Table of Contents

Learning Rust

I am a Web Developer and just learning Rust. In here I tried to summarize what I learned and time to time I'll update this book. Hope this will be helpful for newcomers like me :)

This is based on the posts I wrote on Medium, https://medium.com/learning-rust

01. RUST BASICS

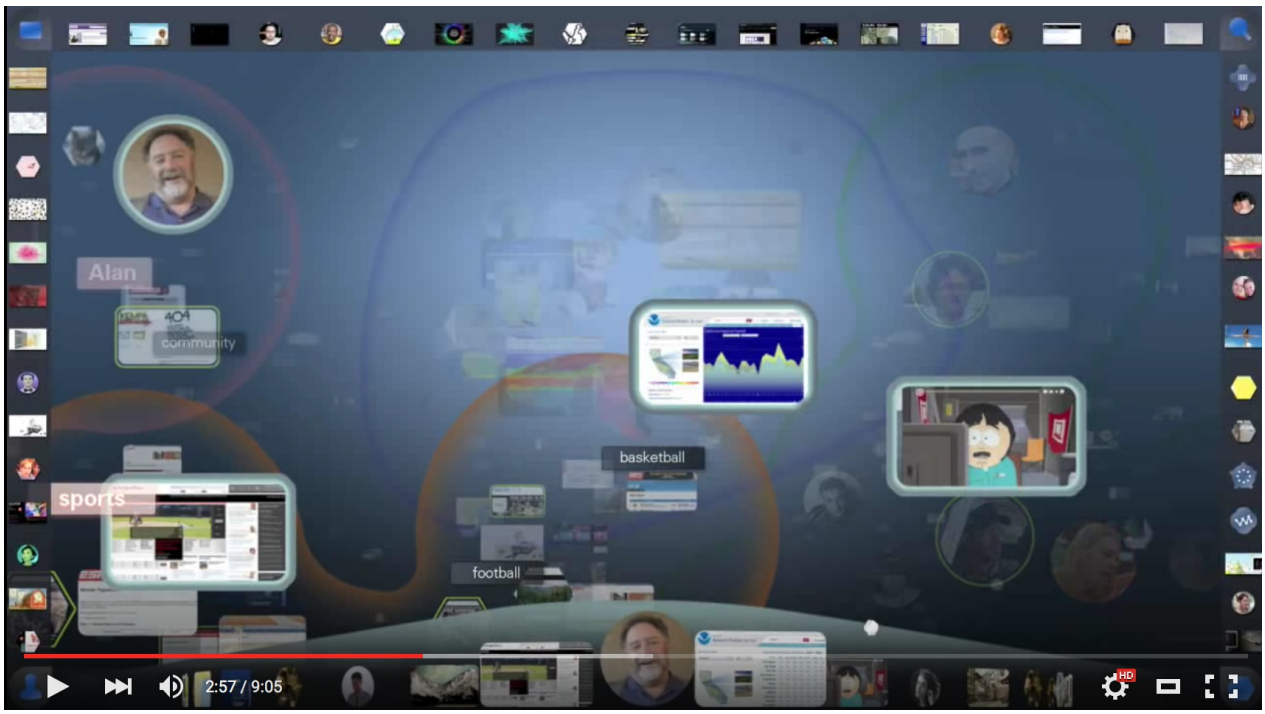
02. RUST : BEYOND THE BASICS

03. RUST : THE TOUGH PART

About Me

Hi, I'm Dumindu Madunuwan. I'm a web developer, mainly PHP.

Around 2007–2009, I chose web development because at that time I truly believed, web will be the next ultimate, system independent & language independent platform and the next generation software ecosystem will be implemented top of browsers. The web I expected is sort of similar to this.



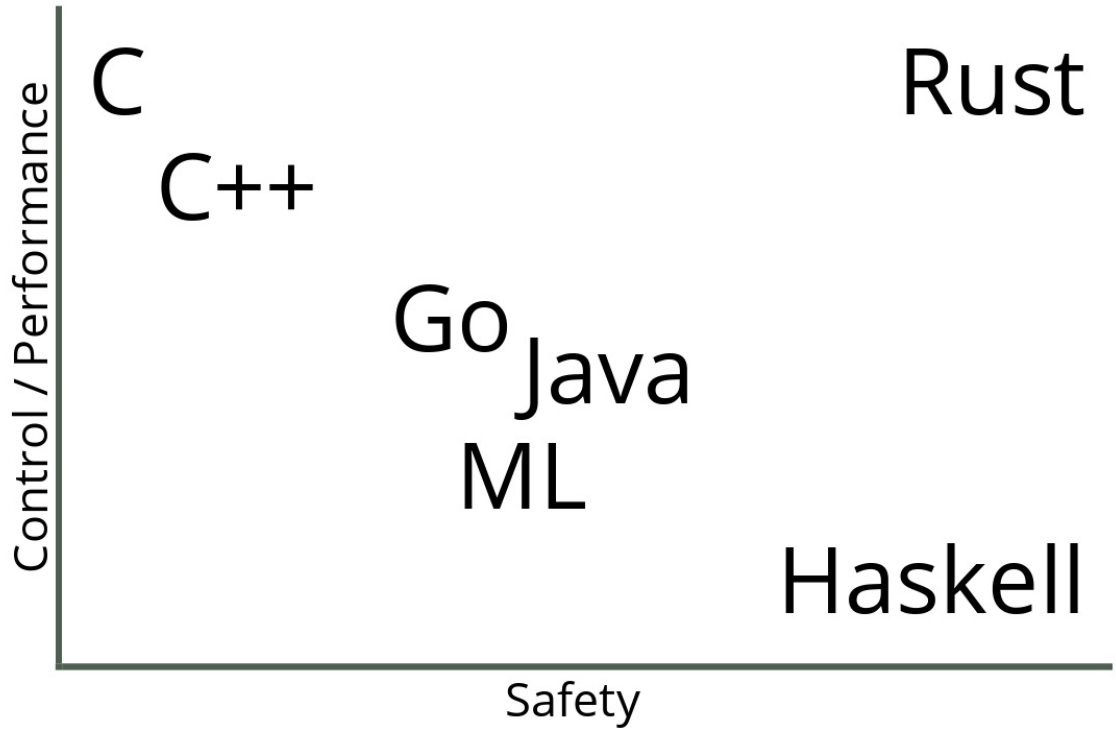
Like some of web developers, I truly bet on HTML5 over native apps on past years. I learned about HTML5, CSS3, RWD, Mobile First Design, UI/UX and etc. Now a days we have hundreds of front-end frameworks but still web technologies can't win over native apps, especially on performance.

It’s true that web technologies are slowly adapting for native app development via asmjs, NativeScript, Electron, WebAssembly, React Native but future web application development will be more complex because of Emerging Connected Cars and VR Ecosystems.

Also in the other hand, now we have much powerful alternatives for PHP, like node and Go. So as a PHP developer its time to learn something new, a new language. I chose Rust because it's an interesting language to learn and next generation browser engine, Servo is written using Rust.

Why Rust?

Rust initially designed and developed by Mozilla employee Graydon Hoare as a personal project. Mozilla began sponsoring the project in 2009 and announced it in 2010. But the first stable release, Rust 1.0 released on May 15, 2015.



The goal of Rust is to be a good language for creating highly concurrent and highly safe systems. Also as you can see on above diagram, Rust designed to provide speed and safety at the same time.

> "Rust is a systems programming language focused on three goals: safety, speed, and concurrency."
> __ Rust Documentation

Rust is very young and very modern language. It's a **compiled programming language** and it uses LLVM on its backend. Also Rust is a **multi-paradigm programming language**, it supports imperative procedural, concurrent actor, object-oriented and pure functional styles. It also supports generic programming and meta programming, in both static and dynamic styles.

Its design elements came from a wide range of sources.

- Abstract Machine Model : **C**
- Data types : **C, SML, OCaml, Lisp, Limbo**
- Optional Bindings : **Swift**
- Hygienic Macros : **Scheme**
- Functional Programming : **Haskell, OCaml, F#**
- Attributes : **ECMA**-335
- Memory Model and Memory Management : **C++, ML Kit, Cyclone**
- Type Classes : **Haskell**
- Crate : Assembly in the **ECMA**-335 CLI model
- Channels and Concurrency : **Newsqueak, Alef, Limbo**
- Message passing and Thread failure : **Erlang**

and etc.

Rust **doesn't use an automated garbage collection** system(GC) by default.

> One of Rust's most unique and compelling features is ownership, which uses to achieves memory safety. Rust creates memory pointers optimistically, checks memory pointers' limited accesses at the compiler time with the usage of References and Borrowing. And it does automatic compile time memory management by checking the Lifetimes.

Rust compiler observe the code at compiler time and help to prevent many types of errors that are possible to write in C++

Installation

There are many ways to install Rust on your system. For the moment the official way to install Rust is using Rustup.

Rustup installs The Rust Programming Language from the official release channels, enabling you to easily switch between stable, beta, and nightly compilers and keep them updated. It makes cross-compiling simpler with binary builds of the standard library for common platforms.

Rustup installs `rustc`, `cargo`, `rustup` and other standard tools to Cargo's `bin` directory. On Unix it is located at `$HOME/.cargo/bin` and on Windows at `%USERPROFILE%\.cargo\bin`. This is the same directory that `cargo install` will install Rust programs and Cargo plugins.

More information can be found on the Github page of Rustup project.

After installing Rust you can check the current version by typing `rustc --version` or `rustc -v` on your terminal to verify the success of the installation.

Hello World

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

`fn` means function. main function is the beginning of every Rust program.
`println!` prints text to the console and its *!* indicate that it's a macro instead of a function.

> Rust files should have .rs file extension and if you're using more than one word for the file name, follow the snake_case.

compiling via `rustc file.rs`
executing by `./file` on Linux and Mac or `file.exe` on Windows

These are the other usages of println! macro,

```
fn main() {
    println!("{}, {}!", "Hello", "world"); // Hello, world!
    println!("{0}, {1}!", "Hello", "world"); // Hello, world!
    println!("{greeting}, {name}!", greeting="Hello", name="world"); // Hello, world!

    println!("{:?}", [1,2,3]); // [1, 2, 3]
    println!("{:#?}", [1,2,3]);
    /*
        [
            1,
            2,
            3
        ]
    */

    //  format! macro is used to store the formatted STRING
    let x = format!("{}, {}!", "Hello", "world");
    println!("{}", x); // Hello, world!
}
```

Cargo, Crates and Basic Project Structure

Cargo is Rust's build-in Package Manager. But mainly it uses for,

▸ Create new project : `cargo new`
▸ Update dependencies : `cargo update`
▸ Build project : `cargo build`
▸ Build and run a project : `cargo run`
▸ Run tests : `cargo test`
▸ Generate documentation via rustdoc : `cargo doc`

Other than that there are some cargo commands, especially for publishing crates directly via cargo.

▸ `cargo login` : acquiring an API token
▸ `cargo package` : make the local create uploadable to crates.io
▸ `cargo publish` : make the local create uploadable to crates.io and upload the crate

 A crate is a package. Crates can be shared via Cargo.

A crate can produce an executable or a library. In other words, it can be a binary crate or a library crate.

1. `cargo new crate_name --bin` : produces an **executable**
2. `cargo new crate_name --lib` OR `cargo new crate_name` : produces a **library**

The first one generates,

```
├── Cargo.toml
└── src
    └── main.rs
```

and the second one generates,

```
├── Cargo.toml
└── src
    └── lib.rs
```

- **Cargo.toml**(capital c) is the configuration file which contains all of the metadata that Cargo needs to compile your project.

- **src** folder is the place to store the source code.
- Each crate has an implicit crate root/ entry point. **main.rs** is the crate root for a binary crate and **lib.rs** is the crate root for a library crate.

> When we build a binary crate via `cargo build` or `cargo run`, the executable file will be stored in **target/debug/** folder. But when build it via `cargo build --release` for a release it will be stored in **target/release/** folder.

This is how Cargo Docs describes about the recommended Project Layout,

```
.
├── Cargo.lock
├── Cargo.toml
├── benches
│   └── large-input.rs
├── examples
│   └── simple.rs
├── src
│   ├── bin
│   │   └── another_executable.rs
│   ├── lib.rs
│   └── main.rs
└── tests
    └── some-integration-tests.rs
```

▸ Source code goes in the `src` directory.
▸ The default library file is `src/lib.rs`.
▸ The default executable file is `src/main.rs`.
▸ Other executables can be placed in `src/bin/*.rs`.
▸ Integration tests go in the `tests` directory (unit tests go in each file they're testing).
▸ Examples go in the `examples` directory.
▸ Benchmarks go in the `benches` directory.

Comments and Documenting the code

```
// Line comments
/* Block comments */
```

Nested block comments are supported.

Always avoid block comments, Use line comments instead.

```
/// Line comments; document the next item
/** Block comments; document the next item */

//! Line comments; document the enclosing item
/*! Block comments; document the enclosing item !*/
```

Doc comments support Markdown notations. Using `cargo doc`, the HTML documentation can be generated from these doc comments. Let's see the difference between the two sets of doc comments.

```
/// This module contains tests
mod test {
    // ...
}


mod test {
    //! This module contains tests

    // ...
}
```

As you can see both use to document the same module. First comment has been added before the module while the second one has been added inside the module.

Only use //! to write crate and module-level documentation, nothing else. When using mod blocks, use /// outside of the block.

Also we can use **doc attributes** for documenting the code.

> An attribute is a general, free-form **metadatum** that is interpreted according to name, convention, and language and compiler version. Any item declaration may have an attribute applied to it.

In here each comments are equivalent to relevant data attributes.

```
/// Foo
#[doc="Foo"]

//! Foo
#![doc="Foo"]
```

Variable bindings , Constants & Statics

 In Rust variable are **immutable by default**, so we call them **Variable bindings**. To make them mutable, `mut` keyword is used.

 Rust is a **statically typed** language; It checks data type at compile time. But it **doesn't require you to actually type it when declare variable bindings**. On that case compiler checks the usage and set a better data type for it. But for **constants and statics you must annotate the type**. Types come after a colon(:)

- Variable bindings

```
let a = true;
let b: bool = true;

let (x, y) = (1, 2);

let mut z = 5;
z = 6;
```

- Constants

```
const N: i32 = 5;
```

- Statics

```
static N: i32 = 5;
```

let keyword is used in binding expressions. We can bind a name to a value or a function. Also because of left-hand side of a let expression is a 'pattern', you can bind multiple names to set of values or function values.

const keyword is used to define constants. It lives for the entire lifetime of a program but have no fixed address in memory. **static** keyword is used to define 'global variable' type facility. There is only one instance for each value, and it's at a **fixed location in memory**.

Always use const, instead of static. It's pretty rare that you actually want a memory location associated with your constant, and using a const allows for optimizations like constant propagation not only in your crate but also in downstream crates.

Usually statics are placed at top of the code file, outside the functions.

Functions

- Functions are declared with the keyword `fn`
- When using **arguments**, you **must declare data types**.
- By default functions **return empty tuple ()**. If you want to return a value, **return type must be specified** after **->**

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

//Function with arguments
fn print_sum(a: i8, b: i8) {
    println!("sum is: {}", a + b);
}

//Returning
fn plus_one(a: i32) -> i32 {
    a + 1 //no ; means an expression, return a+1
}

fn plus_two(a: i32) -> i32 {
    return a + 2; //return a+2 but bad practice,
    //should use only on conditional returnes, except it's last expression
}

// ▫ Function pointers, Usage as a Data Type
let b = plus_one;
let c = b(5); //6

let b: fn(i32) -> i32 = plus_one; //same, with type inference
let c = b(5); //6
```

Primitive Data Types

- **bool** : true or false

```
let x = true;
let y: bool = false;

// ▫ no TRUE, FALSE, 1, 0
```

- **char** : a single Unicode scalar value

```
let x = 'x';
let y = '😎';

// ▫ no "x", only single quotes
//because of Unicode support, char is not a single byte, but four.
```

- **i8 i16 i32 i64** : fixed size(bit) signed(+/-) integer types

DATA TYPE	MIN	MAX
i8	-128	127
i16	-32768	32767
i32	-2147483648	2147483647
i64	-9223372036854775808	9223372036854775807

Min and max values are based on IEEE standard for Binary Floating-Point Arithmetic; From **-2^{n-1} to $2^{n-1}-1$** . You can use **min_value()** and **max_value()** to find min and max of each integer type, ex. i8::min_value();

- **u8 u16 u32 u64** : fixed size(bit) unsigned(+) integer types

DATA TYPE	MIN	MAX
u8	0	255
u16	0	65535
u32	0	4294967295
u64	0	18446744073709551615

Same as signed numbers, min and max values are based on IEEE standard for Binary Floating-Point Arithmetic; From **0 to 2^n-1** . Same way you can use **min_value()** and **max_value()** to find min and max of each integer type, ex. u8::max_value();

- **isize** : variable sized signed(+/-) integer

Simply this is the data type to cover all signed integer types but memory allocates according to the size of a pointer. Min and max values are similar to i64 .

- **usize** : variable sized unsigned(+) integer

Simply this is the data type to cover all unsigned integer types but memory allocates according to the size of a pointer. Min and max values are similar to u64.

- **f32** : 32-bit floating point

Similar to float in other languages, **Single precision**.

Should avoid using this unless you need to reduce memory consumption badly or if you are doing low-level optimization, when targeted hardware not supports for double-precision or when single-precision is faster than double-precision on it.

- **f64** : 64-bit floating point

Similar to double in other languages, **Double precision**.

- **arrays** : fixed-size list of elements of same data type

```
let a = [1, 2, 3]; // a[0] = 1, a[1] = 2, a[2] = 3
let mut b = [1, 2, 3];

let c: [int; 3] = [1, 2, 3]; //[Type; NO of elements]

let d: ["my value"; 3]; //["my value", "my value", "my value"];

let e: [i32; 0] = []; //empty array

println!("{:?}", a); //[1, 2, 3]
println!("{:#?}", a);
//  [
//      1,
//      2,
//      3
//  ]
```

 Arrays are **immutable** by default and also **even with mut, its element count can not be changed**.

> If you are looking for a dynamic/growable array, you can use **Vec**. Vectors can contain any type of elements but all elements must be in the same data type.

- **tuples** : fixed-size ordered list of elements of different(or same) data types

```
let a = (1, 1.5, true, 'a', "Hello, world!");
// a.0 = 1, a.1 = 1.5, a.2 = true, a.3 = 'a', a.4 = "Hello, world!"

let b: (i32, f64) = (1, 1.5);

let (c, d) = b; // c = 1, d = 1.5
let (e, _, _, _, f) = a; //e = 1, f = "Hello, world!", _ indicates not interested of t
hat item

let g = (0,); //single-element tuple

let h = (b, (2, 4), 5); //((1, 1.5), (2, 4), 5)

println!("{:?}", a); //(1, 1.5, true, 'a', "Hello, world!")
```

 Tuples are also **immutable** by default and **even with mut, its element count can not be changed. Also if you want to change an element's value, new value should have the same data type of previous value**.

- **slice** : dynamically-sized reference to another data structure

Think you want to get/pass a part of an array or any other data structure. Instead of copy it to another array (or same data structure), Rust allows to create a view/reference to access only that part of data. And it can be mutable or not.

```
let a: [i32; 4] = [1, 2, 3, 4];//Parent Array

let b: &[i32] = &a; //Slicing whole array
let c = &a[0..4]; // From 0th position to 4th(excluding)
let d = &a[..]; //Slicing whole array

let e = &a[1..3]; //[2, 3]
let e = &a[1..]; //[2, 3, 4]
let e = &a[..3]; //[1, 2, 3]
```

- **str** : unsized UTF-8 sequence of Unicode string slices

```
let a = "Hello, world."; //a: &'static str
let b: &str = "こんにちは, 世界!";
```

 It's an **immutable/statically allocated slice** holding an **unknown sized sequence of UTF-8** code points stored in somewhere in memory. **&str** is used to borrow and assign the whole array to the given variable binding.

> A String is a **heap**-allocated string. This string is growable, and is also guaranteed to be UTF-8. They are commonly created by converting from a string slice using the **to_string()** or **String::from()** methods. ex: `“Hello”.to_string();` `String::from("Hello");`

In general, you should use **String** when you need **ownership**, and **&str** when you just need to **borrow a string**.

- **functions**

As we discussed on functions section, b is a function pointer, to plus_one function

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

let b: fn(i32) -> i32 = plus_one;
let c = b(5); //6
```

Operators

- **Arithmetic Operators** : + - * / %

```
let a = 5;
let b = a + 1; //6
let c = a - 1; //4
let d = a * 2; //10
let e = a / 2; // □ 2 not 2.5
let f = a % 2; //1

let g = 5.0 / 2.0; //2.5
```

> Also + is used for **array and string concatenation**

- **Comparison Operators** : == != < > <= >=

```
let a = 1;
let b = 2;

let c = a == b; //false
let d = a != b; //true
let e = a < b; //true
let f = a > b; //false
let g = a <= a; //true
let h = a >= a; //true

//
let i = true > false; //true
let j = 'a' > 'A'; //true
```

- **Logical Operators** : ! && ||

```
let a = true;
let b = false;

let c = !a; //false
let d = a && b; //false
let e = a || b; //true
```

> On integer types, ! inverts the individual bits in the two's complement representation of the value.

```
let a = !-2; //1
let b = !-1; //0
let c = !0; //-1
let d = !1; //-2
```

- **Bitwise Operators** : & | ^ << >>

```
let a = 1;
let b = 2;

let c = a & b; //0  (01 && 10 -> 00)
let d = a | b; //3  (01 || 10 -> 11)
let e = a ^ b; //3  (01 != 10 -> 11)
let f = a << b; //4  (add 2 positions to the end -> '01'+'00' -> 100)
let g = a >> a; //0  (remove 2 positions from the end -> 01 -> 0)
```

- **Assignment and Compound Assignment Operators**

The = operator is used to assign a name to a value or a function. Compound Assignment Operators are created by composing one of + - * / % & | ^ << >> operators with = operator.

```
let mut a = 2;

a += 5; //2 + 5 = 7
a -= 2; //7 - 2 = 5
a *= 5; //5 * 5 = 25
a /= 2; //25 / 2 = 12 not 12.5
a %= 5; //12 % 5 = 2

a &= 2; //10 && 10 -> 10 -> 2
a |= 5; //010 || 101 -> 111 -> 7
a ^= 2; //111 != 010 -> 101 -> 5
a <<= 1; //'101'+'0' -> 1010 -> 10
a >>= 2; //1010 -> 10 -> 2
```

- **Type Casting Operator** : as

```
let a = 15;
let b = (a as f64) / 2.0; //7.5
```

- **Borrowing and Dereference Operators** : & &mut *

The **& or &mut** operators are used for **borrowing** and ***** operator for **Dereferencing**.

> Usage of these operators is an advanced topic, for more information use Rust Reference Documentation.

Control Flows

- **if - else if - else**

```
// Simplest Example
let team_size = 7;
if team_size < 5 {
    println!("Small");
} else if team_size < 10 {
    println!("Medium");
} else {
    println!("Large");
}

// partially refactored code
let team_size = 7;
let team_size_in_text;
if team_size < 5 {
    team_size_in_text = "Small";
} else if team_size < 10 {
    team_size_in_text = "Medium";
} else {
    team_size_in_text = "Large";
}
println!("Current team size : {}", team_size_in_text);

//optimistic code
let team_size = 7;
let team_size_in_text = if team_size < 5 {
    "Small" //▫no ;
} else if team_size < 10 {
    "Medium"
} else {
    "Large"
};
println!("Current team size : {}", team_size_in_text);


let is_below_eighteen = if team_size < 18 { true } else { false };
```

□ Return data type should be same on each block, when using this as an expression.

- **match**

```
let tshirt_width = 20;
let tshirt_size = match tshirt_width {
    16 => "S", // check 16
    17 | 18 => "M", // check 17 and 18
    19 ... 21 => "L", // check from 19 to 21 (19,20,21)
    22 => "XL",
    _ => "Not Available",
};
println!("{}", tshirt_size); // L


let is_allowed = false;
let list_type = match is_allowed {
    true => "Full",
    false => "Restricted"
    // no default/ _ condition can be skipped
    // Because data type of is_allowed is boolean and all possibilities checked on con
ditions
};
println!("{}", list_type); // Restricted


let marks_paper_a: u8 = 25;
let marks_paper_b: u8 = 30;
let output = match (marks_paper_a, marks_paper_b) {
    (50, 50) => "Full marks for both papers",
    (50, _) => "Full marks for paper A",
    (_, 50) => "Full marks for paper B",
    (x, y) if x > 25 && y > 25 => "Good",
    (_, _) => "Work hard"
};
println!("{}", output); // Work hard
```

- **while**

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

// Usage of break and continue
let mut b = 0;
while b < 5 {
    if b == 0 {
        println!("Skip value : {}", b);
        b += 1;
        continue;
    } else if b == 2 {
        println!("Break At : {}", b);
        break;
    }
    println!("Current value : {}", b);
    b += 1;
}

// Outer break
let mut c1 = 1;
'outer_while: while c1 < 6 { //set label outer_while
    let mut c2 = 1;
    'inner_while: while c2 < 6 {
        println!("Current Value : [{}][{}]", c1, c2);
        if c1 == 2 && c2 == 2 { break 'outer_while; } //kill outer_while
        c2 += 1;
    }
    c1 += 1;
}
```

- **loop**

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

// Usage of break and continue
let mut a = 0;
loop {
    if a == 0 {
        println!("Skip Value : {}", a);
        a += 1;
        continue;
    } else if a == 2 {
        println!("Break At : {}", a);
        break;
    }
    println!("Current Value : {}", a);
    a += 1;
}

// Outer break
let mut b1 = 1;
'outer_loop: loop { //set label outer_loop
  let mut b2 = 1;
  'inner_loop: loop {
    println!("Current Value : [{}][{}]", b1, b2);
    if b1 == 2 && b2 == 2 {
        break 'outer_loop; // kill outer_loop
    } else if b2 == 5 {
        break;
    }
    b2 += 1;
  }
  b1 += 1;
}
```

- **for**

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

// Usage of break and continue
for b in 0..6 {
  if b == 0 {
    println!("Skip Value : {}", b);
    continue;
  } else if b == 2 {
    println!("Break At : {}", b);
    break;
  }
  println!("Current value : {}", b);
}

// Outer break
'outer_for: for c1 in 1..6 { //set label outer_for
  'inner_for: for c2 in 1..6 {
    println!("Current Value : [{}][{}]", c1, c2);
    if c1 == 2 && c2 == 2 { break 'outer_for; } //kill outer_for
  }
}


// Working with arrays/vectors
let group : [&str; 4] = ["Mark", "Larry", "Bill", "Steve"];

for n in 0..group.len() { //group.len() = 4 -> 0..4  check group.len()on each iteratio
n
  println!("Current Person : {}", group[n]);
}

for person in group.iter() { // group.iter() turn the array into a simple iterator
  println!("Current Person : {}", person);
}
```

Vectors

If you remember, array is a fixed-size list of elements, of same data type. Even with mut, it’s element count can not be changed. Vector is kind of a re-sizable array but all elements must be in the same type.

 It’s a generic type, written as Vec . T can have any type, ex. The type of a Vec of i32s is Vec. Also Vectors always allocate their data in dynamically allocated heap.

```
//Creating vectors - 2 ways
let mut a = Vec::new(); //1.with new() keyword
let mut b = vec![]; //2.using the vec! macro

//Creating with data types
let mut a2: Vec<i32> = Vec::new();
let mut b2: Vec<i32> = vec![];
let mut b3 = vec![1i32, 2, 3];//sufixing 1st value with data type

//Creating with data
let mut b4 = vec![1, 2, 3];
let mut b5: Vec<i32> = vec![1, 2, 3];
let mut b6  = vec![1i32, 2, 3];
let mut b7 = vec![0; 10]; //ten zeroes

//Accessing and changing exsisting data
let mut c = vec![5, 4, 3, 2, 1];
c[0] = 1;
c[1] = 2;
//c[6] = 2; can't assign values this way, index out of bounds
println!("{:?}", c); //[1, 2, 3, 2, 1]

//push and pop
let mut d: Vec<i32> = Vec::new();
d.push(1); //[1] : Add an element to the end
d.push(2); //[1, 2]
d.pop(); //[1] : : Remove an element from the end


//  Capacity and reallocation
let mut e: Vec<i32> = Vec::with_capacity(10);
println!("Length: {}, Capacity : {}", e.len(), e.capacity()); //Length: 0, Capacity :
10

// These are all done without reallocating...
for i in 0..10 {
    e.push(i);
}
// ...but this may make the vector reallocate
e.push(11);
```

 Mainly a vector represent 3 things; a pointer to the data, No of elements currently have(length), capacity (Amount of space allocated for any future elements). If the length of a vector exceeds its capacity, its capacity will be increased automatically. But its elements will be reallocated(which can be slow). So always use Vec::with_capacity whenever it’s possible.

> String data type is a UTF-8 encoded vector. But you can not index into a String because of encoding.

Vectors can be used with iterators in three ways,

```rust
let mut v = vec![1, 2, 3, 4, 5];

for i in &v {
    println!("A reference to {}", i);
}

for i in &mut v {
    println!("A mutable reference to {}", i);
}

for i in v {
    println!("Take ownership of the vector and its element {}", i);
}
```

Structs

 Structs are used to encapsulate related properties into one unified datatype.

By convention, the name of the struct starts with a capital letter and follows CamelCase.

There are 3 variants of structs,

1. C-like structs
    - one or more comma separated name:value pairs
    - brace-enclosed list
    - similar to classes (without it’s methods) in other languages like Java
    - because fields have names, we can access them through dot notation
2. Tuple structs
    - one or more comma separated values
    - parenthesized list like tuples
    - looks like a named tuples
3. Unit structs
    - a struct with no members at all
    - it defines a new type but it resembles an empty tuple, ()
    - rarely in use, useful with generics

 When regarding OOP in Rust, attributes and methods are placed separately on structs and traits. Structs contain only attributes, traits contain only methods. They are getting connected via impls

01. C-like structs

```
// Struct Declaration
struct Color {
    red: u8,
    green: u8,
    blue: u8
}

fn main() {
  // creating an instance
  let black = Color {red: 0, green: 0, blue: 0};

  // accessing it's fields, using dot notation
  println!("Black = rgb({}, {}, {})", black.red, black.green, black.blue); //Black = r
gb(0, 0, 0)

  // structs are immutable by default, use `mut` to make it mutable but doesn't suppor
t field level mutability
  let mut link_color = Color {red: 0,green: 0,blue: 255};
  link_color.blue = 238;
  println!("Link Color = rgb({}, {}, {})", link_color.red, link_color.green, link_colo
r.blue); //Link Color = rgb(0, 0, 238)

  // copy elements from another instance
  let blue = Color {blue: 255, .. link_color};
  println!("Blue = rgb({}, {}, {})", blue.red, blue.green, blue.blue); //Blue = rgb(0,
 0, 255)

  // destructure the instance using a `let` binding, this will not destruct blue insta
nce
  let Color {red: r, green: g, blue: b} = blue;
  println!("Blue = rgb({}, {}, {})", r, g, b); //Blue = rgb(0, 0, 255)

  // creating an instance via functions & accessing it's fields
  let midnightblue = get_midnightblue_color();
  println!("Midnight Blue = rgb({}, {}, {})", midnightblue.red, midnightblue.green, mi
dnightblue.blue); //Midnight Blue = rgb(25, 25, 112)

  // destructure the instance using a `let` binding
  let Color {red: r, green: g, blue: b} = get_midnightblue_color();
  println!("Midnight Blue = rgb({}, {}, {})", r, g, b); //Midnight Blue = rgb(25, 25,
112)
}

fn get_midnightblue_color() -> Color {
    Color {red: 25, green: 25, blue: 112}
}
```

02. Tuple structs

 When a tuple struct has only one element, we called it ‘newtype’ pattern. Because it helps to create a new type.

```
struct Color (u8, u8, u8);
struct Kilometers(i32);

fn main() {
  // creating an instance
  let black = Color (0, 0, 0);

  // destructure the instance using a `let` binding, this will not destruct black inst
ance
  let Color (r, g, b) = black;
  println!("Black = rgb({}, {}, {})", r, g, b); //black = rgb(0, 0, 0);

  //newtype pattern
  let distance = Kilometers(20);
  // destructure the instance using a `let` binding
  let Kilometers(distance_in_km) = distance;
  println!("The distance: {} km", distance_in_km); //The distance: 20 km
}
```

03. Unit structs

This is rarely useful on its own, but in combination with other features, it can become useful.

> ex: A library may ask you to create a structure that implements a certain trait to handle events. If you don’t have any data you need to store in the structure, you can create a unit-like struct.

```
struct Electron;

fn main() {
  let x = Electron;
}
```

Enums

 An enum is a single type. It contains variants, which are possible values of the enum at a given time. For example,

```
enum Day {
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}

// Day is the enum
// Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday are the variants
```

 Variants can be accessed through :: notation , ex. Day::Sunday

 Each enum variant can have,

- no data (unit variant)
- unnamed ordered data (tuple variant)
- named data (struct variant)

```
enum FlashMessage {
  Success, //a unit variant
  Warning{ category: i32, message: String }, //a struct variant
  Error(String) //a tuple variant
}

fn main() {
  let mut form_status = FlashMessage::Success;
  print_flash_message(form_status);

  form_status = FlashMessage::Warning {category: 2, message: String::from("Field X is
required")};
  print_flash_message(form_status);

  form_status = FlashMessage::Error(String::from("Connection Error"));
  print_flash_message(form_status);
}

fn print_flash_message(m : FlashMessage) {
  // pattern matching with enum
  match m {
    FlashMessage::Success =>
      println!("Form Submitted correctly"),
    FlashMessage::Warning {category, message} => //Destructure, should use same field
names
      println!("Warning : {} - {}", category, message),
    FlashMessage::Error(msg) =>
      println!("Error : {}", msg)
  }
}
```

Generics

> Sometimes, when writing a function or data type, we may want it to work for multiple types of arguments. In Rust, we can do this with generics.

The concept is, instead of declaring a specific data type we use an uppercase letter(or CamelCase identifier). ex, instead x : u8 we use x : T . but we have to inform to the compiler that T is a generic type(can be any type) by adding at first.

```
// generalizing functions
//-----------------------
fn takes_anything<T>(x: T) { // x has type T, T is a generic type
}

fn takes_two_of_the_same_things<T>(x: T, y: T) { // both x and y has same type
}

fn takes_two_things<T, U>(x: T, y: U) { // multiple types
}


// generalizing structs
//---------------------
struct Point<T> {
  x: T,
  y: T,
}

fn main() {
  let point_a = Point { x: 0, y: 0 }; // T is a int type
  let point_b = Point { x: 0.0, y: 0.0 }; // T is a float type
}

//  When addding an implementation for a generic struct, the type parameters should be
 declared after the impl as well
// impl<T> Point<T> {


// generalizing enums
//-------------------
enum Option<T> {
    Some(T),
    None,
}

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

> Above Option and Result types are kind of special generic types which are already defined in Rust's standard library.
>
> - An optional value can have either Some value or no value/ None.
> - A result can represent either success/ Ok or failure/ Err

```
// 01 - - - - - - - - - - - - - - - - - - - - -
fn getIdByUsername(username: &str) -> Option<usize> {
    //if username can be found in the system, set userId
        return Some(userId);
    //else
        None
}

// So on above function, instead of setting return type as usize
// set return type as Option<usize>
//Instead of return userId, return Some(userId)
// else None (remember? last return statement no need return keyword and ending ;)

// 02 - - - - - - - - - - - - - - - - - - - - -
struct Task {
    title: String,
    assignee: Option<Person>,
}

// Instead of assignee: Person, we use Option<Person>
//Because the task has not been assigned to a specific person

// - - - - - - - - - - - - - - - - - - - - - -
//when using Option types as return types on functions
// we can use pattern matching to catch the relevant return type(Some/None) when calli
ng them

fn main() {
    let username = "anonymous"
    match getIdByUsername(username) {
        None => println!("User not found"),
        Some(i) => println!("User Id: {}", i)
    }
}
```

> The Option type is a way to use Rust's type system to express the possibility of absence. Result expresses the possibility of error.

```
// - - - - - - - - - - - - - - - - - - - - - - -
fn get_word_count_from_file(file_name: &str) -> Result<u32, &str> {
  //if the file is not found on the system, return error
    return Err("File can not be found!")
  //else, count and return the word count
    //let mut word_count: u32; ....
    Ok(word_count)
}

// on above function,
// instead panic(break) the app, when a file can not be found; return Err(something)
// or when it could get the relevant data; return Ok(data)


// - - - - - - - - - - - - - - - - - - - - - - -
// we can use pattern matching to catch the relevant return type(Ok/Err) when calling
it

fn main() {
    let mut file_name = "file_a";
    match get_word_count_from_file(file_name) {
        Ok(i) => println!("Word Count: {}", i),
        Err(e) => println!("Error: {}", e)
    }
}
```

> Many useful methods have been implemented around Option and Result types. More information can be found on std::option::Option and std::result::Result pages on Rust doc.

 Also more practical examples of options & results can be found on Error Handling section in Rust doc.

Impls & Traits

When we discussed about C-like structs, I mentioned that those are similar to classes in other languages like Java, but without their methods. impls are used to define methods for Rust structs and enums.

Traits are kind of similar to interfaces in other languages like Java. They are used to define the functionality a type must provide. Multiple traits can be implemented to a single type.

□□□ But traits can also include default implementations of methods. Default methods can be override when implementing types.

```
// 01 ----------------------------------------
// Adding methoids directly; without using traits

struct Player {
    first_name: String,
    last_name: String,
}

impl Player {
    fn full_name(&self) -> String {
        format!("{} {}", self.first_name, self.last_name)
    }
}

fn main() {
    let player_1 = Player {
        first_name: "Rafael".to_string(),
        last_name: "Nadal".to_string(),
    };

    println!("Player 01: {}", player_1.full_name());
}


// 02 ----------------------------------------
// Adding methoids by implementing traits

struct Player {
    first_name: String,
    last_name: String,
}

trait FullName {
    fn full_name(&self) -> String;
```

```
}

impl FullName for Player {
    fn full_name(&self) -> String {
        format!("{} {}", self.first_name, self.last_name)
    }
}

fn main() {
    let player_2 = Player {
        first_name: "Roger".to_string(),
        last_name: "Federer".to_string(),
    };

    println!("Player 02: {}", player_2.full_name());
}


// 03 ----------------------------------------
// a trait with default implementations of methods

trait Foo {
    fn bar(&self);
    fn baz(&self) { println!("We called baz."); }
}

// ----------------------------------------------
// ▫ In case 01, implementation must appear in the same crate as the self type

//  And also in Rust, new traits can be implemented for existing types even for types
like i8, f64 and etc.
// Same way existing traits can be implemented for new types you are creating.
// But we can not implement existing traits into existing types

//  Other than functions, traits can contain constants and types

//  Traits also supprot generics; should specify after the trait name like generic fun
ctions

trait From<T> {
    fn from(T) -> Self;
}
    impl From<u8> for u16 {
        //...
    }
    impl From<u8> for u32{
        //...
    }
```

 As you can see methods take a special first parameter, the type itself. It can be either self, &self, or &mut self . self if it's a value on the stack(taking ownership), &self if it's a reference, and &mut self if it's a mutable reference.

> Some other languages support static methods. At such times, we call a function directly through the class without creating an object. In Rust, we call them Associated Functions. we use :: instead of . when calling them from struct. ex. Person::new("Elon Musk Jr");

```
struct Player {
    first_name: String,
    last_name: String,
}

impl Player {
    fn new(first_name: String, last_name: String) -> Player {
        Player {
            first_name : first_name,
            last_name : last_name,
        }
    }

    fn full_name(&self) -> String {
        format!("{} {}", self.first_name, self.last_name)
    }
}

fn main() {
    let player_name = Player::new("Serena".to_string(), "Williams".to_string()).full_n
ame();
    println!("Player: {}", player_name);
}

// we have used :: notation for `new()` and . notation for `full_name()`

//  Also in here we have used `Method Chaining`. Instead of using two statements for n
ew() and full_name()
// calls, we can use a single statement with Method Chaining.
// ex. player.add_points(2).get_point_count();
```

 Traits may inherit from other traits.

```
trait Person {
    fn full_name(&self) -> String;
}

    trait Employee : Person { //Employee inherit from person trait
      fn job_title(&self) -> String;
    }

    trait ExpatEmployee : Employee + Expat { //ExpatEmployee inherit from Employee and
 Expat traits
      fn additional_tax(&self) -> f64;
    }
```

While Rust favors static dispatch, it also supports dynamic dispatch through a mechanism called ‘trait objects.’

> Dynamic dispatch is the process of selecting which implementation of a polymorphic operation (method or function) to call at run time.

```
trait GetSound {
    fn get_sound(&self) -> String;
}

struct Cat {
    sound: String,
}
    impl GetSound for Cat {
        fn get_sound(&self) -> String {
            self.sound.clone()
        }
    }

struct Bell {
    sound: String,
}
    impl GetSound for Bell {
        fn get_sound(&self) -> String {
            self.sound.clone()
        }
    }


fn make_sound<T: GetSound>(t: &T) {
    println!("{}!", t.get_sound())
}

fn main() {
    let kitty = Cat { sound: "Meow".to_string() };
    let the_bell = Bell { sound: "Ding Dong".to_string() };

    make_sound(&kitty); // Meow!
    make_sound(&the_bell); // Ding Dong!
}
```

Ownership

```
fn main() {
    let a = [1, 2, 3];
    let b = a;
    println!("{:?} {:?}", a, b); // [1, 2, 3] [1, 2, 3]
}

fn main() {
    let a = vec![1, 2, 3];
    let b = a;
    println!("{:?} {:?}", a, b); // Error; use of moved value: `a`
}
```

In the above examples, we are just trying to **assign the value of ‘a’ to ‘b’** . Almost the same code in both code blocks, but having **two different data types**. And the second one gives an error. This is because of the **Ownership**.

 Variable bindings have **ownership** of what they’re bound to. A piece of data can only have **one owner at a time**. When a binding goes out of scope, Rust will free the bound resources. This is how Rust achieves **memory safety**.

> Ownership (noun)
> The act, state, or right of possessing something.

 When assigning a variable binding to another variable binding **or when passing it to a function**(Without referencing), if its data type is a

1. **Copy Type**
    - Bound resources are **made a copy and assign** or pass it to the function.
    - The ownership state of the original bindings are set to **“copied” state**.
    - **Mostly Primitive types**
2. **Move type**
    - Bound resources are **moved** to the new variable binding and we **can not access the original variable binding** anymore.
    - The ownership state of the original bindings are set to **“moved” state**.
    - **Non-primitive types**

> The functionality of a type is handled by the traits which have been implemented to it. By default, variable bindings have ‘move semantics.’ However, if a type implements **core::marker::Copy trait** , it has a 'copy semantics'.

So in the above second example, ownership of the Vec object moves to “b” and “a” doesn’t have any ownership to access the resource.

Borrowing

In real life applications, most of the times we have to pass variable bindings to other functions or assign them to another variable bindings. In this case we **referencing** the original binding; **borrow** the data of it.

> Borrow (verb)
> To receive something with the promise of returning it.

 There are two types of Borrowing,

1. **Shared Borrowing** `(&T)`
    - A piece of data can be **borrowed by a single or multiple users**, but **data should not be altered**.
2. **Mutable Borrowing** `(&mut T)`
    - A piece of data can be **borrowed and altered by a single user**, but the data should not be accessible for any other users at that time.

 And there are **very important rules** regarding borrowing,

1. One piece of data can be borrowed **either** as a shared borrow **or** as a mutable borrow **at a given time. But not both at the same time**.
2. Borrowing **applies for both copy types and move types**.
3. The concept of **Liveness** ↴

```
fn main() {
  let mut a = vec![1, 2, 3];
  let b = &mut a;  //  &mut borrow of a starts here
                   //  ⋮
  // some code     //  ⋮
  // some code     //  ⋮
}                  //  &mut borrow of a ends here


fn main() {
  let mut a = vec![1, 2, 3];
  let b = &mut a;  //  &mut borrow of a starts here
  // some code

  println!("{:?}", a); // trying to access a as a shared borrow, so giving error
}                  //  &mut borrow of a ends here


fn main() {
  let mut a = vec![1, 2, 3];
  {
    let b = &mut a;  //  &mut borrow of a starts here
    // any other code
  }                  //  &mut borrow of a ends here

  println!("{:?}", a); // allow to borrow a as a shared borrow
}
```

Let's see how to use shared and mutable borrowings in examples.

- Examples for Shared Borrowing

```
fn main() {
    let a = [1, 2, 3];
    let b = &a;
    println!("{:?} {}", a, b[0]); // [1, 2, 3] 1
}


fn main() {
    let a = vec![1, 2, 3];
    let b = get_first_element(&a);

    println!("{:?} {}", a, b); // [1, 2, 3] 1
}

fn get_first_element(a: &Vec<i32>) -> i32 {
    a[0]
}
```

- Examples for Mutable Borrowing

```
fn main() {
    let mut a = [1, 2, 3];
    let b = &mut a;
    b[0] = 4;
    println!("{:?}", b); // [4, 2, 3]
}


fn main() {
    let mut a = [1, 2, 3];
    {
        let b = &mut a;
        b[0] = 4;
    }

    println!("{:?}", a); // [4, 2, 3]
}


fn main() {
    let mut a = vec![1, 2, 3];
    let b = change_and_get_first_element(&mut a);

    println!("{:?} {}", a, b); // [4, 2, 3] 4
}

fn change_and_get_first_element(a: &mut Vec<i32>) -> i32 {
    a[0] = 4;
    a[0]
}
```

Lifetimes

When we are dealing with references, we have to make sure that the referencing data stay alive until we are stop using the references.

Think,

- We have a variable binding, “**a**”.
- We are referencing the value of “a”, from another variable binding “**x**”. We have to make sure that “a” **lives** until we stop using “x”

> **Memory management** is a form of resource management applied to computer memory. Up until the mid-1990s, the majority of programming languages used **Manual Memory Management** which **requires the programmer to give manual instructions** to identify and deallocate unused objects/ garbage. Around 1959 John McCarthy invented **Garbage collection**(GC), a form of **Automatic Memory Management**(AMM). It determines what memory is no longer used and frees it automatically instead of relying on the programmer. However **Objective-C and Swift** provide similar functionality through **Automatic Reference Counting**(ARC).

In Rust,

- A resource can only have **one owner** at a time. When it goes **out of the scope**, Rust removes it from the Memory.
- When we want to reuse the same resource, we are **referencing** it/ **borrowing** its content.
- When dealing with **references**, we have to specify **lifetime annotations** to provide instructions for the **compiler** to set **how long** those referenced resources **should be alive**.
- But because of lifetime annotations make **code more verbose**, in order to make **common patterns** more ergonomic, Rust allows lifetimes to be **elided/omitted** in `fn` definitions. In this case, the compiler assigns lifetime annotations **implicitly**.

Lifetime annotations are **checked at compile-time**. Compiler checks when a data is used for the first and the last times. According to that, Rust manages memory in **run time**. This is the major reason of having **slower compilation times** in Rust.

> - Unlike C and C++, **usually** Rust doesn't explicitly drop values at all.
> - Unlike GC, Rust doesn't place deallocation calls where the data is no longer referenced.
> - Rust places deallocation calls where the data is about to go out of the scope and then enforces that no references to that resource exist after that point.

Lifetimes are denoted with an apostrophe. By convention, a lowercase letter is used for naming. Usually **starts with** `'a` and **follows alphabetic order** when we need to add **multiple lifetime** annotations.

When using references,

. On **Function Declaration**

- Input and output parameters with references should attach lifetimes after `&` sign. ex `..(x: &'a str)` , `..(x: &'a mut str)`
- After the function name, we should mention that the given lifetimes are generic types. ex `fn foo<'a>(..)` , `fn foo<'a, 'b>(..)`

```
// no inputs, return a reference
fn function<'a>() -> &'a str {}

// single input
fn function<'a>(x: &'a str) {}

// single input and output, both has same lifetime
// output should live at least as long as input exists
fn function<'a>(x: &'a str) -> &'a str {}

// multiple inputs, only one input and the output share same lifetime
// output should live at least as long as y exists
fn function<'a>(x: i32, y: &'a str) -> &'a str {}

// multiple inputs. both inputs and the output share same lifetime
// output should live at least as long as x and y exist
fn function<'a>(x: &'a str, y: &'a str) -> &'a str {}

// multiple inputs. inputs can have diffent lifetimes
// output should live at least as long as x exists
fn function<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {}
```

. On **Struct or Enum Declaration**

- Elements with references should attach lifetimes after `&` sign.
- After the name of the struct or enum, we should mention that the given lifetimes are generic types.

```
// single element
// data of x should live at least as long as Struct exists
struct Struct<'a> {
    x: &'a str
}

// multiple elements
// data of x and y should live at least as long as Struct exists
struct Struct<'a> {
    x: &'a str,
    y: &'a str
}


// variant with single element
// data of the variant should live at least as long as Enum exists
enum Enum<'a> {
    Variant(&'a Type)
}
```

. With **Impls and Traits**

```
struct Struct<'a> {
    x: &'a str
}
    impl<'a> Struct<'a> {
        fn function<'a>(&self) -> &'a str {
            self.x
        }
    }


struct Struct<'a> {
    x: &'a str,
    y: &'a str
}
    impl<'a> Struct<'a> {
        fn new(x: &'a str, y: &'a str) -> Struct<'a> { //no need to specify <'a> after
 new; impl already has it
          Struct {
              x : x,
              y : y
          }
        }
    }


//
impl<'a> Trait<'a> for Type
impl<'a> Trait for Type<'a>
```

. With **Generic Types**

```
//
fn function<F>(f: F) where for<'a> F: FnOnce(&'a Type)
struct Struct<F> where for<'a> F: FnOnce(&'a Type) { x: F }
enum Enum<F> where for<'a> F: FnOnce(&'a Type) { Variant(F) }
impl<F> Struct<F> where for<'a> F: FnOnce(&'a Type) { fn x(&self) -> &F { &self.x } }
```

Lifetime Elision

As I mentioned earlier, in order to make **common patterns** more ergonomic, Rust allows lifetimes to be **elided/omitted**. This process is called **Lifetime Elision**.

For the moment Rust supports Lifetime Elisions only on `fn` definitions. But in the future it will support for `impl` headers as well.

 lifetime annotations of `fn` definitions can be elided
if its **parameter list** has either,

- **only one input parameter passes by reference**.
- a parameter with **either** `&self` **or &mut self** reference.

```
fn triple(x: &u64) -> u64 { //only one input parameter passes by reference
    x * 3
}


fn filter(x: u8, y: &str) -> &str { // only one input parameter passes by reference
    if x > 5 { y } else { "invalid inputs" }
}


struct Player<'a> {
    id: u8,
    name: &'a str
}
    impl<'a> Player<'a> { //so far Lifetime Elisions are allowed only on fn definition
s; but in the future they might support on impl headers as well.
        fn new(id: u8, name: &str) -> Player { //only one input parameter passes by re
ference
            Player {
                id : id,
                name : name
            }
        }

        fn heading_text(&self) -> String { // a fn definition with &self (or &mut self
) reference
            format!("{}: {}", self.id, self.name)
        }
    }

fn main() {
    let player1 = Player::new(1, "Serena Williams");
    let player1_heading_text = player1.heading_text()
    println!("{}", player1_heading_text);
}
```

> In the Lifetime Elision process of fn definitions,
>
> - Each parameter passes by reference is got a distinct lifetime annotation. ex. `..(x: &str, y: &str)` `..<'a, 'b>(x: &'a str, y: &'b str)`
> - If the parameter list has only one parameter passes by reference, that lifetime is assigned to all elided lifetimes in the return values of that function. ex. `..(x: i32, y: &str) -> &str` `..<'a>(x: i32, y: &'a str) -> &'a str`
> - Even it has multiple parameters pass by reference, if one of them has &self or &mut self, the lifetime of self is assigned to all elided output lifetimes. ex. `impl Impl{ fn function(&self, x: &str) -> &str {} }` `impl<'a> Impl<'a>{ fn function(&'a self, x: &'b str) -> &'a str {} }`
> - For all other cases, we have to write lifetime annotations manually.

'static

 `'static` lifetime annotation is a **reserved** lifetime annotation. These references are valid for the entire program. They are saved in the data segment of the binary and the data referred to will never go out of scope.

Let's see how to use lifetime annotations in examples.

```
fn greeting<'a>() -> &'a str {
  "Hi!"
}


fn fullname<'a>(fname: &'a str, lname: &'a str) -> String {
  format!("{} {}", fname, lname)
}


struct Person<'a> {
    fname: &'a str,
    lname: &'a str
}
  impl<'a> Person<'a> {
      fn new(fname: &'a str, lname: &'a str) -> Person<'a> { //no need to specify <'a>
 after new; impl already has it
          Person {
              fname : fname,
              lname : lname
          }
      }

      fn fullname(&self) -> String {
          format!("{} {}", self.fname , self.lname)
      }
  }

fn main() {
    let player = Person::new("Serena", "Williams");
    let player_fullname = player.fullname();

    println!("Player: {}", player_fullname);
}
```

For more information you can go through,

- Rust Documentation & Upcoming New Rust Documentation
- The Rust Reference & Rust Syntax Index
- Rust by Example
- Rust Programming Course of University of Pennsylvania

To get a help,

- The Rust Community & #rust-beginners IRC
- Sub-Reddit, /r/rust

Books

- Programming Rust
- Rust Essentials