Rust

Basics

Cargo is a Package Manager

Cargo.toml

```
[package]
name = “hello_world”
version  = “0.0.1”
authors = [“Author of the package”]

[dependencies]
actix = “0.9”
```

To install dependencies and compile a rust application, use

“cargo run”

All variables are immutable

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

Wrong!

Compiler will complain

```
    x = 6;
    ^^^^^ cannot assign twice to immutable variable
```

To make variable is mutable

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

OK!

Data types

Integer and Float

i8 (-127..128), u8 (0..255)

i16, u16

i32, u32, f32

i64, u64, f64

i128, u128

isize, usize
(depends on PC
architecture)

```
fn main() {
    let x: i8 = 5;
    let x: f32 = 5.1;
}
```

Boolean

```
fn main() {
    let x: bool = true;
}
```

Character

```
fn main() {
    let x = 'z';
}
```

Array

```
fn main() {
    let x = [1,2];
}
```

Data types

Tuple

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

    let (x, y, z) = tup;

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

A tuple is a general way of grouping together a number of values with a variety of types into one compound type.

Function

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

    another_function();
}

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

Function with arguments and return value

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

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

fn plus_one(x: i32) -> i32 {
    x + 1
}
```

Structure

Definition

```
struct User {
    username: String,
    email: String,
}
```

Instantiation

```
let user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("someusername123"),
};
```

Mutation of values

```
let mut user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("someusername123"),
};

user1.email = String::from("anotheremail@example.com");
```

A *struct*, or *structure*, is a custom data type that lets you name and package together multiple related values that make up a meaningful group

Structure methods implementation

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

Enum

```
enum IpAddrKind {
    V4,
    V6,
}
```

```
let four = IpAddrKind::V4;
let six = IpAddrKind::V6;
```

```
enum IpAddrKind {
    V4(u8, u8, u8, u8),
    V6(String),
}
```

```
let home = IpAddr::V4(127, 0, 0, 1);
let loopback = IpAddr::V6(String::from("::1"));
```

Enums allow you to define a type by enumerating its possible *variants*

Option

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

```
let some_number = Some(5);
let some_string = Some("a string");

let absent_number: Option<i32> = None;
```

Rust does not have nulls, but it does have an enum that can encode the concept of a value being present or absent

The `match` Control Flow Operator

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

`match` allows to compare a value against a series of patterns and then execute code based on which pattern matches

The `match` Control Flow Operator

The _ Placeholder

```
let some_value = 3;

match some_value {
    1 => println!("one"),
    3 => println!("three"),
    5 => println!("five"),
    7 => println!("seven"),
    _ => (),
}
```

_ pattern can be used when we don’t want to list all possible values

Control Flow with `if let`

with match

```
let some_value = Some(3);

match some_value {
    Some(3) => println!("three"),
    _ => (),
}
```

with _

```
if let Some(3) = some_u8_value {
    println!("three");
}
```

The `if let` syntax lets you combine `if` and `let` into a less verbose way to handle values that match one pattern while ignoring the rest.

Traits

Defining

```
pub trait Summary {
    fn summarize(&self) -> String;
}
```

Trait definitions are a way to group method signatures together to define a set of behaviours necessary to accomplish some purpose.

Implementation of function definition is required

Default implementation

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}
```

Default behaviour for some or all of the methods in a trait instead of requiring implementations for all methods on every type

Traits used to define shared behaviour in an abstract way

Traits

Implementing a Trait on a Type

```rust
pub trait Summary {
    fn summarize(&self) -> String;
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

Traits

```
let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    };

    println!("1 new tweet: {}", tweet.summarize());
```

This code prints **1 new tweet: horse_ebooks: of course, as you probably already know, people.**

Traits

Trait as Parameters

```rust
pub trait Summary {
    fn summarize(&self) -> String;
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

```rust
pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```