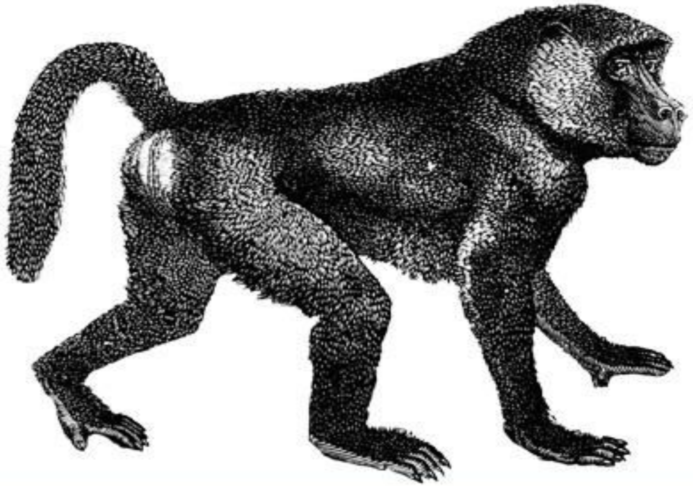
Talk is cheap. Show me the code.

Linus Torvalds

Content

Geeks México



AI based on if / else statements

The Definitive Guide

O RLY?

@raidentrance

Control Flow

`if` Expressions

An if expression allows you to branch your code depending on conditions.

You provide a condition and then state, “If this condition is met, run this block of code. If the condition is not met, do not run this block of code.”

Control Flow

`if` Expressions

```
fn main() {
    let number = 3;

    if number != 0 {
        println!("number was something other than zero");
    }
}
```

Control Flow

Handling Multiple Conditions with `else if`

You can have multiple conditions by combining if and else in an else if expression.

```
let n = 5;

    if n < 0 {
        print!("{} is negative", n);
    } else if n > 0 {
        print!("{} is positive", n);
    } else {
        print!("{} is zero", n);
    }
```

Control Flow

Handling Multiple Conditions with `else if`

You can have multiple conditions by combining if and else in an else if expression.

Control Flow

Using `if` in a `let` Statement

Because if is an expression, we can use it on the right side of a let statement,

```
let number = if true {
        5
   } else {
        6
  };
```

Match Statement

Rust has an extremely powerful control flow operator called match that allows you to compare a value against a series of patterns and then execute code based on which pattern matches.

```rust
let number = 3;
   match number {
       3 => println!("Number is Three"),
       5 => println!("Number is Five"),
       _ => println!("Invalid Number"),
   }
```

Control Flow

Repetition with Loops

It’s often useful to execute a block of code more than once. For this task, Rust provides several loops. A loop runs through the code inside the loop body to the end and then starts immediately back at the beginning.

Rust has three kinds of loops: `loop`, `while`, and `for`.

Control Flow

Repeating Code with `loop`

The loop keyword tells Rust to execute a block of code over and over again forever or until you explicitly tell it to stop.

```rust
fn main() {
    loop {
        println!("Loop forever!");
    }
}
```

Control Flow

Returning Values from Loops

```
   let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
  };
```

Control Flow

Conditional Loops with while

It’s often useful for a program to evaluate a condition within a loop. While the condition is true, the loop runs. A while loop looks like the code

```
fn main() {

   Let mut a=1;
    While a <= 10 {
    println!("Current value : {}”, a);
    a += 1; //no ++ or -- on Rust
    }
}
```

Control Flow

Looping Through a Collection with `for`

you can use a for loop and execute some code for each item in a collection. A `for` loop looks like the code

```
fn main() {

    // other programming languages (a = 1; a <10; a++)
    // 0 to 10(exclusive)
    for element in 1..10 {
            println!("Current value : {}", element);
    }
}
```

Don’t do this

LITERALLY ANY OTHER LANGUAGE
ME
THE LANGUAGE I'M STUDYING
imgflip.com

Data Types(Compound Types)

Compound types can group multiple values into one type. Rust has two primitive compound types: tuples and arrays.

1. A **Tuple** is a general way of grouping together some number of other values with a variety of types into one compound type. Tuples have a fixed length: once declared, they cannot grow or shrink in size.

2. Another way to have a collection of multiple values is with an array. Every element of an array must have the same type. **Arrays** in Rust are different from arrays in some other languages because arrays in Rust have a fixed length, like tuples.

Compound Types (Tuple)

The variable **tup** binds to the entire tuple, because a tuple is considered a single compound element.

```rust
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

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

Compound Types (Tuple)

we can access a tuple element directly by using a period (.) followed by the index of the value we want to access.

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

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
```

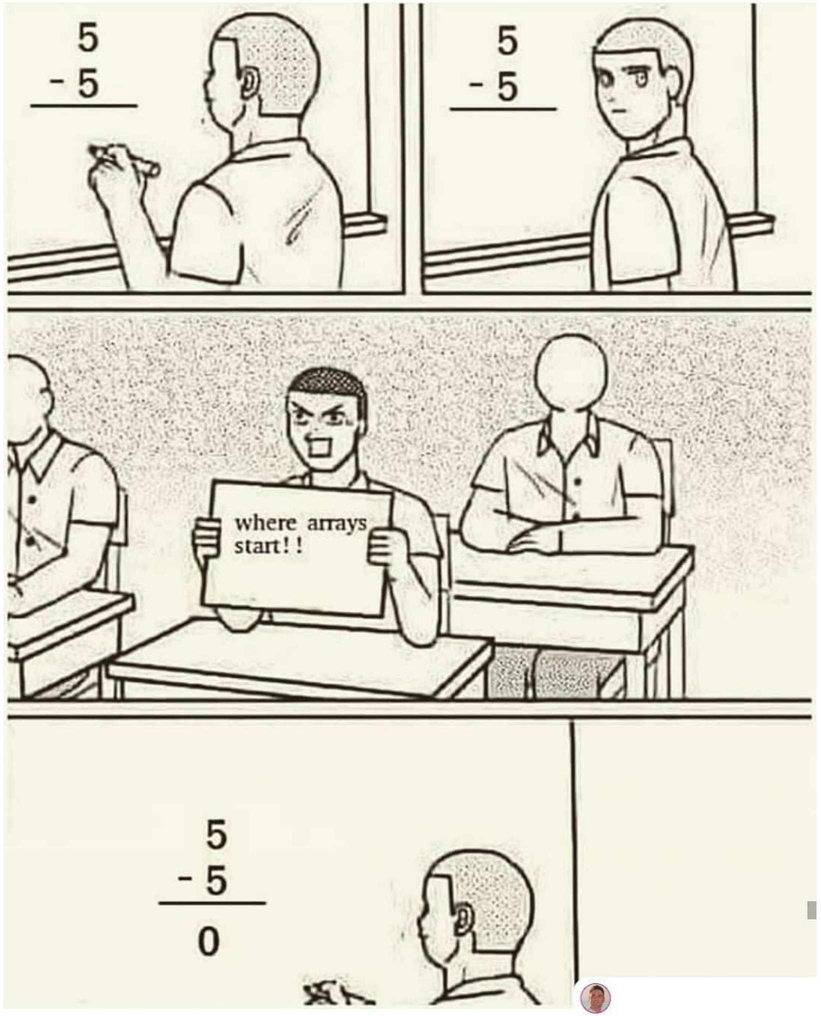
Compound Types (Array)

Arrays are useful when you want your data allocated on the stack rather than the heap or when you want to ensure you always have a fixed number of elements.

```rust
let a = [1, 2, 3, 4, 5];

let a: [i32; 5] = [1, 2, 3, 4, 5];

let a = [3; 5];
```


5
- 5
5
- 5
where arrays start!!
5
- 5
0

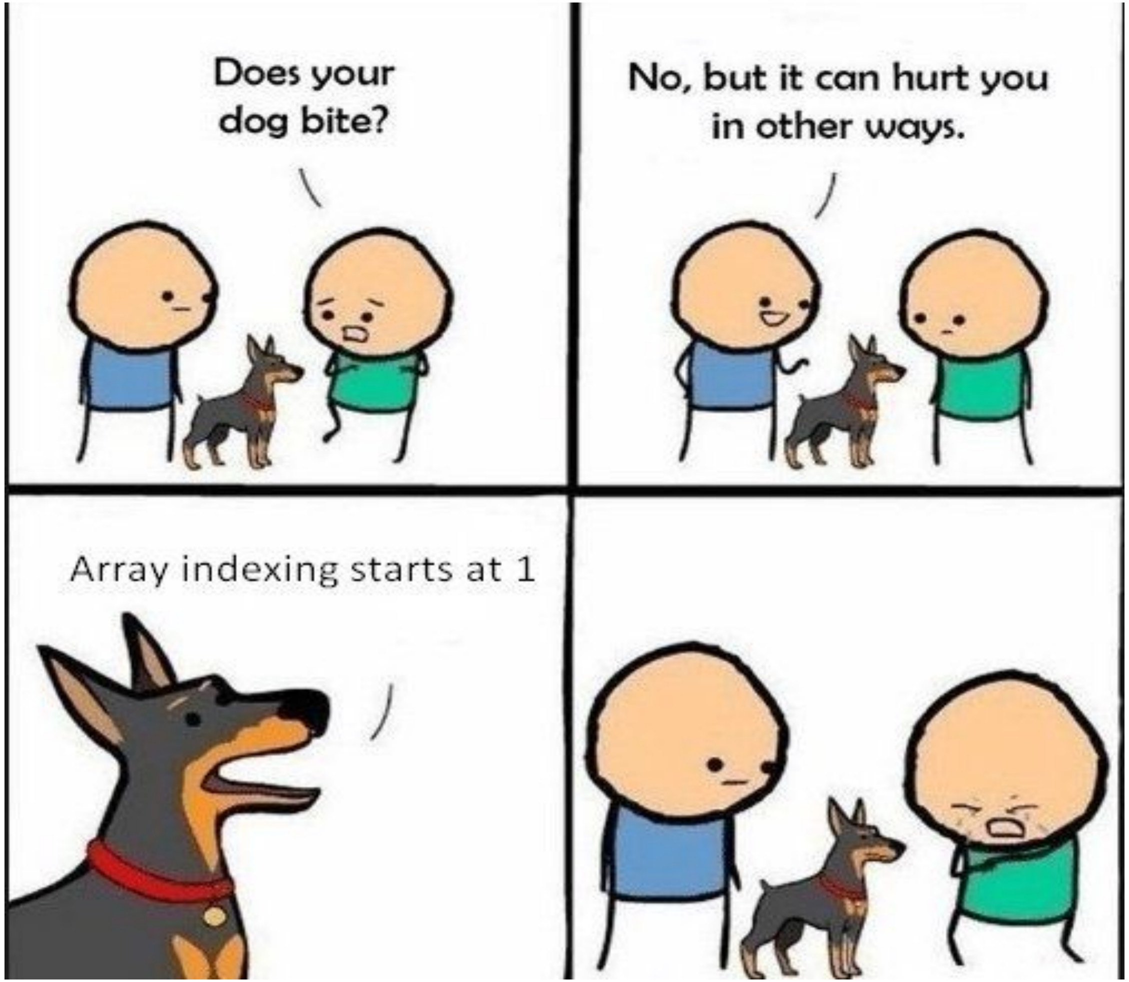
Compound Types (Array)

Accessing Array elements

```
let a = [1, 2, 3, 4, 5];
let first = a[0];
let second = a[1];
```

What happens if you try to access an element of an array that is past the end of the array?

```
let a = [1, 2, 3, 4, 5];
let index = 10;
let element = a[index]; // Note: This will panic
because Accessing invalid index produces a panic
```


Does your dog bite?
No, but it can hurt you in other ways.
Array indexing starts at 1

Control Flow

Array	**Tuples**
An array is a list of items of homogeneous type. You can iterate over it and index or slice it with dynamic indices. It should be used for homogeneous collections of items that play the same role in the code. In general, you will iterate over an array at least once in your code.	A tuple is a fixed-length agglomeration of heterogeneous items. It should be thought of as a struct with anonymous fields. The fields generally have different meaning in the code, and you can't iterate over it.

Eid Holiday for Eid Al Adha

14
AUGUST
Masoom
DIL DIL PAKISTAN
PAKISTAN PAKISTAN
mera inam pakistan muhobat aman hai ur
aman ka name hai pakistan

Eid Holiday on August 11, 2019.

Next class will be on Aug 18, 2019.

Thank You