Rust i napredni tipski sustavi

3. Rust - upravljanje memorijom

Ivan Radiček

27. ožujka 2021.

Zagrijavanje: veličina podataka u memoriji

- Svaki podatak (skoro!) zauzima neku memoriju
- Kako provjeriti koliko?
- Pomoću `std::mem::size_of::<T>` (vidi: https://doc.rust-lang.org/std/mem/fn.size_of.html)
- Primjeri: [Igralište]

Motivacija - rekurzivni tip podataka

- Vrlo popularan način modeliranja podataka u funkcijskim jezicima
- Na ovaj način neće raditi u Rustu . . . zašto?

Definicija liste (pogrešno!)

```
1  enum List<T> {
2    Nil,
3    Cons { head: T, tail: List<T> }
4  }
```

[Igralište]

Stack

- Vrlo jednostavan i brz način alociranja memorije
- Kod izlaska funkcije moguće počistiti svu memoriju funcije
- Ali: zahtjeva da je poznata **veličina** zauzete memorije kod kompajliranja
- Vidi više:
  https://en.wikipedia.org/wiki/Stack-based_memory_allocation

Heap (dinamička memorija)

- Alocira se dinamički (po potrebi za vrijeme izvođenja programa)
  - `malloc` u C-u
  - Automatski (skriveno od korisnika) u višim programskim jezicima: OCaml, Python, C#, Java, ...
- Kako se *"oslobađa"* kada više nije korisna?
  - Eksplicitno sa `free` u C-u
  - U višim programskim jezicima pomoću **garbage collectora**
- Vidi više: https://en.wikipedia.org/wiki/Memory_management#HEAP

Heap - oslobađanje memorije

- Eksplicitano sa `free`:
  - Programer kontrolira memoriju
  - Lagano pogriješiti: korištenje oslobođene ili ne-inicijalizirane memorije, duplo oslobađanje, ne-oslobođena memorija koja više netreba (memory leak)
- Automatski pomoću GC-a:
  - Programer ne mora birinuti o memoriji (nema grešaka)
  - Ali: manja kontrola i **veći "troškovi izvođenja" programa**
- Pristup upravljanju memorije u Rustu je negdje između
  - Kompajler sam umeće ekvivalente `malloc` i `free` naredbama (nema dodatnoga posla kod izvođenja)
  - Tipski sustav koji prati upravljanje memorijom se brine da su naredbe umetnute na ispravna mjesta (nema gore navedenih grešaka)

Rust: primjer podataka promjenjive veličine

- Prije: primjer podataka na heapu u Rustu
- Vektor: niz podataka (istog tipa!) promjenjive duljine (https://doc.rust-lang.org/std/vec/struct.Vec.html)
- Slično kao `list` u Pythonu ili `List` u C#-u
- Što se događa u memoriji/kod debuga?

```
1  fn make_vector(size : isize) -> Vec<isize> {
2    let mut v = Vec::new();
3    for i in 1..=size {
4      v.push(i)
5    }
6    v
7  }
8
9  fn main() {
10   let v = make_vector(7);
11   println!("{:?}", v)
12 }
```

Još jedan jednostavniji tip promjenjive veličine

- Slično kao i `Vec` (memorija/debug?)

```
1  fn hello_world(name : String) -> String {
2    let mut greet = String::new();
3    greet.push_str("Hello ");
4    greet.push_str(&name);
5    greet.push_str("!");
6    greet
7  }
8
9  fn main() {
10   let args : Vec<String> = std::env::args().collect();
11   let name = args.get(1).expect("Expected a name!").to_string();
12   let greet = hello_world(name);
13   println!("{}", greet);
14 }
```

Osnove memorije u Rustu: vlasništvo nad podacima

1. Svaka vrijednost ima varijablu koja joj je **vlasnik** (*owner* u Rust terminologiji)
2. Vrijednost može imati **samo jednog** vlasnika (ali ga može mijenjati)
3. Kada varijabla izađe **"iz scopea"**, vrijednost se oslobađa (*drop* u Rust terminologiji)

Rust: promjena vlasnika i samo jedan vlasnik

```rust
fn main() {
    // s1 je vlasnik stringa
    let s1 = String::from("Hello world");

    // s2 postaje vlasnik string
    let s2 = s1;

    // Ovo više nije OK:
    // println!("{}", s1);

    // Ali ovo je:
    println!("{}", s2);

} // Ovdje se string s2 oslobađa pošto s2 izlazi iz scopea
```

- Promjena vlasnika se u Rustu zove **move** (vidi `cargo check`)
- No postoji i drugi način . . . **copy**

Copy vs. Move

- Svi *skalarni* tipovi su automatski *Copy*
- Što znači da se njihova vrijednost kopira (*Copy*), a ne *premješta* (*Move*) iz varijable u varijablu
- Odnosno . . .

```
1  fn main() {
2    let x = 42;
3    let y = x;
4    println!("Koristim obje vrijednosti: {} {}", x, y)
5  }
```

Još kopiranja

- I kombinacije jednostavnih podataka **mogu biti** *Copy* ako *"implementiraju"* Copy trait
- Za *Copy* trait, treba implementirati i *Clone* trait (kasnije detalji)
- Srećom obje možemo pomoću `derive` (a možemo i "ručno")

```
1  #[derive(Debug, Clone, Copy)]
2  struct Point {
3    x: f64,
4    y: f64
5  }
6
7  fn main() {
8    let p1 = Point { x: 1.0, y: 2.0 };
9    let p2 = p1;
10   println!("p1={:?}, p2={:?}", p1, p2));
11 }
```

Možemo li sve kopirati?

```
1   #[derive(Debug, Clone)]
2   struct Strings {
3     x: String,
4     y: String
5   }
6
7   impl Copy for Strings {}
8
9   fn main() {
10    let s1 = Strings { x: String::from("a"), y: String::from("b") };
11    let s2 = s1;
12    println!("s1={:?}, s2={:?}", s1, s2));
13  }
```

- Više o Copy traitu i koji tipovi ga mogu implementirati: https://doc.rust-lang.org/std/marker/trait.Copy.html
- Trebao bi ga implementirati svaki tip koji može!

Poziv funkcije i move

- Kod poziva funkcije vlasništvo se prenosi u funkciju (varijabla parametra postaje vlasnik)
- Ali . . . isto tako ga može i vratiti

```
1   fn takes_ownership(s : String) {
2     println!("I'm the owner now! {}", s)
3   } // Nakon ovoga se memorija koja drži "s" oslobađa
4
5   fn returns_ownership(s : String) -> String {
6     println!("I'm the owner now, but returning it :) {}", s);
7     s
8   } // Nakon ovoga se memorija koja drži "s" ne oslobađa
9
10  fn main() {
11    let s1 = String::from("A");
12    takes_ownership(s1);
13    // Ovo nije više OK jer je memorija već oslobođena
14    // let s2 = s1;
15
16    let s3 = String::from("B");
17    let s4 = returns_ownership(s3);
18    // Ono, naravno, ipak nije više OK, jer je vlasnik "s4"
19    // println!("{}", s3);
20    // Ali je ovo OK
21    println!("{}", s4);
22  }
```

Više o vraćanju vlasništva

```
1   fn bad_length(s : String) -> usize {
2     s.len()
3   } // Nakon ovoga se memorija koja drži "s" oslobađa
4
5   fn better_length(s : String) -> (usize, String) {
6     (s.len(), s)
7   } // Nakon ovoga se memorija koja drži "s" ne oslobađa
8
9   fn main() {
10    let s1 = String::from("Ovo je testni string");
11    let l1 = bad_length(s1);
12    println!("Prva duljina: {}", l1);
13    // Ovo nije više moguće
14    // println!("{}", s1);
15
16    let s2 = String::from("Ovo je drugi testni string");
17    let (l2, s2) = better_length(s2);
18    println!("Druga duljina: {}", l2);
19    // Ovo je sad moguće
20    // println!("{}", s2);
21  }
```

Više o vraćanju vlasništva

```
1   fn bad_length(s : String) -> usize {
2     s.len()
3   } // Nakon ovoga se memorija koja drži "s" oslobađa
4
5   fn better_length(s : String) -> (usize, String) {
6     (s.len(), s)
7   } // Nakon ovoga se memorija koja drži "s" ne oslobađa
8
9   fn main() {
10    let s1 = String::from("Ovo je testni string");
11    let l1 = bad_length(s1);
12    println!("Prva duljina: {}", l1);
13    // Ovo nije više moguće
14    // println!("{}", s1);
15
16    let s2 = String::from("Ovo je drugi testni string");
17    let (l2, s2) = better_length(s2);
18    println!("Druga duljina: {}", l2);
19    // Ovo je sad moguće
20    // println!("{}", s2);
21  }
```

- Sreća, postoji bolji način na ovo . . . *posudba*!

- Sljedeći put
  - Posudba vrijednosti
  - Pametni pokazivači/reference