

# RUST PROGRAMMING COURSE NOTES

DAVID YANG

## 1. RUST BASICS

---

```
// 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
```

```
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 does not 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
```

---