INTRODUCTION TO RUST

Presented by Kelompok 5

MEET OUR TEAM

ADJI ZAHRA MAHDIYYAH 11231004

MUHAMMAD AZKA YUNASTIO 11231036

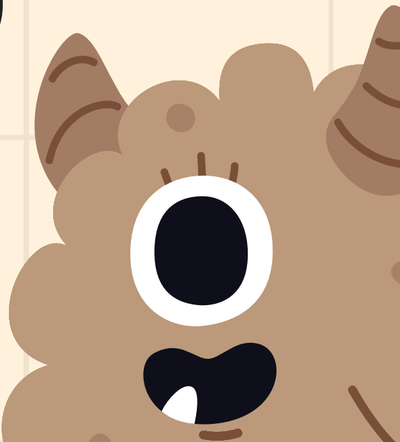
MUHAMMAD FACHRUDY AL-GHIFARI 11231054

OKTIARA AZZAHRA RAHMADINA 11231076

ZAKARIA FATTAWARI 11231092

OUTLINE

1. INTRODUCTION
2. DATA TYPE
3. FUNCTION
4. OWNERSHIP
5. BORROWING
6. CLOSURE
7. ITERATOR



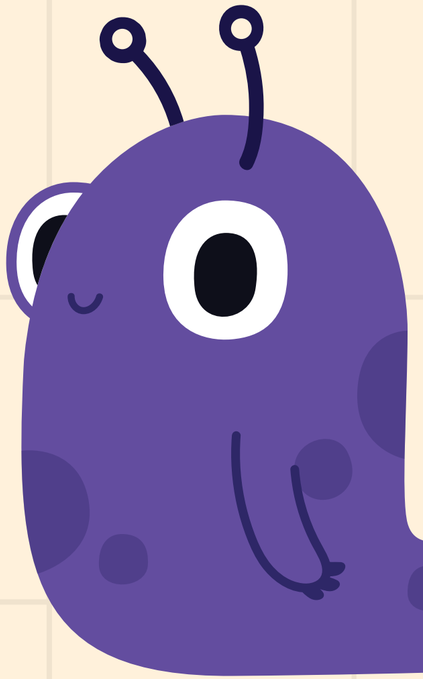
INTRODUCTION

DATA TYPE : SCALAR

1. Integer

Ukuran	Signed	Unsigned
8-bit	i8	u8
16-bit	i16	u16
32-bit	i32	u32

```rust
fn main() {
    let a: i32 = 100;
    let b: u8 = 255;

    println!("Nilai a: {a}");
    println!("Nilai b: {b}");

}
```



```
PS C:\Users\LENOVO\belajar_rust> cargo run
   Compiling belajar_rust v0.1.0 (C:\Users\LENOVO\belajar_rust)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.56s
     Running `target\debug\belajar_rust.exe`
Nilai a: 100
Nilai b: 255
```

DATA TYPE : SCALAR

2. Floating-Point

Rust punya dua tipe floating-point:

- f32 → 32-bit, kurang presisi tapi lebih ringan
- f64 → 64-bit, lebih presisi (dan default di Rust)

```rust
fn main() {
    let x: f64 = 3.14;
    let y: f32 = -10.5;

    println!("Nilai x: {x}");
    println!("Nilai y: {y}");
}
```

```
PS C:\Users\LENOVO\belajar_rust> cargo run
   Compiling belajar_rust v0.1.0 (C:\Users\LENOVO\belajar_rust)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.86s
     Running `target\debug\belajar_rust.exe`
Nilai x: 3.14
Nilai y: -10.5
```

DATA TYPES : SCALAR

3. Boolean

Boolean hanya ada dua nilai: true atau false.

```rust
fn main() {
    let is_rust_awesome: bool = true;
    let is_learning_tired: bool = false;

    println!("Apakah Rust keren? {is_rust_awesome}");
    println!("Apakah belajar itu capek? {is_learning_tired}");
}
```

```
PS C:\Users\LENOVO\belajar_rust> cargo run
   Compiling belajar_rust v0.1.0 (C:\Users\LENOVO\belajar_rust)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.44s
     Running `target\debug\belajar_rust.exe`
Apakah Rust keren? true
Apakah belajar itu capek? false
PS C:\Users\LENOVO\belajar_rust>
```

DATA TYPES : SCALAR

4. Character: Satu Karakter

Tipe char di Rust mewakili satu karakter Unicode. Jadi bukan cuma huruf latin, bisa saja emoji, huruf Arab, simbol matematika, dan sebagainya.

```rust
fn main() {
    let letter: char = 'R';
    let number: char = '8';
    let emoji: char = '😄';

    println!("Karakter: {letter}, {number}, {emoji}");
}
```

```
PS C:\Users\LENOVO\belajar_rust> cargo run
   Compiling belajar_rust v0.1.0 (C:\Users\LENOVO\belajar_rust)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 1.00s
     Running `target\debug\belajar_rust.exe`
Karakter: R, 8, 😄
```

DATA TYPE : COMPOUND

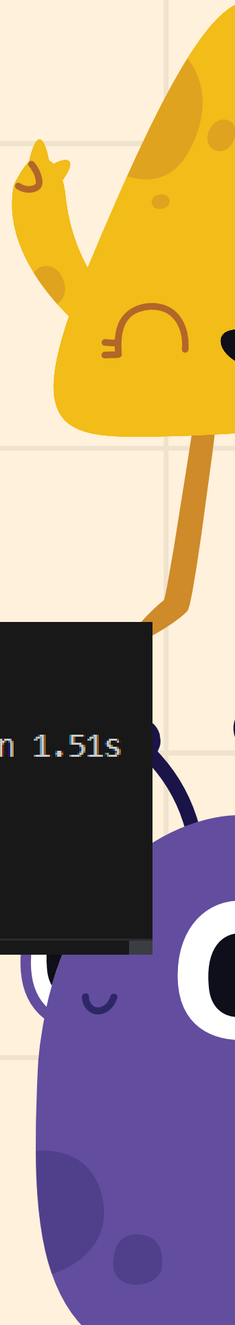
1. Tuple

Cara untuk mengelompokkan beberapa nilai menjadi satu tipe data gabungan. Berbeda dengan array, tiap elemen di dalam tuple bisa memiliki tipe data yang berbeda.

- Hal penting yang perlu diingat:
- Panjang tuple tetap (fixed) — tidak bisa diubah setelah dideklarasikan.
- Elemen-elemen tuple ditulis dalam tanda kurung ( ), dipisahkan oleh koma.

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

    // Destructuring
    let (x, y, z) = tup;
    println!("Dari destructuring: x={x}, y={y}, z={z}");

    // Akses langsung dg indeks
    println!("Dari akses indeks: {}, {}, {}", tup.0, tup.1, tup.2);
}
```

```
PS C:\Users\LENOVO\belajar_rust> cargo run
   Compiling belajar_rust v0.1.0 (C:\Users\LENOVO\belajar_rust)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 1.51s
     Running `target\debug\belajar_rust.exe`
Dari destructuring: x=500, y=6.4, z=1
Dari akses indeks: 500, 6.4, 1
PS C:\Users\LENOVO\belajar_rust>
```



DATA TYPE : COMPOUND

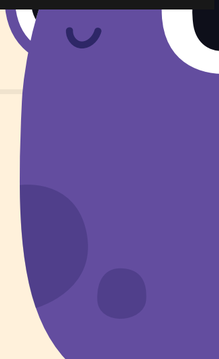
2. Array

Cara lain untuk menyimpan banyak nilai, tapi ada perbedaan penting dari tuple:

- Semua elemen dalam array harus memiliki tipe yang sama.
- Ukuran array juga tetap — tidak bisa berubah setelah dibuat.

```rust
fn main() {
    let angka = [10, 20, 30, 40, 50];

    let pertama = angka[0];
    let ketiga = angka[2];

    println!("Elemen pertama: {pertama}");
    println!("Elemen ketiga: {ketiga}");
}
```

```
PS C:\Users\LENOVO\belajar_rust> cargo run
   Compiling belajar_rust v0.1.0 (C:\Users\LENOVO\belajar_rust)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.77s
     Running `target\debug\belajar_rust.exe`
Elemen pertama: 10
Elemen ketiga: 30
```



FUNCTION IN RUST

Fungsi di Rust didefinisikan dengan kata kunci fn, diikuti nama fungsi yang menggunakan gaya penulisan snake case yang berarti semua huruf kecil dan kata dipisahkan dengan garis bawah. Setelah itu terdapat tanda kurung yang bisa berisi parameter, lalu badan fungsi ditulis di dalam kurung kurawal yang menandai bagian awal dan akhir fungsi

Contoh program yang berisi definisi fungsi :

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

```
[Running] cd "c:\Users\Oktiara Azzahran\pf\src\" && rustc main.rs && "c:\Users\
Hello, world!
```

Kita bisa memanggil fungsi yang sudah didefinisikan dengan menuliskan namanya diikuti tanda kurung. Karena another_function sudah didefinisikan, ia bisa dipanggil dari dalam fungsi main.

Contoh pemanggilan fungsi lain dalam fungsi main

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

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

```
[Running] cd "c:\Users\Oktiara Azzahrah\pf\src\" && rustc main.rs && "c:\
Hello, world!
Another function.
```

PARAMETER

Kita bisa mendefinisikan fungsi untuk memiliki parameter, yaitu variabel khusus yang merupakan bagian dari tanda tangan fungsi.

Contoh versi fungsi another_function dengan satu parameter:

```rust
fn main() {
    another_function(5);
}
fn another_function(x: i32) {
    println!("Nilai x adalah: {}", x);
}
```

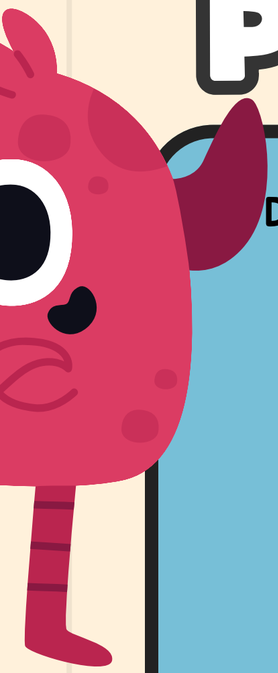
```
[Running] cd "c:\Users\Oktiara Azzahrah\pf\src\" && rustc main.rs && "c:\
Nilai x adalah: 5
```

Contoh versi fungsi another_function dengan dua parameter:

```rust
fn main() {
    greet(name: "Budi", age: 25)
}

fn greet(name: &str, age: u32) {
    println!("Halo, namaku {} dan aku berusia {} tahun.", name, age);
}
```

```
Azzahrah\pf\src\"tempCodeRunnerFile
Halo, namaku Budi dan aku berusia 25 tahun.
```

PERNYATAAN DAN EKSPRESI



Dalam pemrograman Rust, setiap baris kode yang kita tulis termasuk dalam dua kategori: Statement atau Expression

- Statement (Pernyataan) adalah instruksi yang melakukan suatu aksi atau tugas, namun tidak menghasilkan nilai kembalian, contohnya :

  let y = 6; (Tidak menghasilkan nilai kembalian)

- Expression adalah kode yang mengevaluasi dan menghasilkan sebuah nilai, contohnya :

  5 + 3 (Operasi matematika yang menghasilkan nilai 8)

Kunci membedakannya: perhatikan titik koma (;). Jika ada titik koma di akhir baris, itu statement. Jika tanpa titik koma, itu expression yang akan mengembalikan nilai.

FUNGSI DENGAN NILAI KEMBALI

Fungsi dengan nilai kembali adalah fungsi yang menghasilkan sebuah nilai (output) untuk digunakan di bagian lain program

- Fungsi dapat mengembalikan nilai ke pemanggilnya. Kita tidak memberi nama nilai kembali tapi harus mendeklarasikan tipenya setelah tanda panah. Nilai yang dikembalikan fungsi adalah nilai ekspresi terakhir di badan fungsi.

Caranya:

- Tulis tanda panah -> beserta tipe data return setelah nama fungsi.
- Nilai yang dikembalikan fungsi adalah hasil dari ekspresi terakhir di dalam fungsi (tanpa titik koma), atau bisa menggunakan keyword return.

Contoh fungsi dengan nilai kembali

```rust
fn tambah(a: i32, b: i32) -> i32 {
    a + b
}

▶ Run | ⚙ Debug
fn main() {
    let hasil: i32 = tambah(a: 5, b: 3);
    println!("Hasil penjumlahan: {}", hasil);
}
```

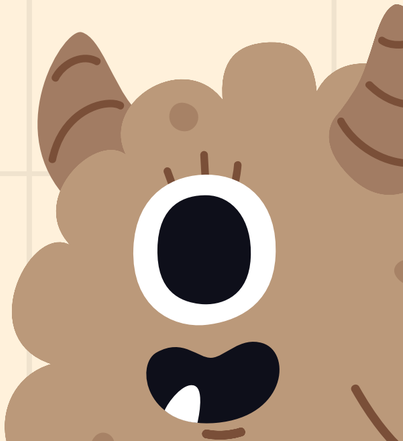
- Fungsi tambah yang mengembalikan nilai.
- Nilai kembali adalah ekspresi terakhir (a + b).
- Main hanya menerima dan memakai hasil nilai balik.

Jadi, fungsi tambah lah yang mengembalikan nilai ke pemanggilnya.

OWNERSHIP





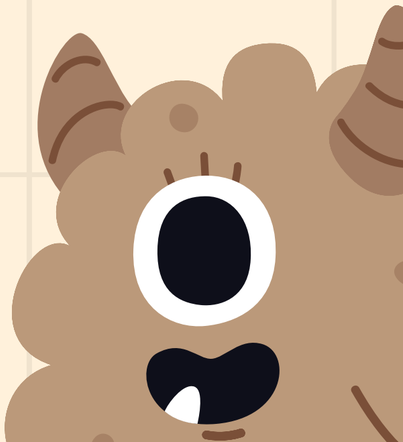
OWNERSHIP: MEMORY MANAGEMENT

Semua bahasa pemrograman memiliki caranya sendiri dalam melakukan pengelolaan memory atau memory management. Ada beberapa macam metode manajemen memori yang diterapkan pada bahasa pemrograman, di antaranya adalah berikut:

1. Garbage Collection (GC),
2. Manual Memory Management, dan
3. Ownership Rules





OWNERSHIP

Ownership merupakan kumpulan atau seperangkat aturan yang ada di Rust yang dijadikan acuan oleh compiler dalam pengelolahan memory. Aturan dalam Ownership:

Rule 1

Semua nilai/data/value di Rust hanya memiliki satu owner.

Rule 2

Hanya ada satu owner yang berjalan dalam satu waktu.

Rule 3

Saat ada owner yang keluar dari scope, maka memorynya akan dibersihkan.

OWNERSHIP

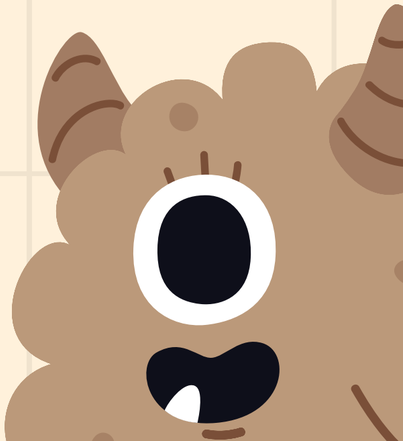
```
▶ Run | ⚙ Debug
fn main() {
    { // s nya belum valid, karena disini belum ada deklarasi apapun.
        let s: String = String::from("hello"); // disini, s nya itu adalah owner dan mulai valid di sini.
        println!("{}", s);
    } // scope nya dah keluar, maka s nya ga lagi valid dan selesai.
}
```

OWNERSHIP: MOVE AND COPY SEMANTICS

Move adalah pemindahan kepemilikan nilai ke variabel lain. Setelah dipindah, variabel lama tidak boleh dipakai lagi. Ini terjadi otomatis saat assignment atau saat nilai dikirim ke fungsi pada tipe data seperti String. Alasannya adalah karena tipe ini memiliki resource di heap, jika tidak dipindah, bisa terjadi dua pemilik dan berujung double free.

Copy adalah penyalinan bit-wise yang murah, sehingga kedua variabel tetap valid. Hanya berlaku untuk tipe yang mengimplementasikan trait Copy seperti i32/integer yang seluruh elemennya Copy. Tipe Copy tidak punya destructor (Drop) dan tidak mengelola resource heap. Setelah disalin, kedua variabel tetap valid dan memiliki salinan nilai masing-masing.

OWNERSHIP: MOVE AND COPY SEMANTICS

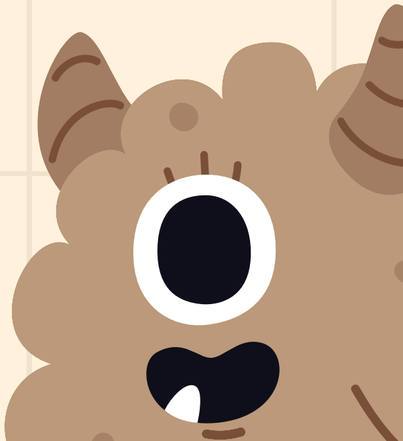


▶ Run | ⚙ Debug

```rust
fn main() {
    // MOVE (non Copy)
    let s1: String = String::from("hi");
    let s2: String = s1;          // s1 berpindah maka tak bisa dipakai lagi.
    // println!("{}", s1); // error: value moved.
    println!("{}", s2);  // real jalan, tapi bakal cuman ada 1 "hi".

    // COPY (Copy types)
    let x: i32 = 42;
    let y: i32 = x;             // x masih boleh dipakai.
    println!("{x}, {y}"); // real jalan dengan output: 42, 42.
}
```

```
hi
42, 42
 *  Terminal will be reused by tasks, press any key to close it.
```

BORROWING

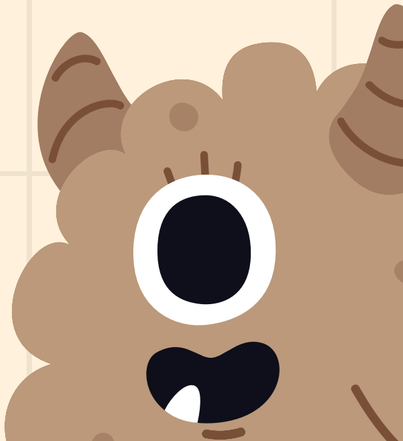


BORROWING AND REFERENCES

Borrowing di Rust adalah meminjam data lewat reference (&T) tanpa mengambil ownership. Fungsi penerima bisa membaca data yang sama tanpa menyalin (yang membuatnya menjadi hemat) dan tanpa memindahkannya (owner tetap punya). Secara default referensi itu immutable, jadi tidak bisa mengubah data yang dipinjam. Aturan ini dicek di compile time, mencegah bug memori (dangling pointer, double free) dan data race.



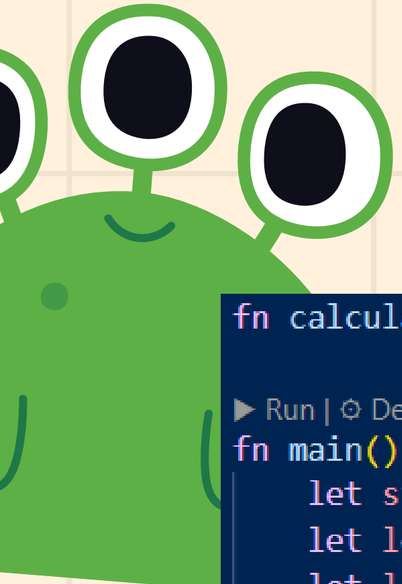
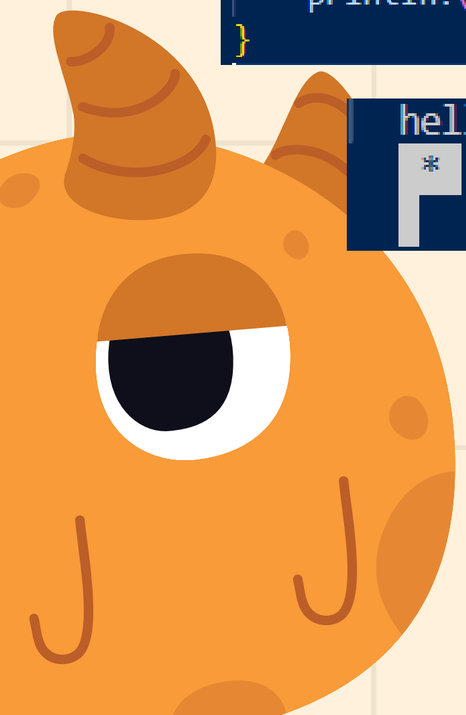


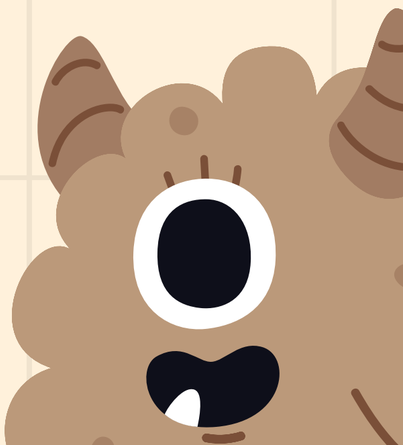



BORROWING AND REFERENCES

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

▶ Run | ⚙ Debug
fn main() {
    let s1: String = String::from("hello");
    let len: usize = calculate_length(&s1);     // pinjam s1 sebagai &str (diubah otomatis oleh Rust).
    let len2: usize = calculate_length("hi!");  // kirim string literal yang sudah bertipe &str tanpa peminjaman.
    println!("{s1} itu panjang hurufnya ada {len}, dan \"hi!\" panjang hurufnya ada {len2}");
}
```

```
hello itu panjang hurufnya ada 5, dan "hi!" panjang hurufnya ada 3
 *  Terminal will be reused by tasks, press any key to close it.
```

MUTABLE REFERENCES AND BORROWING RULES






MUTABLE REFERENCES AND BORROWING RULES

Dalam Rust, mutable reference (&mut T) memungkinkan kita meminjam data sekaligus mengubah isinya tanpa mengambil ownership. Demi keselamatan memori, Rust memberlakukan disiplin ketat, yaitu pada satu momen hanya boleh ada satu mutable reference yang aktif, dan mutable serta immutable reference tidak boleh berjalan bersamaan. Batasan ini menutup peluang data race dan mencegah dua bagian program memodifikasi nilai secara serempak, sehingga Rust tetap memberi keleluasaan menulis kode tanpa mengorbankan stabilitas maupun keamanan memori.

MUTABLE REFERENCES AND BORROWING RULES

```rust
fn change(some_string: &mut String) {
    some_string.push_str(string: ", I'll always love you");
}

▶ Run | ⚙ Debug
fn main() {
    let mut s: String = String::from("My darling");
    change(some_string: &mut s);                // pinjam sebagai &mut.
    println!("{s}");                // outputnya bakal "My darling, I'll always love you".
}
```

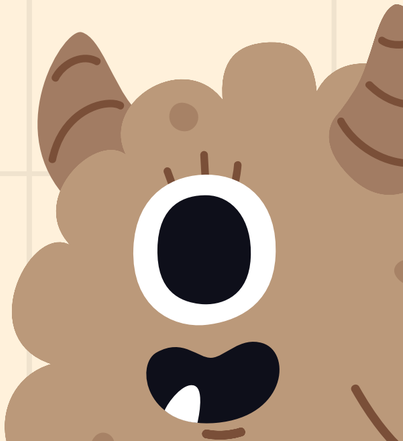
```
My darling, I'll always love you
 *  Terminal will be reused by tasks, press any key to close it.
```

MUTABLE REFERENCES AND BORROWING RULES

```rust
▶ Run | ⚙ Debug
fn main() {
    let mut s: String = String::from("I'll always love you");
    let r1: &mut String = &mut s;
    let r2: &mut String = &mut s; // ERROR: gabisa borrow `s` sebagai mutable lebih dari sekali dalam satu waktu.
    println!("{r1}, {r2}");
}
```

```
   Compiling fp_rust v0.1.0 (E:\Semester 5\Functional Programming\Rust)
error[E0499]: cannot borrow `s` as mutable more than once at a time
  --> src\main.rs:43:14
   |
42 |     let r1 = &mut s;
   |              ------ first mutable borrow occurs here
43 |     let r2 = &mut s; // ERROR: gabisa borrow `s` sebagai mutable lebih dari sekali dalam satu waktu.
   |              ^^^^^^ second mutable borrow occurs here
44 |     println!("{r1}, {r2}");
   |                -- first borrow later used here

For more information about this error, try `rustc --explain E0499`.
error: could not compile `fp_rust` (bin "fp_rust") due to 1 previous error
```

CLOSURE



CLOSURE IN RUST

Closures di Rust adalah fungsi anonim (fungsi tanpa nama) yang dapat disimpan ke dalam variabel atau dikirim sebagai argumen ke fungsi lain. Kita dapat membuat closure di satu tempat, lalu memanggilnya di tempat lain dalam konteks yang berbeda. Berbeda dari fungsi biasa, closure dapat menangkap nilai dari ruang lingkup (scope) tempat closure itu didefinisikan. Fitur ini memungkinkan kita untuk mengulang kode dan menyesuaikan perilaku fungsi dengan fleksibilitas tinggi.

CLOSURE IN RUST

```rust
fn main() {
    let add = |a: i32, b: i32| a + b;
    println!("Hasil: {}", add(3, 4));
}

```

```
PS D:\Kuliah\Season5\PF\iNTRODUCTION TO RUST\closure> cargo run
   Compiling closure v0.1.0 (D:\Kuliah\Season5\PF\iNTRODUCTION TO RUST\closure)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.57s
     Running `target\debug\closure.exe`
Hasil: 7
PS D:\Kuliah\Season5\PF\iNTRODUCTION TO RUST\closure>
```

CLOSURE IN RUST

```rust
let example_closure = |x| x;

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

```

```
PS D:\Kuliah\Season5\PF\iNTRODUCTION TO RUST\closure> cargo run
   Compiling closure v0.1.0 (D:\Kuliah\Season5\PF\iNTRODUCTION TO RUST\closure)
error[E0308]: mismatched types
 --> src\main.rs:5:29
  |
5 |     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:4:29
  |
4 |     let s = example_closure(String::from("hello"));
  |             --------------- ^^^^^^^^^^^^^^^^^^^^^ expected because this argument is of type `String`
  |             |
  |             in this closure call
note: closure parameter defined here
 --> src\main.rs:2:28
  |
2 |     let example_closure = |x| x;
  |                           ^
help: try using a conversion method
  |
5 |     let n = example_closure(5.to_string());
  |                             ++++++++++++

For more information about this error, try `rustc --explain E0308`.
error: could not compile `closure` (bin "closure") due to 1 previous error
```

CLOSURE TYPE INFERENCE AND ANNOTATION

Closure berbeda dari fungsi biasa karena tidak selalu perlu menulis tipe parameter dan tipe kembalian secara eksplisit. Rust bisa menebak tipe tersebut secara otomatis (type inference).
Keterbatasan interferensi tipe akan terjadi jika closure pertama kali dipanggil dengan tipe tertentu, tipe itu akan terkunci. Closure tidak bisa digunakan lagi dengan tipe lain.

```
1   fn add_fn(x: i32) -> i32 { x + 1 } // harus tulis tipe
2   let add_closure = |x| x + 1;       // tipe bisa ditebak
3
```

CAPTURING REFERENCES OR MOVING OWNERSHIP

Capture Method	Reference Type	Trait yang Dipakai	Kapan Digunakan
Borrow immutably	&T	Fn	Kalau closure hanya *membaca* variabel luar
Borrow mutably	&mut T	FnMut	Kalau closure *mengubah* variabel luar
Take Ownership(move)	T (move)	FnOnce	Kalau closure *mengambil alih* variabel luar (ownership berpindah)

BORROW IMMUTABLY

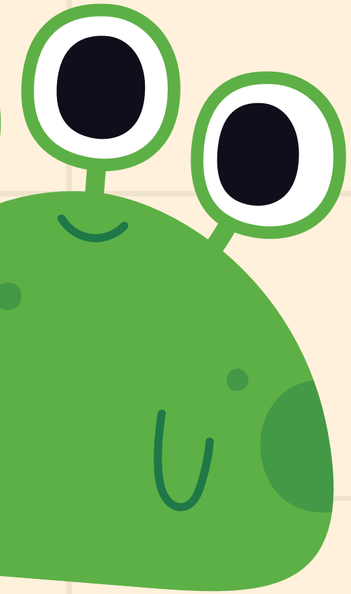
```rust
fn main() {
    let list = vec![1, 2, 3];
    println!("Before defining closure: {list:?}");

    let only_borrows = || println!("From closure: {list:?}");

    println!("Before calling closure: {list:?}");
    only_borrows();
    println!("After calling closure: {list:?}");
}
```

```
PS D:\Kuliah\Season5\PF\iNTRODUCTION TO RUST\closure> cargo run
   Compiling closure v0.1.0 (D:\Kuliah\Season5\PF\iNTRODUCTION TO RUST\closure)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.67s
     Running `target\debug\closure.exe`
Before defining closure: [1, 2, 3]
Before calling closure: [1, 2, 3]
From closure: [1, 2, 3]
After calling closure: [1, 2, 3]
PS D:\Kuliah\Season5\PF\iNTRODUCTION TO RUST\closure>
```

BORROW MUTABLY

```rust
fn main() {
    let mut list = vec![1, 2, 3];
    println!("Before defining closure: {list:?}");

    let mut borrows_mutably = || list.push(7);

    borrows_mutably();
    println!("After calling closure: {list:?}");
}
```

```
     Running `target\debug\closure.exe`
Before defining closure: [1, 2, 3]
After calling closure: [1, 2, 3, 7]
PS D:\Kuliah\Season5\PF\iNTRODUCTION TO RUST\closure>
```



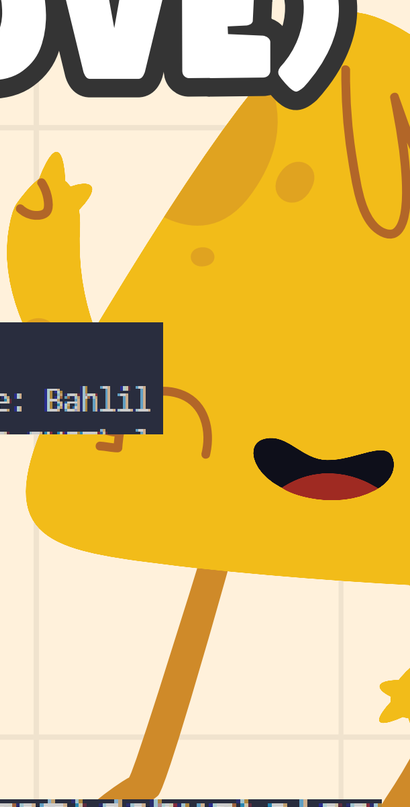



TAKE OWNERSHIP(MOVE)

```rust
fn main() {
    let name = String::from("Bahlil");

    // closure hanya MEMINJAM name (tidak memindahkan)
    let say_hello = || println!("Hello, {name}!");

    say_hello(); // ✅ masih bisa dipakai
    println!("Name masih bisa dipakai di luar closure: {name}");
}
```

=

```
Hello, Bahlil!
Name masih bisa dipakai di luar closure: Bahlil
```

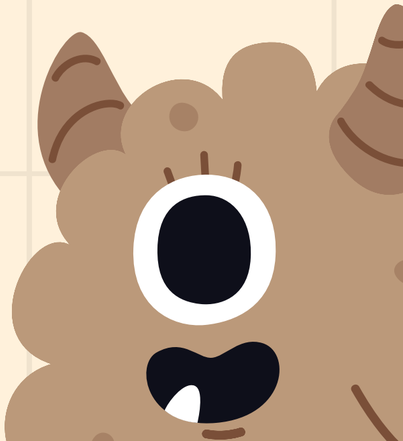


```rust
fn main() {
    let name = String::from("Azka");

    // closure MEMINDAHKAN ownership name
    let say_hello = move || println!("Hello, {name}!");

    say_hello(); // ✅ OK
    // println!("Name: {name}"); // ❌ ERROR: name sudah dipindahkan ke closure
}
```

=

```
Running `target\debug\clos
Hello, Bahlil!
```

ITERATOR



ITERATOR IN RUST

Pola iterator memungkinkan pelaksanaan tugas berulang pada serangkaian item, dengan iterator menangani seluruh logika penelusuran (iterasi) dan penentuan akhir urutan. Hal ini mengurangi kebutuhan untuk mengimplementasikan logika looping sendiri.

Di Rust, iterator bersifat "malas" yang berarti mereka tidak akan melakukan apa pun sampai metode yang menggunakannya terpanggil.

ITERATOR IN RUST

```
    ▶ Run | ⚙ Debug
1   fn main() {
2       let v1: Vec<i32> = vec![1, 2, 3];
3
4       let v1_iter: Iter<'_, i32> = v1.iter();
5
6       for val: &i32 in v1_iter {
7           println!("Got: {val}");
8       }
9   }
```

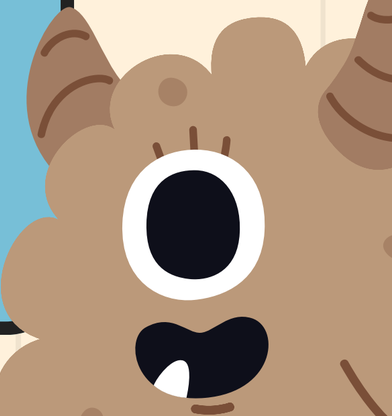
```
Got: 1
Got: 2
Got: 3
 *  Terminal will be reused by tasks, press any key to close it.
```

SIFAT ITERATOR DAN METHOD NEXT

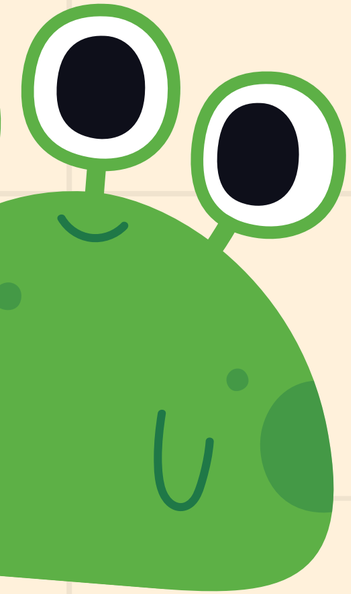
Semua iterator di Rust bekerja dengan mengimplementasikan sebuah trait bawaan bernama iterator. Definisi sederhana dari trait ini adalah:

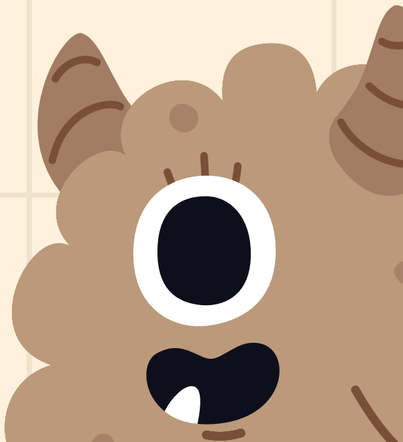
```
1    pub trait Iterator {
2        type Item;
3
4        fn next(&mut self) -> Option<Self::Item>;
5    }
```




SIFAT ITERATOR DAN METHOD NEXT





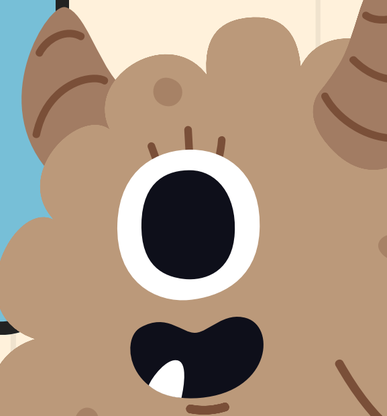
```
▶ Run Tests | ⚙ Debug
#[test]
▶ Run Test | ⚙ Debug
fn iterator_demonstration() {
    let v1: Vec<i32> = vec![1, 2, 3];

    let mut v1_iter: Iter<'_, i32> = v1.iter();

    assert_eq!(v1_iter.next(), Some(&1));
    assert_eq!(v1_iter.next(), Some(&2));
    assert_eq!(v1_iter.next(), Some(&3));
    assert_eq!(v1_iter.next(), None);
}
```

```
running 1 test
test iterator_demonstration ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

METHOD YANG MENGGUNAKAN ITERATOR

Trait iterator di Rust memiliki banyak method bawaan yang sudah disediakan, di mana sebagian besar method ini bekerja dengan cara memanggil next() secara internal. Method-method yang memanggil next() ini disebut "consuming adapters" (adaptor pengonsumsi) karena mereka akan menggunakan (mengonsumsi) iterator hingga habis saat dipanggil.



METHOD YANG MENGGUNAKAN ITERATOR

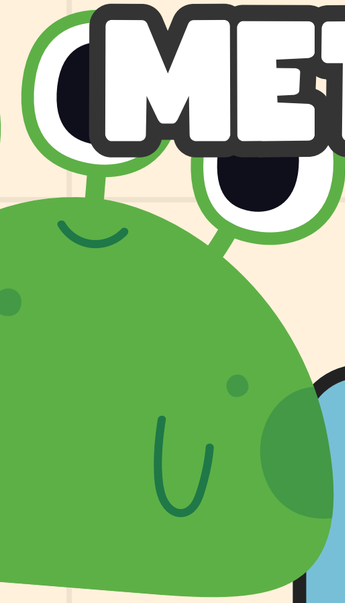
```
    ▶ Run Tests | ⚙ Debug
1   #[test]
    ▶ Run Test | ⚙ Debug
2   fn iterator_sum() {
3       let v1: Vec<i32> = vec![1, 2, 3];
4
5       let v1_iter: Iter<'_, i32> = v1.iter();
6
7       let total: i32 = v1_iter.sum();
8
9       assert_eq!(total, 6);
10  }
```

```
running 1 test
test iterator_sum ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

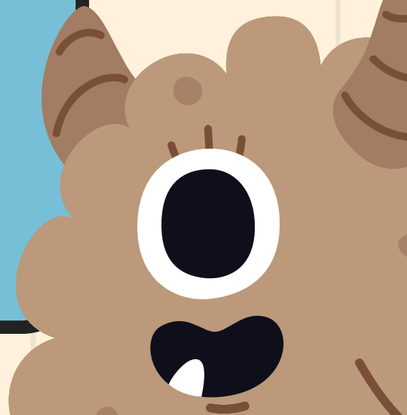
METHOD YANG MENGHASILKAN ITERATOR LAIN

Iterator adapter adalah metode, seperti .map(), yang tidak mengonsumsi iterator, melainkan menghasilkan iterator baru dengan perilaku yang telah diubah.

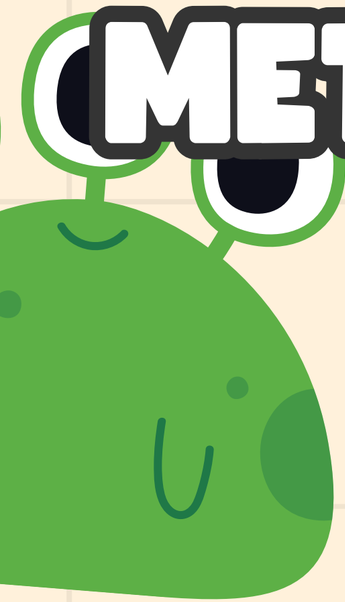
Adaptor ini bersifat "malas", mereka hanyalah sebuah "rencana" komputasi dan tidak akan melakukan apa pun sampai sebuah metode pengonsumsi (consumer) dipanggil.

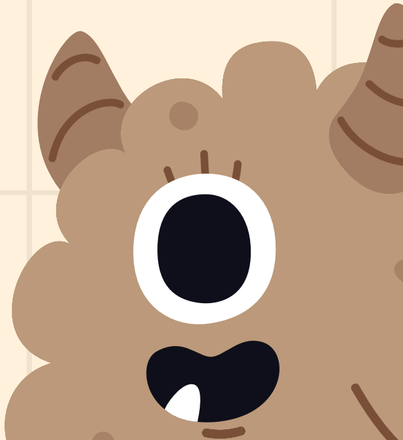




METHOD YANG MENGHASILKAN ITERATOR LAIN





```
▶ Run Tests | ⚙ Debug
#[test]
▶ Run Test | ⚙ Debug
fn map_and_collect_example() {
    let v1: Vec<i32> = vec![1, 2, 3];

    let v2: Vec<_> = v1.iter().map(|x: &i32| *x + 1).collect();

    assert_eq!(v2, vec![2, 3, 4]);
}
```

```
running 1 test
test map_and_collect_example ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

THANK YOU