` .
- The `?` operator can be used to propagate errors with minimal syntactic overhead.
- Enums are excellent for representing possible kinds of errors.
- The `?` operator can perform implicit conversion using the `From<T>` trait.