Introduction to Rust

Adityo Pratomo (@kotakmakan)

This Talk is Also Available at

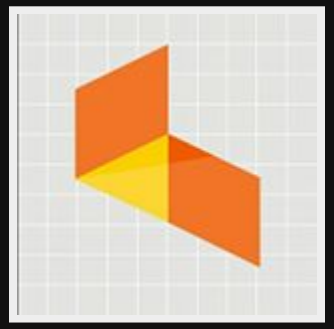
github.com/froyoframework/rust-intro/slide

Background

- Chief Academic Officer at Framework
- Chief Technology Officer at Labtek Indie
- Certified Unity Developer

Framework

- Providing software development course, training and workshop
- Based in BSD



Codewise, I'm

- Generalist
- Creative Coder
- C/C++, Java, JS

When I Meet Rust

- Fast and safe system programming language
- Like C++, without segfaults (yum!)
- Better handling of reference and pointers
- Mixture of imperative and functional paradigm

What to Do with Rust?

- System programming
- Something low-level enough to benefit from precise memory management
  - Web Browser (Mozilla Firefox)
  - Distributed storage system (Dropbox)
  - 3D Games
  - Device drive
  - Operating System
- General tool

Rust's Killer Features

- Concept of "Borrowing and Ownership"
  - Type safety
  - Memory safety
- Zero-cost abstractions
- Pattern matching

How Rust is Fast and Safe?

- Extensive compiler checking
- Fast: No garbage collection, Rust automatically detect when to free memory
  - Lifetime
  - Ownership of data
- Safe: No data race, guaranteed data lifetime, no dangling-pointer
  - Ownership and Borrowing only allows one mutable reference (write access)

Also in Rust

- Built-in unit testing
- Cargo: Rust's built-in package and build manager
- Helpful error messages in compiler

A Tour of Rust's Syntax

github.com/froyoframework/rust-intro/basic-rust-sample

Variable

- Variable by default is immutable
- A binding to value exists

```
let angka = 9;
let salam = "Selamat datang, Android no ";
let halo = format!("{} {}", salam, angka);
println!("{:?}", halo);
```

Function

- Return value in function is explicitly denoted using arrow
- The returned value is the last variable stated without semicolon

```
let angka_saya = calc(angka);

fn calc(x: i32) -> i32 {
    let y;
    match x {
        1...40 => y = 34,
        _ => y = 2,
    }
    y
}
```

Struct

- A simple data structure that contains key-value entities
- Each key-value can use different data Type

```
struct Pemain {
    nama: String,
    umur: i32,
    gol: i32,
}

let buffon = Pemain {nama: "Buffon".to_string(), umur: 39,
```

Make Struct with Function

```
fn tambah_pemain(nama_: &str, umur_: i32, gol_: i32) -> Pe

let pemain_saya = Pemain { nama: nama_.to_string(), umur:

};

pemain_saya
}

let ronaldo = tambah_pemain("Ronaldo", 31, 510);
```

Vector

- Array-like structure that can be dynamically manipulated during runtime
- Can contain anything, from integers, floats, Strings, to Structs

```
let deret = vec![1, 2, 3];
let mut himpunan = Vec::new();
himpunan.push(5);
himpunan.push(6)
```

Vector of Structs

```
fn tambah_para_pemain() -> Vec<Pemain> {
    let ronaldo = tambah_pemain("Ronaldo", 31, 510);
    let bacca = tambah_pemain("Bacca", 31, 235);
    let payet = tambah_pemain("Payet", 28, 150);

    let mut pemain_favorit = Vec::new();
    pemain_favorit.push(ronaldo);
    pemain_favorit.push(bacca);
    pemain_favorit.push(payet);

    pemain_favorit
}

let pemain_keren = tambah_para_pemain();
```

Ownership and Borrowing

- A key concept that ensures safety and concurrency in Rust
- Basically everytime a variable is used, its ownership is transferred to the one uses/calls it
- When an ownership is transferred, the old owner can't use the entity anymore
- Checked at compile time

```
let pemain_bola = pemain_keren;
println!("pemain pertama adalah: {}", pemain_keren[0].nama
```

Ownership and Borrowing

- To solve the previous problem, Rust introduces Borrowing
- This means that a variable can be borrowed, thus, it's still valid for being used elsewhere
- This is accomplished by simple referencing that intended variable, thus the term "reference"

```
let pemain_bola = &pemain_keren;
println!("pemain pertama adalah: {}", pemain_keren[0].nama
```

Ownership and Borrowing

- Another thing that's correlated with referencing, is dereferencing
- This means, accessing the value of a referenced variable
- By default, a reference is immutable
- Change the value of a referenced variable by using mutable reference

```
let mut a = 90;
let b = &mut a;
*b += 1;
println!("{}", b); //prints 91
```

Ownership and Borrowing

- A consequence of borrowing, is the concept of a borrow lifetime
- This is denoted by a curly brace

```
let mut a = 90;
{
    let b = &mut a;      // a dipinjam di sini
    *b += 9;             // isi a diakses di sini
}                        // peminjaman a berakhir di sini
println!("{}", a);
```

Ownership and Borrowing

- To ensure safety, the main rule in borrowing is:
  - one or more references (&T) to a resource,
  - exactly one mutable reference (&mut T).

A Simple Web Service

github.com/lunchboxav/rust-intro/webserver

Why Learn New Language?

- Gains new perspective on how things are done
- Gains new understanding on programming itself
- Make old and new things in a different way

Tips for Learning Rust

- Katas: learn by making familiar things
- Try make small tool to replace your existing tool
- Consult the documentation
- Ask people on SO/Twitter
- Organize a community

Learning Resources

- The Rust Book (https://doc.rust-lang.org/book/)
- Rust 101 (https://www.ralfj.de/projects/rust-101/main.html)
- Rust Tutorial (http://aml3.github.io/RustTutorial/html/toc.html)
- Rust Syntax (https://gist.github.com/brson/9dec4195a88066fa42e6)
- Rust By Example (http://rustbyexample.com/expression.html)
- Rustlings, small Rust Exercises (https://github.com/carols10cents/rustlings)
- 24 Days of Rust (http://zsiciarz.github.io/24daysofrust/)
- Rust FFI Omnibus (http://jakegoulding.com/rust-ffi-omnibus/)
- New Rustacean (http://www.newrustacean.com)

Thank You

didit@froyo.co.id