Sigurnost operacijskih sustava i aplikacija

Programski jezik Rust

Danko Delimar, 18.4.2025.

Pregled predavanja

- Pitanja za ispit
- Motivacija
- Što je Rust i zašto ga koristiti?
- Kako definirati memorijsku sigurnost?
- Dva načina kako Rust osigurava sigurnost i ergonomičnost: vlasništvo i reference

Pitanja za ispite

- Objasnite od kojih vrsta grešaka štiti Rust?
- Objasnite situacije u kojima je Rust potencijalno dobar odabir jezika?
- Objasnite vlasništvo (eng. *Ownership*) u kontekstu Rusta?
- Koje dozvole imaju varijable i reference unutar *borrow checkerovih* pravila u programskom jeziku Rust?
- Što se događa s varijablom u Rustu kada se u funkciju pošalje izmjenjiva referenca na tu varijablu?

Motivacija

- Chromium i Microsoft - 70% sigurnosnih grešaka su vezane uz nepravilno korištenje memorije [1]
- Jezici kao Java rješavaju problem sigurnosti memorije tako da maknu kontrolu iz ruku programera - ograničeniji i sporiji programi
- Ugradbena računala, operacijski sustavi i slično zahtijevaju potpunu kontrolu memorije

Što je Rust?

- Strogo tipizirani sistemski programski jezik s fokusom na memorijsku sigurnost, ergonomičnost i brzinu.
- Za razliku od jezika kao što su Java ili Go, osigurava memorijsku sigurnost tijekom prevođenja, ne pokretanja - nema *garbage collector*
- Memorijsku sigurnost osigurava koncept vlasništva (eng. *Ownership*)

Zašto Rust?

- Koristan kada je potreban presjek sigurnosti i performansi
- Ugradbena računala, operacijski sustavi, kritična oprema (medicinska, svemirska i slično)
- Za naučiti novi pogled na programiranje koji nije tipičan u drugim programskim jezicima

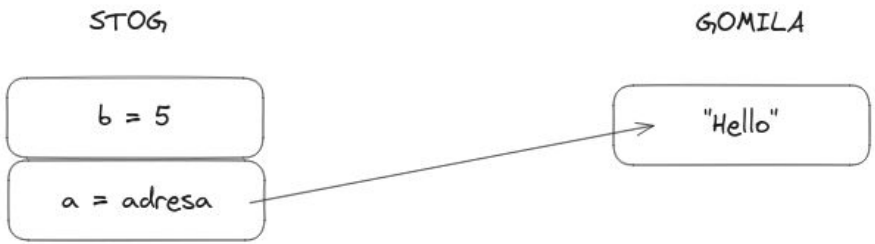
Memorijska (ne)sigurnost

- Memorijska sigurnost je izostanak nedefiniranog ponašanja
- Takva ponašanja dolaze u puno oblika:
  - use-after-free, double free, preljev spremnika
  - korištenje neinicijaliziranih varijabli
  - korištenje varijabli nepodudarajućih tipova
  - podatkovne utrke (eng. *race condition*)
  - više na [3]
- Rust ih sprječava sve u trenutku prevođenja (u teoriji)

Vlasništvo

- Vlasništvo je svojstvo varijable nad memorijom
- Varijable žive na stogu, “kutije” žive na gomili
- *a* je vlasnik memorije u kojoj postoji “Hello”

```
fn main() {
    let a = Box::new("Hello");
    let b = 5;
}
```

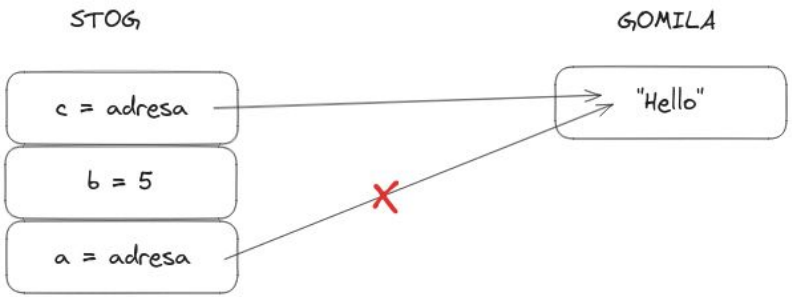


Slika 1. - varijable na stogu i gomili

Svojstva vlasništva

- Vlasništvo se prebacuje kada nastane nova varijabla koja pokazuje na tu memoriju
- Stari vlasnik se invalidira prilikom promjene vlasništva
- Kada se dealocira stogovski okvir novog vlasnika, dealocira se i memorija

```
fn main() {
    let a = Box::new("Hello");
    let b = 5;
    let c = a;
}
```



Slika 2. - prenošenje vlasništva

Primjeri vlasništva 1

Tko je vlasnik u M1, a tko u M2?

```
fn main() {
    let a = Box::new("Hello"); // M1
    let b = a; // M2
}
```

Slika 3. - Primjer vlasništva

Primjeri vlasništva 2

Tko je vlasnik u M1, tko u M2, a tko u M3?

```
fn add_world(mut arg: String) -> String {
    arg.push_str(" world"); // M2
    return arg;
}

fn main() {
    let a = String::from("Hello"); // M1
    let b = add_world(a); // M3
}
```

Slika 4. - Primjer vlasništva 2

Implikacije vlasništva

```
fn add_world(mut arg: String) -> String {
    arg.push_str(" world"); // M2
    return arg;
}

fn main() {
    let a = String::from("Hello"); // M1
    let b = add_world(a); // M3

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

Slika 5. - Implikacija vlasništva

Implikacije vlasništva - objašnjenje

- Ovaj kod se ne prevodi
- Trenutni vlasnik memorije na kojoj se nalazi “Hello world” je *b* što znači da je *a* invalidiran i ne može se koristiti
- Rust to radi jer kada bi i *a* i *b* mogli koristiti istu memoriju dolazi do problema koje će bolje ilustrirati C kod koji slijedi
- Ako *a* i *b* oboje pokazuju na istu memoriju moguće su double free greške i use-after-free kao neki primjeri kritičnih ranjivosti

Implikacije vlasništva - isti kod u C-u

```c
char* add_world(char* arg) {
    char* new_str = realloc(arg, strlen(arg) + strlen(" world"));
    strcpy(new_str, arg);
    strcpy(new_str + strlen(new_str), " world");

    return new_str;
}

int main() {
    char* str = malloc(8);
    strcpy(str, "Hello");
    char* with_world = add_world(str);

    printf("Original: %s | With world: %s", str, with_world);
}
// Ispisuje (na mojem laptopu)
//
// Original: Hello world | With world: Hello world
```

Slika 6. - C nesigurni kod

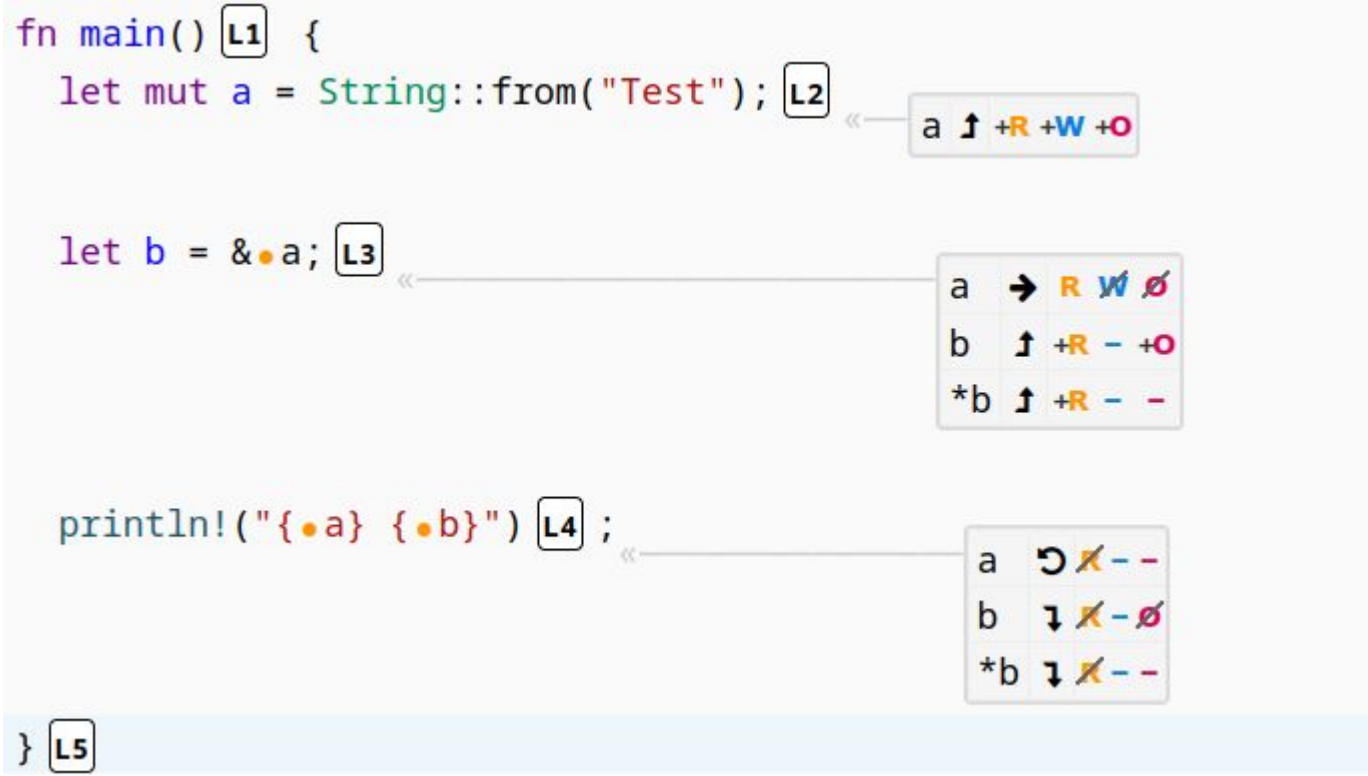
Reference

- Reference su mehanizam pristupa memoriji bez **trajnog** prijenosa vlasništva
- Pravila referenci:
  - Može postojati više referenci na istu memoriju ako nemaju pravo pisanja
  - Može postojati samo jedan referenca s pravom pisanja
  - Referenca s pravom pisanja i pravom čitanja ne mogu postojati
- Interno implementirane pomoću *borrow checkera*

Borrow checker

- Rustov način praćenja pravila referenci i osiguravanja sigurnog korištenja referenci
- Posuđivanje - privremeno prebacivanje vlasništva s varijable na referencu -> vraća se kada završi *lifetime* reference
- *Lifetime* - vrijeme od inicijalizacije do zadnjeg korištenja neke varijable
- Interno implementiran pomoću sustava tri dozvole - čitanje, pisanje, vlasništvo (RWO)

Primjeri posuđivanja 1



Slika 7. - Reference dozvole 1

Kako čitati Aquascope?

- Na prijašnjoj slici, taman nakon trenutaka L2, L3 i L4 možemo vidjeti dozvole koje varijable imaju
- Nakon L2 stvorena je izmjenjiva varijabla *a* koja ima sve dozvole - RWO
- Nakon L3 stvara se referenca *b* koja posuđuje varijablu *a* - time *a* gubi mogućnost pisanja i gubi vlasništvo
- Nakon L4 prestaju postojati i *a* i *b* jer im završava *lifetime* i time gube sve dozvole

Primjeri posuđivanja 2

Slika 8. - Reference i dozvole 2

Primjeri posuđivanja 2 - objašnjenje

- U L3 nastaje promjenjiva referenca na *a* i time *a* gubi sve dozvole (primjetite da sama varijabla *b* nije promjenjiva pa nema W dozvolu)
- Nakon L4 *b* prestaje postojati jer mu završava *lifetime* i time *a* dobija sve dozvole natrag -> *b* ih samo **posuđuje**
- L5 pokazuje da *a* stvarno ima i W dozvolu, što je i očekivano
- Nakon L6 završava *lifetime* od *a*

Zaključak

- Rust postoji u specifičnoj niši na presjeku performansi i potrebe za sigurnošću
- Ako ste programer ugradbenih računala ili vas zanimaju probajte naučiti Rust
- Ako niste, i dalje probajte naučiti Rust jer će vas napraviti boljim programerom općenito

Literatura

[1] Chromium security - memory safety:
https://www.chromium.org/Home/chromium-security/memory-safety/
[2] The Rust Programming Book - Brown University edition:
https://rust-book.cs.brown.edu
[3] Rust lang - Behaviour considered undefined:
https://doc.rust-lang.org/reference/behavior-considered-undefined.html
[4] Aquascope -
https://github.com/cognitive-engineering-lab/aquascope?tab=readme-ov-file

Dodatna literatura

- https://agorism.dev/book/rust/rust-in-action_samuel-mcnamara.pdf
- https://github.com/DioxusLabs/dioxus

Hvala!