Analisis Semantik Variabel, Mutabilitas, dan Siklus Memori dalam Bahasa Pemrograman Rust

Muhammad Riduwan Khafidi
Program Studi Teknik Rekayasa Perangkat Lunak
Politeknik Negeri Batam
4342611034@students.polibatam.ac.id

November 5, 2026

Abstrak

Makalah ini menyajikan kajian analitis mendalam mengenai fondasi semantik variabel dalam bahasa pemrograman Rust. Berbeda dengan mayoritas bahasa arus utama yang menjadikan mutabilitas sebagai perilaku bawaan, Rust secara fundamental menetapkan immutabilitas sebagai standar. Makalah ini menyajikan enam visualisasi yang memetakan tipologi pengikatan, perbedaan krusial antara reassignment dan shadowing, alur keputusan pemilihan kata kunci, serta siklus hidup memori pada blok scope. Melalui penelusuran sintaksis `let`, `const`, dan `static`, makalah ini menguraikan bagaimana Rust mendefinisikan ulang konsep variabel sebagai *binding* (pengikatan nama ke nilai). Studi kasus delapan langkah melengkapi pembahasan dengan menunjukkan secara langsung bagaimana kompilator menolak mutasi tak sah dan bagaimana data hidup di dalam memori.

1 Pendahuluan

1.1 Analogi Pembuka: Sticky Note versus Papan Tulis

Bayangkan sebuah meja kerja dengan dua alat tulis. Alat pertama adalah *sticky note* (kertas tempel). Jika Anda menulis angka 5 di sticky note, Anda tidak bisa menghapusnya menjadi 10. Jika ingin nilai 10, Anda harus mengambil sticky note baru, menulis 10, dan menempelkannya di atas yang lama. Sticky note lama tidak hilang dari meja, tetapi tertutupi. Inilah cara kerja `let` (immutable binding) dan `shadowing` di Rust.

Alat kedua adalah papan tulis putih. Anda bisa menulis 5, menghapusnya dengan penghapus, lalu menulis 10 di tempat yang sama. Alamat papannya tetap, nilainya yang berubah. Inilah cara kerja `let mut` (mutable binding) dan `reassignment`.

Mayoritas bahasa pemrograman di dunia memberikan Anda papan tulis secara bawaan. Rust justru memberikan Anda sticky note terlebih dahulu, dan hanya memberi papan tulis jika Anda memintanya secara eksplisit.

1.2 Mengapa Topik Ini Penting

Rust pertama kali dikembangkan di Mozilla Research pada tahun 2006 dan mencapai versi stabil 1.0 pada Mei 2015. Bahasa ini dirancang untuk menjembatani celah antara bahasa sistem berperforma tinggi seperti C/C++ dengan bahasa tingkat tinggi yang menjamin keamanan memori. Keputusan paling berpengaruh dalam desainnya adalah penetapan sifat *immutable* (tidak dapat diubah) sebagai nilai bawaan.

Kesalahan paling umum pemula saat belajar Rust bukanlah pada sintaksis *borrowing* atau *lifetime*, melainkan pada kesalahpahaman mendasar tentang variabel. Mereka sering bingung mengapa kompilator menolak kode yang terlihat sah, atau mengapa `let x = 5;` diikuti `let x = 10;` tidak menghasilkan error. Memahami semantik variabel adalah gerbang pertama sebelum melangkah ke topik konkurensi dan kepemilikan memori.

1.3 Peta Pembahasan

Makalah ini dibagi menjadi tujuh bagian utama. Bagian kedua membahas tipologi pengikatan nilai melalui `let`, `const`, dan `static`. Bagian ketiga membahas paradigma immutabilitas dan mutabilitas eksplisit. Bagian keempat membedah fenomena *shadowing* dan perbedaannya dengan *reassignment*. Bagian kelima membahas lingkup (*scope*) dan analisis statis inisialisasi. Bagian keenam menyajikan tabel perbandingan dan pohon keputusan. Bagian ketujuh membawa seluruh konsep ke dalam studi kasus delapan langkah. Bagian kedelapan dan kesembilan merangkum kesalahan umum dan lembar rujukan cepat.

2 Tipologi Pengikatan Nilai

2.1 Definisi Binding

Dalam Rust, istilah yang lebih tepat untuk variabel adalah *binding*. Yang terjadi bukan sekadar menyimpan nilai ke dalam wadah, melainkan mengikat sebuah nama (*identifier*) terhadap sebuah nilai di dalam memori. Rust menyediakan tiga bentuk pengikatan nilai tingkat atas yang memiliki aturan berbeda, yaitu `let`, `const`, dan `static`.

2.2 Pengikatan dengan let

Pengikatan nilai paling mendasar dilakukan melalui kata kunci `let` dengan sintaks `let nama = nilai;`. Binding yang dihasilkan oleh `let` secara bawaan bersifat *immutable*. Pada penggunaan `let`, kompilator Rust umumnya mampu melakukan inferensi tipe secara otomatis berdasarkan nilai yang diikatkan.

```
let umur = 20;
let nama = "Budi";
```

```
Kompilasi berhasil, tipe diinferensi
```

2.3 Pengikatan dengan const dan static

Bentuk pengikatan kedua adalah `const`, yang mendeklarasikan konstanta. Nilainya wajib dapat dievaluasi pada saat kompilasi (*compile-time*) dan mewajibkan anotasi tipe eksplisit. Nilai `const` akan disisipkan (*inlined*) secara langsung pada setiap tempat di mana konstanta tersebut dipanggil.

Bentuk ketiga adalah `static`, yang mendeklarasikan nilai global dengan masa hidup sepanjang program berjalan. Berbeda dengan `const`, nilai `static` menempati lokasi memori yang tetap. Sebuah `static` secara teknis dapat dijadikan *mutable*, namun hal ini sangat dibatasi dan menuntut penggunaan blok `unsafe`.

```
const MAX_SKOR: u32 = 100;
static BAHASA: &str = "Rust";
```

```
Kompilasi berhasil, tipe eksplisit
```

Gambar **??** memetakan ketiga tipologi pengikatan ini beserta sifat-sifat utamanya.

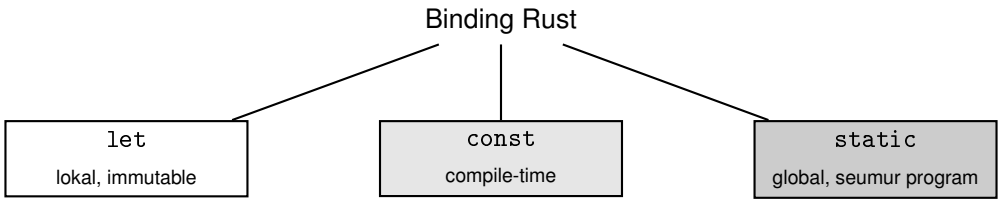


Figure 1. Tipologi pengikatan nilai dalam Rust dan sifat utamanya.

3 Paradigma Immutabilitas dan Mutabilitas Eksplisit

3.1 Immutable by Default

Salah satu ciri khas paling kentara dari Rust adalah sifat immutabilitas pada variabel yang dibuat melalui `let`. Setelah sebuah nilai diikatkan pada sebuah nama, nilai tersebut tidak dapat diganti melalui nama yang sama. Upaya mengubah binding yang tidak dideklarasikan dengan `mut` akan langsung ditolak oleh kompilator dengan pesan *cannot assign twice to immutable variable*.

Bagi kompilator, sifat ini sangat memudahkan proses penalaran terhadap alur program, mencegah kesalahan logika akibat mutasi yang tidak terduga, dan meminimalisir risiko *data race* pada program konkuren. Gambar **??** menunjukkan alur keputusan kompilator saat menemui upaya reassignment.

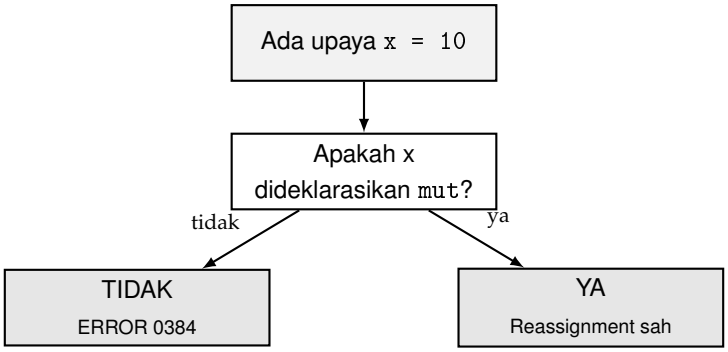


Figure 2. Alur keputusan kompilator Rust saat menemui upaya reassignment.

3.2 Mutabilitas Eksplisit dengan mut

Untuk keperluan di mana sebuah nilai harus berubah sepanjang siklus hidup program, kata kunci `mut` digunakan untuk memberikan izin reassignment. Deklarasi `let mut skor = 0;` mengizinkan baris kode berikutnya untuk melakukan reassignment seperti `skor = 10;`.

Kata kunci `mut` hanya memberikan izin, bukan kewajiban. Apabila pengembang mendeklarasikan `mut` tetapi variabel tersebut tidak pernah diubah nilainya, kompilator Rust akan memberikan peringatan berupa *variable does not need to be mutable*. Ini adalah bentuk kepedulian kompilator terhadap kebersihan dan presisi kode.

4 Shadowing versus Reassignment

4.1 Definisi Shadowing

Shadowing merupakan teknik di mana pengembang membuat binding baru menggunakan nama yang persis sama dengan binding yang sudah ada sebelumnya. Shadowing dilakukan dengan menulis kata kunci `let` sekali lagi untuk nama yang sama, dan hal ini sama sekali tidak membutuhkan kata kunci `mut`.

Jika terdapat baris `let x = 5;` lalu diikuti oleh `let x = 10;`, maka yang terjadi bukanlah perubahan nilai, melainkan pembuatan binding baru yang menutupi binding lama. Gambar **??** membandingkan secara visual apa yang terjadi pada memori antara reassignment dan shadowing.

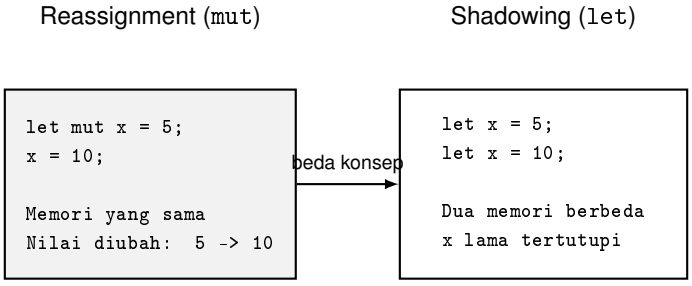


Figure 3. Perbandingan visual antara reassignment dan shadowing pada memori.

4.2 Siklus Hidup Memori pada Shadowing

Terdapat miskonsepsi umum yang menganggap shadowing menghapus nilai lama dari memori. Sesungguhnya, binding lama tidak lagi dapat diakses melalui nama tersebut, tetapi nilai yang diikat padanya tetap hidup di dalam memori. Nilai tersebut baru akan di-*drop* (dibersihkan dari memori) ketika eksekusi program mencapai akhir dari blok scope tempat binding tersebut dideklarasikan.

Shadowing juga memungkinkan terjadinya perubahan tipe data. Sebuah variabel yang awalnya menyimpan nilai integer, misalnya `let x = 5;`, dapat di-*shadow* menjadi tipe string dengan `let x = "hello";`. Kemampuan mengubah tipe inilah yang menjadikan shadowing lebih fleksibel dibandingkan reassignment biasa.

5 Lingkup Scope dan Analisis Statis

5.1 Scope Berbasis Kurung Kurawal

Scope atau lingkup adalah wilayah berlakunya sebuah binding, yang umumnya dibatasi oleh sepasang kurung kurawal. Binding diciptakan saat deklarasinya dieksekusi dan berakhir masa berlakunya tepat pada akhir scope. Binding luar dapat diakses dari scope yang lebih dalam, tetapi tidak sebaliknya.

Teknik shadowing dapat menutupi binding luar di dalam sebuah blok, sehingga untuk sementara waktu akses ke binding luar tersebut terhalang hingga blok selesai. Gambar **??** mengilustrasikan bagaimana sebuah binding lokal di dalam blok menutupi binding global di luarnya.

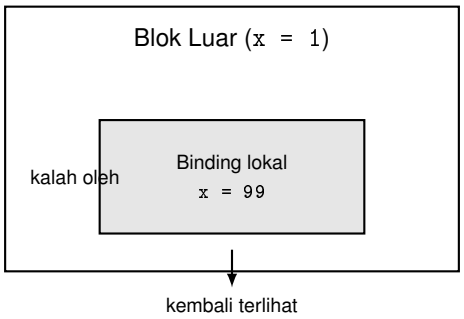


Figure 4. Ilustrasi scope bersarang dan shadowing di dalam blok.

5.2 Analisis Statis Inisialisasi

Berbeda dengan beberapa bahasa lain yang mengizinkan pembacaan variabel yang belum diinisialisasi, Rust menerapkan aturan yang sangat ketat. Sebuah binding diperbolehkan untuk dideklarasikan tanpa nilai awal, asalkan sebelum nilainya dibaca, binding tersebut sudah dipastikan memiliki nilai. Jika fungsi `println!` dipanggil sebelum variabel diisi, kompilator akan menghentikan proses dengan pesan kesalahan *used binding x isn't initialized*.

Kompilator Rust melakukan analisis statis yang peka terhadap alur eksekusi (*flow-sensitive*

analysis) untuk memastikan bahwa dalam skenario percabangan sekalipun, semua jalur eksekusi memberikan nilai pada variabel sebelum variabel itu digunakan.

5.3 Konvensi Penamaan dan Pola Destructuring

Aturan penamaan dalam Rust mewajibkan penggunaan huruf, angka, atau garis bawah, serta tidak boleh diawali dengan angka. Konvensi standar yang digunakan adalah *snake_case*. Selain penamaan individual, Rust juga mendukung *pattern destructuring* pada proses binding, di mana pengembang dapat mengikat beberapa nilai sekaligus, misalnya `let (a, b) = (1, 2);`.

Kompilator Rust juga sangat peduli terhadap kebersihan kode. Jika terdapat binding yang tidak pernah digunakan, kompilator akan memberikan peringatan *unused variable*. Pengembang dapat mengawali nama variabel dengan garis bawah untuk menekan peringatan ini, namun terdapat perbedaan mendasar antara `_` dan `_nama`. Penggunaan garis bawah tunggal (`_`) secara spesifik berarti nilai tersebut langsung dibuang dan tidak terjadi proses binding sama sekali. Sementara itu, `_nama` tetap melakukan proses binding, hanya saja kompilator mengabaikan peringatan jika variabel tersebut tidak digunakan.

6 Perbandingan dan Pohon Keputusan

Tabel **??** menyandingkan empat bentuk pengikatan variabel pada aspek-aspek kunci.

Table 1. Perbandingan bentuk pengikatan variabel dalam Rust.

Aspek	`let`	`let mut`	`const`	`static`
Mutabilitas	tidak bisa	bisa	tidak bisa	terbatas (`unsafe`)
Anotasi tipe	opsional	opsional	wajib	wajib
Waktu evaluasi	runtime	runtime	compile-time	runtime (lazy)
Alamat memori	ada	ada	tidak ada (inlined)	ada (tetap)
Lingkup	blok lokal	blok lokal	global/lokal	global
Bisa di-shadow	ya	ya	tidak	tidak

Gambar **??** menyusun temuan tersebut menjadi pohon keputusan yang dapat dipakai saat merancang kode.

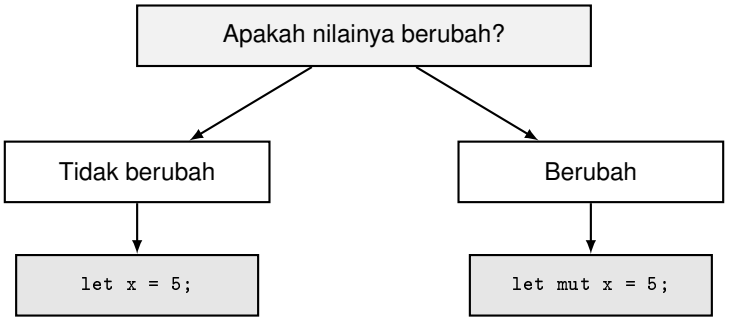


Figure 5. Pohon keputusan sederhana pemilihan kata kunci binding.

7 Studi Kasus: Delapan Langkah Menuju Pemahaman

Bagian ini mengikuti delapan langkah demonstrasi variabel Rust dari nol, dengan setiap keputusan dijelaskan berdasarkan perilaku kompilator.

7.1 Langkah 1: Deklarasi Immutable Dasar

```
fn main() {
    let umur = 20;
    println!("Umur saya {}", umur);
}
```

```
Umur saya 20
```

Binding `umur` dibuat tanpa `mut`, sehingga kompilator mengunci nilainya.

7.2 Langkah 2: Upaya Reassignment Ilegal

```
fn main() {
    let x = 5;
    x = 10;
}
```

```
error[E0384]: cannot assign twice to immutable variable
  --> src/main.rs:3:5
```

Ini membuktikan bahwa Rust menolak mutasi tanpa izin eksplisit.

7.3 Langkah 3: Reassignment yang Sah dengan mut

```
fn main() {
    let mut x = 5;
    println!("Nilai awal: {}", x);
    x = 10;
    println!("Nilai baru: {}", x);
}
```

```
Nilai awal: 5
Nilai baru: 10
```

Penambahan `mut` memberi tahu kompilator bahwa perubahan nilai adalah niat yang disengaja.

7.4 Langkah 4: Shadowing dan Perubahan Tipe

```
fn main() {
    let nilai = 10;
    let nilai = "sepuluh";
    println!("Nilai: {}", nilai);
}
```

```
Nilai: sepuluh
```

Binding lama bertipe integer tertutupi oleh binding baru bertipe string. Ini tidak bisa dilakukan oleh reassignment.

7.5 Langkah 5: Shadowing dalam Blok Scope

```
fn main() {
    let x = 1;
    {
        let x = 99;
        println!("Dalam blok: {}", x);
    }
    println!("Luar blok: {}", x);
}
```

```
Dalam blok: 99
Luar blok: 1
```

Setelah blok selesai, binding luar kembali terlihat karena binding dalam telah di-*drop*.

7.6 Langkah 6: Konstanta Compile-Time

```
const MAX_SKOR: u32 = 100;
fn main() {
    println!("Skor maksimum: {}", MAX_SKOR);
}
```

```
Skor maksimum: 100
```

Nilai `MAX_SKOR` disisipkan langsung ke kode mesin, tanpa alamat memori tetap.

7.7 Langkah 7: Inisialisasi Tertunda

```
fn main() {
    let nanti;
    nanti = 100;
    println!("Nilai nanti: {}", nanti);
}
```

```
Nilai nanti: 100
```

Kompilator mengizinkan deklarasi tanpa nilai, asalkan sebelum dibaca variabel sudah terisi.

7.8 Langkah 8: Destructuring dan Garis Bawah

```
fn main() {
    let (a, b) = (1, "dua");
    println!("a={}, b={}", a, b);

    let _tidak_dipakai = 5;
}
```

```
a=1, b=dua
warning: unused variable: ‘_tidak_dipakai‘
```

Garis bawah di awal nama mematikan peringatan tanpa membuang nilai dari memori.

8 Kesalahan Umum dan Cara Menghindarinya

Table 2. Kesalahan umum pada variabel Rust.

Kesalahan	Akibat	Perbaikan
reassignment tanpa `mut`	error E0384	tambah `mut` pada deklarasi
`const` tanpa anotasi tipe	error E0121	tulis `const X: u32 = 5;`
`mut` tidak pernah diubah	warning *unused mut*	hapus kata kunci `mut`
`static mut` tanpa `unsafe`	error E0133	bungkus akses dengan `unsafe`
mengira shadowing = reassignment	salah paham memori	ingat: shadowing buat binding baru
variabel dibaca sebelum diisi	error E0381	inisialisasi sebelum `println!`
lupa garis bawah variabel tak terpakai	warning *unused variable*	awali dengan `_`
`_` dianggap binding	nilai langsung dibuang	gunakan `_nama` jika perlu diikat

Dua kesalahan yang paling sering muncul adalah lupa menambahkan `mut` saat ingin melakukan reassignment, dan mengira `let x = 10;` menghapus nilai `x` sebelumnya. Keduanya bisa

dihindari dengan satu pertanyaan: apakah nilai ini perlu berubah, atau apakah ini sebenarnya butuh nama baru? Jika berubah, gunakan `mut`. Jika butuh nama baru, gunakan shadowing.

9 Lembar Rujukan Cepat

Table 3. Rujukan cepat semantik variabel Rust.

Kebutuhan	Sintaks
binding immutable	`let x = 5;`
binding mutable	`let mut x = 5;`
reassignment	`x = 10;`
shadowing	`let x = 10;`
konstanta	`const X: u32 = 5;`
variabel global	`static X: u32 = 5;`
destructuring tuple	`let (a, b) = (1, 2);`
mengabaikan nilai	`let _ = hitung();`
variabel tak terpakai	`let _nama = 5;`
inisialisasi tertunda	`let x; x = 5;`
shadowing tipe berbeda	`let x = 5; let x = "a";`
shadowing dalam blok	`{ let x = 1; }`
konstanta lokal	`const A: i32 = 1;` di dalam `fn`
unused mut warning	hapus `mut` jika tak diubah
immutable error	tambah `mut` jika perlu diubah
inisialisasi error	pastikan semua cabang mengisi nilai

10 Kesimpulan

Semantik variabel dalam Rust adalah pergeseran paradigma dalam berpikir mengenai manajemen state. Melalui desain *binding*, penegakan immutabilitas secara bawaan, penggunaan eksplisit kata kunci `mut`, serta analisis statis yang ketat, Rust memaksa pengembang untuk menulis kode yang prediktabel. Detail teknis seperti anotasi tipe pada konstanta, siklus hidup memori pada shadowing, dan pembedaan antara variabel berawalan garis bawah memperkuat fondasi sebelum melangkah ke topik lanjutan.

Tiga prinsip penutup:

1. Immutabilitas adalah bawaan, mutabilitas adalah izin. Variabel Rust terkunci secara default. Menambahkan `mut` adalah pernyataan niat eksplisit yang harus dipertanggungjawabkan.
2. Shadowing menciptakan, reassignment mengubah. Shadowing membuat entitas baru dengan nama yang sama (memungkinkan perubahan tipe), sementara reassignment memodifikasi nilai pada entitas yang sudah ada.
3. Kompilator adalah rekan, bukan musuh. Peringatan *unused variable* atau error *cannot assign twice* adalah mekanisme perlindungan agar kode bebas dari mutasi tak terduga dan data race.

Dengan kerangka mental ini, pembaca siap untuk melangkah ke topik kepemilikan memori (*ownership*) dan peminjaman (*borrowing*) yang menjadi inti dari keamanan memori Rust.

Referensi

References

[1] Klabnik, S., & Nichols, C. (2019). *The Rust Programming Language*. No Starch Press. https://doc.rust-lang.org/book/

[2] The Rust Project Developers. (2026). *The Rust Reference: Variables*. https://doc.rust-lang.org/reference/variables.html

[3] The Rust Project Developers. (2026). *Rust by Example: Variable Bindings*. https://doc.rust-lang.org/rust-by-example/variable_bindings.html

[4] The Rust Project Developers. (2026). *Rust Standard Library: Keyword const*. https://doc.rust-lang.org/std/keyword.const.html

[5] The Rust Project Developers. (2026). *Rust Standard Library: Keyword static*. https://doc.rust-lang.org/std/keyword.static.html

[6] The Rust RFCs. (2016). *RFC 1444: const fn*. https://rust-lang.github.io/rfcs/1444-const-fn.html

[7] Yanovski, J., Dang, H.-H., Jung, R., & Dreyer, D. (2021). GhostCell: Separating Permissions from Data in Rust. *Proceedings of the ACM on Programming Languages*, 5(ICFP), 1–30. https://doi.org/10.1145/3473597

[8] Priya, S., & Gurfinkel, A. (2025). Thrust: A Prophecy-Based Refinement Type System for Rust. *Proceedings of the ACM on Programming Languages*, 9(PLDI), Article 147. https://dl.acm.org/doi/10.1145/3720423

[9] The Rust Project Developers. (2026). *The Rustonomicon: The Dark Arts of Unsafe Rust*. https://doc.rust-lang.org/nomicon/