INTRODUCTION TO RUST

Kelompok 5 - PF A

Anggota Kelompok

Bagas Yoga Pratama Pramudika - 11231014

Michael Peter Valentino Situmeang - 11231039

Muhammad Zaki Afriza 11231067

Rafi Baydar Athaillah 11231081

W11: Introduction to Rust
OUTLINE

01
Data Type

02
Function

03
Ownership & Borrowing (Immutability)

04
Closures & Iterator

Data Type

1) Scalar Types

2) Compound Types

3) String

4) Koleksi (std::collections)

5) Tipe Kustom

6) References

7) Fungsi & Tipe Fungsi

8) Fungsi Anonim

Data Type
1) Scalar Types

1. Integer

Integer Bertanda (iSize): i8, i16, i32, i64, i128 (Range: $-2^{(N-1)}$ s.d $2^{(N-1)} - 1$)
Integer Tak Bertanda (uSize): u8, u16, u32, u64, u128 (Range: 0 s.d $2^{N} - 1$)

```
let a: i32 = -42;
let b: u64 = 1_000;
let hex = 0xff_u8;
let bin = 0b1010_i16;
let oktal = 0o177_i8;
```

2. Float

f32 = $\pm 1.18 \times 10^{-38}$ s.d. $\pm 3.4 \times 10^{38}$
f64 = $\pm 2.23 \times 10^{-308}$ s.d. $\pm 1.80 \times 10^{308}$ (Default)

```
let x: f64 = 3.14;
let y = 2.5f32;
```

Data Type
1) Scalar Types

3. Boolean

True / False

```
let done: bool = true;
let sleep = false;
```

4. Character

Character/emoji

```
let c: char = '🦀' ;
```

5. Unit Type

() melambangkan “ada hasil, tapi tidak ada informasi yang perlu disampaikan.”

```
let u: : ( ) = ( );
```

Data Type

1) Scalar Types

2) Compound Types

3) String

4) Koleksi (std::collections)

5) Tipe Kustom

6) References

7) Fungsi & Tipe Fungsi

8) Fungsi Anonim

Data Type
2) Compound Types

1. Tuple

Gabungan dari Beberapa Tipe Skalar (Heterogen)

```
let t: ( i32, f64, char ) = ( 42, 3.14, 'a' ) ;
```

2. Array

Gabungan dari Beberapa Tipe Skalar (Homogen)

```
let arr: [ i32; 3 ] = [ 1, 2, 3 ];
```

3. Slice

Batasan pada list (Indeks)

```
let mut arr = [10, 20, 30, 40, 50];
let s_all: &[i32]   = &arr[ .. ];
let s_mid: &[i32]   = &arr[1..4];
let s_prefix: &[i32]= &arr[..3];
let s_suffix: &[i32]= &arr[3..];
```

Data Type

1) Scalar Types
2) Compound Types
3) String
4) Koleksi (std::collections)
5) Tipe Kustom
6) References
7) Fungsi & Tipe Fungsi
8) Fungsi Anonim

Data Type
3) String

```
let s: &str = "halo";
let mut ss = String::from("halo");
ss.push(' ');
ss.push_str("rust");
```

Data Type

1) Scalar Types

2) Compound Types

3) String

4) Koleksi (std::collections)

5) Tipe Kustom

6) References

7) Fungsi & Tipe Fungsi

8) Fungsi Anonim

Data Type
4) Koleksi (std::collections)

bukan built-in

1. Vector (Vec<T>)

```
let mut v: Vec<i32> = vec![1, 2, 3];
v.push(4);
let last = v.pop();
println! ( " { : ? } { : ? }", v, last);
```

2. HashMap (HashMap<K, V>)

```
use std::collections::HashMap;

let mut m: HashMap<&str, i32> = HashMap::new();
m.insert("alice", 21);
let age = m.get("alice");
```

Data Type

1) Scalar Types
2) Compound Types
3) String
4) Koleksi (std::collections)
5) Tipe Kustom
6) References
7) Fungsi & Tipe Fungsi
8) Fungsi Anonim

Data Type

5) Tipe Kustom

1. Struct

```
#[derive(Debug)]
struct Person { name: String, age: u8 }
fn main() {
    let p = Person { name: "Bagas".into(), age: 20 };
    println!("{:?}", p);
}
```

2. Tuple struct

```
#[derive(Debug)]
struct Point(u32, i32);

fn main() {
    let p = Point(3, -4);
    println!("p = {:?}", p);
    println!("x={}, y={}", p.0, p.1);
}
```

Data Type

1) Scalar Types
2) Compound Types
3) String
4) Koleksi (std::collections)
5) Tipe Kustom
6) References
7) Fungsi & Tipe Fungsi
8) Fungsi Anonim

Data Type

6) References

&T (shared/immutable)
&mut T (mutable)

```
fn read_then_increase(read_only: &i32, target: &mut i32) {
    println!("dibaca via &T = {}", *read_only);
    *target += 1;
}
```

```
fn main() {
    let a = 10;
    let mut b = 7;

    read_then_increase(&a, &mut b);

    println!("a (tetap) = {}", a);
    println!("b (bertambah) = {}", b);
}
```

Data Type

1) Scalar Types

2) Compound Types

3) String

4) Koleksi (std::collections)

5) Tipe Kustom

6) References

7) Fungsi & Tipe Fungsi

8) Fungsi Anonim

Data Type
7) Fungsi dan Tipe Fungsi

```
// Fungsi biasa
fn add(a: i32, b: i32) -> i32 { a + b }
fn sub(a: i32, b: i32) -> i32 { a - b }
fn mul(a: i32, b: i32) -> i32 { a * b }

// Parameter bertipe fungsi
fn apply(a: i32, b: i32, op: fn(i32, i32) -> i32) -> i32 {
    op(a, b)
}

// Mengembalikan fungsi (tipe fungsi)
fn pick(op: char) -> fn(i32, i32) -> i32 {
    match op {
        '+' => add,
        '-' => sub,
        _   => mul, // default
    }
}
```

```
fn main() {
    println!("add(2, 3) = {}", add(2, 3));
    println!("sub(7, 4) = {}", sub(7, 4));
    println!("mul(6, 5) = {}", mul(6, 5));

    let op1 = pick('+');
    let op2 = pick('-');
    let op3 = pick('*'); // fallback ke mul
    println!("pick('+')(8, 2) = { }", op1(8, 2));
    println!("pick('-')(8, 2) = { }", op2(8, 2));
    println!("pick('*')(8, 2) = { }", op3(8, 2));
}
```

Data Type

1) Scalar Types

2) Compound Types

3) String

4) Koleksi (std::collections)

5) Tipe Kustom

6) References

7) Fungsi & Tipe Fungsi

8) Fungsi Anonim

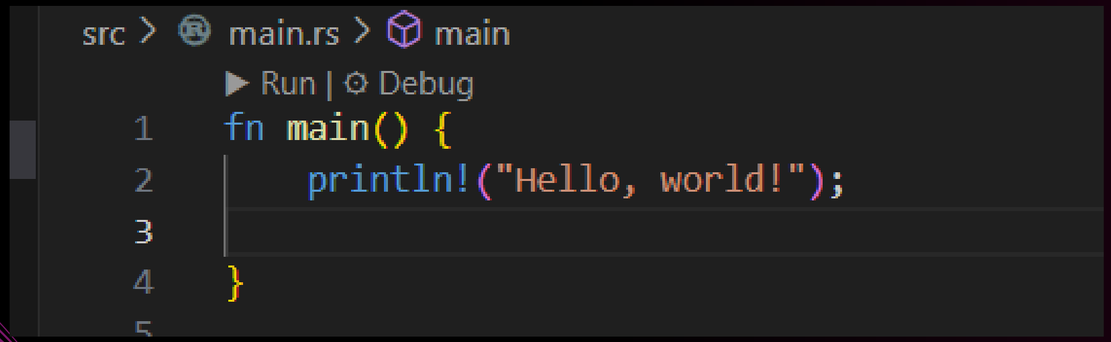
Data Type
8) Fungsi Anonim

```
fn main() {
    let factor = 10;
    let add = |x: i32| x + factor; // tipe parameter diinferensikan
    println!("add(5) = {}", add(5));
    println!("add(7) = {}", add(7));
    println!("factor masih bisa dipakai = {}", factor);
}
```

Function

Fungsi sangat umum digunakan dalam kode Rust.

Rust mendefinisikan fungsi dengan kata kunci fn



Konvensi penamaan: snake_case (huruf kecil + underscore).

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

Function

Mendefinisikan fungsi di Rust dengan memasukkan fn diikuti oleh nama fungsi dan sepasang kurung buka-tutup.

Memanggil fungsi yang telah didefinisikan dengan memasukkan namanya diikuti oleh sepasang kurung buka-tutup.

```
Run | Debug
fn main() {
    println!("Hello, world!");

    another_function();
}

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

```

Output

```
PS D:\Mata Kuliah Semester 5\PF\rust\prak> cargo run
   Compiling prak v0.1.0 (D:\Mata Kuliah Semester 5\PF
    Finished `dev` profile [unoptimized + debuginfo] t
     Running `target\debug\prak.exe`
Hello, world!
Another function.
PS D:\Mata Kuliah Semester 5\PF\rust\prak>
```

Function

Parameters

Mendefinisikan fungsi dengan parameter, memberikan nilai konkret untuk parameter-parameter

```rust
▶ Run | ⚙ Debug
fn main() {
    another_function(5);
}

fn another_function(x: i32) {
    println!("The value of x is: {x}");
}
```

Output

```
PS D:\Mata Kuliah Semester 5\PF\rust\
   Compiling prak v0.1.0 (D:\Mata Kul
    Finished `dev` profile [unoptimiz
     Running `target\debug\prak.exe`
The value of x is: 5
```

```rust
▶ Run | ⚙ Debug
fn main() {
    add_numbers(x: 10, y: 20);
}

fn add_numbers(x: i32, y: i32) {
    println!("Jumlahnya adalah: {}", x + y);
}
```

Output

```
POSTMAN CONSOLE  PROBLEMS  TERMINAL  OUTPUT
PS D:\Mata Kuliah Semester 5\PF\rust\prak> carg
   Compiling prak v0.1.0 (D:\Mata Kuliah Semest
    Finished `dev` profile [unoptimized + debug
     Running `target\debug\prak.exe`
Jumlahnya adalah: 30
PS D:\Mata Kuliah Semester 5\PF\rust\prak>
```

Function

Statement vs Expression

Pernyataan adalah instruksi yang melakukan tindakan tertentu dan tidak mengembalikan nilai

```
▶ Run | ⚙ Debug
fn main() {
    let y: i32 = 6;
}
```

```
PS D:\Mata Kuliah Semester 5\PF\rust\prak> cargo run
   Compiling prak v0.1.0 (D:\Mata Kuliah Semester 5\PF\rust\prak)
warning: unused variable: `y`
  --> src\main.rs:45:9
   |
45 |     let y = 6;
   |         ^ help: if this is intentional, prefix it with an underscore: `_y`
   |
   = note: `#[warn(unused_variables)]` (part of `#[warn(unused)]`) on by default

warning: `prak` (bin "prak") generated 1 warning
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.59s
     Running `target\debug\prak.exe`
PS D:\Mata Kuliah Semester 5\PF\rust\prak>
```

Pernyataan tidak mengembalikan nilai. Oleh karena itu, Anda tidak dapat mengaitkan pernyataan let ke variabel lain, seperti yang dicoba dalam kode berikut;

```
▶ Run | ⚙ Debug
fn main() {
    let x: bool = (let y: i32 = 6);
}
```

```
PS D:\Mata Kuliah Semester 5\PF\rust\prak> cargo run
   Compiling prak v0.1.0 (D:\Mata Kuliah Semester 5\PF\rust\prak)
error: expected expression, found `let` statement
  --> src\main.rs:49:14
   |
49 |     let x = (let y = 6);
   |              ^^^
   |
   = note: only supported directly in conditions of `if` and `while` expressions

warning: unnecessary parentheses around assigned value
  --> src\main.rs:49:13
   |
49 |     let x = (let y = 6);
   |             ^         ^
   |
   = note: `#[warn(unused_parens)]` (part of `#[warn(unused)]`) on by default
help: remove these parentheses
   |
49 -     let x = (let y = 6);
49 +     let x = let y = 6;
   |

warning: `prak` (bin "prak") generated 1 warning
error: could not compile `prak` (bin "prak") due to 1 previous error; 1 warning emitted
PS D:\Mata Kuliah Semester 5\PF\rust\prak>
```

Function

Statement vs Expression

Expression adalah potongan kode yang menghasilkan nilai. Bisa berupa angka, operasi aritmatika, pemanggilan fungsi yang mengembalikan nilai, atau bahkan blok {}
Tidak selalu diakhiri dengan ;

```
▶ Run | ⚙ Debug
fn main() {
    let y: i32 = {
        let x: i32 = 3;
        x + 1
    };

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

Output

```
PS D:\Mata Kuliah Semester 5\PF\rus
   Compiling prak v0.1.0 (D:\Mata K
    Finished `dev` profile [unoptim
     Running `target\debug\prak.exe
The value of y is: 4
PS D:\Mata Kuliah Semester 5\PF\rus
```

Function

Return Values

Fungsi dapat mengembalikan nilai ke kode yang memanggilnya. Kita tidak memberi nama pada nilai yang dikembalikan, tetapi kita harus mendeklarasikan tipenya setelah tanda panah (->).

```
▶ Run | ⚙ Debug
fn main() {
    let x: i32 = plus_one(5);

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

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

Output

```
PS D:\Mata Kuliah Semester 5\PF\ru
   Compiling prak v0.1.0 (D:\Mata
    Finished `dev` profile [unopti
     Running `target\debug\prak.ex
The value of x is: 6
PS D:\Mata Kuliah Semester 5\PF\ru
```

Ownership

Konsep Dasar Ownership

Rust mengatur memori tanpa garbage collector lewat ownership system.

Aturannya:

1. Setiap nilai punya satu owner.
2. Hanya satu owner aktif dalam satu waktu.
3. Saat owner keluar scope → memori otomatis dibersihkan (drop()).

Ownership mencegah bug seperti double free dan use-after-free.

```
{
    let s = String::from("hello");
} // s keluar scope → drop() otomatis
```

Ownership

Move & Copy Semantics

Move

Move terjadi ketika sebuah variabel baru mengambil alih ownership data dari variabel sebelumnya.
Setelah data berpindah, variabel asal tidak lagi valid dan tidak bisa digunakan.
Mekanisme ini memastikan hanya ada satu pemilik data di satu waktu, sehingga mencegah bug seperti double free atau use-after-free.

Copy

Copy terjadi ketika data disalin langsung ke variabel lain tanpa memindahkan ownership.
Hanya tipe data sederhana yang disimpan di stack (seperti i32, char, bool) yang bisa di-copy, karena biayanya murah dan aman.
Setelah disalin, kedua variabel tetap valid dan memiliki salinan nilai masing-masing.

Ownership

Move & Copy Semantics

Move

```
let s1 = String::from("hello");
let s2 = s1;   // ownership pindah

// println!("{s1}"); ❌ invalid, s1 sudah dipindah
```

Copy

```
let x = 5;
let y = x;     // disalin, bukan dipindah
println!("x = {x}, y = {y}"); // ✅ keduanya valid
```

Borrowing

Borrowing & References (Immutability)

Dalam Rust, borrowing berarti meminjam data tanpa mengambil ownership dari pemilik aslinya. Hal ini dilakukan menggunakan reference (&T), yang memungkinkan fungsi mengakses data tanpa memindahkannya. Secara default, reference bersifat immutable, sehingga data yang dipinjam tidak dapat diubah. Mekanisme ini membuat Rust fleksibel sekaligus aman dari bug memori dan data race.

```
fn calculate_length(s: &String) -> usize {
    s.len()
}

fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("{s1} has length {len}");
}
```

Borrowing

Mutable References & Borrowing Rules

Selain reference biasa yang bersifat immutable, Rust juga mendukung mutable reference (&mut T) yang memungkinkan fungsi meminjam data dan mengubah isinya tanpa mengambil ownership. Namun, Rust memiliki aturan ketat untuk menjaga keamanan memori: hanya boleh ada satu mutable reference aktif dalam satu waktu, dan tidak boleh ada mutable serta immutable reference ke data yang sama secara bersamaan. Aturan ini mencegah data race dan memastikan tidak ada dua bagian kode yang mengubah data secara bersamaan. Dengan pendekatan ini, Rust mampu memberikan fleksibilitas pemrograman tanpa mengorbankan kestabilan dan keamanan memori.

Borrowing

Mutable References & Borrowing Rules

```
fn change(some_string: &mut String) {
    some_string.push_str(", world");
}

fn main() {
    let mut s = String::from("hello");
    change(&mut s);
    println!("{s}");
}
```

```
let mut s = String::from("hello");
let r1 = &mut s;
let r2 = &mut s; // ❌ Error: cannot borrow `s` as mutable more than once
println!("{r1}, {r2}");
```

Closures

Closures adalah fungsi anonim yang dapat Anda simpan dalam variabel atau teruskan sebagai argumen ke fungsi lain.

Closures Type Inference dan Annotations

Perbedaan utama antara fungsi (menggunakan fn) dan closures adalah pada sintaksis dan inferensi tipe.

- Fungsi (fn): Wajib mendeklarasikan tipe parameter dan tipe return value secara eksplisit.
- Closures: Biasanya tidak memerlukan anotasi tipe. Compiler Rust dapat menyimpulkan (infer) tipe parameter dan return value secara otomatis berdasarkan konteks.

```
fn add_one(x: i32) -> i32 { x + 1 }
let add_one_closure = |x: i32| -> i32 { x + 1 };
let add_one_inferred = |x| x + 1;
```

Contoh Penggunaan Closures

```
fn main() {
    let add_one = |x| x + 1;

    let angka = 5;
    let hasil = add_one(angka);

    println!("Hasil dari {} + 1 adalah {}", angka, hasil);

    let hasil_lain = add_one(10);
    println!("Hasil dari 10 + 1 adalah {}", hasil_lain);
}
```

Standard Output

```
Hasil dari 5 + 1 adalah 6
Hasil dari 10 + 1 adalah 11
```

Contoh Penggunaan Closures (Error)

```
let example_closure = |x| x;
let s = example_closure(String::from("hello"));
let n = example_closure(5);
```

```
   Compiling playground v0.0.1 (/playground)
error[E0308]: mismatched types
 --> src/main.rs:4:29
  |
4 |     let n = example_closure(5);
  |             --------------- ^ expected `String`, found integer
  |             |
  |             arguments to this function are incorrect
  |
note: expected because the closure was earlier called with an argument of type `String`
 --> src/main.rs:3:29
  |
```

Iterators

Iterator adalah pola yang memungkinkan Anda melakukan tugas pada serangkaian item secara berurutan.

Di Rust, iterator bersifat lazy (malas). Artinya, mereka tidak akan berpengaruh apa-apa sampai Anda memanggil metode yang menggunakannya.

Contoh Penggunaan Iterator

Iterator paling sering digunakan dalam perulangan for. Perulangan for akan secara implisit mengambil kepemilikan iterator dan mengonsumsinya.

```
fn main() {
    let v1 = vec![1, 2, 3];
    let v1_iter = v1.iter();

    for val in v1_iter {
        println!("Got: {val}");
    }
}
```

Standard Output

```
Got: 1
Got: 2
Got: 3
```

Method yang mengkonsumsi Iterator

Metode yang menghabiskan iteratornya dan menghasilkan nilai akhir (bukan iterator). Ini yang "memicu" iteratornya untuk jalan.
Contoh: collect(), sum(), count().

```
fn main() {
    let angka = vec![1, 2, 3, 4, 5];
    let angka_dikali_dua: Vec<_> = angka
        .iter()
        .map(|x| x * 2)
        .collect();
    println!("Hasil kali dua: {:?}", angka_dikali_dua)
}
```

```
Hasil kali dua: [2, 4, 6, 8, 10]
```

Iterator Adapters

Adapter adalah metode yang mengubah iterator jadi iterator baru.
Contoh: map(), filter()

```
fn main(){
    let data_lain = vec![10, 21, 30, 41, 50];

        let total_angka_genap: i32 = data_lain
            .iter()
            .filter(|&&x| x % 2 == 0)
            .sum();

        println!("Total angka genap: {}", total_angka_genap);
}
```

```
Total angka genap: 90
```

Iterator Consumers and Adapters

Adapter adalah metode yang mengubah iterator jadi iterator baru.
Contoh: map(), filter()

Consumers adalah metode yang menghabiskan iteratornya dan menghasilkan nilai akhir (bukan iterator). Ini yang "memicu" iteratornya untuk jalan.
Contoh: collect(), sum(), count().

Contoh

```
fn main(){
    let data_lain = vec![10, 21, 30, 41, 50];

        let total_angka_genap: i32 = data_lain
            .iter()
            .filter(|&&x| x % 2 == 0)
            .sum();

        println!("Total angka genap: {}", total_angka_genap);
}
```

```
Total angka genap: 90
```

Sekian dan Terimakasih

Ada Pertanyaan?