DASAR PEMROGRAMAN RUST

Noval Agung

Pengenalan Rust Programming

Rust adalah sebuah bahasa pemrograman general purpose yang fokus pada performance, type safety, dan concurrency. Bahasa ini diciptakan sebagai alternatif bahasa pemrograman yang berfokus pada hal-hal yang cukup low-level tapi tetap men-support fitur yang sifatnya high-level.

Rust dikembangkan oleh Graydon Hoare sewaktu ia masih bekerja di Mozilla pada tahun 2009, kemudian di-maintain dibawah naungan Rust Foundation hingga sekarang.

Rust memiliki beberapa kelebihan dibanding bahasa system-programming lainnya, yang di antaranya adalah:

- Keamanan memory yang sangat bagus, dengan pengecekan ada di waktu kompilasi
- Ownership memory management
- Type system (traits, generic, struct, dan lainnya)
- Macros untuk metaprogramming
- Tooling dan `std` library yang lengkap
- Built-in package management tool (cargo & crates.io)
- Efisien dan memiliki performa tinggi
- Dukungan komunitas yang bagus. Banyak library open source yang tersedia

Pada buku ini (terutama semua serial chapter A) kita akan belajar tentang dasar pemrograman Rust, mulai dari 0.

Author & Contributors

Ebook Dasar Pemrograman Rust adalah project open source. Siapapun bebas untuk berkontribusi di sini, bisa dalam bentuk perbaikan typo, update kalimat, maupun submit tulisan baru.

Bagi teman-teman yang berminat untuk berkontribusi, silakan fork github.com/novalagung/dasarpemrogramanrust, kemudian langsung saja cek/buat issue kemudian submit relevan pull request untuk issue tersebut 😊.

Original Author

E-book ini di-inisialisasi oleh Noval Agung Prayogo.

Contributors

Berikut merupakan hall of fame kontributor yang sudah berbaik hati menyisihkan waktunya untuk membantu pengembangan e-book ini.

1. ... anda :-)

Lisensi dan Distribusi Konten

Ebook Dasar Pemrograman Rust gratis untuk disebarluaskan secara bebas, dengan catatan sesuai dengan aturan lisensi CC BY-SA 4.0 yang kurang lebih sebagai berikut:

- Diperbolehkan menyebar, mencetak, dan menduplikasi material dalam konten ini ke siapapun.
- Diperbolehkan memodifikasi, mengubah, atau membuat konten baru menggunakan material yang ada dalam ebook ini untuk keperluan komersil maupun tidak.

Dengan catatan:

- Harus ada credit sumber aslinya, yaitu Dasar Pemrograman Golang atau novalagung
- Tidak mengubah lisensi aslinya, yaitu CC BY-SA 4.0
- Tidak ditambahi restrictions baru
- Lebih jelasnya silakan cek https://creativecommons.org/licenses/by-sa/4.0/.

dasarpemrogramanrust 8d59f6 FOSSA

No Issues Found

LICENSE SCAN

DEEP IMPACT STATS

+ 822Deep Dependencies

+ 8 Obligations from 28 Licenses

View More Details on FOSSA

Instalasi Rust

Pada chapter ini kita akan belajar cara instalasi Rust. Pembaca bisa mengikuti panduan instalasi ini, atau langsung saja navigasi ke https://www.rust-lang.org/tools/install.

Ada dua metode instalasi yang penulis anjurkan, yaitu menggunakan `rustup` atau menggunakan installer.

Install Rust menggunakan `rustup`

`rustup` adalah tool resmi untuk manajemen *Rust tooling*. `rustup` mempermudah proses instalasi, update versi, ataupun penggantian versi Rust.

> *Instalasi Rust tooling menggunakan metode ini lebih direkomendasikan*

Cara instalasi `rustup` untuk tiap sistem operasi bisa dilihat di bawah ini:

◉ MacOS, Linux, Unix

Jalankan command berikut untuk instal `rustup` dan mengecek hasil instalasi.

```
curl https://sh.rustup.rs -sSf | sh

rustup --version
rustc --version
cargo --version
```

◉ Windows

Silakan download file `rustup-init.exe` di https://www.rust-lang.org/tools/install. Jalankan installar, lalu run command berikut untuk mengecek hasil instalasi.

```
rustup --version
rustc --version
cargo --version
```

Untuk pengguna WSL (Windows Subsystem for Linux), bisa menggunakan command berikut.

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

rustup --version
rustc --version
cargo --version
```

Install Rust menggunakan installer

Silakan download installer sesuai dengan sistem operasi dan platform yang digunakan menggunakan link berikut:

> *https://forge.rust-lang.org/infra/other-installation-methods.html#standalone-installers*

Lalu run command di bawah ini untuk mengecek hasil instalasi.

```
rustup --version
rustc --version
cargo --version
```

Update versi Rust tooling

Jika pembaca menggunakan `rustup`, update Rust tooling dilakukan cukup dengan menjalankan command berikut:

```
rustup update
rustup --version
```

Terutuntuk Rust yang di-install menggunakan installer, cara update dilakukan dengan download dan run installer versi terbaru yang bisa didownload dari link berikut:

> *https://forge.rust-lang.org/infra/other-installation-methods.html#standalone-installers*

Catatan chapter 📑

◉ Referensi

- https://www.rust-lang.org/tools/install
- https://forge.rust-lang.org/infra/other-installation-methods.html
- https://doc.rust-lang.org/book/ch01-01-installation.html

Rust Editor & Plugin

Pemilihan editor dan plugin

Normalnya semua editor atau IDE bisa digunakan dalam penulisan kode program Rust. Namun development akan menjadi lebih mudah dan menyenangkan dengan bantuan Rust plugin.

Ada beberapa editor yang di-support secara official oleh Rust via plugin.

Editor	Plugin	Link
VS Code	`rust-analyzer`	plugin link
Sublime Text	`rust-enhanced`	plugin link
Atom	`atom-ide-rust`	plugin link
Intellij IDEA	`Rust plugin`	plugin link
Eclipse	`Eclipse IDE for Rust Developers`	plugin link
Vim	`rust.vim`	plugin link
Emacs	`rust-mode`	plugin link
Geany	-	plugin link

Silakan pilih editor sesuai preferensi, lalu install pluginnya. Untuk referensi lebih lanjut silakan navigasi ke https://www.rust-lang.org/tools.

Pada pembuatan ebook ini penulis menggunakan editor VS Code dengan plugin rust-analyzer ter-install.

Catatan chapter 📑

◉ Referensi

- https://www.rust-lang.org/tools

A.1. Program Pertama → Hello Rust

Seperti pada umumnya bahasa pemrograman, belajar membuat program pasti diawali dengan aplikasi `Hello World`, dan pada chapter ini kita akan melakukannya. Kita akan buat program bernama `Hello Rust` menggunakan pemrograman Rust.

A.1.1. Pembuatan project/package

Pembuatan project baru di Rust bisa dilakukan dengan 2 cara, dengan `cargo` atau tanpanya. Disini kita akan skip bagian pembuatan project tanpa `cargo`, karena akan butuh effort lebih banyak nantinya dalam mengelola package dan manajemen build.

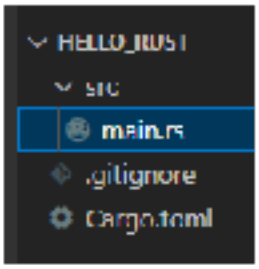
> *Di Rust istilah project lebih dikenal dengan **package**, dan pada ebook ini maknanya adalah sama*

Ok langsung saja, buat project baru dengan mengeksekusi command berikut:

```
cargo new hello_rust
cd hello_rust
```

Command di atas menghasilkan sebuah folder baru bernama `hello_rust` dengan isi beberapa file:

- `Cargo.toml`
- `.gitignore`
- `src/main.rs`



File `src/main.rs` adalah file penting dalam pemrograman Rust. File `main.rs` merupakan file pertama yang dipanggil saat proses build program Rust (yg kemudian di-run). Source code program harus berada dalam folder `src`.

> *Pembahasan detail mengenai file `cargo.toml` nantinya ada pada chapter Module System → Package & Crate. Untuk sekarang penulis anjurkan untuk mengikuti pembelajaran tiap chapter secara urut.*

A.1.2. Run project `Hello Rust`

Sebelum membahas isi kode program dalam `main.rs`, ada baiknya kita run terlebih dahulu program ini untuk melihat hasilnya. Jalankan command berikut untuk run program.

```
cargo run
```

Bisa dilihat pada gambar di atas pesan `Hello, world!` yang ada dalam file `main.rs` muncul, menandakan proses eksekusi program sukses.

Command `cargo run` digunakan untuk menjalankan aplikasi saat proses development. Perlu diingat bahwa Rust merupakan bahasa pemrograman yang kategorinya `compiled language` yang artinya kode program akan dikompilasi terlebih dahulu untuk menghasilkan file binary, dan kemudian file binary itulah yang dijalankan.

Command `cargo run` merupakan shorthand untuk memperpendek proses kompilasi dan eksekusi. Dalam command tersebut, kode program akan di-compile terlebih dahulu, dan hasilnya adalah file executable binary bernama `hello_rust.exe` (karena penulis menggunakan windows). File binary tersebut berada dalam project dalam path `hello_rust/target/debug/hello_rust.exe`. Setelah proses kompilasi, file binary tersebut dijalankan, dan hasilnya adalah pesan `Hello, world!` yang muncul di layar

> *Untuk pengguna windows, file binary akan ber-ekstensi `.exe` seperti*

> *pada contoh yaitu `hello_rust.exe`. Sedangkan untuk non-windows, file tidak mempunyai ekstensi, `hello_rust`.*

Command `cargo run` akan sering kita pakai dalam proses development.

A.1.3. Penjelasan blok kode `main.rs`

Berikut adalah isi (default) dari file `main.rs`, dan kita akan bahas setiap barisnya (cuma 3 baris).

src/main.rs

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

◉ Notasi pendefinisian fungsi

Pembuatan fungsi di rusat menggunakan keyword `fn` dengan notasi penulisan sebagai berikut, contoh:

```
fn namaFungsi() {

}
```

`namaFungsi` di atas adalah nama fungsi. Pada program yang sudah kita buat, fungsi bernama `main`. Penulisan nama fungsi diikuti dengan `()` kemudian

kurung kurawal `{ }`.

- Sintaks `()` nantinya bisa diisi dengan definisi parameter. Pada contoh ini tidak ada parameter yang ditulis.
- Sintaks `{}` dituliskan dalam baris berbeda, isinya adalah kode program.

◉ Notasi pemanggilan fungsi

Notasi penulisan pemanggilan fungsi adalah cukup dengan menuliskan nama fungsinya kemudian diikuti dengan `()`, seperti berikut:

```
namaFungsi();
```

Jika ada argument parameter yang perlu disisipkan saat pemanggilan fungsi, dituliskan diantara kurung `()`. Contoh pemanggilan `sebuahFungsi` dengan argument berupa string `"sebuah argument"`.

```
sebuahFungsi("sebuah argument");
```

Ok, sampai sini penulis rasa cukup jelas. Sekarang kita fokus ke `println`, ada yang aneh dengan pemanggilan fungsi ini. Kenapa `println` tidak dituliskan dalam bentuk `println("Hello, world!")` melainkan `println!("Hello, world!")`?

Sekadar informasi saja, berdasarkan versi Rust terbaru, tidak ada fungsi bernama `println`, yang ada adalah macro bernama `println`.

◉ Macro `println`

Agar pembaca tidak bertambah bingung, setidaknya untuk sekarang pada

chapter awal ini anggap saja macro adalah fungsi ... tapi sedikit berbeda. Yang paling terlihat bedanya secara sintaktis adalah tanda `!`. Pemanggilan macro pasti diikuti tanda `!` sebelum `()`. Contoh:

```
println!("Hello, world!");
```

Macro `println` digunakan digunakan untuk menampilkan string atau pesan ke console output (`stdout`) dan diikuti oleh baris baru (newline/enter). Agar lebih jelas jalankan kode berikut:

src/main.rs

```
fn main() {
    println!("Hello, world!");
    println!("How");
    println!("are");
    println!("you?");
}
```

```
src > main.rs > main
▶ Run | Debug
fn main() {
    println!("Hello, world!");
    println!("How");
    println!("are");
    println!("you?");
}

PROBLEMS 5   OUTPUT   GITLENS   JUPYTER   DEBUG CONSOLE   TERMINAL

(base) PS D:\learn rust\hello_rust> cargo run
   Compiling hello_rust v0.1.0 (D:\learn-rust\hello_rust)
    Finished dev [unoptimized + debuginfo] target(s) in 0.47s
     Running `target\debug\hello_rust.exe`

Hello, world!
How
are
you?
```

Bisa dilihat, setiap pesan yang ditampilkan menggunakan macro `println` muncul di baris baru.

◉ Notasi penulisan statement

Di Rust, statement harus diikuti dengan tanda `;` atau titik koma. Wajib hukumnya. Tanpa tanda `;` maka beberapa statement akan dianggap 1 baris dan kemungkinan menghasilkan error jika sintaks dianggap tidak valid. Tanda `;` adalah penanda akhir statement. Contoh `println!("Hello, world!")`.

◉ Indentation

Mengacu ke keterangan pada dokumentasi Rust, standar indentasi untuk kode program Rust adalah `4 space characters` atau 4 karakter space.

Ok, penulis rasa cukup untuk program pertama ini, semoga tidak membingungkan. Silakan diulang-ulang jika perlu. Jika sudah siap, kita akan lanjut ke pembahasan dasar pemrograman Rust pada chapter berikutnya.

Selamat! Anda telah menjadi programmer Rust!

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../hello_rust
```

◉ Referensi

- https://doc.rust-lang.org/book/ch01-03-hello-cargo.html
- https://doc.rust-lang.org/std/macro.println.html
- https://doc.rust-lang.org/rust-by-example/hello/print.html

A.2. Build dan Run Program Rust

Pada chapter ini kita akan belajar lebih lanjut tentang command untuk proses build dan run dalam program Rust. Sebelumnya kita sudah belajar tentang `cargo new` dan `cargo run`, selain command tersebut ada juga command lain yaitu `cargo build` untuk proses kompilasi build kode program.

A.2.1. Command `cargo build`

Di atas sedikit disinggung bahwa command `cargo build` berguna untuk mem-build kode program, dan command ini menghasilkan file binary. `cargo build` lebih sering digunakan saat proses build untuk distribution/deployment, yang umumnya binary hasil build kemudian distribusikan pengguna program.

Command tersebut juga bisa digunakan di lokal environment, silakan dicoba.

- Untuk pengguna Windows:

```
cargo build
cd target/debug
hello_rust.exe
```

- Untuk pengguna non-Windows:

```
cargo build
```

Hasilnya:

```
(base) PS D:\learn-rust\hello_rust> cargo build
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
(base) PS D:\learn-rust\hello_rust> cd target/debug
(base) PS D:\learn-rust\hello_rust\target\debug> hello_rust.exe

Hello, world!
```

A.2.2. Optimized build

Catatan tambahan saja, command `cargo build` dan juga `cargo run` menghasilkan file binary yang *unoptimized* dan berisi beberapa informasi tambahan untuk proses debugging. Untuk distribution/production dianjurkan untuk generate *optimized* binary. Caranya dengan cukup menambahkan flag `--release` pada saat eksekusi command `cargo run`.

```
cargo build --release
```

```
(base) PS D:\learn-rust\hello_rust> cargo build --release
    Finished release [optimized] target(s) in 0.01s
(base) PS D:\learn-rust\hello_rust> cd target/release
(base) PS D:\learn-rust\hello_rust\target\release> hello_rust.exe

Hello, world!
```

Untuk release, file binary berada dalam path `target/release`. File binary nya adalah optimized, size nya lebih kecil. Mungkin untuk program `hello world` tidak akan signifikan bedanya, tapi untuk project real sangat dianjurkan untuk menggunakan optimized build.

A.2.3. Command `rustc`

Bagian ini merupakan tambahan informasi saja, bahwa selain command `cargo` ada juga command `rustc` yang bisa digunakan untuk kompilasi program.

Sebagai contoh, silakan buat file `hello.rs`, lalu tulis kode berikut:

hello.rs

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

Kemudian *compile* lalu jalankan file *executable*-nya, hasilnya adalah sama seperti eksekusi program menggunakan `cargo run`.

- MacOS, Linux, Unix, WSL

  ```
  rustc hello.rs
  ./hello
  ```

- MacOS, Linux, Unix, WSL

  ```
  rustc hello.rs
  .\hello.exe
  ```

Pada ebook ini kita akan menggunakan `cargo` untuk manajemen dan juga eksekusi kode program Rust. Jadi tidak menggunakan `rustc` ya.

> *Command `cargo run` dan `cargo build` dalam proses kompilasinya meng-invoke command `rustc`*

Catatan chapter 📑

◉ Referensi

- https://doc.rust-lang.org/book/ch01-03-hello-cargo.html

A.3. Komentar

Pada chapter ini kita akan membahas tentang komentar pada pemrograman Rust. Ada beberapa jenis komentar yang tersedia, namun untuk sekarang yang penting diketahui hanya 2.

A.3.1. Baris komentar

Rust menggunakan double slash (`//`) sebagai penanda *line comment* atau baris komentar. Contoh:

```
fn main() {
    // ini adalah komentar
    // komentar tidak akan di-eksekusi
    println!("hello");
}
```

A.3.2. Blok komentar

Blok komentar dituliskan dengan cara diawali `/*` dan diakhiri `*/`. Silakan tambahkan kode berikut dalam blok fungsi `main`.

```
/*
    komentar ini
    tidak akan
    dieksekusi
*/
```

Lalu run program tersebut. O iya, untuk kawan-kawan yang menggunakan editor VSCode dengan ekstensi `rust-analyzer` ter-install. Ada shortcut untuk run program yaitu dengan klik tombol `▶ Run` di atas definisi fungsi `main`.

```
src > main.rs > ...
1
    ▶ Run | Debug
2   fn main() {
3       // ini adalah komentar
4       // komentar tidak akan di eksekusi
5       println!("hello");
6
7       /*
8           komentar ini
9           tidak akan
10          dieksekusi
11      */
12      println!("world");
13  }
14

PROBLEMS 8   OUTPUT   DEBUG CONSOLE   JUPYTER   DEBUG CONSOLE   TERMINAL

   Compiling app2_komentar v0.1.0 (D:\learn-rust\app2_komentar)
    Finished dev [unoptimized + debuginfo] target(s) in 0.53s
     Running `target\debug\app2_komentar.exe`

hello
world
```

Bisa dilihat, komentar tidak menghasilkan efek apa-apa pada program yang dibuat.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../komentar
```

◉ Referensi

- https://doc.rust-lang.org/reference/comments.html

A.4. Variabel

Rust merupakan bahasa yang kalau dikategorikan berdasarkan *type system*-nya adalah *statically typed*, yang artinya semua tipe data harus diketahui saat kompilasi (compile time).

Pada chapter ini kita akan belajar mengenai variabel pada Rust, cara deklarasinya, dan juga mengenai *immutability* variabel.

A.4.1. Deklarasi variabel menggunakan `let`

Keyword `let` digunakan untuk deklarasi variabel. Notasinya kurang lebih seperti berikut. Di contoh ini sebuah variabel bernama `nama_variabel` didefinisikan dan diisi nilainya dengan sebuah string `"predefined value"`.

```
fn main() {
    let nama_variabel = "predefined value";
    println!(nama_variabel);
}
```

Ok, sekarang coba jalankan, dan lanjut ke pembahasan dibawah.

◉ Aturan penamaan variabel (naming convention)

Naming convention variabel di Rust adalah **snake case**. Nama variabel

dituliskan dalam huruf kecil dengan separator tanda garis bawah atau underscore (`_`). Contohnya seperti: `nama_variabel`, `hasil_operasi_pembagian`, `kota_kelahiran`, dll.

◉ Macro `println` untuk print string

```
error: format argument must be a string literal
   > src\main.rs:3:14
    |
  3 |     println!(nama_variabel);
    |              ^^^^^^^^^^^^^
    |
help: you might be missing a string literal to format with
    |
  3 |     println!("{}", nama_variabel);
    |              +++++

error: could not compile `app3_variabel` due to previous error
```

Error? kok bisa?

Perlu diketahui bahwa pada penggunaan macro `println`, parameter pertama wajib diisi dengan sebuah **string literal**. Contohnya seperti `"Hello, world!"`. String literal yang ditampung terlebih dahulu dalam variabel, kemudian variabel-nya di print, akan menghasilkan error, contohnya seperti di atas.

Solusinya bagaimana? ya tetap menggunakan string literal, tapi dengan menerapkan teknik **formatted print**.

Sekarang ubah kode menjadi seperti ini:

```
let nama_variabel = "predefined value";
println!("{}", nama_variabel);
```

Bisa dilihat argument pertama statement macro `println` adalah string literal

`"{}"`. String `{}` pada macro tersebut akan di-replace dengan isi argument setelahnya, yaitu `nama_variabel`.

Jika dijalankan, maka hasilnya tidak ada error dan menampilkan output sesuai harapan.

```
(base) PS D:\learn rust> cd .\app3_variabel\
(base) PS D:\learn-rust\app3_variabel> cargo run
   Compiling app3_variabel v0.1.0 (D:\learn-rust\app3_variabel)
    Finished dev [unoptimized + debuginfo] target(s) in 0.57s
     Running `target\debug\app3_variabel.exe`

predefined value
```

Ok sekarang kembali ke topik, yaitu perihal variabel.

A.4.2. *Immutability* pada variabel

Tulis kode berikut. Ada 2 variabel yang dideklarasikan, `message_number` yang isinya numerik, dan `message1` yang isinya string.

```
let message_number = 1;
let message1 = "hello";
println!("message number {}: {}", message_number, message1);
```

Jalankan, hasilnya tidak ada error, aman.

```
(base) PS D:\learn rust\app3_variabel> cargo run
   Compiling app3_variabel v0.1.0 (D:\learn-rust\app3_variabel)
    Finished dev [unoptimized + debuginfo] target(s) in 0.42s
     Running `target\debug\app3_variabel.exe`

message number 1: hello
```

> *O iya, jika ingin menampilkan banyak variabel via macro `println`, manfaatkan string `{}`.*
>
> *Tulis `{}` sejumlah variabel yang ingin ditampilkan, pada output string `{}` akan di-replace oleh variabel yang disisipkan pada argument saat pemanggilan macro `println` secara berurutan.*
>
> *Pada contoh di atas, string `"message number {}: {}"`*
> *... akan menjadi `"message number {message_number}: {message1}"`*
> *... kemudian menghasilkan `"message number 1: hello"`*

Sekarang ubah isi `message_number` menjadi `2` lalu siapkan juga variabel `message2`, kurang lebih kodenya menjadi seperti berikut, kemudian jalankan.

```
let message_number = 1;
let message1 = "hello";
println!("message number {}: {}", message_number, message1);

message_number = 2;
let message2 = "world";
println!("message number {}: {}", message_number, message2);
```

```
(base) PS D:\learn-rust\app3_variabel> cargo run
   Compiling app3_variabel v0.1.0 (D:\learn rust\app3_variabel)

error[E0384]: cannot assign twice to immutable variable `message_number`
  --> src\main.rs:12:5
   |
8  |     let message_number = 1;
   |
   |         |
   |         first assignment to `message_number`
   |         help: consider making this binding mutable: `mut message_number`
...
12 |     message_number = 2;
   |     ^^^^^^^^^^^^^^^^^^ cannot assign twice to immutable variable

For more information about this error, try `rustc --explain E0384`.
error: could not compile `app1_variabel` due to previous error
```

Error? kok bisa?

Perlu diketahui, bahwa *by default* semua variabel adalah *immutable*. Immutable itu artinya gak bisa di ubah nilai/value-nya. Jadi mirip seperti konstanta.

> *Variabel immutable mirip seperti konstanta, tapi sebenarnya berbeda. Akan kita bahas lebih detail di beberapa chapter berbeda perihal perbedaannya.*

Lalu bagaimana cara agar nilai variabel bisa diubah? Caranya dengan menambahkan keyword `mut` (yang merupakan kependekan dari *mutable*) saat pendefinisian variabel tersebut.

A.4.3. Keyword `mut`

Ok, mari kita coba, ubah statement deklarasi variabel `message_number`,

tambahkan keyword `mut`. Lalu jalankan ulang program.

```
let mut message_number = 1;
let message1 = "hello";
println!("message number {}: {}", message_number, message1);

message_number = 2;
let message2 = "world";
println!("message number {}: {}", message_number, message2);
```

```
(base) PS D:\learn-rust\app3_variabel> cargo run
   Compiling app3_variabel v0.1.0 (D:\learn-rust\app3_variabel)
    Finished dev [unoptimized + debuginfo] target(s) in 0.42s
     Running `target\debug\app3_variabel.exe`

message number 1: hello
message number 2: world
```

Ok, sekarang tidak muncul error.

Dengan menggunakan keyword `let mut` pada pendefinisian `message_number`, membuat variabel tersebut menjadi mutable atau bisa diubah nilainya.

◉ Argument parameter macro `println`

Selanjutnya, coba tambahkan lagi 1 message baru pada program yang sudah dibuat, lalu run.

```
let mut message_number = 1;
let message1 = "hello";
println!("message number {}: {}", message_number, message1);

message_number = 2;
let message2 = "world";
```

```
(base) PS D:\learn rust\app3_variabel> cargo run
   Compiling app1_variabel v0.1.0 (D:\learn-rust\app1_variabel)
    Finished dev [unoptimized + debuginfo] target(s) in 0.42s
     Running `target\debug\app3_variabel.exe`

message number 1: hello
message number 2: world
message number 3: 24
```

Jika dilihat ada yg berbeda pada cara deklarasi variabel `message3` dan juga pada statement `println` untuk `message3` yang disitu digunakan `{1}` dan `{0}`, tidak seperti sebelumnya yg menggunakan `{}`. Kita akan bahas yg ke-2 terlebih dahulu.

- Jika menggunakan `{}`, maka string akan di-replace sesuai urutan argument pada pemanggilan `println`.
- Jika menggunakan `{0}`, maka string akan di-replace dengan data pada argument ke `1` pemanggilan fungsi `println`, yang pada contoh di atas adalah `message3`.
- Jika menggunakan `{1}`, maka string akan di-replace dengan data pada argument ke `2` pemanggilan fungsi `println`, yang pada contoh di atas adalah `message_number`.
- Jika menggunakan `{n}`, maka string akan di-replace dengan data pada argument ke `n+1` pemanggilan fungsi `println`.

Dengan ini maka 3 statement berbeda berikut akan menghasilkan output yang sama:

```
println!("message number {}: {}", message_number, message3);
println!("message number {0}: {1}", message_number, message3);
println!("message number {1}: {0}", message3, message_number);
```

Sekarang perihal perbedaan cara deklarasi `message3` akan kita bahas dibawah

ini.

A.4.4. *Type Inference* vs *Manifest Typing*

Rust mendukung dua metode deklarasi variabel, yaitu *type inference* dan *manifest typing*.

◉ Metode *Type Inference*

Penulisan variabel dengan metode ini ditandai dengan tidak menuliskan tipe data secara jelas/eksplisit. Contoh:

```
let var1 = "hello" // compiler akan secara cerdas mendeteksi var1
tipe data nya string
let var2 = 12      // compiler akan secara cerdas mendeteksi var2
tipe data nya numerik
```

Metode deklarasi yang selama ini sudah kita terapkan adalah *type inference*.

◉ Metode *Manifest Typing*

Metode ini mewajibkan programmer untuk menuliskan secara jelas/eksplisit tipe data variabel. Contoh seperti ditandai dengan menu tidak menuliskan tipe data secara jelas/eksplisit. Contoh seperti pada praktek sebelumnya, yaitu pendefinisian `message3`.

```
let message3: i8 = 24;
```

Notasi penulisan tipe data adalah `namavariabel: tipedata`. Contoh seperti di atas, yaitu `let message3: i8 = 24` artinya variabel `message3` didefinisikan memiliki tipe data `i8`.

`i8` merupakan salah satu tipe data dari sekian banyak tipe data yang ada di Rust. Lebih jelasnya kita akan bahas pada chapter selanjutnya.

A.4.5. Deklarasi variabel tanpa *predefined value*

Sesuai dengan penjelasan di dokumentasi spesifikasi Rust, **tidak diperbolehkan mendeklarasikan variabel tanpa predefined value**. Jika dipaksa akan muncul error pada baris kode yang menggunakan variabel tersebut.

A.4.6. Deklarasi banyak variabel dalam satu statement

Di Rust memungkinkan untuk mendefinisikan banyak variabel dalam 1 baris statement. Notasi penulisannya seperti berikut:

```
let (var1, var2) = (24, "hello");
println!("var1: {0}", var1); // hasilnya => var1: 24
println!("var2: {0}", var2); // hasilnya => var2: hello
```

Pendefinisian banyak variabel dalam 1 statement dilakukan dengan menuliskan semua variabelnya dengan separator tanda `,` dan diapit tanda kurung `()`.

Sebagai contoh di atas 2 variabel didefinisikan, yaitu `var1` dan `var2`, dan keduanya memilik value yang berbeda tipe-datanya.

Bisa juga saat definisi variabel sekaligus ditentukan tipe data variabel, notasi penulisan bisa dilihat pada contoh kode berikut:

```
let (var3, var4): (i8, i8) = (32, 12);
println!("var3: {0}", var3); // hasilnya => var3: 32
println!("var4: {0}", var4); // hasilnya => var4: 12
```

Dengan notasi ini bisa juga untuk definisi variabel *mutable*. Cukup tambahkan keyword `mut` pada variabel yang ingin bisa diubah nilainya.

```
let (var5, mut var6, var7): (i8, i8, i8) = (64, 12, 4);
println!("var5: {0}", var5); // hasilnya => var5: 64
println!("var6: {0}", var6); // hasilnya => var6: 12
var6 = 24;
println!("var6: {0}", var6); // hasilnya => var6: 24
println!("var7: {0}", var7); // hasilnya => var7: 4
```

A.4.7. Deklarasi variabel dengan tipe data ditentukan dari value

Ini merupakan salah satu alternatif cara penulisan untuk men-specify tipe data variabel, caranya dengan menuliskan tipe data tepat setelah value, contohnya seperti berikut:

```
let data1 = 24i8;
println!("data1: {0}", data1); // hasilnya => data1: 24
```

Variabel `data` didefinisikan dengan predefined value adalah numerik `24` dengan tipe data `i8`, penulisan value nya menjadi `24i8`.

Boleh juga menggunakan separator `_` dalam penulisan value-nya sebagai pembatas antara nilai dan tipe data, contoh:

```
let data1 = 24_i8;
println!("data1: {0}", data1); // hasilnya => data1: 24
```

A.4.8. Variable *Shadowing*

Di Rust ada konsep bernama **variable shadowing**. Shadowing sendiri adalah pendefinisan ulang variabel yang sebelumnya sudah didefinisikan. Biasanya teknik ini dipakai untuk isolasi variabel dalam sebuah blok kode.

Contoh shadowing bisa dilihat pada kode berikut:

```
let x = 5;
println!("x: {}", x); // hasilnya => x: 5

let x = x + 1;
println!("x: {}", x); // hasilnya => x: 6
```

Lebih jelasnya akan kita bahas pada chapter terpisah, yaitu Shadowing.

Catatan chapter 📑

◉ Work in progress

- Variabel `_`

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../variabel
```

◉ Chapter relevan lainnya

- Shadowing
- Macro
- Formatted Print

◉ Referensi

- https://doc.rust-lang.org/book/ch03-01-variables-and-mutability.html
- https://doc.rust-lang.org/rust-by-example/hello/print.html
- https://doc.rust-lang.org/rust-by-example/variable_bindings/scope.html
- https://doc.rust-lang.org/reference/variables.html
- https://rust-lang.github.io/api-guidelines/naming.html
- https://stackoverflow.com/questions/38045700/how-do-i-declare-multiple-mutable-variables-at-the-same-time

---

A.5. Tipe Data → Primitive Scalar

Tipe data di Rust dikategorikan menjadi beberapa group, salah satunya adalah group tipe data primitif scalar atau biasa disebut **scalar types**. Pada chapter ini kita akan mempelajarinya.

Tipe data scalar sendiri merupakan tipe data primitif yang isinya hanya 1 nilai. Rust memilik 4 tipe scalar, yaitu integers, floating-point, boolean, dan character.

A.5.1. Signed integers

Signed integer merupakan tipe data numerik/integer yang bisa menampung nilai positif dan juga negatif. Ada beberapa tipe data singed integer tersedia di Rust yang dibedakan sesuai size-nya.

Tipe data ini keyword-nya ditandai dengan huruf awalan `i`, contohnya `i8`, yang dimana tipe ini adalah tipe data numerik integer dengan range value yang bisa ditampung adalah mulai dari angka **−128** (didapat dari $-(2^7)$) hingga **127** (didapat dari 2^7-1).

Contoh:

```
let numerik1 = 24;
let numerik2: i8 = 2;
```

Dengan menggunakan teknik deklarasi *type inference*, maka default angka numerik tipe datanya adalah `i32`.

Range value pada tipe data itu cukup penting untuk diperhatikan, jika ada sebuah variabel yang tipe datanya pasti dan diisi dengan nilai diluar kapasitas yang bisa ditampung, maka akan muncul error. Contoh, tipe data `i8`, jika diisi nilai `128` maka error.

```
main.rs 1, U ×
app4 tipe data > src > main.rs > main
    ▶ Run | Debug
1   fn main() {
2       let numerik1: i32 = 24;
3       let numerik2: i8 = 128;
4       let numerik3: i64 = 17;
5       println!("{} {} {}", numerik1, numerik2, numerik3);
6   }

PROBLEMS 6   OUTPUT   GITLENS   JUPYTER   DEBUG CONSOLE   TERMINAL

(base) PS D:\learn-rust\app4_tipe_data> cargo run
   Compiling app4_tipe_data v0.1.0 (D:\learn-rust\app4_tipe_data)

error: literal out of range for `i8`
 --> src\main.rs:3:24
  |
3 |     let numerik2: i8 = 128;
  |                        ^^^
  |
  = note: `#[deny(overflowing_literals)]` on by default
  = note: the literal `128` does not fit into the type `i8` whose range is `-128..=127`
  = help: consider using the type `u8` instead

error: could not compile `app4_tipe_data` due to previous error
```

Umumnya, tipe data `i32` cukup untuk kebutuhan menampung nilai, tapi ada banyak case dimana kita perlu tipe dengan size yang lebih besar seperti `i64`.

Berikut merupakan list tipe data signed integers yang ada di Rust. Tidak perlu dihafal.

Tipe data	Deskripsi
`i8`	$-(2^7)$ hingga (2^7)-1
`i16`	$-(2^{15})$ hingga (2^{15})-1
`i32`	$-(2^{31})$ hingga (2^{31})-1
`i64`	$-(2^{63})$ hingga (2^{63})-1
`i128`	$-(2^{127})$ hingga (2^{127})-1
`isize`	tipe pointer

> *Lebih jelasnya tentang tipe data **pointer** dibahas pada chapter terpisah, Pointer & References*

A.5.2. Unsigned integers

Perbedaan antara signed dan unsigned adalah pada range value yang ditampung, size-nya sama, hanya angka minimal dan maksimalnya berbeda.

Unsigned disini maksudnya adalah hanya bisa menampung angka mulai dari 0. Tipe data unsigned tidak bisa menampung angka negatif (jika dipaksa akan memunculkan error).

Berikut list tipe data unsigned integer pada Rust:

Tipe data	Deskripsi
`u8`	0 hingga (2^8)-1
`u16`	0 hingga (2^{16})-1
`u32`	0 hingga (2^{32})-1
`u64`	0 hingga (2^{64})-1
`u128`	0 hingga (2^{128})-1
`usize`	tipe pointer

> *Lebih jelasnya tentang tipe data **pointer** dibahas pada chapter terpisah, Pointer & References*

Contoh penerapan unsigned integer:

```
let numerik4: u32 = 28;
let numerik5: u8 = 16;
let numerik6: u64 = 42;

println!("{} | {} | {}", numerik4, numerik5, numerik6);
// output ==> 28 | 16 | 42
```

Catatan saja, variabel yang dideklarasikan dengan predefined value adalah numerik, by default tipe datanya adalah `i32` (signed).

A.5.3. Floating point

Floating point adalah tipe data yang mendukung nilai dibelakang koma, contohnya seperti `3.14`. Di Rust ada dua tipe data floating point, yaitu `f34` dan `f64`. Contoh penggunaan:

```
let fp1: f32 = 3.14;
let fp2: f64 = 3.1415926535;

println!("{} | {:.5}", fp1, fp2);
// output ==> 3.14 | 3.14159
```

O iya, jumlah digit di belakang koma bisa diatur saat diprint menggunakan `println`. Caranya dengan menggunakan notasi `{:.n}`. Sebagai contoh `{:.5}` maka akan menampilkan hanya 5 digit dibelakang koma.

> *Dalam penerapan definisi variabel sekaligus nilai, jika nilai dituliskan dalam desimal seperti contohnya `24`, maka tipe data variabel penampung adalah numerik.*
>
> *Jika dituliskan dalam notasi floating point, contohnya `3.14`, maka tipe data variabel penampung adalah float.*

A.5.4. Bool

Tipe data `bool` menerima dua pilihan nilai saja, `true` atau `false`.

```
let b1 = true;
let b2 = false;

println!("{} | {}", b1, b2);
// output ==> true | false
```

A.5.5. Char

Tipe `char` menampung sebuah data (unicode), contohnya seperti `'n'`, `'-'`, `'2'`. Penulisan literal untuk tipe ini menggunakan notasi `''`, diapit tanda petik satu.

```
let c1 = 'n';
let c2 = '-';
let c3 = '2';

println!("{} | {} | {}", c1, c2, c3);
// output ==> n | - | 2
```

A.5.6. Pointer scalar

Deklarasi tipe data pointer cukup mudah, yaitu dengan menuliskan deklarasinya seperti biasa tapi ditambahkan karakter `&`.

```
let ptr1: &i32 = &24;
println!("{}", ptr1);
// output ==> 24
```

Perihal apa itu tipe data pointer dan apa kegunaan prefix `&` akan dibahas pada

chapter terpisah, yaitu Pointer & References.

A.5.7. Tipe data primitive compound

Selain beberapa tipe data yang sudah dibahas di atas, ada juga jenis tipe data primitif jenis lainnya, yaitu primitive compound yang diantaranya adalah Array, Slice, Tuple, dan string. Tipe-tipe tersebut dibahas pada chapter terpisah.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-
example/../tipe_data_primitive_scalar
```

◉ Chapter relevan lainnya

- Tipe Data → String Literal (&str)
- Array
- Slice (Basic)
- Tuple
- Tipe Data → String Custom Type

◉ Referensi

- https://doc.rust-lang.org/rust-by-example/primitives.html
- https://doc.rust-lang.org/std/fmt/index.html#syntax
- https://en.wikipedia.org/wiki/Integer_(computer_science)

A.6. Tipe Data → String Literal (&str)

String adalah tipe data penting dalam pemrograman manapun. Biasanya tiap bahasa punya cara berbeda dalam meng-handle tipe ini. Di bahasa Rust, ada dua jenis tipe data string:

- Tipe string literal (kadang disebut dengan string slice, atau `&str`). Tipe data ini ada pada nilai yang dideklarasikan dengan diapit tanda petik dua (string literal), contohnya `"Hello, world!"`.
- Tipe `String` yang merupakan tipe data custom atau *custom types* yang merupakan sebuah struct. Lebih jelasnya akan kita bahas pada chapter terpisah.

Chapter ini hanya fokus pada string literal, dengan level pembahasan yang tidak terlalu advance.

A.6.1. String literal atau `&str`

> *Kita tidak akan bahas terlalu low-level untuk topik ini, karena memang masih di chapter awal-awal.*

Rust adalah bahasa yang *statically typed*, tipe data harus diketahui saat kompilasi. Setiap pendefinisian variabel, entah itu dengan di-specify tipe datanya (contoh: `let x: i32 = 5`) atau menggunakan teknik *type inference* (contoh: `let x = 5`), tipe datanya akan diketahui di awal saat kompilasi

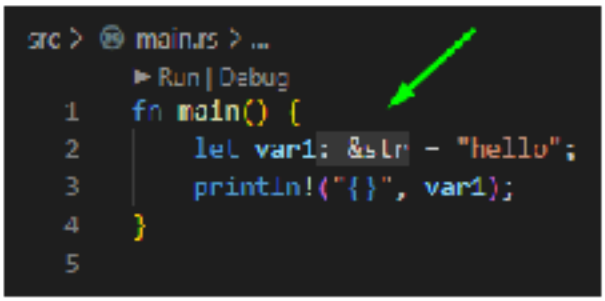
program.

Begitu juga pada tipe string. Sebagai contoh, statement `let y = "hello"`, variabel `y` disini memiliki tipe data, yaitu `&str`.

Apa itu tipe `&str`? Tipe `str` merupakan salah satu tipe primitif yang ada di Rust. Penulisan `&str` menandakan bahwa tipe tersebut adalah **pointer** `str`.

> *Untuk pembahasan lebih detail mengenai tanda `&` pada tipe data akan dibahas pada chapter terpisah, Pointer & References*

Untuk kawan-kawan yang menggunakan VSCode dengan ekstensi `rust-analyzer`, tipe data variabel bisa terlihat saat definisi.



Ok, sampai sini yang paling penting adalah dipahami dulu bahwa string literal tipe datanya adalah `&str`. Jadi selanjutnya kalau melihat tipe data `&str` jangan bingung, itu berarti string.

A.6.2. Escape karakter menggunakan `\`

Tanda `\` digunakan untuk *escape* beberapa karakter string.

Silakan perhatikan contoh berikut. Variabel `var2` isinya adalah string yang

didalamnya ada beberapa karakter yang di escape, yaitu tanda `"` dan baris baru atau newline.

```
let var2 = "hello \
    \"rust\" \
    and \
    \"world\"";
println!("{}", var2);
```

Coba jalankan kode di atas.

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarp
   Compiling tipe_data_string_literal v0.1.0 (D:\Labs\Adam Stu
    Finished dev [unoptimized + debuginfo] target(s) in 0.2"s
     Running `target\debug\tipe data string literal.exe`
hello "rust" and "world"
```

Karena semua baris baru pada contoh di atas di-escape, jadinya string akan tetap 1 baris. Tanda `"` juga tetap muncul karena di-escape.

A.6.3. Multiline string literal

Penulisan string banyak baris atau *multiline string* adalah sama seperti penulisan string biasa, yaitu langsung tulis saja string dengan diapit tanda `"`, tambahkan baris baru didalam string tersebut sesuai kebutuhan.

```
let var3 = "baris satu
baris dua
baris tiga";
println!("{}", var3);
```

Hasilnya adalah sesuai dengan string yang sudah didefinisikan.

```
   Compiling tipe_data_string_literal v0.1.0 (D:\Labs\Adam Stud
    Finished dev [unoptimized + debuginfo] target(s) in 0.45s
     Running `target\debug\tipe_data_string_literal.exe`
baris satu
baris dua
baris tiga
```

Perlu diketahui bahwa karakter spasi, baris baru, dan lainnya adalah **tidak dihiraukan**, jadi jika kawan-kawan menuliskan string multiline seperti ini ...

```
let var4 = "baris satu
    baris dua
    baris tiga";
println!("{}", var4);
```

... maka hasilnya juga sesuai dengan yang ditulis, yaitu ada 4 karakter spasi di baris ke-2 dan ke-3.

```
   Compiling tipe_data_string_literal v0.1.0 (D:\Labs\Adam Stud
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
     Running `target\debug\tipe_data_string_literal.exe`
baris satu
        baris dua
        baris tiga
```

A.6.4. Raw string

Raw string adalah istilah untuk string yang tidak meng-*escape* karakter apapun. Di Rust, string literal bisa didefinisikan dengan menuliskan string diapit `r#"` dan `"#`. Contoh:

```
let var5 = r#"
    {
        "name": "tim drake",
        "gender": "male"
    }
"#;
println!("{}", var5);
```

Kode di atas hasilnya adalah ekuivalen dengan kode dibawah ini, yang dimana string didefinisikan dengan meng-escape karakter `"` menggunakan `\`.

```
let var6 = "
    {
        \"name\": \"cassandra cain\",
        \"gender\": \"female\"
    }
";
println!("{}", var6);
```

```
Compiling tipe_data_string_literal v0.1.0 (D:\Labs\Adam Stu
 Finished dev [unoptimized + debuginfo] target(s) in 0.43s
  Running `target\debug\tipe_data_string_literal.exe`

 {
     "name": "tim drake",
     "gender": "male"
 }


 {
     "name": "cassandra cain",
     "gender": "female"
 }
```

A.6.5. Pembahasan lanjutan

tentang string

String adalah salah satu topik yang sangat luas cakupan pembahasannya, tidak cukup jika dirangkum dalam 1 chapter. Selain itu, bisa bikin makin bingung jika dibahas sekarang.

Penulis anjurkan untuk mempelajari chapter-chapter berikutnya secara urut terlebih dahulu.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-
example/../tipe_data_string_literal
```

◉ Chapter relevan lainnya

- Pointer & References
- String Literal (&str) vs. String Custom Type
- Tipe Data → String Custom Type

◉ Referensi

- https://doc.rust-lang.org/reference/tokens.html#raw-string-literals
- https://doc.rust-lang.org/std/str/index.html

- https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/strings.html

---

A.7. Konstanta

Chapter ini membahas tentang konstanta di bahasa Rust.

A.7.1. Keyword `const`

Berbeda dengan variabel yang dideklarasikan menggunakan keyword `let`, konstanta dibuat dengan keyword `const`. Contoh:

```
const LABEL: &str = "nilai pi adalah:";
const PI: f32 = 22.0/7.0;
println!("{} {}", LABEL, PI);
```

Bisa dilihat di kode di atas, konstanta `LABEL` merupakan string dengan nilai `"nilai pi adalah:"`, sedang `PI` memiliki nilai bertipe float hasil dari operasi `22./7.0`.

Dalam pendefinisian konstanta, tipe data harus dituliskan secara eksplist. Deklarasi seperti `const DATA = "x"` akan menghasilkan error saat proses kompilasi. Dan aturan ini berlaku untuk semua tipe data yang dipergunakan untuk pendefinisian konstanta.

Nilai sebuah konstanta juga bisa dari variabel atau konstanta lain, atau hasil sebuah operasi seperti operasi aritmatika `22.0/7.0`.

Disini penulis tidak menggunakan `22/7` karena hasilnya akan bertipe integer. Di Rust operasi aritmatika harus dilakukan dengan tipe data yang sama, dan hasilnya akan memiliki tipe data sesuai operand. Lebih jelasnya akan kita

bahas pada chapter selanjutnya.

> *Keyword `mut` tidak bisa diterapkan pada konstanta. Jika dipaksa akan menghasilkan error.*

A.7.2. Keyword `static`

Ada cara lain untuk membuat konstanta, menggunakan keyword `static`. Contohnya seperti pada kode berikut. Konstanta `NUMBER` didefinisikan di luar blok fungsi main, lalu di print di dalam fungsi main.

```
static NUMBER: i32 = 18;

fn main() {
    // ...

    println!("{}", NUMBER);
    // output ==> 18
}
```

Pertanyaan, bedanya apa dengan konstanta yang dibuat via keyword `const`? secara teknis bedanya ada di bagaimana manajemen dan alokasi memori dilakukan di belakang layar.

> *Di Rust, konstanta yang dibuat via `const` tidak memiliki alamat memori yang pasti, dan setiap kali dipergunakan maka terjadi proses copy value. Sedangkan konstanta yang dibuat via keyword `static` mempunyai alamat memori yg fix/pasti.*
>
> *Lebih jelasnya mengenai static nantinya dibahas pada chapter Static.*

> *Untuk sekarang silakan lanjut dulu ke pembahasan ke chapter-chapter berikutnya secara berurutan.*

A.7.3. *Naming convention* konstanta

Sesuai anjuran di halaman dokumentasi Rust,*Naming convention*, aturan penulisan konstanta adalah menggunakan **screaming snake case**. Nama konstanta dituliskan dalam huruf kapital dengan separator tanda garis bawah atau underscore (`_`). Contohnya seperti: `NUMBER`, `SOME_DATA`, `CONFIGURATION_ENV_NAME`, dll.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../konstanta
```

◉ Chapter relevan lainnya

- Static Item
- Lifetime

◉ Referensi

- https://rust-lang.github.io/api-guidelines/naming.html
- https://doc.rust-lang.org/rust-by-example/custom_types/constants.html
- https://users.rust-lang.org/t/const-vs-static/52951/2
- https://mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/const-and-static.html

---

A.8. Operator

> *This chapter is still under development*

Chapter ini membahas mengenai operator pada pemrograman rust.

A.8.1. Operator Aritmatika

Berikut merupakan list operator untuk operasi aritmatika beserta contoh penerapannya di rust.

Simbol	Kegunaan
`+`	penambahan
`-`	pengurangan
`*`	pengalian
`/`	pembagian
`%`	*modulus* atau sisa hasil bagi

```
let (num1, num2) = (12, 4);
```

A.8.2. Operator Perbandingan

Operator perbandingan selalu menghasilkan nilai bertipe data `bool`.

Berikut merupakan list operator untuk operasi perbandingan beserta contoh penerapannya di rust.

Simbol	Kegunaan untuk mengecek
`==`	apakah kiri sama dengan kanan?
`!=`	apakah kiri tidak sama dengan kanan?
`>`	apakah kiri lebih besar dari kanan?
`<`	apakah kiri lebih kecil dari kanan?
`>=`	apakah kiri lebih besar atau sama dengan kanan?
`<=`	apakah kiri lebih kecil atau sama dengan kanan?

```
let number_a = 12;
let number_b = 24;

let res_one = number_a == number_b;
println!("res_one: {res_one}");

let res_two = number_a != number_b;
println!("res_two: {res_two}");
```

Pada contoh di atas, variabel di print menggunakan macro `println` tanpa disisipkan paramnya. Penjelasannya ada di bawah ini.

◉ *Named argument* macro `println`

Salah satu teknik *formatted print* macro `println` adalah dengan menerapkan *named argument*. Yang biasanya menggunakan `{}` atau `{1}`, `{2}`, dan seterusnya, diganti dengan nama variabel yang diapit tanda kurung kurawal, contohnya `res_one`. Dengan teknik ini maka jika variabel `res_one` ada, akan langsung mereplace argument `{res_one}` tanpa perlu menyisipkan variabel tersebut saat pemanggilan macro `println`.

```
let res_one = number_a == number_b;
println!("res_one: {res_one}");
// output => res_one: false

let res_two = number_a != number_b;
println!("res_two: {res_two}");
// output => res_one: true
```

A.8.3. Operator Negasi

Berikut merupakan list operator untuk operasi negasi beserta contoh penerapannya di rust.

Simbol	Kegunaan
`-`	negasi numerik

Simbol	Kegunaan
`!`	logika `NOT`

```
let (value_left, value_right) = (12, -12);
let res_one = -value_left == value_right;
let res_two = !(value_left == value_right);
println!("{res_one} {res_two}");
// output => true true
```

Untuk menggunakan operator negasi pada tipe data numerik, caranya dengan langsung menambahkan prefix `-` pada angka atau variabel. Contohnya `-12` atau `value_left`.

Penggunaan operator logika `!` juga sama, tinggal tambahkan saja sebagai prefix dari data `bool` atau statement yang menghasilkan data `bool`, contohnya `!(value_left == value_right)`.

A.8.4. Operator logika / `bool`

Berikut merupakan list operator untuk operasi logika `bool` beserta contoh penerapannya di rust.

Simbol	Kegunaan
`&&`	logika `AND`
||	logika `OR`

```
let (bool_left, bool_right) = (false, true);
println!("AND result \t: {}", bool_left && bool_right);
println!("OR result \t: {}", bool_left || bool_right);
```

◉ *Whitespace character* tab `\t`

Rust mendukung karakter standar whitespace seperti `\t` yang kegunaannya adalah untuk horizontal tab. Contoh penerapannya seperti pada kode di atas, tulis saja `\t` dalam string literal, hasilnya bisa dilihat saat di-print.

```
println!("AND result \t: {}", bool_left && bool_right);
println!("OR result \t: {}", bool_left || bool_right);
```

```
D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust
    Finished dev [unoptimized + debuginfo]
     Running `target\debug\app7_operator.exe`
AND result      : true
OR result       : false
```

A.8.5. Operator reference dan dereference

Untuk jenis operasi ini ada 3 buah operator yang bisa dipergunakan yaitu `*`, `&` dan `&mut`. Untuk sekarang kita tidak akan membahas topik ini karena masih terlalu awal. Nantinya kita akan kupas tuntas di chapter Pointer & References. Sementara kita pelajari chapter per chapter secara berurutan dulu.

A.8.6. Operator bitwise

Rust mendukung operator bitwise standar yang ada di bahasa pemrograman.

Simbol	Kegunaan
`&`	bitwise AND
`|`	bitwise OR
`^`	bitwise XOR
`<<`	left shift
`>>`	right shift

A.8.7. Operator lainnya

Sebenarnya ada banyak lagi jenis operator di Rust programming, seperti type cast operator, reference/borrow operator, dll. Nantinya kita akan bahas operator tersebut satu per satu secara terpisah di chapter berbeda.

> - *Type cast operator dibahas pada chapter Type Alias & Casting*
> - *Reference/borrow operator dibahas pada chapter Pointer & References*

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../operator
```

◉ Chapter relevan lainnya

- Pointer & References
- Macro
- Whitespace Token

◉ Referensi

- https://doc.rust-lang.org/reference/expressions/operator-expr.html
- https://doc.rust-lang.org/reference/whitespace.html
- https://doc.rust-lang.org/rust-by-example/hello/print.html

A.9. Seleksi Kondisi → if, else if, else

Pada chapter ini kita akan bahas tentang penerapan `if` untuk kontrol alur program.

Seleksi kondisi sendiri merupakan teknik untuk grouping blok kode yang eksekusinya tergantung hasil ekspresi seleksi kondisi. Analoginya mirip seperti fungsi rambu lalu lintas di jalan raya. Kapan kendaraan diperbolehkan melaju dan kapan harus berhenti diatur oleh rambu tersebut. Seleksi kondisi pada program juga kurang lebih sama, kapan sebuah blok kode akan dieksekusi dikontrol.

A.9.1. Keyword `if`

`if` adalah salah satu keyword untuk seleksi kondisi di Rust, penggunaannya sangat mudah, yaitu dengan cukup tulis keyword tersebut diikuti dengan data boolean (atau ekspresi logika yang hasilnya boolean), lalu diikuti blok kode. Notasi penulisan `if` seperti berikut.

```
if operasiLogika {
    // blok kode
}
```

Pada notasi di atas, `operasiLogika` bisa diisi dengan variabel yang bertipe `bool`, atau statement ekspresi perbandingan seperti `a == b`. Lebih jelasnya

sekarang silakan perhatikan dan coba kode berikut.

```
let number_a = 3;
if number_a < 5 {
    println!("number_a adalah dibawah 5");
}

let result_a = number_a >= 5;
if result_a {
    println!("result_a adalah diatas atau sama dengan 5");
}
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrograman rust\dasar
   Compiling app8_control_flow_if v0.1.0 (D:\Labs\Adam Studio\
    Finished dev [unoptimized + debuginfo] target(s) in 0.45s
     Running `target\debug\app8_control_flow_if.exe`

number_a adalah dibawah 5
```

Pada kode di atas ada dua buah blok kode `if`. Yang pertama mengecek hasil ekspresi logika `apakah variabel number_a dibawah 5?`. Jika hasilnya benar atau `true` maka blok kode setelahnya yang diapit tanda kurung kurawal akan dieksekusi, hasilnya menampilkan tulisan `angka adalah dibawah 5`.

Blok kode `if` kedua adalah mengecek nilai `bool` variabel `result_a`. Variabel `result_a` sendiri isinya berasal dari ekspresi logika `apakah variabel number_a lebih besar atau sama dengan 5?`. Jika hasilnya `true` maka blok kode setelahnya (yang diapit tanda kurung kurawal) dieksekusi. Namun, pada contoh di atas, hasilnya adalah `false`, karena variabel `number_a` nilainya adalah tidak lebih besar atau sama dengan 5, dengan demikian blok kode tidak dieksekusi.

A.9.2. Keyword `if`, `else if`, dan

`else`

Jika seleksi kondisi lebih dari satu, gunakan `if` dan `else if` dan/atau `else`.

```
let number_b = 3;
if number_b == 2 {
    println!("number_b adalah 2");
} else if number_b < 2 {
    println!("number_b adalah dibawah 2");
} else {
    println!("number_b adalah diatas 2");
}
```

Pada contoh di atas, `number_b` yang nilainya `2` match dengan statement seleksi kondisi `number_b == 2`, maka statement dalam blok kode tersebut dieksekusi, text `number_b adalah 2` muncul.

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasar
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\app8_control_flow_if.exe`
number_b adalah diatas 2
```

Silakan bermain dengan nilai variabel `number_b` untuk coba-coba.

A.9.3. Nested `if`

Sebuah blok kode `if` bisa saja berada didalam sebuah `if`, dan seperti ini umum terjadi di bahasa pemrograman. Di Rust penerapan nested `if` sama seperti pada bahasa lainnya, yaitu dengan langsung tuliskan saja blok kode `if`

ke dalam blok kode `if`. Contoh:

```
let number_c = 10;
if number_c > 6 {
    println!("selamat, anda lulus");

    if number_c == 10 {
        println!("dengan nilai sempurna");
    } else if number_c > 7 {
        println!("dengan nilai baik");
    } else {
        println!("dengan nilai cukup");
    }
} else {
    println!("anda tidak lulus");

    if number_c < 4 {
        println!("belajar lagi ya");
    } else {
        println!("jangan malas belajar!");
    }
}
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarp
   Compiling app8_control_flow_if v0.1.0 (D:\Labs\Adam Studio\E
    Finished dev [unoptimized + debuginfo] target(s) in 0.42s
     Running `target\debug\app8_control_flow_if.exe`

selamat, anda lulus
dengan nilai sempurna
```

A.9.4. Returning From `if`

Returning from `if` adalah salah satu cara unik penerapan `if`. Yang biasanya `if` digunakan untuk eksekusi statements dengan kondisi tertentu, pada case

ini statement yang ada pada blok kode if ditampung sebagai *return value* atau nilai balik. Teknik ini mirip seperti operasi *ternary* hanya saja jumlah kondisinya bisa sebanyak yang kita inginkan.

Agar lebih jelas, silakan pelajari dulu kode berikut.

```
let number_d = 3;
let result_d: bool;

if number_d == 2 {
    result_d = true
} else {
    result_d = false
}

println!("result_d adalah {result_d}");
```

Blok seleksi kondisi pada contoh di atas menjadi penentu nilai variabel `result_d`. Dengan kebutuhan tersebut, kita bisa juga memanfaatkan `let if` untuk mendapatkan hasil yang ekuivalen.

Kode di atas jika dirubah ke bentuk `let if` hasilnya menjadi seperti ini:

```
let number_d = 3;
let result_d = if number_d == 2 {
    true
} else {
    false
};
println!("result_d adalah {result_d}");
```

Penulisannya cukup unik. Blok kode seleksi kondisi di tuliskan sebagai value dari statement `let result_d`. Dan isi blok kode nantinya akan menjadi value

untuk variabel `result_d`, tergantung kondisi mana yang match.

Pada contoh di atas, karena `number_d` nilainya 3, maka `result_d` bernilai false. Blok kode `else` adalah yang dieksekusi.

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrograman
   Compiling app8_control_flow_if v0.1.0 (D:\Labs\Adam Studio\
    Finished dev [unoptimized + debuginfo] target(s) in 1.18s
     Running `target\debug\app8_control_flow_if.exe`
result_d adalah false
```

> *Dalam penggunakan kombinasi keyword `let` dan `if`, pastikan di akhir blok kode seleksi kondisi dituliskan tanda `;`.*

O iya, beberapa orang lebih memilih memanfaatkan indentasi untuk mempermudah memahami statement `let if`. Contohnya seperti ini:

```
let number_d = 3;
let result_d =
    if number_d == 2 {
        true
    } else {
        false
    };
println!("result_d adalah {result_d}");
```

A.9.5. Kombinasi Keyword `let` dan `if`, Dengan Tipe Data Eksplisit

Ada situasi dimana dalam pemanfaatan `let if` kita perlu men-specify secara

eksplisit tipe data variabel penampung. Caranya sama seperti statement deklarasi variabel beserta tipe data, langsung tulis saja tipe data yang diinginkan setelah nama variabel dan sebelum operator `=`.

Pada contoh berikut, variabel `result_e` saya definisikan tipenya adalah `string literal &str`.

```
let number_e = 3;
let result_e: &str = if number_e == 2 {
    "angka adalah 2"
} else if number_e < 2 {
    "angka adalah dibawah 2"
} else {
    "angka adalah diatas 2"
};
println!("angka adalah {result_e}");
```

Contoh lain:

```
let max = 100.0;
let string_f = "nilai minimum kelulusan";
let result_f: f64 = if string_f == "nilai maksimum kelulusan" {
    max
} else {
    max * 3.0 / 4.0
};
println!("angka adalah {result_f}");
```

```
(base) PS D:\Labs\Adam Studio\Ubook\dasarpemrogramanrust\dasar
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\app8_control_flow_if.exe`
angka adalah 75
```

A.9.6. Keyword `if let`

Keyword `if let` berbeda dengan kombinasi `let` dan `if`. Kita akan bahas topik ini secara terpisah pada chapter Pattern Matching.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-
example/../seleksi_kondisi_if
```

◉ Referensi

- https://doc.rust-lang.org/book/ch03-05-control-flow.html
- https://doc.rust-lang.org/std/keyword.if.html
- https://doc.rust-lang.org/std/keyword.else.html
- https://doc.rust-lang.org/rust-by-example/flow_control/if_else.html

---

A.10. Perulangan → while

Di Rust ada beberapa cara untuk melakukan perulangan, salah satunya adalah dengan menggunakan keyword `while`, dan pada chapter ini kita akan mempelajarinya.

Perulangan sendiri merupakan teknik untuk mengulang-ulang eksekusi blok kode dengan kondisi tertentu. Perulangan akan terus menerus dieksekusi ketika kondisi perulangan nilainya `true`, dan hanya akan berhenti kita nilainya `false`.

A.10.1. Keyword `while`

Perulangan menggunakan `while` mengharuskan kondisi perulangan dituliskan di awal dengan notasi penulisan sebagai berikut:

```
while kondisi {

}
```

Contoh berikut adalah penerapan `while` untuk operasi perulangan yang isinya menampilkan angka `i` dengan kondisi `i` dibawah `max`.

```
let mut i = 0;
let max = 5;
```

```
   Compiling app9_control_flow_while v0.1.0 (D:\Labs\Adam Studio
    Finished dev [unoptimized + debuginfo] target(s) in 0.73s
     Running `target\debug\app9_control_flow_while.exe`
nilai: 0
nilai: 1
nilai: 2
nilai: 3
nilai: 4
```

Variabel `i` pada contoh di atas menjadi penentu kapan perulangan berhenti. Didalam blok kode `while` (yang dimana akan dieksekusi setiap kondisi menghasilkan nilai `true`), nilai variabel `i` di-increment, membuat variabel `i` nilainya selalu bertambah 1 setiap kali perulangan. Perulangan akan berhenti ketika nilai `i` sudah tidak dibawah `i` lagi.

A.10.2. Nested `while`

Penerapan nested while (atau `while` didalam `while`) cukup mudah, tulis saja statement `while` didalam `while` sesuai kebutuhan. Contoh bisa dilihat pada kode berikut, yaitu penerapan teknik nested while untuk print karakter `*` membuat bentuk segitiga.

```
let mut i = 0;
let max = 5;

while i < max {
    let mut j = 0;
    let max_inner = i;

    while j <= max_inner {
        print!("* ");
        j += 1;
    }
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrograman-rust\dasarp
   Compiling app9_control_flow_while v0.1.0 (D:\Labs\Adam Studi
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `target\debug\app9_control_flow_while.exe`
*
* *
* * *
* * * *
* * * * *
```

◉ Macro `print`

Macro `print` kegunaannya mirip dengan `println`, hanya saja tanpa *newline* atau baris baru. Jadi semua string yang di-print menggunakan macro `print` akan muncul menyamping ke kanan dan tidak di baris baru. Contoh:

```
print!("* ");
print!("* ");
print!("* ");
print!("* ");
```

Akan menghasilkan output:

```
* * * *
```

A.10.3. Keyword `while let`

Kita akan bahas keyword `while let` pada chapter Pattern Matching.

A.10.4. Menambahkan delay dalam

perulangan

Eksekusi statement perulangan pada program sangatlah cepat. Bisa jadi dalam 1 detik ada ribuan (atau bahkan jauh lebih banyak) statement dalam perulangan yang dieksekusi. Di-tiap eksekusinya, delay bisa saja ditambahkan, misalnya 1 detik, dan pada contoh di bawah ini kita akan coba mengaplikasikannya.

Silakan tulis kode program berikut:

```
use std::thread::sleep;
use std::time::Duration;

fn main() {
    let mut i = 0;
    let max = 5;

    while i < max {
        println!("nilai: {i}");
        i += 1;

        sleep(Duration::from_secs(1));
    }
}
```

Ada beberapa keyword baru dipergunakan pada source code di atas. Untuk sekarang coba jalankan terlebih dahulu program yang baru dibuat, kemudian stop. Text `nilai: {i}` akan muncul di console setiap satu detik, dan hanya akan berhenti ketika program di stop.

Selanjutnya kita akan beberapa sintaks baru di atas.

◉ Keyword `use` dan module dalam Rust

Keyword `use` memiliki banyak kegunaan. Pada contoh ini `use` difungsikan untuk *import module*, yang di bahasa Rust dikenal dengan istilah **import paths**.

Notasi penulisan *module path* di Rust dinotasikan dengan separator `::`, contohnya seperti `std::thread::sleep` dan `std::time::Duration`.

- Statement `use std::thread::sleep` artinya path `std::thread::sleep` digunakan dalam kode program. `sleep` adalah sebuah fungsi yang gunanya untuk menambahkan jeda dengan durasi sesuai keinginan, contohnya, 1 detik, 30 menit, dst.

- Statement `use std::time::Duration` artinya path `std::time::Duration` digunakan dalam kode program. `Duration` merupakan sebuah struct yang isinya banyak hal untuk keperluan yang berhubungan dengan waktu atau *duration*.

Pada contoh di atas, fungsi `sleep` dipanggil dalam blok kode perulangan agar ada jeda ditiap eksekusinya. Sedangkan durasi jedanya sendiri ditentukan oleh argument `Duration::from_secs(1)` yang artinya durasi `1 detik`. Dengan ini durasi delay adalah 1 detik.

Jika ingin jeda durasi yang lebih lama, ubah saja angka `1` pada `Duration::from_secs(1)`. Contoh: `Duration::from_secs(5)` berarti jeda 5 detik.

> - *Lebih jelasnya perihal path dibahas pada chapter Module System → Pach & Item*

> - *Lebih jelasnya perihal keyword `use` dibahas pada chapter Module System → Use*
> - *Lebih jelasnya perihal fungsi dibahas pada chapter Function*

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-
example/../perulangan_while
```

◉ Referensi

- https://doc.rust-lang.org/book/ch03-05-control-flow.html
- https://doc.rust-lang.org/std/keyword.while.html
- https://doc.rust-lang.org/std/keyword.use.html
- https://doc.rust-lang.org/rust-by-example/flow_control/while.html
- https://doc.rust-lang.org/stable/std/time/struct.Duration.html
- https://doc.rust-lang.org/reference/expressions/while-expr.html
- https://doc.rust-lang.org/reference/paths.html

A.11. Perulangan → loop, break, continue, label

Selain keyword `loop`, ada juga keyword `loop` yang fungsi dasarnya adalah sama, yaitu untuk perulangan.

A.11.1. Keyword `loop`

Notasi penulisan dan cara penggunaan `loop` ada sedikit beda dibanding `loop`. Keyword `loop` tidak membutuhkan argument. Blok kode loop akan terus dieksekusi selama program tidak di-stop.

Silakan coba praktekan kode berikut. Angka `i` akan ditampilkan setiap perulangan kemudian di-increment nilainya. Angka akan muncul terus sampai aplikasi di-stop.

```
fn main() {
    let mut i = 0;

    loop {
        println!("nilai: {i}");
        i += 1;
    }
}
```

```
nilai: 665
nilai: 666
nilai: 667
nilai: 668
nilai: 669
nilai: 670
nilai: 671
nilai: 672
nilai: 673
error: process didn't exit successfully: `target\debug\app10_control_flow_loop.exe`
(exit code: 0xc000013a, STATUS_CONTROL_C_EXIT)
```

A.11.2. Keyword `break`

`loop` menghasilkan perulangan tanpa henti, lalu bagaimana cara stop-nya? Disinilah keyword `break` berperan. `break` digunakan untuk menghentikan paksa eksekusi blok kode perulangan. Biasanya keyword ini digunakan dalam kondisi tertentu (sesuai kebutuhan), misalnya `perulangan harus berhenti ketika nilai i di atas max`.

Mari kita ubah kode di atas dengan menambahkan kondisi untuk menghentikan perulangan jika `i` nilainya lebih dari `5`.

```rust
let mut i = 0;
let max = 5;

loop {
    println!("nilai: {i}");
    i += 1;
    if i > max {
        break;
    }
}
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarp
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\app10_control_flow_loop.exe`

nilai: 0
nilai: 1
nilai: 2
nilai: 3
nilai: 4
nilai: 5
perulangan selesai
```

> *Penulisan* `break` *boleh tanpa diakhiri semicolon* `;`

A.11.3. Nested `loop`

Cara menerapkan nested loop (atau `loop` didalam `loop`), tulis saja statement `while` didalam `while` sesuai kebutuhan.

Pada kode berikut, teknik nested while diterapkan untuk membuat bentuk segitiga menggunakan karakter `*`.

```
let mut i = 0;
let max = 5;

loop {
    let mut j = max;
    let max_inner = i;

    loop {
        print!("* ");
        j -= 1;
        if j < max_inner {
            break;
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarp
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\app10_control_flow_loop.exe`
* * * * * *
* * * * *
* * * *
* * *
* *
*
```

A.11.4. Keyword `continue`

`continue` digunakan untuk melanjutkan paksa sebuah perulangan, kebalikan dari `break` yang fungsinya menghentikan paksa sebuah perulangan.

Source code berikut murapakan contoh penerapan `continue`. Variabel `i` berperan sebagai counter perulangan. Jika nilai `i` adalah ganjil, maka perulangan dipaksa lanjut ke iterasi berikutnya. Dengan ini maka macro `println` hanya akan menampilkan nilai genap. Dan program akan berhenti jika `i` nilainya lebih dari `max`.

```
let mut i = 0;
let max = 15;

loop {
    i += 1;

    if i % 2 == 1 {
        continue;
    }

    println!("nilai i: {i}");

    if i > max {
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarp
   Compiling app10_control_flow_loop v0.1.0 (D:\Labs\Adam Studi
    Finished dev [unoptimized + debuginfo] target(s) in 1.19s
     Running `target\debug\app10_control_flow_loop.exe`

nilai i: 2
nilai i: 4
nilai i: 6
nilai i: 8
nilai i: 10
nilai i: 12
nilai i: 14
nilai i: 16
```

A.11.5. Label perulangan

Statement perulangan menggunakan `loop` bisa ditandai dengan label. Manfaat dari penggunaan label adalah bisa mengeksekusi `break` atau `continue` ke perulangan di luar blok kode perulangan dimana statement itu berada. Umumnya label perulangan dipergunakan pada nested loop, dimana ada kebutuhan untuk menghentikan/melanjutkan paksa perulangan terluar.

Berikut adalah notasi penulisan loop dengan dan tanpa label. Nama label diawali dengan tanda petik `'`.

```rust
// loop biasa
loop {
    // statements
    break;
}

// loop dengan label
'namaLabel: loop {
    // statements
    break 'namaLabel;
}
```

Mari kita pelajari dan praktekan kode berikut ini. Dibawah ini adalah sebuah program sederhana menampilkan angka yang hasilnya bisa dilihat digambar dibawahnya. Perulangan di level 2 akan dihentikan secara paksa ketika `j > i`. Sedangkan perulangan level pertama atau terluar (dengan label `'mainLoop`) akan dihentikan paksa dari perulangan level 2 jika kondisi `i > max`.

```
let mut i = 0;
let max = 9;

'mainLoop: loop {
    i += 1;
    let mut j = 0;

    loop {
        if i > max {
            break 'mainLoop;
        }

        j += 1;
        if j > i {
            break;
        }

        print!("{i} ");
    }

    println!();
}
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarpemrograma
    Finished dev [unoptimized + debuginfo] target(s) in 0.81s
     Running `target\debug\app10_control_flow_loop_break_continue.exe`

1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
6 6 6 6 6 6
7 7 7 7 7 7 7
8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9
```

A.11.6. Returning from `loop`

Returning from `loop` merupakan teknik pemanfaatan `loop` dan `break` untuk menampung sebuah return value dari blok kode perulangan `loop`. Agar lebih jelas, silakan coba kode berikut:

```
let mut counter = 0;

let result = loop {
    counter += 1;

    if counter == 10 {
        break counter * 2;
    }
};

println!("result: {result}");
```

Pada kode di atas, variabel `result` dideklarasikan dengan predefined value adalah blok kode `loop`. Disini artinya variabel `result` isinya bukan blok kode perulangan `loop`, melainkan isinya adalah apapun yang dituliskan setelah keyword `break`. Unik ya?

Variabel `counter` yang nilai awalnya 0, di-increment dalam perulangan. Ketika nilai `counter` adalah `10`, nilai `counter * 2` dijadikan sebagai return value. Dengan ini maka variabel `result` nilainya adalah `20`.

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarpemrogramanrust\
   Compiling app10_control_flow_loop_break_continue_label v0.1.0 (D:\Labs\Adam
    Finished dev [unoptimized + debuginfo] target(s) in 0.59s
     Running `target\debug\app10_control_flow_loop_break_continue_label.exe`

result: 20
```

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-
example/../perulangan_loop_break_continue_label
```

◉ Referensi

- https://doc.rust-lang.org/book/ch03-05-control-flow.html
- https://doc.rust-lang.org/std/keyword.loop.html
- https://doc.rust-lang.org/std/keyword.break.html
- https://doc.rust-lang.org/std/keyword.continue.html
- https://doc.rust-lang.org/rust-by-example/flow_control/loop.html
- https://doc.rust-lang.org/rust-by-example/flow_control/loop/nested.html
- https://doc.rust-lang.org/rust-by-example/flow_control/loop/return.html
- https://doc.rust-lang.org/reference/expressions/loop-expr.html

---

A.12. Perulangan → for in

`for in` adalah salah satu keyword untuk operasi perulangan yang ada di Rust selain `loop` dan `while`. Pada chapter ini kita akan mempelajarinya.

A.12.1. Keyword `for in`

`for in` adalah keyword perulangan yang paling sering dipakai untuk meng-iterasi data yang tipe data-nya mengimplementasikan trait `Iterator`. Ada banyak jenis tipe data dengan trait ini, salah satunya adalah tipe data **range** yang akan kita bahas pada chapter ini.

> - *Lebih jelasnya mengenai traits dibahas pada chapter Traits*
> - *Lebih jelasnya mengenai trait `Iterator` dibahas pada chapter Trait → Iterator*

Tipe data range dibuat dengan notasi penulisan `a..b`. Mari lanjut dengan praktek agar lebih jelas. Silakan coba kode sederhana berikut lalu jalankan.

```
for i in 0..5 {
    println!("{i}");
}
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasar
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\perulangan for in.exe`

0
1
2
3
4
```

Keyword `for in` jika digunakan pada notasi iterator `a..b` maka akan menghasilkan sebuah perulangan dari angka `a` hingga angka dibawah `b`.

Pada contoh di atas, `0..5` artinya adalah objek iterator yang dimulai dari angka `0` hingga dibawah `5` (yaitu 4). Object iterator tersebut kemudian diiterasi, dan ditiap perulangan di-print menggunakan `println!("{i}")`. Dengan ini, nilai `i` muncul di layar console dimulai angka `0` hingga `4`.

Jika ingin melakukan perulangan dari `a` ke `b` (bukan dari `a` ke angka dibawah `b`) gunakan notasi iterator `a..=b`. Contoh:

```
for i in 0..=5 {
    println!("{i}");
}
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasar
   Compiling perulangan_for_in v0.1.0 (D:\Labs\Adam Studio\Eb
    Finished dev [unoptimized + debuginfo] target(s) in 0.47s
     Running `target\debug\perulangan_for_in.exe`

0
1
2
3
4
5
```

A.12.2. Label perulangan

Penambahan label pada perulangan `for in` juga bisa dilakukan. Caranya dengan cukup menambahkan sintaks `'namaLabel:` sebelum statement `for in`. Contohnya seperti berikut:

```
'perulangan: for i in 0..=5 {
    if i > 3 {
        println!("perulangan dihentikan paksa pada iterasi {i}");
        break 'perulangan;
    }

    println!("{i}");
}
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarp
   Compiling perulangan_for_in v0.1.0 (D:\Labs\Adam Studio\Ebor
    Finished dev [unoptimized + debuginfo] target(s) in 0.42s
     Running `target\debug\perulangan_for_in.exe`

0
1
2
3
perulangan dihentikan paksa pada iterasi 4
```

Selain `break`, keyword `continue` juga bisa digunakan pada perulangan `for in`.

A.12.3. Perulangan `for in` pada array

Perulangan menggunakan `for in` adalah yang paling praktis digunakan untuk mengiterasi elemen array. Contoh:

```
let array = ["jason", "grayon", "drake", "damian"];
for name in array {
    println!("{name}");
}
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrograman-rust\dasar
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\perulangan_for_in.exe`

jason
grayon
drake
damian
```

Perihal apa itu `array` akan dibahas pada chapter selanjutnya, yaitu Array.

A.12.4. Perulangan `for in` pada tipe iterator lainnya

Di Rust ada banyak cara membuat object iterator. Mengenai topik satu ini kita akan bahas seiring berjalannya proses pembelajaran. Akan berpotesi makin membingungkan jika dibahas terlalu detail pada chapter ini.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-
example/../perulangan_for_in
```

◉ Referensi

- https://doc.rust-lang.org/reference/expressions/loop-expr.html
- https://doc.rust-lang.org/std/keyword.in.html
- https://doc.rust-lang.org/std/keyword.for.html
- https://doc.rust-lang.org/rust-by-example/flow_control/for.html

---

A.13. Tipe Data → Array

Pada chapter awal kita sudah mempelajari tipe data primitif jenis scalas. Selain *scalar types* ada juga tipe data primitif lainnya yaitu **compound types**. Compound types sendiri adalah jenis tipe data kolektif yang isinya banyak data. Kesemua data tersebut memiliki tipe data yang sama dan di-group menjadi satu.

Array adalah salah satu tipe data compound yang tersedia di Rust, dan pada chapter ini kita akan mempelajarinya.

A.13.1. Pengenalan array

Menurut dokumentasi official Rust, array adalah:

> *An array is a collection of objects of the same type T, stored in contiguous memory.*
>
> *A fixed-size array, denoted [T; N], for the element type, T, and the non-negative compile-time constant size, N.*

Array (atau *fixed size array*) adalah kumpulan data dengan tipe sejenis, disimpan dalam 1 variabel. Array memiliki kapasitas yang nilainya ditentukan saat deklarasi/alokasi. Jumlah data dalam array pasti tidak boleh lebih dari kapasitas yang sudah ditentukan di awal. Data dalam array biasa disebut dengan *element* atau item.

Ada beberapa notasi deklarasi array yang bisa dipakai, kesemuanya akan

dibahas pada chapter ini.

Ok, Sekarang kita mulai praktek dengan bermain-main terlebih dahulu dengan tipe data array ini. Silakan pelajari dan praktekan contoh berikut, sebuah program yang isinya menampilkan data array.

```
let mut numbers = [24, 12, 32, 7];
println!("array {:?}", numbers);

let data0 = numbers[0];
println!("elemen array ke 0 {data0}");

let data1 = numbers[1];
println!("elemen array ke 1 {data1}");

numbers[1] = 16;
numbers[3] = 8;
println!("array {numbers:?}");
```

Coba jalankan aplikasi, muncul dua baris output di console.

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarp
   Compiling app11_array v0.1.0 (D:\Labs\Adam Studio\Ebook\dasa
    Finished dev [unoptimized + debuginfo] target(s) in 0.24s
     Running `target\debug\app11_array.exe`

array [24, 12, 32, 7]
elemen array ke 0 24
elemen array ke 1 12
array [24, 16, 32, 8]
```

Selanjutnya mari kita bahas dengan detail contoh di atas.

◉ Deklarasi variabel array (*type inference*)

Variabel `numbers` dideklarasikan sebagai array *mutable* dengan metode

deklarasi type inference, yang tipe datanya didapat langsung dari nilai.

Value dari `numbers` adalah `[24, 12, 32, 7]`, yang dimana artinya sebuah array dengan size 4, bertipe numerik, dengan isi `24`, `12`, `32`, `7`.

Sintaks `[24, 12, 32, 7]` adalah salah satu cara menulis literal array. Tulis saja data yang diinginkan dengan separator `,` dan diapit tanda kurung siku `[ ]`.

Contoh lain deklarasi array:

```
let mut alphabets = ["a", "b", "c", "d"];
let booleans = [true, false];
let floatingNumbers = [32.0000078, 3.14, 0.5];
```

O iya, penulisan literal array juga bisa dituliskan dalam multi-baris, contohnya seperti berikut:

```
let mut alphabets = [
    "a",
    "b",
    "c",
    "d"
];
let booleans = [
    true,
    false
];
```

◉ Pengaksesan elemen array

Pengaksesan elemen array dilakukan dengan menuliskan nama variabel array kemudian diikuti kurung siku yang mengapit angka indeks elemen yang

diinginkan.

Indeks array dimulai dari 0. Pada contoh di atas, `numbers` size-nya adalah 4 elemen, berarti elemen array ada pada indeks ke 0, 1, 2, 3.

```
let data0 = numbers[0];
println!("elemen array ke 0 {data0}");

let data1 = numbers[1];
println!("elemen array ke 1 {data1}");
```

Array hanya bisa diakses elemennya sesuai size saat deklarasi. Sebagai contoh, variabel `numbers` yang size nya 4, jika dipaksa mengakses elemen indeks ke-7 maka akan menghasilkan error.

◉ Mengubah isi elemen array

Array bisa diubah isi elemen-nya jika variabel tersebut adalah `mutable`. Pada contoh yang sudah dibuat, variabel `numbers` dideklarasikan mutable dengan tipe data elemen adalah numerik. Dengan ini kita bisa merubah value elemen array dengan value baru bertipe data sama.

Bisa dilihat pada contoh yang sudah dipraktekan, elemen indeks ke-1 diubah nilainya menjadi `16`, dan elemen indeks ke-3 value-nya menjadi `8`

```
numbers[1] = 16;
numbers[3] = 8;
```

Array hanya bisa diubah elemen-nya sesuai size saat deklarasi. Sebagai contoh, variabel `numbers` yang size nya 4, jika dipaksa mengakses dan/atau mengubah elemen indeks ke-7 maka akan menghasilkan error.

Selain itu, operasi assignment atau pengubahan nilai pada elemen array hanya bisa dilakukan dengan tipe data yang sama. Pada contoh di atas `numbers` adalah array bertipe numerik, karenanya tidak akan bisa diubah nilai elemen-nya dengan tipe lain, dan jika dipaksa akan menghasilkan error.

◉ Formatted print `{:?}`

Formatted print `{:?}` berguna untuk memformat macam-macam tipe data ke bentuk string, yang salah satunya adalah tipe data array. Dengan menggunakan formatted print ini kita bisa menampilkan nilai elemen array dalam bentuk string.

```
println!("array {:?}", numbers);
```

◉ Formatted print `{namaVariabel:?}`

Formatted print `{namavariabel:?}` kegunaannya sama seperti `{:?}`, dengan perbedaan: pada penggunaan formatted print `{namavariabel:?}` tidak perlu menuliskan variabel yang ingin di-format sebagai argumen. Cukup ganti `namavariabel` dengan nama variabel yang ingin di-format.

Silakan lihat contoh berikut. Keduanya adalah ekuivalen, menghasilkan nilai yang sama.

```
println!("array {:?}", numbers);
println!("array {numbers:?}");
```

A.13.2. Notasi penulisan tipe data

array

Pada contoh program yang sudah dipraktekan, variabel `number` tipe datanya adalah array numerik, terlihat saat deklarasinya *predefined value* diisi dengan literal elemen array bertipe `i32`.

Tipe data `i32` disini adalah milik elemen array, sedang array `numbers` itu sendiri tipe datanya adalah `[i32; 4]`, yang artinya adalah **sebuah array dengan elemen bertipe `i32` dengan size `4`**.

Jika kawan-kawan menggunakan ekstensi VSCode `rust-analyzer`, akan terlihat informasi tipe data array-nya.

```
src > main.rs
    ▶ Run | Debug | You, 20 minutes ago | 1 author (You)
1   fn main() {
2       let mut numbers: [i32; 4] = [24, 12, 32, 7];
3       println!("array {:?}", numbers);
4
5       let data0: i32 = numbers[0];
6       println!("elemen array ke 0 {data0}");
7
8       let data1: i32 = numbers[1];
9       println!("elemen array ke 1 {data1}");
10
11      numbers[1] = 16;
12      numbers[0] = 8;
13      println!("array {numbers:?}");
14  }
15
```

A.13.3. Macam-macam deklarasi array

Array lebih mudah dideklarasikan dengan metode *type inference*. Namun tak menutup kemungkinan ada kebutuhan dimana array harus dideklarasikan dengan menuliskan tipe datanya secara eksplisit. Berikut adalah macam-macam cara mendeklarasikan array.

◉ Deklarasi array dengan metode *type inference*

```
let angka_integer = [24, 12, 32, 7];
println!("{angka_integer:?}");
// output: [24, 12, 32, 7]

let angka_float = [24.2, 12.5, 32.00002, 7.2];
println!("{angka_float:?}");
// output: [24.2, 12.5, 32.00002, 7.2]
```

◉ Deklarasi array dengan metode *manifest typing* disertai *predefined value*

```
let data_boolean: [bool; 2] = [false, true];
println!("{data_boolean:?}");
// output: [false, true]

let angka_unsigned_integer: [u32; 3] = [24, 0, 12];
println!("{angka_unsigned_integer:?}");
// output: [24, 0, 12]
```

◉ Deklarasi array dengan notasi penulisan `[T; N]`

Pada contoh berikut, `data_numerik1` dideklarasikan bertipe array dengan tipe data data elemen adalah `i32`, mempunyai size `10`, dengan *predefined value* untuk setiap elemen array adalah angka `0`.

```
let data_numerik1: [i32; 10] = [0; 10];
println!("{data_numerik1:?}");
// output: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Pada contoh ke-dua ini, `data_numerik2` dideklarasikan dengan *predefined value* adalah array yang size-nya `5` dan nilai default tiap elemen adalah angka `4`.

```
let data_numerik2 = [4; 5];
println!("{data_numerik2:?}");
// output: [4, 4, 4, 4, 4]
```

Tambahan penjelasan mengenai notasi penulisan `[T; N]`:

- Jika digunakan pada penulisan tipe data array saat deklarasi, `T` adalah tipe data elemen, dan `N` adalah lebar/size array. Contoh: `let data_numerik1: [i32; 10]`.
- Jika digunakan pada penulisan *predefined value*, `T` adalah nilai setiap elemen array, dan `N` adalah lebar/size array. Contoh: `[4; 2]`, yang artinya semua elemen array diisi dengan angka `4`.

A.13.4. Melihat size array menggunakan method `len`

Array secara *default* properti tipe data **slice**, yang salah satunya adalah method bernama `len` yang berguna untuk melihat size dari sebuah array. Cara penggunaannya cukup dengan menuliskan `.len()` setelah variabel array. Contoh:

```
let names = ["jason", "grayon", "drake", "damian"];
let length = names.len();
println!("array size is {}", length);
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrograman
   Compiling array v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpemr
    Finished dev [unoptimized + debuginfo] target(s) in 0.
     Running `target\debug\array.exe`
array size is 4
```

> *Lebih jelasnya mengenai slice dibahas pada chapter Slice*

A.13.5. Iterasi array menggunakan `for in`

Pada contoh berikut, variabel `names` adalah array bertipe `[&str; 4]`, dideklarasikan dengan *predefined value*. Elemen array tersebut kemudian diiterasi menggunakan `for in` untuk di-print ke layar console value setiap elemennya.

```rust
let names: [&str; 4] = ["jason", "grayon", "drake", "damian"];
for name in names {
    println!("{name}");
}
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasar
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\perulangan for in.exe`

jason
grayon
drake
damian
```

Bisa juga dengan memanfaatkan *range syntax* dalam perulangan array. Benefitnya adalah pengaksesan indeks array lebih mudah.

```rust
let names: [&str; 4] = ["jason", "grayon", "drake", "damian"];
for i in 0..names.len() {
    println!("array index ke-{}: {}", i, names[i]);
}
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasar
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\array.exe`

array index ke-0: jason
array index ke-1: grayon
array index ke-2: drake
array index ke-3: damian
```

Statement `names.len()` mengembalikan informasi size sebuah array, nilainya bisa dimanfaatkan dalam perulangan `for in` untuk iterasi indeks ke-0 hingga terakhir.

A.13.6. Iterasi array menggunakan `while` dan `loop`

Contoh iterasi array menggunakan keyword `while`:

```
let names: [&str; 4] = ["jason", "grayon", "drake", "damian"];

let mut i = 0;
while i < names.len() {
    println!("array index ke-{}: {}", i, names[i]);
    i += 1;
}
```

Contoh iterasi array menggunakan keyword `loop`:

```
let names: [&str; 4] = ["jason", "grayon", "drake", "damian"];

let mut i = 0;
loop {
    if i >= names.len() {
        break;
    }

    println!("array index ke-{}: {}", i, names[i]);
    i += 1;
}
```

Iterasi array menggunakan `while` dan `loop` umumnya kurang praktis jika dibandingkan dengan `for in`. Tapi pastinya ada case dimana `while` dan/atau `loop` akan dibutuhkan.

A.13.7. Iterasi array menggunakan `for in` dan *tuple*

Mengiterasi value sekaligus *counter* perulangan sebenarnya bisa juga dilakukan menggunakan `for in`, contohnya seperti yang sudah kita praktekan di atas yaitu `for i in 0..names.len()`.

Ada juga bentuk lain pemanfaatan `for in` untuk mengiterasi sebuah array. Caranya dengan menggunakan teknik tuple untuk menampung data *counter* sekaligus value tiap elemen. Lebih jelasnya silakan lihat dan praktekan contoh berikut:

```
let names: [&str; 4] = ["jason", "grayon", "drake", "damian"];

for (i, name) in names.iter().enumerate() {
    println!("array index ke-{i}: {name}");
}
```

```
(base) PS D:\Labs\Vulam Studio\Ebook\dasarpemrogramanrust\dasar
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\array.exe`

array index ke-0: jason
array index ke-1: grayon
array index ke-2: drake
array index ke-3: damian
```

Variabel `names` yang notabene bertipe data `[&str; 4]` perlu dikonversi ke tipe `Iterator` terlebih dahulu caranya lewat pemanggilan method `.iter()`. Kemudian dari tipe tersebut perlu dikonversi lagi ke tipe `Enumerate` dengan cara memanggil method `.enumerate()`.

Setelah mendapatkan objek bertipe `Enumerate`, keyword `for in` digunakan untuk menampung tiap elemen array dalam bentuk *tuple* `(i, name)`. Variabel `i` disitu berisi counter iterasi, dan `name` adalah value-nya.

> - *Lebih jelasnya mengenai traits dibahas pada chapter Traits*
> - *Lebih jelasnya mengenai `Enumerate` dibahas pada chapter Trait → Iterator*
> - *Lebih jelasnya mengenai tuple dibahas pada chapter Tuple*

A.13.8. Append elemen ke array

Operasi menambahkan sebuah elemen ke array yang hasilnya melebihi kapasitas ... adalah tidak bisa. Karena array memiliki size fixed, tidak dinamis. Solusinya adalah menggunakan tipe data Vector. Nantinya array perlu dikonversi ke bentu Vector terlebih dahulu kemudian di-append, lebih jelasnya kita bahas pada chapter Vector.

A.13.9. Nested array

Data nested array bisa dibuat dengan level kedalaman tanpa batas, tetapi harus mengikuti aturan tipe data array yaitu: fixed size dan elemen bertipe data sejenis.

Pada contoh berikut variabel `data_arr` didefinisikan sebagai sebuah array bersarang atau nested dengan kedalaman 2 level.

```
let data_arr = [
    ["salad", "fried rice"],
```

Variabel `data_arr` pada contoh di atas bertipe data `[[&str; 2] 3]`, yang artinya adalah sebuah array dengan size 3, dengan isi elemen adalah juga array dengan size 2. Selalu ingat bahwa size array adalah fixed.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../array
```

◉ Chapter relevan lainnya

- Slice (Basic)
- Tipe Data → Vector
- Slice Memory Management

◉ Referensi

- https://doc.rust-lang.org/std/primitive.array.html
- https://doc.rust-lang.org/std/primitive.slice.html
- https://doc.rust-lang.org/std/iter/index.html
- https://doc.rust-lang.org/std/iter/struct.Enumerate.html
- https://doc.rust-lang.org/std/iter/trait.Iterator.html
- https://doc.rust-lang.org/rust-by-example/trait/iter.html
- https://doc.rust-lang.org/rust-by-example/flow_control/for.html

---

A.14. Slice (Basic)

Pada bab ini kita belajar tentang apa itu slice, apa perbedaan slice dan array, *slicing* atau *borrowing* pada slice, dan juga slice mutability.

> *Pada chapter ini kita akan bahas slice secara garis besar saja, tanpa menyinggung masalah memory management.*
>
> *Pembahasan lebih dalam perihal slice ada pada chapter terpisah (Memory Management → Slice), setelah kita mempelajari dasar-dasar memory management nantinya.*

A.14.1. Tipe data slice

Array adalah tipe data kolektif yang isinya bertipe sejenis. Contohnya `["a", "b", "c"]` adalah sebuah array dengan elemen bertipe string dan array tersebut memiliki size `3`.

Lalu apa itu slice? menurut laman dokumentasi Rust, slice adalah:

> *A slice is a dynamically sized type representing a 'view' into a sequence of elements of type T. The slice type is written as [T]*
>
> *A dynamically-sized view into a contiguous sequence, [T]. Contiguous here means that elements are laid out so that every element is the same distance from its neighbors.*

> *Slices are a view into a block of memory represented as a pointer and a size.*

Slice adalah representasi *block of memory* berbentuk pointer dan memiliki size yang dinamis (tidak fixed seperti array). Notasi tipe data slice adalah `&[T]` dimana `T` adalah tipe data element.

Slice bisa dibuat dari data array (atau dari tipe kolektif data lainnya) dengan menggunakan kombinasi operator `&` dan *range syntax* `..` dengan notasi penulisan seperti berikut:

```
let sliced_value1 = &data[start_index..end_index]
let sliced_value2 = &data[start_index..=end_index]
let sliced_value3 = &data[start_index..end_index]
...
```

> *Slice juga bisa dibuat dari tipe data vector. Lebih jelasnya dibahas pada chapter Vector.*

Silakan perhatikan contoh berikut dan praktekan:

```
let numbers = [12, 16, 8, 3];
println!("numbers   : {:?}, len: {}", numbers, numbers.len());
println!("numbers[0]: {:?}", numbers[0]);
println!("numbers[1]: {:?}", numbers[1]);

let slice_a = &numbers[0..3];
println!("slice_a   : {:?}, len: {}", slice_a, slice_a.len());
println!("slice_a[0]: {:?}", slice_a[0]);
println!("slice_a[1]: {:?}", slice_a[1]);
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\slice.exe`

numbers   : |12, 16, 8, 3|, len: 4
numbers[0]: 12
numbers[1]: 16

slice_a   : |12, 16, 8|, len: 3
slice_a[0]: 12
slice_a[1]: 16

slice_b   : |16, 8|, len: 2
slice_b[0]: 16
slice_b[1]: 8
```

Pada contoh di atas, variabel `numbers` didefinisikan bertipe `[i32; 4]`. Data pada variabel tersebut kemudian dipinjam ditampung pada variabel baru beranama `slice_a` yang merupakan slice bertipe `&[i32]`, dengan isi adalah element array `numbers` indeks ke `0` hingga sebelum `3` (yang berarti index ke-`2`). Dengan ini maka `slice_a` nilainya adalah `[12, 16, 8]` dengan size `3`.

Bisa dilihat pada statement print `slice_a[0]` dan `slice_a[1]`, nilai elemennya sesuai dengan dengan hasil peminjaman data array `numbers`.

> *Data yang tipenya `&[T]` biasa disebut sebagai shared slice atau cukup slice. Contohnya seperti `slice_a` dengan tipe data `&[i32]`.*

Slice `slice_a` bukan merupakan pemilik sebenarnya data `[12, 16, 8]`, slice tersebut hanya meminjam datanya dari `numbers` yang notabene dalah owner data `[12, 16, 8]`.

Meminjam disini artinya variabel baru `slice_a` memiliki data yang sama, dan alamat memori (atau pointer) data tersebut juga sama yaitu mengarah ke owner sebenarnya, yang pada contoh ini adalah `numbers`.

> *Di Rust, proses meminjam data secara umum disebut dengan* ***borrowing****. Variabel atau data hasil dari borrowing biasa disebut dengan* ***reference****. Lebih jelasnya akan dibahas pada chapter Pointer & References dan Borrowing.*
>
> *Variabel pemilik data yang sebenarnya disebut dengan* ***owner****. Lebih jelasnya mengenai ownership dibahas pada chapter Ownership*
>
> *Dalam konteks slice, proses meminjam data (yg menggunakan teknik borrowing dan range syntax) disebut dengan* ***slicing****.*

Dengan menggunakan VSCode `rust-analyzer` kita bisa melihat dengan mudah tipe sebuah slice.

```
You, 16 seconds ago | 1 author (You) | ▶ Run | Debug
fn main() {
    let numbers: [i32; 4] = [12, 16, 8, 4];
    println!("numbers   : {:?}, len: {}", numbers, numbers.len());
    println!("numbers[0]: {:?}", numbers[0]);
    println!("numbers[1]: {:?}", numbers[1]);

    let slice_a: &[i32] = &numbers[0..3];
    println!("slice_a   : {:?}, len: {}", slice_a, slice_a.len());
    println!("slice_a[0]: {:?}", slice_a[0]);
    println!("slice_a[1]: {:?}", slice_a[1]);

    let slice_b: &[i32] = &slice_a[1..-2];
    println!("slice_b   : {:?}, len: {}", slice_b, slice_b.len());
    println!("slice_b[0]: {:?}", slice_b[0]);
    println!("slice_b[1]: {:?}", slice_b[1]);
}
```

Kembali ke pembahasan pada contoh di atas. Selain `slice_a`, ada juga slice `slice_b` yang isinya merupakan data pinjaman dari `slice_a`. Statement `&slice_a[1..=2]` artinya adalah *borrowing* slice `slice_a` mulai dari indeks

ke-`1` hingga `2`. Dengan ini indeks ke-1 milik `slice_a` menjadi indeks ke-0 milik `slice_b`, ... dan seterusnya. Hasilnya, nilai `slice_b` adalah `[16, 8]` dengan size `2`.

Silakan juga cek penjelasan tambahan berikut, agar semakin paham tentang slice.

```
let numbers = [12, 16, 8, 3];
// variabel numbers isinya array [12, 16, 8, 3]

let slice_a = &numbers[0..3];
// meminjam data milik numbers elemen ke-0 hingga sebelum 3
(yaitu 2)
// hasilnya adalah [12, 16, 8]

let slice_b = &slice_a[1..=2];
// meminjam data milik slice_a elemen ke-1 hingga 2
// hasilnya adalah [16, 8]
```

A.14.2. Size slice

Gunakan method `len` untuk mencari tau size dari slice.

```
let numbers = [12, 16, 8, 3];
println!("{}", numbers.len());
// output: 4

let slice_a = &numbers[0..3];
println!("{}", slice_a.len());
// output: 3

let slice_b = &slice_a[1..=2];
```

A.14.3. Slice *range syntax*

Di bawah ini adalah range syntax yang bisa digunakan untuk slicing. Variabel `data` pada contoh berikut dijadikan sebagai bahan slicing.

```
let data = ["a", "b", "c", "d"];
```

- Notasi `&data[start_index..end_index]` untuk slicing data dari `start_index` hingga sebelum `end_index`

  ```
  let sliced_data = &data[1..3];
  println!("{:?}", sliced_data);
  // output => ["b", "c"]
  ```

- Notasi `&data[start_index..=end_index]` untuk slicing data dari `start_index` hingga `end_index`

  ```
  let sliced_data = &data[1..=3];
  println!("{:?}", sliced_data);
  // output => ["b", "c", "d"]
  ```

- Notasi `&data[..end_index]` untuk slicing data dari 0 hingga sebelum `end_index`

  ```
  let sliced_data = &data[..3];
  println!("{:?}", sliced_data);
  // output => ["a", "b", "c"]
  ```

- Notasi `&data[..=end_index]` untuk slicing data dari 0 hingga `end_index`

```
let sliced_data = &data[..=2];
println!("{:?}", sliced_data);
// output => ["a", "b", "c"]
```

- Notasi `&data[start_index..]` untuk slicing data dari `start_index` hingga indeks terakhir

```
let sliced_data = &data[1..];
println!("{:?}", sliced_data);
// output => ["b", "c", "d"]
```

- Notasi `&data[..]` untuk slicing semua elemen yang ada

```
let sliced_data = &data[..];
println!("{:?}", sliced_data);
// output => ["A", "b", "c", "d"]
```

> *Perihal apa itu borrowing dan kegunaan dari operator `&` yang ditulis di awal variabel dibahas lebih detail pada chapter Pointer & References.*

A.14.4. Mutability pada slice

Ada dua jenis data hasil operasi *borrowing* (atau biasa disebut dengan data *reference*).

- Read only atau shared reference, operator yang digunakan adalah `&`.

- Mutable reference, operator yang digunakan adalah `&mut`.

Kita akan bahas garis besarnya saja pada chapter ini. Intinya, shared reference adalah data hasil peminjaman/borrowing yang hanya bisa dibaca. Sedangkan mutable reference adalah data hasil borrowing yang bisa diubah nilainya, yang jika ini dilakukan maka akan mengubah juga data pemilik sebenarnya.

Contoh berikut adalah salah satu penerapan `&mut`. Variabel slice `numbers2` dipinjam beberapa elemennya dengan operator `&mut` ke variabel baru bernama `slice_e`. Dengan operator tersebut maka borrowing menghasilkan data *mutable reference*, data yang nilainya diperbolehkan untuk diubah meskipun data pinjaman.

```
let mut numbers2 = [12, 16, 8, 3];
println!("===== before =====");
println!("numbers2 : {:?}", numbers2);

let slice_e = &mut numbers2[..=2];
slice_e[1] = 99;

println!("===== after =====");
println!("slice_e  : {:?}", slice_e);
println!("numbers2 : {:?}", numbers2);
```

```
   Compiling slice v0.1.0 (D:\Labs\Adam Studio\Ubook\dasarpem
    Finished dev [unoptimized + debuginfo] target(s) in 0.40s
     Running `target\debug\slice.exe`
===== before =====
numbers2 : [12, 16, 8, 3]

===== after =====
slice_e  : [12, 99, 8]
numbers2 : [12, 99, 8, 3]
```

Bisa dilihat, indeks ke-1 `slice_e` diubah nilainya yang sebelumnya adalah `16` menjadi `99`. Setelah itu di-print, di output nilai variabel `numbers2` juga ikut berubah. Hal ini karena perubahan data pada variabel mutable reference juga punya pengaruh ke variabel pemilik data sebenarnya (yaitu `numbers2`).

O iya, penggunaan operator `&mut` mengharuskan kita untuk tidak menuliskan keyword `mut` pada variabel yang menampung nilai borrowing.

```
// statement yang direkomendasikan
let slice_e = &mut numbers2[..=2];

// statement yang TIDAK DIREKOMENDASIKAN dan akan memunculkan
warning.
let mut slice_e = &mut numbers2[..=2];
```

A.14.5. Perulangan `for in` pada slice

Slice merupakan tipe data yang implement trait Iterator (seperti array), dan semua data yang memiliki trait tersebut bisa digunakan pada perulangan. Slice `&[T]` jika dipergunakan dalam `for in`, tipe data penampung iterasi perulangan adalah `&T`, bukan `T` yaa jadi jangan sampai keliru.

```
let scores1 = [7, 8, 9];

for score in &scores1[..] {
    print!("{:?} ", score);
}
```

```
32    let scores1: [i32; 3] = [7, 8, 9];

34    for score: &i32 in &scores1[..] {
35        println!("{:?} ", score);
36    }
```

```
PROBLEMS    OUTPUT    GITLENS    JUPYTER    DEBUG CONSOLE    TERMINAL

(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasar
   Compiling slice v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpemr
    Finished dev [unoptimized + debuginfo] target(s) in 0.27s
     Running `target\debug\slice.exe`

7 8 9
```

> *Sangat dianjurkan menggunakan `rust-analyzer` jika menggunakan VSCode, agar lebih mudah melihat tipe data variabel dan informasi penting lainnya yang berguna dalam proses coding atau development.*
>
> - *Lebih jelasnya mengenai traits dibahas pada chapter Traits*
> - *Lebih jelasnya mengenai trait `Iterator` dibahas pada chapter Trait → Iterator*

A.14.6. Perulangan `for in` pada mutable slice

Sama seperti penggunaan `for in` pada shared slice, pada contoh kasus mutable slice perbedaannya hanyalah pada tipe data penampung iterasi yaitu `&mut T`, bukan `&T`.

Pada contoh berikut kita coba eksperimen mengubah nilai elemen sebuah mutable slice yang data aslinya juga hasil dari peminjaman mutable slice.

```rust
let mut scores2 = [7, 8, 9];
println!("(before) scores2 : {:?}", scores2);

let slice_f = &mut scores2[..];

for score in &mut slice_f[..] {
    *score += 1;
}

println!("(after)  scores2 : {:?}", scores2);
```

Variabel `scores` dipinjam menggunakan `&mut` ke variabel baru bernama `slice_f`. Kemudian `slice_f` dipinjam juga sebagai mutable slice pada perulangan `for in`.

Setelah di-increment nilainya menggunakan `*score += 1` bisa dilihat hasil akhirnya juga merubah nilai variabel `scores2` yang merupakan *owner* atau pemilik data sebenarnya.

```
40      let mut scores2: [i32; 3] = [7, 8, 9];
41      println!("(before) scores2 : {:?}", scores2);
42
43      let slice_f: &mut [i32] = &mut scores2[..];
44
45      for score: &mut i32 in &mut slice_f[..] {
46          *score += 1;
47      }
48
49      println!("(after)  scores2 : {:?}", scores2);
```

```
PROBLEMS    OUTPUT    CODELENS    JUPYTER    DEBUG CONSOLE    TERMINAL

(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrograman-rust\dasarp
   Compiling slice v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpemro
    Finished dev [unoptimized + debuginfo] target(s) in 0.43s
     Running `target\debug\slice.exe`
(before) scores2 : [7, 8, 9]
(after)  scores2 : [8, 9, 10]
```

Operasi increment pada `score` tidak bisa dituliskan dalam bentuk `score += 1` karena tipe data score adalah pointer mutable reference (ditandai dengan adanya `&mut`), untuk increment nilainya perlu di-*dereference* terlebih dahulu menggunakan operator `*`. Lebih jelasnya kita bahas pada chapter Pointer & References.

A.14.7. Append slice

Slice by default tidak mendukung operasi append. Sebenarnya bisa saja dilakukan tapi agak panjang caranya (silakan cari referensinya di Google untuk ini).

Perlu diketahui, jika kebutuhannya adalah untuk menampung jenis data yang size-nya bisa bertamah, penulis anjurkan untuk menggunakan Vector yang beberapa bab lagi akan kita bahas.

A.14.8. Memory management pada slice

Nantinya setelah selesai dengan pembahasan dasar memory management di Rust, kita akan bahas lagi topik slice tapi dari sudut pandang memory management pada chapter Memory Management → Slice.

A.14.9. Summary

Catatan ringkas perihal slice:

- Slice memiliki notasi `&[T]`
  - `&` disitu artinya adalah operasi borrowing/peminjaman
  - `T` adalah tipe data tiap elemen
- Slice bisa terpentuk dari hasil meminjam data array, vector, atau tipe data kolektif lainnya
- Data slice adalah selalu data pinjaman
- Slice memiliki lebar/size
- Slicing adalah cara pengaksesan slice menggunakan *range syntax*
- Slice bisa immutable, bisa juga mutable (menggunakan `&mut`)

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../slice
```

◉ Referensi

- https://doc.rust-lang.org/book/ch04-03-slices.html#the-slice-type
- https://doc.rust-lang.org/std/primitive.slice.html
- https://doc.rust-lang.org/std/iter/trait.Iterator.html
- https://doc.rust-lang.org/reference/types/slice.html
- https://doc.rust-lang.org/nomicon/references.html
- https://doc.rust-lang.org/rust-by-example/primitives/array.html

---

A.15. Tipe Data → Tuple

Pada chapter ini kita akan membahas tentang tipe data kolektif bernama Tuple.

A.15.1. Tipe data tuple

Tuple tipe data yang isinya koleksi dari banyak data atau value, yang bisa jadi tiap value tersebut tipe datanya berbeda satu sama lain. Tuple biasa digunakan untuk menampung data yang *heterogeneous* atau campuran.

> *Tipe ini tidak diciptakan sebagai pengganti array.*
>
> *Cukup mudah untuk mengidentifikasi kapan harus menggunakan tuple, contohnya: jika ada kebutuhan data harus bisa di-iterate, atau data memiliki pattern yg pasti seperti size-nya fixed, atau tipe datanya homogeneous atau sejenis, ... maka baiknya gunakan tipe data array atau slice.*

Tuple dibuat dengan notasi penulisan diapit tanpda `(` dan `)`. Contoh deklarasi tuple:

```
let tuple_a = ("jason", 27, ["racing", "working out"], true);
println!("tuple_a: {:?}", tuple_a);
```

Variabel `tuple_a` di atas bertipe data tuple, dengan tipe data spesifik per-elemennya bervariasi, ada string, numerik, array `[&str; 2]`, dan boolean.

Untuk menampilkan nilai per-elemen, gunakan notasi `.N` dimana `N` merupakan indeks elemen. Contohnya seperti berikut:

```
println!("index 0: {:?}", tuple_a.0);
println!("index 1: {:?}", tuple_a.1);
println!("index 2: {:?} {:?}", tuple_a.2[0], tuple_a.2[1]);
println!("index 3: {:?}", tuple_a.3);
```

```
▶ Run | Debug
fn main() {
    let tuple_a: (&str, i32, [&str; 2], bool) = ("jason", 27, ["racing", "working out"], true);
    println!("tuple a: {:?}", tuple_a);

    println!("index 0: {:?}", tuple_a.0);
    println!("index 1: {:?}", tuple_a.1);
    println!("index 2: {:?} {:?}", tuple_a.2[0], tuple_a.2[1]);
    println!("index 3: {:?}", tuple_a.3);

PROBLEMS   OUTPUT   GITLENS   JUPYTER   DEBUG CONSOLE   TERMINAL

(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrograman-rust\dasarpemrograman-rust\hands-on\tuples> cargo run
   Compiling tuple v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpemrograman-rust\dasarpemrograman-rust\hands-on\tuple)
    Finished dev [unoptimized + debuginfo] target(s) in 0.46s
     Running `target\debug\tuple.exe`

tuple_a: ("jason", 27, ["racing", "working out"], true)

index 0: "jason"
index 1: 27
index 2: "racing" "working out"
index 3: true
```

Data pada `tuple_a` indeks ke-2 bertipe slice, untuk mengakses tiap elemennya bisa menggunakan notasi pengaksesan indeks array/slice seperti biasa.

- Tuple `tuple_a.2` nilainya `["racing", "working out"]`
- Tuple `tuple_a.2[0]` nilainya `"racing"`
- Tuple `tuple_a.2[1]` nilainya `"working out"`

A.15.2. Mutable tuple

Cara untuk membuat tuple menjadi data yang mutable adalah dengan menambahkan `mut` pada saat deklarasi. Pada contoh berikut variabel `tuple_b` dideklarasikan sebagai mutable dengan teknik *manifest typing*.

```
let mut tuple_b: (&str, i32, [&str; 2], bool) = ("default", 0,
[""; 2], false);
tuple_b.0 = "damian";
tuple_b.1 = 18;
tuple_b.2 = ["gaming", "adventuring"];
tuple_b.3 = true;

println!("tuple_b: {:?}", tuple_b);
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarp
   Compiling tuple v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpemro
    Finished dev [unoptimized + debuginfo] target(s) in 0.54s
     Running `target\debug\tuple.exe`

tuple_b: ("damian", 18, ["gaming", "adventuring"], true)
```

A.15.3. Notasi deklarasi tuple

Ada beberapa cara yang bisa digunakan dalam deklarasi tuple.

◉ Type inference

Cara pertama menggunakan teknik *type inference* seperti yang sudah dipraktekan pada contoh pertama.

```
let tuple_a = ("jason", 27, ["racing", "working out"], true);
```

◉ Manifest typing

Bisa juga menggunakan teknik deklarasi *manifest typing*, biasanya diterapkan pada pembuatan mutable tuple dengan *predefined value* adalah nilai kosong seperti empty string, 0, dan lainnya.

```
let tuple_b: (&str, i32, [&str; 2], bool) = ("damian", 18,
["gaming", "adventuring"], true);
```

Contoh lain:

```
let mut tuple_b: (&str, i32, [&str; 2], bool) = ("default", 0,
[""; 2], false);
tuple_b.0 = "damian";
tuple_b.1 = 18;
tuple_b.2 = ["gaming", "adventuring"];
tuple_b.3 = true;
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrograman rust\dasar
   Compiling tuple v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpemr
    Finished dev [unoptimized + debuginfo] target(s) in 0.44s
     Running `target\debug\tuple.exe`

name   : "grayson"
age    : 29
hobbies : |"sleeping", "parkour"|
is_male : true
```

◉ Packing tuple

Adalah cara pembuatan tuple yang dimana nilai elemenya bersumber dari

variabel lain.

```
let name = "grayson";
let age = 29;
let hobbies = ["sleeping", "parkour"];

let tuple_c = (name, age, hobbies);

println!("name    : {:?}", tuple_c.0);
println!("age     : {:?}", tuple_c.1);
println!("hobbies : {:?}", tuple_c.2);
```

> *Istilah packing tuple bukan resmi dari official Rust. Istilah ini penulis buat sendiri.*

◉ Unpacking tuple

Unpacking tuple adalah kebalikan dari packing tuple. Data tuple di-distribusikan ke banyak variabel dalam 1 baris deklarasi.

```
let tuple_d = ("stephanie", 28, ["software engineering"], false);
let (name, age, hobbies, is_male) = tuple_d;

println!("name    : {:?}", name);
println!("age     : {:?}", age);
println!("hobbies : {:?}", hobbies);
println!("is_male : {:?}", is_male);
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarp
   Compiling tuple v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpemrog
    Finished dev [unoptimized + debuginfo] target(s) in 0.4s
     Running `target\debug\tuple.exe`

name    : "stephanie"
age     : 28
hobbies : ["software engineering"]
is male : false
```

> *Pada bahasa lain teknik ini biasa disebut dengan destructuring assignment*

A.15.4. Tuple `()`

Tuple bisa didefinisikan dengan tanpa isi dengan menggunakan `()`.

```
let tuple_d = ();
println!("{:?}", tuple_d);
```

A.15.5. Tuple Structs

Pembahasan mengenai tuple structs dibahas pada chapter Struct.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../tuple
```

◉ Referensi

- https://doc.rust-lang.org/std/primitive.tuple.html
- https://doc.rust-lang.org/rust-by-example/primitives/tuples.html

---

A.16. Tipe Data → Vector

Pada chapter ini kita akan belajar tetang tipe data *Vector*. Vector adalah tipe data seperti array tapi dinamis. Dinamis disini artinya bisa bertambah dan berkurang kapanpun sesuai kebutuhan.

Vector sangat mirip seperti array, yang karakteristiknya adalah tipe data elemen wajib sama, punya informasi size, elemen-nya bisa diakses atau diubah. Salah satu perbedaan vector dibanding array adalah jumlah elemen pada vector bisa bertambah lebih dari kapasitas yang sudah ditentukan.

Vector memiliki 3 buah atribut yg penting untuk diketahui:

- pointer ke data asli
- lebar atau size
- kapasitas (representasi dari seberapa banyak memori di-booking untuk data vector tersebut)

Vector bisa bertambah jumlah isinya selama size dibawah kapasitas yang sudah dialokasikan. Jika suatu ketika vector isinya bertambah lebih banyak dari jumlah alokasi maksimal kapasitas, maka vector akan dialokasikan ulang dengan kapasitas yang lebih besar.

A.16.1. Tipe data `Vec<T>`

`Vec<T>` adalah tipe data yang merepresentasikan vector, dimana `T` adalah

generics. Vector datanya dialokasikan di heap memory.

> - *Lebih jelasnya mengenai generic dibahas pada chapter Generics*
> - *Lebih jelasnya mengenai heap dibahas pada chapter Basic Memory Management*

Langsung saja kita praktekan.

◉ Deklarasi vektor, size, dan capacity

Ada beberapa cara yang bisa dipakai untuk membuat data vector. Salah satunya adalah menggunakan macro `vec`, penulisannya seperti pembuatan array hanya saja perlu ditambahi prefix `vec!`. Contoh:

```
let mut data_one = vec!["batman", "superman", "lobo"];

println!("data: {:?}", data_one);
println!("length: {}, capacity: {}", data_one.len(),
data_one.capacity());
```

```
   Compiling vector v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpemr
    Finished dev [unoptimized + debuginfo] target(s) in 0.44s
     Running `target\debug\vector.exe`
data: ["batman", "superman", "lobo"]
length: 3, capacity: 3
```

Pada contoh di atas variabel `data_one` adalah sebuah vector yang isinya 3 elemen, yaitu literal string `batman`, `superman`, dan `lobo`. Vector `data_one` dideklarasikan menggunakan macro `vec` kemudiaan diikuti notasi penulisan yang mirip seperti array.

> *Lebih jelasnya mengenai macro dibahas pada chapter Macro*

Pembuatan vector menggunakan teknik ini hasilnya adalah data vector yang `size` dan `capacity` nya adalah sesuai dengan *predefined value*, yang pada konteks ini adalah 3.

Umumnya vector dideklarasikan dengan keyword `mut` agar bisa diubah nilainya, karena tujuan disediakannya tipe data ini adalah untuk bisa mengakomodir tipe data array tetapi dinamis (bisa manipulasi datanya).

Perihal method `len` dan `capacity`:

- Method `len` digunakan untuk mencari tahu size atau jumlah elemen yang ada pada sebuah vector
- Method `capacity` digunakan untuk mencari tahu kapasitas atau jumlah maksimum elemen yang ada pada sebuah vector

> *0 iya, kode program di atas akan menghasilkan warning saat di run. Hal ini dikarenakan variabel `data_one` yang didefinisikan mutable belum diubah nilainya. Hiraukan saja, karena selanjutnya kita akan manipulasi data variabel tersebut.*

◉ Method `pop` → menghapus elemen terakhir

Oke, selanjutnya mari kita oprek variabel `data_one` yang sudah ditulis.

Tipe data `Vec<T>` memiliki method `pop` yang fungsinya menghapus data elemen terakhir. Mari gunakan method ini pada `data_one`.

```
data_one.pop();

println!("data: {:?}", data_one);
println!("length: {}, capacity: {}", data_one.len(),
data_one.capacity());
```

```
   Compiling vector v0.1.0 (D:\Labs\Vdam Studio\Ebook\dasarpem
    Finished dev [unoptimized + debuginfo] target(s) in 0.49s
     Running `target\debug\vector.exe`

data: ["batman", "superman"]
length: 2, capacity: 3
```

Bisa dilihat pada contoh di atas, setelah menggunakan method `pop`, isi `data_one` menjadi 2 elemen saja. Elemen terakhir (yaitu string `lobo`) dihapus. Efeknya, atribut size menjadi 2, tapi kapasitas tetap 3.

◉ Method `remove` → menghapus elemen index ke `I`

Method `remove` adalah salah satu method lainnya yang ada pada tipe data `Vec<T>`. Kegunaan dari method `remove` adalah untuk menghapus elemen pada indeks tertentu.

```
data_one.remove(1);

println!("data: {:?}", data_one);
println!("length: {}, capacity: {}", data_one.len(),
data_one.capacity());
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarp
   Compiling vector v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpemr
    Finished dev [unoptimized + debuginfo] target(s) in 0.19s
     Running `target\debug\vector.exe`

data: ["batman"]
length: 1, capacity: 3
```

Dicontohkan di atas elemen indeks ke-`1` dihapus dengan cara memanggil method `remove` lalu menyisipkan indeks elemen sebagai parameter pemanggilan method. Hasilnya elemen `superman` dihapus dari vector.

◉ Method `push` → menambahkan elemen baru

Sekarang isi dari vector `data_one` tinggal 1 elemen, mari kita tambahkan 3 elemen baru dengan memanfaatkan method `push`.

Method `push` fungsinya adalah untuk menambahkan elemen baru pada vector.

```
data_one.push("constantine");
data_one.push("trigon");
data_one.push("darkseid");

println!("data: {:?}", data_one);
println!("length: {}, capacity: {}", data_one.len(),
data_one.capacity());
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarp
   Compiling vector v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpemr
    Finished dev [unoptimized + debuginfo] target(s) in 0.44s
     Running `target\debug\vector.exe`

data: ["batman", "constantine", "trigon", "darkseid"]
length: 4, capacity: 6
```

Bisa dilihat sekarang `data_one` isinya adalah 4 elemen dan atribut size-nya cocok, yaitu `4`. Namun ada yang aneh, kenapa kapasitas jadi `6` padahal di awal kapasitas adalah 3.

◉ Realokasi vector

Perubahan kapasitas atau realokasi vector terjadi ketika sebuah vector isinya bertambah lebih banyak dari jumlah alokasi maksimal kapasitas.

Lalu apa efeknya? secara high level bisa dibilang tidak ada, namun kalau dibahas lebih rinci, efeknya adalah di sisi alokasi space untuk menampung elemen. Terjadi proses realokasi dimana vector yang baru akan memiliki kapasitas lebih besar.

◉ Mengubah value sebuah elemen menggunakan notasi `[i]`

Sama seperti array, vector juga bisa dimodifikasi nilai elemennya dengan menggunakan notasi `[i]`.

Pada contoh berikut elemen indeks ke-`2` diubah nilainya dari `trigon` ke `red hood`.

```
data_one[2] = "red hood";

println!("data: {:?}", data_one);
println!("length: {}, capacity: {}", data_one.len(),
data_one.capacity());
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrograman rust\dasarp
   Compiling vector v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpem
    Finished dev [unoptimized + debuginfo] target(s) in 0.49s
     Running `target\debug\vector.exe`

data: |"batman", "constantine", "red hood", "darkseid"|
length: 4, capacity: 6
```

◉ Method `is_empty` → mengecek apakah vector kosong

Method `is_empty` digunakan untuk mengidentifikasi apakah sebuah vector isinya kosong atau tidak.

```
let is_vector_empty = data_one.is_empty();
println!("result: {:?}", is_vector_empty);
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrograman rust\dasarp
   Compiling vector v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpem
    Finished dev [unoptimized + debuginfo] target(s) in 0.49s
     Running `target\debug\vector.exe`

result: false
```

◉ Method `clear` → mengosongkan isi vector

Method `clear` digunakan untuk mengosongkan sebuah vektor.

```
data_one.clear();
println!("data: {:?}", data_one);
println!("length: {}, capacity: {}", data_one.len(),
data_one.capacity());
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.49s
     Running `target\debug\vector.exe`

data: []
length: 0, capacity: 6
```

◉ Method `append` → concatenation/ penggabungan vector

Method `append` digunakan untuk menggabungkan dua buah vector. Penggunaannya cukup mudah, panggil saja method nya lalu sisipkan *mutable reference* dari vector satunya.

```rust
let mut result_one = vec![3, 1, 2];

let mut data_two = vec![7, 6, 8];
result_one.append(&mut data_two);

println!("data: {:?}", result_one);
println!("length: {}, capacity: {}", result_one.len(),
result_one.capacity());
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarp
   Compiling vector v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpem
    Finished dev [unoptimized + debuginfo] target(s) in 0.49s
     Running `target\debug\vector.exe`

data: [3, 1, 2, 7, 6, 8]
length: 6, capacity: 6
```

Pada contoh di atas `result_one` adalah vector mutable dengan isi 3 elemen. Kemudian dikelarasikan `data_two` yang isinya juga vector 3 elemen. Vector `data_two` dimasukan kedalam vector `result_one` dengan menggunakan method `append`, dengan ini maka isi `result_one` adalah gabungan dari `result_one` yang lama dan `data_two`.

Proses append vector mengharuskan parameter method `append` diisi dengan *mutable reference* dari vector yang ingin dimasukan. Cara untuk mengambil *mutable reference* adalah dengan menggunakan keyword `&mut`.

Ok, selanjutnya tambahkan lagi isi `result_one` dengan vector lain.

```
result_one.append(&mut vec![4, 5]);

println!("data: {:?}", result_one);
println!("length: {}, capacity: {}", result_one.len(),
result_one.capacity());
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrograman rust\dasarp
   Compiling vector v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpem
    Finished dev [unoptimized + debuginfo] target(s) in 0.49s
     Running `target\debug\vector.exe`

data: [3, 1, 2, 7, 6, 8, 4, 5]
length: 8, capacity: 12
```

Proses penggabungan vector pada contoh ke-dua di atas sedikit berbeda. Method `append` parameternya adalah langsung *mutable reference* dari literal vector. Ini merupakan salah satu cara yang bisa digunakan dalam penggabungan vector.

◉ Method `sort` → untuk mengurutkan vector

Method `sort` digunakan untuk mengurutkan elemen vector.

```
println!("data: {:?}", result_one);
result_one.sort();
println!("data: {:?}", result_one);
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasar
   Compiling vector v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpem
    Finished dev [unoptimized + debuginfo] target(s) in 0.49s
     Running `target\debug\vector.exe`

data: |3, 1, 2, 7, 6, 8, 4, 5|
data: [1, 2, 3, 4, 5, 6, 7, 8]
```

A.16.2. Macam deklarasi vektor

Ada beberapa cara deklarasi vector. Pada contoh berikut dua buah vector dideklarasikan menggunakan macro `vec!`.

```
let mut vector_4 = vec![1, 2, 3];
let mut vector_5: Vec<i64> = vec![1, 2, 3];
```

Vector `vector_4` didefinisikan dengan cara yang sudah kita terapkan sebelumnya, yaitu menggunakan macro `vec`. Vector `vector_5` juga didefinisikan dengan cara yang sama, hanya saja pada vector ini tipe datanya ditentukan secara eksplisit yaitu `Vec<164>`.

`Vec<i64>` Artinya adalah vector dengan tipe data elemen adalah `i64`. Dengan notasi penulisan `Vec<T>` bisa ditentukan tipe data elemen yang diinginkan.

Cara deklarasi vector selanjutnya adalah pembuatan vector dengan isi kosong. Deklarasi vector ini mewajibkan tipe data vector dituliskan secara eksplisit, dikarenakan tipe data tidak bisa diidentifikasi dari isinya (karena isinya kosong). Contoh:

```
let mut vector_7: Vec<&str> = vec![];
let mut vector_8: Vec<&str> = Vec::new();
```

Vector `vector_7` dan `vector_8` keduanya bertipe vector dengan isi atau elemen bertipe data literal string `&str`.

Deklarasi vector kosong bisa dilakukan dengan macro `vec` yang ditulis tanpa isi, atau bisa menggunakan `vec::new()`.

A.16.3. Iterasi data vector

Keyword `for in` bisa digunakan untuk iterasi vector. Cara penerapannya seperti pada array atau slice.

```
let vec_eight = vec![1, 2, 3];
for e in vec_eight {
    print!("{e} ");
}

// 1 2 3
```

```
let vec_nine = vec![1, 2, 3];
for i in 0..vec_nine.len() {
    print!("{} ", vec_nine[i]);
}

// 1 2 3
```

Keyword perulangan lainnya juga bisa digunakan.

A.16.4. Ownership tipe data vector

Salah satu atribut vector yang penting untuk diketahui adalah, pemilik data

sebenarnya (atau owner). Agar lebih jelas, silakan coba terlebih dahulu kode berikut.

```
let vec_ten = vec![1, 2, 3];
for e in vec_ten {
    print!("{e} ");
}
for i in 0..vec_ten.len() {
    print!("{} ", vec_ten[i]);
}
```

```
65
66      let vec_ten: Vec<i32> = vec![1, 2, 3];
67      for e: i32 in vec_ten {
68          print!("{e} ");
69      }
70      for i: usize in 0..vec_ten.len() {
71          print!("[] ", vec_ten[i]);
72      }
73
```

```
PROBLEMS 1   OUTPUT   CODELENS   JUPYTER   DEBUG CONSOLE   TERMINAL

(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarpemrogramanrust\examples\v
   Compiling vector v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarpemrogra

error[E0382]: borrow of moved value: `vec_ten`
   --> src\main.rs:70:17
67 |     for e in vec_ten {
   |              ------- `vec_ten` moved due to this implicit call to `.into_iter()`
...
70 |     for i in 0..vec_ten.len() {
   |                 ^^^^^^^^^^^^ value borrowed here after move
   |
note: this function takes ownership of the receiver `self`, which moves `vec_ten`
   --> C:\Users\cakno\.rustup\toolchains\stable-x86_64-pc-windows-msvc\lib/rustlib/src/r
   |
261 |     fn into_iter(self) -> Self::IntoIter;
   |                  ^^^^
help: consider iterating over a slice of the `Vec<i32>`'s content to avoid moving into t
   |
67 |     for e in &vec_ten {
   |              +

For more information about this error, try `rustc --explain E0382`.
error: could not compile `vector` due to previous error
```

Terlihat sekilas tidak ada kode yang bermasalah dari program di atas, tapi error, aneh.

Di Rust, ownership atau kepemilikan data adalah hal yang sangat penting. Saking pentingnya, beberapa orang menyebut Rust sebagai bahasa yang *value oriented*.

Dalam kasus kode program vector di atas, ketika keyword `for in` digunakan untuk mengiterasi vector `vec_ten`, membuat pemilik data vektor berpindah ke variabel `e`. Hal ini efeknya adalah ketika kita berusaha mengakses variabel yang sama setelah perulangan selesai, maka yang muncul adalah error, karena value-nya sudah berpindah.

> *Perpindahan owner disebut dengan move semantics. Lebih jelasnya nanti dibahas pada chapter Ownership*

Solusi untuk antisipasi error ini adalah dengan cara meminjam value yang sebenarnya dari owner, untuk kemudian digunakan dalam perulangan. Caranya dengan menggunakan teknik *borrowing* menggunakan operator `reference` yaitu `&`. Data sebenarnya milik owner dipinjam untuk dipergunakan di perulangan.

> - *Lebih jelasnya mengenai reference dibahas pada chapter Pointer & References*
> - *Lebih jelasnya mengenai borrowing dibahas pada chapter Borrowing*

Silakan ubah kode yang sebelumnya ...

```
for e in vec_ten {
    print!("{e} ");
}
```

... menjadi seperti ini, kemudian run, maka error akan hilang.

```
for e in &vec_ten {
    print!("{e} ");
```

Salah satu alternatif cara lainnya untuk antisipasi value berpindah tempat adalah dengan menggunakan method `iter` untuk mengkonversi vector menjadi iterator. Jadi yang di-iterasi bukan vector-nya, melainkan objek iterator yang dibuat dari vector tersebut.

```
for e in vec_ten.iter() {
    print!("{e} ");
}
```

> - *Lebih jelasnya mengenai ownership dibahas pada chapter Ownership*
> - *Lebih jelasnya mengenai borrowing dibahas pada chapter Borrowing*

A.16.5. Vector slice

Seperti array, slice juga bisa dibuat dari vector. Cara penerapannya juga sama persis. Contoh:

```
let vec_population = vec![2, 1, 3];
let vec_sample = &vec_population[0..1];
println!("{:?}", vec_sample); // [2]
```

Semua operasi slice bisa diterapkan di vector.

A.16.6. Tipe data `VecDeque<T>`

Tipe data `VecDeque<T>` adalah sama seperti `Vec<T>` plus mendukung operasi menambah dan mengurangi elemen dari dua sisi secara efisien.

Pada tipe data `Vec<T>`, ada method `pop` yang fungsinya menghapus data elemen terakhir dan method `push` untuk menambah elemen baru dari kanan. Tipe data `VecDeque` memiliki bebebrapa method tambahan, yaitu:

- method `pop_front` untuk hapus data elemen pertama atau paling kiri (indeks ke-0)
- method `push_front` untuk menambah data dari kiri (indeks ke-0)
- method `pop_back` untuk hapus data elemen pertama atau paling kanan (indeks terakhir)
- method `push_back` untuk menambah data dari kanan (indeks terakhir)

Contoh penerapan:

```
use std::collections::VecDeque;

let mut vec_10 = VecDeque::from(vec!["a", "b", "c"]);

vec_10.pop_front();
vec_10.push_front("z");
println!("data: {:?}", vec_10);

vec_10.pop_back();
vec_10.push_back("h");
println!("data: {:?}", vec_10);
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrograman
   Compiling vector v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpemr
    Finished dev [unoptimized + debuginfo] target(s) in 0.49s
     Running `target\debug\vector.exe`
data: ["z", "b", "c"]
data: ["z", "b", "h"]
```

Tipe data `VecDeque<T>` tidak otomatis di-import. Kita perlu mengimport path dimana tipe data itu berada menggunakan keyword `use`.

```
use std::collections::VecDeque;
```

Cara membuat vector `VecDeque<T>` bisa menggunakan `VecDeque::from` dengan parameter diisi data vectornya, seperti pada kode program yang sudah ditulis.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../vector
```

◉ Referensi

- https://doc.rust-lang.org/std/macro.vec.html
- https://doc.rust-lang.org/std/vec/struct.Vec.html
- https://doc.rust-lang.org/rust-by-example/std/vec.html
- https://doc.rust-lang.org/std/collections/struct.VecDeque.html
- https://stackoverflow.com/questions/36672845/in-rust-is-a-vector-an-iterator
- https://stackoverflow.com/questions/28800121/what-do-i-have-to-do-to-solve-a-use-of-moved-value-error
- https://stackoverflow.com/questions/43036279/what-does-it-mean-to-pass-in-a-vector-into-a-for-loop-versus-a-reference-to-a

---

A.17. Function

Pada chapter ini kita akan belajar tentang *function* atau fungsi.

> *Definisi fungsi dalam programming secara terminologi adalah sebuah modul atau sub-program kecil yang digunakan untuk mengeksekusi sebuah perintah, dan bisa di-reuse dalam penggunaannya.*

A.17.1. Keyword `fn`

Fungsi di Rust dibuat menggunakan keyword `fn`. Salah satu contoh penerapan keyword ini dalam pembuatan fungsi sudah kita praktekan berulang kali pada definisi fungsi `main`, yang merupakan fungsi utama atau entrypoint sebuah program.

```
fn main() {

}
```

Fungsi `main` ini spesial, ketika program dijalankan maka `main` otomatis dipanggil atau dieksekusi.

Pembuatan fungsi lainnya-pun juga didefinisikan dengan notasi penulisan yang sama. Pada contoh berikut kita akan buat fungsi bernama `greet` yang tugasnya adalah menampilkan message ke layar output.

```rust
fn greet() {
    println!("hello world");
}
```

Panggil fungsi tersebut di `main`, lalu run program.

```rust
fn main() {
    println!("hello rust");
    greet();
}
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.54s
     Running `target\debug\function.exe`
hello rust
hello world
```

Bisa dilihat ada 2 message muncul, yang pertama adalah pesan `hello rust` yang muncul hasil eksekusi statement `println!("hello rust")`, lalu diikuti `hello world` yang statement print untuk menampilkan message tersebut ada dalam fungsi `greet`.

O iya, definisi fungsi tidak mengenal urutan ya. Sebagai contoh fungsi `greet` dan `main` di atas, bebas mana yang ditulis lebih dulu.

```rust
fn greet() {
    // ..
}

fn main() {
    // ..
}
```

... atau ...

```
fn main() {
    // ..
}

fn greet() {
    // ..
}
```

A.17.2. *Naming convention* fungsi

Sesuai anjuran di halaman dokumentasi Rust, aturan penulisan nama fungsi di Rust adalah menggunakan **snake case**. Nama fungsi dituliskan dalam huruf kecil dengan separator tanda garis bawah atau underscore (`_`). Contohnya seperti: `main`, `greet_message`, `get_report_status`, dll.

A.17.3. Parameter fungsi dan argumen fungsi

Fungsi bisa didefinisikan dengan disertai parameter, dengan itu bisa menyisipkan data saat pemanggilan fungsi.

Parameter dituliskan di statement pendefinisian fungsi dengan notasi seperti berikut:

```
fn func_name(param_a: int32) {
    // ...
}
```

Kembali ke praktek, pada bagian ini kita akan buat sebuah fungsi bernama `greet_custom_message`. Fungsi ini tugasnya masih sama seperti seperti `greet` hanya saja pesan yang ditampilkan adalah sesuai dengan value parameter fungsi.

```
fn greet_custom_message(name: &str, message: &str) {
    println!("hi {name}, {message}");
}
```

Ada dua parameter yang didefinisikan:

- `name`, tipe datanya string literal
- `message`, tipe datanya juga string literal

Kedua nilai parameter tersebut di-*combine* lalu di-print menggunakan macro `println`.

Ok, sekarang panggil fungsi tersebut dalam `main`, sisipkan argumen pertama `Damian` yang ditampung parameter `name`, dan argumen ke-2 `welcome to the castle!` untuk parameter `message`

```
fn main() {
    greet_custom_message("Damian", "welcome to the castle!");
}
```

Jalankan program. Output bisa dilihat di gambar berikut.

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.55s
     Running `target\debug\function.exe`
hi Damian, welcome to the castle!
```

O iya, parameter bisa memiliki tipe data apa saja selama di-*support* oleh Rust,

jadi tidak hanya string literal `&str`.

> *Secara terminologi, parameter adalah variabel yang didefinisikan di fungsi, sedangkan argumen adalah nilai yang disisipkan pada parameter saat pemanggilan fungsi.*
>
> *Seiring berjalannya waktu, terjadi sedikit perubahan mengenai makna untuk istilah `parameter` dan `argumen`. Sekarang kedua istilah tersebut lebih sering dimaknai sama.*

A.17.4. Nilai balik fungsi (*return value*)

Sebuah fungsi bisa di-desain memiliki nilai balik. Caranya dengan menambahkan tanda `->` diikuti tipe data dari nilai balik pada pendefinisian fungsi.

Pada contoh berikut, sebuah fungsi dibuat dengan nama `calculate_box_volume1`, memiliki 3 buah parameter bertipe numerik (`width`, `height`, dan `length`), dan nilai balik bertipe data `i32`. Tugas dari fungsi `calculate_box_volume1` adalah melakukan operasi matematika perhitungan volume terhadap data yang didapat dari parameter, yang kemudian hasil kalkulasinya dijadikan nilai balik.

Keyword `return` (diikuti dengan data) digunakan untuk menentukan nilai balik suatu fungsi. Pada contoh berikut, variabel `volume` adalah nilai balik fungsi.

```
fn calculate_box_volume1(width: i32, height: i32, length: i32) ->
i32 {
```

Selanjutnya siapkan beberapa variabel yang nantinya digunakan saat pemanggilan `calculate_box_volume1`. Lalu panggil fungsi tersebut kemudian tampilkan *return value*-nya.

```
let width = 5;
let height = 8;
let length = 12;

let res1 = calculate_box_volume1(width, height, length);
println!("result: {res1}");
```

Jalankan program, hasilnya adalah sesuai harapan. Volume dari persegi panjang dengan data `5 x 8 x 12` adalah `480`.

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.42s
     Running `target\debug\function.exe`
result: 480
```

◉ Keyword `return` untuk penentuan nilai balik

Salah satu hal yang unik yang ada di Rust adalah beberapa variasi cara penulisan nilai balik fungsi. Cara pertama adalah menggunakan keyword `return` seperti yang sudah dipraktekan.

```
fn calculate_box_volume1(width: i32, height: i32, length: i32) ->
i32 {
    let volume = width * height * length;
    return volume;
}
```

Untuk cara ini penulis rasa cukup jelas.

◉ *Return value* tanpa keyword `return`

Cara ke-2 kita langsung praktekan menggunakan kode berikut, silakan tulis lalu jalankan program:

```
fn calculate_box_volume2(width: i32, height: i32, length: i32) ->
i32 {
    let volume = width * height * length;
    volume
}

fn main() {
    let name2 = "Damian";
    let res2 = calculate_box_volume2(width, height, length);
    println!("hi {name2}, the box volume is {res2}");
}
```

Hasilnya kurang lebih sama, hanya di-bagian outputnya beda karena pada contoh ini pesan yang ditampilkan diubah.

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.485
     Running `target\debug\function.exe`
hi Damian, the box volume is 480
```

Sekarang kita fokus ke kode yang sudah ditulis. Fungsi `calculate_box_volume2` kurang lebih struktur dan isinya sama seperti `calculate_box_volume1`, bedanya hanya pada dua hal:

- Fungsi `calculate_box_volume2` nilai baliknya dituliskan tanpa keyword `return`
- Fungsi `calculate_box_volume2` pada statement terakhir tidak diakhir tanda akhir statement `;`

Kedua poin tersebut merupakan syarat untuk penerapan *return value* tanpa keyword `return`. Caranya kurang lebih adalah dengan menuliskan data nilai balik dengan tanpa keyword `return` dan di akhir statement tidak ditambahi tanda semicolon `;`.

> *Statement terakhir sebuah blok kode fungsi yang ditulis tanpa semicolon `;` disebut sebagai `tail` atau `body tail`.*

◉ Statemen sebagai *return value* tanpa keyword `return`

Cara selanjutnya yang akan kita bahas ini sebenarnya sama seperti sebelumnya, hanya saja bedanya disini tidak menggunakan nama variabel sebagai nilai balik, melainkan langsung statement, yang hasil eksekusi statement tersebut dijadikan nilai balik fungsi.

Langsung saja kita praktekan. Siapkan fungsi `calculate_box_volume3` berikut:

```
fn calculate_box_volume3(width: i32, height: i32, length: i32) ->
i32 {
    width * height * length
}

fn greet_custom_message(name: &str, message: &str) {
    println!("hi {name}, {message}");
}
```

Panggil fungsi tersebut, tampung nilai baliknya ke variabel `res3`. Lalu siapkan `message3` yang isinya adalah sebuah pesan string literal yang dibuat menggunakan macro `format`. Terakhir tampilkan pesan tersebut menggunakan fungsi `greet_custom_message`.

```rust
fn main() {
    let res3 = calculate_box_volume3(width, height, length);
    let message3 = format!("the box volume is {}", res3);

    greet_custom_message("Damian", message3.as_str());
}
```

Bisa dilihat hasilnya sama seperti pada program sebelumnya.

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.48s
     Running `target\debug\function.exe`
hi Damian, the box volume is 480
```

◉ Macro `format` untuk membuat *formatted* string

Bagian ini adalah penjelasan tambahan dari kode program yang baru saja dipraktekan.

Macro `format` digunakan untuk *formatting* sebuah string. *Formatting* disini maksudnya adalah pembuatan string dengan menggunakan teknik penyusunan isi string dalam format tertentu, mirip seperti yang dilakukan menggunakan macro `print` atau `println`, bedanya hanya pada macro `format` hasilnya bukan sebuah output ke console output, melainkan sebagai nilai balik.

```rust
// 2 bari statements berikut ...
let message3 = format!("the box volume is {}", res3);
println!("{}", message3.as_str());

// ... adalah sama dengan statement berikut ...
println!("the box volume is {}", res3);
```

Satu hal yang penting diketahui dari macro `format`. Nilai balik macro ini bukanlah string literal `&str`, tetapi tipe `String`. **Dua tipe tersebut adalah berbeda**.

Pada contoh di atas, method `as_str` milik tipe data `String` digunakan untuk mengambil data string literal-nya. Method `as_str` ini menghasilkan data bertipe `&str` dari sebuah `String`.

A.17.5. *Conditional return value*

Kapan harus menggunakan keyword `return` dalam penentuan nilai balik dan kapan tidak? Jawabannya mungkin adalah preferensi, tapi diluar itu ada juga case dimana keyword `return` harus digunakan, yaitu pada fungsi yang memiliki nilai balik terkondisi. Contoh:

```
fn get_score_message(score: f32) -> &'static str {
    if score == 100.0 {
        return "you got a perfect score!"
    }

    if score > 76.0 {
        return "congrats, you passed the exam!"
    }

    "your score is below the passing grade"
}

fn main() {
    println!("{}", get_score_message(100.0));
    println!("{}", get_score_message(98.2));
    println!("{}", get_score_message(33.12));
}
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.44s
     Running `target\debug\function.exe`
you got a perfect score!
congrats, you passed the exam!
your score is below the passing grade
```

Teknik penentuan nilai balik tanpa keyword `return` hanya bisa dipergunakan di akhir blok kode, contohnya pada fungsi `get_score_message` statement terakhir blok kode adalah string literal `"your score is below the passing grade"`.

Jika pada selain akhir blok ada kondisi dimana nilai balik harus ditentukan, maka wajib menggunakan keyword `return`. Bisa dilihat pada fungsi yang sudah ditulis, `return` statement dalam blok kode seleksi kondisi `if` dipergunakan.

A.17.6. Nilai balik fungsi bertipe string literal `&str`

Khusus untuk beberapa jenis tipe data, seperti `&str`, jika digunakan sebagai tipe data *return value* fungsi harus ditambahi keyword `static` dengan penulisan `&'static str`. Contoh penerapan:

```
fn get_score_message(score: f32) -> &'static str {
    return "you got a perfect score!"
}
```

Agar tidak makin bingung, pembahasannya dipisah pada chapter lain, yaitu Static dan Lifetime.

A.17.7. *Default return value*

Tambahan info saja, bahwa di Rust, semua fungsi itu memiliki nilai balik, yep semuanya. Untuk fungsi yang tidak didefinisikan nilai baliknya maka nilai baliknya adalah tuple kosong `()`. Sebagai contoh fungsi `greet_custom_message` yang sudah ditulis, coba saja tampung nilai baliknya dan print. Outputnya adalah *empty tuple* atau tuple kosong `()`.

```
let a = greet_custom_message("Damian", "welcome to the castle!");
println!("result: {:?}", a);
```

```
4
5   let a: () = greet(name: "Damian", message: "welcome to the castle!");
6   println!("result: {:?}", a);
7
PROBLEMS   OUTPUT   CODELENS   JUPYTER   DEBUG CONSOLE   TERMINAL

(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarpemrogramanrust\examples\f
  Compiling function v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarpemrog
   Finished dev [unoptimized + debuginfo] target(s) in 0.40s
    Running `target\debug\function.exe`

hi Damian, welcome to the castle!
result: ()
```

A.17.8. Pembahasan lanjutan

Ada beberapa topik lainnya yang relevan dengan function, yaitu:

- **Associated function**, pembahasannya ada di chapter Associated Function
- **Method**, pembahasannya juga di chapter terpisah, yaitu Method

Selain itu nantinya juga ada pembahasan mengenai topik yang cukup advance yang berhubungan dengan fungsi, yaitu:

- **Unsafe function**, pembahasannya ada di chapter Safe & Unsafe
- **Async function**, pembahasannya ada di chapter Async
- **Constant function**, pembahasannya ada di chapter Constant Evaluation
- **Trait → Function**, pembahasannya ada di chapter Trait → Function

Untuk sekarang silakan lanjut ke pembahasan chapter berikutnya terlebih dahulu, jangan langsung loncat ke chapter di atas.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../function
```

◉ Work in progress

- Pembahasan tentang diverging function (`() -> !`)

◉ Referensi

- https://doc.rust-lang.org/book/ch03-03-how-functions-work.html
- https://doc.rust-lang.org/std/keyword.fn.html
- https://doc.rust-lang.org/std/keyword.return.html
- https://doc.rust-lang.org/std/keyword.static.html

- https://doc.rust-lang.org/rust-by-example/fn.html
- https://rust-lang.github.io/api-guidelines/naming.html

A.18. Module System → Path & Item

Chapter ini membahas tentang konsep *path* dan *item* dalam Rust programming.

A.18.1. Rust *Paths*

Paths (atau Path) adalah notasi penulisan alamat sebuah item, contohnya `std::time::Duration`.

`std::time::Duration` adalah *path* untuk item yang isinya adalah `struct` bernama `Duration`. Item bisa berupa banyak jenis, bisa saja struct, atau *macro*, konstanta, atau lainnya. Lebih jelasnya silakan cek pada halaman dokumentasi `std::time::Duration`.

Sebuah path bisa memiliki banyak bagian (biasa disebut *segment*), sebagai contoh, path `std::time::Duration` memiliki 3 segmen yaitu `std`, `time`, dan `Duration`. Karakter `::` digunakan dalam penulisan path sebagai pembatas antar segmen (jika path memiliki lebih dari 1 segmen).

Dalam sebuah path, yang disebut dengan item adalah segment terakhir. Contohnya path `std::time::Duration`, maka item yang dituju adalah struct `Duration`.

> *Rust paths mirip seperti konsep filesystem path di sistem operasi. Seperti `C:\Users\novalagung\Desktop` di windows, atau `/etc/nginx/conf.d/nginx.conf` di Unix/Linux.*
>
> *Di Rust, path tidak menggunakan `\` atau `/` sebagai separator, melainkan `::`.*

Jika di-breakdown, berikut adalah penjelasan dari setiap kombinasi segmen path pada contoh `std::time::Duration`.

- Path `std` → adalah path untuk **crate** bernama **Rust Standard Library**, isinya adalah sangat banyak item untuk keperluan umum di Rust programming. Lebih jelasnya akan dibahas pada chapter Rust standard library.
- Path `std::time` → adalah path untuk **module** bernama `time`, isinya banyak item yang berhubungan dengan operasi waktu/time.
- Path `std::time::Duration` → adalah path untuk **struct** bernama `Duration`, yang merupakan representasi dari unit waktu.

> - *Lebih jelasnya mengenai crate dibahas pada chapter Module System → Package & Crate*
> - *Lebih jelasnya mengenai module dibahas pada chapter Module System → Module*
> - *Lebih jelasnya mengenai struct dibahas pada chapter Struct*

A.18.2. Absolute & relative paths

Rust mengenal dua jenis path:

- Absolute path → adalah path yang penulisannya lengkap dari root path, contohnya seperti `std::time::Duration`.
- Relative path → adalah path yang penulisannya relatif terhadap current path, contohnya seperti `self::my_func`, `super::my_mod::my_consntan`.

> *Lebih jelasnya mengenai relative path dibahas pada chapter Module System → Scope & Akses Item.*

A.18.3. Rust *Items*

Seperti yang sudah dibahas bahwa path adalah notasi penulisan alamat untuk item. Lalu apa saja yang disebut dengan item? Di Rust ada banyak hal, yang kurang lebih list-nya bisa dilihat berikut:

- modules *(dibahas pada chapter Module System → Module)*
- extern crate declarations
- use declarations *(dibahas pada chapter Module System → Use)*
- function definitions *(dibahas pada chapter Function, Associated Function, dan Method)*
- type definitions *(dibahas pada chapter-chapter tentang tipe data)*
- struct definitions *(dibahas pada chapter Struct)*
- enumeration definitions *(dibahas pada chapter Enum)*
- union definitions
- constant items *(dibahas pada chapter Konstanta)*
- static items *(dibahas pada chapter Static Items)*
- trait definitions *(dibahas pada chapter Traits)*
- implementations ((dibahas pada chapter Function, Associated Function, dan Method))

- extern blocks

A.18.4. Penerapan paths dalam penggaksesan item

Pada bagian ini, kita akan coba terapkan path untuk mengakses beberapa item.

Dalam program sederhana berikut, inputan user ditampung sebagai string, kemudian ditampilkan.

```
fn main() {
    // tampilkan intro untuk user agar menginput sebuah pesan
    println!("enter a message:");

    // variabel yang akan menampung inputan user dalam string
    let mut message = std::string::String::new();

    // objek reader untuk membaca inputan user
    let stdin_reader = std::io::stdin();

    // proses pembacaan inputan user
    let reader_res = stdin_reader.read_line(&mut message);

    // pengecekan apakah ada error dalam pembacaan inputan.
    // jika iya, maka tampilkan error dan hentikan program
    if reader_res.is_err() {
        println!("error! {:?}", reader_res.err());
        return;
    }

    // tampilkan pesan inputan user
    println!("message: {}", message);
```

Jalankan program, lalu inputkan sebuah pesan, kemudian enter.

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarp
   Compiling module_system_paths_items v0.1.0 (D:\Labs\Adam Stu
    Finished dev [unoptimized + debuginfo] target(s) in 0.62s
     Running `target\debug\module_system_paths_items.exe`

enter a message:
halo, selamat sore
message: halo, selamat sore
```

Bisa dilihat pesan ditampilkan sesuai inputan. Oke, sekarang kita akan bahas program di atas per-barisnya.

◉ `std::string::String::new()`

Variabel `message` adalah string yang dibuat menggunakan tipe data custom `String`. Salah satu cara pembuatan string bertipe data ini adalah menggunakan statement `std::string::String::new()`.

Bisa dilihat tanda `::` digunakan pada statement tersebut. Path tersebut jika dijabarkan per segment:

- Segment `std` adalah crate *Rust Standard Library*
- Pada path `std::string`, segment `string` adalah module
- Pada path `std::string::String`, segment `String` adalah struct String, yang biasa disebut dengan *custom type* `String`
- Pada path `std::string::String::new`, segment `new` adalah sebuah fungsi milik struct `String` yang mengembalikan data bertipe custom string `String`.
- Di segment terakhir, yaitu `new`, ditambahkan tanda pemanggilan fungsi `()`, yang menandakan bahwa fungsi `new` dipanggil. Dengan itu maka nilai baliknya (yang berupa `String`) ditampung oleh variabel `message`.

> *Terkesan banyak sekali penjelasan dalam 1 baris kode, tapi tidak perlu dihafal, lama-kelamaan akan terbiasa.*

Salah satu yang menarik dari create *Rust Standard Library* adalah, beberapa segmen otomatis di-import atau dipakai. Jadi tidak perlu menuliskan path secara full.

Pada contoh yang sudah dibuat, statement pembuatan data `string` bisa diubah dari ...

```
let mut message = std::string::String::new();
```

... menjadi cukup ...

```
let mut message = String::new();
```

◉ `std::io::stdin()`

Berbeda dengan `String`, path `std::io::stdin` tidak otomatis ter-import, jadi harus dituliskan secara full meskipun sama-sama dibawah crate *Rust Standard Library*

Path `std::io` berisi module untuk keperluan I/O atau input output. Salah satu item yang ada dalam module ini adalah `stdin`, yang merupakan sebuah fungsi berguna untuk pembuatan objek handler untuk keperluan yang berhubungan dengan console (*stdin*). Objek tersebut ditampung oleh variabel `stdin_reader`.

> *Secara terminologi, stdin (yang merupakan kependekan dari standard input) adalah sebuah input stream dimana data dikirim dan dibaca oleh*

> *program*.

Variabel `stdin_reader` ini kemudian kita gunakan untuk berinteraksi dengan input stream, untuk menangkap inputan user.

◉ `stdin_reader.read_line(&mut message)`

Method `read_line` milik variabel `stdin_reader` berguna untuk menangkap inputan user. Variabel yang disisipkan sebagai argumen pemanggilan method tersebut menjadi penampung inputan user, yang pada contoh ini adalah `message`.

Perlu diperhatikan bahwa `message` disisipkan *mutable reference*-nya sebagai argumen pemanggilan fungsi.

> *Lebih jelasnya mengenai pointer dan mutable reference dibahas pada chapter Pointer & References*

Eksekusi dari statement `stdin_reader.read_line(&mut message)` adalah blocking, artinya program akan berhenti untuk sementara di baris tersebut, hingga ada inputan dari user dan tombol enter ditekan.

◉ Pengecekan error `stdin_reader`

Blok statement `if` pada contoh diatas bertugas melakukan pengecekan error. Jika ada error, maka `reader_res.is_err()` bernilai `true`, dan pesan error-nya dimunculkan.

```
if reader_res.is_err() {
```

◉ Menampilkan isi `message`

Jika program berlajan sesuai harapan, tanpa error, pada baris terakhir data dalam `message` ditampilkan ke layar.

> *Tipe data `string` ini tidak perlu di-konversi ke bentuk literal string `&str` untuk ditampilkan menggunakan `println`. Langsung saja sisipkan variabel `string` ke macro tersebut dan `println` akan tau harus menampilkan apa.*

Oke, Penulis rasa sudah cukup jelas perihal bagaimana cara menggunakan path untuk mengakses item. Cukup tulis saja path-nya. Jika path-nya panjang? ya ditulis semua.

A.18.5. Penggunaan keyword `use` untuk import path

Ada alternatif cara lain untuk memperpendek penulisan dan pengaksesan path, yaitu dengan menggunakan keyword `use`.

> *Penggunaan `use` juga sempat dipraktekan pada chapter sebelumnya, yaitu Perulangan → while.*

Cara penerapannya bisa dilihat pada kode berikut:

```
let stdin_reader = std::io::stdin();
```

Dengan menggunakan `use` kita bisa memperpendek pengaksesan sebuah path.

O iya keyword ini bisa digunakan dimana saja, artinya tidak harus di luar fungsi `main`. Bisa saja di dalam fungsi, atau didalam blok kode seleksi kondisi atau lainnya.

A.18.6. Pembahasan lanjutan

Kita sudah beberapa kali menerapkan path untuk mengakses item milik crate *Rust Standard Library*, lalu bagaimana penerapan path untuk internal item, pastinya pada proyek real dalam 1 program akan ada banyak item.

Jawabannya akan ada di beberapa chapter berikutnya. Untuk sekarang khusus pada bagian **module system** ini, penulis anjurkan untuk mengikuti urutan pembelajaran ebook terlebih dahulu.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../path_item
```

◉ Referensi

- https://doc.rust-lang.org/book/ch07-03-paths-for-referring-to-an-item-in-the-module-tree.html

- https://doc.rust-lang.org/reference/paths.html
- https://doc.rust-lang.org/reference/items.html
- https://doc.rust-lang.org/std/keyword.use.html
- https://doc.rust-lang.org/edition-guide/rust-2018/path-changes.html

---

A.19. Module System → Package & Crate

Chapter ini membahas mengenai konsep package dan path dalam Rust programming. Pembelajaran dimulai dari pembahasan mengenai konsep crate terlebih dahulu, kemudian masuk ke package.

A.19.1. Rust Crate

Crate adalah satu unit kompilasi di Rust. Eksekusi command `cargo run`, `cargo build`, atau `rustc` men-*trigger* proses kompilasi, dan unit (yang disini disebut dengan crate) akan di-compile.

Crate bisa berisi banyak *module*. Sebuah module definisinya bisa berada di banyak file. Agar lebih jelas silakan perhatikan contoh berikut:

- `XYZ` adalah sebuah crate, isinya ada dua module, yaitu module `Mod_ABC` dan module `Mod_DEF`.
- `Mod_ABC` adalah module yang didefinisikan dalam crate `XYZ`, source code-nya berada di file bernama `modul_a.rs`.
- `Mod_DEF` adalah module yang didefinisikan dalam crate `XYZ`, source code-nya berada di beberapa file `module_b_one.rs` dan `module_b_two.rs`.

Dari contoh di atas, crate `XYZ` adalah 1 unit kompilasi, yang dimana didalam crate tersebut ada dua modules yaitu `Mod_ABC` dan `Mod_DEF`

Rust mengkategorikan crate menjadi 2 jenis, *binary crate* dan *library crate*

◉ Binary crate

Binary crate adalah program yang dikompilasi ke bentuk *executable*, untuk kemudian dijalankan, seperti program-program yang sudah kita buat menggunakan `rago create` dan run menggunakan `cargo run` itu adalah contoh dari binary crate.

Binary crate berada dalam sebuah package yang dibuat menggunakan command `cargo create <nama_package>` atau `cargo create --bin <nama_package>`, kedua command ini menjalankan perintah yang sama.

Ciri khas dari binary crate adalah memiliki fungsi `main`, sebuah fungsi yang merupakan *entrypoint* program.

◉ Library crate

Library crate berbeda dengan binary crate. Library crate tidak di-compile ke bentuk *executable* dan tidak memiliki fungsi `main`. Library crate digunakan untuk mendefinisikan set *functionality* yang *reusable* atau bisa digunakan di banyak project/package.

> *Library crate di-import/digunakan dalam binary crate. Dalam proses kompilasinya, yang di-compile adalah binary crate. Library crate juga akan ikut dalam kompilasi tersebut.*

Sebagai contoh item `Duration` (yang sudah dipraktekan pada chapter Perulangan → while) dan `stdin` (pada chapter Module System → Path & Item) adalah dua buah item milik crate *Rust Standard Library* atau `std`. Crate `std` ini akan sangat sering kita gunakan dalam package/project, isinya banyak sekali functionality untuk keperluan standar dalam Rust programming.

> *Di komunitas Rust, ketika ada kata library atau crate maka yang dimaksud biasanya adalah library crate*

Kita bisa membuat library crate kemudian di-publish ke crates.io agar bisa digunakan banyak orang. Command `cargo new --lib <nama_package>` digunakan untuk membuat library crate.

> *Lebih jelasnya mengenai library crate dibahas terpisah pada chapter Library Crate*

A.19.2. Rust Package

Istilah package dalam Rust programming masih sama dengan package dalam pemrograman lain. Package adalah sebuah set yang berisi banyak *functionality*. Satu buah package bisa berisi satu atau banyak crates.

Package di-manage oleh Cargo, yang merupakan package manager Rust. Command `cargo new <nama_package>` digunakan untuk membuat package. Command tersebut menghasilkan beberapa file yaitu `src/main.rs` yang isinya adalah kode program, dan juga file `cargo.toml` yang isinya adalah informasi mengenai package tersebut.

Ok, sekalian praktek mungkin lebih pas. Silakan jalankan command berikut, kemudian ikuti penjelasan selanjutnya.

```
cargo new belajar_package_crate
```

◉ *Naming convention* package

Khusus untuk penamaan package dan juga crate, tidak ada rekomendasi dari dokumentasi official Rust. Pada ebook ini penulis menggunakan *snake case* sebagai naming convention dalam pembuatan pcakage.

Referensi: https://rust-lang.github.io/api-guidelines/naming.html

◉ File `Cargo.toml`

File `Cargo.toml` menampung beberapa informasi penting milik package, diantaranya adalah nama package, versi package dan juga versi rust, serta *dependencies* atau *3rd-party* yang digunakan dalam package (dalam konteks Rust adalah *crate*).

Command yang sebelumnya di run menghasilkan file `cargo.toml` berikut:

Cargo.toml

```
[package]
name = "belajar_package_crate"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-
lang.org/cargo/reference/manifest.html

[dependencies]
```

Blok `package` berisi berisi 3 buah field:

- `name` isinya adalah nama package, sesuai dengan argument command

`cargo new <nama_package>`.

- `version` default-nya selalu `0.1.0`, namun kita bisa ubah nilainya seiring berjalannya proses pengembangan aplikasi.
- `edition` disini me-refer ke edisi rust yang dipakai. Di ebook ini, rust versi **1.65.0** digunakan, dan edisi untuk versi tersebut adalah `2021`.

Blok `dependencies` default-nya berisi kosong. Jika kita menambahkan external dependencies atau crates, maka detailnya tercatat pada blok `dependencies` ini.

◉ Website crates.io

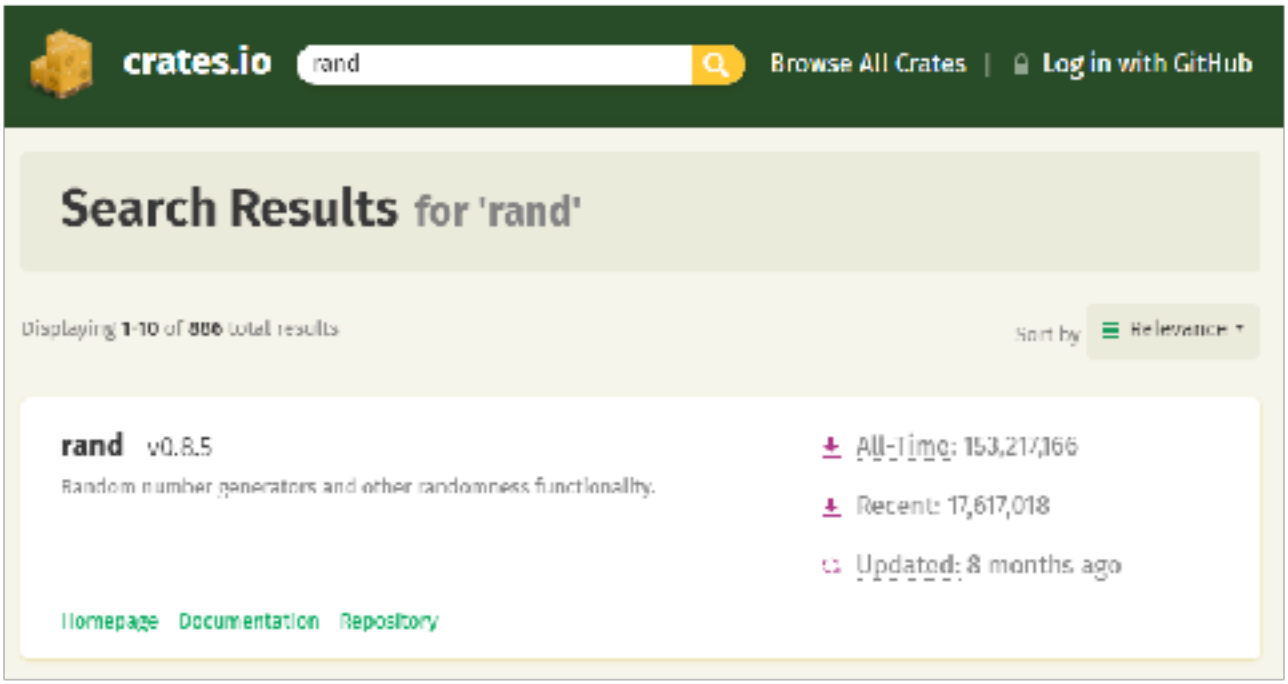
crates.io merupakan official website dari Rust untuk pencarian dan juga *managing* dependency. Silakan manfaatkan website tersebut untuk mencari crates sesuai kebutuhan.

> *https://crates.io/*

◉ Menambahkan dependency atau external crate

proses pembelajaran akan dilanjutkan sambil praktek. Pada bagian ini kita akan buat program sederhana yang didalamnya memanfaatkan sebuah dependency atau external crate.

Pertama-tama buka crates.io, lalu gunakan keyword `rand` dalam pencarian, hasilnya adalah dependency bernama `rand`.



`rand` merupakan crate yang isinya banyak fungsi untuk keperluan *generate* data random.

Sekarang buka file `cargo.toml`, dan tambahkan dependency `rand`. Sesuaikan dengan versi yang muncul di layar masing-masing.

Cargo.toml

```
[package]
name = "package_crate"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
rand = "0.8.5"
```

Jalankan command `cargo build` untuk memaksa Cargo agar mendownload depdency yang sudah ditambahkan ke file `Cargo.toml`.

```
(base) package_crate> cargo build
  Compiling cfg-if v1.0.0
  Compiling ppv lite86 v0.2.16
  Compiling getrandom v0.2.7
  Compiling rand_core v0.6.4
  Compiling rand_chacha v0.3.1
  Compiling rand v0.8.5
  Compiling package_crate v0.1.0 (D:\Labs\Adam Studio\Ebook\da
   Finished dev [unoptimized + debuginfo] target(s) in 2.62s
```

Sukses! Sekarang dependency `rand` sudah bisa digunakan dalam package yang sudah dibuat.

> *Jika pembaca menemui error `failed to authenticate when downloading repository`, jalankan beberapa command berikut secara berurutan:*
>
> - `ssh-agent -s`
> - `ssh-add`
> - `cargo build`

Jadi kurang lebih seperti itu cara menambahkan dependency di Rust. Seiring berjalannya proses pembelajaran penulis yakin pembaca akan terbiasa dengan flow dan juga command di atas.

A.19.3. Praktek membuat program menampilkan angka random

Ok, sekarang kita lanjutkan praktek pembuatan program sederhana untuk

menampilkan angka random.

Angka random disini akan di-generate menggunakan fungsi yang ada dalam dependency `rand` yang sudah ditambahkan ke package.

Silakan tulis kode berikut ke file `src/main.rs`.

```
use rand::Rng;

fn generate_random_number() -> i32 {
    rand::thread_rng().gen_range(0..100)
}

fn main() {
    for i in 0..5 {
        println!("random number {}: {}", i,
generate_random_number());
    }
}
```

Jalankan program, lalu lihat hasilnya. Sejumlah data angka random muncul di layar output.

```
(base) PS D:\Labs\Wam Studio\Ebook\dasarpemrograman
   Finished dev [unoptimized + debuginfo] target(s) in 0.02s
    Running `target\debug\package_crate.exe`
random number 0: 45
random number 1: 94
random number 2: 99
random number 3: 7
random number 4: 25
```

Silakan coba untuk run beberapa kali, angka random berbeda setiap kali run.

◉ Fungsi `generate_random_number`

`generate_random_number` adalah fungsi yang kita buat, yang tugasnya mengembalikan sebuah nilai numerik bertipe `i32`. Angka tersebut berasal dari proses *generate random* hasil eksekusi statement `rand::thread_rng().gen_range(0..100)`.

◉ Statement `rand::thread_rng().gen_range(0..100)`

`rand::thread_rng().gen_range` digunakan untuk generate data random. Fungsi `gen_range` menerima argument bertipe range, dan angka random akan di-generate sesuai range tersebut. Sebagai contoh, range `0..100` menghasilkan angka random antara `0` hingga `99`.

O iya, penggunaan fungsi `gen_range` mewajibkan kita untuk import path `rand::Rng`. Itulah kenapa ada statement `use rand::Rng`.

◉ Fungsi `main`

Dalam blok kode `main`, isinya sebuah perulangan sederhana yang menampilkan angka random hasil eksekusi fungsi `generate_random_number` disetiap iterasinya.

```
for i in 0..5 {
    println!("random number {}: {}", i, generate_random_number());
}
```

A.19.4. Pembahasan lanjutan

Sekian pembahasan mengenai package dan crate. Semoga membantu, silakan diulang-ulang jika perlu agar tidak bingung, **module system** merupakan salah satu hal yang sangat penting di Rust.

Masih dalam topik *module system*, penulis anjurkan untuk lanjut ke chapter berikutnya karena berkaitan.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-
example/../package_crate
```

◉ Referensi

- https://doc.rust-lang.org/book/ch07-01-packages-and-crates.html
- https://doc.rust-lang.org/rust-by-example/crates.html
- https://rust-lang.github.io/api-guidelines/naming.html
- https://aloso.github.io/2021/03/28/module-system.html
- https://github.com/rust-lang/cargo/issues/3381

A.20. Module System → Module

Module adalah salah satu chapter yang cukup penting dalam pemrograman Rust. Pada bagian ini kita akan mempelajari dasarnya.

A.20.1. Rust module system

Setiap bahasa pemrograman memiliki caranya sendiri dalam hal pengelolahan struktur files dan folder dalam project. Project yang isinya ada sangat banyak hal diatur sedemikian rupa menjadi beberapa bagian dan/atau sub-bagian sesuai dengan fungsinya masing-masing.

Di Rust, module memiliki hirarki (biasa disebut dengan *module tree*) yang *root*/akarnya adalah file entrypoint crate, yaitu `main.rs` untuk *binary crate* dan `lib.rs` untuk *library crate*. Kedua file ini biasa disebut dengan *crate root file*.

> *Disini pembahasan akan fokus pada penerapan module dalam binary crate. Kita belum masuk ke pembahasan tentang library crate.*

Rust memiliki 2 jenis modul, yaitu *normal module* dan *inline module*. Pembahasan dimulai dengan normal module terlebih dahulu.

Keyword `mod` digunakan untuk mendefinisikan/mendaftarkan sebuah module. Nama module menjadi path dimana isi module atau *module item* harus berada. Sebagai contoh:

- module yang didefinisikan dengan nama `my_number`, maka item-nya harus berada pada file `my_number.rs` atau `my_number/mod.rs`
- module yang didefinisikan dengan nama `my_io`, maka item-nya harus berada pada file `my_io.rs` atau `my_io/mod.rs`

Pendefinisian nama module sendiri berada pada file entrypoint *crate* yaitu `main.rs` (atau `lib.rs` untuk library crate). Jadi pendefinisian nama module dan isi/item-nya terpisah.

- Nama module ditulis di `main.rs` (atau `lib.rs` untuk library crate)
- Item atau isi module ditulis dalam file `nama_module.rs` atau `nama_module/mod.rs`

> *Sebenarnya ada beberapa hal lainnya lagi yang masih relevan yang perlu dibahas di-awal, yaitu perihal sub-module. Akan tetapi agar tidak makin bingung, mari kita lanjut ke praktek terlebih dahulu.*

A.20.2. Praktek #1 - `nama_module.rs`

Mari buat program sederhana, yang isinya mencakup pembahasan tentang module. Pada program kecil ini, inputan user ditampung ke sebuah variabel, kemudian ditampilkan ke layar. Proses pembacaan inputan user akan di-split sebagai module.

Ok, langsung saja, buat package/project baru dengan nama bebas. Disini penulis menggunakan nama package `module_1`.

```
cargo new module_1
```

Setelah itu siapkan 1 buah file bernama `my_io.rs`, letakan di dalam folder `src` (1 level dengan file `main.rs`). File ini difungsikan sebagai tempat definisi *module item* milik sebuah module bernama `my_io` (io disini kependekan dari *input output*).

package source code structure

```
module_1
|── Cargo.toml
└── src
    |── main.rs
    └── my_io.rs
```

Lanjut, definisikan fungsi `read_entry` di file `my_io.rs`, isinya kurang lebih adalah kode untuk membaca inputan user lalu mengembalikannya dalam bentuk `String`.

src/my_io.rs

```rust
pub fn read_entry() -> String {
    let mut message = std::string::String::new();
    let stdin_reader = std::io::stdin();
    let reader_res = stdin_reader.read_line(&mut message);

    if reader_res.is_err() {
        println!("error! {:?}", reader_res.err());
    }

    message.trim().to_string()
}
```

Ok, kita telah menyiapkan satu item milik module `my_io` yaitu sebuah fungsi bernama `read_entry`. 1 hal yang sedikit berbeda pada definisi fungsi diatas

adalah penambahan keyword `pub` yang ditulis sebagai prefix definisi fungsi.

Keyword `pub` digunakan untuk menjadikan suatu item menjadi **public**, agar bisa diakses dari luar module.

Fungsi `read_entry` ini berada dalam module `my_io`. Jika tidak ada keyword `pub` disitu, maka fungsi `read_entry` hanya bisa diakses dari dalam `my_io` saja, tidak bisa diakses dari luar module contohnya seperti dari `main.rs`. Dengan menjadikan `read_entry` sebagai fungsi yang public, maka fungsi tersebut bisa diakses dari `main.rs`.

> *Lebih jelasnya mengenai keyword `pub` dibahas pada chapter Module System → Visibility & Privacy*

Isi module sudah siap, selanjutnya lanjut ke pendefinisian modul. Umumnya pada bahasa pemrograman, definisi module adalah ada dalam file dimana isi module berada, namun tidak untuk Rust.

Di Rust, definisi sebuah module (sekali lagi bukan item/isinya ya, tapi definisi dari module itu sendiri) dituliskan pada file terpisah, yaitu di file entrypoint crate, yaitu `main.rs` atau `lib.rs`.

Lanjut, sekarang buka file `main.rs` dan tulis definisi modul `my_io` menggunakan statement `mod my_io`.

src/main.rs

```
// definisi module my_io
mod my_io;

// fungsi main
```

Sekarang aplikasikan fungsi `read_entry` milik module `my_io` di fungsi `main` untuk membaca inputan user.

src/main.rs

```
mod my_io;

fn main() {
    println!("enter any number:");
    let message = my_io::read_entry();
    println!("your number: {}", message);
}
```

Jalankan program, masukan sebuah angka, lalu enter. Lihat hasilnya, program berjalan sesuai harapan.

```
EXPLORER
MODUL_1
  src
    main.rs U
    my_io.rs U
  target
  Cargo.lock U
  Cargo.toml U

main.rs U
src > main.rs > _
mod my_io;

Run | Debug
fn main() {
    println!("enter any number:");
    let message: String = my_io::read_entry();
    println!("your number: {}", message);
}

my_io.rs U
src > my_io.rs > read_entry
pub fn read_entry() -> String {
    let mut message: String = std::string::String::new();
    let stdin_reader: Stdin = std::io::stdin();
    let reader_res = stdin_reader.read_line(buf: &mut message);

    if reader_res.is_err() {
        println!("error! {:?}", reader_res.err());
    }

    message.trim().to_string()
}

PROBLEMS   OUTPUT   COMMENTS   JUPYTER   DEBUG CONSOLE   TERMINAL

(base) PS D:\dev\programsamurai\examples\module_1> cargo run
   Compiling module_1 v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in 0.98s
     Running `target\debug\module_1.exe`
enter any number:
23
your number: 23
```

Cara pengaksesan item dari sebuah module yang kita definisikan sendiri adalah sama seperti pengaksesan item dari crate lainnya, yaitu menggunakan *path*. Statement `my_io::read_entry()` artinya item `read_entry` yang merupakan fungsi dalam module `my_io` digunakan.

◉ Summary praktek #1

Pada praktek pertama ini kita telah belajar penerapan module dengan mengaplikasikan beberapa hal berikut:

- Penggunaan normal module dalam binary carete
- Pembuatan module dengan nama `my_io`, dengan isi/item ditulis pada file `my_io.rs`
- Penggunaan keyword `pub` untuk meng-export atau menjadikan item menjadi public, agar bisa diakses dari luar module
- Pengaksesan item milik module, yaitu: `my_io::read_entry`

◉ *Naming convention* module

Sesuai anjuran di halaman dokumentasi Rust, aturan penulisan nama module adalah menggunakan snake case, contohnya `my_io`.

◉ Module item

Module item adalah apapun yang didefinisikan didalam sebuah module. Pada contoh praktek ke-1 di atas, module `my_io` memiliki 1 buah item yaitu fungsi bernama `read_entry`.

Selain fungsi, module item bisa dalam bentuk lainnya, contohnya: konstanta, submodule, struct, dan lainnya.

> *Lebih jelasnya tentang macam-macam item dibahas pada chapter Module System → Path & Item*

A.20.3. Praktek #2 - `nama_module/mod.rs`

Bagian ini merupakan kelanjutan dari praktek sebelumnya. Program sederhana yang sudah di-buat ditambahi beberapa hal. Data inputan user dikonversi ke bentuk angka untuk kemudian dicek apakah angka tersebut bilangan ganjil atau genap.

Fungsi untuk konversi string ke bentuk numerik dan juga untuk pengecekan bilangan ganjil genap, adalah dua buah item milik module bernama `my_number` yang akan kita definisikan sebentar lagi.

Definisi item module `my_io` menggunakan notasi penulisan file `nama_module.rs` (yaitu `my_io.rs`). Pada module `my_number` ini kita akan gunakan notasi penulisan `nama_module/mod.rs` (menjadi `my_number/mod.rs`) untuk menampung definisi item module `my_number`.

package source code structure

```
module_1
|── Cargo.toml
└── src
     |── my_number
     |     └── mod.rs
     |── main.rs
     └── my_io.rs
```

Ok, sekarang buat saja folder dan filenya, yaitu `my_number/mod.rs`. Lalu pada file `mod.rs` tulis 2 buah fungsi berikut:

- Fungsi untuk konversi string ke numerik `i32`

  src/my_number/mod.rs

  ```rust
  pub fn string_to_number(text: String) -> i32 {
      return text.parse::<i32>().unwrap();
  }
  ```

- Fungsi untuk pengecekan bilangan ganjil

  src/my_number/mod.rs

  ```rust
  // ...

  pub fn is_odd_number(number: i32) -> bool {
      number % 2 == 1
  }
  ```

Kemudian tulis definisi module `my_number` dalam file entrypoint crate, yaitu `main.rs`.

src/main.rs

```rust
// definisi module my_io
mod my_io;

// definisi module my_number
mod my_number;

// fungsi main
fn main() {
```

Terakhir, aplikasikan dua buah fungsi yang sudah didefinisikan.

src/main.rs

```
fn main() {
    println!("enter any number:");
    let message = my_io::read_entry();
    println!("your number: {}", message);

    let number = my_number::string_to_number(message);
    let result = my_number::is_odd_number(number);
    println!("is odd number: {}", result);
}
```

Jalankan program, lihat hasilnya.

```
EXPLORER
MODULE 1
src
my_number
mod.rs
main.rs
my_io.rs
target
Cargo.lock
Cargo.toml
```

```
src > main.rs > ...
mod my_io;
mod my_number;

Run | Debug
fn main() {
    println!("enter any number:");
    let message: String = my_io::read_entry();
    println!("your number: {}", message);

    let number: i32 = my_number::string_to_number(text: message);
    let result: bool = my_number::is_odd_number(number);
    println!("is odd number: {}", result);
}
```

```
src > my_number > mod.rs > is_odd_number
pub fn string_to_number(text: String) -> i32 {
    return text.parse::<i32>().unwrap();    };
}

pub fn is_odd_number(number: i32) -> bool {
    number % 2 == 1
}
```

```
PROBLEMS   OUTPUT   GITLENS   JUPYTER   DEBUG CONSOLE   TERMINAL

(base) PS D:\dasarpemrograman-rust\examples\module_1> cargo run
   Compiling module_1 v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in 0.08s
     Running `target\debug\module_1.exe`
enter any number:
23
your number: 24
is odd number: true

(base) PS D:\dasarpemrograman-rust\examples\module_1> cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\module_1.exe`
enter any number:
24
your number: 24
is odd number: false
```

◉ Summary praktek #2

Hingga praktek ke-2 ini, kita telah belajar penerapan module dengan

mengaplikasikan beberapa hal berikut:

- Penggunaan normal module dalam binary carete
- Pembuatan module dengan nama `my_io`, dengan isi/item ditulis pada file `my_io.rs`
- Pembuatan module dengan nama `my_number`, dengan isi/item ditulis pada file `my_number/mod.rs`
- Penggunaan keyword `pub` untuk meng-export atau menjadikan item menjadi public, agar bisa diakses dari luar module
- Pengaksesan item milik module, yaitu: `my_io::read_entry`, `my_number::string_to_number`, dan `my_number::is_odd_number`

> *Penerapan notasi penulisan isi module `nama_module.rs` dan `nama_module/mod.rs` bisa diterapkan dalam satu package*

◉ Konversi string ke numerik

Rust mengenal beberapa jenis konversi antar tipe data. Teknik konversi yang kita praktekan pada chapter ini menggunakan method `.parse()`, cara ini bisa dilakukan untuk konversi tipe data dari custom types ke primitive. Contohnya seperti di atas, dari `String` ke `i32`.

Penggunaannya cukup mudah, akses saja method `.parse()` kemudian sisipkan tipe data sebagai parameter generic. Lalu chain dengan method `unwrap` (yang method tersebut merupakan item milik tipe data *generic result type* atau `Result<T, E>`).

Contoh lain untuk konversi dari tipe primitif lainnya bisa dilihat berikut:

```
let data_number = "24".parse::<i32>().unwrap();
let data_bool = "true".parse::<bool>().unwrap();
let data_float = "3.14".parse::<f64>().unwrap();
```

`bcrypt::hash` menghasilkan data bertipe *generic result type* atau `Result<T, E>`. Tipe ini memiliki method bernama `unwrap` yang berguna untuk mengambil nilai.

> - *Lebih jelasnya mengenai casting dibahas pada chapter Type Alias & Casting dan Trait → Conversion (From & Into)*
> - *Lebih jelasnya mengenai generic dibahas pada chapter Generics*
> - *Lebih jelasnya mengenai result type dibahas pada chapter Tipe Data → Result*

A.20.4. Submodules

Sampai bagian ini kita telah belajar tentang module beserta 2 macam cara penerapannya.

Sebuah module bisa saja memiliki module dibawahnya (biasa disebut submodule), dan hal ini adalah konsep yang umum dalam bahasa pemrograman. Di Rust, aturan dalam pembuatan submodule masih sama seperti module, perbedaannya adalah tempat dimana submodule didefinisikan. Jika pada root module definisi ada pada file `main.rs` atau `lib.rs`, maka pada submodule definisi ada pada file dimana *parent module* berada.

Sebagai contoh jika pada program sebelumnya kita tambahkan module `my_number` yang sudah dibuat, jika ada submodule dengan nama `conversion_utility`, maka definisi module berada di `my_number/mod.rs` dan

itemnya di `my_number/conversion_utility/mod.rs`.

package source code structure

```
my_package
|── Cargo.toml
└── src
     |── main.rs
     |── my_io.rs
     └── my_number
          |── mod.rs              // <----- definisi submodule
conversion_utility
          └── conversion_utility
               └── mod.rs         // <----- definisi item/isi
conversion_utility
```

Atau definisi module tetap di `my_number/mod.rs` namun itemnya di `my_number/conversion_utility.rs`.

package source code structure

```
my_package
|── Cargo.toml
└── src
     |── main.rs
     |── my_io.rs
     └── my_number
          |── mod.rs                // <----- definisi submodule
conversion_utility
          └── conversion_utility.rs // <----- definisi item/isi
conversion_utility
```

> *Penerapan notasi penulisan `nama_module.rs` biasanya dalam case ketika module tersebut tidak memiliki submodule.*
>
> *Untuk module yang memiliki submodule, parent module harus menerapkan notasi penulisan `nama_module/mod.rs`, hal ini karena pendefinisian submodule berada pada file `mod.rs` dalam parent module tersebut.*

Lalu bagaimana jika sebuah submodule memiliki submodule yang memiliki submodule ... dst, aturannya tetap sama seperti aturan submodule.

Mari lanjut praktek agar tidak bingung. Kita akan modifikasi program sebelumnya. Item `string_to_number` yang sebelumnya adalah item milik `my_number` kita pindah ke sebuah module baru bernama `conversion_utility` yang merupakan submodule dari `my_number`. Silakan buat file baru `my_number/conversion_utility/mod.rs`, kemudian isi dengan fungsi berikut:

src/my_number/conversion_utility/mod.rs

```
pub fn string_to_number(text: String) -> i32 {
    return text.parse::<i32>().unwrap();
}
```

Fungsi `string_to_number` yang sebelumnya ada di `my_number/mod.rs` silakan dihapus.

Kemudian pada file `my_number/mod.rs`, tambahkan definisi submodule `conversion_utility`. Isi file tersebut kurang lebih menjadi seperti berikut:

src/my_number/mod.rs

```rust
pub mod conversion_utility;

pub fn is_odd_number(number: i32) -> bool {
    number % 2 == 1
}
```

O iya, karena submodule merupakan sebuah item milik module, maka harus ditambahkan juga keyword `pub`, agar submodule bisa diakses dari luar scope-nya. Contoh penerapannya bisa dilihat di atas.

Terakhir, pada `main.rs`, ubah pemanggilan fungsi `string_to_number` dari ...

src/main.rs

```rust
let number = my_number::string_to_number(message);
```

... menjadi ...

src/main.rs

```rust
let number =
my_number::conversion_utility::string_to_number(message);
```

Kurang lebih strukturnya menjadi seperti berikut:

```rust
mod my_io;
mod my_number;

fn main() {
    println!("enter any number:");
    let message: String = my_io::read_entry();
    println!("your number: {}", message);

    let number: i32 = my_number::conversion_utility::string_to_number(text: message);
    let result: bool = my_number::is_odd_number(number);
    println!("is odd number: {}", result);
}
```

```rust
pub fn read_entry() -> String {
    let mut message: String = std::string::String::new();
    let stdin_reader: Stdin = std::io::stdin();
    let reader_res: Result<usize, Error> = stdin_reader.read_line(buf: &mut message);

    if reader_res.is_err() {
        panic!("error! {:?}", reader_res.err());
    }

    message.trim().to_string()
}
```

```rust
pub mod conversion_utility;

pub fn is_odd_number(number: i32) -> bool {
    number % 2 == 1
}
```

```rust
pub fn string_to_number(text: String) -> i32 {
    return text.parse::<i32>().unwrap();
}
```

Jalankan program untuk mengetest hasilnya.

◉ Summary praktek #3

Hingga pembahasan pada praktek submodule, kita telah belajar penerapan module dengan mengaplikasikan beberapa hal berikut:

- Penggunaan normal module dalam binary carete
- Pembuatan module dengan nama `my_io`, dengan isi/item ditulis pada file `my_io.rs`
- Pembuatan module dengan nama `my_number`, dengan isi/item ditulis pada file `my_number/mod.rs`
- Pembuatan submodule dengan nama `my_number/conversion_utility`, dengan isi/item ditulis pada file `my_number/conversion.rs` yang di-import menggunakan `path` attribute.
- Penggunaan keyword `pub` pada fungsi agar bisa diakses dari luar module
- Penggunaan keyword `pub` pada submodule agar bisa diakses dari luar parent module
- Pengaksesan item milik module, yaitu: `my_io::read_entry`, `my_number::conversion_utility::string_to_number`, dan `my_number::is_odd_number`

> *Penerapan notasi penulisan isi module `nama_module.rs` dan `nama_module/mod.rs` bisa diterapkan dalam satu package*

A.20.5. Penerapan keyword `use`

Keyword `use` bisa digunakan untuk meng-import module atau item tertentu, dan dengannya pengaksesan item menjadi lebih pendek. Contohnya:

```
my_number::conversion_utility::string_to_number(message);
```

... bisa dituliskan menjadi ...

```rust
use my_number::conversion_utility::string_to_number;
conversion_utility(message);
```

> *Lebih jelasnya mengenai keyword `use` dibahas pada chapter Module System → Use*

A.20.6. Module `path` attribute

Selain menggunakan dua teknik definisi module item di atas yang fokusnya ada pada penamaan file, ada juga cara lain pendefinisian module item, yaitu dengan memanfaatkan `path` attribute.

Mari kita praktekan, silakan ubah struktur package yang sudah dibuat dari ...

package source code structure

```
my_package
|── Cargo.toml
└── src
     |── main.rs
     |── my_io.rs
     └── my_number
          |── mod.rs                // <----- definisi submodule
conversion_utility
          └── conversion_utility.rs // <----- definisi item/isi
conversion_utility
```

... menjadi ...

package source code structure

```
my_package
|── Cargo.toml
└── src
     |── main.rs
     |── my_io.rs
     └── my_number
          |── mod.rs        // <----- definisi submodule
conversion_utility
          └── conversion.rs // <----- definisi item/isi
conversion_utility
```

Yang telah kita lakukan adalah me-rename file `conversion_utility.rs` menjadi `conversion.rs`. Efeknya akan muncul error dalam pengaksesan module item, karena submodule `my_number/conversion_utility` isi/item-nya harus berada pada file `my_number/conversion_utility.rs` atau `my_number/conversion_utility/mod.rs`. Sedangkan file `conversion.rs` tidak memenuhi kriteria tersebut.

Sekarang buka isi file `my_number/mod.rs`, lalu ubah statement pendefinisian submodule dari ...

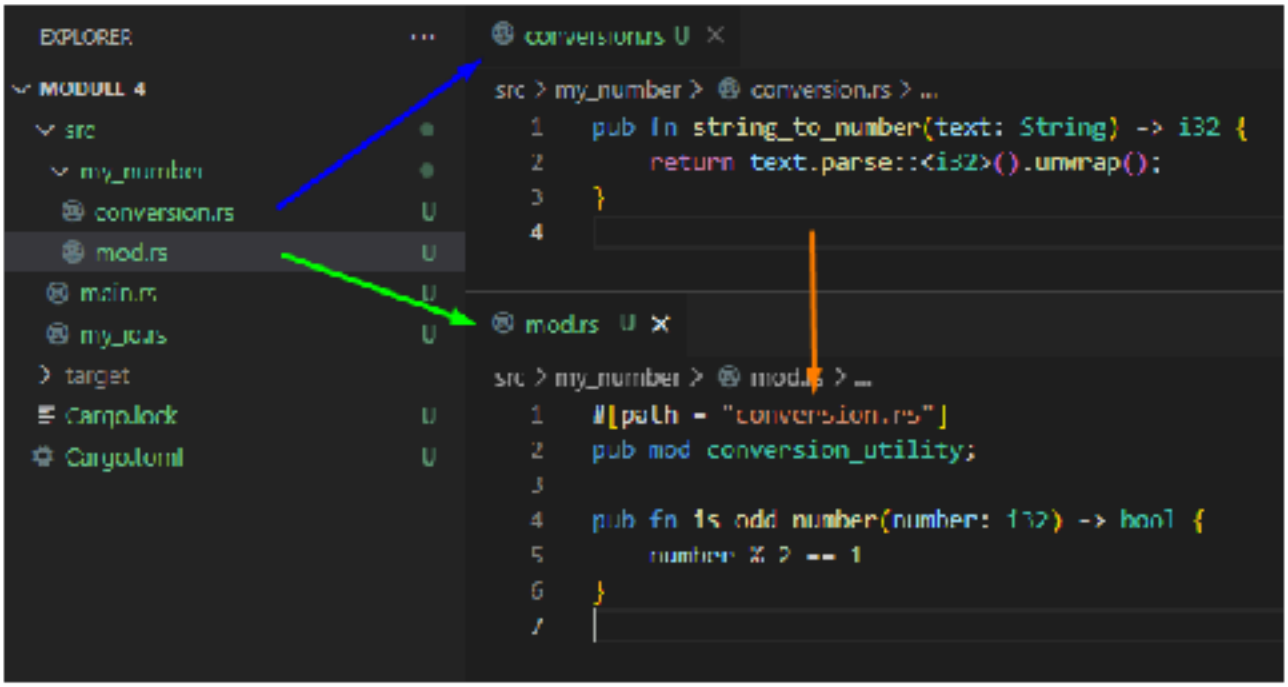
src/my_number/mod.rs

```rust
pub mod conversion_utility;
```

... menjadi ...

src/my_number/mod.rs

```rust
#[path = "conversion.rs"]
pub mod conversion_utility;
```



Lalu run, dan program akan jalan normal tanpa error.

Statement `#[path = "conversion.rs"]` diatas merupakan contoh penerapan dari Rust path attributes. Dengannya kita bisa menempatkan isi sebuah module pada file yang namanya bebas (pada contoh di atas, file bernama `conversion.rs`).

◉ Summary praktek #4

Hingga pembahasan pada praktek module `path` attribute, kita telah belajar penerapan module dengan mengaplikasikan beberapa hal berikut:

- Penggunaan normal module dalam binary carete
- Pembuatan module dengan nama `my_io`, dengan isi/item ditulis pada file `my_io.rs`
- Pembuatan module dengan nama `my_number`, dengan isi/item ditulis pada

file `my_number/mod.rs`

- Pembuatan submodule dengan nama `my_number/conversion_utility`, dengan isi/item ditulis pada file `my_number/conversion_utility/mod.rs`
- Penggunaan keyword `pub` pada fungsi agar bisa diakses dari luar module
- Penggunaan keyword `pub` pada submodule agar bisa diakses dari luar parent module
- Pengaksesan item milik module, yaitu: `my_io::read_entry`, `my_number::conversion_utility::string_to_number`, dan `my_number::is_odd_number`

A.20.7. Pembahasan lanjutan

Pembahasan topik module dilanjutkan pada beberapa chapter lain. Pada chapter Module System → Inline Module kita akan bahas secara mendetail mengenai apa itu inline module dan perbedaannya dibanding normal module. Lalu nantinya di cahpter Module System → Visibility & Privacy akan dibahas secara lengkap mengenai keyword `pub` dan kontrol privasi lainnya di pemrograman Rust.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../module
```

◉ Referensi

- https://doc.rust-lang.org/book/ch07-02-defining-modules-to-control-scope-

and-privacy.html

- https://doc.rust-lang.org/std/keyword.pub.html
- https://doc.rust-lang.org/std/keyword.mod.html
- https://doc.rust-lang.org/rust-by-example/mod.html
- https://aloso.github.io/2021/03/28/module-system.html
- https://stackoverflow.com/questions/69275034/what-is-the-difference-between-use-and-pub-use

---

A.21. Module System → Inline Module

Pada chapter sebelumnya kita telah belajar cara manajemen module. Pembahasan tersebut dilanjut pada chapter ini, yaitu tentang inline module.

A.21.1. Inline Module

Inline module adalah cara lain dalam pendefinisian module beserta item-nya, caranya dengan tetap menggunakan keyword `mod` hanya saja isi/item ditulis didalam blok kode keyword `mod` tersebut. Agar lebih jelas silakan lihat penerapan inline module berikut:

```
mod module_a {

    pub fn fungsi_satu() {
        // ...
    }

    pub mod submodule_b {

        pub const PI: u32 = 3.14;

        pub fn fungsi_dua() {
            // ...
        }
    }
}
```

Inline module cukup mudah diterapkan, yang sebelumnya module dan submodule di-manage dengan mengacu ke penamaan file, dalam inline module cukup tulis saja sebagai blok kode definisi module.

Pada contoh di atas, `module_a` memiliki beberapa item:

- Item berupa fungsi `module_a::fungsi_satu`
- Item berupa submodule `module_a::submodule_b`
- Item berupa konstanta `module_a::submodule_b::PI`
- Item berupa fungsi `module_a::submodule_b::fungsi_dua`

Aturan definisi submodule dan item pada inline module masih sama seperti pada *normal module*, salah satunya adalah agar item bisa diakses dari luar module maka perlu menggunakan keyword `pub`.

A.21.2. Praktek inline module

Mari lanjut proses pembalajaran dengan praktek. Kita akan buat program sederhana yang didalamnya ada proses generate random string, yang kemudian di-hash.

Silakan buat package baru menggunakan `cargo new`. Penulis disini memilih nama `inline_module_1` sebagai nama package. Setelah itu, siapkan fungsi main dan juga module bernama `utilities` dengan penulisan kode menerapkan inline module. O iya, tulis keduanya (`module utilities` dan fungsi `main`) dalam satu file yang sama yaitu `main.rs`.

src/main.rs

```
mod utilities {
```

- Item `utilities::random::string` adalah fungsi nantinya digunakan untuk generate data random string
- Item `utilities::password::hash` adalah fungsi untuk melakukan proses hashing password menggunakan bcrypt
- Item `utilities::password::is_valid` adalah fungsi untuk pengecekan apakah password sama dengan data setelah di-hash

Ketiga fungsi tersebut kemudian diterapkan pada program kecil yang logic-nya ditulis dalam fungsi `main`.

> *Pada contoh di atas, semua item dan sub-item milik `utilities` didefinisikan public menggunakan keyword `pub` agar bisa diakses dari module.*
>
> *Sedangkan module `utilities` sendiri tidak wajib di-expose menggunakan keyword `pub`, karena definisi module-nya berada dalam satu file yang sama dengan fungsi `main`, yaitu file `main.rs`.*
>
> *Lebih jelasnya mengenai keyword `pub` dibahas pada chapter Module System → Visibility & Privacy*

Sebelum lanjut proses koding, silakan tambahkan dulu beberapa dependencies berikut pada `Cargo.toml`, kemudian jalankan `cargo build` di-download.

Cargo.toml

```
[package]
name = "module_inline_1"
version = "0.1.0"
edition = "2021"
```

- Package `bcrypt` adalah crate yang isinya banyak fungsi untuk kebutuhan enkripsi bcrypt
- Package `rand` berisi item untuk keperluan generate data random. Kita sudah beberapa kali menggunakan crate ini.

Ok, sekarang kembali ke source code. Tulis isi fungsi `utilities::random::string` berikut:

src/main.rs

```
mod utilities {

    pub mod random {

        pub fn string(length: u32) -> String {
            use rand::Rng;

            const CHARSET: &[u8] =
"abcdefghijklmnopqrstuvwxyz".as_bytes();
            let mut arr = Vec::new();
            for _ in 0..=length {
                let n =
rand::thread_rng().gen_range(0..(CHARSET.len()));
                let char = CHARSET[n];
                arr.push(char);
            }

            std::str::from_utf8(&arr[..]).unwrap().to_string()
        }
    }

    // ...
}
```

> *Di atas adalah salah satu contoh penerapan generate random string dengan lebar parameterized.*
>
> *Cara tersebut sebenarnya bukan yang paling efisien, namun karena jumlah topik yang kita pelajari masih belum terlalu banyak, penulis menghindari penerapan beberapa hal yang sifatnya baru dan bikin tambah bingung.*

Lanjut, silakan tulis kode untuk hashing password dan juga untuk validasi password berikut:

src/main.rs

```
mod utilities {

    // ...

    pub mod password {

        pub fn hash(text: &str) -> String {
            let result = bcrypt::hash(text,
bcrypt::DEFAULT_COST).unwrap();
            result
        }

        pub fn is_valid(plain: &str, hashed: &str) -> bool {
            let valid = bcrypt::verify(plain, hashed).unwrap();
            valid
        }
    }
}
```

Dalam enkripsi menggunakan bcrypt ada dua hal yang penting diketahui, yang

pertama adalah data yang akan di-hash (pada contoh di atas adalah `text`), dan *cost* atau biaya enkripsi dalam bentuk numerik. Pada praktek ini, `bcrypt::DEFAULT_COST` digunakan sebagai cost enkripsi.

Pengecekan apakah data terenkripsi adalah sama dengan data asli dilakukan menggunakan fungsi `bcrypt::verify`.

Kedua fungsi `bcrypt::hash` dan `bcrypt::verify` menghasilkan data bertipe *generic result type* atau `Result<T, E>`. Tipe ini memiliki method bernama `unwrap` yang berguna untuk pengambilan result atau nilai.

> *Lebih jelasnya tentang result type dibahas pada chapter Tipe Data → Result*

Oke, sekarang fungsi utility sudah siap, mari implementasikan semua fungsi tersebut pada `main`.

Silakan tulis kode berikut. Sebuah variabel bernama `password` dibuat, diisi dengan kombinasi dari string dan juga random string. Lalu data password tersebut di-hash kemudian dicek nilainya.

src/main.rs

```
fn main() {
    let password = format!("zereth mortis {}",
utilities::random::string(10));
    println!("raw password: {}", password);

    let hashed = utilities::password::hash(&password);
    println!("hashed password: {}", hashed);

    let is_valid = utilities::password::is_valid(&password,
```

Jalankan program untuk melihat hasilnya.

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\module_inline_1> cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.03s
     Running `target\debug\module_inline_1.exe`
raw password: zereth morlis tcyaislzbmy
hashed password: $2b$12$CB/97gOsFkbBQzVAPNC7k.mthcXThcpW3AhpWE.BcCo2zXOzoSoam
is password matched? true
```

Statement `utilities::random::string(10)` menghasilkan random string dengan lebar `10`, sesuai kebutuhan. Hasil dari pemanggilan fungsi tersebut di-concat dengan text kemudian ditampung variabel `password`.

Variabel `password` kemudian di-hash menggunakan `utilities::password::hash`, kemudian dibandingkan hasil hash-nya apakah cocok dengan text aslinya menggunakan `utilities::password::is_valid`.

A.21.3. Inline module item

Module item dalam inline module masih sama seperti pada module normal, yaitu adalah apapun yang didefinisikan didalam sebuah module. Bisa berupa fungsi, konstanta, submodule, dan lainnya. Pada contoh di atas ada beberapa item didefinisikan:

- Item `utilities::random` yang merupakan sebuah submodule (inline module).
- Item `utilities::random::string` yang merupakan sebuah fungsi.
- Item `utilities::password` yang merupakan sebuah submodule (inline module).
- Item `utilities::password::hash` yang merupakan sebuah fungsi.
- Item `utilities::password::is_valid` yang merupakan sebuah fungsi.

> *Lebih jelasnya tentang macam-macam item dibahas pada chapter Module System → Path & Item*

A.21.4. Inline module file `nama_module.rs` atau `nama_module/mod.rs`

Dalam *real project* hampir tidak mungkin semua kode ditulis di file `main.rs`. Pastinya kode akan di-split menjadi banyak module sesuai kebutuhan.

Kode sebelumya, akan kita refactor. Module `utilities` yang berada di `main.rs` perlu dipindah ke file baru yang sesuai dengan aturan penulisan module, yaitu `nama_module.rs` atau `nama_module/mod.rs`. Disini penulis memilih `nama_module.rs`, jadi silakan buat file bernama `utilities.rs`, lalu pindah isi itemnya kesana (statement definisi inline module `utilities` tidak perlu ikut dipindah, hanya isinya saja, seperti normalnya definisi module).

O iya, jangan lupa untuk menambahkan statement `mod utilities` pada `main.rs`, agar module terdaftar dan bisa digunakan.

main.rs U ×

module_inline_2 > src > main.rs > ...

```rust
mod utilities;

Run | Debug
fn main() {
    let password: String = format!("zeneth mortis {}", utilities::random::string(10));
    println!("raw password: {}", password);

    let hashed: String = utilities::password::hash(text: &password);
    println!("hashed password: {}", hashed);

    let is_valid: bool = utilities::password::is_valid(plain: &password, &hashed);
    println!("is password matched? {}", is_valid);
}
```

utilities.rs U ×

module_inline_2 > src > utilities.rs > ...

```rust
pub mod random {

    pub fn string(length: u32) -> String {
        use rand::Rng;

        const CHARSET: &[u8] = "abcdefghijklmnopqrstuvwxyz".as_bytes();
        let mut arr: Vec<u8> = Vec::new();
        for _ in 0..=length {
            let n: usize = rand::thread_rng().gen_range(0..(CHARSET.len()));
            let char: u8 = CHARSET[n];
            arr.push(char);
        }

        std::str::from_utf8(&arr[..]).unwrap().to_string()
    }
}

pub mod password {

    pub fn hash(text: &str) -> String {
        let result: String = bcrypt::hash(password: text, bcrypt::DEFAULT_COST).unwrap();
        result
    }

    pub fn is_valid(plain: &str, hashed: &str) -> bool {
        let valid: bool = bcrypt::verify(password: plain, hash: hashed).unwrap();
        valid
    }
}
```

Jalankan program untuk melihat hasilnya.

Pada contoh ke-2 ini modul `utilities` didefinisikan menggunakan cara biasa, sedang isi dari module tersebut didefinisikan menggunakan teknik inline module.

A.21.5. Inline module `path` attribute

`path` attribute bisa digunakan dalam manajemen inline module. Silakan utak-atik kode yang sudah dibuat dengan mengikuti step berikut:

1. Buat file baru bernama `utilities/password.rs`, kemudian pindah isi definisi module `password` ke file tersebut.
2. Buat file baru bernama `utilities/random.rs`, kemudian pindah isi definisi module `random` ke file tersebut.
3. Hapus file `utilities.rs`, pindah definisi module `utilities` ke file `main.rs`, hasilnya kurang lebih seperti ini:

src/main.rs

```
mod utilities {
    #[path = "random.rs"]
    pub mod random;

    #[path = "password.rs"]
    pub mod password;
}
```

```rust
mod utilities {
    #[path = "random.rs"]
    pub mod random;

    #[path = "password.rs"]
    pub mod password;
}

fn main() {
    let password: String = format!("zareth mortis {}", utilities::random::string(10));
    println!("raw password: {}", password);

    let hashed: String = utilities::password::hash(text: &password);
    println!("hashed password: {}", hashed);

    let is_valid: bool = utilities::password::is_valid(plain: &password, &hashed);
    println!("is password matched? {}", is_valid);
}
```

```rust
pub fn hash(text: &str) -> String {
    let result: String = bcrypt::hash(password: text, bcrypt::DEFAULT_COST).unwrap();
    result
}

pub fn is_valid(plain: &str, hashed: &str) -> bool {
    let valid: bool = bcrypt::verify(password: plain, hash: hashed).unwrap();
    valid
}
```

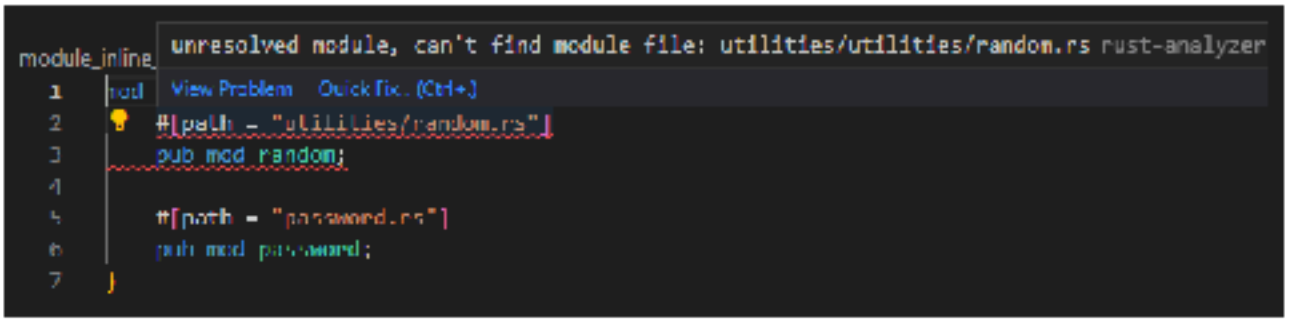
```rust
pub fn string(length: u32) -> String {
    use rand::Rng;

    const CHARSET: &[u8] = "abcdefghijklmnopqrstuvwxyz".as_bytes();
    let mut arr: Vec<u8> = Vec::new();
    for _ in 0..length {
        let n: usize = rand::thread_rng().gen_range(0..(CHARSET.len()));
        let char: u8 = CHARSET[n];
        arr.push(char);
    }
```

Jalankan program untuk melihat hasilnya.

Pada kode di atas, `path` yang digunakan bukan `utilities/random.rs` melainkan `random.rs`, hail ini dikarenakan `path` attribute dipanggil **di dalam module `utilities`**, menjadikan current path pada blok kode tersebut menjadi

`utilities/`.

Silakan coba ubah isi `path` attribute menjadi `utilities/random.rs`, hasilnya adalah error. Rust akan menggunakan gabungan dari current path (`utilities/`) dan path pada `path` attribute (`utilities/random.rs`) dalam lookup, jadinya yang di-lookup adalah `utilities/utilities/random.rs`, dan hasilnya error karena tidak ada file disana.



Module system di Rust merupakan topik yang cukup membingungkan (menurut penulis) karena desain-nya yang cukup unik jika dibandingkan dengan bagaimana bahasa pemrograman lain mengelola module-nya. Penulis anjurkan untuk mencoba praktek membuat program lainnya yang menerapkan module system, agar cepat terbiasa.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-
example/../module_inline
```

◉ Referensi

- https://doc.rust-lang.org/book/ch07-02-defining-modules-to-control-scope-and-privacy.html
- https://doc.rust-lang.org/std/keyword.mod.html
- https://doc.rust-lang.org/rust-by-example/mod.html
- https://aloso.github.io/2021/03/28/module-system.html
- https://stackoverflow.com/questions/69275034/what-is-the-difference-between-use-and-pub-use

---

A.22. Module System → Scope & Akses Item

Pembahasan chapter ini masih dalam lingkup module system, yaitu tentang scope dan pengakesan module item.

A.22.1. Scope

Scope bisa diartikan dengan: representasi dimana kode berada. Apapun yang ditulis dalam blok kode (ditandai dengan diapit tanda kurung kurawal `{ }`) berarti dalam satu scope yang sama.

Agar lebih jelas, lihat kode berikut kemudian pelajari penjelasan dibawahnya:

```
const PI: f64 = 3.14;

fn main() {
    my_func();
}

fn my_func() {
    let nama = "Sylvanas Windrunner";
    let occupation = "ex-Warchief of the Horde";

    // ...
}
```

- Konstanta `PI`, fungsi `main`, dan juga fungsi `my_func` di definisikan satu

level dan berada di scope terluar (yang pada ebook ini disebut sebagai *root*).

- Statement pemanggilan fungsi `my_func()` berada dalam scope blok kode fungsi `main`.
- Variabel `nama` dan `occupation` berada pada scope blok kode fungsi `my_func`.

> *Sebenarnya pembahasan mengenai `scope` harus diiringi dengan pembahasan tentang Block expression, namun karena kita belum mempelajarinya, chapter ini pembahasan hanya akan fokus pada penerapan scope yang berhubungan dengan* ***module scope****.*

A.22.2. Module Scope

Module scope adalah scope untuk module. Apa bedanya dengan scope secara umum? Silakan perhatikan kode berikut terlebih dahulu:

```
mod my_module {

    const event_one: &str = "Siege of Ogrimmar";

    mod my_submodule {

        const event_two: &str = "Battle for Azeroth";

        fn func_two() {
            const event_three: &str = "Sepulcher of the First
Ones";
        }
    }
}
```

Pada kode di atas:

- Konstanta `event_one` berada dalam module scope `my_module`.
- Module `my_submodule` adalah sebuah inline module yang berada dalam module scope `my_module`. Module `my_submodule` berada dalam satu scope yang sama dengan variabel `event_one`.
- Konstanta `event_two` berada dalam module scope `my_submodule`.
- Fungsi `func_two` berada dalam module scope `my_submodule`.
- Konstanta `event_three` **scope**-nya adalah dalam blok fungsi `func_two`. Sedangkan **module scope**-nya adalah dalam module scope `my_submodule`. Jadi kelihatan ya bedanya.

Pemahaman tentang module scope ini penting karena akan berhubungan dengan apa yang akan dipelajari di *section* berikutnya.

A.22.3. Keyword `self` dan *crate root*

Keyword `self` merepresentasikan current module scope. Dengannya kita bisa mengakses item yang deklarasinya ada pada module scope yang sama.

> *Sebenarnya selain penjelasan di atas, keyword `self` juga digunakan untuk hal lain yaitu sebagai receiver method.*
>
> *Pembahasan tentang penerapan keyword `self` sebagai receiver method dibahas pada chapter Method*

Selanjutnya, apa itu *crate root*? **Crate root** adalah apapun yang didefinisikan di file entrypoint crate (yaitu `src/main.rs` untuk binary crate, dan `src/lib.rs`

untuk library crate). **Crate root** adalah module scope yang berada di root (paling atas). Apapun yang berada di *crate root* berarti berada dalam module scope yang sama.

Sebagai contoh, pada kode program berikut, module `my_mod` dan fungsi `main` dideklarasikan di *crate root* `src/main.rs`. Artinya kedua item tersebut berada dalam satu module scope yang sama, yaitu di *crate root*.

```
mod my_mod {
    pub fn run_the_app(note: &str) {
        println!("calling `my_mod::run_the_app()`. note {}",
note);
    }
}

fn main() {
    my_mod::run_the_app("1st call");
    self::my_mod::run_the_app("2nd call");
}
```

```
(base) PS D:\Labs\Adam Studio\module_scope_item_access_1> cargo run
   Compiling module_scope_item_access_1
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\module_scope_item_access_1.exe`
calling `my_mod::run_the_app()`. note 1st call
calling `my_mod::run_the_app()`. note 2nd call
```

Pada fungsi `main`, ada dua statement pemanggilan module item `my_mod::run_the_app`. Kedua statement tersebut adalah mirip, pembedanya ada pada statement ke-2, yaitu keyword `self` digunakan di segment path sebagai prefix.

Keyword `self` menginstruksi program bahwa item yang dipanggil adalah item yang module scope-nya sama.

Pada contoh di atas, module `my_mod` berada dalam satu scope yang sama dengan fungsi `main`, yaitu di *crate root*. Karena itu, `my_mod` bisa diakses dengan menggunakan keyword `self`, atau lansung panggil saja nama module seperti biasanya, `my_mod()`.

Bisa dibilang keyword `self` ini opsional dalam penggunaannya, boleh ikut dituliskan dan boleh juga tidak.

```
my_mod::run_the_app("1st call");
self::my_mod::run_the_app("2nd call");
```

A.22.4. Keyword `self` pada module scope

Lanjut ke praktek berikutnya. Masih tentang keyword `self`, kita akan gunakan keyword ini untuk mengakses item dalam module.

Silakan tulis kode berikut kemudian jalankan.

```
fn my_func() {
    println!("calling `my_func()`");
}

mod my_mod {

    pub fn my_func() {
        println!("calling `my_mod::my_func()`");
    }

    pub fn run_the_app() {
        println!("calling `my_mod::run_the_app()`");
```

Pada contoh bisa dilihat ada dua buah fungsi dideklarasikan dengan nama yang sama persis, yang satu berada di *crate root*, satunya lagi merupakan item milik `my_mod`.

Didalam `my_mod::run_the_app` ada 2 kali pemanggilan fungsi `my_func`, satunya menggunakan keyword `self` dan satunya tidak. Fungsi `my_func` manakah yang dipanggil? Hasilnya bisa dilihat pada gambar dibawah ini.

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrograman-rust\module_scope_item_access_2> cargo run
   Compiling module_scope_item_access_2 v0.1.0
warning: function is never used: `my_func`
 --> src\main.rs:1:4
  |
1 | fn my_func() {
  |    ^^^^^^^
  |
  = note: `#[warn(dead_code)]` on by default
warning: `module_scope_item_access_2` (bin "module_scope_item_access_2") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.60s
     Running `target\debug\module_scope_item_access_2.exe`
calling `my_mod::run_the_app()`
calling `my_mod::my_func()`
calling `my_mod::my_func()`
```

Kedua statement `my_func()` dan `self::my_func()` dalam `my_mod::run_the_app` adalah mengarah ke fungsi yang sama, yaitu `my_mod::my_func`.

Jadi kesimpulan dari penerapan keyword `self` dalam module item adalah sama seperti penerapannya pada *crate root*, yaitu boleh dipakai boleh tidak. Keyword `self` adalah opsional.

Lalu bagaimana cara memanggil `my_func` yang berada di *crate root*, dengan pemanggilan adalah dari dalam module item? Caranya adalah menggunakan keyword `crate`.

A.22.5. Keyword `crate` pada module scope

Keyword ini digunakan untuk mengakses apapun yang ada di *crate root*.

Pada contoh diatas, fungsi `my_func` yang berada di *crate root* bisa dipanggil dari fungsi `main` dengan statement `my_func()` atau `self::my_func()`. Untuk bagian ini penulis rasa sudah cukup jelas.

Beda cerita kalau fungsi tersebut dipanggil dari dalam module item `run_the_app`. Kedua statement `my_func()` dan `self::my_func()` mengarah ke `my_mod::my_func`, bukan ke fungsi `my_func` di *crate root*. Hal ini karena **current module scope** dalam statement `run_the_app` adalah module `my_mod`, maka pemanggilan `my_func` tanpa `self` atau dengannya mengarah ke fungsi yang sama yaitu `my_mod::my_func`.

Pada *section* ini kita akan belajar tentang keyword baru, yaitu `crate`. Keyword tersebut digunakan untuk mengakses apapun yang berada di *crate root*. Kita bisa memanfaatkannya untuk mengakses `my_func` yang berada di *crate root* dengan pengaksesan dari dalam module item.

Silakan ubah kode diatas menjadi seperti berikut. Perubahannya ada pada statement dalam fungsi `run_the_app`.

```
fn my_func() {
    println!("calling `my_func()`");
}

mod my_mod {
```

Jalankan program, hasilnya berbeda dengan eksekusi program sebelumnya. Statement `crate::my_func()` dalam fungsi `run_the_app` mengarah ke fungsi `my_func` di root, sedangkan `self::my_func` mengarah ke `my_mod::my_func`.

```
(base) PS D:\Labs\Adam Studio\module_scope_item_access_3> cargo run
   Compiling module_scope_item_access_1 v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in 0.83s
     Running `target\debug\module_scope_item_access_3.exe`
calling `my_mod::run_the_app()`
calling `my_func()`
calling `my_mod::my_func()`
```

A.22.6. Keyword `super`

Selain `self` dan `crate` ada juga keyword `super`, yang gunanya adalah untuk mengakses *parent* module scope atau 1 scope diatas *current* module scope. Agar lebih jelas silakan pelajari kode berikut:

```rust
fn my_func() {
    println!("calling `my_func()`");
}

mod my_mod {

    pub fn my_func() {
        println!("calling `my_mod::my_func()`");
    }

    pub mod my_submod {

        pub fn my_func() {
            println!("calling `my_mod::my_submod::my_func()`");
        }
```

Ada 3 buah fungsi `my_func` dideklarasikan:

- Fungsi `my_func` yang berada di *crate root*.
- Fungsi `my_func` yang merupakan module item milik `my_mod`.
- Fungsi `my_func` yang merupakan module item milik submodule `my_submod`.

Dalam fungsi `run_the_app`, ketiga fungsi dengan nama tersebut dipanggil.

- Statement `crate::my_func()` akan mengarah ke fungsi `my_func` yang ada di *crate root*.
- Statement `super::my_func()` akan mengarah ke fungsi `my_func` yang ada di parent module scope, yaitu `my_mod::my_func`.
- Statement `self::my_func()` akan mengarah ke fungsi `my_func` yang ada di current module scope, yaitu `my_mod::my_submod::my_func`.

```
(base) PS D:\Labs\Adam Studio\module_scope_item_access_4> cargo run
   Compiling module_scope_item_access_4 v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in 0.48s
     Running `target\debug\module_scope_item_access_1.exe`
calling `my_mod::my_submod::run_the_app()`
calling `my_func()`
calling `my_mod::my_func()`
calling `my_mod::my_submod::my_func()`
```

Semoga cukup jelas ya. Silakan ulangi terus praktek diatas jika perlu, agar makin paham.

O iya, keyword `super` ini bisa digunakan banyak kali sesuai kebutuhan. Setiap kali keyword ditulis sebagai segment path, maka artinya 1 level parent module scope.

Jika mengacu ke contoh program di atas, maka kedua statement berikut adalah ekuivalen.

```
crate::my_func();
super::super::my_func();
```

A.22.7. Praktek lanjutan

Agar pemahaman makin mantab, silakan pelajari program berikut.

```
fn my_func() {
    println!("call `my_func()`");
}

mod module_a {

    // path item → `module_a::my_func`.
    pub fn my_func() {
        println!("call `module_a::my_func()`");
    }
}

mod module_b {

    // path item → `module_b::submodule_b_one`.
    mod submodule_b_one {

        // path item → `module_b::submodule_b_one::my_func`.
        pub fn my_func() {
            println!("call
`module_b::submodule_b_one::my_func()`");
        }
    }

    // path item → `module_b::submodule_b_two`.
    mod submodule_b_two {
```

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-
example/../module_scope_item_access
```

◉ Referensi

- https://doc.rust-lang.org/book/ch07-02-defining-modules-to-control-scope-and-privacy.html
- https://doc.rust-lang.org/std/keyword.self.html
- https://doc.rust-lang.org/std/keyword.super.html
- https://doc.rust-lang.org/std/keyword.crate.html
- https://doc.rust-lang.org/rust-by-example/mod.html
- https://doc.rust-lang.org/rust-by-example/meta/doc.html
- https://aloso.github.io/2021/03/28/module-system.html

A.23. Struct

Pada chapter ini kita akan belajar tentang struct.

A.23.1. Konsep Struct

Struct (kependekan dari *structure*) adalah tipe data custom yang dengannya kita bisa mengumpulkan beberapa definisi tipe data lalu menjadikannya sebagai satu buah tipe data dalam struktur tertentu.

Contoh analogi seperti sebuah mobil. Mobil memiliki roda, mesin, tempat kemudi, dan banyak lainnya. Mobil pada contoh ini adalah struct, sedang isi dari mobil tersebut biasa disebut dengan *attribute* atau *property* atau *field*.

Selain property, mobil juga bisa melakukan aksi, contohnya seperti jalan, belok kanan, berhenti. Aksi tersebut pada contoh ini adalah yang disebut dengan *method*. Method sendiri adalah fungsi yang merupakan property sebuah struct.

> *Chapter ini fokus pada pembahasan struct beserta property-nya. Topik tentang method dibahas pada chapter selanjutnya, yaitu Method.*

A.23.2. Pembuatan struct

Keyword `struct` digunakan untuk membuat custom type struct. Notasi penulisannya seperti ini:

```
struct NamaStruct {
    property_satu: String,
    property_dua: u64,
    // ...
}
```

Contoh, sebuah struct bernama `User` yang memiliki 4 buah property:

- `name` bertipe `String`
- `sign_in_count` bertipe numerik `u64`
- `affliation` bertipe vektor `Vec<String>`
- `active` bertipe boolean

```
struct User {
    name: String,
    sign_in_count: u64,
    affliation: Vec<String>,
    active: bool,
}
```

Struct adalah *hanya definisi structure*-nya saja, struct tidak menampung nilai atau value. Contoh pada struct `User` di atas bisa dilihat bahwa yang didefinisikan hanya nama struct dan property (beserta tipe datanya). Tidak ada pengisian nilai sama sekali.

> *Notasi `Vec<String>` merupakan salah satu contoh penerapan generics. Topik ini nantinya dibahas lebih detail pada chapter Generics.*

Struct merupakan tipe data data custom, yang berarti tipe data tersebut bisa digunakan dalam pembuatan variabel. Sebagai contoh dibawah ini, sebuah variabel bernama `user_one` didefinisikan dengan tipe adalah struct `User` yang

telah dibuat.

```rust
struct User {
    name: String,
    sign_in_count: u64,
    affliation: Vec<String>,
    active: bool,
}

fn main() {
    let user_one = User{
        name: String::from("Orgrim Doomhammer"),
        sign_in_count: 12,
        affliation: vec![
            String::from("Warchief of the Horde"),
            String::from("Blackrock Chieftain"),
            String::from("The Doomhammer"),
        ],
        active: false,
    };

    println!("name: {}", user_one.name);
    println!("sign-in count: {}", user_one.sign_in_count);
    println!("affliation: {:?}", user_one.affliation);
    println!("is active? {}", user_one.active);
}
```

Jalankan program diatas, lihat hasilnya.

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.45s
     Running `target\debug\struct_example.exe`
name: Orgrim Doomhammer
sign in count: 12
affliation: ["Warchief of the Horde", "Blackrock Chieftain", "The Doomhammer"]
is active? false
```

Variabel `user_one` didefinisikan tipe datanya adalah struct `User`. Notasinya penulisannya:

```
let user_one = User{
    // property_satu: value,
    // property_dua: value,
    // ...
};
```

Deklarasi variabel bertipe data struct mewajibkan isi masing-masing property harus dituliskan beserta nilainya. Sebagai contoh variabel `user_one` yang bertipe struct `User` di atas, dalam statement bisa dilihat ke-empat property milik struct `User` dituliskan beserta nilainya.

```
let user_one = User{
    name: String::from("Orgrim Doomhammer"),
    sign_in_count: 12,
    affliation: vec![
        String::from("Warchief of the Horde"),
        String::from("Blackrock Chieftain"),
        String::from("The Doomhammer"),
    ],
    active: false,
};
```

> *Variabel yang tipe data-nya adalah struct biasa disebut dengan object atau instance.*

◉ Fungsi `String::from()`

Pada chapter Module System → Path & Item kita telah sedikit mengenal fungsi

`String::new()` yang kegunaannya adalah untuk membuat data string kosong bertipe data `String`.

Selain fungsi tersebut, ada juga fungsi `String::from()` yang kegunaannya juga untuk pembuatan data string bertipe `String`, tapi dengan isi ditentukan lewat argumen pemanggilan fungsi.

```
let a = "";
// variabel `a` bertipe data `&str`, isi value-nya ""

let b = String::new();
// variabel `b` bertipe data `String`, isi value-nya ""

let c = "hello";
// variabel `c` bertipe data `&str`, isi value-nya "hello"

let d = String::from("world");
// variabel `d` bertipe data `String`, isi value-nya "world"
```

Untuk sekarang, boleh menggunakan kesimpulan berikut:

- Hasil dari `String::new()` adalah sama dengan literal string `""` tapi bertipe data `String` (bukan `&str`)
- Hasil dari `String::from("hello")` adalah sama dengan literal string `"hello"` tapi bertipe data `String` (bukan `&str`)

Sebenarnya perbedaannya tidak hanya itu saja, nantinya kita pelajari lebih detail pada chapter String Literal (&str) vs. String Custom Type.

◉ *Naming convention* struct

Sesuai anjuran di halaman dokumentasi Rust, upper camel case digunakan dalam penamaan struct dan camel case untuk penamaan property-nya.

Contoh:

```
struct User {
    name: String,
    sign_in_count: u64,
    affliation: Vec<String>,
    active: bool,
}
```

A.23.3. Mutable struct

Penentuan *mutability* sebuah struct dilakukan dengan cara menambahkan keyword `mut` seperti pada umumnya variabel. Contohnya bisa dilihat pada variabel `user_two` berikut:

```
let mut user_two: User = User{
    name: String::from("Varian Wrynn"),
    sign_in_count: 12,
    affliation: vec![
        String::from("High King of the Alliance"),
        String::from("King of Stormwind"),
        String::from("Champion of the Crimson Ring"),
    ],
    active: false,
};

user_two.name = String::from("Anduin Wrynn");
user_two.affliation.pop();
user_two.active = true;

println!("name: {}", user_two.name);
println!("sign-in count: {}", user_two.sign_in_count);
println!("affliation: {:?}", user_two.affliation);
```

Pada contoh di atas, `user_two` adalah object struct yang bisa diubah nilainya (mutable). Perubahan nilai struct bisa dilakukan pada property, contohnya seperti property `name`, `affliation`, dan `active` yang dicontohkan diubah nilainya.

Atau bisa juga perubahannya pada value struct itu sendiri, diganti dengan value baru.

```
let mut user_two = User{
    name: String::from("Varian Wrynn"),
    sign_in_count: 12,
    affliation: vec![
        String::from("High King of the Alliance"),
        String::from("King of Stormwind"),
        String::from("Champion of the Crimson Ring"),
    ],
    active: false,
};

user_two = User{
    name: String::from("Anduin Wrynn"),
    sign_in_count: 12,
    affliation: vec![
        String::from("High King of the Alliance"),
        String::from("King of Stormwind"),
    ],
    active: true,
};
```

A.23.4. Macam-macam notasi deklarasi variabel struct

◉ *type inference / manifest typing*

Metode deklarasi *type inference* ataupun *manifest typing* bisa digunakan dalam variabel struct.

```
struct Car {
    brand: String,
    model: String,
}

let car_one = Car{
    brand: String::from("Toyota"),
    model: String::from("Sprinter Trueno AE86"),
};

let car_two: Car = Car{
    brand: String::from("BMW"),
    model: String::from("M3 GTR"),
};
```

◉ Variabel struct tanpa *predefined value*

Variabel struct boleh didefinisikan tanpa *predefined value*, jadi cukup tipe datanya saja yang ditentukan saat deklarasi variabel. Contoh:

```
let mut car_three: Car;
car_three = Car{
```

◉ Variabel struct dengan nilai berasal dari struct lain

Jika ada kebutuhan untuk membuat variabel object struct yang nilai property-nya sebagian adalah berasal dari variabel struct lain, bisa manfaatkan syntax `..`.

```
let mut car_three: Car;
car_three = Car{
    brand: String::from("Audi"),
    model: String::from("Le Mans Quattro"),
};
println!("{} {}", car_three.brand, car_three.model);

let mut car_four: Car;
car_four = Car{
    brand: String::from("Audi Brand"),
    ..car_three
};
println!("{} {}", car_four.brand, car_four.model);
```

Pada contoh di atas, variabel `car_four` property `brand`-nya diisi dengan `"Audi Brand"`. Sedangkan property-property lainnya nilai didapat dari nilai propery milik `car_three`.

```
warning: `struct_example` (bin "struct_example") generated 3 warnings
    Finished dev [unoptimized + debuginfo] target(s) in 0.4s
     Running `target\debug\struct_example.exe`

Audi Le Mans Quattro
Audi Brand Le Mans Quattro
```

Coba jalankan program, akan terlihat nilai dari variabel `car_four` untuk property selain `brand` adalah sama dengan nilai property-property `car_three`.

◉ *Field init shorthand*

Jika ada variabel yang namanya sama persis dengan nama property sebuah struct, maka dalam deklarasi variabel struct bisa menggunakan teknik penulisan seperti berikut:

```
let model = String::from("Corvette C1");

let car_five = Car{
    brand: String::from("Chevrolet"),
    model,
};
```

Cukup tulis nama variabelnya saja tanpa value.

Teknik *shorthand* ini bisa juga digunakan dalam fungsi. Contoh dibawah ini ada fungsi bernama `new_car` yang memiliki nama parameter adalah sama persis dengan nama property struct `Car`.

```
fn new_car(brand: String, model: String) -> Car {
    Car{
        brand,
        model,
    }
}

fn main() {
    let car_six = new_car(
        String::from("Chevrolet"),
        String::from("Corvette C6")
    );
    // ...
```

◉ Deklarasi nilai struct secara horizontal

Umumnya, operasi *assignment* nilai struct dituliskan secara vertikal. Untuk struct yang property-nya sedikit biasanya dituliskan secara horizontal contohnya seperti di bawah ini:

```
struct Point {
    x: f32,
    y: f32,
}

let point_one = Point { x: 3.14, y: 8.0 };
```

◉ *Destructuring assignment*

Teknik penulisan ini bisa dipakai dalam case dimana nilai property struct perlu ditampung ke variabel baru. Contoh:

```
let point_one = Point { x: 3.14, y: 8.0 };

let Point { x: x_one, y: y_one } = point_one;
println!("x_one: {}", x_one);
println!("y_one: {}", y_one);
```

- Variabel `x_one` di atas akan menampung nilai dari `point_one.x`
- Variabel `y_one` di atas akan menampung nilai dari `point_one.y`

Jika tidak semua property struct perlu untuk ditampung ke variabel baru, maka gunakan `_` untuk property yang nilainya tidak ditampung. Contoh:

```
let point_one = Point { x: 3.14, y: 8.0 };

let Point { x: _, y: y_one } = point_one;
println!("y_one: {}", y_one);
```

A.23.5. *Unit-like structs*

Unit-like structs adalah struct yang didefinisikan tanpa property. Cara deklrasinya bisa dilihat pada contoh berikut:

```
struct StructOne;

let data_one = StructOne;
```

Teknik pembuatan struct ini berguna ketika ada case dimana ada kebutuhan untuk mengimplementasikan sebuah trait ke suatu tipe data. Lebih jelasnya akan dibahas pada chapter Traits.

A.23.6. Debugging value struct menggunakan `#[derive(Debug)]`

By default, error akan muncul saat berusaha menampilkan nilai variabel struct (bukan nilai property-nya, tapi nilai variabel struct-nya) menggunakan macro `println`. Ini disebabkan karena data yang bisa ditampilkan menggunakan macro `println` harus memiliki trait `Debug` (atau `Display`).

```
let console_one: GamingConsole

Go to GamingConsole

`GamingConsole` doesn't implement `Debug`
the trait `Debug` is not implemented for `GamingConsole`
add `#[derive(Debug)]` to `GamingConsole` or manually `impl Debug for
GamingConsole` rustc(E0277)

macros.rs(186, 28): Error originated from macro call here

main.rs(130, 5): Error originated from macro call here

main.rs(124, 5): consider annotating `GamingConsole` with `#[derive(Debug)]`:
[derive(Debug)]
`

struct Gaming
    name: Str
}
let console_
    name: Str
};
println!("data_struct_one: {:#?}", console_one);
```

Lalu jika ingin melihat nilai property struct bagaimana solusinya? Bisa dengan menampilkan nilai property satu per satu, tapi pastinya butuh effort.

Cara yang lebih elegan adalah dengan menggunakan atribute `#[derive(Debug)]`. Penerapannya cukup dengan menuliskan atribut tersebut tepat diatas deklarasi struct. Contoh:

```rust
#[derive(Debug)]
struct GamingConsole {
    name: String
}

let console_one = GamingConsole{
    name: String::from("PlayStation 5"),
};

println!("data_struct_one: {:#?}", console_one);
```

```
113     #[derive(Debug)]
114     struct GamingConsole {
115         name: String
116     }
117     let console_one: GamingConsole = GamingConsole{
118         name: String::from("PlayStation 5"),
119     };
120     println!("data_struct_one: {:#?}", console_one);
```

```
PROBLEMS 4   OUTPUT   CODELENS   JUPYTER   DEBUG CONSOLE   TERMINAL

warning: `struct_example` (bin "struct_example") generated 4 warnings
    Finished dev [unoptimized + debuginfo] target(s) in 0.68s
     Running `target\debug\struct_example.exe`

data_struct_one: GamingConsole {
    name: "PlayStation 5",
}
```

> - *Lebih jelasnya mengenai traits dibahas pada chapter Traits*

A.23.7. Tuple struct

Tuple struct adalah struct yang didefinisikan dengan gaya tuple. Property pada tuple struct diakses menggunakan notasi pengaksesan tuple item.

Pada contoh berikut tuple struct `Color` didefinisikan dengan isi 3 buah item yang kesemuanya bertipe data `i32`. Lalu tipe data tuple struct tersebut digunakan untuk membuat variabel baru bernama `red`.

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

let red = Color(255, 0, 0);

println!("{:?} {:?} {:?}", red.0, red.1, red.2);
```

Bisa dilihat di contoh, property diakses menggunakan nomor indeks dengan notasi penulisan `variable.index`.

Contoh lainnya, tuple struct `SomeTupleStruct` berikut dideklarasikan memiliki item ke-1 bertipe `i32` dan item ke-2 bertipe boolean.

```
struct SomeTupleStruct(i32, bool);

let some_data = SomeTupleStruct(0, false);

println!("{:?} {:?}", some_data.0, some_data.1);
```

A.23.8. Struct property visibility

Sekarang mari kita coba praktekan satu program lagi, namun kali ini sedikit berbeda, struct kita definisikan sebagai module item.

Silakan siapkan program dengan struktur seperti berikut:

package source code structure

```
my_package
|── Cargo.toml
└── src
    |── main.rs
    ├── models.rs
    └── models
        └── game.rs
```

Lalu isi file program dengan kode di bawah ini. Sebuah struct dengan path `models::game::GamingConsole`, dan satu buah object di fungsi `main` yang

dibuat dari struct tersebut. Pastikan definisi semua module item adalah publik (ada keyword `pub`).

src/main.rs

```
mod models;

fn main() {
    let ps5 = models::game::GamingConsole{
        name: String::from("PS 5")
    };
    println!("{:#?}", red);
}
```

src/models.rs

```
pub mod game;
```

src/models/game.rs

```
#[derive(Debug)]
pub struct GamingConsole {
    name: String
}
```

Ada yang aneh, baris kode pembuatan variabel object `ps5` terdeteksi sebagai error. Padahal definisi submodule dan item-nya sudah publik semua.

```
struct_2::models::game::GamingConsole
name: String
Go to String
field `name` of struct `GamingConsole` is private
private field rustc(E0451)
View Problem  Quick Fix... (Ctrl+.)
name: String::from("PS 5")
```

Jika dilihat dari keterangan error, sebenarnya cukup jelas bagian mana yang menjadi sumber masalah, yaitu field `name` yang terdeteksi sebagai private property.

Struct jika didefinisikan di file yang sama dengan statement pemanggilan struct tersebut tidak akan menghasilkan error. Tetapi jika definisi struct-nya terpisah dari statement pemanggilan struct (seperti contoh di atas), maka field dari struct tersebut harus publik.

Caranya membuat field sebagai publik adalah dengan menambahkan keyword `pub` pada property struct. Silakan ubah definisi struct `GamingConsole` menjadi seperti berikut:

```rust
// before
pub struct GamingConsole {
    name: String
}

// after
pub struct GamingConsole {
    pub name: String
}
```

Bisa dilihat keyword `pub` ditambahkan pada deklarasi property struct. yang

sebelumnya `name: String` sekarang ada keyword `pub` didepannya.

> *Lebih jelasnya mengenai visibility property dibahas pada chapter Module System → Visibility & Privacy*

A.23.9. Tuple struct property visibility

Lalu bagaimana dengan tuple struct? apakah property-nya juga harus didefinisikan publik agar bisa diakses dari tempat lain? Jawabannya ada di praktek berikut:

package source code structure

```
my_package
├── Cargo.toml
└── src
    ├── main.rs
    ├── models.rs
    └── models
        └── color.rs
```

src/main.rs

```rust
mod models;

fn main() {
    let red = models::color::Color(255, 255, 0);
    println!("{:#?}", red);
}
```

src/models.rs

```rust
pub mod color;
```

src/models/color.rs

```rust
#[derive(Debug)]
pub struct Color(i32, i32, i32);
```

```
struct_2::models::color
pub struct Color

1 implementation

tuple struct constructor `Color` is private
private tuple struct constructor rustc(E0603)

color.rs(2, 18): a constructor is private if any of the fields is
private

color.rs(2, 1): the tuple struct constructor `Color` is defined here

View Problem    No quick fixes available
models::color::Color(255, 255, 0);
```

Yap, error yang mirip juga muncul. Jadi jawaban dari pertanyaan sebelumnya adalah iya, property tuple struct juga harus publik agar bisa diakses dari tempat lain.

Cara deklarasi tuple struct dengan property publik adalah dengan menambahkan keyword `pub` di masing-masing deklarasi parameter tuple struct. Contoh:

```rust
// before
pub struct Color(i32, i32, i32);
```

Jalankan program untuk melihat hasilnya, error tidak muncul.

> *Lebih jelasnya mengenai visibility property dibahas pada chapter Module System → Visibility & Privacy*

A.23.10. Generic pada struct

Pembahasan mengenai generic pada struct ada pada chapter Generics.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../struct
```

◉ Referensi

- https://doc.rust-lang.org/book/ch05-00-structs.html
- https://doc.rust-lang.org/book/ch05-01-defining-structs.html
- https://doc.rust-lang.org/std/keyword.struct.html
- https://doc.rust-lang.org/rust-by-example/custom_types/structs.html

---

A.24. Associated Function

Chapter ini membahas tentang *associated function*. Kita akan belajar apa itu associated function dan apa perbedannya dengan fungsi biasa.

A.24.1. *Associated item & associated function*

Associated item adalah item yang memiliki asosiasi/hubungan dengan struct atau trait. Item disini bisa dalam banyak hal, bisa berupa fungsi atau lainnya.

Fungsi yang terhubung dengan suatu struct atau trait disebut dengan *associated function*. Fungsi jenis ini ekuivalen seperti fungsi biasa, perbedaannya adalah pada deklarasinya yang harus berada didalam blok kode `impl`, dan pemanggilannya harus menggunakan notasi path `NamaStruct::nama_fungsi`.

> - *Lebih jelasnya mengenai trait dibahas pada chapter Traits*
> - *Lebih jelasnya mengenai macam-macam item dibahas pada chapter Path & Item*

Ok, mari kita lanjut ke bagian praktek. Silakan tulis kode berikut terlebih dahulu.

```rust
#[derive(Debug)]
struct LegoSet {
    code: i32,
    name: String,
    category: String,
    age_minimum: i32,
}

fn main() {
    let rough_terrain_crane = LegoSet{
        code: 42082,
        name: String::from("Rough Terrain Crane"),
        category: String::from("Technic"),
        age_minimum: 11,
    };

    println!("{:#?}", rough_terrain_crane);
}
```

Sebuah struct bernama `LegoSet` didefinisikan memiliki 4 buah property. Di blok kode fungsi `main`, dibuat sebuah variabel bernama `rough_terrain_crane` yang merupakan object/instance dari struct `LegoSet`, kemudian object tersebut di-print.

> *Object struct bisa di-print menggunakan macro `println` karena pada definisi struct ditambahkan atribut `#[derive(Debug)]`*

Selanjutnya kita siapkan dua buah fungsi yang berasosiasi dengan struct `LegoSet`.

A.24.2. Keyword `impl`

Keyword `impl` digunakan untuk membuat *associated item*.

Pada konteks ini keyword tersebut digunakan untuk membuat *associated function* untuk struct `LegoSet`, sebuah fungsi bernama `new` dengan tugas adalah untuk membantu pembuatan object `LegoSet`.

Silakan tambahkan blok kode `impl` berikut setelah definisi struct `LegoSet`.

```
#[derive(Debug)]
struct LegoSet {
    code: i32,
    name: String,
    category: String,
    age_minimum: i32,
}

impl LegoSet {

    fn new(code: i32, name: String, category: String,
age_minimum: i32) -> LegoSet {
        LegoSet { code, name, category, age_minimum }
    }
}
```

Notasi penulisan keyword `impl` bisa dilihat pada contoh diatas, cukup tulis saja keyword tersebut diikuti nama struct yang diinginkan, kemudian diikuti dengan blok kode berisi definisi fungsi.

Fungsi dalam blok kode `impl` adalah yang disebut dengan *associated function*. Pada contoh di atas, fungsi `new` memiliki asosiasi dengan struct `LegoSet`.

Lanjut, panggil fungsi `new` tersebut untuk membuat object baru bernama `xtreme_offroader` lalu print isinya.

```
fn main() {
    let rough_terrain_crane = LegoSet{
        code: 42082,
        name: String::from("Rough Terrain Crane"),
        category: String::from("Technic"),
        age_minimum: 11,
    };
    println!("{:#?}", rough_terrain_crane);

    let xtreme_offroader = LegoSet::new(
        42099,
        String::from("4X4 X-treme Off-Roader"),
        String::from("Technic"),
        11,
    );
    println!("{:#?}", xtreme_offroader);
}
```

```
(base) PS D:\Labs\Adam Studio\Book\dasarpemrogramanrust\dasar
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s

LegoSet {
    code: 42082,
    name: "Rough Terrain Crane",
    category: "Technic",
    age_minimum: 11,
}
LegoSet {
    code: 42099,
    name: "4X4 X-treme Off-Roader",
    category: "Technic",
    age_minimum: 11,
}
```

Bisa dilihat pada blok kode `main`, cara pemanggilan associated function adalah

dengan menuliskan nama struct diikuti nama fungsi dengan notasi penulisan path.

```
NamaStruct::nama_fungsi();
NamaStruct::nama_fungsi(arg1, arg2, arg3, arg4);
LegoSet::new(arg1, arg2, arg3, arg4);
```

Jadi seperti itu, semoga cukup jelas. Silakan berkreasi dengan menambahkan *associated function* lainnya agar makin terbiasa. Contoh:

```
impl LegoSet {

    fn new(code: i32, name: String, category: String,
age_minimum: i32) -> LegoSet {
        LegoSet { code, name, category, age_minimum }
    }

    fn what_is_lego() {
        println!("Lego is a line of plastic construction toys")
    }
}

fn main() {
    LegoSet::what_is_lego();

    let xtreme_offroader = LegoSet::new(
        42099,
        String::from("4X4 X-treme Off-Roader"),
        String::from("Technic"),
        11,
    );
    println!("{:#?}", xtreme_offroader);
}
```

◉ Fungsi `LegoSet::new`

O iya, sedikit tambahan info saja, fungsi `new` di-atas didefinisikan menggunakan metode field init shorthand.

```
fn new(code: i32, name: String, category: String, age_minimum:
i32) -> LegoSet {
    LegoSet { code, name, category, age_minimum }
}

// ... adalah ekuivalen dengan ...

fn new(code: i32, name: String, category: String, age_minimum:
i32) -> LegoSet {
    LegoSet {
        code: code,
        name: name,
        category: category,
        age_minimum: age_minimum
    }
}
```

◉ *Naming convention* associated function

Sesuai anjuran di halaman dokumentasi Rust, snake case digunakan dalam penamaan associated function. Contoh:

```
impl NamaStruct {

    fn nama_associated_func() {
        // ...
    }
```

A.24.3. Tipe data `Self`

Tipe data `Self` (perhatikan huruf `S`-nya adalah kapital) merupakan representasi untuk tipe data struct atau trait dimana blok kode `impl` dideklarasikan.

> *Tipe data `Self` hanya bisa digunakan dalam blok kode `impl`*

Sebagai contoh, pada kode yang sudah dipraktekan, keyword `impl` diterapkan dalam pembuatan *associated items* untuk struct `LegoSet`. Dalam blok kode tersebut, tipe data `LegoSet` bisa diganti dengan `Self`.

Silakan lihat contoh dibawah ini, ada 4 buah cara deklarasi fungsi `new` yang kesemuanya adalah ekuivalen.

```
impl LegoSet {
    fn new(code: i32, name: String, category: String,
age_minimum: i32) -> LegoSet {
        LegoSet { code, name, category, age_minimum }
    }
}
```

```
impl LegoSet {
    fn new(code: i32, name: String, category: String,
age_minimum: i32) -> LegoSet {
        Self { code, name, category, age_minimum }
    }
}
```

```rust
impl LegoSet {
    fn new(code: i32, name: String, category: String,
age_minimum: i32) -> Self {
        LegoSet { code, name, category, age_minimum }
    }
}
```

```rust
impl LegoSet {
    fn new(code: i32, name: String, category: String,
age_minimum: i32) -> Self {
        Self { code, name, category, age_minimum }
    }
}
```

A.24.4. Associated function dalam module

Struct adalah salah satu dari beberapa item yang ada di Rust. Struct bisa saja berada dalam sebuah module, baik inline maupun *normal* module.

Sekarang, kode yang sudah dipraktekan di atas akan kita refactor. Struct `LegoSet` beserta associated items-nya dipindah ke module file bernama `lego`, dengan itu maka pemanggilan struct tersebut harus menggunakan path `lego::LegoSet`.

Kurang lebih struktur package mendi seperti ini:

package source code structure

```
my_package
```

```rust
#[derive(Debug)]
struct LegoSet {
    code: i32,
    name: String,
    category: String,
    age_minimum: i32,
}

impl LegoSet {

    fn new(code: i32, name: String, category: String, age_minimum: i32) -> Self {
        Self { code, name, category, age_minimum }
    }
}
```

```rust
mod lego;

fn main() {
    let rough_terrain_crane: LegoSet = lego::LegoSet{
        code: 42082,
        name: String::from("Rough Terrain Crane"),
        category: String::from("Technic"),
        age_minimum: 11,
    };
    println!("{:#?}", rough_terrain_crane);

    let xtreme_offroader = lego::LegoSet::new(
        42099,
        String::from("4X4 X-treme Off-Roader"),
        String::from("Technic"),
        11,
    );
    println!("{:#?}", xtreme_offroader);
}
```

Ada yang aneh, padahal pemanggilan struct `LegoSet` sudah diganti menjadi `lego::LegoSet`, tapi kenapa ada error di kode? Silakan *hover* baris kode yang ada highlight merah, atau jalankan saja program untuk melihat detail errornya.

```
associated function 2::lego
struct LegoSet
2 implementations
struct `LegoSet` is private
private struct rustc(E0603)
lego.rs(2, 1): the struct `LegoSet` is defined here
View Problem  Quick Fix... (Ctrl+.)
lego::LegoSet{
```

Error muncul karena struct `Legoset` adalah private. Solusinya cukup tambahkan keyword `pub` dalam definisi struct beserta *associated function*-nya.

> *Untuk blok kode `impl` tidak perlu ditambahi keyword `pub`*

Ok, setelah update diaplikasikan, coba lihat lagi. Sekarang error-nya berubah, bagian baris pengisian property struct yang jadi error.

```
associated function 2 > src > legos.rs > {} impl LegoSet > new
#[derive(Debug)]
2 implementations
pub struct LegoSet {
    code: i32,
    name: String,
    category: String,
    age_minimum: i32,
}

impl LegoSet {

    pub fn new(code: i32, name: String, category: String, age_minimum: i32) -> Self {
        Self { code, name, category, age_minimum }
    }
}

```

```
main.rs 4, U ×
associated function 2 > src > main.rs > main
mod lego;

▶ Run | Debug
fn main() {
    let rough_terrain_crane: LegoSet = lego::LegoSet{
        code: 42082,
        name: String::from("Rough Terrain Crane"),
        category: String::from("Technic"),
        age_minimum: 11,
    };
    println!("{:#?}", rough_terrain_crane);

    let xtreme_offroader = lego::LegoSet::new(
        42099,
        String::from("4x4 X-treme Off-Roader"),
        String::from("Technic"),
        11,
    );
    println!("{:#?}", xtreme_offroader);
}

```

Error ini terjadi karena property dari struct terdeteksi sebagai **private**. Solusi yang bisa dipergunakan ada 2:

- Tambahkan saja keyword `pub` pada definisi property struct.

src/lego.rs

```rust
pub struct LegoSet {
    pub code: i32,
    pub name: String,
    pub category: String,
    pub age_minimum: i32,
}
```

- Atau, tetap biarkan property struct sebagai **private**, namun pada semua statement pembuatan object menggunakan tipe tersebut harus selalu dilakukan via `lego::LegoSet::new()`.

src/main.rs

```rust
let object = lego::LegoSet::new(
    42099,
    String::from("4X4 X-treme Off-Roader"),
    String::from("Technic"),
    11,
);
```

Opsi mana yang paling pas? Pertanyaan ini jawabannya adalah tergantung kebutuhan dan *case*. Jika memang property struct di-desain agar private (tidak perlu diakses secara publik), maka gunakan saja associated function dalam pembuatan object-nya.

Namun jika memang ada kebutuhan salah satu property atau kesemuanya harus bisa diakses secara publik, maka tambahkan keyword `pub` sesuai kebutuhan.

A.24.5. Tuple struct associated function

Sama seperti struct, tuple struct juga bisa memiliki *associated items*. Cara deklarasi dan pemanggilan item-nya juga sama.

Pada contoh berikut, tuple struct `Color` memiliki 3 buah associated functions, yaitu `red`, `green`, dan `blue`.

package source code structure

```
my_package
|── Cargo.toml
└── src
     |── model.rs
     └── main.rs
```

src/model.rs

```rust
#[derive(Debug)]
pub struct Color(i32, i32, i32);

impl Color {

    pub fn red() -> Self {
        Self(255, 0, 0)
    }

    pub fn green() -> Self {
        Self(0, 255, 0)
```

src/main.rs

```rust
mod model;

fn main() {
    let red = model::Color::red();
    let green = model::Color::green();
    let blue = model::Color::blue();

    println!("{:#?} {:#?} {:#?}", red, green, blue);

    let random_color = model::Color(12, 25, 47);
    println!("{:#?} ", random_color);
}
```

Hmm, ada yang aneh, karena suatu alasan statement `model::Color` dianggap error.

```
associated function 3::model
pub struct Color

2 implementations

tuple struct constructor `Color` is private
private tuple struct constructor rustc(E0603)

model.rs(2, 18): a constructor is private if any of the fields is private

model.rs(2, 1): the tuple struct constructor `Color` is defined here

View Problem    No quick fixes available
model::Color(12, 25, 47);
```

Error ini sebenarnya mirip dengan error praktek sebelumnya, ketika mencoba membuat object baru dari struct sedangkan property struct tersebut adalah private.

Pada kasus struct, solusinya cukup dengan tambahkan keyword `pub` atau

siapkan *associated function* untuk pembuatan object. Pada tuple struct, solusinya juga mirip, ada dua opsi yang bisa dipilih.

- Tambahkan saja keyword `pub` pada definisi parameter tuple struct.

  src/model.rs

  ```
  // before
  pub struct Color(i32, i32, i32);

  // after
  pub struct Color(pub i32, pub i32, pub i32);
  ```

- Atau, tetap biarkan parameter tuple struct sebagai **private**, namun siapkan *associated function* baru untuk pembuatan object struct tuple, seperti ini:

  src/model.rs

  ```
  // ...

  impl Color {
      pub fn new(r: i32, g: i32, b: i32) -> Self {
          Self(r, g, b)
      }

      // ...
  }
  ```

  src/main.rs

  ```
  fn main() {
  ```

> *Lebih jelasnya mengenai visibility property dibahas pada chapter Module System → Visibility & Privacy*

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-
example/../associated_function
```

◉ Chapter relevan lainnya

- Function
- Method
- Module System → Visibility & Privacy

◉ Referensi

- https://doc.rust-lang.org/book/ch05-03-method-syntax.html
- https://doc.rust-lang.org/std/keyword.impl.html
- https://doc.rust-lang.org/std/keyword.SelfTy.html
- https://doc.rust-lang.org/rust-by-example/fn/methods.html

A.25. Method

Pada chapter ini kita akan belajar tentang method beserta perbedaannya dengan *associated function*.

A.25.1. Method vs *associated function*

Method adalah *associated item* yang hanya bisa diakses lewat instance/object, berbeda dengan *associated function* yang pengaksesan fungsinya via tipe data struct.

Silakan lihat ilustrasi kode berikut, lalu pelajari penjelasan dibawahnya untuk mencari tau perbedaan *associated function* vs method.

```
struct Car {
    // ...
}

impl Car {
    // ...
}

// associated function
let my_car: Car = Car::new();

// method
let info: String = my_car.info();
```

- Fungsi `new` disitu adalah *associated function* milik struct `car`. Dengannya object baru bernama `my_car` bertipe `car` dibuat.
- Object `my_car` adalah variabel bertipe `car`. Via object tersebut method bernama `info` diakses.
- Method `info` tidak bisa diakses via struct `car`. Dan *associated function* `new` juga tidak bisa diakses dari instance/object `my_car`.

> *Di bahasa pemrograman lain, associated function disebut dengan **class method**, sedangkan method disebut dengan **instance method***

Pengaksesan *associated function* dilakukan menggunakan notasi path dengan separator `::`, contohnya seperti `car::new()`. Sedangkan pengaksesan method menggunakan separator `.`, contoh: `my_car.info()`.

Agar lebih jelas, mari lanjut ke bagian praktek. Siapkan package dengan struktur seperti berikut:

package source code structure

```
my_package
|── Cargo.toml
└── src
    |── main.rs
    └── models.rs
```

Buka file `models.rs`, isi dengan deklarasi struct `car` berikut diikuti dengan blok kode `impl` untuk associated items-nya.

src/models.rs

```rust
#[derive(Debug)]
pub struct Car {
    brand: String,
    model: String,
    manufacture_year: i32
}

impl Car {
    // ...
}
```

A.25.2. Deklarasi method

Cara deklarasi method mirip dengan *associated function*, perbedaannya adalah parameter pertama harus diisi dengan `&self` pada deklarasi method. Parameter tersebut menjadi identifier apakah fungsi merupakan *associated function* atau *method*.

> *Object `self` merupakan representasi dari current instance atau current object.*
>
> *Statement `&self` artinya kita melakukan operasi borrowing terhadap object `self`.*
>
> *Lebih jelasnya mengenai borrowing dibahas pada chapter terpisah.*

Silakan tambahkan *associated function* bernama `new` dan method bernama `info` berikut. Tulis keduanya dalam blok kode `impl`.

src/models.rs

```rust
// ...

impl Car {

    pub fn new(brand: String, model: String) -> Self {
        Self { brand, model, manufacture_year: 0 }
    }

    pub fn info(&self) -> String {
        if self.manufacture_year == 0 {
            format!("{} model {}", self.brand, self.model)
        } else {
            format!(
                "{} model {}, manufactured at {}",
                self.brand,
                self.model,
                self.manufacture_year
            )
        }
    }
}
```

Bisa dilihat, deklarasi method adalah mirip dengan fungsi biasa, perbedaannya ada pada deklarasi parameter pertama fungsi yang diisi dengan `&self`.

Cara mengakses property milik *current object* dari dalam method adalah menggunakan keyword `self`, contohnya seperti `self.brand`, `self.model`, dan `self.manufacture_year`, pemanggilan ketiganya adalah mengarah ke value property milik *current object* (`self`).

Selanjutnya panggil keduanya di fungsi `main`.

src/main.rs

```rust
mod models;

fn main() {
    let car = models::Car::new(
        String::from("Mercedes-Benz"),
        String::from("Vision Gran Turismo")
    );
    println!("car: {:?}", car);

    let info = car.info();
    println!("info: {:?}", info);
}
```

Bisa dilihat, variabel `car` adalah instance dari struct `models::Car`, dibuat menggunakan *associated function* `models::Car::new`. Dari variabel tersebut kemudian diakses method `info`.

Jalankan program untuk melihat hasilnya.

```
warning: `method_1` (bin "method_1") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.47s
     Running `target\debug\method_1.exe`
car: Car { brand: "Mercedes-Benz", model: "Vision Gran Turismo", manufacture_year: 0 }
info: "Mercedes-Benz model Vision Gran Turismo"
```

◉ *Naming convention* method

Sesuai anjuran di halaman dokumentasi Rust, snake case digunakan dalam penamaan method. Contoh:

```rust
impl NamaStruct {

    fn nama_method(&self) {
```

A.25.3. Method parameter

Ok, penulis rasa sudah cukup jelas perihal perbedaan *associated function* dengan method. Sekarang, bagaimana dengan method yang ada parameternya? Cara deklarasinya adalah cukup dengan menuliskan parameter yang diinginkan setelah `&self`.

Lanjut ke praktek berikutnya. Silakan buat method baru bernama `congratulate` yang memiliki 1 buah parameter bertipe `String`, dengan tugas adalah menampilkan pesan selamat.

src/models.rs

```
// ...

impl Car {

    // ...

    pub fn congratulate(&self, name: String) {
        println!("hello {}", name);
        println!("congrats with your new car {}", self.info());
        println!("vroooom vroooooooommmmm!");
    }
}
```

Dalam method `congratulate` ada statement pemanggilan method `info`, yang return value-nya ikut di-print. Notasi `self.nama_method()` digunakan untuk pemanggilan method dari dalam method. Mirip seperti pemanggilan property yaitu `self.nama_property`.

Selanjutnya, panggil method `congratulate` di fungsi `main`. Isi argumen pemanggilan method dengan sebuah `String`.

src/main.rs

```
// ...

fn main() {
    let car = models::Car::new(
        String::from("Mercedes-Benz"),
        String::from("Vision Gran Turismo")
    );

    car.congratulate(String::from("Sylvanas Windrunner"));
}
```

Meskipun pada definisi method parameter pertama adalah `&self`, pada saat pemanggilan method yang menjadi parameter pertama adalah parameter setelah `&self` yaitu `name`.

```
warning: `method_1` (bin "method_1") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.47s
     Running `target\debug\method_1.exe`

hello Sylvanas Windrunner,
congrats with your new car Mercedes-Benz model Vision Gran Turismo
vroooom vroooooooommmm!
```

A.25.4. Struct property mutability via method

Nilai property struct bisa diubah dari dalam method. Pada bagian ini kita akan buat method baru untuk struct `Car` dengan nama `set_manufacture_year`.

Method ini bertugas untuk melakukan perubahan nilai property `manufacture_year`.

Silakan tulis kode berikut:

src/models.rs

```
// ...

impl Car {

    // ...

    pub fn set_manufacture_year(&self, year: i32) {
        self.manufacture_year = year
    }
}
```

```
method 1::models::Car
manufacture_year: i32

cannot assign to `self.manufacture_year`, which is behind a `&` reference
`self` is a `&` reference, so the data it refers to cannot be written rustc(E0594)

models.rs(32, 33): consider changing this to be a mutable reference: `&mut self`

View Problem    No quick fixes available
self.manufacture_year = year
```

Hmm, tapi kenapa terdeteksi error? Penyebabnya error tersebut adalah karena **mutable reference** tidak digunakan dalam pengaksesan current object yang padahal ada operasi *mutable* atau perubahan nilai terhadap property disitu. Syntax `&self` artinya operasi peminjaman object `self` adalah *read only*.

Cara mengambil mutable reference dari object `self` adalah dengan menggunakan `&mut self`. Cara tersebut kurang lebih sama seperti

pengambilan mutable reference dari variabel biasa.

Ok, sekarang kita coba modifikasi deklarasi method `set_manufacture_year` menjadi seperti berikut:

src/models.rs

```
// ...

impl Car {

    // ...

    pub fn set_manufacture_year(&mut self, year: i32) {
        self.manufacture_year = year
    }
}
```

Lalu panggil method `set_manufacture_year` di fungsi `main`.

src/main.rs

```
// ...

fn main() {
    let mut car = models::Car::new(
        String::from("Mercedes-Benz"),
        String::from("Vision Gran Turismo")
    );

    let info = car.info();
    println!("info: {:?}", info);

    car.set_manufacture_year(2013);
```

Jalankan program, lihat hasilnya.

```
warning: `method_1` (bin "method_1") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.4/s
     Running `target\debug\method_1.exe`

info: "Mercedes-Benz model Vision Gran Turismo"
detailed info: "Mercedes-Benz model Vision Gran Turismo, manufactured at 2013"
```

Setelah `manufacture_year` di-set, method `info` mengembalikan pesan yang berbeda. Dari sini bisa disimpulkan bahwa method `set_manufacture_year` sukses menjalankan tugasnya untuk mengubah property `manufacture_year`.

A.25.5. Generic pada method

Pembahasan mengenai generic pada method ada pada chapter Generics.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../method
```

◉ Referensi

- https://doc.rust-lang.org/book/ch05-03-method-syntax.html
- https://doc.rust-lang.org/std/keyword.impl.html
- https://doc.rust-lang.org/std/keyword.SelfTy.html

- https://doc.rust-lang.org/std/keyword.self.html
- https://doc.rust-lang.org/rust-by-example/fn/methods.html

A.26. Enum

Enum atau *enumerated type* adalah sebuah tipe data yang digunakan untuk menampung nilai konstan. Pada chapter ini kita akan mempelajarinya.

Enum ada sedikit kemiripan dengan konstanta, bedanya ada pada nilai atau *underlying value*-nya. Jika di konstanta, yang didefinisikan adalah nama beserta value-nya, di enum yang didefinisikan adalah tipe data enum dan enum value. Enum value ini bentuknya seperti variabel tanpa nilai (lebih tepatnya nama dari enum value tersebut adalah nilainya). Lebih jelasnya silakan ikut pembahasan chapter ini.

A.26.1. Keyword `enum`

Keyword `enum` digunakan untuk membuat *enumerated type*. Cara penulisannya seperti berikut:

```
enum NamaEnum {
    NilaiEnum1,
    Nilai2,
    NilaiEnumKe3,
    // ...
}
```

`NamaEnum` di atas adalah tipe data custom yang didefinisikan bertipe enum. Sedangkan `NilaiEnum1`, `Nilai2`, dan `NilaiEnumKe3` adalah yang disebut dengan enum value. Dengan itu maka ketiga enum values tersebut tipe datanya adalah sama, yaitu `NamaEnum`.

Mari kita lanjut praktek. Berikut ini adalah definisi konstanta yang menggunakan tipe data string untuk menampung nilai konstan-nya. Lalu dibawahnya ada lagi definisi nilai konstan tetapi menggunakan enum sebagai tipe data yang digunakan.

```
// definisi konstanta
const SuperheroSuperman: &str = "superman";
const SuperheroOmniMan: &str = "omniman";
const SuperheroHomelander: &str = "homelander";
const SuperheroHyperion: &str = "hyperion";

// definisi enum
enum Superhero {
    Superman,
    OmniMan,
    Homelander,
    Hyperion,
};
```

Di contoh bisa dilihat, `Superhero` adalah tipe data enum baru. Dari tipe data tersebut dibuat 4 buah enum values, yaitu `Superman`, `OmniMan`, `Homelander`, dan `Hyperion`.

Pada pembuatan konstanta, tipe data beserta value-nya harus ditentukan di awal. Pada enum, yang perlu didefinisikan adalah tipe data enum-nya (sebagai contoh `Superhero`) kemudian diikut dengan enum value yang dituliskan tanpa pengisian nilai.

- Definisi variabel dengan isi konstanta:

  ```
  let value1: &str = SuperheroSuperman;
  let value2 = SuperheroOmniMan;
  // ...
  ```

- Definisi variabel bertipe data enum `Superhero`:

```
let value3: Superhero = Superhero::Superman;
let value4 = Superhero::OmniMan;
// ...
```

Notasi path digunakan dalam penulisan enum value dengan format `NamaEnum::EnumValue`

A.26.2. *Naming convention* enum

Sesuai anjuran di halaman dokumentasi Rust, upper camel case digunakan dalam penamaan Enum beserta value-nya.

Contoh:

```
enum Superhero {
    Superman,
    OmniMan,
    Homelander,
    Hyperion,
};
```

A.26.3. Seleksi kondisi enum

Tipe data enum biasa dipakai pada seleksi kondisi, namun caranya sedikit berbeda. Default-nya keyword `if` tidak bisa digunakan pada tipe data enum.

Pada contoh berikut, statement seleksi kondisi `value3` menghasilkan error:

```rust
// seleksi kondisi pada konstanta
if value1 == SuperheroSuperman {
    println!("hello superman!");
}

// seleksi kondisi pada enum
if value3 == Superhero::Superman {
    println!("hello superman!");
}
```

```
if value1 == SuperheroSuperman {
    println!("hello superman!");
}

if value3 == Superhero::Superman {
    printl
}
core::cmp::PartialEq
pub fn eq(&self, other: &Rhs) -> bool

This method tests for `self` and `other` values to be equal, and is used by `==`.

binary operation `==` cannot be applied to type `Superhero` rustc(E0369)
main.rs(25, 8): Superhero
main.rs(25, 18): Superhero
main.rs(0, 1): an implementation of `PartialEq<_>` might be missing for `Superhero`
main.rs(0, 1): consider annotating `Superhero` with `#[derive(PartialEq)]`: `#[derive(PartialEq)]`
```

> *Error tersebut muncul karena tipe data enum `Superhero` tidak memiliki trait `PartialEq`. Lebih jelasnya mengenai trait dibahas pada chapter Traits.*

Lalu bagaimana cara pengaplikasian seleksi kondisi pada tipe enum? Caranya adalah menggunakan keyword `match`.

A.26.4. Keyword `match`

`match` adalah salah satu keyword untuk operasi seleksi kondisi di Rust. Penerapan keyword ini cukup luas, namun pada chapter ini hanya akan dibahas penerapannya yang relevan dengan topik enum.

Mari kita pelajarinya sembari praktek. Silakan buat package baru, lalu definisikan tipe enum `Food` berikut beserta 4 enum value-nya.

```
enum Food {
    PenyetanTerangBulan,
    PizzaNanas,
    EsKrimIkanMujaer,
    MiGorengKuah,
}
```

Lalu buat sebuah variabel bernama `makanan_favorit` untuk menampung salah satu nilai enum. Kemudian gunakan keyword `match` untuk menerapkan operasi seleksi kondisi dengan aksi menampilkan sebuah pesan sesuai dengan nilai yang cocok.

```
fn main() {
    let makanan_favorit: Food = Food::PenyetanTerangBulan;

    match makanan_favorit {
        Food::PenyetanTerangBulan => {
            println!("your food taste is quite ... unique");
        },
        Food::PizzaNanas => {
            println!("it's morally wrong to have pineaple on top
of pizza");
```

Di atas bisa dilihat bagaimana cara penggunaan keyword `match` untuk penerapan seleksi kondisi pada tipe data enum.

Notasi penulisannya kurang lebih seperti ini:

```
match variabel_enum {
    TipeEnum::ValueEnum1 => {
        // ...
    },
    TipeEnum::ValueEnum2 => {
        // ...
    },

    // ...
}
```

Kembali ke contoh program, variabel `makanan_favorit` dicek nilainya menggunakan keyword `match`.

- Jika nilainya adalah `Food::PenyetanTerangBulan`, muncul pesan:

  ```
  "your food taste is quite ... unique"
  ```

- Jika nilainya adalah `Food::PizzaNanas`, muncul pesan:

  ```
  "it is morally wrong to have pineaple on top of pizza"
  ```

- Jika nilainya adalah `Food::EsKrimIkanMujaer`, muncul pesan:

  ```
  "I don't know what to say"
  ```

- Jika nilainya adalah `Food::MiGorengKuah`, muncul pesan:

```
"sometimes people do eat this, but it's ok"
```

Jalankan program untuk melihat hasilnya:

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.98s
     Running `target\debug\enum 2.exe`
your food taste is quite ... unique
```

Keyword `match` ini sebenarnya tidak hanya digunakan untuk seleksi kondisi saja. Di Rust ada yang disebut dengan *pattern matching*. Metode *pattern matching* ini memahami *special syntax* yang kegunaanya lebih luas dibanding hanya sekedar seleksi kondisi biasa.

> *Lebih jelasnya mengenai pattern matching dibahas pada chapter Pattern Matching*

A.26.5. Enum value → *tuple struct*-like

Enum value di struct bisa juga didesain seperti *tuple struct*. Sebagai contoh, enum `Food` di atas akan kita tambahi dengan satu enum value baru berbentuk *tuple struct*.

Silakan tambahkan enum value `MakananLainnya` berikut. Enum ini kita fungsikan untuk mengidentifikasi data makanan lainnya selain dari yang sudah ada di enum `Food`. Notasi penulisan *tuple struct* `MakananLainnya(String)`

artinya enum value `MakananLainnya` didefinisikan untuk bisa menampung data property dalam bentuk `String`.

```
enum Food {
    PenyetanTerangBulan,
    PizzaNanas,
    EsKrimIkanMujaer,
    MiGorengKuah,
    MakananLainnya(String), // <---- enum value baru
}
```

Sekarang ubah isi variabel `makanan_favorit` dengan enum value baru yang sudah dibuat. Syntax `Food::MakananLainnya(nasi_goreng)` artinya enum value yang digunakan adalah `Food::MakananLainnya` dengan isi property didapat dari variabel `nasi_goreng`.

Tambahkan juga `Food::MakananLainnya` dalam blok kode `match`.

```
fn main() {
    // enum value MakananLainnya digunakan
    // dengan isi property adalah string "nasi goreng"
    let nasi_goreng = String::from("nasi goreng");
    let makanan_favorit = Food::MakananLainnya(nasi_goreng);

    match makanan_favorit {
        Food::PenyetanTerangBulan => {
            println!("your food taste is quite ... unique");
        },
        Food::PizzaNanas => {
            println!("it's morally wrong to have pineaple on top
of pizza");
        },
        Food::EsKrimIkanMujaer => {
            println!("I don't know what to say. this should be
```

Bisa dilihat ada keunikan dalam penulisan seleksi kondisi `Food::MakananLainnya` dalam blok kode `match`. Disitu ada parameter bernama `m` yang parameter tersebut akan berisi data property jika memang *match* dengan `makanan_favorit`.

Coba jalankan untuk melihat hasilnya:

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.90s
     Running `target\debug\enum_2.exe`
do you like nasi goreng? nice taste!
```

O iya, jumlah property value enum berbentuk *tuple struct* ini tidak terbatas ya. Pada contoh di atas, `Food::MakananLainnya` hanya memiliki 1 property. Lebih dari satu juga bisa.

A.26.6. Enum value → *struct*-like

Enum value bisa juga didesain memiliki property seperti *struct*.

Mari kita terapkan pada kode sebelumnya. Tambahkan 1 buah enum value lagi dengan nama `MieSetan` yang ditulis dalam bentuk *struct*-like, dan memiliki 2 buah property.

```
enum Food {
    PenyetanTerangBulan,
    PizzaNanas,
    EsKrimIkanMujaer,
    MiGorengKuah,
    MakananLainnya(String),
    MieSetan { level_pedas: i32, pakek_piring: bool }
}
```

Setelah itu, isi variabel dengan nilai adalah enum value `Food::MieSetan`, level pedasnya 5, dan tanpa piring.

Tak lupa tambahkan seleksi kondisi untuk `Food::MieSetan` pada blok kode `match`.

```
fn main() {
    let makanan_favorit = Food::MieSetan {
        level_pedas: 5,
        pakek_piring: false
    };

    match makanan_favorit {
        Food::PenyetanTerangBulan => {
            println!("your food taste is quite ... unique");
        },
        Food::PizzaNanas => {
            println!("it's morally wrong to have pineaple on top
of pizza");
        },
        Food::EsKrimIkanMujaer => {
            println!("I don't know what to say. this should be
illegal");
        },
        Food::MiGorengKuah => {
            println!("sometimes people do eat this, but it's ok");
        },
        Food::MakananLainnya(m) => {
            println!("do you like {m}? nice taste!");
        },
        Food::MieSetan { level_pedas, pakek_piring } => {
            if level_pedas > 3 {
                println!("mie setan lvl {} is too much!",
level_pedas);
            } else {
                println!("mie setan lvl {} is perfect!",
```

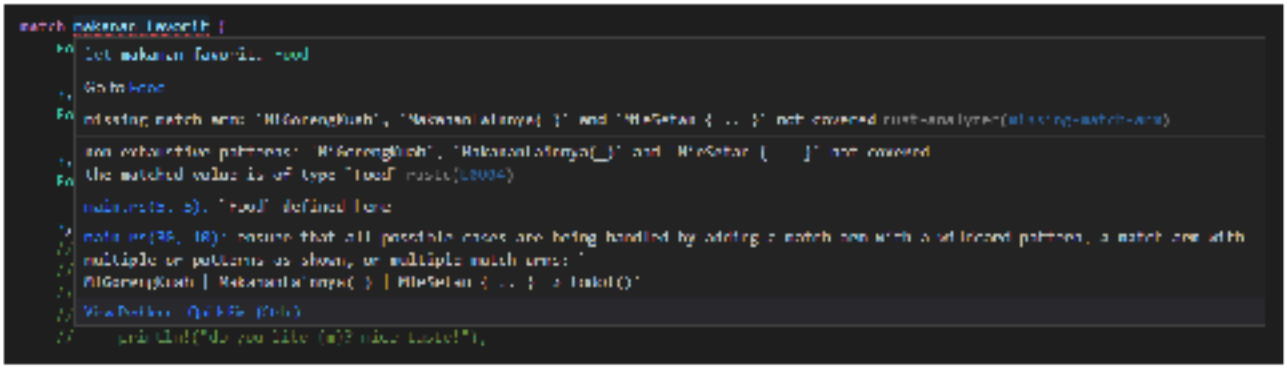
Dalam seleksi kondisi `Food::MieSetan` bisa dilihat ada beberapa statement. Kurang lebih jika nilai dari variabel `makanan_favorit` adalah `Food::MieSetan` maka:

- Akan memunculkan pesan yang berbeda tergantung level pedasnya
- Dan jika terdeteksi tidak menggunakan piring, dimunculkan pesan tambahan

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.98s
     Running `target\debug\enum_2.exe`

mie setan lvl 5 is too much!
how are you going to eat the food without a plate, huh?
```

A.26.7. Aturan *pattern matching* enum

Dalam blok kode `match`, semua enum value harus dituliskan. Jika tidak, pasti muncul error. Contohnya bisa dilihat di gambar berikut, beberapa seleksi kondisi enum value di-remark, hasilnya ada error.



Error tersebut sebenarnya bisa diantisipasi dengan menambahkan seleksi kondisi dengan penulisan seperti berikut:

```
match makanan_favorit {
    Food::PenyetanTerangBulan => {
        println!("your food taste is quite ... unique");
    },
    Food::PizzaNanas => {
        println!("it's morally wrong to have pineaple on top of
pizza");
    },
    _ => {
        println!("never heard about that food");
    }
}
```

Menggunakan blok kode `match` di atas, jika nilai `makanan_favorit` adalah selain `Food::PenyetanTerangBulan` dan `Food::PizzaNanas`, maka pesan `never heard about that food` adalah yang muncul di layar.

Selain variabel `_` bisa juga menggunakan nama variabel apapun, misalnya `some_var`. Namun jika variabel tersebut tidak digunakan dalam blok kode, akan muncul warning.

A.26.8. Enum module & visibility

Mari kita coba cek perihal visibility dari enum. Siapkan package baru dengan struktur seperti berikut:

package source code structure

```
my_package
|── Cargo.toml
└── src
    |── constants.rs
```

Pada file `constants.rs`, tambahkan enum `Company` berikut. Pastikan enum adalah publik dengan menambahkan keyword `pub` pada deklarasinya.

src/constants.rs

```
pub enum Company {
    Apple,
    Microsoft,
    Google,
    Github
}
```

Kemudian tambahkan kode berikut di `main.rs`.

src/main.rs

```
mod constants;

fn main() {
    let company = constants::Company::Apple;

    match company {
        constants::Company::Apple => {
            print!("apple")
        },
        _ => {
            print!("other than apple")
        }
    }
}
```

Jalankan, hasilnya tidak error, karena `Company` didefinisikan publik.

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.90s
     Running `target\debug\enum_2.exe`
apple
```

Coba lakukan modifikasi dengan menghilangkan keyword `pub` saat definisi enum, hasilnya pasti error.

> *Pada tipe data enum, keyword `pub` cukup ditambahkan pada definisi enum type, tidak perlu ditambahkan satu persatu di tiap enum values.*

A.26.9. Generic pada enum

Pembahasan mengenai generic pada enum ada pada chapter Generics.

A.26.10. Pembahasan lanjutan *pattern matching*

Pembahasan yang lebih mendetail tentang keyword `match` dan *pattern matching* ada pada chapter Pattern Matching.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../enum
```

◉ Work in progress

- Pembahasan tentang associated function dan method pada enum

◉ Referensi

- https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html
- https://doc.rust-lang.org/rust-by-example/custom_types/enum.html
- https://doc.rust-lang.org/reference/items/enumerations.html
- https://doc.rust-lang.org/book/ch06-02-match.html
- https://doc.rust-lang.org/std/keyword.enum.html

A.27. Type Alias & Casting

Chapter ini membahas tentang type alias dan juga casting (explicit conversion) pada tipe data primitif scalar.

> *Pembahasan mengenai conversion pada tipe data non-primitif scalar (seperti struct dan enum) dibahas pada chapter Trait → Conversion (From & Into)*

A.27.1. Type Alias

Type alias adalah pemberian nama baru ke suatu tipe data. Cara pembuatan alias sangat mudah yaitu menggunakan keyword `type`.

Berikut adalah 2 contoh penerapan type alias untuk membuat tipe data baru:

```
type Inch = u64;
```

Tipe data `Inch` di atas adalah tipe baru yang merupakan alias dari tipe `u64`.

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
```

Struct `Point` di atas dibuat memiliki 2 item (`x` dan `y`). Dari struct tersebut dibuat tipe data baru bernama `Coordinate` yang merupakan alias dari `Point`.

A.27.2. Casting tipe data & alias

Casting adalah pengubahan tipe data tertentu ke tipe data lain yang keduanya masih compatible. Metode casting bisa diterapkan antara tipe data asli dan alias, dan juga antar tipe data scalar lainnya (yang memang compatible satu sama lain).

Pada contoh berikut, tipe data `Inch` di cast ke tipe data `u64` menggunakan keyword `as`.

```
let height: Inch = 6;
println!("height: {height}");

let height_in_u64 = height as u64;
println!("height_in_u64: {height_in_u64}");
```

```
warning: `type_alias_casting` (bin "type_alias_casting") generated 1 warnings
    Finished dev [unoptimized + debuginfo] target(s) in 0.47s
     Running `target\debug\type_alias_casting.exe`

height: 6
height_in_u64: 6
```

Contoh lainnya bisa dilihat pada kode berikut, variabel `p` dibuat menggunakan struct `Point`, kemudian di-cast ke tipe `Coordinate` sebagai data mutable lalu di ubah nilai itemnya, dan terakhir di-cast sekali lagi ke tipe `Point`.

```
let p = Point{ x: 0, y: 10 };
println!("p: {:?}", p);
```

```
warning: `type_alias_casting` (bin "type_alias_casting") generated 3 warnings
    Finished dev [unoptimized + debuginfo] target(s) in 0.47s
     Running `target\debug\type_alias_casting.exe`

p: Point { x: 0, y: 10 }
q: Point { x: 12, y: 10 }
r: Point { x: 12, y: 10 }
```

> *Operasi assignment dan type casting pada custom type* `struct` *membuat owner-nya berpindah. Perpindahan owner ini disebut dengan move semantics.*
>
> *Lebih jelasnya perihal topik ini dibahas pada chapter Ownership.*

A.27.3. Casting antar tipe scalar

Casting antar tipe data numerik dilakukan dengan cara yang sama seperti casting antar tipe data dan alias. Contoh:

```
let number = 32;
println!("number: {number}");

let number_in_u8 = number as u8;
println!("number_in_u8: {number_in_u8}");

let number_in_f64 = number as f64;
println!("number_in_f64: {number_in_f64}");

let new_number = 23.4 as f32;
println!("new_number: {new_number}");
```

```
warning: `type_alias_casting` (bin "type_alias_casting") generated 3 warnings
    Finished dev [unoptimized + debuginfo] target(s) in 0.47s
     Running `target\debug\type_alias_casting.exe`

number: 32
number_in_u8: 32
number_in_f64: 32
new_number: 23.4
```

Tipe data integer, unsigned integer, dan floating point bisa di-cast satu sama lain.

> *Selain itu, tipe `char` juga bisa di-cast ke tipe lainnya (selain tipe float)*

A.27.4. Konsekuensi casting tipe numerik

Salah satu hal yang harus diperhatikan dalam casting adalah pemilihan tipe data beserta size yang tepat.

Setiap tipe data memiliki besaran alokasi memory yang berbeda. Sebagai contoh, tipe data `u64` bisa menampung angka yang jauh lebih besar dibanding `u16`.

Bagaimana jika misalnya ada data numerik yang angkanya cukup besar dan hanya bisa ditampung pada tipe data `u64` (atau tipe data lainnya yang size-nya lebih besar), kemudian tipe tersebut di-cast ke tipe data yang lebih kecil contohnya `u16`?

Sebagai contoh pada kode berikut, variabel `timestamp` bertipe `u64` menampung data unix timestamp waktu sekarang. Data tersebut di cast ke tipe yang lebih kecil yaitu `u16`.

```rust
use std::time::{SystemTime, UNIX_EPOCH};

let timestamp: u64 =
SystemTime::now().duration_since(UNIX_EPOCH).unwrap().as_secs();

println!("timestamp (u64): {timestamp}");
println!("timestamp (as u16): {}", timestamp as u16);
```

```
warning: `type_alias_casting` (bin "type_alias_casting") generated 1 warnings
    Finished dev [unoptimized + debuginfo] target(s) in 0.47s
     Running `target\debug\type_alias_casting.exe`

timestamp (u64): 1671519631
timestamp (as u16): 58415
```

Bisa dilihat hasilnya angka menjadi lebih kecil, ini karena `u16` tidak bisa menampung nilai numerik sebesar `u64`. Lalu bagaimana jika angka tersebut di-cast lagi ke tipe `u64`?

```rust
let timestamp: u64 =
SystemTime::now().duration_since(UNIX_EPOCH).unwrap().as_secs();
println!("timestamp (u64): {timestamp}");
println!("timestamp (as u16): {}", timestamp as u16);
println!("from u16 back to u64: {}", (timestamp as u16) as u64);
```

```
warning: `type_alias_casting` (bin "type_alias_casting") generated 3 warnings
    Finished dev [unoptimized + debuginfo] target(s) in 0.47s
     Running `target\debug\type_alias_casting.exe`

timestamp (u64): 1671619792
timestamp (as u16): 58576
from u16 back to u64: 58576
```

Hasilnya adalah nilai tetap tidak akan kembali seperti semula. Jadi silakan berhati-hati dalam melakukan operasi casting antar tipe data numerik.

◉ Module `std::time`

`SystemTime` dan `UNIX_EPOCH` keduanya merupakan item milik module `std::time`.

Pengaksesan `SystemTime::now()` mengembalikan data waktu sekarang, dan jika di chain dengan method `duration_since(UNIX_EPOCH)` hasilnya adalah data duration bertipe `Result<Duration, SystemTimeError>`.

Dari data tersebut bisa kita chain lagi dengan method `unwrap` dan `as_second` untuk mengambil datanya dalam bentuk `u64`.

> - *Lebih jelasnya mengenai module `std::time` dibahas pada chapter Time*
> - *Lebih jelasnya mengenai tipe data `Result` dibahas pada chapter Tipe Data → Result*

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-
example/../type_alias_casting
```

◉ Referensi

- https://doc.rust-lang.org/rust-by-example/types/alias.html
- https://doc.rust-lang.org/rust-by-example/types/cast.html

A.28. Module System → Visibility & Privacy

Kita sebenarnya sudah mempelajari banyak hal yang berhubungan dengan visibility & privacy pada beberapa chapter sebelumnya. Jika pembaca mempelajari ebook ini secara urut, maka pastinya sudah familiar dengan keyword `pub`, `self`, `crate`, dan `super`.

Chapter ini merupakan pembahasan tambahan untuk ke-4 keyword tersebut, dan fokusnya lebih ke visibility & privacy di Rust secara general.

O iya, perihal *visbility* dan *privacy* itu sendiri, kedua istilah tersebut disini kita maknai sama, yang artinya kurang lebih adalah tentang manajemen akses item di Rust.

A.28.1. Pembahasan module system

Pastikan sudah mempelajari 5 buah chapter tentang module system yang sebelumnya sudah dibahas. Kesemua chapter tersebut sangat berhubungan dengan pembahasan chapter ini.

- A.18. Module System → Path & Item
- A.19. Module System → Package & Crate
- A.20. Module System → Module
- A.21. Module System → Inline Module

- A.22. Module System → Scope & Akses Item

A.28.2. Default visibility

Di Rust, *by default*, hampir semua item adalah private. Apa efeknya ketika item adalah private atau publik? Silakan ingat 2 aturan penting berikut:

1. Jika suatu item adalah private, maka item tersebut hanya bisa diakses dari *current module scope* dan dari *submodules* milik *current module*.
2. Jika suatu item adalah publik, maka dia bisa diakses dari module lain di luar *current module scope*, dengan catatan parent module scope item tersebut harus publik.

> *Kita sepakati disini, pada istilah* ***current module*** *kata module disitu bisa saja tertuju untuk module atau juga submodule*

Dua point di atas sangat penting untuk dipahami, karena digunakan sebagai landasan pertimbangan dalam penyusunan hirarki module. Sebagai contoh, kita bisa membuat program yang hanya meng-expose API tertentu (yang memang diperlukan untuk diakses oleh publik), tanpa perlu ikut meng-expose detail implementasinya.

Ok, sekarang silakan perhatikan path sederhana di bawah ini. Diasumsikan ada sebuah fungsi yang path aksesnya adalah berikut:

```
messaging::service_layer::some_black_magic
```

Segmen pertama yaitu `messaging` pasti adalah publik, karena di-import ke *crate root*. Lalu bagaimana dengan segmen `service_layer` dan juga `some_black_magic`?

Jika item `some_black_magic` disitu adalah publik, maka idealnya pengaksesan menggunakan path tersebut memungkinkan. Tapi kembali ke point ke-2 aturan yang sudah dibahas diatas, yaitu meskipun `some_black_magic` adalah publik, jika parent-nya (yang pada konteks ini adalah `service_layer`) adalah private, maka pengaksesan menggunakan path tersebut menghasilkan error.

Intinya, **sebuah item bisa diakses jika item tersebut adalah publik, dan parent item tersebut juga publik. Sedangkan default visibility untuk hampir semua item adalah private.**

Ok, sekarang mari lanjut ke praktek menggunakan contoh dengan pembahasan yang lebih mendetail. Silakan perhatikan dan praktekan kode berikut:

package source code structure

```
my_package
|── Cargo.toml
└── src
     |── messaging.rs
     └── main.rs
```

src/messaging.rs

```rust
const SOME_MESSAGE: &str = "hello rust";

mod service_layer {

    pub fn some_black_magic() {
        println!("{}", crate::messaging::SOME_MESSAGE)
    }
}
```

src/main.rs

```rust
mod messaging;

fn main() {
    messaging::say_hello();
}
```

- Konstanta `messaging::SOME_MESSAGE` adalah **private**. Penjelasan:
  - Konstanta ini merupakan module item milik `messaging`.
  - Konstanta ini **bisa diakses** dari *current module scope* (`messaging`).
  - Konstanta ini **bisa diakses** dari submodule milik *current module*, yaitu submodule dari `messaging`. Contohnya bisa dilihat pada fungsi `messaging::service_layer::some_black_magic` yang disitu ada statement pemanggilan `SOME_MESSAGE`.
  - Konstanta ini **tidak bisa diakses** dari luar *current module scope* (`messaging`).
- Submodule `messaging::service_layer` adalah **private**. Penjelasan:
  - Submodule ini merupakan module item milik `messaging`.
  - Submodule ini **bisa diakses** dari *current module scope* (`messaging`). Contohnya bisa dilihat pada fungsi `messaging::say_hello` yang disitu ada statement pemanggilan `service_layer`.
  - Submodule ini **bisa diakses** dari submodule milik *current module*, yaitu submodule dari `messaging`.
  - Submodule ini **tidak bisa diakses** dari luar *current module scope* (`messaging`).
- Fungsi `messaging::service_layer::some_black_magic` adalah **publik**.

Penjelasan:

- Fungsi ini merupakan module item milik `messaging::service_layer`.
- Fungsi ini **bisa diakses** dari *current module scope* (`messaging::service_layer`).
- Fungsi ini **bisa diakses** dari submodule milik *current module*, yaitu submodule dari `messaging::service_layer`. Contohnya bisa dilihat pada fungsi `messaging::say_hello` yang disitu ada statement pemanggilan fungsi `some_black_magic`.
- Fungsi ini **bisa diakses** dari luar *current module scope* (`messaging::service_layer`).
- Namun meskipun demikian, bisa tidaknya fungsi ini diakses dari luar *current module scope* (`messaging::service_layer`) juga tergantung dengan visibility dari current module itu sendiri, yaitu `messaging::service_layer`.
- Karena module `messaging::service_layer` adalah private, meskipun fungsi `some_black_magic` didalamnya adalah publik, pengaksesan fungsi tersebut dari luar module scope `messaging::service_layer` tidak dimungkinkan.
  - Pengaksesan `service_layer::some_black_magic` dari `messaging::say_hello` tidak error karena submodule `service_layer` meskipun private, posisinya adalah masih dalam satu module scope yang sama dengan fungsi `say_hello`.
  - Dimisalkan jika `service_layer::some_black_magic` dipaksa diakses dari `main`, maka muncul error karena `service_layer` adalah private dan posisinya tidak berada dalam module scope yang sama dengan crate root (`main`).

- Fungsi `messaging::say_hello` adalah **public**. Penjelasan:

- Fungsi ini merupakan module item milik `messaging`.
- Fungsi ini **bisa diakses** dari *current module scope* (`messaging`).
- Fungsi ini **bisa diakses** dari submodule milik *current module*, yaitu submodule dari `messaging`.
- Fungsi ini **bisa diakses** dari luar *current module scope* (`messaging`). Contohnya bisa dilihat pada crate root fungsi `main`, disitu ada pemanggilan statement `say_hello`.

A.28.3. Re-export item

Pada contoh, fungsi `messaging::say_hello` didesain sebagai media untuk mengakses fungsi `some_black_magic`. Di situasi *real world* pastinya sangat jarang terjadi sebuah fungsi isinya hanya satu baris pemanggilan fungsi lainnya. Jika memang ada situasi seperti itu, (kontekstual) lebih baik hapus saja fungsi yang jadi media pemanggilan dan langsung saja panggil fungsi didalamnya sesuai kebutuhan.

Pada praktek selanjutnya ini kita misalkan bahwa fungsi `say_hello` isinya memang cuma 1 baris, dan yang paling penting adalah isi fungsi `some_black_magic` perlu untuk bisa diakses dari `main`. Untuk kasus seperti ini ada 3 alternatif solusi:

1. Tidak perlu mengubah apapun, gunakan saja kode yang sudah ditulis di atas. Kode tersebut sudah bisa mengakomodir pemanggilan `some_black_magic` via `say_hello`.
2. Atau, hapus saja fungsi `say_hello`, lalu ubah visibility module `service_layer` menjadi publik, dengan demikian kita bisa mengakses `some_black_magic` dari `main` menggunakan path `messaging::service_layer::some_black_magic`.
3. Atau, gunakan teknik **re-export item**.

Re-export item adalah sebuah cara untuk mem-*bypass* pengaksesan item yang secara hirarki memang tidak bisa diakses dari luar module (bisa jadi karena visibility item ataupun parent module nya adalah private). Dengan teknik ini, maka item pasti bisa diakses dari luar module.

Item yang di-re-export akan menjadi item milik *current module* dimana statement re-export tersebut ditulis.

Cara re-export item adalah menggunakan keyword `pub use` kemudian diikuti dengan path yang ingin di-export dan juga nama export item dengan notasi penulisan berikut:

```
pub use the_path as exported_name;
pub use self::service_layer::some_black_magic as say_hello;
```

Contoh jika diterapkan pada program yang sebelumnya sudah ditulis:

src/messaging.rs

```
pub use self::service_layer::some_black_magic as say_hello;

const SOME_MESSAGE: &str = "hello rust";

mod service_layer {

    pub fn some_black_magic() {
        println!("{}", crate::messaging::SOME_MESSAGE)
    }
}
```

src/main.rs

```rust
mod messaging;

fn main() {
    messaging::say_hello();
}
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrograman rust\dasarpe
   Compiling visibility_privary_2 v0.1.0 (D:\Labs\Adam Studio\E
    Finished dev [unoptimized + debuginfo] target(s) in 0.43s
     Running `target\debug\visibility_privacy_2.exe`
hello rust
```

Bisa dilihat di contoh di atas, fungsi `say_hello` dihapus, kemudian item `service_layer::some_black_magic` di-re-export dengan nama `say_hello`. Dengannya kita bisa mengakses `some_black_magic` dari luar module `messaging` menggunakan path `messaging::say_hello`.

Jika item ingin di-re-export tanpa perubahan nama item, bisa gunakan notasi berikut:

```rust
pub use the_path;
pub use self::service_layer::some_black_magic;
```

Jika diterapkan pada program sebelumnya, kurang lebih seperti ini:

src/messaging.rs

```rust
pub use self::service_layer::some_black_magic;

const SOME_MESSAGE: &str = "hello rust";

mod service_layer {
```

src/main.rs

```rust
mod messaging;

fn main() {
    messaging::some_black_magic();
}
```

> *Lebih jelasnya mengenai keyword `use` dibahas pada chapter Module System → Use*

A.28.4. Public visibility scope

Keyword `pub` digunakan untuk mengubah visibility item menjadi publik. Keyword ini bisa dikombinasikan dengan salah satu dari keyword `self`, `crate`, dan `super`; denganya kita bisa menentukan visibility sebuah publik item dengan scope yang lebih spesifik.

◉ Keyword `pub`

Penulis rasa untuk penerapan keyword `pub` ini sudah sangat jelas, kita beberapa kali mempraktekannya.

Dengan keyword `pub`, sebuah item visibility-nya menjadi publik.

◉ Keyword `pub(in path)`

Keyword ini menjadikan visibility item hanya didalam `path` yang ditulis di `pub(in path)`, dengan ketentuan `path` tersebut merupakan parent dari

module item dimana keyword digunakan.

Contohnya bisa dilihat pada kode berikut. Fungsi `say_hello` didefinisikan publik dengan scope path ditentukan secara eksplisit adalah `crate::outer_mod`. Dengan demikian fungsi `say_hello` hanya bisa diakses dari dalam `outer_mod`.

Bisa dilihat di contoh, fungsi `say_hello` diakses dari `do_something`. Silakan coba saja paksa untuk mengaksesnya dari fungsi `main`, hasilnya pasti error.

```
pub mod outer_mod {

    pub mod inner_mod {

        // fungsi say_hello berikut hanya bisa diakses dari dalam
`outer_mod`.
        // pengaksesannya dari luar `outer_mod` menghasilkan
error.
        pub(in crate::outer_mod) fn say_hello() {
            println!("hello rust")
        }
    }

    pub fn do_something() {
        inner_mod::say_hello();
    }
}

fn main() {
    outer_mod::do_something();
}
```

◉ Keyword `pub(crate)`

Keyword `pub(crate)` digunakan untuk membuat visibility item menjadi publik dengan scope akses *current crate*. Dengan ini item bisa diakses dari manapun asalakan masih dalam crate yang sama.

Contoh penerapannya bisa dilihat berikut ini. Fungsi `say_hello` visibility scope nya ditentukan adalah *current crate*. Fungsi tersebut bisa diakses dari `outer_mod_one::do_something`, dari `outer_mod_two::do_something`, dan juga dari fungsi `main`.

```
pub mod outer_mod_one {

    pub mod inner_mod {

        // fungsi ini visibility scope-nya di level crate
        pub(crate) fn say_hello() {
            println!("hello rust")
        }
    }

    pub fn do_something() {
        inner_mod::say_hello();
    }
}

pub mod outer_mod_two {

    pub fn do_something() {
        crate::outer_mod_one::inner_mod::say_hello();
    }
}
```

◉ Keyword `pub(super)`

Keyword `pub(super)` digunakan untuk membuat visibility item menjadi publik dengan scope akses *parent module*.

Pada contoh berikut, fungsi `say_hello` visibility scope nya ditentukan adalah *parent module*, artinya fungsi tersebut hanya bisa diakses dari dalam *parent module* (yang pada konteks ini adalah `outer_mod`).

```
pub mod outer_mod {

    pub mod inner_mod {

        // fungsi ini visibility scope-nya di parent module scope,
        // yaitu `outer_mod`
        pub(super) fn say_hello() {
            println!("hello rust")
        }
    }

    pub fn do_something() {
        inner_mod::say_hello();
    }
}

fn main() {
    outer_mod::do_something();
}
```

◉ Keyword `pub(self)`

Keyword ini digunakan untuk membuat visibility item menjadi publik dengan scope akses hanya pada current module. Contohnya bisa dilihat pada kode

program berikut.

Fungsi `say_hello` visibility scope-nya adalah *current module*. Fungsi tersebut hanya bisa diakses dari tempat yang merupakan module item dari *current module* yaitu `inner_mod`.

```
pub mod outer_mod {

    pub mod inner_mod {

        // fungsi ini visibility scope-nya di current module
scope,
        // yaitu `inner_mod`
        pub(self) fn say_hello() {
            println!("hello rust")
        }

        pub fn do_something() {
            say_hello();
        }
    }
}

fn main() {
    outer_mod::inner_mod::do_something();
}
```

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-
example/../visibility_privacy
```

◉ Referensi

- https://doc.rust-lang.org/reference/visibility-and-privacy.html
- https://aloso.github.io/2021/03/28/module-system.html

A.29. Module System → Use, Import, Re-export

Keyword `use` digunakan untuk dua hal, yaitu *import* path dan *re-export* path. Sebenarnya kita telah mempelajari kedua penerapan tersebut pada beberapa chapter sebelumnya. Pada chapter ini kita akan ulang lagi pembahasan agar lebih jelas.

A.29.1. Keyword `use` untuk import path

Untuk bisa menggunakan sebuah item dari crate lain, entah itu dari rust standard library crate maupun 3rd-party, caranya cukup dengan menuliskan path item. Contohnya bisa dilihat dibawah ini, fungsi `current_dir` digunakan untuk mengambil path dari current directory. Fungsi tersebut merupakan item dari module `std::env`, maka untuk mengaksesnya kita harus menuliskan path secara lengkap.

```
let package_path = std::env::current_dir().unwrap();
println!("package_path: {:?}", package_path);
```

Selain cara di atas, bisa juga gunakan keyword `use` untuk meng-import fungsi `current_dir` ke kode program, dengannya kita tidak perlu menulis path secara lengkap. Contoh:

```rust
use std::env::current_dir;

fn main() {
    let package_path = current_dir().unwrap();
    println!("package_path: {:?}", package_path);
}
```

```
(base) PS D:\Labs\Adam Studio\Ebook\use\use_1> cargo run
   Compiling use_1 v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in 0.68s
     Running `target\debug\use_1.exe`
package_path: "D:\\Labs\\Adam Studio\\Ebook\\dasarpemrogramanrust\\dasarpemrogramanrust\\examples\\use\\use_1"
```

> *Di contoh, pengaksesan current directory di-chain dengan method* `unwrap` *karena nilai balik fungsi* `current_dir` *adalah bertipe* `std::io::Result`.
>
> *Lebih jelasnya mengenai tipe tersebut dibahas pada chapter terpisah, Tipe Data → Result*

◉ Import items dari module yang sama

Pada contoh selanjutnya ini, kita akan coba praktekan penerapan fungsi `std::env::args` untuk mengambil *argument* saat eksekusi program, kemudian menampilkannya ke layar.

Ok, berarti akan ada 2 path yang akan di-import:

```rust
use std::env::current_dir;
use std::env::args;
```

Ada notasi penulisan import path lainnya yang bisa digunakan jika path-nya

memiliki parent segment yang sama. Contohnya seperti di atas, kedua fungsi tersebut memiliki parent segment path yang sama yaitu `std::env`. Yang seperti ini bisa dituliskan juga dengan notasi penulisan import berikut:

```
use std::env::{current_dir, args};
```

Ok, lanjut ke praktek. Silakan tulis kode berikut:

```
use std::env::{current_dir, args};

fn main() {
    let package_path = current_dir().unwrap();
    println!("package_path: {:?}", package_path);

    for i in 1..=args().len() {
        let each_arg = args().nth(i);
        if each_arg != None {
            println!("arg{}: {:?}", i, each_arg.unwrap());
        }
    }
}
```

Fungsi `std::env::args` digunakan untuk mengambil argument eksekusi program baik eksekusi via `cargo run` ataupun via pemanggilan binary. Fungsi tersebut nilai baliknya adalah iterator, jadi bisa digunakan dalam blok kode `for in` dengan mudah.

Statement `args().nth(i)` mengembalikan nilai argument pada index ke-`i` dalam tipe `Option`, dan nilai tersebut bisa saja tidak ada (direpresentasikan dengan keyword `None`). Oleh karena itu penting untuk dicek terlebih dahulu menggunakan keyword `if`. Jika memang nilainya adalah selain `None`, gunakan `unwrap` untuk mengambil nilainya dalam tipe data `String`.

> - *Lebih jelasnya mengenai Iterator akan dibahas pada chapter terpisah, Trait → Iterator.*
> - *Lebih jelasnya mengenai tipe `Option` akan dibahas pada chapter terpisah, Tipe Data → Option.*

Silakan coba jalankan menggunakan dua command, yang pertama `cargo run`, kemudian `cargo run tulis argumen disini dengan pembatas spasi`.

Bisa dilihat pada gambar berikut, jika ada argument disisipkan dalam eksekusi program, maka ditampilkan. Pada gambar berikut dicontohkan argument yang dipakai adalah `hello` dan `world`.

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrograman-rust\use\use_1> cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\use_1.exe`
package_path: "D:\\Labs\\Adam Studio\\Ebook\\dasarpemrograman-rust\\dasarpemrograman-rust\\examples\\use\\use_1"

(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrograman-rust\use\use_1> cargo run hello world
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\use_1.exe hello world`
package_path: "D:\\Labs\\Adam Studio\\Ebook\\dasarpemrograman-rust\\dasarpemrograman-rust\\examples\\use\\use_1"
arg1: "hello"
arg2: "world"
```

> *Lebih jelasnya mengenai Iterator dibahas pada chapter terpisah, Trait → Iterator*

A.29.2. Keyword `use` untuk re-export path

Re-export item adalah sebuah cara untuk mem-*bypass* pengaksesan item yang secara hirarki memang tidak bisa diakses dari luar module (bisa jadi karena

visibility item ataupun parent module nya adalah private). Dengan teknik ini, maka item pasti bisa diakses dari luar module.

Item yang di-re-export akan menjadi item milik *current module* dimana statement re-export tersebut ditulis.

Keyword `pub use` digunakan untuk operasi re-export. Notasi penulisannya bisa dilihat di bawah ini:

```
// pub use the_path
pub use self::sub_module::say_hello_message;

// pub use the_path as exported_name;
pub use self::sub_module::say_hello_message as say_hello;
```

Contoh penerapan bisa dilihat pada kode berikut. Submodule `sub_module` milik module `messaging` adalah private module, yang didalamnya ada item dengan visibility publik.

Agar `say_hello_message` tidak bisa diakses dari *crate root* karena `sub_module` yang merupakan module scope item tersebut adalah private. Agar item tersebut bisa diakses dari publik, maka bisa dengan menggunakan teknik *re-export*.

src/messaging.rs

```
pub use self::sub_module::say_hello_message;

mod sub_module {

    pub fn say_hello_message() {
        println!("hello rust")
```

src/main.rs

```rust
mod messaging;

fn main() {
    messaging::say_hello_message();
}
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarpe
   Compiling visibility_privacy_2 v0.1.0 (D:\Labs\Adam Studio\El
    Finished dev [unoptimized + debuginfo] target(s) in 0.43s
     Running `target\debug\visibility_privacy_2.exe`
hello rust
```

Bisa dilihat item `say_hello_message` sekarang bisa diakses dari fungsi `main` via path `messaging::say_hello_message` setelah di-re-export.

O iya, jika suatu item ingin di-re-export dengan nama berbeda, tambahkan keyword `as` diikuti alias atau nama item export. Contoh:

src/messaging.rs

```rust
pub use self::sub_module::say_hello_message as say_hello;

mod sub_module {

    pub fn say_hello_message() {
        println!("hello rust")
    }
}
```

src/main.rs

```rust
mod messaging;

fn main() {
    messaging::say_hello();
}
```

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../use
```

◉ Referensi

- https://doc.rust-lang.org/std/keyword.use.html
- https://doc.rust-lang.org/reference/items/use-declarations.html

---

A.30. Block Expression

Pada chapter ini kita akan belajar tentang block expression.

A.30.1. Konsep dan penerapan block

Block expression (atau cukup block), adalah salah satu control flow yang ada di Rust yang berguna untuk isolasi items, variabel, ataupun proses dalam sebuah scope yang sifatnya *anonymous*.

Block expression berbeda jika dibandingkan block kode `if` (yang berarti dia adalah milik `if`) atau block kode fungsi (yang dia adalah milik fungsi), block expression tidak berasosiasi dengan keyword tertentu / *anonymous*.

> *Pada ebook ini, penulis akan menggunakan istilah **block fungsi** untuk block fungsi, **block kode if** untuk block kode seleksi kondisi `if`, dan **block kode X** untuk X.*
>
> *Untuk block expression, penulis akan gunakan istilah **block expression** atau cukup **block** saja.*

Cara penerapan block cukup dengan menuliskan kode program diapit diantara tanda `{` dan `}`. Contoh penerapannya:

```
fn main() {
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasar
   Compiling block v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpemr
    Finished dev [unoptimized + debuginfo] target(s) in 0.25s
     Running `target\debug\block.exe`

x: 24
(from block) y: 12
(from block) z: 36
```

Pada contoh di atas, dalam fungsi `main` ada variabel `x` yang dideklarasikan kemudian di-print. Setelahnya diikuti statement block expression. Bisa dilihat seperti itu kurang lebih cara penulisannya.

Di dalam block, nilai `x` di-print. Operasi seperti ini bisa dilakukan, karena *by default* semua items pada scope di luar block adalah bisa diakses dari dalam block. Contohnya adalah variabel `x` yang deklarasinya berada di luar block, tapi diakses di dalam block.

Kemudian ada beberapa variabel baru yang didefinisikan di dalam block, yaitu `y` dan `z` yang variabel tersebut juga di-print. Variabel tersebut akan valid sampai eksekusi block selesai. Intinya semua statement dalam block expression ter-isolasi dalam anonymous scope tersebut. Setelah block kode selesai dieksekusi, jika kita berusaha memanggil items yang ada di dalam block dengan pemanggilan dari luar scope, hasilnya error.

Sebagai contoh, kode berikut menghasilkan error:

```rust
let x = 24;

{
    let y = 12;
    let z = x + y;
};

println!("z: {}", z); // <------ error
```

O iya, dalam penulisannya, statement block harus diikuti tanda semicolon `;` sebagai penanda akhir statement, dengan beberapa pengecualian yang akan ikut dibahas pada section setelah ini.

A.30.2. Return value block

Sebuah block expression bisa memiliki return value, dengannya maka nilainya bisa ditampung dalam sebuah variabel. Cara penerapannya menggunakan notasi berikut:

```
let varOne = {
    // ...
    returnvalue
};

let varTwo: tipeData = {
    // ...
    returnvalue
};
```

Dalam block expression, tidak perlu menuliskan keyword `return` dan tidak perlu juga menuliskan tanda semicolon di akhir statement return value.

Berikut adalah contoh praktek return value block. Ada sebuah block yang nilai baliknya ditampung ke variabel `a`. Isi block sendiri adalah generate data numerik random, yang kemudian dikalikan dengan angka `2`, lalu dijadikan return value.

```
use rand::Rng;

fn main() {
```

A.30.3. Nested block

Block bisa berada di dalam block. Contohnya seperti kode berikut, nested block dengan kedalaman 3 layer.

```
{
    let b = 12;
    let mut total = 0;

    {
        let c = 13;

        {
            let d = 14;
            total = b + c + d;
        }
    }

    println!("{total}")
}
```

Tidak ada yang istimewa tentang cara penulisan nested block. Langsung tulis saja block expression dalam block sesuai kebutuhan.

A.30.4. Labeled block

Tak hanya perulangan, sebuah block expression juga bisa memiliki label. Cara penerapannya dengan menuliskan nama label (diawaili dengan tanda `'`), kemudian diikuti tanda `:` dan block expression. Silakan lihat notasi penulisan berikut agar lebih jelas:

```
'nama_label: {
    // ...
}
```

Mari coba terapkan. Pada kode berikut, ada block expression dengan nama `'append_with_even_number`.

Di dalam block itu, ada proses generate data numerik secara acak. Angka random tersebut kemudian di cek, jika ganjil maka block expression dihentikan secara paksa. Statement `break 'append_with_even_number` artinya menghentikan eksekusi block `'append_with_even_number`.

Jika angka genap, maka ditambahkan ke `total` kemudian di-print.

```
fn main() {
    {
        let mut total = 24;

        'append_with_even_number: {
            let n = rand::thread_rng().gen_range(0..100);

            if n % 2 == 1 {
                break 'append_with_even_number
            }

            total = n
        }

        println!("{total}");
    }
}
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasar
   Compiling block_expression v0.1.0 (D:\Labs\Adam Studio\Eboo
    Finished dev [unoptimized + debuginfo] target(s) in 0.40s
     Running `target\debug\block_expression.exe`
24
```

A.30.5. Async block

Pembahasan mengenai asynchronous block ada pada chapter Async.

A.30.6. *Unsafe* block

Pembahasan mengenai unsafe block expression ada pada chapter Safe & Unsafe.

A.30.7 Manfaat penerapan block

Block biasa diterapkan untuk isolasi sebuah proses yang tidak perlu di-reuse. Jika proses adalah di-reuse, dianjurkan untuk menggunakan fungsi dalam penerapannya.

Di bahasa pemrograman lain juga ada block yang penerapannya kurang lebih adalah sama. Namun perlu diketahui, di Rust block memiliki berbedaan yang bisa dibilang signifikan, yaitu dalam hal manajemen memory.

Rust menerapkan konsep memory management bernama **ownership**. Setiap kali Rust selesai mengeksekusi block kode, baik itu fungsi, block expression, atau jenis block lainnya; akan dilakukan evaluasi pengecekan ownership yang ada dalam block tersebut. Untuk data yang owner-nya tidak berpindah ke luar scope, maka akan dilakukan proses dealokasi memory untuk data tersebut.

Dengan approach ini penggunaan memory menjadi efisien.

- Lebih jelasnya mengenai memory management dibahas pada chapter Memory Management
- Lebih jelasnya mengenai ownership dibahas pada chapter Ownership

---

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-
example/../block_expression
```

◉ Referensi

- https://doc.rust-lang.org/reference/expressions/block-expr.html

---

A.31. Shadowing

Pada chapter ini kita akan belajar tentang shadowing pada variable dan apa perbedaannya dibanding variable mutable.

A.31.1 Konsep variable shadowing

Variable shadowing adalah istilah untuk variable yang dideklarasikan dalam sebuah block, yang variabel tersebut memiliki nama sama persis dengan variable pendahulunya (baik dalam scope yang sama ataupun variable lain yang berada di luar current scope).

Penulis tekankan disini, meskipun namanya sama, variabel-variabel tersebut dianggap variabel yang benar-benar berbeda oleh Rust, tipe datanya bisa jadi juga berbeda.

Ciri khas variable shadowing adalah deklarasi selalu menggunakan keyword `let`. Contoh penerapannya:

```
fn main() {
    let some_data = "Hello";
    println!("{}", some_data);
    // output => Hello

    let some_data = 12;
    println!("{}", some_data);
    // output => 12

    let some_data = "Rust!";
```

```
warning: `shadowing` (bin "shadowing") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.7s
     Running `target\debug\shadowing.exe`

Hello
17
Rust!
true
3.14
```

Bisa dilihat ada banyak variabel dengan nama `some_data` dideklarasikan dalam block fungsi `main`. Kesemuanya dideklarasikan menggunakan keyword `let`, ada yang immutable ada juga yang mutable.

Variable shadowing BERBEDA dengan variable mutable. Pada variable mutable, saat deklarasinya, di belakang layar terjadi proses alokasi alamat memory untuk menampung data, kemudian saat ada perubahan nilai, maka data yang baru disimpan ke alamat memory yang sama menggantikan data sebelumnya.

Pada variable shadowing, yang terjadi di balik layar adalah: ketika ada deklarasi variabel baru menggunakan keyword `let` dan namanya sama, maka dianggap sebagai variabel baru, dan Rust akan mengalokasikan alamat memory baru untuk menampung data variable baru tersebut.

Lebih jelasnya mengenai memory management akan dibahas secara terpisah pada chapter Memory Management. Untuk sekarang mari lanjut pembahasan tentang variable shadowing.

A.31.2 Shadowing pada block berbeda

Variable shadowing bisa terjadi dalam satu scope yang sama, bisa juga terjadi

pada scope yang berbeda. Pada case ke-2 (scope yang berbeda), variabel baru hanya akan valid pada block scope itu saja, dan tidak mempengaruhi data variable di luar block.

Agar lebih jelas silakan pelajari contoh berikut:

```
fn main() {
    let name = "Orgrim Doomhammer";
    let mut race = "Orc";
    let mut number = 27;

    println!("{}\t {}\t {}", name, race, number);

    {
        let name = "Sylvanas Windrunner";
        race = "Undead";
        let number = 24;

        println!("{}\t {}\t {}", name, race, number);
    }

    println!("{}\t {}\t {}", name, race, number);
}
```

```
warning: `shadowing` (bin "shadowing") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.25s
     Running `target\debug\shadowing.exe`
Orgrim Doomhammer       Orc     27
Sylvanas Windrunner     Undead  24
Orgrim Doomhammer       Undead  27
```

Pada contoh di atas, ada 3 variabel dideklarasikan kemudian di-print, yaitu `name`, `race`, dan `number`. Variabel `name` adalah immutable, dan 2 variable lainnya adalah mutable. Ok, kita akan bahas variable tersebut satu per satu.

◉ Variable `name`

Variabel `name` dideklarasikan dengan nilai awal adalah `Orgrim Doomhammer`, kemudian diikuti block statement yang isinya adalah variable shadowing bernama `name` dengan value `Sylvanas Windrunner`. Variabel tersebut ketika di-print dalam block, nilainya adalah `Sylvanas Windrunner`.

Setelah eksekusi block selesai, apapun yang ada di-dalam block tersebut dianggap selesai, dan tidak mempengaruhi nilai di scope atasnya. Pada contoh, nilai variable `name` setelah eksekusi block adalah tetap `Orgrim Doomhammer` (tidak berubah menjadi `Sylvanas Windrunner`).

◉ Variable `race`

Variabel `race` dideklarasikan mutable dengan nilai awal `Orc`. Di dalam block baru, variabel tersebut diubah nilainya. Perhatikan baik-baik statement `race = "Undead"`, statement ini adalah perubahan nilai, bukan deklarasi variable shadowing. Efeknya apa? setelah block selesai dieksekusi, nilai `race` adalah `Undead` karena memang sengaja diubah nilainya dari dalam block.

Penulis tekankan, variabel shadowing ciri khasnya adalah pasti diawali keyword `let`. Jika tidak ada keyword tersebut, maka statement adalah operasi perubahan nilai, bukan deklarasi variable shadowing.

◉ Variable `number`

Variabel `number` dideklarasikan mutable dengan nilai awal `27`. Kemudian pada block terjadi deklarasi variable shadowing `let number = 24`. Meskipun variabel ini adalah mutable, yang terjadi di dalam block adalah variable shadowing karena statement diawali dengan keyword `let`.

Setelah eksekusi block selesai, nilai `number` pada scope tidak berubah, yaitu `27`.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../shadowing
```

◉ Referensi

- https://doc.rust-lang.org/rust-by-example/variable_bindings/scope.html

---

A.32. Basic Memory Management

Pada chapter ini kita akan belajar tentang salah satu hal penting dalam topik pemrograman secara general, yaitu *memory management* yang mencakup pembahasan tentang alamat memori, pointer, heap, stack, dan juga error-error yang terjadi karena kesalahan dalam manajemen memori.

Di bahasa pemrograman high-level biasanya topik tersebut jarang disentuh, tetapi di Rust yang notabene adalah system programming, hal di atas wajib untuk dipelajari.

> *Penulis tekankan bahwa mungkin pembelajaran pada bagian ini akan terasa seperti oversimplified karena tujuannya adalah untuk para pembaca yang masih dalam proses belajar atau malah belum mengenal sama sekali tentang manajemen memori.*

A.32.1. 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, diantaranya adalah berikut:

◉ Garbage collection (GC)

GC adalah metode manajemen memori otomatis pada bahasa pemrograman. GC memiliki suatu unit yang disebut dengan *garbage collector*. Collector tersebut aktif memonitor program, dan pada periode atau event tertentu ia akan berusaha untuk mengambil kembali (reclaim) memory yang sebelumnya telah dialokasikan dengan catatan memori tersebut sudah tidak lagi digunakan. Proses ini disebut dengan dealokasi memory.

Proses dealokasi pada GC terjadi di belakang layar secara asynchronous.

Beberapa bahasa pemrograman yang menerapkan GC diantara adalah Java, C#, Go, Lisp, dan banyak bahasa lainnya.

◉ Automatic reference counting (ARC)

ARC adalah metode manajemen memori yang diterapkan pada bahasa Objective-C dan Swift. Cara ARC me-manage memory adalah dengan mencatat *reference* object dan segala aktifitas yang terjadi pada object tersebut.

Di ARC, ada satuan yang disebut dengan *retain count* yang merupakan representasi jumlah banyaknya variabel atau object yang memegang suatu *reference*. Ketika *reference* sudah pindah ke luar scope atau dihapus isinya dan dilihat pada catatan rupanya tidak ada variabel yang memegang *reference* tersebut, maka dilakukan proses dealokasi memory.

Dalam bahasa yang menerapkan ARC, programmer dianjurkan untuk perhatian dan bijak dalam pengalokasian variabel beserta nilainya. Mana data yang diperlukan untuk di-retain secara *strong* dan mana yang tidak, harus pas sesuai dengan kebutuhan. Jika tidak hati-hati maka program mempunyai resiko lebih tinggi untuk menemui error *deadlocks* ataupun *memory leaks* (yang juga

akan dibahas pada chapter ini).

◉ Manual memory management

Manual memory management berarti programmer dibebani secara penuh dalam hal manajemen memori, mengharuskan programmer untuk super hati-hati dalam pengalokasian memory, kapan waktunya, dimana alokasinya (apakah *heap* atau *stack*), dan kapan harus melakukan operasi dealokasi memory.

Metode manajemen memori ini dipakai dalam system programming contohnya bahasa C dan C++.

◉ Ownership rules

Manajemen memori yang dilakukan dengan menerapkan konsep *ownership* beserta aturan-aturannya. Metode manajemen memori ini adalah yang digunakan di Rust.

> *Lebih jelasnya mengenai ownership rules pada Rust dibahas pada chapter selanjutnya, yaitu Ownership. Untuk sekarang silakan selesaikan terlebih dahulu pembahasan chapter ini.*

A.32.2. Memory Address

Memory address atau alamat memori adalah sebuah lokasi spesifik di memori yang digunakan oleh software maupun hardware untuk menyimpan suatu data.

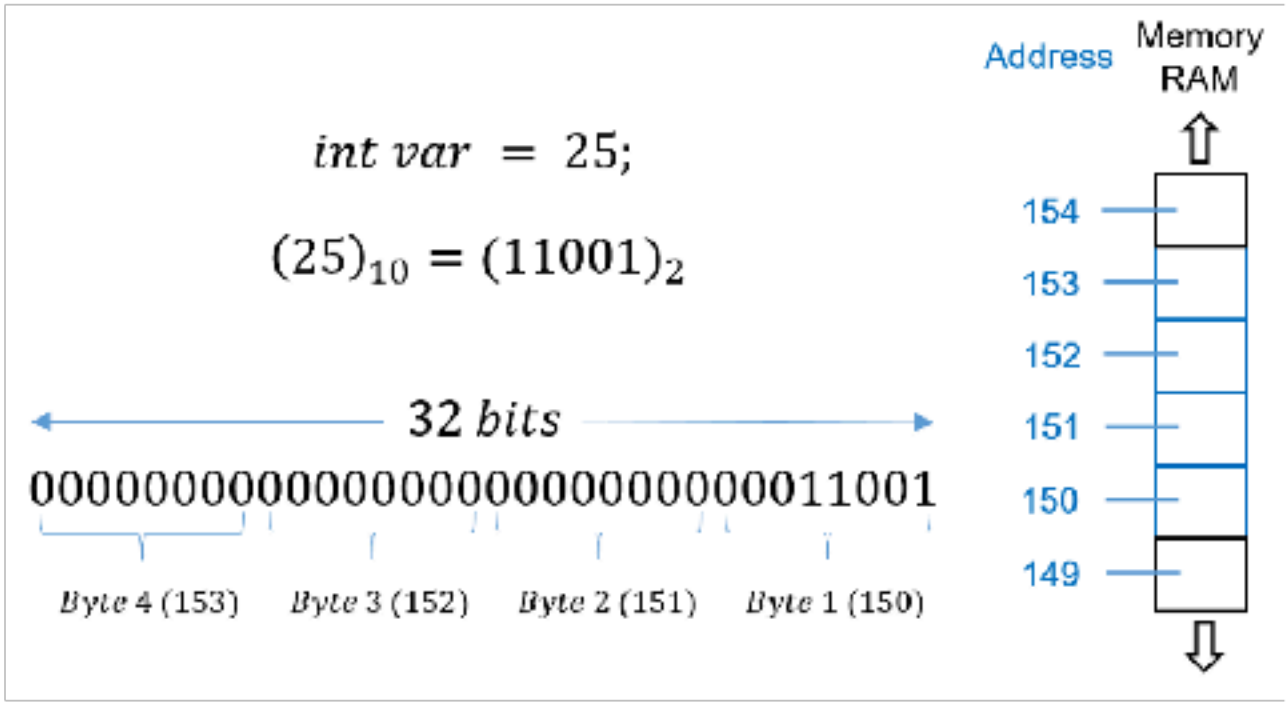
Pembahasan mengenai memory address ini sangatlah luas. Pada chapter ini

kita tidak akan membahasnya secara menyeluruh, melainkan hanya poin penting yang perlu diketahui dan dibutuhkan dalam proses pembelajaran.

Ok lanjut ke contoh agar lebih jelas. Dimisalkan ada sebuah variabel bertipe data numerik `i32`, variabel tersebut akan membutuhkan sejumlah bit alokasi alamat memori untuk bisa menyimpan value-nya yang pada contoh ini adalah `32 bits` (karena tipenya `i32`) atau jika dikonversi ke bentuk *bytes* adalah `4 bytes`.

Alokasi memory address mengacu ke tipe data (bukan value), sebagai contoh pada data bertipe `i32` maka berapapun value data tersebut (entah `1`, `2`, `1999999`, atau lainnya) tetap membutuhkan `32 bits` alokasi alamat memori untuk menyimpan data tersebut.

Silakan perhatikan ilustrasi berikut agar lebih jelas. Gambar diambil dari post medium.com/@luischaparroc.



Pada contoh di atas, variabel adalah bertipe data `i32`, maka di memory

dialokasikanlah alamat memory dengan lebar 32 bit.

Nilai variabel tersebut adalah `25`, yang jika dikonversi ke bentuk binary adalah `11001`. Dengan ini maka pada 32 bit yang sudah dialokasikan, akan terisi dengan nilai `11001`. Penulisannya dari kanan dan jika ada slot kosong sebelah kiri maka terisi dengan `0`.

Hasilnya adalah angka biner berikut:

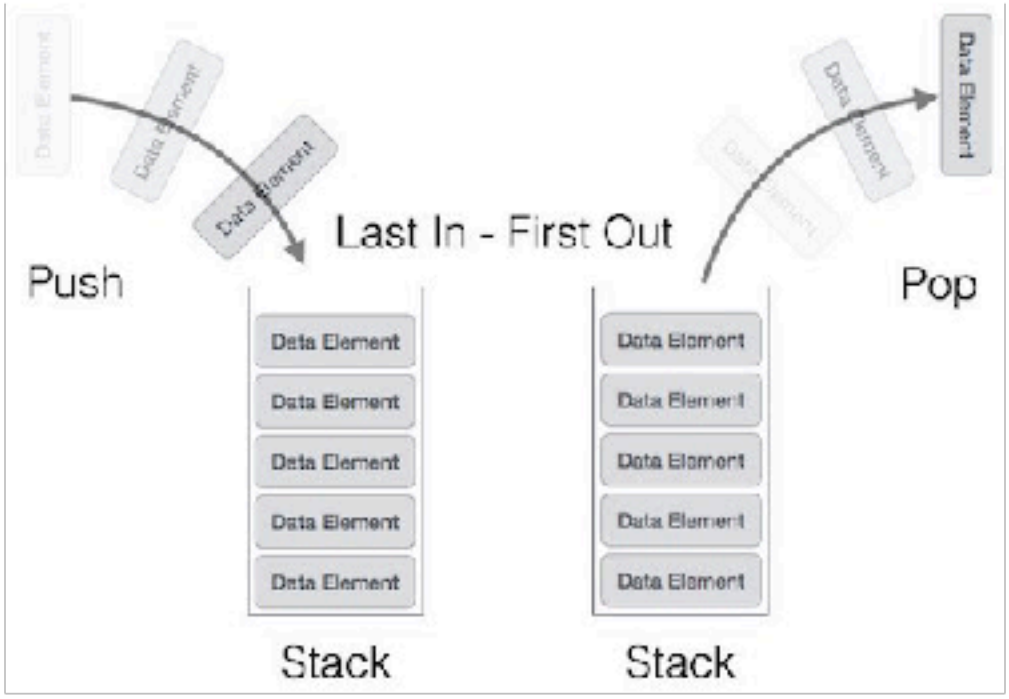
```
00000000000000000000000000011001
```

A.32.3. Stack memory

Masih dalam topik manajemen memori, ada dua hal lagi yang sangat penting untuk diketahui, yaitu *stack* dan *heap*. Keduanya adalah bagian dari memory, tempat dimana alokasi dilakukan.

Data disimpan dalam stack memory dalam bentuk stack. Karakteristik dari stack:

- Di Rust programming, stack digunakan sebagai default tempat alokasi memori
- Data yang terakhir masuk adalah yang pertama akan keluar (LIFO)
- Data yang disimpan diketahui size/ukurannya, dan memiliki batas
- Alokasi bersifat lokal terhadap pemanggilan fungsi
- Kecepatan pengaksesan data sangat tinggi



Data untuk tipe primitif (seperti `i32`, `bool`, dll) disimpan di stack.

◉ Contoh ke-1

Selanjutnya kita akan pelajari secara garis besar tentang bagaimana sebuah data dialokasikan di stack. Silakan mulai dengan mempelajari kode sederhana berikut (tanpa perlu dipraktekan), kemudian ikuti pembahasan setelahnya.

```
fn main() {
    let x = 24;
}
```

Fungsi `main` hanya berisi 1 buah data, yaitu variabel `x` dengan nilai `24`. Ketika fungsi tersebut dipanggil, maka data variabel `x` disimpan di stack (karena di Rust by default semua data disimpan di stack). Nilai `x` adalah numerik bertipe `i32`, maka compiler akan mengalokasikan 32 bits di stack memory untuk menyimpan nilai `24`.

Pada catatan karakteristik stack di atas, telah disinggung bahwa alokasi data di stack bersifat lokal terhadap pemanggilan fungsi, artinya apa? → Semua data dalam suatu blok fungsi akan disimpan dalam sebuah group yang disebut dengan *stack frame*.

Pada contoh di atas, ketika fungsi `main` dipanggil, sebuah stack frame terbuat, dan data variabel `x` disimpan dalam stack frame tersebut.

No.	Variabel	Nilai	*Stack frame*
0	x	24	milik fungsi `main()`

> *Kolom `No.` di atas merepresentasikan urutan data dalam stack agar lebih mudah dipahami*

◉ Contoh ke-2

Ok, sekarang mari lanjut contoh ke-2 berikut ini agar makin jelas.

```
fn do_something() {
    let y = 13;
    let z = 11;
}

fn main() {
    let x = 24;
    do_something();
    let a = 4;
    let b = 18;
}
```

Program sederhana di atas memiliki dua buah fungsi, `main` dan `do_something`. Saat program dijalankan, lebih tepatnya saat fungsi `main` dipanggil maka sebuah stack frame dibuat dan data dalam fungsi tersebut dialokasikan pada stack frame.

Perlu diketahui bahwa eksekusi statement dalam fungsi adalah per baris, dimulai dari atas. Dengan ini maka data yang pertama dialokasikan ke memory adalah variabel `x`.

No.	Variabel	Nilai	*Stack frame*
0	x	24	milik fungsi `main()`

Setelah itu, lanjut ke statement ke-2 yaitu pemanggilan fungsi `do_something`.

Kembali ke teori, bahwa alokasi data stack adalah bersifat lokal terhadap pemanggilan fungsi. Maka dibuatlah stack frame baru untuk pemanggilan fungsi `do_something` dengan isi adalah alokasi data `y` dan `z`.

No.	Variabel	Nilai	*Stack frame*
2	**z**	**11**	**milik fungsi `do_something()`**
1	**y**	**13**	**milik fungsi `do_something()`**
0	x	24	milik fungsi `main()`

Setelah eksekusi blok kode do_something selesai maka stack frame pemanggilan fungsi `do_something` akan di-dealokasi, dikosongkan, dihapus.

> *Sebenarnya tidak se-sederhana itu proses dealokasi memori, ada pengecekan ownership yang harus dilakukan terlebih dahulu. Namun agar makin tidak bingung, untuk sementara kita gunakan penjelasan di atas.*
>
> *Silakan selesaikan dulu pembahasan chapter ini, kemudian pada chapter selanjutnya kita akan bahas tentang apa itu Ownership.*

Ok, maka dari 2 stack yang sebelumnya ada, sekarang tinggal 1 stack frame saja yaitu milik pemanggilan fungsi `main`.

Dari yang sebelumnya ...

No.	Variabel	Nilai	*Stack frame*
~~2~~	~~z~~	~~11~~	~~milik fungsi `do_something()`~~
~~1~~	~~y~~	~~13~~	~~milik fungsi `do_something()`~~
0	x	24	milik fungsi `main()`

... sekarang menjadi ...

No.	Variabel	Nilai	*Stack frame*
0	x	24	milik fungsi `main()`

Ok, lanjut ke block fungsi `main` berikutnya, yaitu `let a = 4`. Saat dipanggil maka ada penambahan data baru pada stack frame pertama.

No.	Variabel	Nilai	*Stack frame*
3	**a**	**4**	**milik fungsi `main()`**
0	x	24	milik fungsi `main()`

Kemudian sampai di statement terakhir fungsi `main`. Ketika `let b = 14` dipanggil maka ada penambahan data baru pada stack frame pertama.

No.	Variabel	Nilai	*Stack frame*
4	**b**	**18**	**milik fungsi `main()`**
3	a	4	milik fungsi `main()`
0	x	24	milik fungsi `main()`

Setelah fungsi selesai dieksekusi, stack frame di-dealokasi.

Kurang lebih seperti itu sekilas peran dari stack dalam Rust programming. Selanjutnya kita bahas tentang heap memory.

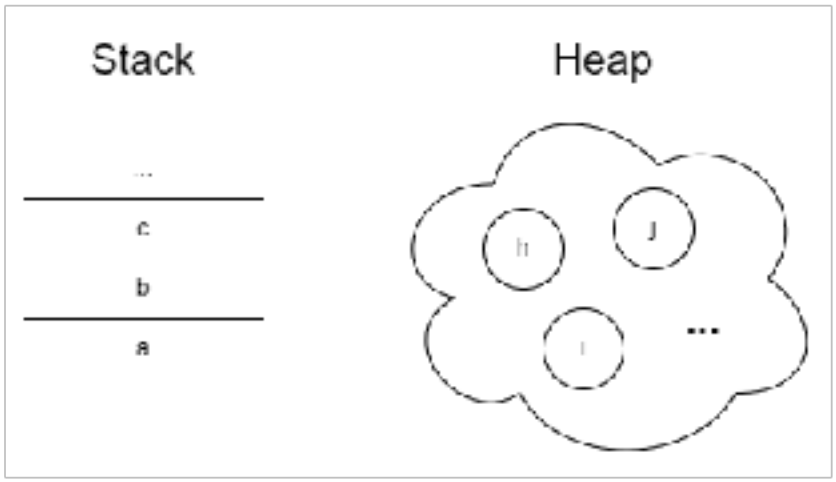
A.32.4. Heap memory

Heap adalah salah satu tempat alokasi memory selain stack. Karakteristik dari heap:

- Heap digunakan untuk alokasi data yang sifatnya dinamis, tidak diketahui size-nya, atau bisa berubah size-nya

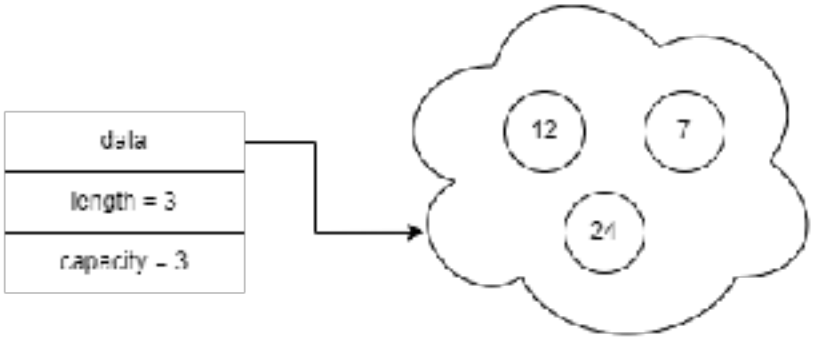
- Data di heap tidak memiliki pattern tertentu
- Alokasi dan dealokasi data di heap bisa dilakukan kapanpun
- Kecepatan pengaksesan data di heap lebih lambat dibanding stack

Ilustrasi perbandingan stack dan heap:



Tipe data non-primitive di Rust data-nya disimpan di heap, contohnya seperti Vector, `string`, dan beberapa lainnya. Penulis tekankan, bahwa **yang disimpan di heap adalah data-nya saja, sedangkan atribut lainnya (seperti `length` dan `capacity`) tetap disimpan disimpan di stack**. Lebih jelasnya silakan lihat ilustrasi berikut:

```
let numbers = vec![12, 24, 7];
```



Heap, selain digunakan sebagai penyimpanan data dinamis, kita juga bisa manfaatkan sebagai penyimpanan data jenis lainnya (secara eksplisit) menggunakan tipe data `Box`.

Pembahasan Lebih jelasnya mengenai heap allocation ada pada chapter String Custom Type vs `&str` dan juga Box. Namun untuk sekarang, penulis anjurkan untuk lanjut ke pembahasan berikutnya terlebih dahulu.

A.32.5. Error memory management

Rust sangat disiplin dalam hal penulisan source code, terutama untuk kode-kode yang berhubungan dengan memory management. Hal ini dilakukan oleh Rust untuk meminimalisir munculnya error seperti memory leak dan sejenisnya.

Namun meski demikian, potensi error memory tetap ada, dan kita akan bahas itu nantinya setelah masuk chapter Safe & Unsafe.

Nantinya akan dibahas juga tentang beberapa error saat compile time yang error tersebut berhubungan dengan memory management, yaitu pada chapter Ownership dan Borrowing. Untuk sekarang, mari lanjut ke chapter berikutnya terlebih dahulu.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-
```

◉ Referensi

- https://doc.rust-lang.org/nomicon/ownership.html
- https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html
- https://doc.rust-lang.org/1.22.0/book/first-edition/the-stack-and-the-heap.html
- https://doc.rust-lang.org/rust-by-example/std/box.html
- https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html
- https://docs.elementscompiler.com/Concepts/ARC/
- https://en.wikipedia.org/wiki/Memory_address
- https://en.wikipedia.org/wiki/Code_segment
- https://en.wikipedia.org/wiki/Automatic_Reference_Counting
- https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)
- https://log2base2.com/C/pointer/computer-memory-address-basics.html
- https://quora.com/How-does-memory-management-work-in-Rust
- https://medium.com/the-legend/management-memori-pada-bahasa-pemrograman-a33772635aa5
- https://imam.digmi.id/post/belajar-rust-memahami-stack-dan-heap/
- https://javatpoint.com/rust-ownership
- https://stackoverflow.com/questions/24158114

A.33. Pointer & References

Chapter ini membahas tentang apa itu pointer, references, dan dereferences. Pembelajaran dimulai tentang konsep dan praktek dasar tentang ketiga topik tersebut, kemudian diikuti dengan pembahasan tentang karakteristik pointer & reference.

> *Penulis tekankan bahwa* ***pada chapter ini, konsep reference yang dibahas adalah dalam konteks programming secara general.*** *Tidak dari sudut pandang ownership.*

Reference pada Rust memiliki keunikan, tapi kita akan bahas itu secara terpisah pada chapter Borrowing. Untuk sekarang silakan ikuti terlebih dahulu pembahasan chapter-per-chapter secara berurutan.

A.33.1. Pointer

Pointer artinya adalah alamat memori. Variabel pointer artinya adalah variabel yang berisi alamat memory (hanya alamat memory-nya saja, bukan value yang sebenarnya).

Di Rust, variabel pointer ditandai dengan adanya karakter `&` pada tipe data. Sebagai contoh `&i32` artinya adalah tipe data pointer `i32`. Contoh lain adalah `&bool` yang merupakan tipe data pointer `bool`.

> *Ada dua jenis tipe pointer di Rust programming, yaitu smart pointer dan raw pointer. Pada chapter ini kita tidak membahasnya karena termasuk topik yang cukup advance. Pembahasan akan ada pada chapter terpisah Smart Pointer vs Raw Pointer.*

A.33.2. Reference (operator `&`)

Ok, selanjutnya apa itu *reference*? Istilah ini sudah beberapa kali disinggung pada chapter sebelum-sebelumnya.

Reference artinya adalah pointer dari sebuah variabel atau data. Operasi pengambilan pointer dari variabel disebut dengan *referencing*, caranya dilakukan dengan menggunakan karakter `&`.

O iya, semua jenis variabel bisa diambil nilai pointernya. Contohnya bisa dilihat berikut ini:

```
let number: i32 = 24;
println!("value: {:?}", number);

let pointer_number: &i32 = &number;
println!("pointer: {:p}", pointer_number);
```

Pada contoh di atas, sebuah variabel dideklarasikan bernama `number` dengan tipe data adalah numerik dan value `24`. Variabel tersebut jika di-print akan muncul nilainya, yaitu `24`.

Ada satu lagi variabel yang dideklarasikan yaitu `pointer_number`, yang nilainya adalah *reference* dari variabel `number`. Cara pengambilan reference yabng dilihat pada contoh, yaitu dengan menambahkan operator `&` pada

variabel yang ingin dimabil pointernya.

```
// variabel pointer_number nilainya adalah reference variabel
number.
// contoh statement:
let pointer_number_1 = &number;
let pointer_number_2: &i32 = &number;
```

By default, ketika variabel pointer di print, yang muncul adalah *underlying value* atau nilai sebenarnya, yang pada contoh di atas adalah `24`. Untuk menampilkan alamat memory gunakan formatted print `{:p}`.

Coba jalankan program kemudian lihat hasilnya. Value muncul sesuai dengan yang ditulis, sedangkan pointer memunculkan data alamat memory yaitu `0xa3bbeff6dc`.

```
src > main.rs > main
    ▶ Run | Debug
1   fn main() {
2       let number: i32 = 24;
3       println!("value: {:?}", number);
4
5       let pointer_number: &i32 = &number;
6       println!("pointer: {:p}", pointer_number);
7
PROBLEMS   OUTPUT   GITLENS   JUPYTER   DEBUG CONSOLE   TERMINAL

(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarpemro
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\heap_stack.exe`
value: 24
pointer: 0xa3bbeff6dc
```

Seperti yang sudah dijelaskan di awal bahwa variabel pointer isinya adalah alamat memory. Penulisannya dalam notasi heksadesimal diawali dengan karakter `0x`. Contohnya adalah nilai `0xa3bbeff6dc` yang merupakan alamat

memory, yang alamat tersebut adalah reference ke pemilik data sebenarnya (yaitu variabel `number`).

> *Di environment lokal masing-masing, alamat memory yang muncul sangat mungkin berbeda karena alokasi adalah random*

A.33.3. Dereference (operator `*`)

Kita sudah belajar cara mengambil data pointer dari sebuah variabel. Pada bagian ini kita belajar cara mengambil nilai sebenarnya atau underlying value dari sebuah pointer (istilahnya adalah *dereference*).

Cara melakukan operasi *dereferencing* adalah dengan menambahkan karakter `*` pada variabel pointer. Contohnya seperti berikut, silakan tambahkan kode ini ke program yang sudah ditulis.

```
// variabel underlying_value nilainya adalah nilai sebenarnya
dari pointer pointer_number
let underlying_value = *pointer_number;
println!("value: {:}", underlying_value);
```

```
src > main.rs > main
Run | Debug
fn main() {
    let number: i32 = 24;
    println!("value: {:?}", number);

    let pointer_number: &i32 = &number;
    println!("pointer: {:p}", pointer_number);

    let underlying value: i32 = *pointer number;
    println!("value: {:}", underlying_value);

PROBLEMS   OUTPUT   CODELENS   JUPYTER   DEBUG CONSOLE   TERMINAL

(base) PS D:\Labs\Adam Studio\Ubook\dasarpemrogramanrust\dasarpem
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\heap_stack.exe`

value: 24
pointer: 0xef46cff3e4
value: 24
```

Bisa dilihat, nilai dari variabel `underlying_value` adalah nilai sebenarnya dari pointer `pointer_number`, yang pastinya adalah sama dengan nilai variabel `number`.

A.33.4. Mutable References (operator `&mut`)

By default, reference sifatnya *read-only* atau *immutable*, artinya tidak bisa diubah *underlying-value*-nya.

Jika ada variabel (sebut saja X) yang merupakan underlying value dari reference variabel lain, maka perubahan value pada variabel X tersebut *default*-nya menghasilkan error.

```rust
let mut number = 24;
println!("number: {}", number);

let pointer_number = &number;
println!("pointer_number: {:p}", pointer_number);

*pointer_number = 12;

println!("*pointer_number: {}", *pointer_number);
println!("number: {}", number);
```

```
▶ Run | Debug
fn main() {
    let mut number: i32 = 24;
    println!("number: {}", number);

    let pointer_number: &i32 = &number;
    println!("pointer_number: {:p}", pointer_number);

    *pointer_number = 12;
      let pointer_number: &i32
      cannot assign to `*pointer_number`, which is behind a `&` reference
      `pointer_number` is a `&` reference, so the data it refers to cannot be
      written rustc(Click for full compiler diagnostic)
      main.rs(14, 26): consider changing this to be a mutable reference: `&mut number`
      View Problem (Alt+F8)    No quick fixes available
```

Pada gambar di atas bisa dilihat, variabel `pointer_number` merupakan reference dari variabel mutable `number`. Setelahnya ada statement `*pointer_number = 12`, yang artinya adalah underlying value dari variabel pointer `pointer_number` diakses kemudian di-isi nilainya dengan angka `12`.

> *Statement `*pointer_number = 12` tidak menghasilkan error, dan statement ini berbeda dengan `pointer_number = 12`.*

> *Statement `pointer_number = 12` menghasilkan error karena `pointer_number` adalah variabel bertipe **pointer `i32` atau `&i32` (bukan `i32`)**. Jika ingin mengubah nilainya perlu mengakses dulu underlying value-nya menggunakan `*pointer_number`.*

Ok, lalu kenapa muncul error? Di gambar terlihat ada garis merah dan popup pesan error muncul, padahal tidak ada yang salah dengan statement-nya.

Penyebab erronya bukan dari statement tersebut, tetapi pada baris statement pengambilan reference variabel `number`. Statement `&number` artinya adalah mengambil reference dari variabel `number`. Di atas sempat kita bahas bahwa *by default* sebuah reference tidak bisa diubah nilainya (*immutable*), dan ini adalah penyebab error yang dialami.

> *Silakan perhatikan pesan di popup error message agar mudah untuk tau dimana sumber masalahnya.*

Perubahan isi nilai variabel `number` tidak menghasilkan error, hal ini karena number adalah variabel `number` adalah mutable. Sedangkan operasi perubahan nilai variabel `*pointer_number` pada contoh di atas, dianggap sebagai error karena variabel `pointer_number` reference-nya adalah bukan mutable (meskipun reference diperoleh dari variabel `number` yang notabene mutable).

Solusi dari error di atas adalah menerapkan **mutable reference**. Mutable reference sama seperti reference biasa tetapi nilainya diperbolehkan untuk diubah (mutable). Caranya pengaksesan mutable reference dilakukan menggunakan operator `&mut`.

Ok, sekarang ubah statement berikut ini, lalu jalankan program:

```rust
// before
let pointer_number = &number;

// after
let pointer_number = &mut number;
```

```rust
fn main() {
    let mut number: i32 = 24;
    println!("number: {}", number);

    let pointer_number: &mut i32 = &mut number;
    println!("pointer_number: {:p}", pointer_number);

    *pointer_number = 12;

    println!("*pointer_number: {}", *pointer_number);
    println!("number: {}", number);

```

```
PROBLEMS    OUTPUT    CODELENS    JUPYTER    DEBUG CONSOLE    TERMINAL

(base) PS D:\Labs\Adam Studio\Ubook\dasarpemrogramanrust\dasarpemr
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\heap_stack.exe`
number: 24
pointer number: 0x18001196c
*pointer_number: 12
number: 12
```

Deklarasi variable pointer yang menampung mutable reference bisa via metode penulisan *manifest typing* atau *type inference*. Pada contoh berikut, variabel `pointer_number` tipe data-nya adalah mutable reference `i32`, maka penulisan tipe data adalah `&mut i32`.

```rust
// type inference
let pointer_number = &mut number;
```

Pada contoh ini kita menggunakan tipe data primitif, lalu bagaimana dengan tipe data non-primitive seperti custom type struct atau `string`, apakah penerapan mutable reference juga sama? Jawabannya tidak. Ada beberapa perbedaan dan nantinya kita akan bahas secara detail pada chapter Borrowing. Untuk sekarang, dianjurkan mengikuti pembahasan chapter per chapter secara urut.

A.33.5. Aturan Reference

Ada dua aturan penting yang harus dipatuhi dalam penerapan reference baik mutable atau immutable reference.

- Dalam waktu yang sama, hanya boleh ada satu mutable reference atau banyak immutable reference (keduanya tidak bisa bersamaan, harus salah satu).
- Reference harus selalu valid.

Mengenai penjelasan tentang dua aturan penting di atas akan bahas pada chapter Borrowing.

A.33.6. Karakteristik pointer & reference

Kita sudah belajar tentang definisi beserta cara penerapan pointer, reference, dereference, dan mutable reference. Sekarang lanjut ke pembahasan tentang karakteristik dari pointer & reference.

Pointer merupakan variabel yang isinya adalah alamat memory (bukan nilai sebenarnya). Dan reference adalah alamat memory suatu data/variabel. Dengan menerapkan keduanya kita bisa menerapkan efisiensi penggunaan

memory yang dampaknya sangat positif terhadap performa program.

Sekarang perhatikan statement berikut:

```
let number_one = 24;
let number_two = number_one;
```

Variabel `number_one` adalah data numerik bertipe `i32`, eksekusi statement tersebut menghasilkan alokasi memori selebar 32-bit.

Kemudian ada lagi statement `number_two`. Meskipun nilainya didapat dari variabel `number_one`, yang terjadi di balik layar adalah Rust akan mengalokasikan lagi alamat memory selebar 32-bit untuk menampung data `number_two` yang didapat dari hasil operasi **copy** dari variabel `number_one`.

> *Semua variabel primitif di Rust mengadopsi copy semantics, yang artinya jika variabel tersebut digunakan dalam statement assignment, maka nilai akan di-duplikasi untuk kemudian ditampung pada variabel baru.*
>
> *Lebih jelasnya mengenai copy semantics dibahas pada chapter Ownership.*

Selanjutnya, bandingkan dengan statement berikut:

```
let number_one = 24;
let number_two = &number_one;
```

Pada contoh di atas, variabel `number_one` datanya disimpan di memory dengan lebar 32-bit (masih sama seperti contoh sebelumnya). Namun ada yang berbeda dengan dengan variabel `number_two`, variabel ini adalah

variabel pointer yang nilainya reference ke alamat memory data variabel `number_one`. Yang terjadi di belakang layar, Rust tidak mengalokasikan lagi memory selebar 32-bit untuk menampung data `number_two`, melainkan menggunakan alamat memory data `number_one` sebagai reference, menjadikannya sebagai alamat tujuan variabel pointer `number_two`.

Ok, lanjut. Per sekarang, reference variabel `number_one` dan `number_two` adalah sama. Dengan ini, jika dicontohkan isi data satu variabel diubah, maka variabel lainnya juga akan berubah, hal karena reference-nya adalah sama.

◉ Contoh ke-1

Contoh penerapannya bisa kita lihat pada chapter Pointer & References chapter ini, disitu bisa dilihat ada variabel mutable `number` dan `pointer_number` yang reference-nya adalah sama dengan variabel `number`. Ketika underlying value `pointer_number` diubah (dari `24` ke `12`), isi data variabel `number` juga berubah.

```rust
fn main() {
    let mut number: i32 = 24;
    println!("number: {}", number);

    let pointer_number: &mut i32 = &mut number;
    println!("pointer_number: {:p}", pointer_number);

    *pointer_number = 12;

    println!("*pointer_number: {}", *pointer_number);
    println!("number: {}", number);
```

PROBLEMS OUTPUT GITLENS JUPYTER DEBUG CONSOLE TERMINAL

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarpemr
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\heap stack.exe`
number: 24
pointer_number: 0x18080ff96c
*pointer_number: 12
number: 12
```

◉ Contoh ke-2

Agar makin familiar dengan cara penerapan pointer & reference, silakan lanjut dengan praktek program berikut:

package source code structure

```
my_package
|—— Cargo.toml
└—— src
      └—— main.rs
```

Cargo.toml

```
# ...

[dependencies]
rand = "0.8.5"
```

main.rs

```rust
use rand::Rng;

fn main() {
    let mut number = 24;
    println!("number: {}", number);

    for _ in 0..=5 {
        change_value(&mut number);
        println!("number: {}", number);
    }
}

fn change_value(n: &mut i32) {
    *n = generate_random_number()
}

fn generate_random_number() -> i32 {
    rand::thread_rng().gen_range(0..100)
}
```

Pada sederhana di atas, ada sebuah variable mutable bernama `number` yang dideklarasikan. Variabel tersebut kemudian diakses *mutable reference*-nya untuk dijadikan argumen statement pemanggilan fungsi `change_value`.

Di dalam fungsi `change_value`, disiapkan 1 buah angka random hasil pemanggilan fungsi `generate_random_number`, yang angka tersebut kemudian

dijadikan sebagai nilai baru variable mutable reference yang mengarah ke `number`. Perubahan dalam fungsi `change_value` akan mempengaruhi data variabel `number`, karena reference-nya adalah sama.

```
    Finished dev [unoptimized + debuginfo] target(s) in 4m 04s
(base) PS D:\Labs\Wlam Studio\Ebook\dasarpemrogramanrust
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `target\debug\pointer_references_1.exe`
number: 24
number: 68
number: 79
number: 1[illegible]
number: 33
number: 85
number: 11
```

A.33.7. Reference & borrowing

Di Rust, reference (atau pengaksesan alamat memory suatu data) memiliki hubungan yang sangat erat dengan konsep **borrowing**.

Ketika kita mengambil reference suatu data, yang terjadi sebenarnya adalah kita meminjam data tersebut dari owner/pemilik sebenarnya.

Pada chapter ini kita tidak akan membahasnya lebih jauh lagi, karena akan ada chapter lain yang fokus mengulas topik tersebut secara detail, yaitu chapter Borrowing.

Untuk sekarang penulis anjurkan untuk lanjut ke chapter berikutnya secara urut.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-
example/../pointer_references
```

◉ Referensi

- https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html
- https://doc.rust-lang.org/std/marker/trait.Copy.html
- https://en.wikipedia.org/wiki/Reference_(computer_science)
- https://progressivecoder.com/understanding-rust-ownership-and-borrowing-with-examples/
- https://os.phil-opp.com/heap-allocation/

A.34. Ownership

Chapter ini berisi pembahasan tentang ownership, bagaimana Rust melakukan manajemen memory dengan menerapkan approach ownership ini.

O iya, diwajibkan untuk mempelajari terlebih dahulu tentang dasar Memory Management dan juga Pointer & References, yang keduanya adalah dibahas pada chapter sebelumnya. Jika pembaca mempelajari ebook ini secara urut maka tidak usah khawatir.

> *Pembahasan pada chapter ini adalah salah satu hal yang paling penting untuk dipahami dalam Rust programming, karena topik yang berhubungan dengan memory management adalah hal yang krusial pada system programming.*
>
> *Silakan ulang-ulang chapter ini jika diperlukan.*

A.34.1. Konsep ownership

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

Sudah disinggung pada chapter Memory Management bahwa Rust tidak menerapkan GC ataupun ARC dalam manajemen memory-nya. Rust membebankan manajemen memory pada penulis kode program, yaitu kita/programmer.

Aturan ownership ada banyak, dan programmer harus mengikutinya, karena

jika tidak, maka proses kompilasi program dan eksekusi program akan gagal dan hasilnya error.

A.34.2. Aturan ownership

Ada 3 aturan penting yang wajib diketahui:

- **Semua nilai/data/value di Rust memiliki owner**. Misal kita berbicara tentang deklarasi variabel dengan predefined value, maka value variabel adalah yang dimaksud dengan nilai/data/value, dan variabel itu sendiri adalah owner dari nilai/data/value tersebut.

- Pada waktu yang sama, **hanya boleh ada 1 owner**. Satu data, ownernya hanya satu.

- **Ketika eksekusi sebuah block scope selesai, maka owner dari data-data yang ada dalam scope tersebut akan di-drop atau di-dealokasi** (dengan pengecualian yaitu owner berpindah ke luar scope).

A.34.3. Variable scope

Sebelum kita masuk ke pembahasan yang lebih detail mengenai ownership, mari pelajari terlebih dahulu tentang apa itu variable scope.

Variable scope maksudnya adalah di block scope mana suatu variabel dideklarasikan, dan dalam block scope tersebut variabel menjadi valid (bisa digunakan). Di luar scope-nya variabel menjadi tidak valid, tidak bisa digunakan.

Agar lebih jelas, silakan perhatikan kode berikut, tidak perlu dipraktekan.

```rust
fn do_something() {
    let data_one = "one";
    // ...
}

fn main() {
    let data_two = "two";
    // ...

    {
        let data_three = "three";
        // ...
    }

    do_something();

    if true {
        let data_four = "four";
        // ...
    }
}
```

Program di atas memiliki 4 buah block kode:

- Block fungsi `main`, yang isinya adalah variabel `data_two` dan beberapa sub-block dan 1 buah pemanggilan fungsi. Di dalam block fungsi `main`, variabel `data_two` adalah valid, dan bisa digunakan dalam fungsi scope tersebut beserta sub-block lainnya (block expression dan block kode `if`).

- Block expression yang berisi variabel `data_three`. Pada block scope ini, variabel tersebut dan juga variable `data_two` adalah valid.

- Block fungsi `do_something`, yang isinya variabel `data_one`. Variabel tersebut hanya akan valid dalam block fungsi `do_something`.

- Block seleksi kondisi `if`. Variabel `data_four` berada dalam block ini, maka variabel tersebut adalah valid dalam block tersebut. Selain itu `data_two` juga valid dalam block seleksi kondisi `if` ini.

Intinya, variabel adalah valid ketika berada di dalam scope-nya (istilahnya *into scope*), dan variabel menjadi tidak valid atau invalid ketika keluar dari scope (istilahnya *out of scope*).

A.34.4. Copy semantics vs. move semantics

Di atas sudah dijelaskan mengenai aturan ownership, yang salah satunya adalah setiap data yang ada di Rust memiliki owner, dan 1 data owner pasti 1 (tidak lebih).

Agar lebih jelas mari kita perhatikan contoh berikut:

```
let x = 24;
let y = x;
println!("x: {:?}, y: {:?}", x, y);
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpem
   Compiling ownership_1 v0.1.0 (D:\Labs\Ada
    Finished dev [unoptimized + debuginfo] t
     Running `target\debug\ownership_1.exe`
x: 24, y: 24
```

Variabel `x` dideklarasikan dengan nilai `24`, artinya variabel tersebut adalah owner dari data `24`. Kemudian variabel `x` dijadikan sebagai nilai variabel baru bernama `y`, dari sini apakah berarti owner data `24` adalah berpindah dari

variabel `x` ke `y`? jawabannya adalah **tidak**.

Yang terjadi adalah data `24` milik owner `x` di-copy atau diduplikasi sebagai data baru yang owner-nya juga baru, yang pada contoh ini adalah variabel `y`.

Hasilnya, kedua variabel tersebut masing-masing adalah owner dari data yang berbeda, meskipun sumbernya adalah dari data yang sama. Perilaku ini disebut dengan **copy semantics**.

Semua tipe data primitif di Rust *by default* mengadopsi *copy semantics*. Ketika terjadi operasi assignment seperti pada contoh di atas, maka yang terjadi adalah data di-copy sebagai data baru, dengan owner baru, dan di sisi memory juga terjadi alokasi alamat baru untuk menampung data hasil copy tersebut.

> *Copy semantics by default berlaku pada tipe primitif.*
>
> *Sudah dibahas pada chapter Memory Management → Stack bahwa data primitif disimpan di stack dan pengaksesannya sangat cepat.*
>
> *Dengan ini, copy data pada tipe primitif meskipun menghasilkan alokasi memory baru, konsekuensinya tidak terlalu besar karena pengaksesannya sangat cepat.*

Ok, lalu bagaimana dengan contoh ke-2 berikut? Apakah yang terjadi juga sama?

```
let a = String::from("hello rust");
let b = a;
println!("a: {:?}, b: {:?}", a, b);
```

```
main.rs 1, M ×
src > main.rs > ...
    ▶ Run | Debug | You, 14 seconds ago | 1 author (You)
1   fn main() {
2       let a: String = String::from("hello rust");
3       let b: String = a;
4       println!("a: {:?}, b: {:?}", a, b);
let a: String

Go to String

borrow of moved value: `a`
value borrowed here after move rustc(Click for full compiler diagnostic)

macros.rs(186, 28): Error originated from macro call here

main.rs(4, 5): Error originated from macro call here

main.rs(3, 19): value moved here

main.rs(2, 9): move occurs because `a` has type `String`, which does not implement the `Copy` trait

View Problem (Alt+F8)   No quick fixes available
```

Wow, malah muncul error. Kok bisa, padahal kode program sangat straightforward.

Jadi begini, sebelumnya sudah dibahas bahwa semua tipe data primitif mengadopsi *copy semantics*. Namun untuk tipe data non-primitif (yang salah satunya adalah custom type `String`) yang diadopsi adalah **move semantics**.

Pada *move semantics*, ketika ada operasi assignment seperti `let a = b;`, maka yang terjadi adalah owner dari data string `hello rust` berpindah dari variabel `a` ke `b`. Jadinya, mulai dari statement `let a = b;` dan statement seterusnya, owner dari data string `hello rust` bukan lagi variabel `a`, tetapi variabel `b`.

Ok, sampai sini cukup jelas. Tapi kenapa bisa error? Error muncul karena variabel `a` sudah tidak bisa digunakan lagi, data-nya (string `hello rust`) sudah berpindah ke variabel lain. Variabel `a` menjadi tidak valid setelah owner-nya berpindah, dan ini adalah penyebab kenapa pemanggilan variabel `a` via macro `println` menghasilkan error.

> *Ingat, setiap data pasti punya owner, dan satu data ownernya hanya satu.*
>
> *Pada contoh di atas, owner data `hello rust` sudah berpindah dari variabel `a` ke `b`.*

Intinya, variabel yang mengadopsi *move semantics*, setiap kali ada operasi assignment maka owner akan berpindah ke variabel baru.

Jika perlu, silakan coba praktekan dengan tipe data non-primitif lainnya, seperti struct atau lainnya. Dengan pseudocode yang sama seperti contoh di atas, hasilnya adalah sama, yaitu error.

```
#[derive(Debug)]
struct MyStruct;

let g = MyStruct{};
let h = g;

println!("g: {:?}, h: {:?}", g, h);
```

```
main.rs 1, M
src > main.rs > main
fn main() {
    #[derive(Debug)]
    struct MyStruct;
    let g: MyStruct = MyStruct{};
    let h: MyStruct = g;
    println!("g: {:?}, h: {:?}", g, h);
let g: MyStruct
Go to MyStruct
borrow of moved value: `g`
value borrowed here after move rustc(Click for full compiler diagnostic)
macros.rs(106, 20): Error originated from macro call here
main.rs(6, 5): Error originated from macro call here
main.rs(5, 19): value moved here
main.rs(4, 9): move occurs because `g` has type `MyStruct`, which does not implement the `Copy` trait
```

> *Copy semantics di Rust merupakan sifat yang dimiliki oleh trait* `std::marker::Copy`.
>
> - *Semua data primitif meng-implement trait* `std::marker::Copy` *yang berarti mengadopsi copy semantics.*
> - *Data non-primitif mengadopsi move semantics.*
>
> *Lebih jelasnya mengenai traits dibahas pada chapter Traits.*

A.34.5. Alokasi & dealokasi

Ok, masuk ke pembahasan selanjutnya, yaitu tentang alokasi dan dealokasi sebuah variabel dalam scope-nya.

Disini kita akan gunakan custom type `string` sebagai contoh untuk mempelajari ownership. Untuk tipe non-primitif lainnya yang juga mengadopsi

move semantics sebenarnya bisa dijadikan contoh. Tapi penulis memilih tipe `String` karena sering digunakan.

Sekarang perhatikan kode berikut:

```
fn main() {
    do_something();
}

fn do_something() {
    let mut k = String::from("hello");

    {
        let m = String::from("hello world");
        let n = String::from("from rust");
        k = n;

        println!("{:?}", m);
    }

    println!("{:?}", k);
}
```

Fungsi `main` dideklarasikan, isinya adalah pemanggilan fungsi `do_something`.

Di dalam fungsi `do_something`, variabel `k` dideklarasikan. Statement deklarasi tersebut adalah kapan proses alokasi memory berlangsung untuk data variabel `k`.

Kemudian ada block expression, dan didalamnya terjadi lagi proses alokasi memory untuk data variabel `m` dengan nilai adalah string `hello world`, dan variabel `n` berisi string `from rust`.

Masih di dalam block expression, terjadi assignment operation, data variabel `n`

berpindah ke variabel `k`. Lalu bagaimana nasib data string `hello` yang sebelumnya dimiliki oleh `n`? Yang terjadi adalah data tersebut tidak memiliki reference, dan nantinya di akhir fungsi akan di-dealokasi. Proses dealokasi terjadi setelah fungsi `do_something` selesai dieksekusi karena variable scope data tersebut adalah di block kode fungsi `do_something`.

Ok, masih dalam block expression. Ada statement print untuk data variable `m` yang merupakan statement terakhir di block expression tersebut. Setelah eksekusi block expression selesai, yang terjadi kemudian adalah:

- data variabel `m` di-dealokasi, karena sudah *out of scope*.
- data variabel `n` tidak di-dealokasi, karena sudah berpindah scope-nya ke fungsi `do_something`. Data tersebut sekarang ownernya adalah variabel `k` yang scope-nya ada di block fungsi `do_something`.

Kemudian variabel `k` di-print, dan fungsi selesai dieksekusi. Pada moment inilah semua data dalam scope block fungsi `do_something` di-dealokasi, karena kesemua data tersebut adalah *out of scope* dan tidak ada yang berpindah ke block fungsi lainnya.

Secara garis besar seperti itu proses manajemen memory pada Rust yang menerapkan konsep *ownership*.

A.34.6. Transfer ownership

Di atas sudah dibahas bagaimana cara untuk transfer ownership data yang mengadopsi *move semantics*, yaitu cukup dengan statement assignment.

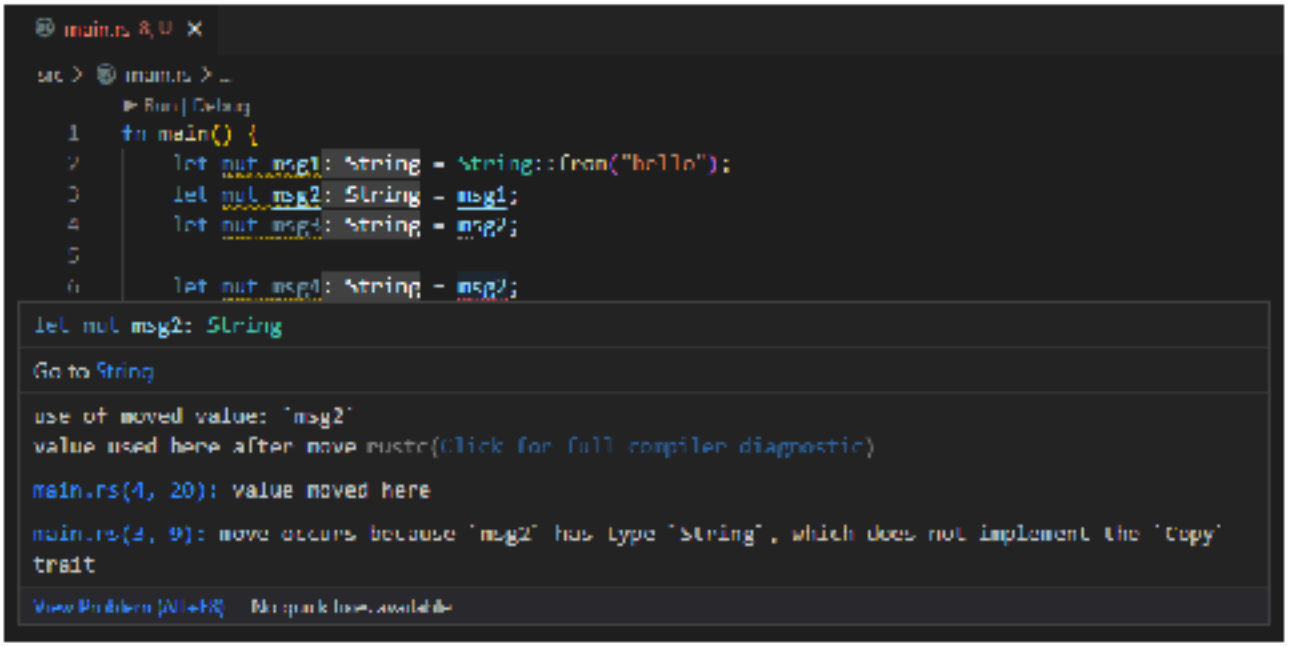
Berikut adalah contoh lain perihal transfer ownership. Data string yang owner awalnya adalah `msg1`, berpindah ke `msg2`, kemudian berpindah lagi ke `msg3`.

```rust
let msg1 = String::from("hello");
let msg2 = msg1;
let msg3 = msg2;
println!("{:?}", msg3);
```

Coba modifikasi sedikit kode tersebut, dengan menambahkan deklarasi variabel `msg4` yang nilai-nya didapat dari `msg2`.

```rust
let msg1 = String::from("hello");
let msg2 = msg1;
let msg3 = msg2;

let msg4 = msg2;
println!("{:?}", msg4);
```

Hasilnya adalah error, karena `msg2` sudah invalid.



Jika ingin memindah datanya ke `msg4`, maka gunakan statement `let msg4 =`

`msg3`.

◉ Transfer ownership via return value

Transfer ownership data yang mengadopsi *move semantics* juga bisa dilakukan antar fungsi via return value. Sebagai contoh pada kode berikut, variabel `m` yang berada di dalam block expression berpindah ke luar scope yaitu ke block fungsi `do_something` via operasi assignment `k = m`. Kemudian berpindah lagi ke fungsi `main` via return value pemanggilan fungsi `do_something`.

```
fn main() {
    let msg = do_something();
    println!("{:?}", msg);
}

fn do_something() -> String {
    let mut k = String::from("hello");

    {
        let m = String::from("hello world");
        k = m;
    }

    return k;
}
```

◉ Transfer ownership via parameter/argument

Pemanggilan fungsi dengan menyisipkan argument juga menghasilkan proses transfer ownership untuk data yang mengadopsi *move semantics*. Contoh:

```
fn main() {
```

Pada kode di atas, data variabel `msg` owner-nya berpindah ke parameter bernama `param` milik fungsi `say_hello`.

Ok, sampai disini semoga cukup jelas ya tentang bagaimana proses transfer ownership terjadi pada data yang mengadopsi *move semantics*.

Untuk data bertipe primitif (yang mengadopsi *copy semantics*) kita tidak perlu repot memikirkan dimana owner datanya, karena setiap operasi assignment, data akan di-copy dan hasilnya ada data baru dengan owner baru.

Tapi kalau dipikir-pikir justru lebih repot mengurus data yang ownernya berpindah saat assignment. Sebagai contoh, misal variabel digunakan di fungsi lain, kemudian digunakan lagi di scope asalnya. Repot juga kalau setiap saat harus dikembalikan lagi via return value.

Misalnya pada kasus berikut ini. Hasilnya pasti error, karena variabel `msg` di print setelah owner-nya berpindah ke fungsi `say_hello`.

```
fn main() {
    let msg = String::from("hello rust");
    say_hello(msg);
    println!("{:?}", msg);
}

fn say_hello(param: String) {
    println!("{:?}", param);
}
```

Lalu apa solusinya? apakah harus mengembalikannya via return value? Contohnya seperti kode berikut:

```
fn main() {
```

Boleh-boleh saja sebenarnya pakai approach tersebut, tapi malah makin repot bukan?

Ada lagi solusi lainnya yang bisa digunakan, yaitu dengan memanfaatkan method `clone` untuk cloning data.

A.34.7. Clone data

Semua tipe data yang mengadopsi *move semantics* meng-implement trait `std::clone::Clone`. Trait ini memiliki method bernama `clone` yang gunanya adalah menduplikasi atau clonging data. Cukup panggil saja method tersebut, maka data akan ter-cloning.

> *Lebih jelasnya mengenai traits dibahas pada chapter Traits.*

Kita akan terapkan pada kode sebelumnya, hasilnya kurang lebih seperti ini. Pada argument pemanggilan fungsi `say_hello` disisipkan data cloning via statement `msg.clone()`.

```
fn main() {
    let msg = String::from("hello rust");
    say_hello(msg.clone());
    println!("{:?}", msg);
}

fn say_hello(param: String) {
    println!("{:?}", param);
}
```

Ok, dengan ini masalah untuk me-reuse data yang bisa berpindah ownernya dianggap beres. Memang beres, tapi apakah cara ini baik kalau dilihat dari

sudut pandang memory management?

Konsekuensi dari cloning data adalah terjadi proses alokasi lagi di memory. Data akan di-duplikasi dan dialokasikan ke alamat memory baru, jadinya kurang efisien. Ditambah lagi, jika kita mengacu ke penjelasan pada chapter Memory Management → Heap Memory, data `String` isinya disimpan di heap memory yang pengaksesannya lebih lambat dibanding pengaksesan data stack. Dari sini bisa disimpulkan bahwa cloning bukan solusi yang paling baik (kecuali terpaksa).

Solusi yang lebih baik adalah dengan melakukan operasi pinjam data dari owner aslinya tanpa perlu melakukan operasi perpindahan owner, yang pada Rust programming disebut dengan **borrowing**.

Kita sebenarnya sudah menerapkannya beberapa kali pada chapter yang lalu, tapi kita akan bahas lagi lebih detail (dari sudut pandang ownership) pada chapter Borrowing.

A.34.8. Ownership pada data literal

Kita akan bahas topik ini pada chapter berikutnya, yaitu chapter Borrowing.

A.34.9. *Move semantics* pada macro `println`

Ada yang unik dengan macro `println`. Silakan coba kode berikut agar terlihat keunikannya.

```rust
fn main() {
    let str1 = String::from("luwe");
    println!("{str1}");
    do_something(str1);

    let str2 = String::from("ngelak");
    do_something(str2);
    println!("{str2}");
}

fn do_something(str: String) {
    println!("{}", str);
}
```

Jika dijalankan ada error di statement terakhir fungsi `main`.

```
main.rs 1, U ×
src > main.rs > do_something
▶ Run | Debug
fn main() {
    let str1: String = String::from("luwe");
    println!("{str1}");
    do_something(str: str1);

    let str2: String = String::from("ngelak");
    do_something(str: str2);
    println!("{str2}");
}

fn do_something(str: String) {
    println!("{}", str);
}
```

```
PROBLEMS 1   OUTPUT   GITLENS   DEBUG CONSOLE   TERMINAL

(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrograman-rust\dasarpemrograman-rust\examples\playground> cargo run
   Compiling playground v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpemrograman-rust\dasarpemrograman-rust\examples\playground)
error[E0382]: borrow of moved value: `str2`
  --> src\main.rs:8:16
   |
6  |     let str2 = String::from("ngelak");
   |         ---- move occurs because `str2` has type `String`, which does not implement the `Copy` trait
7  |     do_something(str2);
   |                  ----- value moved here
8  |     println!("{str2}");
   |                ^^^^^ value borrowed here after move
   |
   = note: this error originates in the macro `$crate::format_args_nl` which comes from the expansion of the macro `println`
   (in Nightly builds, run with -Z macro-backtrace for more info)

For more information about this error, try `rustc --explain E0382`.
error: could not compile `playground` due to previous error
```

Seperti yang sudah dibahas, bahwa tipe `string` mengadopsi move semantics. Ketika data bertipe ini digunakan pada operasi assignment seperti contohnya sebagai argument pemanggilan fungsi, maka owner berpindah.

Tapi entah kenapa, khusus dalam pemanggilan macro `println`, owner-nya tidak berpindah. Ajaib.

Silakan lihat sendiri di gambar di atas, ketika `str1` digunakan pada macro `println`, kemudian digunakan lagi pada argument pemanggilan fungsi `do_something`, hasilnya tidak error.

Akan tetapi ketika digunakan pada pemanggilan fungsi terlebih dahulu, jika digunakan lagi di statement di bawahnya hasilnya error.

Ini adalah keistimewaan dari macro `println` dan beberapa macro untuk keperluan printing lainnya.

Lebih jelasnya mengenai macro dibahas pada chapter Macro.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../ownership
```

◉ Referensi

- https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html
- https://doc.rust-lang.org/std/marker/trait.Copy.html
- https://doc.rust-lang.org/nomicon/ownership.html

A.35. Borrowing

Pada chapter ini kita akan belajar tentang apa itu borrowing dalam Rust programming.

Sebelum masuk ke pembelajaran, penulis anjurkan untuk paham terlebih dahulu tentang Basic Memory Management, Pointer & References, dan juga konsep Ownership. Ketiga topik tersebut dipelajari pada chapter sebelum ini.

A.35.1. Konsep borrowing

Pada chapter sebelumnya kita telah belajar bahwa tipe data yang mengadopsi *move semantics* (seperti `string`), ketika digunakan pada operasi assignment maka owner-nya berpindah. Variabel yang sebelumnya adalah owner, setelah proses assignment menjadi invalid, karena owner data tersebut telah berpindah ke variabel baru. Efeknya, semua operasi yang dilakukan pada variabel owner sebelumnya menghasilkan error.

Sekarang perhatikan kode berikut:

```
let msg_1 = String::from("hello");
let msg_2 = msg_1;

println!("{:?}", msg_2);
println!("{:?}", msg_1);
```

Pada contoh di atas, statement print `msg_1` menghasilkan error karena variabel tersebut telah invalid setelah statement `let msg_2 = msg_1`.

Programmer harus ekstra hati-hati dan bijak dalam pengelolaan owner data, agar memory ter-manage dengan baik. Tapi kalau dipikir-pikir, repot juga kalau owner harus dilempar-lempar setiap selesai digunakan agar tidak error.

Salah satu cara yang bisa digunakan agar tidak terlalu repot adalah dengan menerapkan cloning (seperti yang sudah dibahas pada chapter sebelumnya). Namun ini bukan opsi yang baik karena boros memory, sedangkan di sisi lain programmer dianjurkan untuk efisien dalam penggunaan memory.

Solusi yang paling pas adalah dengan menerapkan **borrowing**. Borrowing artinya adalah meminjam. Pada konteks Rust programming, borrowing berarti meminjam data milik owner, dipinjam agar bisa diakses tanpa perlu memindah owner-nya. Kemudian setelah peminjaman selesai, data dikembalikan.

Cara meminjam data di Rust sangat mudah, yaitu:

- Untuk borrowing dengan level akses immutable/read-only, gunakan operator reference `&`

  ```
  let msg_3 = String::from("hello rust");
  let msg_4 = &msg_3; // <----- borrow operation

  println!("{:?}", msg_4); // output => hello rust
  println!("{:?}", msg_3); // output => hello rust
  ```

- Untuk borrowing dengan level akses mutable, gunakan operator reference `&mut`

  ```
  let mut msg_3 = String::from("hello");
  let msg_4 = &mut msg_3; // <----- mutable borrow operation

  *msg_4 = String::from("hello rust");
  ```

Di Rust, semua statement reference (baik mutable ataupun immutable) adalah operasi *borrowing*. Yang terjadi pada statement reference adalah data milik owner dipinjam dalam bentuk pointer. Pointer itu sendiri merupakan alamat memory yang mengarah ke data sebenarnya (milik owner).

Dari contoh di atas, bisa ditarik kesimpulan bahwa data string `hello rust` memiliki dua reference:

- Yang pertama adalah owner data, yaitu variabel `msg_3`
- Yang kedua adalah peminjam data, yaitu variabel pointer `msg_4`

A.35.2. Rust Borrow Checker

Rust compiler memiliki 1 bagian bernama **borrow checker**, tugasnya untuk melakukan pengecekan pada source code apakah ada kode yang berhubungan dengan ownership dan borrowing, dan apakah kode tersebut mengikuti aturan borrowing yang sudah ditetapkan oleh Rust, atau tidak. Jika ada yang menyalahi aturan, maka borrow checker memunculkan error.

> *Borrow checker sangat galak, pastikan kode yang ditulis mengikuti aturan yang berlaku di pemrograman Rust.*

A.35.3. Aturan borrowing

Aturan borrowing atau reference sempat disinggung pada chapter Pointer & References, yang kurang lebih adalah:

- Dalam waktu yang sama, hanya boleh ada satu mutable reference atau banyak immutable reference (keduanya tidak bisa bersamaan, harus salah

satu).

- Reference harus selalu valid.

Dua aturan tersebut wajib dipatuhi, jika tidak maka pasti muncul error.

Ok, selanjutnya mari kita test aturan tersebut.

A.35.4. Borrowing mutable/ immutable reference

◉ Contoh ke-1

Pada contoh berikut kita simulasikan 1 data memilik banyak immutable reference. Jika mengacu ke aturan di atas, maka hal seperti ini adalah boleh dan tidak menghasilkan error.

```
let msg_5 = String::from("hello rust");

let msg_6 = &msg_5;
let msg_7 = &msg_5;
let msg_8 = &msg_5;

println!("{:?} {:?} {:?}", msg_6, msg_7, msg_8);
```

```
28        let msg_5: String = String::from("hello rust");
29
30        let msg_6: &String = &msg_5;
31        let msg_7: &String = &msg_5;
32        let msg_8: &String = &msg_5;
33
34        println!("{:?} {:?} {:?}", msg_6, msg_7, msg_8);
35
```

PROBLEMS OUTPUT GITLENS JUPYTER DEBUG CONSOLE TERMINAL

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrograman rust\dasarpe
   Compiling borrowing_1 v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarp
    Finished dev [unoptimized + debuginfo] target(s) in 0.70s
     Running `target\debug\borrowing_1.exe`
"hello rust" "hello rust" "hello rust"
```

◉ Contoh ke-2

Pada contoh ke-2 ini, kita coba simulasikan 1 data memilik 1 mutable reference. Harusnya tidak muncul error karena diperbolehkan di aturan yang tertulis.

```rust
let mut msg_9 = String::from("hello rust");

let msg_10 = &mut msg_9;

println!("{:?}", msg_10);
```

```
38      let mut msg_9: String = String::from("hello rust");
39
40      let msg_10: &mut String = &mut msg_9;
41
42      println!("{:?}", msg_10);
43
```

PROBLEMS OUTPUT GITLENS JUPYTER DEBUG CONSOLE TERMINAL

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarpemro
   Compiling borrowing_1 v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpem
    Finished dev [unoptimized + debuginfo] target(s) in 0.69s
     Running `target\debug\borrowing_1.exe`
"hello rust"
```

Bagaimana jika ada lebih dari 1 mutable reference? mari kita test.

```rust
let mut msg_9 = String::from("hello rust");

let msg_10 = &mut msg_9;
let msg_11 = &mut msg_9;

println!("{:?} {:?}", msg_10, msg_11);
```

```
38      let mut msg_9: String = String::from("hello rust");
39
40      let msg_10: &mut String = &mut msg_9;
41      let msg_11: &mut String = &mut msg_9;
42
43      println!("{:?} {:?}", msg_10, msg_11);
44
```

PROBLEMS 1 OUTPUT GITLENS JUPYTER DEBUG CONSOLE TERMINAL

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarpemrogramanrust\examples\ho
   Compiling borrowing_1 v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarpemr

warning: unused variable: `msg_11`
    > src\main.rs:41:9
    |
41  |     let msg_11 = &mut msg_9;
    |         ^^^^^^ help: if this is intentional, prefix it with an underscore: `_msg_11`
    |
    = note: `#[warn(unused_variables)]` on by default

error[E0499]: cannot borrow `msg_9` as mutable more than once at a time
  --> src\main.rs:41:18
    |
40  |     let msg_10 = &mut msg_9;
    |                  ---------- first mutable borrow occurs here
41  |     let msg_11 = &mut msg_9;
    |                  ^^^^^^^^^^ second mutable borrow occurs here
42  |
43  |     println!("{:?} {:?}", msg_10, msg_10);
    |                           ------ first borrow later used here

For more information about this error, try `rustc --explain E0499`.
warning: `borrowing_1` (bin "borrowing_1") generated 1 warning
error: could not compile `borrowing_1` due to previous error; 1 warning emitted
```

Hasilnya error. Hal seperti ini tidak diperbolehkan. Sebuah data tidak boleh memiliki lebih dari 1 mutable reference.

◉ Contoh ke-3

Sekarang mari kita coba test lagi aturan di atas dengan skenario: dalam waktu yang sama, 1 data memiliki 1 mutable reference dan 1 immutable reference. Hasilnya seperti apa, (jika mengacu ke aturan, hal seperti ini adalah tidak

diperbolehkan).

```rust
let mut msg_12 = String::from("hello rust");

let msg_13 = &msg_12;
let msg_14 = &mut msg_12;

println!("{:?} {:?}", msg_13, msg_14);
```

```
46
47      let mut msg_12: String = String::from("hello rust");
48
49      let msg_13: &String = &msg_12;
50      let msg_14: &mut String = &mut msg_12;
51
52      println!("{:?} {:?}", msg_13, msg_14);
53
PROBLEMS 1   OUTPUT   GITLENS   JUPYTER   DEBUG CONSOLE   TERMINAL

(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarpemrograman
  Compiling borrowing 1 v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpemrograman

error[E0502]: cannot borrow `msg_12` as mutable because it is also borrowed as immutable
   --> src\main.rs:50:18
    |
49  |     let msg_13 = &msg_12;
    |                  ------- immutable borrow occurs here
50  |     let msg_14 = &mut msg_12;
    |                  ^^^^^^^^^^^ mutable borrow occurs here
51  |
52  |     println!("{:?} {:?}", msg_13, msg_14);
    |                           ------- immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `borrowing 1` due to previous error
```

Hasilnya error, kenapa? karena memang tidak boleh.

Jadi sampai sini cukup jelas ya. Sebuah data dalam waktu yang sama hanya diperbolehkan memiliki satu atau lebih immutable reference, atau hanya 1

mutable reference, dan keduanya tidak bisa bersamaan dalam satu waktu (harus pilih salah satu).

A.35.5. Borrowing valid/invalid reference

Kita telah belajar tentang valid/invalid variable dan juga variable scope pada chapter sebelumnya. Sekarang kita akan coba gabungkan dua aspek tersebut dengan borrowing.

Silakan praktekan kode berikut:

```
fn main() {
    let mut fact_one = String::from("Arthas is the true lich
king");

    change_value(&mut fact_one);

    let fact_two = &mut fact_one;
    println!("{:?}", fact_two);
}

fn change_value(txt: &mut String) {
    *txt = String::from("Bolvar is better lich king");
}
```

Pada kode di atas, ada satu buah mutable string bernama `fact_one`. String tersebut dipinjam menggunakan `&mut` (yang berarti adalah mutable borrow) sebagai argumen pemanggilan fungsi `change_value` yang isinya kurang lebih adalah perubahan isi data string pada variabel pointer.

Kemudian, ada operasi peminjaman lagi (yang juga mutable borrow) dari

variabel `fact_one` ke `fact_one`.

Ketika program di run, hasilnya sukses. Kok bisa? padahal jelas di aturan tertulis bahwa dalam waktu yang sama tidak boleh ada lebih dari satu mutable borrow.

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasa
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\borrowing_2.exe`
"Bolvar is better lich king"
```

Kata **dalam waktu yang sama** disini adalah yang penting untuk dipahami. Arti *dalam waktu yang sama* adalah dalam 1 scope yang sama. Jika ada 2 scope, maka itu sudah bukan dalam waktu yang sama lagi.

Pada contoh di atas, statement `&mut fact_one` terjadi pada block fungsi `change_value`, lebih tepatnya pada parameter `txt` fungsi tersebut.

Kemudian setelah eksekusi fungsi tersebut selesai, yang terjadi adalah: reference yang tadinya dipinjam, sekarang dikembalikan. Tepat setelah eksekusi fungsi `change_value`, state data `fact_one` adalah tidak dipinjam siapapun. Tadinya memang ada yang meminjam (yaitu parameter `txt` di fungsi `change_value`), tapi setelah eksekusi fungsi `change_value` selesai, data dikembalikan lagi ke owner.

Proses dealokasi pada variable scope, jika terjadi pada variable reference maka yang sebenarnya terjadi adalah pengembalian data hasil operasi borrow ke pemilik aslinya.

Kemudian, ada operasi pinjam lagi, yaitu `let fact_two = &mut fact_one`. Statement borrow ini tidak menghasilkan error karena memang kondisi data `fact_one` sudah tidak ada yang meminjam. Bisa dibilang statement peminjaman ke-2 ini tidak terjadi dalam waktu yang sama dengan statement

peminjaman pertama (pemanggilan fungsi `change_value`).

Pada fungsi `change_value`, variabel `txt` adalah valid saat fungsi dijalankan. Setelah pemanggilan fungsi selesai, variabel `txt` di-dealokasi. Namun karena variabel tersebut bukanlah owner, melainkan hanya borrower yang meminjam data dari owner `fact_one`, maka yang terjadi adalah: data yang dipinjam sekarang dikembalikan lagi ke pemilik aslinya.

Ok sampai sini semoga cukup jelas ya. Jika perlu silakan ulang-ulang penjelasan di atas.

A.35.6. Borrowing pada block

Kita sudah cukup paham kapan waktu terjadinya alokasi alamat memory dan juga kapan proses dealokasi terjadi, yaitu ketika variabel **into scope** dan ketika variabel **out of scope**.

Scope disini adalah variable scope, yang maksudnya adalah block scope, bisa berupa block expression, fungsi, block seleksi kondisi `if`, dan juga jenis block lainnya.

Ok, sekarang mari kita praktekan beberapa jenis block untuk isolasi operasi borrowing. Dengan ini maka klausul **dalam satu waktu** akan terpenuhi.

Silakan modifikasi program sebelumnya menjadi seperti berikut:

```
fn main() {
    let mut fact_one = String::from("Arthas is the true lich
king");
    println!("{:?}", fact_one);

    change_value(&mut fact_one);
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarp
   Compiling borrowing_2 v0.1.0 (D:\Labs\Adam Studio\Ebook\das
    Finished dev [unoptimized + debuginfo] target(s) in 0.[illegible]s
     Running `target\debug\borrowing_2.exe`

"Arthas is the true lich king"
"Bolvar is better lich king"
"There must always be a Lich King"
"Who is the real jailer?"
"Is it Zovaal or Primus?"
```

Bisa dilihat, hasilnya tidak error, meskipun terjadi beberapa kali operasi mutable borrow, tapi karena kesemuanya terjadi di waktu yang berbeda (di scope yang berbeda) maka tidak error.

Pada kode di atas, mutable borrow terjadi di banyak tempat, yaitu di block fungsi `change_value`, block expression, block seleksi kondisi `if`, dan juga block perulangan `for`. Variabel pointer yang menampung data pinjaman akan valid di masing-masing block, kemudian menjadi invalid setelah eksekusi block selesai, dan yang terjadi setelah itu adalah pengembalian data yang telah dipinjam.

◉ Method `contains` milik `String`

Pada contoh di atas kita menerapkan method baru bernama `contains`. Method ini tersedia untuk tipe data `string`, gunanya adalah untuk mengecek apakah string memiliki substring `x`, dimana `x` adalah argumen pemanggilan method. Method ini mengembalikan nilai `bool`.

Contoh penerapan method `contains`:

```rust
let fact = String::from("There must always be a lich king");
println!("{:?}", fact.contains("lich king")); // output => true
println!("{:?}", fact.contains("bolvar"));    // output => false
```

O iya, pengecekan string-nya adalah case sensitive ya.

```
let fact = String::from("There must always be a lich king");
println!("{:?}", fact.contains("lich king")); // output => true
println!("{:?}", fact.contains("Lich King")); // output => false
```

> *Lebih jelasnya mengenai tipe data string dan method yang tersedia pada tipe tersebut akan dibahas terpisah pada chapter Tipe Data → String Custom Type vs &str.*

A.35.7. Owner dan borrower data literal

Perhatikan statement berikut:

```
let number = 12;
let a = &number;

let text = String::from("hello");
let b = &text;
```

Pada kode di atas, variabel `number` dan `text` adalah *owner* data masing-masing. Sedangkan variabel `a` dan `b` adalah *borrower* atau peminjam data (yang lebih jelasnya akan dibahas pada chapter). Sampai sini penulis rasa cukup jelas.

Selanjutnya, bagaimana dengan contoh ini:

```
let c = &24;
let d = &false;
let e = &String::from("rust");
```

Ketiga variabel, semuanya adalah *borrower* atau peminjam data. Lalu siapa *owner*-nya? jawabannya adalah tidak ada, atau silakan juga simpulkan bawah data tersebut owner-nya adalah program.

Yang lebih penting untuk diurusi pada contoh di atas bukan siapa owner-nya, melainkan bagaimana caranya agar data pinjaman tersebut tidak di-dealokasi saat block kode selesai.

Untuk sekarang sampai sini dulu. Kita akan bahas topik ini lebih mendetail pada chapter Static item dan Lifetime.

A.35.8. Borrowing pada macro `println`

Pada pemanggilan macro println untuk menampilkan data yang mengadopsi *move semantics*, operasi borrowing tidak perlu dilakukan sewaktu pengisian argument. Karena macro tersebut secara cerdas akan melakukan opearsi peminjaman tanpa memindah owner-nya. Contoh:

```
let str1 = String::from("luwe");
println!("{:?}", str1);
```

... adalah ekuivalen dengan ...

```
let str2 = String::from("ngelak");
```

Lebih jelasnya mengenai macro dibahas pada chapter Macro.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../borrowing
```

◉ Referensi

Beberapa referensi terkait chapter ini:

- https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html
- https://stackoverflow.com/questions/57225055/in-rust-can-you-own-a-string-literal

---

A.36. Traits

Trait jika diartikan dalam bahasa Indonesia artinya adalah sifat. Chapter ini akan membahas tentang apa itu trait, apa kegunaannya, dan bagaimana cara penerapannya di Rust programming.

Pembahasan chapter ini cukup panjang. Makin mendekati akhir pembahasan, makin berat yang dibahas. Penulis anjurkan jika nantinya setelah section A.36.4. Trait sebagai tipe parameter dirasa cukup susah untuk dipahami, silakan lanjut ke chapter berikutnya dulu, dan nanti bisa kembali ke pembahasan chapter ini lagi.

> *Chapter ini butuh tambahan detail*

A.36.1. Konsep traits

Di Rust kita bisa mendefinisikan trait/sifat, isinya adalah definisi header method yang bisa di-share ke banyak tipe data.

Trait isinya hanya definisi header method (bisa diartikan method tanpa isi). Ketika ada tipe data yang meng-implement suatu trait, maka tipe tersebut wajib untuk menuliskan implementasi method sesuai dengan header method yang ada di trait.

Ada dua bagian penting dalam trait yang harus diketahui:

1. Deklarasi trait
2. Implementasi trait ke tipe data

Perihal point pertama, intinya kita bisa menciptakan trait sesuai kebutuhan. Terlepas dari itu, Rust juga menyediakan cukup banyak traits yang di-implement ke banyak tipe data yang ada di Rust standard library. Beberapa diantaranya:

- Trait `std::fmt::Debug`, digunakan agar data bisa di-print menggunakan formatted print `{:?}`.
- Trait `std::iter::Enumerate`, digunakan agar data bisa di-iterasi menggunakan keyword `for`.
- Trait `std::ops::Add`, di-implementasikan agar data bisa digunakan pada operasi aritmatik penambahan `+`.

> *Pada bahasa pemrograman lain, contohnya Java, konsep trait mirip dengan `interface`*

Ok, biar lebih jelas, mari lanjut pembelajaran menggunakan contoh. Kita mulai dengan pembahasan tentang cara implementasi trait. Contoh yang digunakan adalah implementasi salah satu trait milik Rust standard library, yaitu trait `std::fmt::Debug`.

A.36.2. Implementasi trait

Kita pilih trait `std::fmt::Debug` milik Rust standard library untuk belajar cara implementasi trait pada tipe data.

Kegunaan dari trait ini adalah: jika di-implement ke tipe data tertentu maka data dengan tipe tersebut bisa di-print via macro `println` atau macro printing lainnya, dengan menggunakan formatted print `{:?}`.

Trait `Debug` ini diimplementasikan ke pada banyak tipe data yang di Rust

standard library, baik itu tipe primitif maupun non-primitif. Contohnya bisa dilihat pada kode berikut:

```
let number = 12;
println!("{:?}", number);

let text = String::from("hello");
println!("{:?}", text);
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasar
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\traits_1.exe`
12
"hello"
```

Dua variabel di atas sukses di-print tanpa error, karena tipe data `i32` dan `String` *by default* sudah implement trait `std::fmt::Debug`.

> *Jika tertarik untuk pengecekan lebih lanjut, silakan lihat di halaman dokumentasi tipe data i32 dan String.*

Bagaimana dengan custom type yang kita buat sendiri? Misalnya struct.

```
fn main() {
    let circle_one = Circle{raidus: 6};
    println!("{:?}", circle_one);
}

struct Circle {
    raidus: i32,
}
```

```
src > main.rs > main
▶ Run | Debug
fn main() {
    let number: i32 = 12;
    println!("{:?}", number);

    let text: String = String::from("hello");
    println!("{:?}", text);

    let circle_one: Circle = Circle{radius: 6};
    println!("{:?}", circle_one);
}

0 implementations
struct Circle {
    radius: i32,
}
```

```
let circle_one: Circle

Go to Circle

`Circle` doesn't implement `Debug`
the trait `Debug` is not implemented for `Circle`
add `#[derive(Debug)]` to `Circle` or manually `impl Debug for
Circle` rustc(click for full compiler diagnostic)

macros.rs(106, 20): Error originated from macro call here

main.rs(9, 5): Error originated from macro call here

main.rs(12, 1): consider annotating `Circle` with `#[derive(Debug)]`: `#
[derive(Debug)]
```

Hasilnya error, karena struct `Circle` yang dibuat tidak implement trait `std::fmt::Debug`.

Solusi agar tidak error adalah dengan mengimplementasikan trait `std::fmt::Debug` ke tipe `Circle`, dengan itu semua data bertipe `Circle` akan bisa di-print menggunakan formatted print `{:?}`.

> *Selain via implementasi trait, tipe data custom bisa di-print dengan cara menambahkan atribut `#[derive(Debug)]` pada definisi tipe data-nya. Namun kita tidak membahas itu pada chapter ini.*

Langkah pertama untuk implementasi trait adalah mencari tau terlebih dahulu spesifikasi trait yang ingin diimplementasikan. Trait `std::fmt::Debug` adalah traits milik Rust standard library, maka harusnya spesifikasi bisa dilihat di dokumentasi Rust.

> *https://doc.rust-lang.org/std/fmt/trait.Debug.html*

Pada URL dokumentasi bisa dilihat kalau trait `Debug` memiliki struktur kurang lebih seperti berikut:

```
pub trait Debug {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}
```

Trait `Debug` mempunyai satu spesifikasi method, bernama `fmt` yang detail strukturnya bisa dilihat di atas.

Kita akan implement trait `Debug` ini ke tipe `circle`, maka wajib hukumnya untuk menuliskan implementasi method sesuai dengan yang ada di trait `Debug`.

Di bawah ini adalah contoh cara implementasi trait.

```
struct Circle {
    raidus: i32,
}

impl std::fmt::Debug for Circle {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) ->
std::fmt::Result {
        write!(f, "Circle radius: {}", self.raidus)
    }
}

fn main() {
    let circle_one = Circle{raidus: 6};
    println!("{:?}", circle_one);
```

Ketika program di run, hasilnya sukses tanpa error. Artinya implementasi trait `Debug` pada tipe data struct `circle` adalah sukses.

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasa
   Compiling traits_1 v0.1.0 (D:\Labs\Adam Studio\Ebook\dasar
    Finished dev [unoptimized + debuginfo] target(s) in 0.44s
     Running `target\debug\traits_1.exe`
Circle radius: 6
```

Cara implementasi trait ke struct `circle` memang step-nya agak panjang, tapi penulis yakin lama-kelamaan pasti terbiasa. Ok, sekarang kita bahas satu per satu kode di atas.

◉ Struct `Circle`

Block kode definisi struct `Circle` cukup straightforward, isinya hanya 1 property bernama `radius` bertipe `i32`.

◉ Block kode `impl X for Y`

Notasi penulisan implementasi trait adalah `impl X for Y`, dimana `X` adalah trait yang ingin diimplementasikan dan `Y` adalah tipe data tujuan implementasi.

Pada contoh di atas, trait `Debug` diimplementasikan ke custom type struct `circle`. Maka statement-nya adalah:

```rust
impl std::fmt::Debug for Circle {
    // ...
}
```

◉ Block kode method dalam `impl`

Block kode `impl` harus diikuti dengan implementasi method. Pada contoh ini, method `fmt` milik trait `Debug` wajib untuk diimplementasikan. Spesifikasi method ini adalah `fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>` (lebih jelasnya silakan lihat dokumentasi).

Silakan copy method tersebut kemudian paste ke dalam block kode `impl` yang sudah ditulis, kemudian tambahkan block kurung kurawal.

```
impl std::fmt::Debug for Circle {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) ->
std::fmt::Result {
        // ...
    }
}
```

Kemudian tulis implementasi method `fmt` dalam block method. Tulis statement macro `write` untuk data string (yang ingin di-print) dengan tujuan adalah variabel `f`.

Di contoh, format `circle radius: {}` digunakan. Dengan ini nantinya saat printing data, yang muncul adalah text `Circle radius: {}`.

```
impl std::fmt::Debug for Circle {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) ->
std::fmt::Result {
        write!(f, "Circle radius: {}", self.raidus)
    }
}
```

> *Tips untuk pengguna visual studio code dengan rust-analyzer extension ter-install, setelah selesai menulis block kode `impl`, cukup jalankan `ctrl+space` atau `cmd+space` untuk men-trigger autocomplete suggestion. Kemudian klik opsi method yang ada disitu, maka kode implementasi method langsung muncul dengan sendirinya.*

◉ Macro `write`

Macro ini digunakan untuk menuliskan sebuah data ke object tertentu. Pada contoh kita gunakan untuk menulis string `Circle radius: {}` ke variabel `f` yang bertipe `std::fmt::Formatter<'_>`.

Notasi penulisan macro `write`:

```
// notasi penulisan
write!(variabel_tujuan, data_yang_ingin_di_print, arg1, arg2,
...);

// contoh penerapan
write!(f, "Circle radius: {}", self.raidus);
```

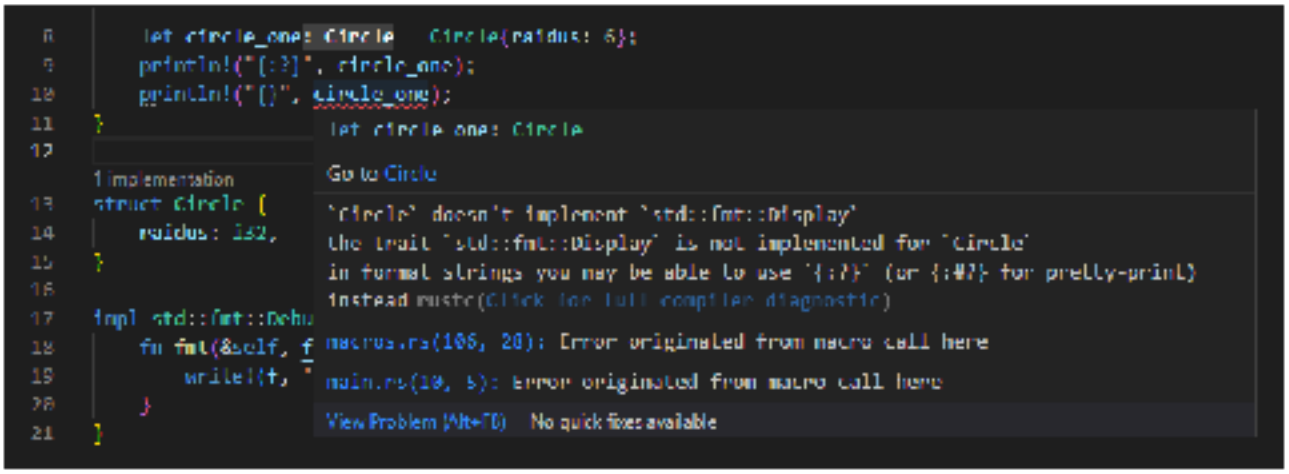
◉ Print data menggunakan formatted print `{:?}`

Step terakhir adalah print variabel `circle` menggunakan macro `println`. Hasilnya sukses, tidak error seperti sebelumnya.

◉ Print data menggunakan formatted print `{}`

Coba tambahkan statement `println`, tetapi kali ini gunakan formatted print

`{}`, apakah hasilnya juga tidak error?



Hasilnya error, karena trait `std::fmt::Debug` hanya berguna untuk formatted print `{:?}`. Agar data bertipe `Circle` bisa di-print menggunakan formatted print `{}` maka trait `std::fmt::Display` harus di-implementasikan juga.

Ubah kode dengan menambahkan implementasi trait `Display`. Hasilnya kurang lebih seperti ini:

```
struct Circle {
    raidus: i32,
}

impl std::fmt::Debug for Circle {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) ->
std::fmt::Result {
        write!(f, "Circle radius: {}", self.raidus)
    }
}

impl std::fmt::Display for Circle {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) ->
std::fmt::Result {
```

> - *Link dokumentasi trait `Debug` https://doc.rust-lang.org/std/fmt/trait.Debug.html*
> - *Link dokumentasi trait `Display` https://doc.rust-lang.org/std/fmt/trait.Display.html*

A.36.3. Membuat custom trait

Pada section di atas kita telah membahas bagaimana cara implementasi traits ke tipe data. Pada bagian ini kita akan belajar tentang cara membuat definisi trait (membuat custom trait).

Masih sama dengan metode sebelumnya, pembelajaran dilakukan dengan praktek. Kita gunakan skenario praktek berikut pada program selanjutnya:

1. Buat struct bernama `circle`.
2. Buat struct bernama `Square`.
3. Buat trait bernama `Area` dengan isi satu buah method untuk menghitung luas bangun datar (`calculate`).
4. Implementasikan trait `Area` ke dua struct tersebut.

Ok, mari mulai praktekan skenario di atas. Pertama siapkan project dengan struktur berikut:

package source code structure

```
my_package
|── Cargo.toml
└── src
     |── calculation_spec.rs
```

Module `calculation_spec` berisi definisi trait `Area`. Trait ini punya visibility akses publik, isinya hanya satu buah definisi method header bernama `calculate`. Trait ini nantinya diimplementasikan ke struct `Circle` dan juga `Square`, agar nantinya kedua struct tersebut memiliki method `calculate` yang berguna untuk kalkulasi luas bangun datar 2d.

src/calculation_spec.rs

```
pub trait Area {
    fn calculate(&self) -> f64;
}
```

Kemudian siapkan file `two_dimensional`, isinya dua buah struct: `Circle` dan `Square`. Pada file yang sama, siapkan juga block kode implementasi trait `Area`. Dengan ini maka kedua struct tersebut wajib untuk memiliki method bernama `calculate` dengan isi adalah operasi perhitungan aritmatika untuk mencari luas bangun datar.

src/two_dimensional.rs

```
pub struct Circle {
    pub radius: i32,
}

impl crate::calculation_spec::Area for Circle {
    fn calculate(&self) -> f64 {
        // PI * (r ^ 2)
        // ada operasi casting ke tipe f64 karena self.radius
bertipe i32
        3.14 * (self.radius.pow(2) as f64)
    }
}
```

Bisa dilihat pada kode di atas, deklarasi struct beserta property memiliki visibility publik. Idealnya, saat sturct tersebut digunakan di fungsi `main` nantinya tidak akan ada error terkait visibility akses.

Selanjutnya, pada file `main.rs` siapkan kode yang isinya registrasi module `calculate_spec` dan `two_dimensional`, juga definisi fungsi `main` dengan isi statement pembuatan 2 variabel object untuk masing-masing tipe data struct `circle` dan `square`.

src/main.rs

```
mod calculation_spec;
mod two_dimensional;

use crate::calculation_spec::Area;

fn main() {
    let circle_one = two_dimensional::Circle{ radius: 10 };
    println!("circle area: {}", circle_one.calculate());

    let square_one = two_dimensional::Square{ length: 5 };
    println!("square area: {}", square_one.calculate());
}
```

Method `calculate` milik object bertipe `circle` dan `square` diakses untuk kemudian di-print.

Coba jalankan program.

```
main.rs 2, U ×
src > main.rs > main
1   mod calculation_spec;
2   mod two_dimensional;
3
    ▶ Run | Debug
4   fn main() {
5       let circle_one: Circle = two_dimensional::Circle{ radius: 10 };
6       println!("circle area: {}", circle_one.calculate());
7
8       let square_one: Square = two_dimensional::Square{ length: 5 };
9       println!("square area: {}", square_one.calculate());
10  }
11

PROBLEMS 2   OUTPUT   COMMENTS   JUPYTER   DEBUG CONSOLE   TERMINAL

(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarpemrogramanrust\examples\tr
  Compiling traits_2 v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarpemrogr

error[E0599]: no method named `calculate` found for struct `Circle` in the current scope
 --> src\main.rs:6:44
  |
  |
  ::: src\two_dimensional.rs:11:1
  |
11 | pub struct Square {
   | ---------------- method `calculate` not found for this
  |
  ::: src\calculation_spec.rs:2:8
  |
2 |     fn calculate(&self) -> f64;
  |                 the method is available for `Square` here
  |
  = help: items from traits can only be used if the trait is in scope
help: the following trait is implemented but not in scope; perhaps add a `use` for it:
  |
1 | use crate::calculation_spec::Area;
  |

For more information about this error, try `rustc --explain E0599`.
error: could not compile `traits_2` due to 2 previous errors
```

Hmm, error. Padahal trait `Area` sudah publik, dan struct `Circle` & `Square` beserta property-nya juga sudah publik. Tapi masih error.

Error ini disebabkan oleh trait `Area` yang belum di-import di crate root (main). Meskipun kita tidak mengakses trait tersebut secara langsung (melainkan via method `calculate` milik struct `Circle` dan `Square`), diharuskan untuk meng-import-nya juga.

> *Detail error beserta solusi dari error ini sebenarnya bisa dilihat di error message. Bagaimana Rust menginformasikan error sangat luar biasa informatif.*

Ok, sekarang ubah isi file `main.rs` menjadi seperti ini, kemudian jalankan ulang program. Hasilnya tidak ada error.

src/main.rs

```
mod calculation_spec;
mod two_dimensional;

use crate::calculation_spec::Area; // <------- tambahkan
statement import module

fn main() {
    let circle_one = two_dimensional::Circle{ radius: 10 };
    println!("circle area: {}", circle_one.calculate());

    let square_one = two_dimensional::Square{ length: 5 };
    println!("square area: {}", square_one.calculate());
}
```

O iya, ada beberapa hal baru pada penerapan kode di atas, berikut adalah pembahasannya:

◉ Method `pow` untuk operasi pangkat

Method `pow` adalah item milik tipe data numerik (`i8`, `i16`, `i32`, ...) yang fungsinya untuk operasi pangkat.

```
3.pow(2); // ===> 3 pangkat 2
8.pow(5); // ===> 8 pangkat 5
```

◉ Keyword `as` untuk casting tipe data

Keyword `as` digunakan untuk casting tipe data. Keyword ini bisa diterapkan pada beberapa jenis tipe data, salah satunya adalah semua tipe data numerik.

```
1024 as f32; // ===> 1024 dikonversi ke tipe f32, hasinya adalah
1024.0
3.14 as i32; // ===> 3.14 dikonversi ke tipe i32, hasinya 3
karena ada pembulatan
```

A.36.4. Trait sebagai tipe parameter

Trait bisa digunakan sebagai tipe data parameter sebuah fungsi, contoh notasi penulisannya bisa dilihat pada kode berikut:

```
fn calculate_and_print_result(name: String, item: &impl Area) {
    println!("{} area: {}", name, item.calculate());
}
```

Manfaat penerapan trait sebagai tipe data parameter fungsi adalah saat pemanggilan fungsi, parameter tersebut bisa diisi dengan argument data bertipe apapun dengan catatan tipe dari data tersebut mengimplementasikan trait yang sama dengan yang digunakan pada parameter.

Misalnya, pada fungsi `calculate_and_print_result` di atas yang parameter ke-2 bertipe `&impl Area`, nantinya saat fungsi tersebut dipanggil, kita bisa sisipi parameter ke-2 dengan object `circle_one` ataupun `circle_two`.

```
let circle_one = two_dimensional::Circle{ radius: 10 };
calculate_and_print_result("circle".to_string(), &circle_one);

let square_one = two_dimensional::Square{ length: 5 };
calculate_and_print_result("square".to_string(), &square_one);
```

> *`&impl Area` ini tipe data pointer ya, tipe non-pointer-nya adalah `impl Area`. Disini digunakan tipe data pointer untuk antisipasi move semantics pada tipe data custom type (borrowing).*

Dimisalkan, fungsi tersebut parameter `item`-nya bisa menampung beberapa jenis traits, kira-kira apakah bisa dibuat seperti itu? Misalnya ada trait lain bernama `Circumference`, dan parameter `item` milik fungsi `calculate_and_print_result` harus bisa menampung data baik dari tipe yang implement trait `Area` ataupun trait `Circumference`.

Hal seperti itu bisa, caranya dengan menggunakan notasi penulisan berikut:

```
fn calculate_and_print_result(name: String, item: &(impl Area +
Circumference)) {
    println!("{} area: {}", name, item.calculate());
    println!("{} circumference: {}", name,
```

Tambahkan tanda `()` sebelum `impl NamaTrait`, lalu ganti `NamaTrait` dengan traits apa saja yang diinginkan dengan separator tanda `+`.

A.36.5. Trait bound syntax

Penerapan trait sebagai parameter fungsi juga bisa dituliskan dalam notasi yang memanfaatkan generic. Teknik penulisan ini disebut dengan *trait bound syntax*.

Contohnya bisa dilihat pada kode berikut. Ada generic bernama `T` yang merepresentasikan trait `Area`, kemudian pada definisi parameter ke-2 fungsi (yaitu parameter `item`) tipenya menggunakan `&T`. Tipe `&T` disini adalah ekuivalen dengan `&impl Area`.

```
fn calculate_and_print_result2<T: Area>(name: String, item: &T) {
    println!("{} area: {}", name, item.calculate());
}
```

Dimisalkan jika ada lebih dari satu trait yang digunakan sebagai tipe data paramater (misalnya trait `Area` dan `Circumference`), maka penulisannya seperti ini:

```
fn calculate_and_print_result2<T: Area + Circumference>(name:
String, item: &T) {
    println!("{} area: {}", name, item.calculate());
    println!("{} circumference: {}", name,
item.calculateCircumference());
}
```

Satu tambahan contoh lagi untuk ilustrasi yang lebih kompleks:

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u:
&U) -> i32 {
    // ...
}
```

Pada contoh di atas fungsi `some_function` memiliki 2 generics param, yaitu `T` dan `U`.

- `T` merepresentasikan trait `Display` dan `Clone`
- `U` merepresentasikan trait `Clone` dan `Debug`

> *Lebih jelasnya mengenai generics dibahas pada chapter Generics*

A.36.6. Trait `where` clause

Ada lagi alternatif penulisan trait bound syntax, yaitu menggunakan keyword `where`. Contoh pengaplikasiannya bisa dilihat pada kode berikut. Semua definisi fungsi di bawah ini adalah ekuivalen.

```
fn calculate_and_print_result(name: String, item: &(impl Area +
Circumference)) {
    println!("{} area: {}", name, item.calculate());
    println!("{} circumference: {}", name,
item.calculateCircumference());
}

fn calculate_and_print_result2<T: Area + Circumference>(name:
String, item: &T) {
    println!("{} area: {}", name, item.calculate());
    println!("{} circumference: {}", name,
```

> *Lebih jelasnya mengenai generics dibahas pada chapter Generics*

A.36.7. Trait sebagai return type

Trait bisa juga digunakan sebagai tipe data return value. Caranya gunakan notasi penulisan `impl NamaTrait` sebagai tipe data.

Contohnya bisa dilihat pada kode berikut. Ada dua fungsi baru dideklarasikan:

1. Fungsi `new_circle` dengan return type adalah `impl Area`, dan data yang dikembalikan adalah bertipe `two_dimensional::Circle`.
2. Fungsi `new_square` dengan return type adalah `impl Area`, dan data yang dikembalikan adalah bertipe `two_dimensional::Square`.

```
fn main() {
    let circle_one = new_circle(5);
    calculate_and_print_result4("circle".to_string(),
&circle_one);

    let square_one = new_square(10);
    calculate_and_print_result4("square".to_string(),
&square_one);
}

fn new_circle(radius: i32) -> impl Area {
    let data = two_dimensional::Circle{
        radius
    };
    data
}

fn new_square(length: i32) -> impl Area {
```

Salah satu konsekuensi dalam penerapan trait sebagai return type adalah: tipe data milik nilai yang dikembalikan terdeteksi sebagai tipe trait. Contohnya variabel `circle_one` di atas, tipe data-nya bukan `Circle`, melainkan `impl Area`.

Tipe data aslinya tetap bisa diakses, tapi butuh tambahan effort. Lebih jelasnya dibahas pada chapter Trait → Conversion (From & Into).

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../traits
```

◉ Work in progress

- Pembahasan tentang trait associated types
- Pembahasan tentang trait bounds untuk implementasi method kondisional
- Pembahasan tentang trait overloading

◉ Referensi

- https://doc.rust-lang.org/book/ch10-02-traits.html
- https://doc.rust-lang.org/std/primitive.i32.html
- https://doc.rust-lang.org/std/fmt/trait.Debug.html
- https://doc.rust-lang.org/std/fmt/trait.Display.html
- https://doc.rust-lang.org/std/string/struct.String.html

A.37. Generics

Chapter ini membahas tentang generics.

Generics sendiri merupakan salah satu fitur yang ada pada beberapa bahasa pemrograman (termasuk Rust), yang digunakan untuk menambahkan fleksibilitas dalam pemanfaatan tipe data pada suatu block kode. Dengan adanya generics, kita bisa menentukan tipe data yang digunakan pada parameter maupun return value sbuah block fungsi, method dan lainnya.

Generics dinotasikan dengan `<T>`. Kita sempat sedikit memanfaatkan generic pada chapter Vector dimana dalam pendefinisian tipe data harus dituliskan juga tipe data item (via generics parameter), contoh `Vec<i32>`, `Vec<&str>`, dll. Kita juga sempat sedikit belajar tentang topik generic pada chapter Traits.

A.37.1. Generics basic

Mari mulai pembahasan dengan sebauh contoh definisi fungsi yang memiliki generics.

```
fn do_something<T>(arg1: i32, arg2: T) {
    // ...
}
```

Pada contoh di atas, fungsi `do_something` didefinisikan dengan 2 buah parameter argument dan 1 buah parameter generics. Parameter argument pertama, yaitu `arg1` bertipe `i32`, kemudian diikuti parameter ke-2 bertipe `T` dimana `T` adalah parameter generic fungsi.

Dalam pemanggilan fungsi tersebut, `T` generics dan tipe data argument `arg2` harus sama. Contoh:

```
fn main() {
    do_something::<bool>(24, false);
}
```

Fungsi `do_something` dipanggil dengan disisipi argument `24` dan `false`. Parameter ke-2 fungsi tersebut tipe data-nya jelas `bool` karena pada argument-nya nilai `false` digunakan, oleh karena itu tipe data `bool` disisipkan pada parameter generics saat pemanggilan fungsi; dengan notasi penulisan adalah `::<tipe_data>`, posisi penulisannya diantara nama fungsi dan tanda kurung pemanggilan fungsi `(`.

Apa yang terjadi jika nilai `T` diisi dengan tipe data argument `arg2` berbeda? program error.

```
                &str
                mismatched types
                expected `bool`, found `&str` rustc(Click for full compiler diagnostic)
▶ Run | Debug   View Problem (Alt+F8)   No quick fixes available
fn main() {
    do_something::<bool>(arg1: 24, arg2: "Hello");
}
```

Ok sampai sini semoga jelas.

Khusus untuk fungsi yang generic parameter-nya dipergunakan sebagai tipe data parameter, maka boleh untuk tidak ditentukan tipe data-nya saat pemanggilan fungsi. Contohnya bisa dilihat pada kode berikut, dua statements ini adalah ekuivalen.

```
do_something::<bool>(24, false);
// T adalah bool, karena ditulis secara eksplisit

do_something(24, false);
// T adalah bool, karena diketahui dari tipe data argument.
// nilai `false` pasti tipe data-nya bool
```

Tipe data generic boleh tidak ditulis karena nilai `T` bisa diketahui dari argument pemanggilan fungsi. Sedangkan jika nilai `T` tidak digunakan sebagai tipe data parameter, maka wajib untuk di-isi nilai `T` saat pemanggilan fungsi.

O iya, jumlah parameter generics tidak ada batasan. Bisa saja ada lebih dari satu parameter generic yang didefinisikan, contohnya bisa dilihat pada fungsi `do_something_v2` ini yang memiliki 2 buah parameter generic, yaitu `R` dan `T`.

```
fn do_something_v2<R, T>(arg1: R, arg2: T) {
    // ...
}
```

Tipe `R` digunakan sebagai tipe parameter `arg1` dan tipe `T` pada parameter `arg2`.

A.37.2. Mengasosiasikan traits ke parameter generic

Fungsi `do_something` yang telah dipraktekan, susah untuk diisi dengan apapun. Memang parameter `arg1` tipe-nya adalah `i32`, dan harusnya mudah untuk bermain-main dengan parameter tersebut. Yang agak repot adalah parameter ke-2, yaitu `arg2` yang bertipe `T`.

```
fn do_something<T>(arg1: i32, arg2: T) {
    // ...
}
```

Parameter `arg2` hampir tidak bisa diapa-apakan.

- Misal mau di-print, tidak bisa, karena tipe `T` tidak implement trait `std::fmt::Debug`.
- Misal diisi nilai numerik kemudian dijadikan operand operasi aritmatika, juga tidak bisa karena tipe `T` tidak implement trait `std::ops::Add`, dan trait operasi bilangan lainnya.
- Misal diisi dengan nilai `bool`, tidak bisa digunakan pada seleksi kondisi `if` karena tipe `T` tidak implement trait `std::cmp::PartialOrd`, dan trait operasi logika lainnya.

Repot kan? Tapi tenang, tidak usah khawatir, ada soluasi agar tipe `T` bisa dimanfaatkan, yaitu dengan mengasosiasikan trait ke tipe data generic (sesuai kebutuhan).

◉ Contoh ke-1

Contoh pengaplikasiannya bisa dilihat pada kode berikut. Ada sebuah fungsi bernama `print_x_times` yang tugasnya adalah menampilkan data `T` sejumlah `x` kali, dimana `T` adalah parameter generics.

```
fn main() {
    print_x_times("Hello guys", 10);
}

fn print_x_times<T: std::fmt::Debug>(data: T, x: i32) {
    for _ in 0..x {
```

Parameter generic `T` diasosiasikan dengan trait `Debug`, dengan ini maka kita akan bisa print parameter `data` yang tipe data-nya adalah `T`.

Untuk mengetes hasilnya, jalankan program. Bisa dilihat text `Hello guys` muncul 10x.

```
warning: `generics` (bin "generics")
    Finished dev [unoptimized + debuginfo]
     Running `target\debug\generics.exe`

"Hello guys"
"Hello guys"
"Hello guys"
"Hello guys"
"Hello guys"
"Hello guys"
"Hello guys"
"Hello guys"
"Hello guys"
"Hello guys"
```

Cara untuk mengasosiasikan trait ke parameter generic adalah dengan menuliskannya dalam notasi berikut:

```
fn nama_fungsi<T: TraitYangInginDiasosiasikan>(arg1 ...) {
    // ...
}
```

◉ Contoh ke-2

Pada contoh ke-2 ini, dideklarasikan sebuah fungsi bernama `find_largest_number` yang tugasnya adalah mencari nilai maksimum dari sebuah tipe slice.

```
fn find_largest_number<T: std::cmp::PartialOrd>(list: &[T]) -> &T
```

Seperti yang sudah dibahas pada bagian sebelumnya, bahwa tipe `T` tidak akan bisa diapa-apakan kalau tidak diasosiasikan dengan trait. Maka pada contoh ini, tipe `T` diasosiasikan dengan trait `std::cmp::PartialOrd`. Benefitnya, semua data dengan tipe `T` bisa dipergunakan dalam operasi perbandingan.

> *Tanpa adanya trait `std::cmp::PartialOrd`, maka statement `if item > largest { ... }` menghasilkan error.*

Ok, sekarang panggil fungsi tersebut 2x, yang pertama diisi dengan data slice dari sebuah array, dan yang kedua data slice dari sebuah vector.

```
fn main() {
    let data_arr = [0, 1, 2, 3];
    let largest_number1 = find_largest_number(&data_arr);
    println!("largest_number1: {:?}", largest_number1);

    let data_vec = vec![4, 5, 6, 7];
    let largest_number2 = find_largest_number(&data_vec);
    println!("largest_number2: {:?}", largest_number2);
}
```

Hasilnya sesuai harapan.

```
warning: `generics` (bin "generics") generated 7 warnings
    Finished dev [unoptimized + debuginfo] target(s) in 0.89s
     Running `target\debug\generics.exe`

largest_number1: 3
largest_number2: 7
```

◉ Contoh ke-3

Trait yang bisa diasosiasikan dengan parameter generics adalah semua jenis

traits (tanpa terkecuali), termasuk custom trait yang kita buat sendiri.

Contohnya bisa dilihat pada kode berikut. Ada custom trait bernama `MyTrait` dideklarasikan, kemudian diasosiasikan dengan parameter generic `T` milik fungsi `do_something_v3`.

```
trait MyTrait {
    // methods declaration
}

fn do_something_v3<T: MyTrait>(arg1: T) {
    // do something
}
```

A.37.3. Multi traits pada parameter generic

Bagaimana jika `T` perlu untuk diasosiasikan dengan banyak traits (lebih dari satu), apakah bisa? Bisa. Cara penulisannya kurang lebih seperti berikut:

```
// fn nama_fungsi<T: Trait1 + Trait2 + ...>(arg1 ...)

fn print_largest_number<T: std::cmp::PartialOrd +
std::fmt::Debug>(list: &[T]) {
    let largest = find_largest_number::<T>(list);
    println!("largest number: {:?}", largest);
}
```

Fungsi di atas adalah fungsi baru, namanya `print_largest_number`, tugasnya adalah mencari nilai maksimum kemudian menampilkannya.

Proses pencarian nilai maksimum dilakukan dengan memanfaatkan fungsi `find_largest_number` yang sebelumnya sudah dibuat. Fungsi tersebut memerlukan trait `std::cmp::PartialOrd` untuk diasosiasikan dengan tipe data `T`.

Setelah nilai maksimum diketemukan, nilainya di-print ke stdout menggunakan macro `println`. Nilai maksimum ditampung pada variabel `largest`, tipe data-nya adalah `T`. Agar bisa di-print maka perlu diasosiasikan trait `std::fmt::Debug`.

Bisa dilihat cara penulisan asosiasi multi trait ke parameter generic seperti apa. Cukup tulis saja dengan separator tanda `+`.

A.37.4. Keyword `where`

Selain notasi penulisan yang sudah dipelajari di atas ada lagi alternatif lainnya, yaitu menggunakan keyword `where`. Kurang lebih seperti berikut penerapannya.

```
fn print_largest_number<T: std::cmp::PartialOrd +
std::fmt::Debug>(list: &[T]) {
    let largest = find_largest_number::<T>(list);
    println!("largest number: {:?}", largest);
}

// ... adalah ekuivalen dengan ...

fn print_largest_number<T>(list: &[T])
where
    T: std::cmp::PartialOrd + std::fmt::Debug,
{
    let largest = find_largest_number::<T>(list);
```

Silakan gunakan sesuai preferensi dan kesepakatan tim. Kalau penulis lebih suka cara ke-2, karena terasa lebih rapi untuk penulisan fungsi yang ada banyak parameter generic beserta asosiasi traits-nya. Contohnya:

```
fn do_something<T, U, V>(arg1: T, arg2: U, arg3: V)
where
    T: some::traits:TraitA,
    U: some::traits:TraitB + some::traits:TraitC +
some::traits:TraitD,
    V: some::traits:TraitA + some::traits:TraitD,
{
    // do something
}
```

A.37.5. Generics struct

Selain diterapkan di fungsi, generics bisa juga diterapkan di struct. Cara penulisannya, tambahkan notasi parameter generic diantara nama struct dan block struct.

```
struct Point<T, U> {
    x: T,
    y: T,
    z: U
}

fn main() {
    let num_one: Point<i32, f64> = Point { x: 502, y: 120, z: 4.5
};
    let num_two: Point<f64, i32> = Point { x: 1.2, y: 4.3, z: 534
};
}
```

Pada contoh di atas, struct `Point` memiliki 2 parameter generic. Kemudian struct tersebut digunakan untuk membuat dua variabel berbeda:

- Variabel `num_one`, bertipe `Point<i32, f64>`. Tipe data property `x` dan `y` adalah `i32` sedangkan tipe data property `z` adalah `f64`.
- Variabel `num_two`, bertipe `Point<f64, i32>`. Tipe data property `x` dan `y` adalah `f64` sedangkan tipe data property `z` adalah `i32`.

A.37.6. Generics method

Generic bisa diterapkan pada method. Notasi penulisannya kurang lebih sama seperti pada penulisan method, hanya saja pada syntax `impl` perlu diikuti block parameter generics. Perbandingannya kurang lebih seperti berikut:

```
// method biasa
impl Square {
    fn x(&self) -> &i32 {
        &self.x
    }
}

// method dengan generic
impl<T> Square<T> {
    fn x(&self) -> &T {
        &self.x
    }
}
```

> *Di syntax bagian `impl<T>` dan `Square<T>`, penulisan parameter generics di keduanya harus sama persis. Misal pada struct ada 3 buah parameter*

> *generic* `square<T, U, V>` *maka pada syntax* `impl` *juga harus sama, yaitu* `impl<T, U, V>`.
>
> *Jika tidak sama akan muncul error.*

Ok, sekarang mari kita praktekan. Struct `Point` yang sudah dibuat, kita siapkan method-nya. Ada 3 buah method yang akan dibuat dan kesemuanya adalah method *getter* untuk masing-masing property struct (yaitu `x`, `y`, dan `z`).

```
struct Point<T, U> {
    x: T,
    y: T,
    z: U
}

impl<T, U> Point<T, U> {

    fn get_x(&self) -> &T {
        &self.x
    }

    fn get_y(&self) -> &T {
        &self.y
    }

    fn get_z(&self) -> &U {
        &self.z
    }
}
```

Bisa dilihat pada kode di atas, ada method `get_x` untuk mengambil nilai `x`. Nilai baliknya bertipe `T` dimana tipe tersebut juga dipakai sebagai tipe data `x`.

Kemudian coba gunakan struct `Point` untuk membuat satu atau dua variabel, lalu akses method-nya.

```
fn main() {
    let num_one: Point<i32, f64> = Point { x: 502, y: 120, z: 4.5
};
    println!("{} {} {}", num_one.get_x(), num_one.get_y(),
num_one.get_z());
    // 502 120 4.5

    let num_two: Point<f64, i32> = Point { x: 1.2, y: 4.3, z: 534
};
    println!("{} {} {}", num_two.get_x(), num_two.get_y(),
num_two.get_z());
    // 1.2 4.3 534
}
```

Hasilnya ketika di run:

```
warning: `generics_2` (bin "generics_2")
    Finished dev [unoptimized + debuginfo]
     Running `target\debug\generics_2.exe`
502 120 4.5
1.2 4.3 534
```

A.37.7. Method khusus untuk spesifik tipe parameter generic tertentu

Pada contoh di atas, struct `Point<T, U>` bisa digunakan dalam banyak kombinasi tipe data, misalnya: `Point<i32, f64>`, `Point<i8, i32>`, `Point<f32, u64>`, dan lainya.

Diluar itu, bisa lho mendefinisikan method hanya untuk tipe parameter generic. Misalnya, method hanya bisa diakses ketika `T` adalah `i32` dan `U` adalah `f64`. Caranya kurang lebih seperti ini:

```
impl Point<i32, f64> {

    fn get_x(&self) -> &i32 {
        &self.x
    }

    fn get_y(&self) -> &i32 {
        &self.y
    }

    fn get_z(&self) -> &f64 {
        &self.z
    }
}
```

Block kode `impl` tidak diterapkan dengan notasi penulisan `impl<T, U> Point<T, U>`, melainkan `impl Point<i32, f64>`. Tipe `T` diganti dengan `i32` dan `U` diganti `f64`. Dengan penulisan yang seperti ini, maka method dalam block kode hanya bisa diakses ketika data memiliki tipe data sesuai dengan yang dideklarasikan di block kode `impl` (yang pada contoh di atas adalah `Point<i32, f64>`).

Bisa dilihat pada gambar berikut, sekarang statement pengaksesan method `num_two` menjadi error, karena method-method tersebut hanya tersedia untuk tipe data `Point<i32, f64>` sedangkan `num_two` bertipe `Point<f64, i32>`.

```
impl Point<i32, f64> {

    fn get_x(&self)  > &i32 {
        &self.x
    }

    fn get_y(&self)  > &i32 {
        &self.y
    }

    fn get_z(&self)  > &f64 {
        &self.z
    }
}

▶ Run | Debug
fn main() {
    let num_one: Point<i3
    println!("{} {} {}",
    // 102 120 0.2

    let num_two: Point<f6
    println!("{} {} {}", num_two.get_x(), num_two.get_y(), num_two.get_z());
    // 1.2 4.3 514
}
```

```
[unknown]

no method named `get_x` found for struct `Point<f64, i32>` in the current scope
the method was found for
  `Point<i32, f64>` rustc(Click for full compiler diagnostic)

main.rs(1, 1): method `get_y` not found for this

View Problem (Alt+F8)   No quick fixes available
```

Jika ada keperluan untuk mendeklarasikan method lainnya khusus untuk tipe lainnya, cukup tulis lagi block `impl` diikuti dengan tipe yang diinginkan. Misalnya:

```rust
impl Point<i32, f64> {
    // method untuk tipe data Point<i32, f64>
}

impl Point<f32, u64> {
    // method untuk tipe data Point<f32, u64>
}

impl Point<i8, i64> {
    // method untuk tipe data Point<i8, i64>
}

// ...
```

A.37.8. Generics enum

Generic juga bisa diterapkan pada tipe enum. Caranya tulis saja deklarasi parameter generic setelah nama enum, lalu gunakan parameter generic-nya sesuai kebutuhan.

Contohnya pada kode berikut ini, enum `Kendaraan` memiliki parameter generic `T` yang tipe tersebut dipakai pada value enum `Gledekan(T)`.

```
enum Kendaraan<T> {
    Skateboard,
    SepedaPancal,
    Gledekan(T),
}

let kendaraan1 = Kendaraan::<&str>::Skateboard;
let kendaraan2 = Kendaraan::<&str>::SepedaPancal;
let kendaraan3 = Kendaraan::<&str>::Gledekan("Artco");
```

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../generics
```

◉ Referensi

- https://doc.rust-lang.org/book/ch10-01-syntax.html
- https://doc.rust-lang.org/std/fmt/trait.Debug.html
- https://doc.rust-lang.org/std/ops/trait.Add.html
- https://doc.rust-lang.org/std/cmp/trait.PartialOrd.html

A.38. Tipe Data → Option

`Option` adalah salah satu tipe data penting pada Rust programming, digunakan untuk menampung data yang isinya bisa berpotensi kosong (`None`). Chapter ini membahas tentang tipe data tersebut.

A.38.1. Konsep `Option`

Tipe data `Option` adalah enum dengan isi 2 buah enum value:

- `Option::Some<T>` (atau `Some<T>`), digunakan untuk menandai bahwa data memiliki value/nilai.
- `Option::None` (atau `None`), digunakan untuk menandai bawah data adalah tidak ada nilainya.

> - *`None` bisa disamakan dengan nilai `null` atau `nil` pada bahasa pemrograman lain.*
> - *`T` merupakan parameter generic. Lebih jelasnya mengenai generic dibahas pada chapter Generics.*

Tipe data `Option` memiliki notasi penulisan `Option<T>` dimana `T` adalah tipe data sebenarnya yang dibungkus oleh enum value `Some`.

Berikut adalah contoh cara penerapan `Option`.

```rust
fn divider(a: i32, b: i32) -> Option<i32> {
    if b == 0 {
        return None;
    }

    let result = a / b;
    return Some(result);
}

fn main() {
    let result1 = divider(10, 5);
    println!("result: {:?}", result1);

    let result2: Option<i32> = divider(10, 0);
    println!("result: {:?}", result2);
}
```

Fungsi `divider` di atas tugasnya adalah melakukan operasi aritmatika pembagian angka numerik `i32`, parameter `a` dibagi `b`.

Pada fungsi tersebut terdapat pengecekan apabila nilai `b` adalah `0`, maka yang dikembalikan adalah `None`, selainnya maka hasil operasi pembagian dikembalikan dibungkus dalam enum value `Some<i32>`. Bisa dilihat pada statement return value fungsi `divider`, nilai `result` dibungkus menggunakan tipe `Some`.

Fungsi `divider` nilai baliknya bertipe `Option<i32>`. Dari tipe data yang digunakan nantinya bisa diprediksi pasti akan ada 2 potensi value:

- Return value adalah enum value `None`, muncul ketika nilai `b` adalah `0`
- Return value adalah nilai hasil numerik yang dibungkus oleh enum value `Some<i32>`

Output program di atas saat di-run:

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasa
   Compiling playground v0.1.0 (D:\Labs\Adam Studio\Ebook\das
    Finished dev [unoptimized + debuginfo] target(s) in 0.57s
     Running `target\debug\playground.exe`

result: Some(2)
result: None
```

A.38.2. Pattern matching pada tipe `Option`

Dalam penerapannya, ketika ada data bertipe `Option` artinya data tersebut berpotensi untuk berisi nilai `None` atau `Some<T>`, pasti antara 2 nilai tersebut.

Umumnya penggunaan tipe `Option` selalu diikuti dengan seleksi kondisi. Keyword `if` bisa digunakan dalam seleksi kondisi, namun dalam prakteknya lebih baik menggunakan keyword `match` karena memberikan kemudahan dalam pengaksesan nilai `T` milik `Some` (dimana `T` adalah data yang kita cari dibungkus dalam enum value `Some`).

Mari kita praktekan. Ubah isi fungsi `main` dengan kode berikut:

```
let result1 = divider(10, 5);
match result1 {
    None    => println!("cannot divide by 0"),
    Some(x) => println!("result: {x}"),
}

let result2 = divider(10, 0);
match result2 {
    None    => println!("cannot divide by 0"),
    Some(x) => {
        println!("result: {}", x)
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrograman rust\dasa
   Compiling playground v0.1.0 (D:\Labs\Adam Studio\Ebook\das
    Finished dev [unoptimized + debuginfo] target(s) in 0.22s
     Running `target\debug\playground.exe`

result: 2
cannot divide by 0
```

Bisa dilihat cara mengambil nilai `T` dari enum value `Some<T>` cukup mudah dengan menggunakan keyword `match`. Penerapan `match` untuk seleksi kondisi biasa disebut dengan **pattern matching** dan teknik ini sangat fleksibel dan advance.

Sebagai contoh, dengan penerapan match yang seperti ini kita bisa meng-handle 3 skenario seleksi kondisi:

```
let result1 = divider(10, 5);
match result1 {
    None    => println!("cannot divide by 0"),
    Some(2) => println!("the result is 2"),
    Some(x) => println!("result: {x}"),
}
```

- Kondisi ke-1: jika nilai adalah `None`, maka munculkan pesan `cannot divide by 0`
- Kondisi ke-2: jika nilai adalah `2`, maka munculkan pesan `the result is 2`
- Kondisi ke-1: jika nilai adalah `Some` selain dari `None` dan `Some(2)`, maka munculkan pesan `result: {x}`

◉ Tips pattern matching

Silakan perhatikan kode yang sudah kita praktekan berikut ini:

```
let result = divider(10, 5);
match result {
    None    => println!("cannot divide by 0"),
    Some(x) => println!("result: {x}"),
}
```

Penerapan pattern matching seperti contoh di atas memiliki konsekuensi, yaitu variabel `x` hanya bisa diakses pada block `Some(x)` saja.

Adakalanya kita butuh untuk mengeluarkan variabel `x` ke luar block. Hal seperti ini mudah untuk dilakukan, dan ada beberapa cara yang bisa dipilih, namun menurut penulis yang paling elegan adalah cara berikut ini:

```
fn main() {
    let result = match divider(10, 5) {
        None => {
            println!("cannot divide by 0");
            0
        },
        Some(x) => x,
    };

    println!("result: {:?}", result);
}
```

Statement `divider(10, 5)` mengembalikan data bertipe `Option<i32>`. Data tersebut digunakan pada keyword `match` seperti biasa. Namun pada contoh di atas ada yang berbeda, yaitu return value dari statement `match` ditampung ke variabel (`result`).

Isi dari pattern matching `match` sendiri ada dua:

- Ketika block `None` match, pesan error di-print kemudian nilai `0` dijadikan

return statement `match`.

- Ketika block `Some` match, data `x` dijadikan return value statement `match`.

Dengan penerapan pattern matching seperti di atas, maka variabel `result` akan selalu berisi data hasil operasi `divider(10, 5)`. Dengan pengecualian ketika ada error, pesan errornya dimunculkan kemudian hasil operasi pembagian di-set sebagai `0`.

> *Lebih jelasnya mengenai pattern matching dibahas pada chapter Pattern Matching*

A.38.3. Method tipe data `Option`

◉ Method `unwrap`

Isi dari enum value `Some<T>` bisa diakses tanpa menggunakan keyword `match` dengan cara memanfaatkan method `unwrap` milik `Option<T>`.

```
let result1 = divider(10, 5);
if result1 != None {
    let number = result1.unwrap();
    println!("result: {}", number);
}
```

Penggunaan method tersebut sangat dianjurkan diiringi dengan seleksi kondisi untuk memastikan data `Option` tidak berisi `None`. Jika data ternyata adalah `None` dan method `unwrap` diakses, hasilnya adalah panic error. Contohnya bisa dilihat pada gambar berikut:

```
36      let result2: Option<i32> = divider(a: 10, b: 0);
37      let number: i32 = result2.unwrap();
38      println!("result: {}", number);
39
```

```
PROBLEMS    OUTPUT    GITLENS    DEBUG CONSOLE    TERMINAL

(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarpemrogramanrust\examples\play
  Compiling playground v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarpemrogr
   Finished dev [unoptimized + debuginfo] target(s) in 0.26s
    Running `target\debug\playground.exe`
thread 'main' panicked at 'called `Option::unwrap()` on a `None` value', src\main.rs:37:26
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
error: process didn't exit successfully: `target\debug\playground.exe` (exit code: 101)
```

Selain method `unwrap` ada beberapa method sejenis lainnya yang bisa dimanfaatkan untuk pengambilan nilai `T`. Kita akan bahas satu per satu.

◉ Method `is_some` & `is_none`

Method `is_some` menghasilkan nilai `true` jika data isinya adalah enum value `Some<T>`. Sedangkan `is_none` bernilai `true` jika data berisi `None`.

Contoh penerapannya bisa dilihat pada program berikut. Kesemua seleksi kondisi pada konteks ini menghasilkan nilai `true`.

```rust
let result1 = divider(10, 5);

if result1 != None {
    let number = result1.unwrap();
    println!("result: {}", number);
}

if result1.is_some() {
    let number = result1.unwrap();
    println!("result: {}", number);
}
```

◉ Method `unwrap_or_default`

Method `unwrap_or_default` mengembalikan nilai `T` ketika data berisi `Some<T>`. Jika data ternyata isinya adalah `None`, maka nilai yang dikembalikan adalah *default value* dari tipe data `T`.

Sebagai contoh, pada kode berikut statement `divider(10, 0)` return type-nya adalah `Option<i32>`, sedangkan return value-nya adalah `None`. Pengaksesan method `unwrap_or_default` menghasilkan *default value* dari tipe data `i32`, yaitu `0`

```
let result2 = divider(10, 0);
let number = result2.unwrap_or_default();
println!("result: {}", number);
// result: 0
```

◉ Method `unwrap_or`

Method ini mengembalikan nilai `T` ketika data berisi `Some<T>`, namun jika data ternyata isinya adalah `None`, maka nilai yang dikembalikan adalah argument pemanggilan method tersebut.

```
let result2 = divider(10, 0);
let number = result2.unwrap_or(0);
println!("result: {}", number);
// result: 0
```

Pada contoh di atas argument pemanggilan method `unwrap_or` adalah angka `0`, artinya ketika `result2` isinya adalah `None` maka angka `0` adalah return value pengaksesan method `unwrap_or`.

◉ Method `unwrap_or_else`

Method ini mengembalikan nilai `T` ketika data berisi `Some<T>`, namun jika data isinya adalah `None`, maka nilai yang dikembalikan adalah hasil eksekusi closure yang disisipkan saat memanggil method `unwrap_or_else`. Contoh pengaplikasiannya:

```
let result2 = divider(10, 0);
let number = result2.unwrap_or_else(|| 0);
println!("result: {}", number);
// result: 0
```

Closure harus dalam notasi `FnOnce() -> T` dimana `T` pada konteks ini adalah `i32`.

> *Closure `|| 0` adalah kependekan dari `|| -> i32 { 0 }`.*
>
> *Lebih jelasnya mengenai closure dibahas pada chapter Closures.*

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../option_type
```

◉ Chapter relevan lainnya

- Generics
- Pattern Matching
- Closures

◉ Referensi

- https://doc.rust-lang.org/book/ch18-03-pattern-syntax.html
- https://doc.rust-lang.org/std/option/index.html

A.39. Tipe Data → Result

Chapter ini membahas tentang tipe data `Result`. Tipe data ini digunakan untuk menampung nilai hasil suatu proses yang isinya adalah bisa sukses (`Ok`) atau error (`Err`).

Tipe data `Result` biasa digunakan untuk menampung hasil eksekusi proses dan error handling.

A.39.1. Konsep `Result`

Tipe data `Result` adalah enum dengan isi 2 buah enum value:

- `Result::Ok<T>` (atau `Ok<T>`), digunakan untuk menandai bahwa data isinya adalah kabar baik (oke / mantab / jos / sukses).
- `Result::Err<E>` (atau `Err<E>`), digunakan untuk menandai bawah data berisi kabar buruk.

> - *`T` dan `E` merupakan parameter generic. Lebih jelasnya mengenai generic dibahas pada chapter Generics.*

Tipe data `Result` memiliki notasi penulisan `Result<T, E>` dimana `T` digunakan pada enum value `Ok<T>` dan `E` digunakan enum value `Err<E>`.

Cara penerapan tipe data ini bisa dilihat pada kode berikut:

```rust
#[derive(Debug)]
enum MathError {
    DivisionByZero,
    InfinityNumber,
    OtherError,
}

fn main() {
    let result1 = divider(10.0, 5.0);
    println!("result: {:?}", result1);

    let result2: Result<f64, MathError> = divider(10.0, 0.0);
    println!("result: {:?}", result2);
}

fn divider(a: f64, b: f64) -> Result<f64, MathError> {
    if b == 0.0 {
        return Err(MathError::DivisionByZero);
    }

    let result = a / b;
    return Ok(result);
}
```

Fungsi `divider` di atas tugasnya adalah melakukan operasi aritmatika pembagian angka numerik `f64`, parameter `a` dibagi `b`.

Pada fungsi tersebut terdapat pengecekan apabila nilai `b` adalah `0`, maka yang dikembalikan adalah `Err<E>` dengan `E` berisi pesan error, selainnya maka hasil operasi pembagian dikembalikan dibungkus dalam enum value `Ok<f64>`.

Fungsi `divider` nilai baliknya bertipe `Result<f64, MathError>`. Dari tipe data yang digunakan nantinya bisa diprediksi pasti akan ada 2 potensi value:

- Return value adalah enum value `Err<MathError>`, muncul ketika nilai `b` adalah `0`
- Return value adalah nilai hasil numerik yang dibungkus oleh enum value `Ok<f64>`

Output program di atas saat di-run:

```
warning: `playground` (bin "playground") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.40s
     Running `target\debug\playground.exe`
result: Ok(2.0)
result: Err(DivisionByZero)
```

A.39.2. Pattern matching pada tipe `Result`

Dalam penerapannya, ketika ada data bertipe `Result` artinya data tersebut berpotensi untuk berisi nilai `Err<E>` atau `Ok<T>`, pasti antara 2 nilai tersebut.

Umumnya penggunaan tipe `Result` selalu diikuti dengan pattern matching menggunakan keyword `match`. Selain itu keyword `if` sebenarnya juga bisa diterapkan pada pattern matching tipe data ini, namun kurang dianjurkan.

Mari kita praktekan. Ubah isi fungsi `main` dengan kode berikut:

```
let result = divider(10.0, 5.0);
match result {
    Err(m) => println!("ERROR! {:?}", m),
    Ok(r)  => println!("result: {r:.2}"),
}
```

```
warning: `playground` (bin "playground") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.40s
     Running `target\debug\playground.exe`
result: 2.00
```

Bisa dilihat pada kode di atas mudahnya pengambilan nilai `m` dari `Err(m)` dan juga `r` dari `Ok(r)`. Penerapan `match` untuk seleksi kondisi biasa disebut dengan **pattern matching** dan teknik ini sangat fleksibel dan advance.

Sebagai contoh, dengan penerapan match yang seperti ini kita bisa meng-handle 5 skenario seleksi kondisi:

```
let result = divider(10.0, 5.0);
match result {
    Err(MathError::DivisionByZero) => println!("ERROR! unable to divide number by 0"),
    Err(MathError::InfinityNumber) => println!("ERROR! result is infinity number (∞)"),
    Err(_)                         => println!("ERROR! unknown error"),
    Ok(2.0)                        => println!("the result is 2"),
    Ok(x)                          => println!("result: {x:.2}"),
}
```

- Kondisi ke-1: jika nilai adalah `Err(MathError::DivisionByZero)`, maka munculkan pesan `ERROR! unable to divide number by 0`.
- Kondisi ke-2: jika nilai adalah `Err(MathError::InfinityNumber)`, maka munculkan pesan `ERROR! result is infinity number (∞)`.
- Kondisi ke-3: jika nilai adalah `Err` selain dari `Err(MathError::DivisionByZero)` dan `Err(MathError::InfinityNumber)`, maka munculkan pesan `ERROR! unknown error`.
- Kondisi ke-4: jika nilai adalah `Ok(2.0)`, maka munculkan pesan `the`

`result is 2`.

- Kondisi ke-5: jika nilai adalah `Ok` selain dari `Ok(2.0)`, maka munculkan pesan `result: {x:.2}`.

◉ Tips pattern matching

Silakan perhatikan kode yang sudah kita praktekan berikut ini:

```
let result = divider(10.0, 5.0);
match result {
    Err(m) => println!("ERROR! {:?}", m),
    Ok(r)  => println!("result: {r:.2}"),
}
```

Penerapan pattern matching seperti contoh di atas memiliki konsekuensi, yaitu variabel `r` hanya bisa diakses pada block `Ok(r)` saja.

Adakalanya kita butuh untuk mengeluarkan variabel `r` ke luar block. Hal seperti ini mudah untuk dilakukan, dan ada beberapa cara yang bisa dipilih, namun menurut penulis yang paling elegan adalah cara berikut ini:

```
fn main() {
    let result: f64 = match divider(10.0, 5.0) {
        Err(m) => {
            println!("ERROR! {:?}", m);
            0.0
        },
        Ok(r) => r,
    };

    println!("result: {:?}", result);
}
```

Statement `divider(10.0, 5.0)` mengembalikan data bertipe `Result<f64, MathError>`. Data tersebut digunakan pada keyword `match` seperti biasa. Namun pada contoh di atas ada yang berbeda, yaitu return value dari statement `match` ditampung ke variabel (`result`).

Isi dari pattern matching `match` sendiri ada dua:

- Ketika block `Err(m)` match, error di-print kemudian nilai `0.0` dijadikan return statement `match`.
- Ketika block `Ok` match, data `r` dijadikan return value statement `match`.

Dengan penerapan pattern matching seperti di atas, maka variabel `result` akan selalu berisi data hasil operasi `divider(10.0, 5.0)`. Dengan pengecualian ketika ada error, pesan errornya dimunculkan kemudian hasil operasi pembagian di-set sebagai `0.0`.

> *Lebih jelasnya mengenai pattern matching dibahas pada chapter Pattern Matching*

A.39.3. Method tipe data `Result`

◉ Method `is_ok` & `unwrap`

Isi dari enum value `Ok<T>` bisa diakses tanpa menggunakan keyword `match` dengan cara memanfaatkan method `unwrap` milik `Result<T, E>`. Sebelum mengakses method tersebut sangat dianjurkan untuk mengecek apakah data berisi `Ok<T> atau tidak`, karena jika data adalah `Err<E>` pengaksesan method `unwrap` menghasilkan error.

Pengecekan nilai ok atau tidak bisa dilakukan menggunakan method `is_ok`.

```
let result = divider(10.0, 5.0);
if result.is_ok() {
    let number = result.unwrap();
    println!("result: {}", number);
    // result: 2
}
```

```
warning: `playground` (bin "playground") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.41s
     Running `target\debug\playground.exe`

result: 2
```

◉ Method `as_ref`

Method `as_ref` digunakan untuk mengakses reference `T` dan `E` pada `Result<T, E>`. Method ini sering kali dibutuhkan untuk menghindari terjadinya *move semantics* pada owner data bertipe `Result<T, E>`.

Method `as_ref` mengembalikan data dalam tipe `Result<&T, &E>`. Jadi reference yang dipinjam bukan milik `Result`-nya melainkan milik `T` dan `E`.

```
let result: Result<f64, MathError> = divider(10.0, 0.0);
let result_borrow: Result<&f64, &MathError> = result.as_ref();
```

> *Lebih jelasnya mengenai move semantics dibahas pada chapter Ownership*

◉ Method `is_err` & `err`

Method `err` mengembalikan data dalam tipe `Err<E>`. Pada pengaksesan method ini, pastikan untuk mengecek apakah `Result` berisi data error atau ok dengan via method `is_err`. Selain itu, wajib untuk menggunakan method `as_ref` sebelum method `err` agar ownership data `Result` tidak berpindah (*move semantics*).

```
let result = divider(10.0, 0.0);
if result.is_err() {
    let err = result.as_ref().err();
    let message = err.unwrap();
    println!("error: {:?}", message);
    // error: DivisionByZero
}
```

◉ Method `ok`

Aturan yang sama juga berlaku pada pengaksesan method `ok` yang mengembalikan data `Ok<T>`. Method `as_ref` harus diakses terlebih dahulu sebelum memanggil method `ok` agar tidak terjadi *move semantics*.

```
let result = divider(10.0, 5.0);
if result.is_ok() {
    let data = result.as_ref().ok();
    let number = data.unwrap();
    println!("result: {:?}", number);
    // result: 2
}
```

◉ Method `unwrap_or_default`

Method `unwrap_or_default` milik `Result<T, E>` mengembalikan nilai `T` ketika data berisi `Ok<T>`, namun jika data berisi `Err<E>` maka yang dikembalikan adalah *default value* dari tipe data `T`.

```
let result = divider(10.0, 0.0);
let number = result.unwrap_or_default();
println!("result: {}", number);
// result: 0
```

◉ Method `unwrap_or`

Method `unwrap_or` milik `Result<T, E>` mengembalikan nilai `T` ketika data berisi `Ok<T>`, namun jika data ternyata isinya adalah `Err<E>` maka yang dikembalikan adalah argument pemanggilan method tersebut.

```
let result = divider(10.0, 0.0);
let number = result.unwrap_or(0.0);
println!("result: {}", number);
// result: 0
```

◉ Method `unwrap_or_else`

Method ini mengembalikan nilai `T` ketika data berisi `Ok<T>`, namun jika data isinya adalah `Err<E>` maka yang dikembalikan adalah hasil eksekusi closure yang disisipkan saat memanggil method `unwrap_or_else`. Contoh pengaplikasiannya:

```
let result = divider(10.0, 0.0);
let number = result.unwrap_or_else(|_| 0.0);
println!("result: {}", number);
// result: 0
```

Closure harus dalam notasi `FnOnce(E) -> T` dimana `T` pada konteks ini adalah `f64`.

> *Lebih jelasnya mengenai closure dibahas pada chapter Closures.*

A.39.4. Error handling tipe `Result`

Tipe data `Result<T, E>` banyak digunakan pada fungsi milik Rust standard library, dan kita selaku programmer pastinya juga akan menggunakannya dalam *real life* project.

Tipe ini dipakai salah satunya untuk manajemen error. Lebih jelasnya mengenai topik tersebut dibahas pada chapter Error Handling & Panic

---

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../result_type
```

◉ Chapter relevan lainnya

- Generics
- Pattern Matching
- Closures

◉ Work in progress

- Operator `?`

◉ Referensi

- https://doc.rust-lang.org/book/ch18-03-pattern-syntax.html
- https://doc.rust-lang.org/std/result/index.html

A.40. Pattern Matching

Chapter ini membahas tentang pattern matching, sebuah teknik yang lebih *advance* dibanding seleksi kondisi biasa.

Dalam pattern matching, pengecekan dilakukan dengan melihat kecocokan suatu pola/pattern.

A.40.1. Keyword `match`

Keyword `match` digunakan untuk pattern matching. Contoh penerapan versi sederhananya bisa dilihat berikut:

```
let time = "morning";

match time {
    "morning"   => println!("isuk"),
    "afternoon" => println!("awan"),
    "evening"   => println!("bengi"),
    _           => println!("mbuh kapan"),
}
```

```
warning: `pattern_matching` (bin "pattern_matching") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 1.48s
     Running `target\debug\pattern_matching.exe`
isuk
```

Pada contoh di atas, `time` dicek nilainya menggunakan keyword `match` dengan 4 buah klausul:

- Jika value-nya `morning`, tampilkan pesan `isuk`
- Jika value-nya `afternoon`, tampilkan pesan `awan`
- Jika value-nya `evening`, tampilkan pesan `bengi`
- Jika tidak ada yang cocok dari klausus di atas, maka tampilkan pesan `mbuh kapan`

Contoh di atas adalah ekuivalen dengan seleksi kondisi `if` berikut:

```
let time = "morning";

if time == "morning" {
    println!("isuk")
} else if time == "afternoon" {
    println!("awan")
} else if time == "evening" {
    println!("bengi")
} else {
    println!("mbuh kapan")
}
```

Ada satu syarat yang harus dipenuhi dalam penerapan pattern matching, yaitu semua kondisi yang memungkinkan harus ditulis, harus lengkap. Ibarat `if` yang harus ada block `else`-nya.

Variabel `_` digunakan sebagai else-nya block `match`. Tanpa adanya kondisi `_` maka besar kemungkinan block `match` error jika klausulnya tidak lengkap.

```
let time: &str

non-exhaustive patterns: `&_` not covered
The matched value is of type `&str` rustc(Click for full compiler diagnostic)

main.rs(24, 49): ensure that all possible cases are being handled by adding a
match arm with a wildcard pattern or an explicit pattern as shown: `,
&_ => todo()`

View Problem (Alt+F8)   No quick fixes available

fn main()
    let ti
    match time {
        "morning" -> println("isuk"),
        "afternoon" -> println("awan"),
        "evening" -> println("bengi"),
    }
```

◉ Menampung nilai balik `match`

Block statement `match` bisa saja menghasilkan return value. Contohnya bisa dilihat berikut ini, hasil dari pattern matching ditampung ke variabel `time_but_in_javanese`.

```
let time = "morning";

let time_but_in_javanese = match time {
    "morning"   => "isuk",
    "afternoon" => "awan",
    "evening"   => "bengi",
    _           => "mbuh kapan",
};

println!("{time_but_in_javanese}");
```

A.40.2. Pattern matching

Contoh di atas bisa dikategorikan sebagai seleksi kondisi biasa meskipun menggunakan keyword `match`. Setelah ini kita akan pelajari macam-macam

pattern/pola yang di-support dalam pattern matching di Rust.

◉ Pengecekan nilai enum

Tipe data Option adalah salah satu enum yang paling sering dipakai pada pattern matching. Enum `Option` memiliki 2 enum value, `Some` yang merepresentasikan sebuah nilai, dan `None` yang berarti tidak ada nilai.

Pattern matching pada enum cukup mudah, caranya bisa dilihat pada contoh berikut:

```
let value: Option<i32> = Option::Some(5);

match value {
    Some(1) => println!("one"),
    Some(2) => println!("two"),
    Some(x) => println!("{x} greater than two"),
    _       => println!("none"),
}
```

Variabel `value` nilainya adalah `Some(5)`. Variabel tersebut dimasukan ke block `match` dengan 4 buah kondisi pengecekan:

- Jika `value` nilainya `Some(1)`, tampilkan pesan `one`
- Jika `value` nilainya `Some(2)`, tampilkan pesan `two`
- Jika `value` nilainya `Some(x)`, tampilkan pesan `{x} greater than two`
- Jika tidak ada yang cocok dari klausus di atas, maka tampilkan pesan `none`

Tipe `Option` pasti berpotensi berisi `Some` atau `None`, tidak mungkin selainnya. Klausul terakhir di contoh di atas (`_ => println!("none")`) terpenuhi ketika nilai `value` adalah `None`. Pada konteks ini mengganti `_` dengan `None` menjadikan klausul pada pattern matching tetap lengkap.

```
match value {
    Some(1) => println!("one"),
    Some(2) => println!("two"),
    Some(x) => println!("{x} greater than two"),
    None    => println!("none"),
}
```

> *Lebih jelasnya mengenai `Some` dan `None` dibahas pada chapter Tipe Data → Option*

◉ Pattern `|` dan `..`

Klausul pattern matching bisa berisi operasi `OR` maupun `IN` caranya dengan memanfaatkan operator berikut:

- Operator `|` digunakan sebagai logika `OR`
- Operator `..` atau `..=` digunakan sebagai logika `IN`

Contoh penerapannya:

```
let value = 6;

match value {
    1 | 2 => println!("one or two"),
    3..=5 => println!("three through five"),
    6     => println!("six"),
    _     => println!("other number"),
}
```

- Jika `value` nilainya `1` atau `2`, tampilkan pesan `one or two`
- Jika `value` nilainya antara `3` hingga `5`, tampilkan pesan `three through`

`five`

- Jika `value` nilainya `6`, tampilkan pesan `six`
- Jika tidak ada yang cocok dari klausus di atas, maka tampilkan pesan `other number`

Pattern di atas juga bisa diterapkan dalam variabel enum value, contohnya:

```
let value: Option<i32> = Some(5);

match value {
    Some(1 | 2) => println!("one or two"),
    Some(3..=5) => println!("three through five"),
    Some(6)     => println!("six"),
    Some(x)     => println!("{x} greater than six"),
    _           => println!("none"),
}
```

> *Operator `|` memiliki 2 kegunaan:*
>
> - *Pada statement biasa, fungsinya adalah untuk **bitwise OR**.*
> - *Pada pattern matching, fungsinya untuk **OR**, kegunaannya sama seperti `||` pada statement biasa.*
>
> *Lebih jelasnya mengenai bitwise operator dibahas pada chapter Bitwise Operation*

◉ Match guard

Match guard adalah teknik menambahkan sub seleksi kondisi pada klausul `match`. Contoh:

```
let value = Some(4);

let message = match value {
    Some(x) if x % 2 == 0 => format!("number {} is even", x),
    Some(x)               => format!("number {} is odd", x),
    None                  => String::new(),
};

println!("{message}");
```

Klausul pertama di atas, yaitu `Some(x)` ditambahkan match guard `if x % 2 == 0`.

◉ Binding `@`

Operator `@` digunakan untuk menampung nilai klausul `match` yang default-nya tidak bisa ditampung. Agar lebih jelas, silakan pelajari pattern matching berikut:

```
let value = 3;
match value {
    1 | 2 => println!("one or two"),
    3..=5 => println!("three through five"),
    6     => println!("six"),
    _     => println!("other number"),
}
```

Klausul `1 | 2` dan `3..=5` nilai by default tidak bisa diakses. Kita hanya tau bahwa nilai pasti antara `1` dan `2` untuk klausul `1 | 2`, dan `3 / 4 / 5` untuk klausul `3..=5`.

Nilai pasti klausul tersebut bisa ditampung menggunakan operator `@`.

```
let value = 3;
match value {
    n @ (1 | 2) => println!("one or two ({})", n),
    n @ 3..=5   => println!("three through five ({})", n),
    6           => println!("six"),
    _           => println!("other number"),
}
```

> *Khusus untuk penggunaan `@` binding pada operator `|`, pada penulisan klausul seleksi kondisinya harus diapit tanda `()`.*

◉ `if let`

Untuk memahami pattern matching menggunakan keyword `if let`, silakan pelajari kode berikut terlebih dahulu.

```
let value: Option<i32> = Some(5);

match value {
    Some(1) => println!("one"),
    Some(x) => println!("{x} greater than two"),
    _       => println!("none"),
}
```

Pattern matching di atas cukup mudah dipahami, isinya ada 2 kondisi `Some` dan 1 buah else (menggunakan `_`).

Block kode tersebut jika dikonversi ke bentuk `if` hasilnya seperti ini:

```
let value = Some(5);
```

Dari sini cukup jelas kegunaan dari `if let`. Meskipun menggunakan operator `=` (bukan `==`) block kode seleksi kondisi di atas adalah pattern matching, yang isinya melakukan pengecekan sama persis seperti pattern matching pada kode sebelumnya.

Tambahan contoh, 2 block kode berikut adalah juga ekuivalen.

```
let value = 6;
match value {
    1 | 2 => println!("one or two"),
    3..=5 => println!("three through five"),
    6     => println!("six"),
    _     => println!("other number"),
}

// ... vs ...

let value = Some(5);
if let Some(1 | 2) = value {
    println!("one or two");
} else if let Some(3..=5) = value {
    println!("three through five");
} else if let Some(6) = value {
    println!("six");
} else {
    println!("other number");
}
```

A.40.3. Destructuring assignment

◉ Struct destructuring

Operasi *destructuring* (menampung item suatu tipe) bisa dilakukan

menggunakan pattern matching.

Pada kode di bawah ini, variabel `p` yang bertipe struct `Point { x: i32, y: i32 }` dimasukan pada block pattern matching. Item dari struct tersebut di-*destructure* ke variabel `x` dan `y` masing-masing klausul pattern matching item-nya ditampung ke variabel `x` dan `y`.

```
struct Point {
    x: i32,
    y: i32,
}

let p = Point { x: 0, y: 7 };

match p {
    Point { x, y: 0 } => println!("x axis at {x}"),
    Point { x: 0, y } => println!("y axis at {y}"),
    Point { x, y }    => println!("axis: ({x}, {y})")
}
```

- Jika `p.y` nilainya `0`, tampilkan pesan `x axis at {x}`
- Jika `p.x` nilainya `0`, tampilkan pesan `y axis at {y}`
- Jika tidak ada yang cocok dari klausus di atas, maka tampilkan pesan `axis: ({x}, {y})`

Operasi destructuring hasilnya pasti sukses, karena alasan ini keyword `match` boleh tidak digunakan. Contohnya bisa dilihat pada kode berikut, variabel `p` di-*destructure* ke variabel baru yaitu `x` dan `y`.

```
let Point { x, y } = p;

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

◉ Enum destructuring

Destructuring juga bisa dilakukan pada tipe data enum caranya dengan menggunakan keyword `match` atau `if` (wajib menggunakan salah satu keyword tersebut).

```
enum Color {
    Black,
    White,
    Rgb(i32, i32, i32)
}

let color = Color::Rgb(0, 160, 255);

if let Color::Rgb(r, g, b) = color {
    println!("r: {r}");
    println!("g: {g}");
    println!("b: {b}");
}

match color {
    Color::Rgb(r, g, b) => println!("r: {r}, g: {g}, b: {b}"),
    _                   => println!("other color")
}
```

◉ Tuple destructuring

Tuple bisa di-destructure secara langsung tanpa menggunakan keyword `if` atau `match`.

```
let grades = ("A", "B", "C");
```

◉ Variabel `_`

Variabel `_` bisa dimanfaatkan pada statement *destructuring* untuk menampung item yang tidak digunakan. Contoh penerapannya bisa dilihat di bawah ini. Tuple `numbers` di-destructure dan hanya diambil elemen ke-2-nya.

```
let numbers = (2, 4, 32);

let (_, second, _) = numbers;
println!("second number: {second}");
```

◉ Operator `..`

Operator `..` bisa digunakan untuk meng-exclude item dalam range tertentu. Sebagai contoh, tuple `numbers` di-destructure dan hanya diambil nilai elemen ke-1 dan terakhirnya.

```
let numbers = (2, 4, 8, 16, 32);

let (first, .., last) = numbers;
println!("first number: {first}");
println!("last number: {last}");
```

Opeartor `..` hanya bisa digunakan pada statement destructuring di posisi tengah, awal, atau akhir (pilih salah satu). Contoh:

```
let (first, .., last) = numbers;
println!("first number: {first}");
println!("last number: {last}");
```

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-
example/../pattern_matching
```

◉ Chapter relevan lainnya

- Seleksi Kondisi → if, else if, else
- Tipe Data → Tuple
- Struct
- Enum
- Tipe Data → Option

◉ Work in progress

- Pembahasan tentang `while let`

◉ Referensi

- https://doc.rust-lang.org/book/ch18-03-pattern-syntax.html
- https://doc.rust-lang.org/rust-by-example/flow_control/match.html

---

A.41. Static Item

Pada chapter ini kita akan bahas tentang apa itu static item, dan perbedaanya dibanding konstanta.

Namun sebelum masuk ke inti pembahasan, mari kita sedikit belajar tentang apa itu *lifetime* dalam Rust programming.

A.41.1. Sekilas tentang *lifetime*

Di Rust ada yang disebut dengan **lifetime**. Lifetime merupakan sebuah identifier yang digunakan compiler untuk memantau berapa lama reference valid.

Di balik layar, Rust compiler (lebih tepatnya Rust borrow checker) menggunakan sebuah anotasi dalam penerapan lifetime. Penulisan anotasinya diawali tanda petik satu `'`, contohnya `'a`, `'b`, dan `'c`.

Untuk sekarang, silakan dipahami bahwa sebuah syntax yang diawali dengan tanda `'` (contohnya seperti `'a`) adalah lifetime.

> *Pembahasan detail mengenai lifetime dibahas pada chapter selanjutnya, yaitu Lifetime.*

A.41.2. Static item

Ok, sekarang kembali ke topik utama, yaitu static. Static adalah item yang

mirip dengan Konstanta, tapi memiliki perbedaan yaitu alamat memory yang dialokasikan untuk menampung data static item adalah fix/jelas. Semua reference terhadap static item mengarah ke alamat memory yang sama.

Dengan karakteristik yang seperti itu, static tepat diterapkan pada data yang sifatnya shared atau bisa diakses secara global.

Ada dua cara membuat static item:

- Menggunakan keyword `static` pada pendefinisian konstanta
- Menggunakan lifetime `'static` pada tipe data string literal (`&str`)

A.41.3. Keyword `static`

Ok, kita terapkan cara pertama, penerapan keyword `static` untuk pembuatan konstanta.

Pada deklarasi konstanta static, harus ditulis juga tipe datanya secara eksplisit.

```
static PI: f64 = 3.14;

fn main() {
    println!("PI: {:?}", PI);
}
```

Cukup mudah bukan?

> *Konstanta static bisa saja didefinisikan mutable, tetapi dengan konsekuensi item tersebut akan menjadi unsafe.*
>
> *Lebih jelasnya mengenai topik ini akan dibahas nantinya pada chapter*

> *terpisah, yaitu Safe & Unsafe.*

Ok, sekarang kita coba terapkan keyword static pada tipe lainnya, contohnya `String`.

```
static PI: String = String::from("v1.2.4");
```

```
alloc::string::String
fn from(s: &str) -> String

Converts a &str into a String.

The result is allocated on the heap.

Go to String

cannot call non const fn `<String as From<&'static str>>::from` in statics
calls in statics are limited to constant functions, tuple structs and tuple
variants rustc(Click for full compiler diagnostic)

View Problem (Alt+F8)    No quick fixes available
```

Hmm, malah error.

Perlu diketahui bahwa keyword `static` bisa digunakan pada semua tipe data primitif. Selain itu bisa juga diterapkan dalam *constants function* (yang nantinya dibahas pada chapter Constant Evaluation), Tuple Struct, dan juga variant Tuple lainnya, tetapi tidak bisa digunakan untuk custom type seperti `String`.

Lalu bagaimana jika ada kebutuhan membuat konstanta bertipe string? Solusinya dengan menggunakan tipe data `&'static str` yang sebentar lagi akan kita bahas.

A.41.4. Lifetime `'static`

Lifetime `'static` digunakan untuk deklarasi reference sebagai static item. Data yang memiliki lifetime ini tidak akan pernah di-dealokasi kecuali eksekusi

program selesai.

> *Karena alasan di atas, ada baiknya data dengan lifetime static dideklarasikan secara global.*
>
> *Dimisalkan ada variabel dengan lifetime ini dideklarasikan dalam suatu block, variabel tersebut tidak akan di-dealokasi meskipun eksekusi block selesai nantinya.*

Lifetime ini biasa dikombinasikan dengan tipe data pointer, contohnya seperti `&str` jika dikombinasikan dengan lifetime `'static` jadinya adalah `&'static str`.

```
const VERSION: &'static str = "v1.2.3";
```

Sebelumnya telah dijelaskan bahwa custom type `String` tidak bisa digunakan untuk menyimpan data string sebagai static item, dan cara di atas ini adalah solusinya.

Penulisannya agak kurang friendly memang (`&'static str`), namun kabar baiknya semenjak Rust versi 1.17 rilis di tahun 2017, by default semua item yang dideklarasikan menggunakan keyword `static` ataupun `const` otomatis memiliki `'static lifetime`. Jadi sekarang cukup tulis saja:

```
const VERSION: &str = "v1.2.3";
```

```
const VERSION: &'static str = "v1.2.3";
```

A.41.5. Static item data literal

Pada chapter Borrowing sempat kita bahas sedikit tentang siapa owner dan borrower data literal.

Tipe `&str` adalah salah satu tipe data yang tidak memiliki owner (atau boleh disimpulkan owner-nya adalah program). Pada variabel yang bertipe ini, yang ia tampung adalah data pinjaman, jadi variabel tersebut bukan owner. Contohnya pada kode berikut konstanta `VERSION` dan `BUILD_COUNTER` di atas adalah borrower.

```
const VERSION: &str = "v1.2.3";
const BUILD_COUNTER: &i32 = &15;
```

Yang penting untuk diperhatikan bukan siapa owner-nya, tapi bagaimana kita memastikan data pinjaman tersebut tidak di-dealokasi. Inilah kenapa lifetime `'static` dan/atau keyword `static` & `const` dipergunakan. Dengan adanya lifetime tersebut, data borrow tidak akan pernah di-dealokasi.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-
example/../static_example
```

◉ Referensi

- https://doc.rust-lang.org/reference/items/static-items.html
- https://doc.rust-lang.org/beta/rust-by-example/custom_types/constants.html
- https://doc.rust-lang.org/rust-by-example/scope/lifetime/static_lifetime.html
- https://stackoverflow.com/questions/49684657/what-is-the-difference-between-str-and-static-str-in-a-static-or-const

---

A.42. Lifetime

Pada chapter ini kita akan belajar tentang lifetime. Lifetime adalah yang digunakan oleh Rust compiler untuk memonitor umur dari references agar tetap dianggap valid.

Normalnya kita tidak perlu berurusan dengan lifetime, karena Rust-lah yang mengelola lifetime sebuah reference. Namun diluar itu, pada beberapa case kita bisa me-manage lifetime data dengan memanfaatkan *annotation*.

Ketika berurusan dengan data primitif maupun non-primitif tak perlu khawatir perihal urusan lifetime. Aspek lifetime hanya perlu diperhatikan sewaktu berurusan dengan data pointer/reference, apalagi kalau data tersebut keluar masuk block scope.

> *Topik lifetime adalah salah satu yang paling membingungkan di Rust. Wajar jika membutuhkan waktu lebih lama untuk menguasainya. Take your time, pelajari pelan-pelan dan ulangi berkali-kali jika perlu.*

A.42.1. Konsep Lifetime

Lifetime adalah yang digunakan oleh Rust compiler untuk memonitor umur dari references agar tetap valid. Lifetime menempel di variabel, lebih tepatnya di reference variabel.

Rust menerapkan default lifetime dalam pengecekan reference. Beberapa aturan pada default lifetime sudah kita pelajari pada chapter sebelumnya, seperti variabel yang hanya akan valid didalam block dan invalid diluar block

dan data yang akan di-dealokasi ketika sudah tidak ada reference-nya.

> *Rust mengidentifikasi default lifetime menggunakan **lifetime elision**, yang juga akan dibahas pada chapter ini.*

Agar mudah memahami konsep default lifetime, mari mulai dengan kode sederhana berikut:

```
fn main() {
    let r;
    {
        let x = 5;
        r = &x;
    }
    println!("r: {}", r);
}
```

Kode di atas kalau di jalankan hasilnya error, karena `x` di-dealokasi ketika block expression selesai dieksekusi, meskipun data tersebut dipinjamkan pada `r` yang scope-nya berada diatasnya.

Kalau diilustrasikan, lifetime variabel `r` dan `x` kurang lebih seperti ini:

```
fn main() {
    let r;                  // -----------+-- 'lf1
    {                       //            |
        let x = 5;          // -+-- 'lf2  |
        r = &x;             //  |         |
    }                       // -+         |
    println!("r: {}", r); //             |
}                           // -----------+
```

Setiap data memiliki default lifetime.

- Variabel `r` memiliki lifetime yang pada contoh di atas diilustrasikan sebagai `'lf1`.
- Variabel `x` memiliki lifetime yang pada contoh di atas diilustrasikan sebagai `'lf2`.
- Lifetime `'lf2` milik variabel `x` menjadikan umur variabel tersebut valid mulai variabel tersebut dideklarasikan, hingga block expression selesai.
- Lifetime `'lf2` sudah tidak valid diluar block expression. Inilah kenapa program di atas menjadi error.
- Lifetime `'lf1` milik variabel `r` menjadikan umur variabel tersebut valid mulai variabel tersebut dideklarasikan, hingga block fungsi `main` selesai.

Default lifetime bisa di-override menggunakan lifetime yang kita definisikan sendiri.

A.42.2. Relasi antara lifetime dengan owner dan borrower

Lifetime menjadi salah satu hal yang wajib diperhatikan ketika bermain dengan references. Operasi seperti melempar reference ke luar scope, atau memasukan reference ke block scope baru berpotensi memunculkan error yang berhubungan dengan lifetime.

Setelah ini kita akan praktek penerapan pembuatan lifetime, namun sebelum itu mari pelajari dulu pembahasan pada section berikut agar tau kenapa dan kapan kita harus menerapkan lifetime yang kita buat sendiri.

```
fn main() {
```

Program di atas menampilkan pesan string via fungsi `print_message`. Data string didapat dari parameter pointer `m` milik fungsi tersebut.

Pada fungsi `main`, ada string bernama `message`, niainya dipinjamkan sebagai argument pemanggilan fungsi `print_message`.

Setelah eksekusi fungsi `print_message` selesai, yang terjadi di block kode fungsi tersebut adalah data `m` di-dealokasi. Hasil dari dealokasi sendiri adalah nilai sebenarnya dikembalikan ke owner (pemilik aslinya). Sampai sini harusnya cukup jelas.

Sekarang lanjut ke contoh ke-2 berikut:

```
fn main() {
    let m: &String = get_message();
    println!("the message: {m}");
}

fn get_message() -> &String {
    let message = String::from("darkspear is better than
zandalari");
    &message
}
```

Esensi program ke-2 ini sama seperti program sebelumnya, yaitu menampilkan pesan string yang sama persis. Perbedaannya, pesan string datanya ada di dalam fungsi `get_message`. Fungsi tersebut dipanggil kemudian reference dari pesan string dipinjamkan, maka dengan ini variabel `m` pada fungsi `main` nilainya adalah data pinjaman (borrowing).

Ketika di run, hasilnya error.

```
consider using the `'static` lifetime: `&'static ` rustc(E0106)
main.rs(22, 21): original diagnostic
missing lifetime specifier
this function's return type contains a borrowed value, but there is no value for it to be
borrowed from rustc(Click for full compiler diagnostic)
main.rs(22, 21): consider using the `'static` lifetime: `&'static `
View Problem (Alt+F8)   Quick Fix... (Ctrl+.)
fn get_message()  -> &String {
    let message: String = String::from("darkspear is better than candalari");
    &message
}
```

Error tersebut muncul karena setelah eksekusi fungsi `get_message` selesai, semua data dalam fungsi tersebut di-dealokasi. Termasuk variabel `message` yang merupakan owner dari data yang dipinjamkan ke variabel `m`.

Ok, jadi owner-nya sudah di-dealokasi, lalu bagaimana nasib dari peminjam data (variabel `m`)? Variabel tersebut menjadi bermasalah, karena owner data aslinya sudah tidak ada di memory, dan itulah kenapa muncul error.

> *Error ini disebut dengan **dangling reference**, muncul ketika data di-share ke variabel lain tapi owner-nya sudah tidak ada di-memory.*
>
> *Di pemrograman Rust, error jenis ini bisa di-identifikasi saat kompilasi.*

Solusi pada error di atas, salah satunya adalah dengan tidak menggunakan tipe data pointer sebagai nilai balik. Gunakan saja tipe data `string`. Solusi ini aman, karena tipe data `string` owner-nya selalu berpindah saat ada operasi asignment, dengan ini maka manajemen memory menjadi efisien.

```
// ganti kode berikut ...
fn get_message() -> &String {
    let message = String::from("darkspear is better than
```

Ok, bagaimana dengan tipe data lain yang mengadopsi *copy semantics*, misalnya ada kebutuhan untuk share reference data tersebut ke block lain. Mari kita coba.

```
fn main() {
    let n = get_number();
    println!("the number: {n}");
}

fn get_number() -> &i32 {
    let number = 13;
    &number
}
```

```
consider using the `'static` lifetime: `&'static ` rustc(E0106)
main.rs(42, 20): original diagnostic
missing lifetime specifier
this function's return type contains a borrowed value, but there is no value for it to be
borrowed from rustc(Click for full compiler diagnostic)
main.rs(42, 20): consider using the `'static` lifetime: `&'static `
View Problem (Alt+F8)   Quick Fix... (Ctrl+.)
fn get_number() -> &i32 {
    let number: i32 = 13;
    &number
}
```

Bisa dilihat, ada error, karena kondisi tersebut menimbulkan dangling reference (sama seperti contoh sebelumnya). Namun pada contoh ini kita tidak bisa menerapkan solusi yang sama, karena variabel `number` mengadopsi *copy semantics*, bukan *move semantics*.

Di Rust error dangling reference diantisipasi saat kompilasi, menjadikan kode tidak bisa sukses dikompilasi.

Solusi yang bisa diambil adalah dengan menjadikan data tersebut tidak

memiliki owner (atau owner-nya adalah program), caranya dengan langsung mengembalikan reference data tanpa perlu menampungnya terlebih dahulu ke variabel, seperti ini:

```
fn main() {
    let n = get_number();
    println!("the number: {n}");
}

fn get_number() -> &i32 {
    &13
}
```

Ok, tapi entah kenapa ketika dilihat masih muncul error.

```
consider using the `'static` lifetime: `&'static ` rustc(E0106)
main.rs(30, 20): original diagnostic

missing lifetime specifier
this function's return type contains a borrowed value, but there is no value for it to be
borrowed from rustc(Click for full compiler diagnostic)
main.rs(30, 20): consider using the `'static` lifetime: `&'static `
View Problem (Alt+F8)   Quick Fix... (Ctrl+.)
fn get_number()  > &i32 {
    &13;
}
```

Error tersebut muncul karena meskipun owner data `&13` adalah program, ketika eksekusi fungsi `get_number` selesai, data borrow tersebut langsung di-dealokasi, dan tidak ada variabel lain di-luar scope yang menampung reference data tersebut.

Agar tidak terjadi proses dealokasi, harus ada variabel yang menampung reference tersebut di-luar scope, tapi cara ini tidak bisa dilakukan karena data-nya saja baru dideklarasikan dalam block fungsi, tidak mungkin tiba-tiba ada

yang menampung di-luar scope.

Solusi dari masalah ini adalah menggunakan lifetime `'static` (yang detailnya sudah dibahas pada chapter sebelumnya). Dengan ini maka data reference `&13` hidup lebih lama dari umur yang sebenarnya sudah ditakdirkan untuk data data tersebut.

```
fn get_number() -> &'static i32 {
    &13
}
```

Namun perlu diingat, bahwa data efek dari lifetime `'static` adalah data tidak akan pernah di-dealokasi. Data tersebut akan hidup selamanya di program hingga kecuali program dimatikan. Karena alasan itulah penggunaan `'static` pada contoh ini bisa disebut berlebihan.

Solusi yang lebih pas adalah dengan membuat lifetime sendiri dengan cara menerapkan **lifetime annotation** (tidak menggunakan lifetime `'static`).

A.42.3. Lifetime annotation dan penerapannya pada return value

Lifetime dituliskan dengan notasi `'nama_lifetime`. Dengan notasi tersebut, kita bisa menciptakan lifetime baru misalnya `'a`, `'b`, `'ini_lifetime`, dst.

```
&i32        // => tipe data reference i32
&'a i32     // => tipe data reference i32 dengan lifetime 'a
&'a mut i32 // => tipe data mutable reference i32 dengan lifetime
'a
```

> *Lifetime dan block label memiliki bentuk literal yang sama, keduanya diawali tanda kutip `'`. Yang membedakan hanya pada tempat dimana syntax tersebut ditulis.*

Kegunaan dari lifetime annotation adalah untuk menginformasikan compiler agar reference tidak langsung didealokasikan setelah eksekusi block selesai. Agar lebih jelas mari kita langsung terapkan saja pada fungsi `get_number` yang sudah ditulis. Silakan tambahkan lifetime dengan nama bebas. Disini penulis gunakan `'my_lifetime`.nnnn

```
fn get_number() -> &'my_lifetime i32 {
    &13
}
```

```
use of undeclared lifetime name `'my_lifetime`
undeclared lifetime rustc(click for full compiler diagnostic)
main.rs(30, 14): consider introducing lifetime `'my_lifetime` here: `<'my_lifetime>`
View Problem (⌥⇧F8)   No quick fixes available
fn get_number() -> &'my_lifetime i32 {
    &13
}
```

Meski sudah ditambahkan, error tetap muncul, karena lifetime tersebut tidak dikenal. Berbeda dengan lifetime `'static` yang memang sudah disediakan oleh Rust.

Step selanjutnya adalah mengenalkan lifetime tersebut, caranya dengan menambahkannya dalam notasi parameter generic fungsi, seperti ini:

```
fn get_number<'my_lifetime>() -> &'my_lifetime i32 {
```

Fungsi `get_number` sekarang tidak menghasilkan error, karena reference yang dikembalikan memiliki umur `'my_lifetime` yang membuatnya tetap valid sampai menjadi return value, meski eksekusi block fungsi-nya sendiri sudah selesai. Efeknya, data yang di-return bisa ditampung di luar scope fungsi `get_number`.

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrograman-rust\dasarpe
   Compiling lifetime v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpem
    Finished dev [unoptimized + debuginfo] target(s) in 0.91s
     Running `target\debug\lifetime.exe`
the number: 13
```

Tanpa adanya lifetime pada return value, maka data return value akan langsung di-dealokasi setelah block fungsi selesai dieksekusi. Tapi karena hal ini *by default* sudah di-handle Rust, maka kita tidak perlu memikirkannya.

A.42.4. Lifetime pada parameter

Pada praktek ini kita akan bahas penerapan lifetime pada parameter.

Silakan perhatikan kode berikut:

```
fn do_something_v1(x: &str) -> &str {
    x
}
```

Kode di atas tidak menghasilkan error, karena kalau dilihat dari sudut pandang lifetime (sesuai dengan aturan lifetime elision), yang sebenarnya terjadi adalah kurang lebih seperti ini:

```
fn do_something_v2<'a>(x: &'a str) -> &'a str {
```

Pada saat parameter `x` masuk ke fungsi, *by default* variabel tersebut memiliki lifetime (yang pada contoh di atas diilustrasikan dengan `'a'). Lifetime tersebut akan aktif hingga menjadi return value karena lifetime yang sama digunakan juga pada return value. Dengan ini nantinya setelah fungsi selesai dieksekusi, nilai baliknya tetap bisa ditampung.

Beda lagi jika lifetime lain digunakan pada nilai balik fungsi, misalnya:

```
fn do_something_v3<'a, 'b>(x: &'a str) -> &'b str {
    x
}
```

```
x: &str
lifetime of reference outlives lifetime of borrowed content... rustc(Click for full
compiler diagnostic)
main.rs(34, 21): ...the reference is valid for the lifetime `'b` as defined here...
main.rs(34, 17): ...but the borrowed content is only valid for the lifetime `'a` as
defined here
View Problem (Alt+F8)    No quick fixes available
```

Error muncul dengan keterangan kurang lebih: data yang dijadikan return value pada block fungsi akan valid untuk lifetime `'b`, namun data yang dijadikan return value hanya valid untuk lifetime `'a`.

Dari sini bisa ditarik kesimpulan: data yang didapat dari luar scope (yang memiliki lifetime sendiri) ketika digunakan sebagai nilai balik, lifetime-nya harus sama.

Sekarang mari kita modifikasi lagi fungsi menjadi seperti ini:

```
fn do_something_v4<'a, 'b, 'c>(x: &'a str, y: &'b str) -> &'c str
```

Fungsi di atas memiliki 2 buah lifetime, yaitu:

- Lifetime `'a`, digunakan pada parameter `x`
- Lifetime `'b`, digunakan pada parameter `y`
- Lifetime `'c`, digunakan pada return value

Karena yang dikembalikan ada data baru, yaitu string `hello`, maka kode di atas tidak error. Data tersebut lifetime-nya adalah `'c`, dan akan tetap valid setelah pemanggilan fungsi selesai.

Beda situasi jika yang dikembalikan adalah data dari parameter, misalnya, `x`. Jika seperti ini, maka lifetime yang sama dengan lifetime parameter `x` harus digunakan, yaitu lifetime `'b`. Contoh:

```
fn do_something_v5<'a, 'b, 'c>(x: &'a str, y: &'b str) -> &'b str
{
    y
}
```

Pada kode di atas, lifetime `'c` tidak dipergunakan sama sekali, maka lifetime tersebut boleh dihapus dari fungsi. Silakan cukup definisikan lifetime yang hanya digunakan saja pada block parameter generic.

```
fn do_something_v6<'a, 'b>(x: &'a str, y: &'b str) -> &'b str {
    y
}
```

Dimisalkan ada seleksi kondisi, jadi return value bisa saja `x`, bisa juga `y`. Jika seperti ini, maka `x`, `y`, dan juga return value harus memiliki lifetime yang sama.

```
fn do_something_v7<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() < y.len() {
        x
    } else {
        y
    }
}
```

A.42.5. Lifetime elision

Sampai section ini kita telah mempelajari kurang lebih 4 point berikut:

1. Setiap data, lebih tepatnya setiap reference memiliki lifetime.
2. Lifetime digunakan oleh Rust dalam penentuan kapan reference tersebut di-dealokasi.
3. Pada beberapa case, lifetime perlu di-urus secara eksplisit (contohnya seperti pada fungsi `do_something_vx` di atas).
4. Pengecekan lifetime terjadi saat kompilasi.

Rust memiliki sesuatu yang disebut dengan **lifetime elision**, isinya adalah aturan yang digunakan oleh Rust dalam menganalisa reference untuk menentukan lifetime *default*-nya.

Namun, bukan berarti Rust akan selalu tau lifetime tiap reference. Pada beberapa case, Rust membutuhkan bantuan kita selaku programmer untuk menginformasikan lifetime reference kode yang ditulis, contohnya seperti pada fungsi `do_something_v7` di atas.

Meskipun demikian, tak usah terlalu khawatir, karena pengecekan lifetime reference terjadi saat kompilasi, dan ketika ada reference yang perlu dikasih *annotication*, Rust akan menginformasikan ke kita via pesan error.

> *Untuk sekarang, pembahasan detail mengenai lifetime elision tidak dibahas pada ebook ini. Silakan gunakan dokumentasi official Rust untuk lifetime elision jika berminat untuk pengkajian yang lebih dalam.*

Jika kawan-kawan menggunakan `rust-analyzer` extension di VSCode, tak perlu meng-compiler program untuk memunculkan error-nya, karena langsung muncul saat penulisan kode program.

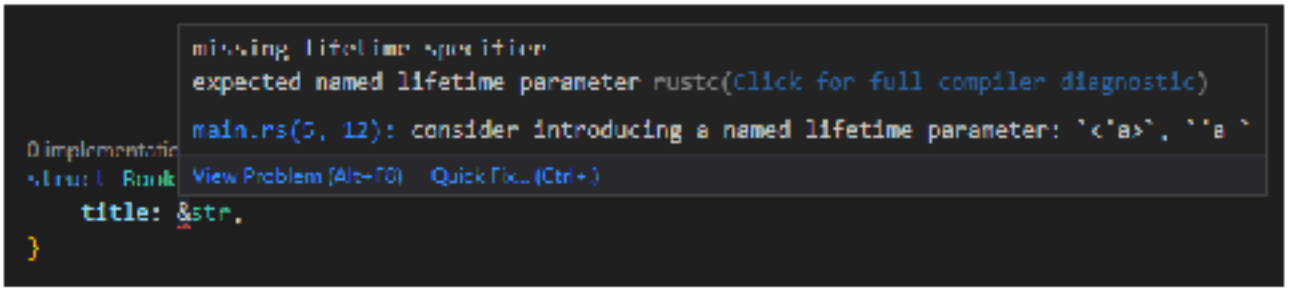
A.42.6. Lifetime pada struct

Tak hanya pada parameter fungsi dan return value fungsi, lifetime juga bisa diterapkan pada (property) struct.

Ketika menggunakan tipe data reference sebagai tipe property struct, Rust langsung menginformasikan kita via error message jika tidak ada anotasi lifetime pada field tersebut.

Contohnya bisa dilihat pada kode di bawah ini. Pesan error muncul karena tipe data `&str` pada property struct tidak ada lifetime annotation-nya.

```
struct Book {
    title: &str,
}
```



Solusinya, tambahkan anotasi lifetime pada tipe data `&str`, dan juga daftarkan lifetime tersebut pada struct. Notasi penulisannya seperti ini:

```
// struct NamaStruct<'lifetime_annotication> {
//     field: &'lifetime_annotication tipe_data,
// }

struct Book<'abc> {
    title: &'abc str,
}
```

Contoh jika ada lebih dari 1 field dengan lifetime sama:

```
struct Book<'abc> {
    title: &'abc str,
    description: &'abc str,
}
```

Contoh jika ada beberapa field yang diantaranya memiliki lifetime berbeda (artinya ada lebih dari 1 lifetime):

```
struct Book<'abc, 'def> {
    title: &'abc str,
    description: &'abc str,
    price: &'def i32,
}

fn main() {
    let book = Book {
        title: "The Silmarillion",
        description: "Good story, 10/10, would read again",
        price: &99,
    };
```

A.42.7. Lifetime pada method

Ada 2 hal yang perlu diketahui dalam penerapan lifetime pada method. Yang pertama, lifetime annotication harus ditulis pada block `impl` meskipun pada block method tidak digunakan secara langsung.

```
struct Book<'abc, 'def> {
    title: &'abc str,
    description: &'abc str,
    price: &'def i32,
}

impl<'abc, 'def> Book<'abc, 'def> {

    fn get_info(&self) -> String {
        let info = format!("{} (${}), {}", self.title,
self.price, self.description);
        info
    }
}
```

Bisa dilihat statement block `impl<'abc, 'def> Book<'abc, 'def>`, lifetime annotation-nya sama persis dengan yang ada di struct. Jika ingin mengetes, silakan coba saja hapus syntax lifetime dari statement itu, pasti muncul error.

Hal ke-2 yang penting diketahui, pada block kode method, tidak perlu menuliskan lifetime annotation, karena sudah ditulis di block kode `impl`.

Sebagai contoh, dua method berikut tidak memunculkan error:

```
// ...
```

Bandingkan dengan fungsi `get_book_price` berikut, error muncul karena lifetime annotation tidak ditambahkan ke block fungsi (meskipun fungsi tersebut mengembalikan property `price` milik struct `Book` yang sudah memiliki lifetime sendiri). Hal ini karena data `&i32` yang dikembalikan statement `book.get_price()` langsung didealokasi setelah block fungsi `get_book_price` selesai dieksekusi.

```
missing lifetime specifier
this function's return type contains a borrowed value, but the signature does not say which one of
`book`'s 3 lifetimes it is borrowed from rustc(Click for full compiler diagnostic)
main.rs(32, 25):
main.rs(32, 18): consider introducing a named lifetime parameter: `<'a>`, `'a `, `'a `, `<'a, 'a>`
View Problem (Alt+F8)   Quick Fix... (Ctrl+.)
fn get_book_price(book: &Book)  -> &i32 {
    book.get_price()
}
```

Agar error tidak muncul, tambahkan lifetime:

1. ke return value (karena tipe data return value adalah reference)
2. lalu pada parameter `book` (karena di block fungsi return value berasal dari data milik property variabel `book`),
3. dan juga tak lupa daftarkan lifetime annotation pada block fungsi `get_book_price`.

Kurang kode menjadi seperti berikut:

```rust
// ...

fn get_book_price<'ghi>(book: &'ghi Book) -> &'ghi i32 {
    book.get_price()
}

fn main() {
```

Run program, hasilnya sukses.

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrograman-rust\dasarpe
   Compiling lifetime_3 v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarp
    Finished dev [unoptimized + debuginfo] target(s) in 0.41s
     Running `target\debug\lifetime_3.exe`
The Silmarillion ($99), Good story, 10/10, would read again
the price: 99
```

A.42.8. Generic parameter + trait bounds + lifetime

Lalu bagaimana jika ada fungsi yang disitu ada penerapan trait bounds, ada juga generic parameter, dan lifetime annotation. Cara penulisannya seperti apa? Silakan lihat contoh berikut:

```rust
fn find_greater_number<'a, T>(
    x: &'a T,
    y: &'a T,
) -> &'a T
where
    T: std::cmp::PartialOrd,
{
    if x > y {
        x
    } else {
        y
    }
}
```

Fungsi `find_greater_number` di atas digunakan untuk mencari angka terbesar dari dua inputan parameter (`x` dan `y`), dengan tipe data yang digunakan adalah generic `T`. Dari sini maka nantinya fungsi ini bisa digunakan pada data

bertipe `i32`, `f64`, dan data numerik lainnya.

Tipe `T` diasosiasikan dengan trait `std::cmp::PartialOrd` agar variabel dengan tipe tersebut bisa digunakan dalam seleksi kondisi `if` yang ada dalam block fungsi tersebut.

Ok, sampai sini semoga cukup jelas. Lalu bagaimana dengan lifetime annotation-nya? Karena lifetime annotation definisinya berada pada tempat yang sama dengan definisi tipe generic, maka langsung saja tulis disitu tanpa memperhatikan urutan. Sebagai contoh, dua definisi block fungsi berikut adalah ekuivalen.

```
fn say_hello_v1<'abc, T>()
fn say_hello_v2<T, 'abc>()
```

Sekarang kembali ke pembahasan di atas. Fungsi `find_greater_number` mengembalikan tipe data reference `&T` yang nilainya bisa saja dari `x` atau `y`. Dari sini maka wajib hukumnya untuk return value memiliki lifetime yang sama dengan `x` dan `y`. Itulah kenapa definisi fungsi `find_greater_number` agak panjang.

```
fn find_greater_number<'a, T>(x: &'a T, y: &'a T) -> &'a T
```

Lanjut, mari kita panggil fungsi tersebut pada dua block expression. Block pertama untuk pengecekan data numerik `i32`, dan yang kedua untuk tipe `f64`.

```
fn main() {

    {
```

Jalankan program, hasilnya sesuai harapan, tidak ada error.

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarpe
   Compiling lifetime_3 v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarp
    Finished dev [unoptimized + debuginfo] target(s) in 0.41s
     Running `target\debug\lifetime_3.exe`

The Silmarillion ($99), Good story, 10/10, would read again
the price: 99
```

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../lifetime
```

◉ Referensi

- https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html
- https://doc.rust-lang.org/nomicon/lifetimes.html
- https://dev.to/takaakifuruse/rust-lifetimes-a-high-wall-for-rust-newbies-3ap
- https://anooppoommen.medium.com/lifetimes-in-rust-7f2331be998b
- https://blog.logrocket.com/understanding-lifetimes-in-rust/
- https://educative.io/answers/what-are-generic-lifetime-parameters-in-a-rust-function

---

A.43. Slice Memory Management

Kita telah mempelajari tipe data Array dan Vector, serta sudah beberapa kali menggunakan tipe data string slice (`string`). 3 tipe data itu memiliki kemiripan, yaitu kesemuanya termasuk dalam kategori tipe data slice.

> *Ciri khas dari tipe data yang termasuk dalam kategori slice adalah jika diakses reference-nya menghasilkan data bertipe `&[T]` dimana `T` adalah tipe data elemen.*

Pada chapter ini, kita akan bahas lebih dalam lagi tentang apa itu slice terutama bagian memory management-nya.

> *Perbedaan chapter ini dengan chapter ini dengan chapter Slice (Basic) adalah disini fokusnya lebih banyak di memory management.*

A.43.1. Konsep slice

Slice adalah representasi *block of memory* berbentuk pointer dan memiliki size yang dinamis, dengan isi adalah koleksi element data. Slice merupakan reference atau data pinjaman (borrow).

Pada program berikut, beberapa variabel dideklarasikan menggunakan tiga tipe data di atas, kemudian masing-masing data dipinjam kemudian di-print.

```rust
let data_arr = [1, 2, 3];
println!("data_arr: {} {:?}", data_arr.len(), data_arr);
let slice1 = &data_arr[1..];
println!("slice1  : {} {:?}", slice1.len(), slice1);
let slice2 = &data_arr[..2];
println!("slice2  : {} {:?}", slice2.len(), slice2);

let data_vec = vec![1, 2, 3];
println!("data_vec: {} {:?}", data_vec.len(), data_vec);
let slice1 = &data_vec[1..];
println!("slice1  : {} {:?}", slice1.len(), slice1);
let slice2 = &data_vec[..2];
println!("slice2  : {} {:?}", slice2.len(), slice2);

let data_str = String::from("sesuk prei jarene, mosokk");
println!("data_str: {} {:?}", data_str.len(), data_str);
let slice1 = &data_str[1..];
println!("slice1  : {} {:?}", slice1.len(), slice1);
let slice2 = &data_str[..2];
println!("slice2  : {} {:?}", slice2.len(), slice2);
```

```
data_arr: 3 [1, 2, 3]
slice1  : 2 [2, 3]
slice2  : 2 [1, 2]

data_vec: 3 [1, 2, 3]
slice1  : 2 [2, 3]
slice2  : 2 [1, 2]

data_str: 25 "sesuk prei jarene, mosokk"
slice1  : 24 "esuk prei jarene, mosokk"
slice2  : 2 "se"
```

Terlihat kemiripan-nya, slice bisa terbentuk dari ketiga jenis data di atas.

Penulis ingatkan lagi, bahwa slice adalah tipe data reference yang berarti isi adalah data pinjaman (borrow). Tipe data slice selalu `&[T]` dimana `T` adalah

tipe data element.

Karena slice adalah data borrow, maka operasi standar borrowing termasuk mutable borrowing bisa dilakukan di slice.

A.43.2. Memory management pada slice

Sekarang lanjut ke pembahasan tentang bagaimana data bertipe slice di manage di-memory. Sebagai bahan belajar, kita perlu memilih satu dari 3 tipe data slice yang ada. Bebas sebenarnya mau pilih yang mana. Penulis memilih `string` untuk memulai pembahasan.

Silakan perhatikan statement sederhana berikut:

```
let data_str = String::from("sesuk preiii");
```

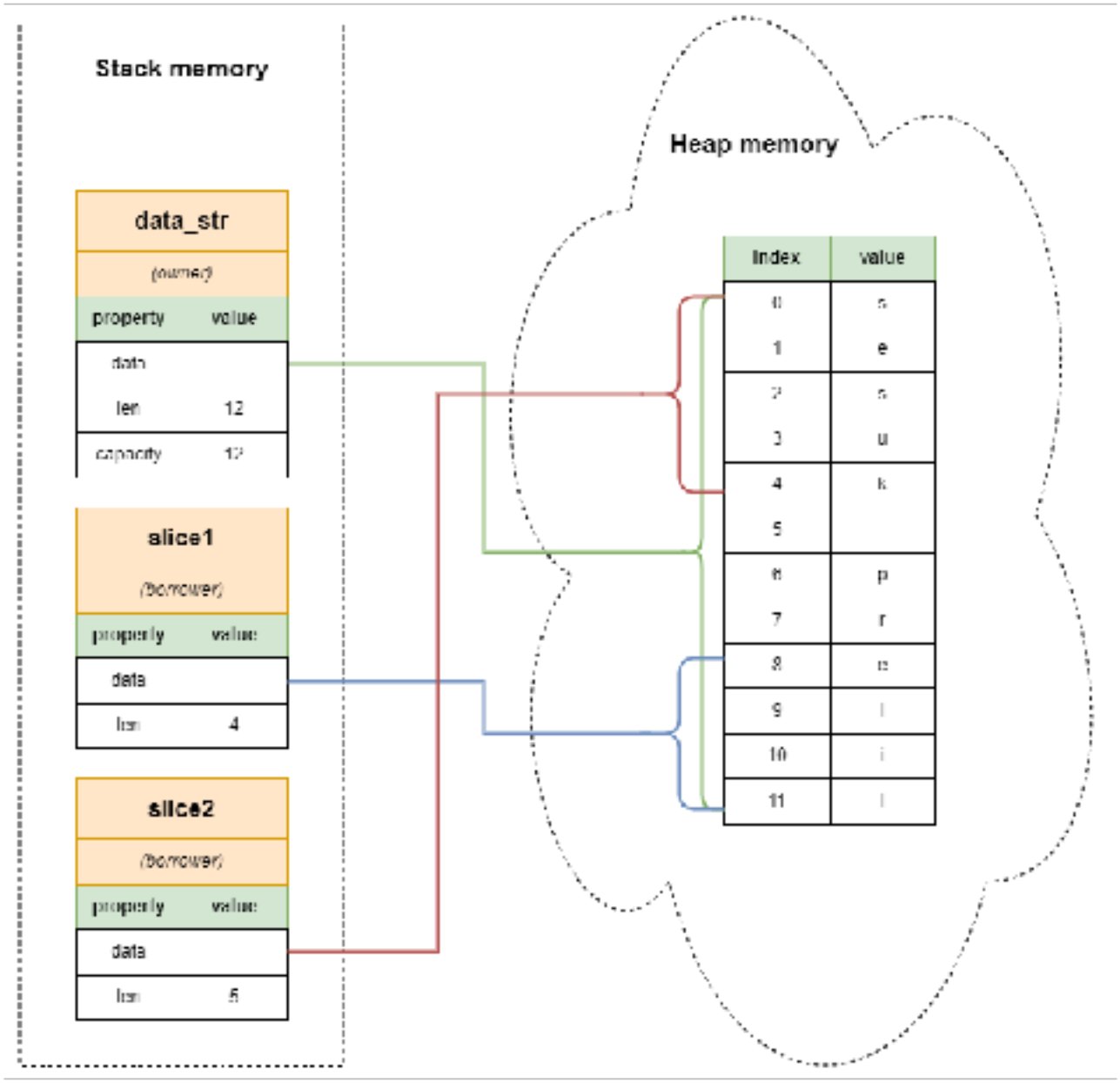
Ada sebuah data `string` dideklarasikan, value-nya adalah `sesuk preiii`, dan owner-nya adalah variabel `data_str`. Data variabel `data_str` disimpan di memory pada 2 tempat, yaitu **heap** dan **stack**.

Selanjutnya, 2 buah slice tercipta hasil operasi slicing pada variabel `data_str`.

```
let data_str = String::from("sesuk preiii"); // "sesuk preiii"
let slice1 = &data_str[8..];                 // "eiii"
let slice2 = &data_str[..5];                 // "sesuk"
```

Slice adalah data borrow, artinya jika ada beberapa variabel baru dibuat hasil dari operasi slicing, maka isi variabel tersebut merupakan reference yang mengarah ke data sebenarnya. Di heap memory tidak ada perubahan, namun

di stack memory ada beberapa data baru. Lebih jelasnya silakan perhatikan ilustrasi berikut.



Sekarang di stack memory ada 3 buah metadata informasi disimpan, yaitu `data_str` (yang merupakan owner sebenarnya data), dan `slice1` & `slice2`. Sedangkan untuk data-nya sendiri tetap berada di heap memory tanpa ada perubahan.

Tiga variabel di atas kesemuanya mengakses reference yang sama, yang membedakan adalah elemen-nya saja. Owner (yaitu `data_str`) bisa mengakses seluruh data, selain itu juga tau informasi kapasitas data. Sedangkan borrower hanya bisa mengakses data yang dia pinjam sesuai dengan operasi slicing-nya. Borrower tidak mengetahui kapasitas data, namun ia tau size dari elemen yang ia pinjam.

A.43.2. Mutable slice

Bagaimana dengan *mutability* pada slice, apa yang terjadi di belakang layar ketika elemen slice nilainya diubah?

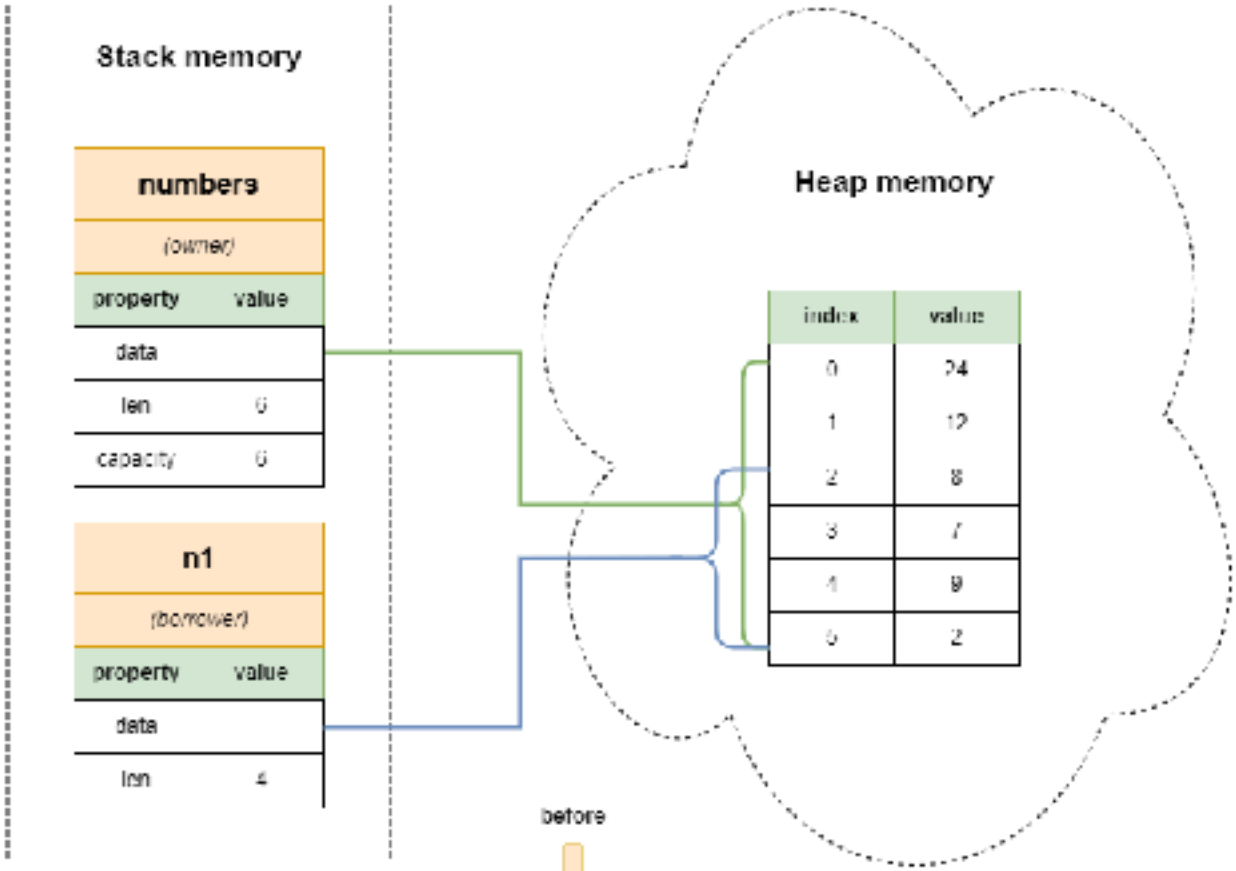
Ok, mari kita bahas dengan contoh. Pada kode berikut ada sebuah data mutable array bertipe `[i32; 6]` dengan owner bernama `numbers`. Dari variabel tersebut, dilakukan operasi mutable borrowing untuk disimpan pada variabel `n1`.

```
let mut numbers = [24, 12, 8, 7, 9, 2];
println!("numbers: {:?}", numbers);
let n1 = &mut numbers[2..];
println!("n1      : {:?}", n1);
n1[0] = 3022;
println!("numbers: {:?}", numbers);
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrograman
   Compiling slice_memory_management v0.1.0 (D:\Labs\Ad
    Finished dev [unoptimized + debuginfo] target(s) in
     Running `target\debug\slice_memory_management.exe`

numbers: [24, 12, 8, 7, 9, 2]
n1      : [8, 7, 9, 2]
numbers: [24, 12, 3022, 7, 9, 2]
```

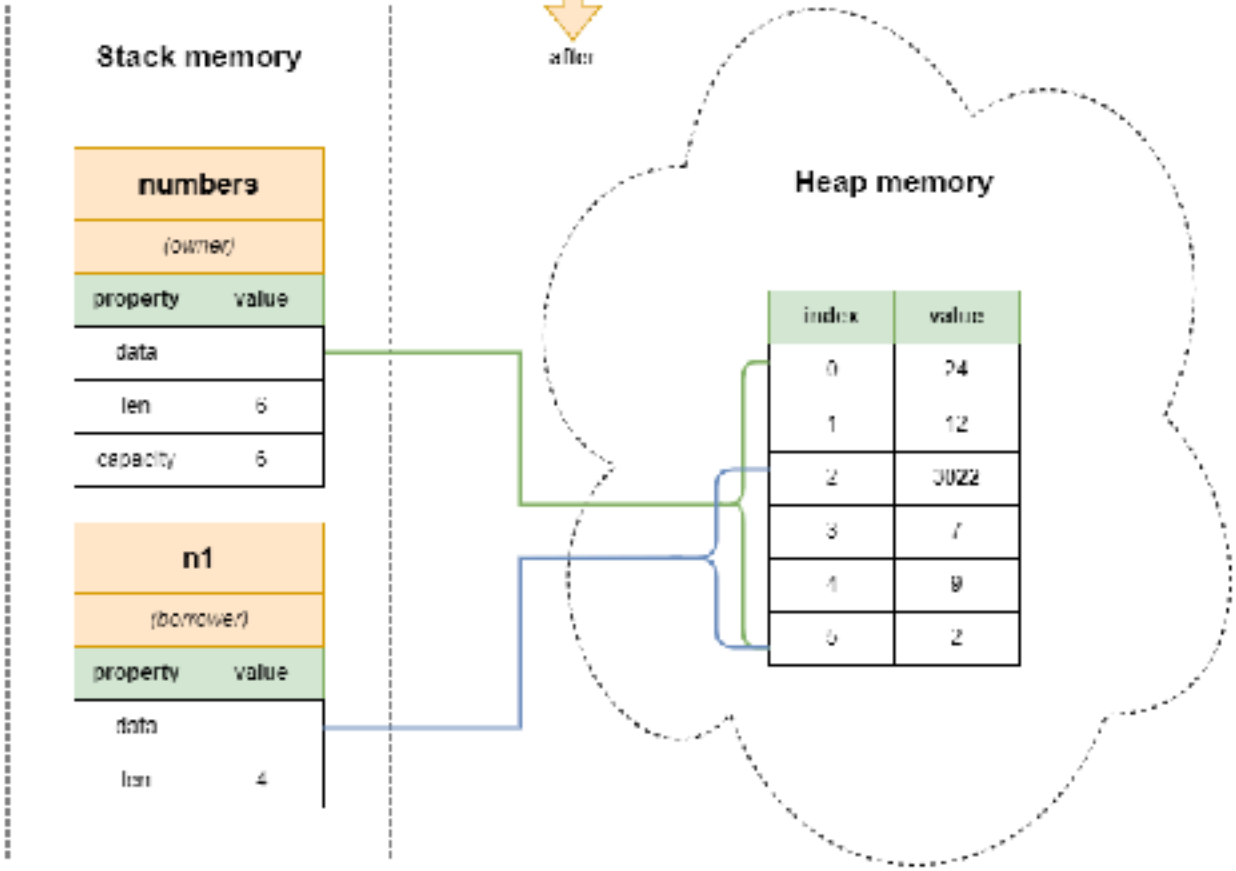
Salah satu elemen `n1` diubah nilainya, maka efeknya juga berpengaruh pada owner. Ketika variabel `numbers` di-print, element indeks `2` berubah nilainya dari yang sebelumnya `8` sekarang `3022`.

Visualisasi memory management-nya kurang lebih seperti ini:



before

after



Variabel `numbers` dan `n1` sama-sama mengarah ke reference yang sama. Itulah kenapa perubahan data pada peminjam membawa efek ke owner.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-
example/../slice_memory_management
```

◉ Chapter relevan lainnya

- Array
- Slice (Basic)
- Vector
- Basic Memory Management
- Tipe Data → String Custom Type
- String Literal (&str) vs. String Custom Type

◉ Referensi

- https://doc.rust-lang.org/book/ch04-03-slices.html
- https://users.rust-lang.org/t/why-rust-slice-has-not-ownership/27356

A.44. String Literal (&str) vs. String Custom Type

Pada chapter sebelumnya kita telah membahas tentang bagaimana data slice di-manage di memory. Ada beberapa tipe data yang masuk dalam kategori slice, yang salah satunya adalah string slice atau `string`.

Di chapter ini kita akan bahas apa perbedaan antara tipe data string slice (`string`) dan string literal `&str`.

Pembahasan mengenai topik ini sengaja dilakukan tidak di awal-awal ebook, karena ada banyak hal yang perlu dipahami sebelum mempelajarinya, contohnya seperti aspek management memory dan ownership. Dan karena topik tersebut sudah selesai dibahas, berarti ini adalah waktu yang tepat untuk membahas string slice.

> *Silakan pelajari kembali pembahasan detail tentang tipe slice pada chapter sebelumnya jika diperlukan. Chapter Slice Memory Management*

A.44.1. String slice (`String`)

String slice atau custom type `string` merupakan tipe data bawaan Rust, dibuat via `struct`, kegunaannya untuk menampung data UTF-8 bytes yang dinamis (bisa berkembang isinya).

`String` masuk dalam kategori tipe data slice, isinya adalah data kolektif bertipe bytes, datanya disimpan di heap memory, dan metadata-nya di stack memory. Tipe data ini dikategorikan sebagai tipe data **owned**, yang artinya owner data bisa direpresentasikan oleh variabel. Sebagai contoh:

```
let str1 = String::from("Lisa Blackpink");
println!("str1: {str1}");
```

Variabel `str1` di atas merupakan owner dari string `Lisa Blackpink`. Dari string tersebut operasi borrow bisa dilakukan. Contohnya bisa dilihat pada kode berikut:

```
let str1 = String::from("Lisa Blackpink");
let slice1 = &str1[..];   // "Lisa Blackpink"
let slice2 = &str1[4..7]; // " Bl"
```

`slice1` adalah variabel baru yang datanya didapat dari borrowing seluruh elemen string milik `str1`. Sedangkan variabel `slice2` hanya meminjam elemen indeks ke-4 hingga elemen sebelum 7 (yaitu 6).

Karena `String` sebenarnya adalah UTF-8 bytes, maka kita bisa juga membuatnya menggunakan data Bytes. Tipe bytes (atau kadang disebut *chars*) di Rust direpresentasikan oleh tipe `[u8]`.

```
let bytes = vec![69, 108, 117, 118, 101, 105, 116, 105, 101, 32,
243, 159, 164, 152];
let str2 = String::from_utf8(bytes).unwrap();
println!("str2: {}", str2);
```

Pada contoh di atas, data bytes dipersiapkan dalam bentuk `Vec<u8>`. Data tersebut kemudian digunakan untuk membuat string menggunakan fungsi

`String::from_utf8()`. Nilai balik fungsi tersebut adalah `Result<String, FromUtf8Error>`. Pemanggilan method `unwrap` disitu agar data `String`-nya di-return.

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrograman-rust\dasar
   Compiling string_slice_vs_string_literal v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in 0.44s
     Running `target\debug\string_slice_vs_string_literal.exe`
str2: Eluveitie 🤘
```

> *Kita akan bahas tipe data `Result` pada chapter terpisah, Tipe Data → Result*

A.44.2. String literal (`&str`)

Tipe data string literal atau `&str` adalah tipe yang menampung data kolektif UTF-8 bytes (seperti `String`) tetapi **immutable** dan disimpannya tidak di heap dan tidak juga di stack, melainkan di static storage.

String literal hanya bisa direpresentasikan dalam bentuk reference `&str` (pointer yang mengarah ke suatu bytes).

Tipe `&str` termasuk kategori tipe data yang **unowned** atau reference tanpa owner (atau boleh juga diartikan sebagai tipe data yang owner-nya adalah program).

Rust menjamin data string literal selalu valid. Kita juga bisa menentukan lifetime-nya secara eksplist jike diperlukan, contohnya pada tipe `&'static str`.

Cara termudah membuat `&str` adalah menggunakan string literal.

```
let str3 = "Helena Iren Michaelsen Epica";
println!("str3: {str3}");
```

A.44.3. Konversi data string

◉ Konversi `String` ke `&str`

Data bertipe `&str` juga didapat melalui operasi borrow dari data bertipe `String`, caranya dengan menggunakan method `as_str`.

```
let str4: String = String::from("Hiroyuki Sawano");
let str4_slice1: &str = str4.as_str();
println!("str4: {str4}");                // str4: Hiroyuki Sawano
println!("str4_slice1: {str4_slice1}"); // str4_slice1: Hiroyuki
Sawano
```

Bisa juga menggunakan method `as_mut_str` untuk mutable borrow. Namun dalam penggunaannya, owner data diwajibkan mutable. Contohnya:

```
let mut str5: String = String::from("Hans Zimmer");
let str5_slice1: &mut str = str5.as_mut_str();
println!("str5: {str5}");
println!("str5_slice1: {str5_slice1}");
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarpemrogramanrust\examples\str
   Compiling string_slice_vs_string_literal v0.1.0 (D:\Labs\Adam Studio\Ebook\dasarpemrog
error[E0502]: cannot borrow `str5` as immutable because it is also borrowed as mutable
  --> src\main.rs:28:26
   |
27 |         let str5_slice1 = str5.as_mut_str();
   |                                 ------------ mutable borrow occurs here
28 |         println!("str5: {str5}");
   |                          ^^^^^ immutable borrow occurs here
29 |         println!("str5_slice1: {str5_slice1}");
   |                                 ----------- mutable borrow later used here
   |
```

Hmm, error. Perlu diingat kembali aturan Borrowing bahwa tidak boleh ada operasi borrow dan juga mutable borrow dalam satu waktu. Pada contoh di atas, `str5` dipinjam oleh `str5_slice1` menggunakan method `as_mut_str` (yang didalamnya menjalankan operasi mutable borrow). Selain itu, operasi mutable borrow maksimal hanya boleh dilakukan 1x dalam satu waktu, inilah kenapa ketika kita berusaha menampilkan data `str5` hasilnya error, karena statement print data `str5` adalah operasi borrow.

Solusinya masalah di atas bisa menggunakan block expression:

```rust
let mut str5: String = String::from("Hans Zimmer");
{
    let str5_slice1: &mut str = str5.as_mut_str();
    println!("str5_slice1: {str5_slice1}");  // str5_slice1: Hans
Zimmer
}
println!("str5: {}", str5);                  // str5_slice1: Hans
Zimmer
```

Tipe data `String` bisa diakses mutable reference-nya karena memang tipe tersebut size-nya adalah dinamis.

◉ Konversi `&str` ke `String`

Masih dalam topik konversi tipe data string. Tipe `&str` memiliki method bernama `to_string` yang gunanya adalah untuk konversi data `&str` ke `String`.

```
let str6: &str = "John Towner Williams";
let str6_slice1: String = str6.to_string();
println!("str6: {str6}");                // str6: John Towner
Williams
println!("str6_slice1: {str6_slice1}"); // str6_slice1: John
Towner Williams
```

Konversi pada tipe data ini sedikit berbeda dibandingkan konversi `String` ke `&str`. Pada contoh di atas, yang terjadi adalah data `&str` di-copy sebagai data baru bertipe `String` yang kemudian ditampung variabel `str6_slice1` (yang juga berperan sebagai owner untuk data baru tersebut).

Method `to_string` melakukan operasi copy, bukan borrow. Artinya setelah dipanggil akan ada 2 data yang reference-nya sudah berbeda.

A.44.4. String literal & string slice

Tipe `String` memiliki hubungan dekat dengan `&str`. Data bertipe `String` reference-nya bisa diakses dalam bentuk `&String`, maupun dalam bentuk `&str` (menggunakan method `as_str` atau `as_mut_str`). Data text pada string tersebut bisa dimodifikasi, ditambahi, dan juga dikurangi.

Berbeda dengan data bertipe `&str` **(di paragraph ini dan setelahnya yang kita bahas adalah data yang dari awal tipe-nya sudah `&str`, bukan**

data hasil operasi pinjam dari `string`), data bertipe `&str` adalah fixed dan immutable. Konversi data `&str` ke `string` menghasilkan data baru dengan owner baru. Bisa dibilang sangat terbatas apa yang bisa kita lakukan pada tipe data `&str`.

Meski demikian, tipe `&str` lebih cepat performa-nya dibanding `string` karena disimpan di static storage. Selain itu dijamin valid oleh Rust. Kekurangannya hanya pada ownership-nya. Tipe `&str` adalah **unowned**, operasi mutability tidak bisa dilakukan pada tipe ini.

Dalam case normal, sangat dianjurkan untuk menggunakan `&str`, kecuali memang yang dibutuhkan adalah **owned** string.

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-
example/../string_slice_vs_string_literal
```

◉ Chapter relevan lainnya

- Tipe Data → String Literal (&str)
- Slice Memory Management
- Tipe Data → String Custom Type

◉ Referensi

- https://doc.rust-lang.org/book/ch08-02-strings.html
- https://users.rust-lang.org/t/str-string-literals/29635
- https://www.quora.com/Why-does-Rust-have-two-different-string-types-static-str-and-String
- https://stackoverflow.com/questions/24158114/what-are-the-differences-between-rusts-string-and-str
- https://stackoverflow.com/questions/30154541/how-do-i-concatenate-strings

---

A.45. Tipe Data → String Custom Type

Pembahasan kali ini lingkupnya masih dalam topik custom type `string`.

Penulis rasa 2 chapter terakhir sudah cukup banyak menjelaskan perihal apa itu tipe data `string`, apa perbedaannya dibanding `&str`, dan kapan harus menggunakannya.

> - *Pembahasan mengenai perbedaan `string` vs `&str` bisa dilihat pada chapter String Literal (&str) vs. String Custom Type*
> - *Pembahasan tentang bagaimana data pada string slice di-manage di memory bisa dilihat pada chapter Slice Memory Management*

Chapter ini akan lebih fokus ke fitur yang tersedia pada tipe `string`.

A.45.1. Pembuatan string slice

Ada banyak cara yang bisa digunakan dalam membuat data `string`, diantaranya:

◉ Method `to_string` milik `&str`

Method `to_string` milik `&str` mengkonversi string literal menjadi `string`.

```rust
let str2 = "iPhone 8".to_string();
println!("{str2}"); // iPhone 8
```

◉ `String::from`

Associated function `string::from` digunakan untuk mengkonversi `&str` ke bentuk `String`. Kita sudah cukup sering menggunakan fungsi ini.

```rust
let str1 = String::from("Nokia 3310");
println!("{str1}"); // Nokia 3310
```

◉ `String::new`

Associated function `string::new` menghasilkan data string kosong.

```rust
let str3 = String::new();
println!("{str3}"); // ""
```

◉ `String::from_utf8`

Berguna untuk mengkonversi data bytes ke tipe `string`. Fungsi ini nilai baliknya bertipe `Result<String, FromUtf8Error>`, maka dalam penerapannya harus di-chain dengan method `unwrap` untuk mendapatkan data `string`-nya.

```rust
let str4 = String::from_utf8(vec![78, 55, 51]).unwrap();
println!("{str4}"); // N73
```

> *Lebih jelasnya mengenai tipe `Result` dibahas pada chapter Tipe Data → Result*

A.45.2. String mutability

◉ Keyword `mut`

Keyword `mut` bisa digunakan untuk mengganti/replace data string dengan data baru. Sebagai contoh, `str5` berikut yang awalnya adalah string kosong di-replace dengan `Pixel 5`.

```
let mut str5 = String::new();
println!("{str5}");
// ""

str5 = String::from("Pixel 5");
println!("{str5}");
// Pixel 5
```

◉ Replace string / method `replace`

Method `replace` digunakan untuk mengganti suatu substring dengan string lain. Method ini menghasilkan object `String` baru dan tidak mengubah data string aslinya. Karena alasan itu juga kenapa tidak perlu menggunakan keyword `mut` dalam penerapannya.

```
let str9 = String::from("my phone is Pixel 6");
let str10 = str9.replace("Pixel 6", "Nokia 3310");
```

◉ Prepend string / method `insert_str`

Method `insert_str` digunakan untuk menyisipkan substring pada posisi tertentu.

```
let mut str6 = String::from("Pixel 6");
println!("{str6}");
// Pixel 6

str6.insert_str(0, "my phone");
println!("{str6}");
// my phonePixel 6

str6.insert_str(8, " is ");
println!("{str6}");
// my phone is Pixel 6
```

Parameter pertama perlu diisi dengan indeks dimana string akan disisipkan.

- String `Pixel 6` pada indeks 0 disisipi string `my phone`, hasilnya `my phonePixel 6`
- String `my phonePixel 6` pada indeks 8 disisipi string `is`, hasilnya `my phone is Pixel 6`

◉ Prepend char / method `insert`

Method `insert` kegunaannya sama seperti `insert_str` yaitu untuk menyisipkan string pada posisi tertentu. Perbedaannya, pada method `insert` parameter kedua diisi dengan char.

```rust
let mut str7 = String::from("3310");

str7.insert(0, 'N'); // N3310
str7.insert(1, 'o'); // No3310
str7.insert(2, 'k'); // Nok3310
str7.insert(3, 'i'); // Noki3310
str7.insert(4, 'a'); // Nokia3310
str7.insert(5, ' '); // Nokia 3310
```

◉ Append string / method `push_str`

Method `push_str` digunakan untuk menambahkan string di akhir.

```rust
let mut str8 = String::from("Pixel 6"); // Pixel 6
str8.push_str(" is a good phone");      // Pixel 6 is a good phone
```

◉ Append char / method `push`

Method `push` sama kegunaannya seperti `push_str`, namun untuk penambahan data `char`.

```rust
let mut str8 = String::from("Pixel");

str8.push(' '); // "Pixel "
str8.push('7'); // "Pixel 7"
```

◉ Clear string / method `clear`

Method `clear` digunakan untuk mengosongkan data string.

```
let mut str11 = String::from("Nokia 3310");
str11.clear();
println!("{str11}"); // ""
```

A.45.3. Operasi string lainnya

◉ Cek substring / method `contains`

Method `contains` digunakan untuk mengecek apakah suatu substring yang dicari ada atau tidak. Method ini mengembalikan nilai bertipe `bool`.

```
let str11 = String::from("Nokia 3310");

let is_exists = str11.contains("3310");
println!("{is_exists}"); // true

let is_exists = str11.contains("3315");
println!("{is_exists}"); // false
```

◉ Concat strings / slice `join`

Operasi concat string bisa dilakukan memanfaatkan method `insert_str` atau `push_str`. Selain itu juga bisa dengan menggunakan method `.join` milik slice. Caranya, jadikan string yang ingin di-gabung sebagai element array, kemudian akses method `.join`.

```
let str12 = String::from("iPhone");
let str13 = String::from("12");
let str14 = String::from("Pro");
```

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-
example/../tipe_data_custom_type_string_slice
```

◉ Chapter relevan lainnya

- Tipe Data → String Literal (&str)
- Static Item
- Lifetime
- Slice Memory Management
- String Literal (&str) vs. String Custom Type

◉ Referensi

- https://doc.rust-lang.org/std/string/struct.String.html

A.46. Closures

Chapter ini membahas tentang closures. Closures sendiri merupakan block fungsi anonimus (anonymous function) yang memiliki kelebihan bisa mengakses item-tem yang posisinya berada di luar block closure tersebut tetapi masih dalam current block scope.

Topik closures sangat erat hubungannya dengan Trait → Function, namun pada chapter ini pembahasan hanya difokuskan pada bagian penerapan closures saja. Penulis anjurkan untuk lanjut ke chapter berikutnya setelah selesai dengan chapter ini.

A.46.1. Konsep Closures

Cara penerapan closure sangat mirip seperti fungsi, perbedaannya ada pada notasi penulisan-nya. Agar lebih jelas silakan perhatikan program sederhana di bawah ini.

```
fn main() {
    let r = 10.0;
    let volume = calculate_circle_volume_v1(r);
    println!("{volume:.2}");
}

fn calculate_circle_volume_v1(e: f64) -> f64 {
    const PI: f64 = 3.14;
    let volume = 4.0 / 3.0 * PI * e.powi(3);
    volume
}
```

Fungsi `calculate_circle_volume_v1` akan kita refactor ke bentuk closure, hasilnya adalah berikut:

```
fn main() {
    let calculate_circle_volume_v2 = |e: f64| -> f64 {
        const PI: f64 = 3.14;
        let volume = 4.0 / 3.0 * PI * e.powi(3);
        volume
    };

    let r = 10.0;
    let volume = calculate_circle_volume_v2(r);
    println!("{:.2}", volume);
}
```

Silakan pelajari perbedaan dan juga kemiripannya.

Fungsi dideklarasikan menggunakan keyword `fn` dan memiliki nama. Closure tidak memiliki nama, namun bisa disimpan dalam variabel (yang disimpan adalah block closure-nya, bukan return value-nya). Contohnya variabel `calculate_circle_volume_v2` di atas.

Perbedaan minor lainnya ada pada notasi penulisan parameter. Pada fungsi tanda `()` digunakan sebagai penanda parameter, sedangkan pada closure tanda `||` digunakan.

◉ formatted print `{:.n}`

Notasi penulisan formatted print `{:.n}` digunakan untuk mem-format bilangan desimal dimana `n` adalah jumlah digit setelah tanda `.`.

Sebagai contoh, variabel `pi` berikut memiliki 0 digit angka dibelakang koma. Untuk menampilkan hanya 4 angka terdepan, bisa gunakan `{:.4}`. Perlu

diketahui bahwa angka dibelakang koma yang muncul otomatis dibulatkan.

```
let pi = 3.1415926535;

println!("{:.4}", pi); // 3.1416
println!("{pi:.4}");   // 3.1416
```

A.46.2. Notasi penulisan closure

Closure `calculate_circle_volume_v2` pada contoh di atas adalah salah satu contoh penulisan closure. Sebuah closure bisa memiliki parameter, bisa juga tidak, dan aturan tersebut juga berlaku pada return value.

Bebebrapa contoh lain penulisan closure bisa dilihat di bawah ini:

```
// closure dengan 2 parameter tanpa return value
let do_something_v1 = | a: i32, b: String | {
    // ...
};

// closure dengan 2 parameter dan return value bertipe tuple
let do_something_v2 = | a: i32, b: String | -> (i32, bool) {
    // ...
};

// closure tanpa parameter dan return value bertipe Vec<String>
let do_something_v3 = || -> Vec<String> {
    // ...
};

// closure tanpa parameter dan tanpa return value
let do_something_v4 = || {
    // ...
```

Jika tipe return value tidak dideklarasikan secara eksplist, maka Rust menganggap tipe return value adalah sesuai dengan tipe data pada statement terakhir.

Untuk closure yang isinya hanya 1 baris statement, boleh tidak dituliskan block kurung kurawal-nya (`{}`).

Tambahan contoh, satu fungsi dan tiga buah closures berikut adalah ekuivalen.

```
fn pow_v1(x: i32) -> i32 {
    x.pow(2)
}

let pow_v2 = |x: i32| -> i32 {
    x.pow(2)
};

let pow_v3 = |x: i32| {
    x.pow(2)
};

let pow_v4 = |x: i32| x.pow(2);
```

Ok, sekarang bagaimana dengan closure yang tidak memiliki parameter dengan isi hanya return value saja? Penulisannya seperti ini:

```
let get_pi = || 3.14;

println!("{:?}", get_pi());
```

A.46.3. Mutable closure

Tidak ada yang spesial mengenai cara mengakses item yang berada di luar block closure. Caranya cukup dengan panggil saja item seperti biasanya.

```
let num = 5;
let display = || println!("{num}");

println!("{num}");  // 5
display();          // 5
```

Beda lagi jika data di luar block closure adalah diubah (di-mutate) nilainya dari dalam closure, jika seperti itu maka ada beberapa hal yang perlu diperhatikan.

Sebagai contoh, pelajari kode berikut.

```
let mut num = 5;

let increase_by = |x: i32| {
    num += x
};

increase_by(10);
println!("{num}"); // 15
```

```
let increase_by: |i32| -> ()
cannot borrow `increase_by` as mutable, as it is not declared as mutable
cannot borrow as mutable rustc(Click for full compiler diagnostic)
main.rs(14, 13): calling `increase_by` requires mutable binding due to mutable borrow of `num`
main.rs(13, 13): consider changing this to be mutable: `mut increase_by`
View Problem (Alt+F8)   No quick fixes available
```

Solusi untuk menghilangkan error di atas adalah dengan menambahkan keyword `mut` pada variabel closure `increase_by`.

```
// before
let increase_by = |x: i32| {
    num += x
};

// after
let mut increase_by = |x: i32| {
    num += x
};
```

Keyword `mut` wajib ditambahkan ke variabel penampung closure ketika di dalamnya terdapat operasi perubahan data terhadap variabel yang posisinya di-luar closure. Contohnya bisa dilihat di atas, variabel `num` nilainya di-mutate atau diubah dari dalam closure, karena inilah variabel `increase_by` harus didefinisikan mutable.

A.46.4. Borrowing pada closure

Semua variabel di luar block closure ketika digunakan di dalam closure maka terjadi operasi borrowing pada variabel tersebut. Variabel di luar block closure dipinjam agar bisa digunakan di dalam closure.

Sebagai contoh, kode sederhana berikut menghasilkan error, karena variabel `num` adalah dipinjam oleh closure `increase_by` untuk dipergunakan di dalam block-nya.

```
let mut num = 5;
let mut increase_by = |x: i32| num += x;
```

```
let mut num: i32

cannot use `num` because it was mutably borrowed
use of borrowed `num` rustc(Click for full compiler diagnostic)

main.rs(4, 27): borrow of `num` occurs here

main.rs(4, 36): borrow occurs due to use of `num` in closure

main.rs(7, 5): borrow later used here

View Problem (Alt+F8)    No quick fixes available
num += 5;
```

Error muncul di statement setelahnya, yaitu `num += 5` karena `num` statusnya masih dipinjam oleh closure `increase_by`.

Variabel `num` dipinjam dengan mode peminjaman adalah *mutable borrow* karena closure didefinisikan mutable. Salah satu aturan pada borrowing: bahwa dalam waktu yang sama, ketika sudah terjadi mutable borrow, maka tidak boleh ada borrowing lainnya. Itulah alasan kenapa statement `num += 5` menghasilkan error.

Solusi dari masalah di atas ada beberapa, yang pertama adalah menggunakan block expression untuk meng-isolasi closure, agar peminjaman pada closure tersebut dan statement `num += 5` tidak terjadi dalam waktu yang sama.

```rust
let mut num = 5;
num += 5;

{
    let mut increase_by = |x: i32| num += x;
    increase_by(10);
}

println!("{num}"); // 20
```

Solusi di atas efektif untuk menghilangkan error borrowing yang sebelumnya muncul, tapi setelah d-refactor kode menjadi tidak sesuai spesifikasi awal. Pada kode yang baru di atas, deklarasi closure `increase_by` terjadi di dalam block expression, artinya closure ini hanya akan bisa digunakan pada block kode tersebut saja, tidak bisa di-reuse di luar block.

Solusi yang lebih baik pada kasus di atas adalah dengan tidak menggunakan default borrowing (yang terjadi di dalam block closure ketika mengakses variabel yang posisinya berada di luar closure), melainkan gunakan saja borrowing pada parameter closure. Kita ubah lagi kodenya menjadi seperti ini:

```
let mut num = 5;
let increase_by = |num: &mut i32, x: i32| *num += x;

num += 5;
increase_by(&mut num, 10);

println!("{num}"); // 20
```

Pada contoh di atas, closure `increase_by` ditambahi parameter baru. Sekarang ada 2 parameter, yaitu `num` yang tipe-nya adalah pointer `&mut i32` dan parameter `x`. Di dalam block closure, `num` di-dereference (menggunakan operator `*`) kemudian diubah nilainya. Setelah eksekusi statement selesai, data pinjaman tersebut langsung dikembalikan ke pemilik. Inilah kenapa kode di atas tidak menghasilkan error.

> *Pengaksesan variabel yang berada di luar scope closure tanpa via parameter berarti adalah borrowing, maka dalam penerapannya wajib untuk memperhatikan aturan yang berlaku pada ownership dan borrowing.*

A.46.5. Keyword `move`

Telah dijelaskan di atas bahwa variabel di luar closure, jika diakses dari dalam closure maka terjadi borrowing. Ada cara agar variabel tersebut ownership-nya berpindah ke dalam closure (*move semantics*), yaitu menggunakan keyword `move`.

Contohnya bisa dilihat berikut ini:

```
let mut num = 5;
let mut increase_by = move |x: i32| {
    num += x;
    println!("{num} (from closure)"); // 15
};

increase_by(10);
println!("{num}"); // 5
```

Closure `increase_by` di atas memiliki keyword `move` dalam pendefinisiannya. Dengan ini maka semua variabel di luar scope closure jika diakses dari dalam closure, maka variabel tersebut berpindah owner-nya (*move semantics*).

Lalu bagaimana dengan nasib variabel `num` yang berada di luar closure setelah owner-nya berpindah? Pada kondisi normal jawaban pertanyaan ini adalah tergantung tipe data-nya, jika *by default* variabel adalah mengadopsi *move semantics* maka variabel tersebut menjadi invalid. Namun pada contoh di atas `num` bertipe data `i32` yang mengadopsi *copy semantics*, maka variabel `num` tersebut masih bisa digunakan di luar closure.

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpe
    Finished dev [unoptimized + debuginfo]
     Running `target\debug\closures_1.exe`
15 (from closure)
```

Bisa dilihat dari gambar di atas, statement `print` dari dalam block closure hasilnya adalah benar, yaitu `5 + 10 = 15`. Dan variabel `num` di luar block closure tidak berubah nilainya.

A.46.6. Closure sebagai return type

◉ Praktek ke-1

Fungsi bisa memiliki nilai balik bertipe closure. Caranya dengan menggunakan `impl Fn()` sebagai tipe data nilai balik. Contoh penerapannya:

```
fn do_something() -> impl Fn() {
    println!("hello (from do_something)");

    return || {
        println!("hello (from closure)");
    };
}
```

Tipe `impl Fn()` adalah ekuvalen dengan closure `|| { }`.

```
fn main() {
    let my_closure = do_something();
    println!("hello (from main)");
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpe
    Finished dev [unoptimized + debuginfo]
     Running `target\debug\closures_1.exe`
hello (from do_something)
hello (from main)
hello (from closure)
```

Bisa dilihat, fungsi `do_something` di atas mengembalikan closure yang menampilkan string `hello (from closure)`. String tersebut hanya muncul ketika closure dipanggil.

Sedangkan fungsi `do_something` sendiri juga menampilkan pesan string lainnya, yaitu `hello (from do_something)`, yang pesan ini adalah muncul langsung saat pemanggilan fungsi `do_something`. Berbeda dengan pesan `hello (from closure)` yang hanya muncul ketika closure dieksekusi.

◉ Praktek ke-2

Pada contoh di atas, closure yang dikembalikan fungsi memiliki skema sangat sederhana, tanpa parameter dan argument. Mari coba praktek dengan contoh yang lebih kompleks.

```rust
fn do_something_v2() -> impl Fn(i32, String) -> String {
    println!("hello (from do_something_v2)");

    return |a: i32, b: String| -> String {
        let message = format!("{b} {a}");
        message
    };
}
```

Pada kode di atas fungsi `do_something_v2` mengembalikan closure dengan skema `Fn(i32, String) -> String`, yang artinya:

- Parameter pertamanya bertipe `i32`
- Parameter keduanya bertipe `String`
- Mengembalikan nilai bertipe `String`

Di dalam closure tersebut, data parameter digabung menjadi sebuah pesan string yang kemudian dijadikan nilai balik.

Sekarang jalankan fungsi `do_something_v2` di atas, kemudian lihat hasilnya.

```
let my_closure = do_something_v2();
let message = my_closure(123, "hello rust".to_owned());
println!("{message} (from main)");
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpe
    Finished dev [unoptimized + debuginfo]
     Running `target\debug\closures_3.exe`
hello (from do_something_v2)
hello rust 123 (from main)
```

A.46.7. Closure sebagai parameter fungsi

◉ Praktek ke-1

Pengaplikasian closure sebagai parameter fungsi banyak di lakukan di Rust, hal ini bisa dilakukan dengan memanfaatkan generic parameter. Silakan perhatikan kode berikut untuk contoh penerapannya.

```
fn run_x_times<F>(x: i32, my_closure: F)
where
```

Fungsi `run_x_times` dibuat dengan tugas mengeksekusi closure `my_closure` sebanyak `x` kali. Parameter closure harus selalu memiliki tipe data generic, contohnya `my_closure` di atas yang tipe data-nya adalah `F`.

Fungsi yang memiliki parameter closure wajib menggunakan keyword `where` yang keyword ini digunakan untuk mendaftarkan skema closure yang nantiya bisa diterima saat pemanggilan fungsi. Tipe generic `F` di daftarkan dengan notasi `Fn(i32)`, artinya dalam pemanggilan fungsi `run_x_times`, parameter `my_closure` harus diisi dengan closure yang skema-nya adalah `Fn(i32)` yang jika di ilustrasikan dalam bentuk closure adalah `|param1: i32| { }`.

Sekarang mari kita test fungsi di atas. Panggil fungsi `run_x_times` kemudian pada bagian parameter closure isi dengan block closure yang memiliki skema sesuai requirement, contohnya `|i: i32| { println!("hello rust {i}") }`.

```
run_x_times(4, |i: i32| {
    println!("hello rust {i}");
});
```

```
(base) PS D:\Labs\Vdam Studio\Ebook\dasarpe
   Compiling closures_4 v0.1.0 (D:\Labs\Vda
    Finished dev [unoptimized + debuginfo]
     Running `target\debug\closures_4.exe`
hello rust 0
hello rust 1
hello rust 2
hello rust 3
```

O iya, closure di atas juga bisa dituliskan dalam bentuk seperti ini. Bebas, pilih sesuai preferensi dan/atau kesepakatan team perihal code convention.

```
run_x_times(4, |i: i32| println!("hello rust {i}"));
```

◉ Praktek ke-2

Ok, sekarang mari kita coba praktekan contoh yang lebih kompleks.

Pada praktek ini kita akan buat sebuah fungsi yang tugasnya melakukan pencarian indeks slice dengan kondisi pencarian didapat dari eksekusi closure.

```
fn find_index<T, F>(data: &[T], cond_fn: F) -> i32
where
    F: Fn(&T) -> bool,
{
    for i in 0..data.len() {
        if cond_fn(&data[i]) {
            return i as i32
        }
    }

    return -1
}
```

Fungsi `find_index` memiliki 2 parameter generic, yaitu:

- `T` yang digunakan sebagai tipe data element slice `data`.
- `F` yang digunakan sebagai tipe data closure `cond_fn` dengan skema `Fn(&T) -> bool`, yang jika diilustrasikan dalam bentuk closure adalah `|param1: &T| -> bool { }`.

Di dalam fungsi tersebut, data slice di-loop, kemudian tiap elemen-nya digunakan sebagai parameter pemanggilan closure `cond_fn`.

Jika nilai balik pemanggilan closure adalah `true` maka `i` dikembalikan dalam bentuk `i32` (ada proses casting). Dan jika tidak diketemukan, maka indeks `-1`

dikembalikan.

> *Keyword* `as` *digunakan untuk casting tipe data. Lebih jelasnya dibahas pada chapter Type Alias & Casting dan Trait → Conversion (From & Into)*

Jalankan program tersebut.

```
let numbers = [24, 13, 2, 53, 3];
let number_to_find = 53;
let index = find_index(&numbers, |e: &i32| -> bool {
    if *e == number_to_find {
        true
    } else {
        false
    }
});

println!("number_to_find: {number_to_find}");
println!("index: {index}");
```

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpe
    Finished dev [unoptimized + debuginfo]
     Running `target\debug\closures_1.exe`

number_to_find: 53
index: 3
```

Bisa dilihat program berjalan sesuai harapan.

> *Keyword* `Fn` *merupakan salah satu trait function yang ada di Rust. Lebih jelasnya perihal keyword tersebut dibahas pada chapter selanjutnya, yaitu Trait → Function*

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-example/../closures
```

◉ Chapter relevan lainnya

- Function
- Generics
- Borrowing
- Trait → Function (Fn, FnMut, FnOnce)

◉ Referensi

- https://doc.rust-lang.org/book/ch13-01-closures.html
- https://doc.rust-lang.org/beta/rust-by-example/fn/closures.html

---

A.47. Trait → Function (Fn, FnMut, FnOnce)

Chapter ini merupakan topik lanjutan dari chapter sebelumnya, disini kita akan bahas beberapa jenis trait yang digunakan di closure.

Rust memiliki 3 buah trait yang otomatis ter-implement pada closure, yaitu `Fn`, `FnMut`, dan `FnOnce`. Ketiga trait ini juga otomatis ter-implement pada fungsi yang didefinisikan menggunakan keyword `fn`.

A.47.1. Trait `Fn`

`Fn` sudah kita terapkan beberapa kali saat praktek pembuatan closure di chapter sebelumnya. Fn adalah trait (std::ops::Fn), yang dipakai oleh Rust untuk menandai bahwa suatu closure bisa dipanggil berkali-kali dengan catatan di dalam closure tersebut tidak ada operasi mutable terhadap variabel yang scope-nya berada di luar block closure.

Contoh penerapan trait `Fn` bisa dilihat pada kode berikut. Trait tersebut secara otomatis ter-implement pada closure `pow`.

```
let pow_number = |n: i32| n.pow(2);

println!("pow_number(2): {}", pow_number(2));
println!("pow_number(3): {}", pow_number(3));
println!("pow_number(4): {}", pow_number(4));
```

Contoh penerapan trait `Fn` lainnya pada fungsi yang memiliki parameter closure:

```
fn main() {
    let result = do_something_with_number_v1(13, |d: i32| d * 2);
    println!("result: {result}");
}

fn do_something_with_number_v1<F>(n: i32, f: F) -> i32
where
    F: Fn(i32) -> i32, // <----- Fn digunakan
{
    // Statement `f(n)` bisa dipanggil berkali-kali
    return f(n);
}
```

A.47.2. Trait `FnMut`

Trait `FnMut` (std::ops::FnMut) merupakan trait yang menjadikan suatu closure bisa diakses berkali-kali dan bisa me-mutate atau mengubah data suatu variabel yang berada di luar scope block closure. Trait ini juga otomatis ter-implement pada closure yang di dalamnya ada kode pengaksesan variabel yang berada di luar scope block closure.

Contoh penerapannya silakan lihat closure `square_x` berikut. Closure tersebut di-dalamnya mengubah nilai `x` yang dideklarasikan di luar blcok closure, oleh karenanya trait `FnMut` otomatis ter-implement.

```
let mut x = 5;
{
    let mut square_x = || x *= x;
```

Contoh lain penerapan `FnMut` pada fungsi yang memiliki parameter closure:

```
fn main() {
    let mut number = 1;
    do_something_with_number_v2(14, |x| number += x);
    println!("number: {number}");
}

fn do_something_with_number_v2<F>(n: i32, mut f: F)
where
    F: FnMut(i32), // <----- FnMut digunakan
{
    // Statement `f(n)` berisi kode yang mengubah isi variavel
`number` (mutable).
    // `f(n)` bisa dipanggil berkali-kali
    f(n);
}
```

Parameter ke-2 fungsi `do_something_with_number_v2` adalah closure yang memutate nilai `number`, berarti terjadi mutable borrow karena variabel `number` berada di luar block closure. Karena alasan tersebut maka closure harus dideklarasikan menggunakan `FnMut` (bukan `Fn`).

Jika dipaksa deklarasi menggunakan `Fn`, hasilnya pasti error.

```
▶ Run | Debug
fn main() {
    let mut number: i32 = 1;
    do_something_with_number_v2(number: 14, f: |x: i32| number += x);
    x: i32

    cannot assign to `number`, as it is a captured variable in a `Fn` closure
    cannot assign rustc(Click for full compiler diagnostic)

    main.rs(9, 55): change this to accept `FnMut` instead of `Fn`

    main.rs(5, 5): expects `Fn` instead of `FnMut`

    main.rs(5, 37): in this closure

    View Problem (Alt+F8)    No quick fixes available
```

`FnMut` merupakan supertrait dari `Fn`, artinya closure dengan trait `Fn` juga bisa digunakan sebagai argument pemanggilan fungsi dimana parameter fungsi tersebut bertipe `FnMut`.

> *Lebih jelasnya mengenai supertrait dibahas pada chpater Supertrait*

A.47.3. Trait `FnOnce`

Trait `FnOnce` (std::ops::FnOnce) adalah trait yang menjadikan suatu closure hanya boleh di akses sekali saja, dan closure tersebut bisa berisi operasi mutable ataupun tidak.

Contoh penerapan `FnOnce` pada fungsi yang memiliki parameter closure.

```rust
fn main() {
    let mut number = 1;
    do_something_with_number_v3(14, |x| number += x);
    println!("number: {number}");
}
```

Jika closure dengan tipe `FnOnce` dipaksa diakses dua kali, pasti muncul error. Contohnya bisa dilihat pada parameter `f` berikut.

```
60  fn do_something_with_number_v3<F>(n: i32, f: F)
61  where
62      F: FnOnce(i32),
63  {
64      f(n);
65      f(n);
66  }
```

```
PROBLEMS 1   OUTPUT   GITLENS   DEBUG CONSOLE   TERMINAL

53 | fn do_something_with_number_v2<F>(n: i32, mut f: F)
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasarpemro
   |      ^
help: consider further restricting this bound
   |
62 |     F: FnOnce(i32) + Copy,
   |                    ++++++

For more information about this error, try `rustc --explain E0382`.
error: could not compile `playground` due to previous error
```

> *Lebih jelasnya mengenai supertrait dibahas pada chpater Supertrait*

A.47.4. Relasi antara trait function dengan function

Trait `Fn`, `FnMut`, dan juga `FnOnce` tidak hanya ter-implement pada closure, tapi juga ter-implement pada fungsi juga (secara otomatis).

Contohnya bisa dilihat pada contoh berikut. Fungsi `do_something_with_number_v1` dipanggil 3 kali.

- Pemanggilan pertama, argument `f` diisi dengan closure `|d: i32| d * 2`
- Pemanggilan kedua, argument `f` diisi dengan fungsi `double`
- Pemanggilan ketiga, argument `f` diisi dengan fungsi `pow_number`

```
fn main() {
    let result = do_something_with_number_v1(13, |d: i32| d * 2);
    println!("result: {result}");

    let result = do_something_with_number_v1(13, double);
    println!("result: {result}");

    let result = do_something_with_number_v1(13, pow_number);
    println!("result: {result}");
}

fn do_something_with_number_v1<F>(n: i32, f: F) -> i32
where
    F: Fn(i32) -> i32,
{
    return f(n);
}

fn double(d: i32) -> i32 {
    d * 2
}

fn pow_number(d: i32) -> i32 {
    d.pow(2)
}
```

Dari contoh di atas terbukti bahwa fungsi `double` dan juga `pow_number` memenuhi kriteria tipe `F` yaitu `Fn(i32) -> i32`.

```
(base) PS D:\Labs\Adam Studio\Ebook\dasarpemrogramanrust\dasa
   Compiling playground v0.1.0 (D:\Labs\Adam Studio\Ebook\das
    Finished dev [unoptimized + debuginfo] target(s) in 0.42s
     Running `target\debug\playground.exe`

result: 76
result: 26
result: 169
```

Catatan chapter 📑

◉ Source code praktek

```
github.com/novalagung/dasarpemrogramanrust-
example/../trait_function
```

◉ Chapter relevan lainnya

- Function
- Traits
- Closures

◉ Referensi

- https://doc.rust-lang.org/book/ch13-01-closures.html
- https://doc.rust-lang.org/beta/rust-by-example/fn/closures.html
- https://doc.rust-lang.org/std/ops/trait.Fn.html
- https://doc.rust-lang.org/std/ops/trait.FnMut.html
- https://doc.rust-lang.org/std/ops/trait.FnOnce.html

Work In-Progress

Semua chapter dalam kategori ini sedang dalam proses penulisan, dan secara periodik akan di update dengan tambahan chapter-chapter baru lainnya.

A.??. Async

> *Work in progress*

A.??. Attributes

> *Work in progress*

A.??. Bitwise Operation

> *Work in progress*

A.??. Box

> *Work in progress*

A.??. Bytes

> *Work in progress*

A.??. Cara Membaca Dokumentasi

> *Work in progress*

A.??. Cara Membaca Error Stack Trace

> *Work in progress*

A.??. Constant Evaluation

> *Work in progress*

A.??. Copy, Clone, Move, Drop

> *Work in progress*

A.??. dyn Trait

> *Work in progress*

A.??. Error Handling & Panic

> *Work in progress*

A.??. Extern

> *Work in progress*

A.??. Formatted Print

> *Work in progress*

A.??. Glossary / Kata Kunci

> *Work in progress*

A.??. HashMap & HashSet

> *Work in progress*

A.??. Library Crate

> *Work in progress*

A.??. Linked List

> *Work in progress*

A.??. Macro

> *Work in progress*

A.??. Memory Leak

> *Work in progress*

A.??. NLL

> *Work in progress*

A.??. Rust Standard Library

> *Work in progress*

A.??. Safe & Unsafe

> *Work in progress*

A.??. Smart Pointer vs. Raw Pointer

> *Work in progress*

A.??. Supertrait

> *Work in progress*

A.??. Testing

> *Work in progress*

A.??. Time

> *Work in progress*

A.??. Trait → Conversion (From & Into)

> *Work in progress*

A.??. Trait → Copy

> *Work in progress*

A.??. Trait → Iterator

> *Work in progress*

A.??. Whitespace Token

> *Work in progress*