Dr Dinu Dragan

PARALELNE I DISTRIBUIRANE ARHITEKTURE I JEZICI (ČAS 4)

ŠTA RADIMO DANAS?

- Vlasništvo (**Ownership**)
- Pomeranje (**Move**)
- Kopirani tipovi (**Copy Types**)
- Deljeno vlasništvo (**Rc** i **Arc**)

OWNERSHIP

OWNERSHIP

- **Ownership** (vlasništvo) je Rust-ova karakteristika po čemu je on jedinstven
- Ima duboke implikacije na ostatak jezika
- Omogućava Rust-u da garantuje bezbednost memorije bez potrebe za sakupljačem smeća (garbage collector GC), tako da je važno razumeti kako funkcioniše vlasništvo
- Ownership je **skup pravila** koja regulišu kako Rust program upravlja memorijom
- Kako radi GC memorija?
- Kako radi memorija kojom upravlja programer?

UPRAVLJANJE MEMORIJOM

- Kada je u pitanju upravljanje memorijom, postoje dve karakteristike koje se očekuju od programskog jezika
  - da se memorija oslobodi odmah, u trenutku po našem izboru, što daje punu kontrolu nad potrošnjom memorije programa
  - da se ne koristi pokazivač na objekat nakon što je oslobođen, što bi bilo nedefinisano ponašanje, koje bi dovelo do padova i bezbednosnih rupa
- U većini slučaja, ova dva pristupa se međusobno isključuju
- Postoje dva pristupa u upravljanju memorijom
  - „Bezbednost na prvom mestu“ – koristi sakupljanje smeća za upravljanje memorijom i automatsko oslobađanje objekata kada svi dostupni pokazivači na njih nestanu; ipak prepušta se kontrola nad tim kada se oslobađa memorija
  - „Kontrola na prvom mestu“ sva odgovornost za oslobađanje memorije na strani programera, ali izbegavanje visećih pokazivače takođe postaje u potpunosti briga programera

UPRAVLJANJE MEMORIJOM

- Rust ograničava kako programi koriste pokazivače
- Navikavanje na ove restrikcije je osnov učenja programiranja u Rustu
- Rust upravlja memorijom kroz sistem vlasništva čija pravila korišćenja nameće sam kompajler
- Ako je neko od pravila prekršeno, program se neće iskompajlirati
- Dobra stvar je to što upotreba ovog sistema ne usporava rad programa tokom izvršavanja
- Šta je to heap?
- Šta je to stack?
- Kako oni rade?

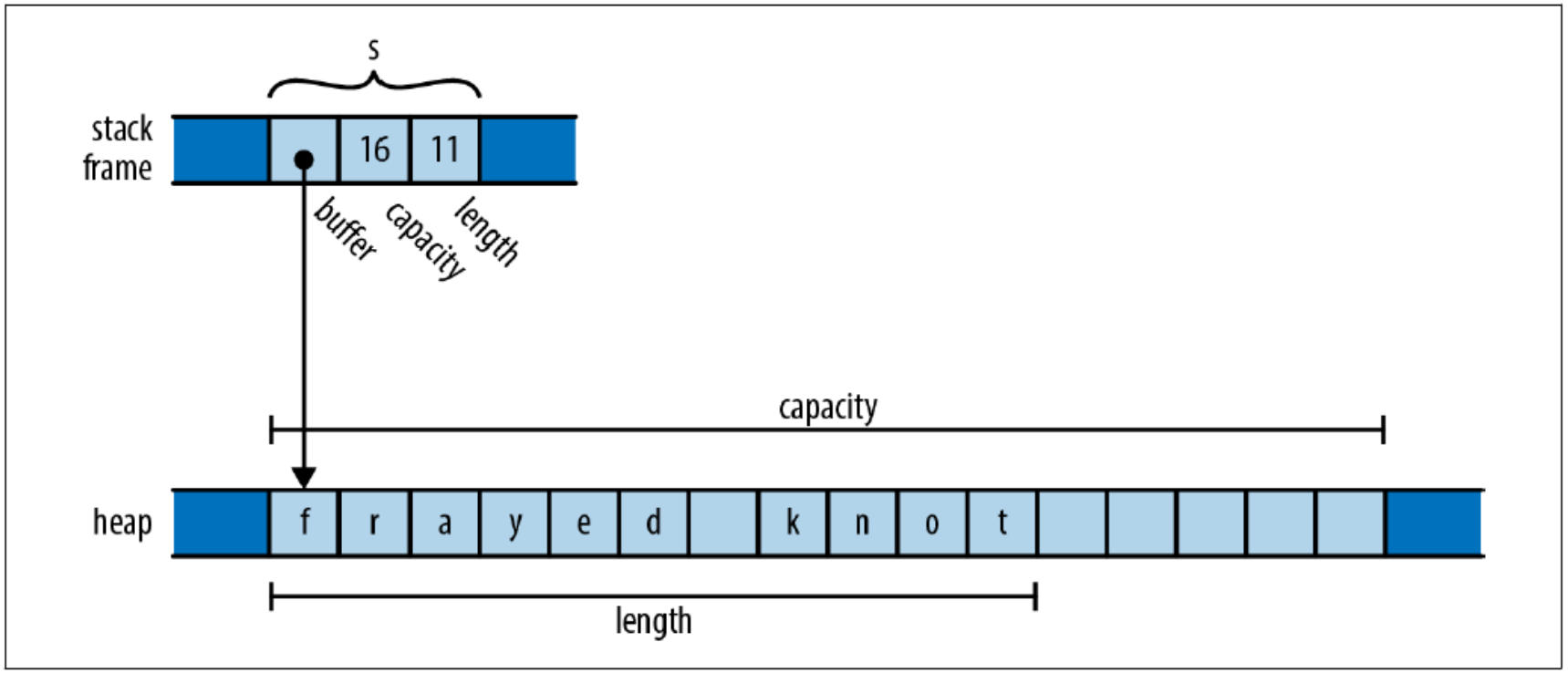
ŠTA VLASNIŠTVO ZAPRAVO ZNAČI

- Šta znači kada se kaže da instanca neke klase poseduje (**owns**) neki drugi objekat?
- U opštem slučaju to znači da je instanca koja poseduje objekat odlučuje i kada se taj objekat oslobađa (briše) – u svakom slučaju, brisanje instance dovodi i do brisanja posedovanog objekta
- Na primer, u C++

```
std::string s = "frayed knot";
```

- Na steku se pravi pokazivač na bafer sa stringom
- **std::string s** poseduje bafer
- Kada program oslobodi string **s** obrisaće se i bafer
- Moguće je napraviti druge reference na taj bafer, ali podrazumeva se da je **string s** odgovoran za bafer i na programeru je da se postara da su sve reference na bafer uništene, pre no što vlasnik oslobodi bafer

ŠTA VLASNIŠTVO ZAPRAVO ZNAČI



- Vlasnik određuje životni vek objekta, a svi ostali moraju da poštuju njegove odluke i da se postaraju da reference ne pokazuju slučajno na objekat koji više ne postoji

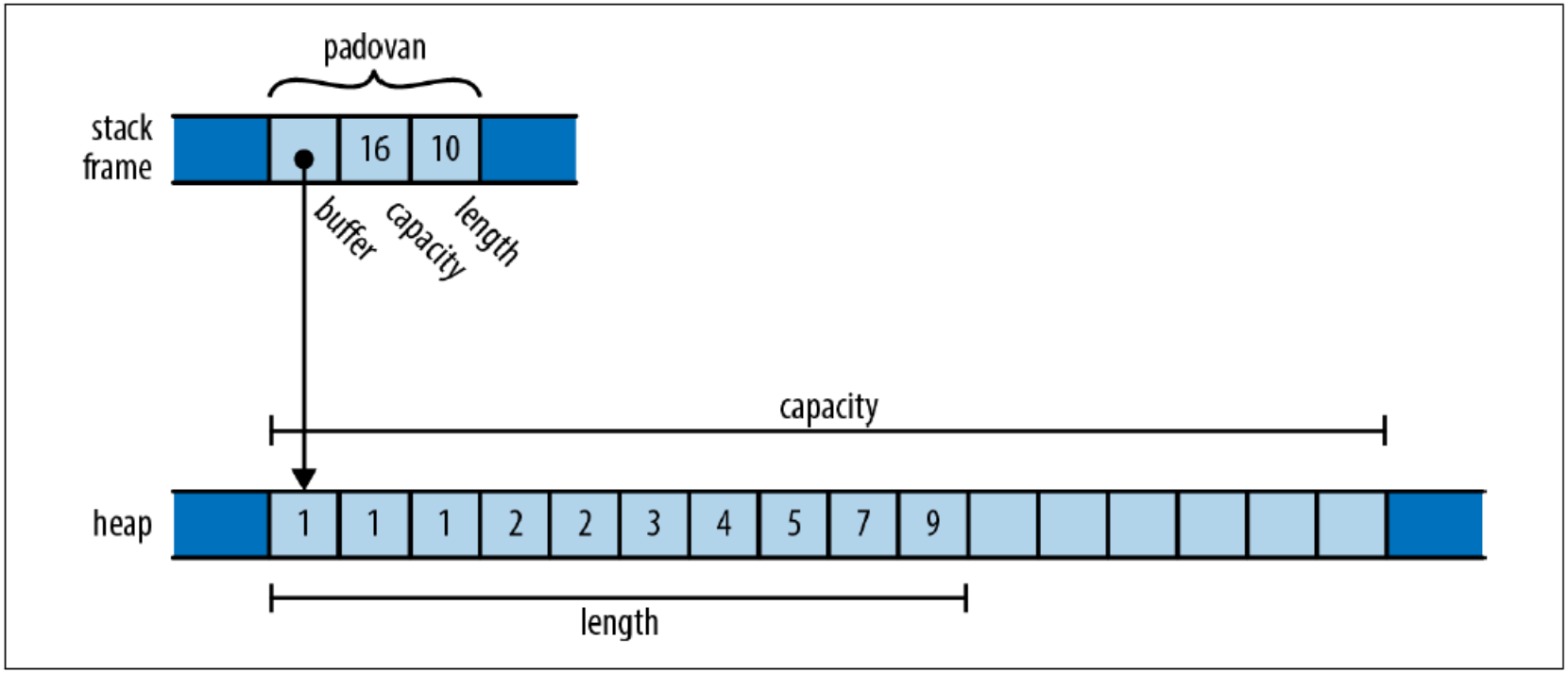
ŠTA VLASNIŠTVO ZNAČI U RUSTU

- U Rustu koncept vlasništva je ugrađen u sam jezik i nametnut proverama u vreme kompajliranja
- Svaka vrednost ima jednog vlasnika koji određuje njen životni vek
- Kada je vlasnik oslobođen — obrisan **(dropped),** briše se i vrednost u vlasništvu
- Promenljiva poseduje svoju vrednost – kad se izađe iz bloka u kome je promenljiva deklarisana, promenljiva se briše a i njena vrednost

```
fn print_padovan() {
    let mut padovan = vec![1,1,1];  // allocated here
    for i in 3..10 {
        let next = padovan[i-3] + padovan[i-2];
        padovan.push(next);
    }
    println!("P(1..10) = {:?}", padovan);
}                                   // dropped here
```

ŠTA VLASNIŠTVO ZNAČI U RUSTU

- Referenca na vektor **padovan** se nalazi na steku, dok se sam vektor nalazi na heapu i sam bafer je u njenom vlasništvu
- Tako da kad promenljiva **padovan** izađe izvan opsega, briše se i ona i njen bafer



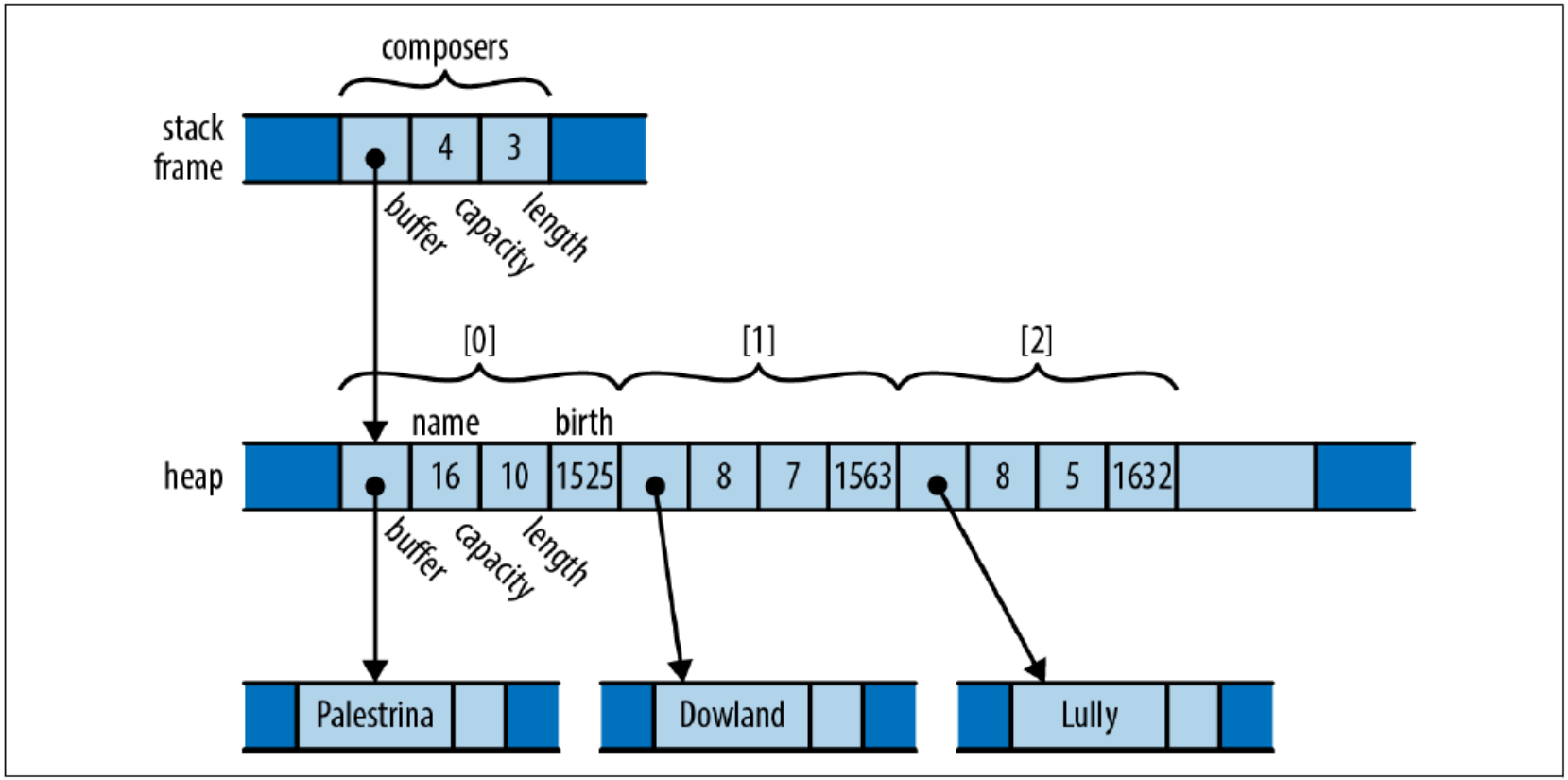
ŠTA VLASNIŠTVO ZNAČI U RUSTU

- Kao što promenljiva poseduje svoju vrednost, tako i nizovi, vektori, n-torke i strukture poseduju svoje elemente i polja
- Na primer:

```
struct Person { name: String, birth: i32 }

let mut composers = Vec::new();
composers.push(Person { name: "Palestrina".to_string(),
                        birth: 1525 });
composers.push(Person { name: "Dowland".to_string(),
                        birth: 1563 });
composers.push(Person { name: "Lully".to_string(),
                        birth: 1632 });
for composer in &composers {
    println!("{}, born {}", composer.name, composer.birth);
}
```

ŠTA VLASNIŠTVO ZNAČI U RUSTU

- Ove je **composers** vektor **Vec<Person>** struktura koje poseduju stringove i brojeve



- Vlasništvo je ovde jasno i jednostavno, brisanje **composers** briše sve

KAKO VLASNIŠTVO ZAPRAVO RADI U RUSTU

- U Rustu se prati jednostavna struktura vlasništva, **Vlasnici i vrednosti koje su u njihovom vlasništvu formiraju stabla:**
  - vlasnik je nadređeni čvor stabla, a vrednosti koje poseduje su podređeni čvorovi
  - u krajnjem korenu svakog drveta je promenljiva; kada ta promenljiva izađe van opsega, celo stablo se briše sa njom
  - čvorovi stabla su različitog tipa
  - ne mogu biti nikakve druge (direktne) veze između čvorova stabla osim veza samog stabla, tj. veza nadređeni-podređeni
- U Rustu, memorija se ne oslobađa eksplicitno, tj. nema **drop**, **delete**, ili **free** metode
  - do oslobađanja dolazi kada promenljiva izađe iz opsega, ili kada se element obriše iz vektora, ili nekom sličnom operacijom, pri čemu Rust brine da je memorija pravilno oslobođena

KAKO VLASNIŠTVO ZAPRAVO RADI U RUSTU

- U izvesnom smislu, Rust je manje moćan od drugih jezika:
  - svaki drugi praktični programski jezik vam omogućava da pravite proizvoljne grafikone objekata koji upućuju jedni na druge na koji god način smatrate prikladnim
- Ali upravo zbog ovih ograničenja, analize koje Rust može da izvrši nad programima su moćnije
- Rust-ove sigurnosne garancije su moguće upravo zato što je lakše analizirati ove ograničene odnose (nema kompleksnih odnosa u kodu)
- U ovome je Rustove radikalno nov i radikalno striktan
- Autori Rusta tvrde da obično postoji više nego dovoljno fleksibilnosti u načinu rešavanja problema kako bi se osiguralo da bar nekoliko savršeno finih rešenja potpadne unutar ograničenja jezika nameće, ali se na to treba navići i pristupiti drugačije od onoga na šta se naviklo!

PRAVILA VLASNIŠTVA U RUSTU

- Svaka vrednost **uvek** ima vlasnika!
- Svaka vrednost **uvek** ima samo **jednog** vlasnika!
- Kada vlasnik izađe iz opsega, vrednost će se **sigurno** obrisati!

- Ova pravila su garantovana od strane Rust kompajlera

FLEKSIBILNOST U MEHANIZMU VLASNIŠTVA

- Mehanizmi koji omogućuju fleksiblnost u Rustovom rigidnom pristupu vlasništu:
  - Vlasništvo se može prenositi (**move**) od jednog vlasnika do drugog; to omogućava da se izgrade nova, preurede stara ili potpuno rasformiraju stabla vlasništva
  - Veoma jednostavni tipovi kao što su celi brojevi, brojevi sa pokretnim zarezom i karakteri su izuzeti od pravila o vlasništvu; to su tzv. kopirani tipovi (**copy types**)
  - Standardna biblioteka obezbeđuje tipove pokazivača sa brojanjem referenci **Rc** i **Arc**, koji dozvoljavaju vrednostima da imaju više vlasnika, pod određenim ograničenjima
  - Može se pozajmiti referenca na vrednost (**borrow**); reference su pokazivači koji nisu vlasnici, sa ograničenim životnim vekom
- Svaki od ovih mehanizama dodaje fleksibilnost mahanizmu vlasništva ali i dalje obezbeđuju Rustov rigidni model vlasništva

POMERANJE (MOVE)

POMERANJE (MOVE)

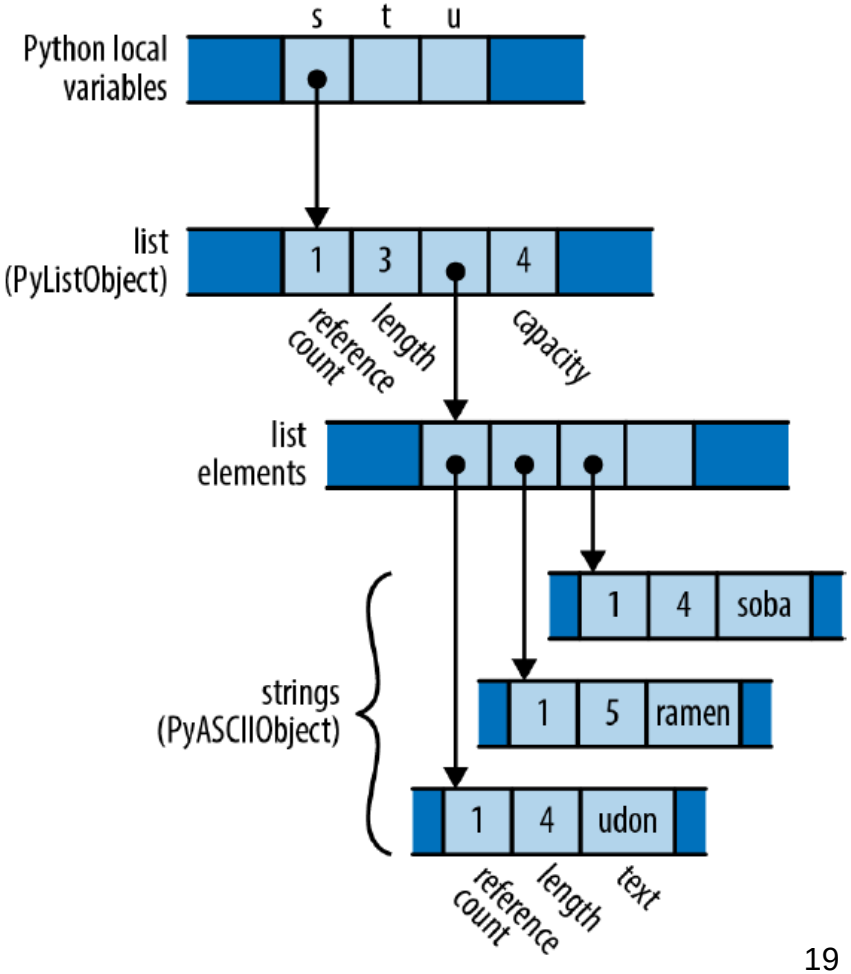
- U Rustu (za većinu tipova) operacije kao što je dodeljivanje vrednosti promenljivoj, prosleđivanje parametara funkciji ili vraćanje iz funkcije ne kopiraju vrednost: **one je pomeraju**
  - **Izvor se odriče vlasništva** nad vrednošću odredištu i postaje neinicijalizovan;
  - **odredište** sada **kontroliše** životni vek vrednosti
  - Rust programi izgrađuju i ruše složene strukture jednu po jednu vrednost, jedan potez u isto vreme
- Zašto to?

POMERANJE (MOVE)

- Iako to ne deluje tako, različiti programa implementiraju dodelu vrednosti na različite načine (skrivene od programera)
- Npr. Python:

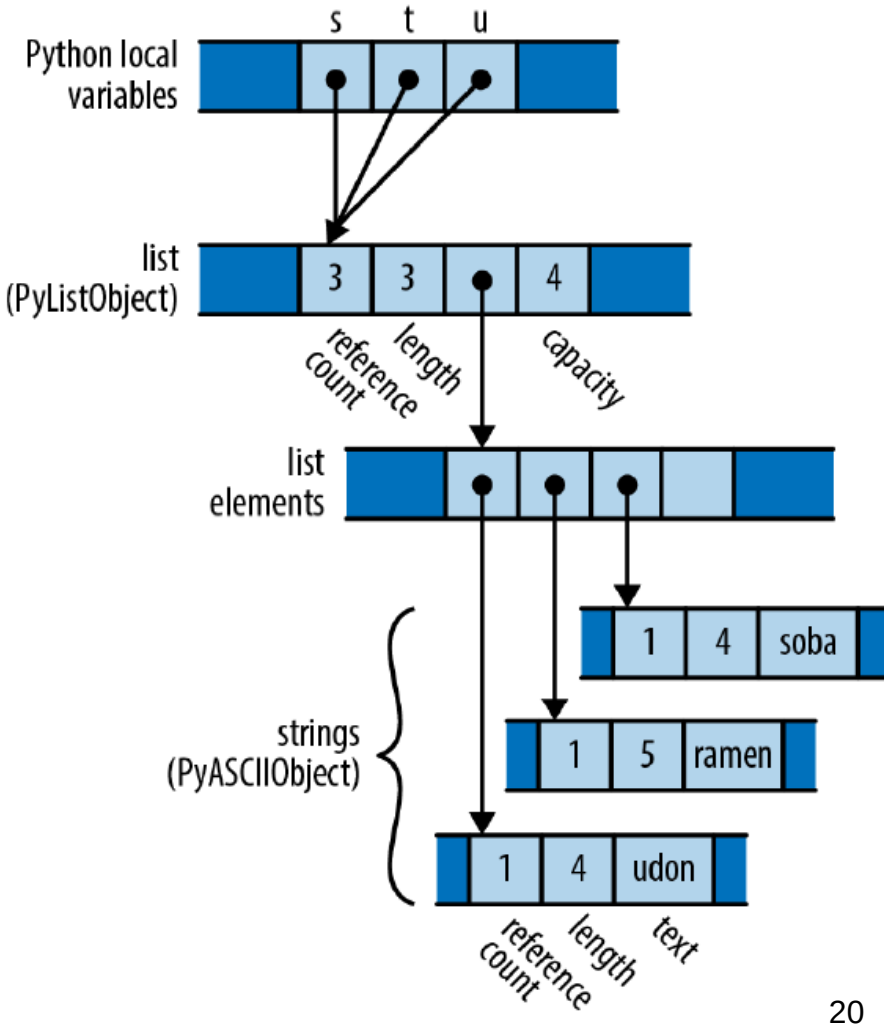
```
s = ['udon', 'ramen', 'soba']
t = s
u = s
```

- Svaki Python objekat nosi broj referenci, prateći broj vrednosti koje se trenutno odnose na njega
- Pošto samo **s** pokazuje na listu, broj referenci liste je 1; i pošto je lista jedini objekat koji ukazuje na stringove, svaki njihov broj referenci je takođe 1



POMERANJE (MOVE)

- Šta se desi u Pythonu kada **t** i **u** dobiju vrednost?
- Python implementira dodelu tako što odredište pokazuje na isti objekat kao i izvor, pri čemu se poveća broj referenci
- Očigledno je da lista sada ima 3 reference na sebe
- Dodeljivanje vrednosti je lako i jednostavno, ali je komplikovano što se broj referenci mora stalno održavati

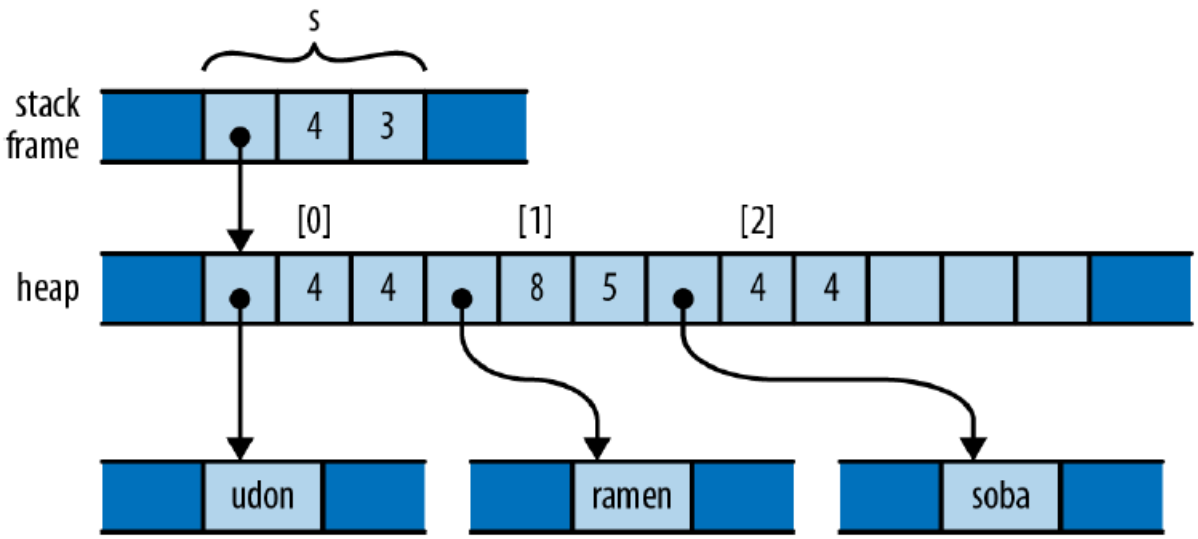


POMERANJE (MOVE)

- Kako radi C++?

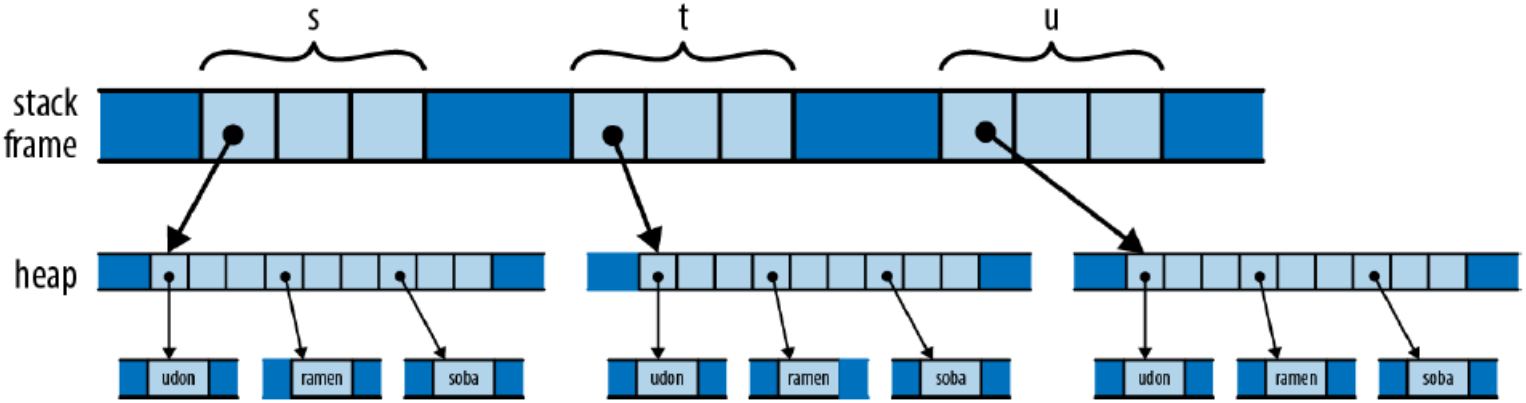
```
using namespace std;
vector<string> s = { "udon", "ramen", "soba" };
vector<string> t = s;
vector<string> u = s;
```

- Struktura memorije za **s** izgleda na sledeći način:



POMERANJE (MOVE)

- Šta se desi kada **u** i **t** dobiju vrednost?
- Za svaki string se zauzme nova vrednost u memoriji



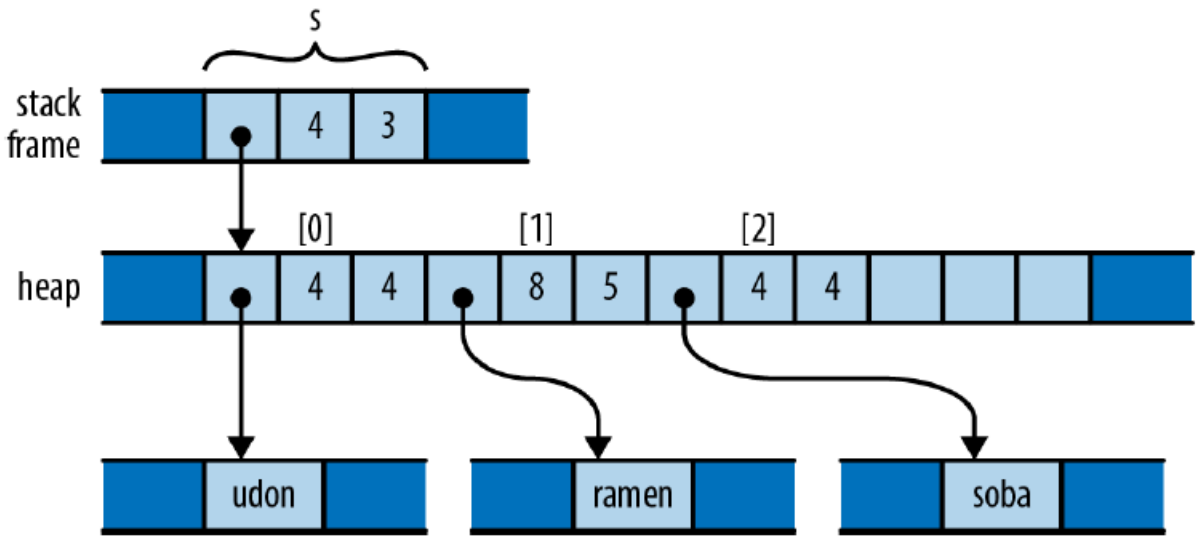
- U zavisnosti od upotrebljenih vrednosti, dodela vrednosti može da zauzme ogromnu količinu memorije i potroši veliku količinu procesorskog vremena potrebnog za alokaciju nove memorije
- Lepota svega je da se lako odlučuje koju memoriju treba osloboditi i kada treba da se ta memorija oslobodi

POMERANJE (MOVE)

- Kako bi onaj kod izgledao u Rustu?

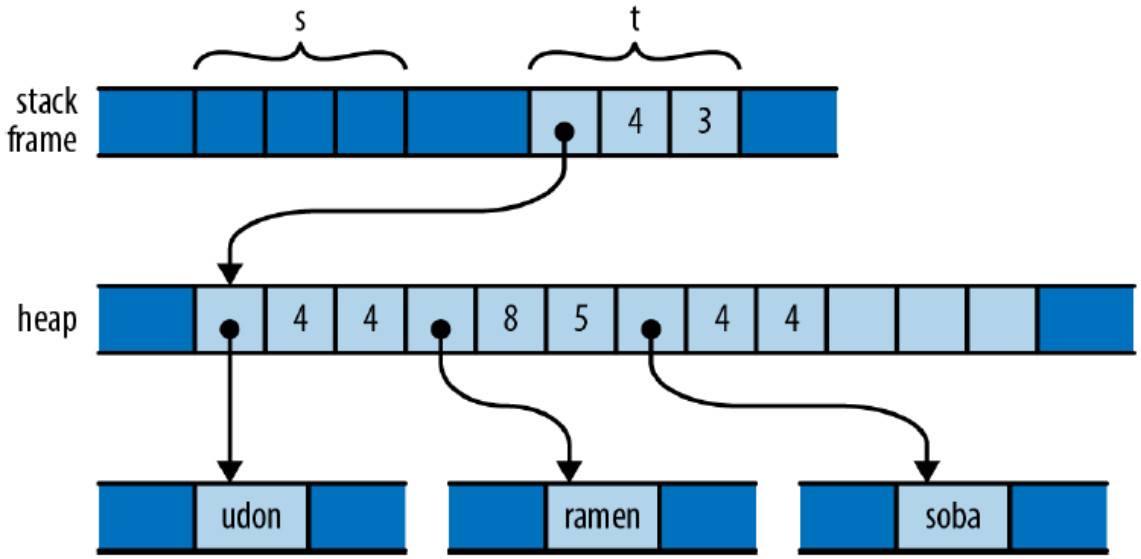
```
let s = vec!["udon".to_string(), "ramen".to_string(), "soba".to_string()];
let t = s;
let u = s;
```

- Nakon dodele vrednosti promenljivoj **s**, memorija ima sledeći izgled:



POMERANJE (MOVE)

- Nakon što promenljiva **t** dobije vrednost, izgled memorije se malo menja:



- Vlasništvo se sa promenljive **s** prebacilo na promenljivu **t** (ostaje 1 vlasnik)
- Sam bafer je ostao gde je bio, ali nema ni kopiranja u memoriji, ni održavanja liste referenci
- Promenljiva **s** se sada smatra neinicijalizovanom

POMERANJE (MOVE)

- Šta se onda desi kod dodele vrednosti promenljivoj **u**?
- Pa, dodela neinicijalizovane vrednosti promenljivoj izbacuje grešku

```
error[E0382]: use of moved value: `s`
  |
7 |     let s = vec!["udon".to_string(), "ramen".to_string(), "soba".to_string()];
  |         - move occurs because `s` has type `Vec<String>`,
  |           which does not implement the `Copy` trait
8 |     let t = s;
  |             - value moved here
9 |     let u = s;
  |             ^ value used here after move
```

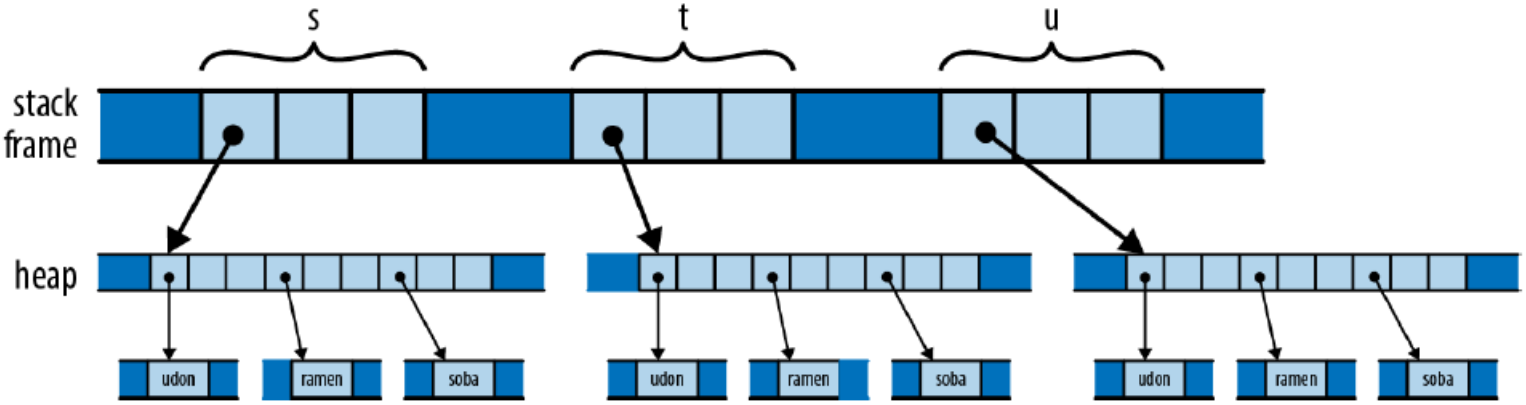
- Dodela vrednosti je jeftina kao u Pajtonu, ali je kao i u C++ dodela uvek jasna (nema nedoumica oko toga kada se dešava brisanje)
- Cena je što se kopije moraju eksplicitno tražiti

POMERANJE (MOVE)

- Da bi se napravila kopija, mora koristi sintaksa za kloniranje

```
let s = vec!["udon".to_string(), "ramen".to_string(), "soba".to_string()]
let t = s.clone();
let u = s.clone();
```

- Onda će nastati situacija slična onoj iz C++



- Gde se sada prave kopije na heapu (svaka promenljiva je vlasnik svoje vrednosti koja je zasebna u memoriji)

POMERANJE (MOVE)

- Kada se promenljivoj koja već ima vrednost dodeli nova vrednost (ili kada se desi pomeranje), Rust briše staru vrednost iz memorije

```
let mut s = "Govinda".to_string();
s = "Siddhartha".to_string(); // value "Govinda" dropped here
```

- Ako se vrednost želi sačuvati, onda mora da se desi pomeranje pre dodele vrednosti

```
let mut s = "Govinda".to_string();
let t = s;
s = "Siddhartha".to_string(); // nothing is dropped here
```

- Rust koristi mahanizam pomeranja za gotovo svaku upotrebu vrednosti
- Prenos vrednosti kao parametra funkcije dovodi do pomeranja vlasništva, tj. vlasništvo se iz lokalne promenljive prenosi na parametar funkcije
- Isto se dešava i kod kreiranje Tuple iz promenljivih, i tako dalje

POMERANJE (MOVE)

- Primeri:

```
struct Person { name: String, birth: i32 }

let mut composers = Vec::new();
composers.push(Person { name: "Palestrina".to_string(),
                        birth: 1525 });
```

- Kod kreiranja vektora, funkcija zauzima memoriju za novi vektor, i vraća sam vektor, pri čemu **composers** postaje vlasnik novog vektora
- Polje **name** nove **Person** strukture je inicijalizovano kroz povratnu vrendost metode **to_string** i struktura preuzima vlasništvo nad stringom
- Čitava struktura **Person** se prosleđuje metodi **push** koja je dodaje na kraj vektora, pri čemu vektor preuzima vlasništvo nad strukturom a samim tim posredno preuzima vlasništvo i nad stringom **name**

POMERANJE (MOVE)

- Ovo pomeranje vrednosti možda zvuči neefikasno, ali treba voditi računa o nekoliko stvari
  - Pomeranje se uvek primenjuje nad vrednošću koja je na steku (zaglavlje vektora na primer), a ne nad vlasništvom, tj. vrednošću koja se nalaz na heapu
  - Za vektore i stringove, zaglavlje se sastoji od samo tri polja (potencijalno veliki nizovi elemenata i tekstualni baferi se nalaze tamo heapu, tamo ostaju i ne pomeraju se
  - Prilikom generisanje koda, Rust kompajler dobro analizira i vidi sva ta pomeranja, tako da u praksi, mašinski kod često čuva vrednost direktno tamo gde bi i trebali biti (odmah na odgovarajućem mestu) čime se sve značajno ubrzava u praksi

POMERANJE (MOVE) UNUTAR KONTROLI TOKA

- Zbog načina na koji mehanizma pomeranja radi, mora da se vodi računa o dodeljivanju vrednosti unutar selekciji i petlji
- Ako promenljiva ima vrednost pre ulaska u selekciju i ako ona zadrži vrednost nakon provere uslova, onda se ona može koristiti i po potrebi pomeriti u obe grane

```
let x = vec![10, 20, 30];
if c {
    f(x); // ... ok to move from x here
} else {
    g(x); // ... and ok to also move from x here
}
h(x); // bad: x is uninitialized here if either path uses it
```

- Zašto dolazi do greške po izlasku iz selekcije

POMERANJE (MOVE) UNUTAR KONTROLI TOKA

- U petlji je potpuno druga priča čak i ako promenljiva zadrži vrednost nakon provere uslova

```
let x = vec![10, 20, 30];
while f() {
    g(x); // bad: x would be moved in first iteration,
          // uninitialized in second
}
```

- Osim ako promenljiva u međuvremenu ne dobije novu vrednost do sledeće iteracije

```
let mut x = vec![10, 20, 30];
while f() {
    g(x);           // move from x
    x = h();        // give x a fresh value
}
e(x);
```

POMERANJE (MOVE) I INDEKSIRANE STRUKTURE

- Šta se kad dođe do pomeranje jednog od elemenata unutar vektora?

```
// Build a vector of the strings "101", "102", ... "105"
let mut v = Vec::new();
for i in 101 .. 106 {
    v.push(i.to_string());
}

// Pull out random elements from the vector.
let third = v[2]; // error: Cannot move out of index of Vec
let fifth = v[4]; // here too
```

- Da bi se ispratilo pomeranje nekog od elemenata unutar vektora, trebalo bi da postoji vrlo sofisticiran sistem praćenja koji je elemenat inicijalizovan, a koji element nije inicijalizovan, što pobija svrhu čitavog Rust jezika

```
error[E0507]: cannot move out of index of `Vec<String>`
   |
14 |     let third = v[2];
   |                 ^^^^
   |                 |
   |                 move occurs because value has type `String`,
   |                 which does not implement the `Copy` trait
   |                 help: consider borrowing here: `&v[2]`
```

POMERANJE (MOVE) I INDEKSIRANE STRUKTURE

- U osnovi, Rust preporučuje korišćenje referenciranja kad treba da se koristi neka od vrednosti iz indeksirane strukture koja se nalazi na heapu
- Međutim, ako se baš želi izvući vrednost iz vektora, postoje metode za to koje respektuju ograničenja vektora kao tipa

```
// Build a vector of the strings "101", "102", ... "105"
let mut v = Vec::new();
for i in 101 .. 106 {
    v.push(i.to_string());
}

// 1. Pop a value off the end of the vector:
let fifth = v.pop().expect("vector empty!");
assert_eq!(fifth, "105");

// 2. Move a value out of a given index in the vector,
// and move the last element into its spot:
let second = v.swap_remove(1);
assert_eq!(second, "102");

// 3. Swap in another value for the one we're taking out:
let third = std::mem::replace(&mut v[2], "substitute".to_string());
assert_eq!(third, "103");

// Let's see what's left of our vector.
assert_eq!(v, vec!["101", "104", "substitute"]);
```

POMERANJE (MOVE) I INDEKSIRANE STRUKTURE

- Kad postoji potreba da se nešto dinamički pomera i da postoje situacije kada nešto ima ili nema vrednost, najbolje je iz osnovnog tipa preći u tip kod kojeg se može dinamički pratiti da li postoji ili ne vrednost
- **Option** je upravo jedna takav tip (enumeracija)

```
struct Person { name: Option<String>, birth: i32 }

let mut composers = Vec::new();
composers.push(Person { name: Some("Palestrina".to_string()),
                        birth: 1525 });
```

- Ovo ne može:

```
let first_name = composers[0].name;
```

- Ali zato ovo može:

```
let first_name = std::mem::replace(&mut composers[0].name, None);
assert_eq!(first_name, Some("Palestrina".to_string()));
assert_eq!(composers[0].name, None);
```

POMERANJE (MOVE) I INDEKSIRANE STRUKTURE

- Zapravo, upotreba **Option** tipa na način sa prethodnog slajda je tooliko uobičajena da za to postoji zasebna metoda, **take**

```
let first_name = composers[0].name.take();
```

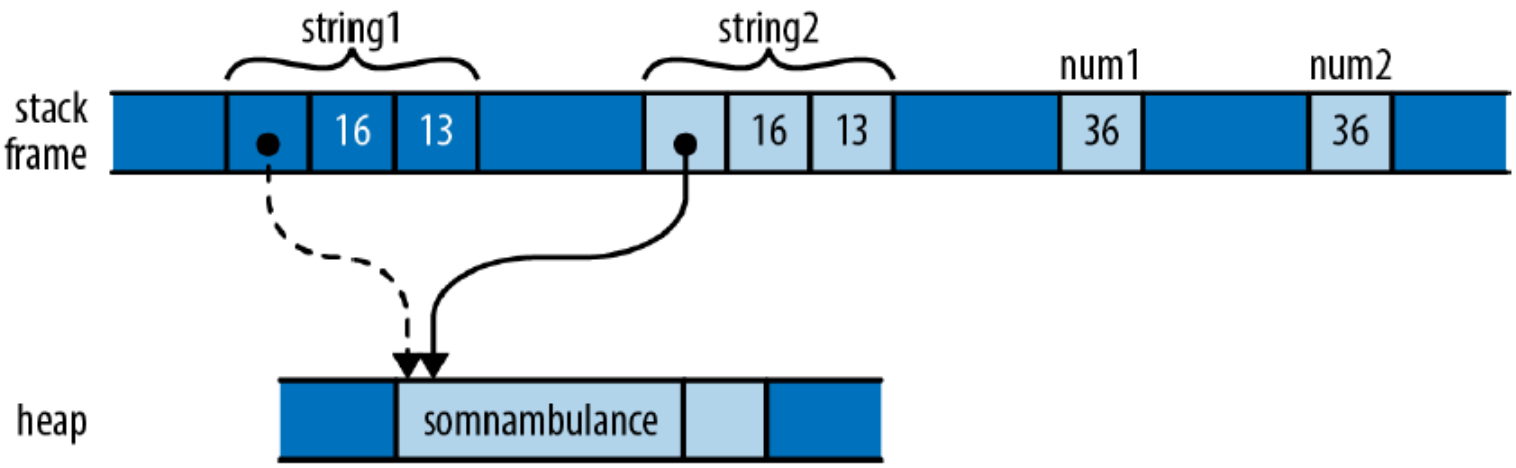
KOPIRANI TIPOVI

KOPIRANI TIPOVI

- Jednostavni tipovi poput int ne zahtevaju gimnastiku kao i kompleksniji tipovi

```
let string1 = "somnambulance".to_string();
let string2 = string1;

let num1: i32 = 36;
let num2 = num1;
```

- Kako to izgleda u memoriji



KOPIRANI TIPOVI

- Nema pomeranja Kopiranih tipova
- Dodeljivanje vrednosti jedne promenljive kopiranog tipa drugoj promenljivoj, vrši kopiranja te vrednost (jer su vrednosti na steku)
- Isto važi i kada se kopirani tipovi prenose kao parametri funkcija
- U ovo spadaju int tipovi, karakteri, tipovi sa pokretnim zarezom, boolean tip, n-torke, kao i nizovi fiksne dužine napravljeni od kopiranih tipova
- Bilo koji tip za koji treba da uradi nešto posebno kada se vrednost briše, ne može biti Kopi tip:
  - vektor treba da oslobodi svoje elemente,
  - File treba da zatvori svoj hendler datoteke,
  - MutexGuard treba da otključa svoj muteks, i tako dalje

KOPIRANI TIPOVI

- Šta kada pravite svoje tipove? Strukture i enumeracije nisu Kopirani tipovi

```
struct Label { number: u32 }

fn print(l: Label) { println!("STAMP: {}", l.number); }

let l = Label { number: 3 };
print(l);
println!("My label number is: {}", l.number);
```

```
error: borrow of moved value: `l`
   |
10 |     let l = Label { number: 3 };
   |         - move occurs because `l` has type `main::Label`,
   |           which does not implement the `Copy` trait
11 |     print(l);
   |           - value moved here
12 |     println!("My label number is: {}", l.number);
   |                                        ^^^^^^^^
   |                   value borrowed here after move
```

KOPIRANI TIPOVI

- Ako su sva polja u novom tipu definisanom od strane korisnika Kopirani tipovi, onda i taj novi tip može biti Kopirani tip samo treba to eksplicitno navesti pomoću atributa **#[derive(Copy, Clone)]** iznad definicije novog tipa

```
#[derive(Copy, Clone)]
struct Label { number: u32 }
```

- Ovo ne radi ako nisu sva polja Kopirani tipovi

```
#[derive(Copy, Clone)]
struct StringLabel { name: String }
```

```
error[E0204]: the trait `Copy` may not be implemented for this type
 --> ownership_string_label.rs:7:10
  |
7 | #[derive(Copy, Clone)]
  |          ^^^^
8 | struct StringLabel { name: String }
  |                      ------------ this field does not implement `Copy`
```

KOPIRANI TIPOVI

- Korisnički definisani tipovi nisu automatski Kopirani tipovi, čak ni pod pretpostavkom da ispunjavaju uslove
- Da li je tip Kopirani li ne, ima veliki uticaj na to kako je kodu dozvoljeno da ga koristi:
  - Kopirani tipovi su fleksibilniji, pošto dodeljivanje i povezane operacije ne ostavljaju original neinicijalizovanim
  - Ali sa druge strane: kopirani tipovi su veoma ograničeni u tome koje tipove mogu da sadrže, dok tipovi koji nisu Kopirani mogu da koriste alokaciju memorije (heapa) i poseduju razne druge vrste resursa
- Dakle, pravljenje Kopiranog tipa predstavlja ozbiljnu obavezu od strane onoga koji ga implementira: ako je kasnije neophodno da se promeni u non-Copy, verovatno će morati da se prilagodi veći deo koda koji ga koristi

DELJENO VLASNIŠTVO (Rc I Arc)

DELJENO VLASNIŠTVO

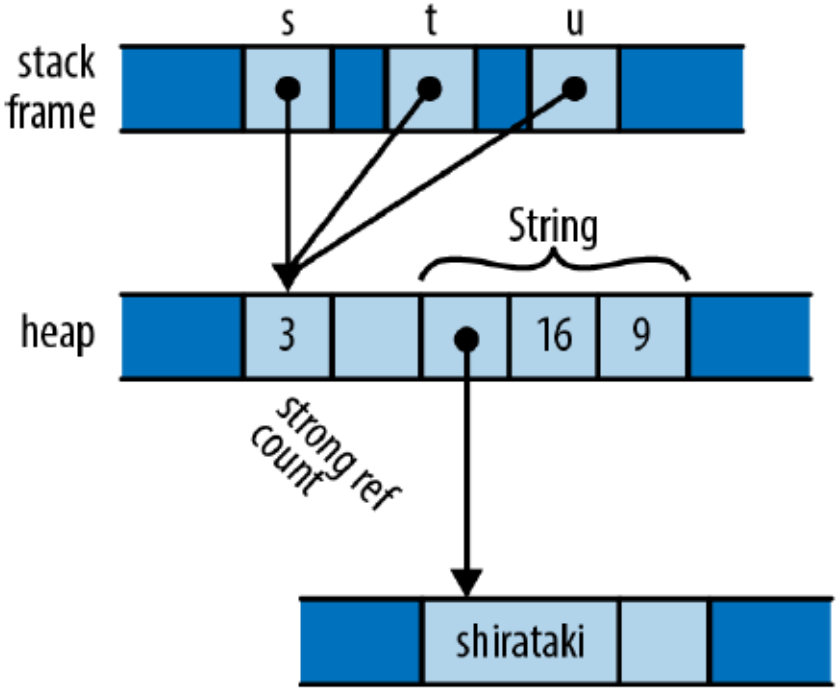
- Rust omogućuje deljeno vlasništvo kroz pokazivače kod kojih se broji broj referenci na datu adresu, **Rc** i **Arc**, koji su u Rust smislu potpuno sigurni za upotrebu
- Ova 2 tipa su vrlo slična jedino što je **Arc** sigurno deliti između više niti izvršavanja (siguran je i u smislu paralelnog programiranja), dok je **Rc** siguran samo u slučaju izvršavanja unutar jedne niti
  - **Arc** je skraćenica od **Atomic Referebce Count** kako bi se sugerisalo da je u pitanju atomična operacija
  - Samim tim mehanizmi u pozadini ova dva tipa se razlikuju po kompleksnosti i brzini gde **Rc** radi brže od **Arc**
  - Rust brine o tome da se **Rc** ne koristi u situacijama u kojima bi trebalo koristiti **Arc**
- Operacije potrebne za korišćenje **Rc** i **Arc** nalaze se u odgovarajućim imenskim prostorima

DELJENO VLASNIŠTVO

- Primer rada sa **Rc**

```
use std::rc::Rc;

// Rust can infer all these types; written out for clarity
let s: Rc<String> = Rc::new("shirataki".to_string());
let t: Rc<String> = s.clone();
let u: Rc<String> = s.clone();
```

- Za bilo koji tip **T**, vrednost **Rc<T>** je pokazivač na **T** koji je negde na heapu
- Vrednost **Rc<T>** pored pokazivača sadrži i broj referenci na memoriju



DELJENO VLASNIŠTVO

- Kloniranje **Rc<T>** ne dovodi do kopiranja vrednosti **T** već samo do stvaranja novog pokazivača i uvećenja broja referenci na **T**
- Svaki od **Rc<String>** pokazivača pokazuje na istu memoriju na heapu koja čuva pokazivač na sam string i broj referenci
- Metode za rad sa stringovima su dostupne

```
assert!(s.contains("shira"));
assert_eq!(t.find("taki"), Some(5));
println!("{} are quite chewy, almost bouncy, but lack flavor", u);
```

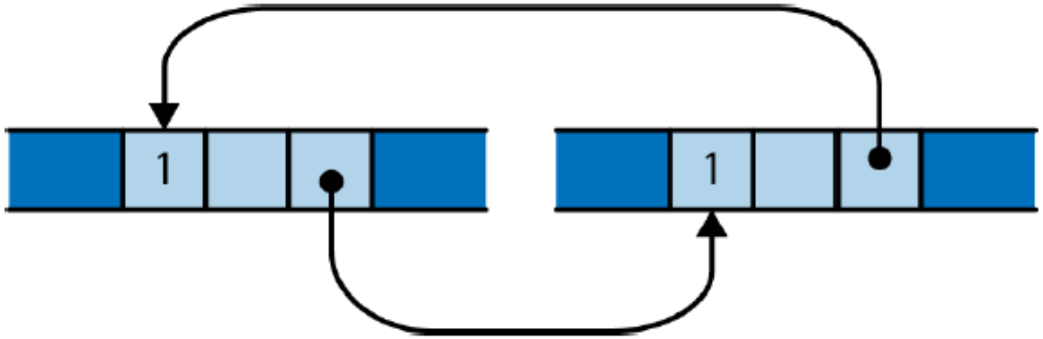
- Ali je sama vrednost na koju Rc pokazuje nepromenljiva, tako da ovo nije dozvoljeno:

```
s.push_str(" noodles");
```

```
error: cannot borrow data in an `Rc` as mutable
  --> ownership/ownership_rc_mutability.rs:13:5
   |
13 |     s.push_str(" noodles");
   |     ^ cannot borrow as mutable
   |
```

DELJENO VLASNIŠTVO

- Rust brine o tome da nijedna vrednost nije simultano deljena a da joj se može promeniti vrednost
- Pošto **Rc** podrazumeva deljenje, onda je to znači i nepromenljivost
- Treba voditi računa o mogućnostim cirkularnog referenciranja, jer će to dovesti do toga da se vrednosti nikada ne oslobode



- Iako je mogućnost ovoga vrlo mala u Rustu, ipak se može desiti ako ne pazite, jer Rust dozvoljava nešto što se zove unutrašnja promenljivost (**interior mutability**), o čemu ćemo možda pričati

REFERENCE I POZAJMLJIVANJE

REFERENCE

- Sve do sada, reference koja sam pokazao i Box<T> i reference na String i Vec su bile vlasničke reference (preuzimale su vlasništvo nad podacima)
- Rust ima i ne-vlasničke (**non-owning**) pokazivače, tj. reference koje ne utiču na životni vek onoga na šta se pokazuje
- U Rustu, ove reference ne smeju da nadžive vrednost koju referišu i to mora biti eksplicitno vidljivo iz samog koda
- Zbog toga se kreiranje reference u Rustu naziva pozajmljivanje (**borrowing**), jer se vlasništvo nad vrednošću samo pozajmljuje i eventualno na kraju vraća vlasniku
- Sve do Rusta ovaj mehanizam oko običnih referenca (adresa na memoriju) je postojao samo u istraživačkim radovima i čini Rust jedinstvenim, ali izrazito moćnim

REFERENCE – PRIMER

- Primer Hash mape
- Mapira string na vektor stringova

```
use std::collections::HashMap;

type Table = HashMap<String, Vec<String>>;
```

- Izlistavanje sadržaja mape:

```
fn show(table: Table) {
    for (artist, works) in table {
        println!("works by {}:", artist);
        for work in works {
            println!("  {}", work);
        }
    }
}
```

REFERENCE – PRIMER

- Primer Hash mape
- Izgradnja mape

```
fn main() {
    let mut table = Table::new();
    table.insert("Gesualdo".to_string(),
                 vec!["many madrigals".to_string(),
                      "Tenebrae Responsoria".to_string()]);
    table.insert("Caravaggio".to_string(),
                 vec!["The Musicians".to_string(),
                      "The Calling of St. Matthew".to_string()]);
    table.insert("Cellini".to_string(),
                 vec!["Perseus with the head of Medusa".to_string(),
                      "a salt cellar".to_string()]);

    show(table);
}
```

REFERENCE – PRIMER

- To iz nekog razloga radi:

```
$ cargo run
     Running `/home/jimb/rust/book/fragments/target/debug/fragments`
works by Gesualdo:
  many madrigals
  Tenebrae Responsoria
works by Cellini:
  Perseus with the head of Medusa
  a salt cellar
works by Caravaggio:
  The Musicians
  The Calling of St. Matthew
$
```

- Zašto radi?

REFERENCE – PRIMER

- Šta ako pokušamo nešto nakon što smo izlistali mapu?

```
...
show(table);
assert_eq!(table["Gesualdo"][0], "many madrigals");
```

- Da li to može? Kako radi pomerenje?

```
error: borrow of moved value: `table`
   |
20 |     let mut table = Table::new();
   |         --------- move occurs because `table` has type
   |                   `HashMap<String, Vec<String>>`,
   |                   which does not implement the `Copy` trait
...
31 |     show(table);
   |          ----- value moved here
32 |     assert_eq!(table["Gesualdo"][0], "many madrigals");
   |                ^^^^^ value borrowed here after move
```

- Gde se sve desi pomeranje?

REFERENCE – PRIMER

- Pravilan način da se to reši je primenom pozajmljivanja, tj. referenci
- Sama sintaksa ostaje ista

```
fn show(table: &Table) {
    for (artist, works) in table {
        println!("works by {}:", artist);
        for work in works {
            println!("  {}", work);
        }
    }
}
```

- Sada poziv funkcije izgleda:

```
show(&table);
```

- Umesto da prosledimo vrednost prosleđujemo deljenu referencu, tj. pozajmljujemo je funkciji

REFERENCE – PRIMER

- Spoljašnja petlja u **show** funkciji sada pozajmljuje vlasništvo i ne konzumira heš mapu
- Iteracija nad deljenom referencom HashMap tipa proizvodi deljene reference na svako polje iz mape
  - **artist** više nije tipa **String**, već tipa **&String**
  - **works** više nije tipa **Vec<String>**, već tipa **&Vec<String>**
- Na sličan način se menja i unutrašnja petlja
- Iteracija nad deljenom referencom koja je referenca nad vektorom proizvodi deljene reference nad elementima vektora
  - **work** je sada deljena referenca tipa **&String**
- Ni jednom nije došlo do promene vlasništva

```
fn show(table: &Table) {
    for (artist, works) in table {
        println!("works by {}:", artist);
        for work in works {
            println!("  {}", work);
        }
    }
}
```

VRSTE REFERENCI

- Postoje 2 vrste referenci: deljene referenci i mutabilne reference
- Deljena referenca (**shared reference**) dozvoljava samo čitanje, ali ne i izmene vrednosti koju referencira
- Ne postoji ograničenje na broj deljenih referenci, te može biti onoliko deljenih (zajedničkih) referenci na određenu vrednost u isto vreme koliko se želi
- Izraz **&e** generiše deljenu referencu na vrednost e;
- Ako je **e** tipa **T**, onda je **&e** tipa **&T**
- Deljene reference su Kopirani tip (**Copy**)

VRSTE REFERENCI

- Promenljiva referenca (**mutable reference**) omogućuju čitanje i menjanje vrednosti koju referenciraju
- Međutim, u datom slučaju u isto vreme ne sme postojati ni jedna druga aktivna referenca (bilo koje vrste) na tu vrednost
- Izraz **&mut e** generiše promenljivu referencu na vrednost **e**
- Ako je **e** tipa **T**, onda je **&mut e** tipa **&mut T**
- Promenljive reference nisu Kopiranog tipa (Copy)
- Na deljene i promenljive reference se može gledati kao na mehanizam koji dozvoljava višestruka prava čitanja ali samo jedno pravo pisanja
- Ovo pravilo pokriva i vlasnika vrednosti, ne samo pozajmljene reference, tako da dok god postoje aktivne reference, čak ni vlasnik ne može da menja vrednost promenljive
- Niko ne može da modifikuje **table** dok god **show** ne završi sa njom

VRSTE REFERENCI

- Ako bi se želela napisati funkcija koja menja promenljivu **table** tako što alfabetski sortira radove svakog umetnika, onda se mora koristiti promenljiva referenca

```
fn sort_works(table: &mut Table) {
    for (_artist, works) in table {
        works.sort();
    }
}
```

- Promenljiva referenca se mora koristiti i u pozivu funkcije

```
sort_works(&mut table);
```

- Ovo omogućuje funkciji **sort_works** da pozajmi **table** sa mogućnošću promene njegovih vrednosti (**mutable borrow**), tj. može i da čita i da menja strukturu heš mape
- Prenos parametara funkcije bez pozajmljivanja je zapravo prenos po vrednosti, dok je sa pozajmljivanjem prenos po referenci

RAD SA REFERENCAMA

- Reference se u Rustu kreiraju eksplicitno primenom **& operatora** – on mora stojati i u definiciji odgovarajuće promenljive i ispred promenljive na koju se pravi referenca
- Referenca se deferencira eksplicitno primenom *** operatora**

```
// Back to Rust code from this point onward.
let x = 10;
let r = &x;              // &x is a shared reference to x
assert!(*r == 10);       // explicitly dereference r

let mut y = 32;
let m = &mut y;          // &mut y is a mutable reference to y
*m += 32;                // explicitly dereference m to set y's value
assert!(*m == 64);       // and to see y's new value
```

RAD SA REFERENCAMA

- Pošto se reference toliko često koriste **operator .** vrši implicitno derefenciranje svog levog operanda
- Rust sam dedukuje na osnovu okolnih tipova da li treba da izvrši dereferenciranje

```
struct Anime { name: &'static str, bechdel_pass: bool };
let aria = Anime { name: "Aria: The Animation", bechdel_pass: true };
let anime_ref = &aria;
assert_eq!(anime_ref.name, "Aria: The Animation");

// Equivalent to the above, but with the dereference written out:
assert_eq!((*anime_ref).name, "Aria: The Animation");
```

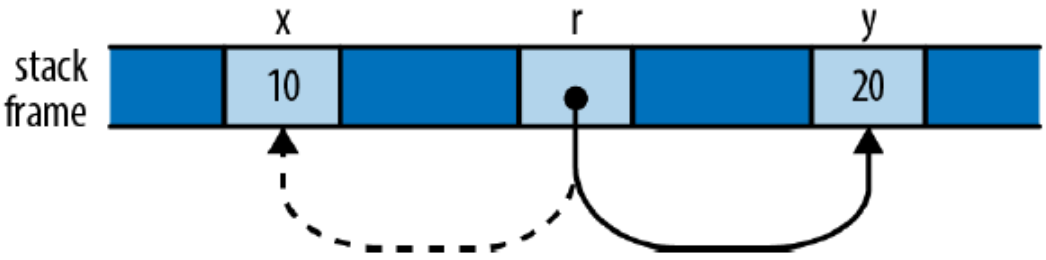
- **Operator .** vrši implicitno pozajmljivanje reference svom levom operandu ako je potrebno za poziv metode

```
let mut v = vec![1973, 1968];
v.sort();           // implicitly borrows a mutable reference to v
(&mut v).sort();    // equivalent, but more verbose
```

RAD SA REFERENCAMA

- Dodeljivanje nove reference promenljivoj dovodi do toga da promenljiva pokazuje negde drugde
- Ovo se razlikuje od C++, gde jednom kada se promenljivoj dodeli referenca, ne može se promeniti to našta pokazuje

```
let x = 10;
let y = 20;
let mut r = &x;

if b { r = &y; }

assert!(*r == 10 || *r == 20);
```



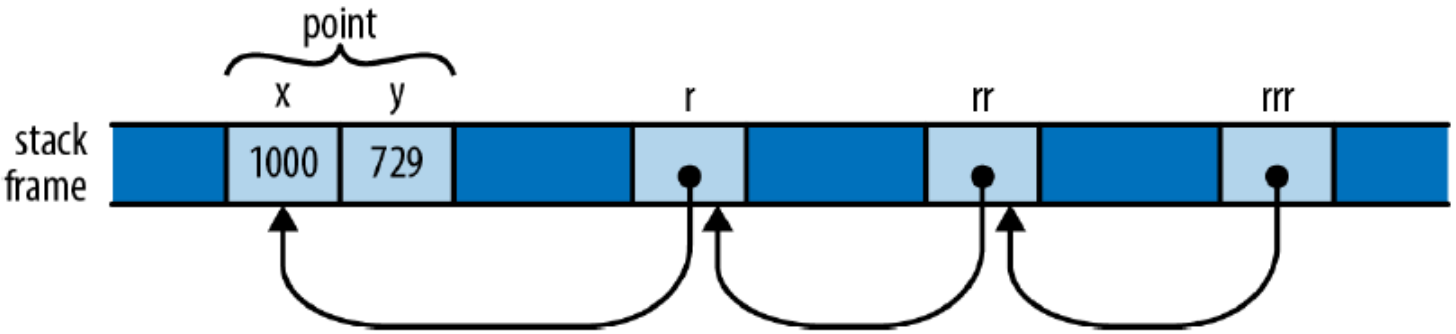
RAD SA REFERENCAMA

- Rust dozvoljava referencu na referencu

```
struct Point { x: i32, y: i32 }
let point = Point { x: 1000, y: 729 };
let r: &Point = &point;
let rr: &&Point = &r;
let rrr: &&&Point = &rr;
```

- Operator . može da implicitno dereferencira reference sve dok ne dođe do vrednosti

```
assert_eq!(rrr.y, 729);
```



RAD SA REFERENCAMA

- Relacioni operatori u Rustu takođe vrše dereferenciranje do vrednosti, pod uslovom da su reference koje se porede istog nivoa

```
let x = 10;
let y = 10;

let rx = &x;
let ry = &y;

let rrx = &rx;
let rry = &ry;

assert!(rrx <= rry);
assert!(rrx == rry);
```

- Takođe, reference moraju da pokazuju na vrednosti istog tipa

```
assert!(rx == rrx);    // error: type mismatch: `&i32` vs `&&i32`
assert!(rx == *rrx);   // this is okay
```

RAD SA REFERENCAMA

- Ako se žele uporediti adrese na koje reference ukazuju, koristi se metoda **std::ptr::eq**

```
assert!(rx == ry);                // their referents are equal
assert!(!std::ptr::eq(rx, ry)); // but occupy different addresses
```

RAD SA REFERENCAMA

- Rust reference nisu nikada **null** (null zapravo ne postoji u Rustu)
- Referenca nema inicijalnu vrednost (mora da se inicijalizuje) i ne može se konvertovati int u referencu (tako da referenca ne može imati nulu za vrednost)
- U Rustu ako se želi da neka promenljivost može imati vrednost ili ne, koristi se se tip **Option<&T>**
- Ako postoji vrednost, onda je vrednost ovog tipa **Some(r)** gde je **r** jedna od vrednosti koju može imati **&T**, a **None** ako nema vrednosti
- Na nivou mašinsko jezika, **None** je isto što i **null**

RAD SA REFERENCAMA

- U Rustu je moguće referencirati čak i vrednost izraza

```
fn factorial(n: usize) -> usize {
    (1..n+1).product()
}
let r = &factorial(6);
// Arithmetic operators can see through one level of references.
assert_eq!(r + &1009, 1729);
```

- U ovakvim situacijama Rust jednostavno kreira anonimnu promenljivu u koju smešta vrednost izraza i na koju pravi referencu
- Životni vek privremene promenljive zavisi od njene upotrebe
  - Ako se referenca odmah dodeli promenljivoj u **let** iskazu, onda privremena promenljiva traje koliko i data promenljiva (promenljiva r)
  - U suprotnom, živi onoliko dugo koliko i iskaz u kojem se koristi

RAD SA REFERENCAMA

- Zasebna vrsta referenci su debele reference (**fat pointers**)
- One pored adrese sadrže i neki dodatni podatak
- **Slice** je ova vrste reference
- Postoje još neke takve reference, kao što su **trait objects** – vrednost koja implementira određeno ponašanje (**trait**)
  - Ova referenca sadrži adresu i pokazivač na metodu koja implementira ponašanje odgovarajuće za datu vrednost (pokretanje odgovarajuće **trait** metode)
- Debeli pokazivači se, osim tog dodatka, ponašaju potpuno isto kao i reference

OGRANIČENJA U RADU SA REFERENCAMA

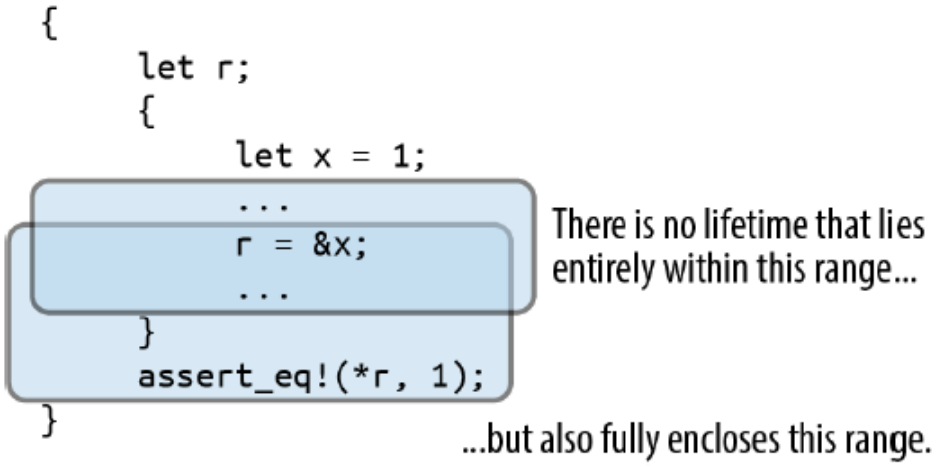
- Ne može se uzeti referenca na lokalnu promenljivu i da se pri tome iznese izvan njenog opsega

```
{
    let r;
    {
        let x = 1;
        r = &x;
    }
    assert_eq!(*r, 1);  // bad: reads memory `x` used to occupy
}
```

```
error: `x` does not live long enough
  --> references_dangling.rs:8:5
   |
7  |         r = &x;
   |             ^^ borrowed value does not live long enough
8  |     }
   |     - `x` dropped here while still borrowed
9  |     assert_eq!(*r, 1);  // bad: reads memory `x` used to occupy
10 | }
```

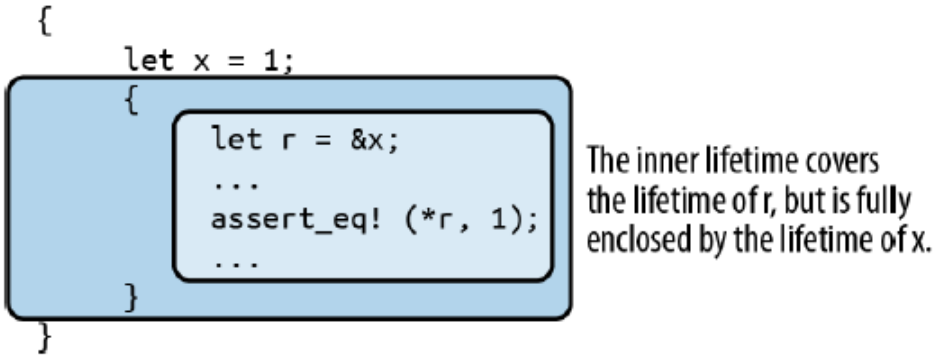
OGRANIČENJA U RADU SA REFERENCAMA

- Rust svakom tipu i promenljivi definiše životni vek
- Kad se napravi referenca, kompajler provera da li se njen životni vek poklapa sa životnim vekom promenljive koju referencira i da li se ona može bezbedno koristiti
- Referenca na neku promenljivu **ne sme da živi duže od promenljive** na koju pokazuje, obrnuto je dozvoljeno
- Ako se referenca nalazi u nekoj promenljivoj, onda ta **promenljiva mora čitavog svog životnog veka pokazivati na nešto**



OGRANIČENJA U RADU SA REFERENCAMA

- Ako ta 2 pravila nisu zadovoljena (prvo obuhvata drugo), nije sigurna upotreba referenci i Rust kompajler se buni
- Rešenje?



- Ovo pravilo se prirodno nameće i u situacijama kada se referencira deo neke veće celine (isečak iz niza, ili kada je referenca deo strukture)

```
let v = vec![1, 2, 3];
let r = &v[1];
```

OGRANIČENJA U RADU SA REFERENCAMA

- Što se funkcija i rada sa referencama tiče, referenca koja se prosledi funkciji će imati životni vek funkcije, tako da mora da se pazi da njen životni vek ne nadživi funkciju
- Takođe, mora da se vodi računa i kada se referenca vraća kao rezultat funkcije, naročito ako je ona deo neke celine prenesene kao referenca u funkciji
- Rust podrazumeva da su reference istog životnog veka, ako se jedna referenca prosleđuje kao argument funkcije a druga referenca vreća kao rezultat funkcije

```
// v should have at least one element.
fn smallest(v: &[i32]) -> &i32 {
    let mut s = &v[0];
    for r in &v[1..] {
        if *r < *s { s = r; }
    }
    s
}
```

OGRANIČENJA U RADU SA REFERENCAMA

- Pretpostaviti da se minimum niza (smallest) poziva na sledeći način:

```
let s;
{
    let parabola = [9, 4, 1, 0, 1, 4, 9];
    s = smallest(&parabola);
}
assert_eq!(*s, 0); // bad: points to element of dropped array
```

- Dolazi do sledeće greške

```
error: `parabola` does not live long enough
  --> references_lifetimes_propagated.rs:12:5
   |
11 |         s = smallest(&parabola);
   |                      -------- borrow occurs here
12 |     }
   |     ^ `parabola` dropped here while still borrowed
13 |     assert_eq!(*s, 0); // bad: points to element of dropped array
   |                - borrowed value needs to live until here
14 | }
```

OGRANIČENJA U RADU SA REFERENCAMA

- Problem rešava pomeranje **s** tako da njen životni vek odgovara životnom veku reference koja se prosleđuje funkciji, **parabola**

```
{
    let parabola = [9, 4, 1, 0, 1, 4, 9];
    let s = smallest(&parabola);
    assert_eq!(*s, 0); // fine: parabola still alive
}
```

OGRANIČENJA U RADU SA REFERENCAMA

- Moguće je uspostaviti vezu između životnih vekova različitih promenljivih kroz anotacije životnog veka
- Ove anotacije ne menjaju trajanje bilo koje reference, već opisuju odnose između životnih vekova više međusobnih referenci (bez uticaja na životna vek)
- Funkcije mogu prihvatiti reference sa bilo kojim životnim vekom ako se koristi generička specifikacija životnog veka
- Anotacija životnog veka imaju sledeću sintaksu:
  - imena parametara trajanja moraju da počinju apostrofom (') i obično su sva mala slova i vrlo kratki, kao generički tipovi
  - **’a** se uglavnom koristi za prvu anotaciju životnog veka, ’b za drugu itd.
  - Anotacija se stavlja nakon **& reference**, koristeći razmak za odvajanje anotacije od tipa reference

OGRANIČENJA U RADU SA REFERENCAMA

- Evo nekoliko primera: referenca na i32 bez anotacije životnog veka, referenca na i32 koji ima anotaciju životnog veka pod nazivom 'a, i promenljiva referenca na i32 koji takođe ima životni vek ’a

```
  &i32        // a reference
&'a i32     // a reference with an explicit lifetime
&'a mut i32 // a mutable reference with an explicit lifetime
```

- Anotacije ovako samostalno nemaju nekog smisla
- Smisao dobijaju tek kada se želi iskazati međusobni odnos životnih vekova različitih funkcija

OGRANIČENJA U RADU SA REFERENCAMA

- U zaglavlju funkcije anotacija životnog veka sa u samom nazivu stavlja unutar **<>** između naziva funkcije i liste parametara, kao za generičke tipove

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

- Ovim je sada rečeno Rustu da za neki životni vek 'a, funkcija uzima dva parametra, od kojih su oba isečci stringa koji žive najmanje koliko životni vek ’a
- Potpis funkcije takođe govori Rustu da će isečak stringa vraćen iz funkcije živeti najmanje onoliko koliko je životni vek ’a

OGRANIČENJA U RADU SA REFERENCAMA

- U praksi, to znači da je životni vek reference koju vraća **longest** funkcija isti kao i najkraći od životnih vekova vrednosti na koje se referenciraju argumenti funkcije
- Ovi odnosi su ono što želimo da Rust koristi kada analizira ovaj kod i ovo pre svega predstavlja poruku kompajleru da zna kako da tumači naš kod
- Kada koristimo anotaciju životnog veka u potpisu funkcije, ne menjamo životni vek bilo prosleđenih parametara ili vraćene vrednosti
- Zapravo olakšavamo kompajleru tako što mu kažemo kojih ograničenja mehanizam pozajmljivanja treba da se pridržava i koje vrednosti treba da odbije ako se ne pridržavaju ovih ograničenja
- Ne zanima nas dužina životnog veka, već da je odnos životnih vekova parametara i rezultata takav da zadovoljava ovaj odnos
- Ovo olakšava kompajleru da pored kontrole, daje i smislene poruke

OGRANIČENJA U RADU SA REFERENCAMA

- Primer sa promenljivama različitog životnog veka

```
fn main() {
   let string1 = String::from("long string is long");

   {
       let string2 = String::from("xyz");
       let result = longest(string1.as_str(), string2.as_str());
       println!("The longest string is {}", result);
   }
}
```

- Da li je ovo dobro? Da li zadovoljava zaglavlje i potpis funkcije?

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

OGRANIČENJA U RADU SA REFERENCAMA

- A sad sa malom izmenom:

```
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {}", result);
}
```

- Da li je ovo dobro? Da li zadovoljava zaglavlje i potpis funkcije?

OGRANIČENJA U RADU SA REFERENCAMA

- Dobija se sledeća greška:

```
 $ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0597]: `string2` does not live long enough
 --> src/main.rs:6:44
  |
6 |         result = longest(string1.as_str(), string2.as_str());
  |                                            ^^^^^^^^^^^^^^^^ borrowed value does not live long enough
7 |     }
  |     - `string2` dropped here while still borrowed
8 |     println!("The longest string is {}", result);
  |                                          ------ borrow later used here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `chapter10` due to previous error
```

- Rezultat živi duže od string2 čiji je životni vek najkraći
- Mada u samom primeru print bi i dalje radio jer je string1 duži od string2 (ali pravilo je pravilo i kompajler nije baš toliko sofisticiran!)

OGRANIČENJA U RADU SA REFERENCAMA

- Ne mora se definisati anotacija životnog veka za sve parametre (zavisi od konteksta), kao ni za rezultat
- Na primer:

```
fn longest<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

- **y** nema anotaciju životnog veka, jer ovde postoji veza samo između parametra **x** i povratne vrednosti funkcije
- Životni vek povratne vrednost funkcije treba povezati sa životnim vekom jednog od parametara, u suprotnom i nema smisla uvoditi anotaciju životnog veka, zašto?

```
fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```

OGRANIČENJA U RADU SA REFERENCAMA

- Životni vek se može definisati i za strukture

```
struct S<'a> {
    x: &'a i32,
    y: &'a i32
}
```

- To samo kaže da životni vek strukture mora da traje koliko i životni vek onoga na šta se polje iz strukture referiše, tj. životni vek bilo koje reference uskladištene u **r** mora da bude najmanje dužine **'a**, a **'a** mora da nadživi životni vek onoga u šta se uskladišti **S**
- Može biti problem u primeru na sledećem slajdu

OGRANIČENJA U RADU SA REFERENCAMA

- Moguće je napraviti i da se različiti životni vekovi različito anotiraju

```
struct S<'a, 'b> {
    x: &'a i32,
    y: &'b i32
}
```

- Sa ovom anotacijom, polja **s.a** i **s.b** imaju različite životne vekove i mogu da traju odvojeno

```
let x = 10;
let r;
{
    let y = 20;
    {
        let s = S { x: &x, y: &y };
        r = s.x;
    }
}
println!("{}", r);
```

OGRANIČENJA U RADU SA REFERENCAMA

- Različite anotacije životnog veka se mogu primeniti i nad funkcijama
- Time se smanjuje ograničenje pozivaoca i daje mu se veća fleksibilnost u životnim vekovima
- Stroga anotacija životnog veka:

```
fn f<'a>(r: &'a i32, s: &'a i32) -> &'a i32 { r } // perhaps too tight
```

- Malo relaksiranija anotacija životnog veka:

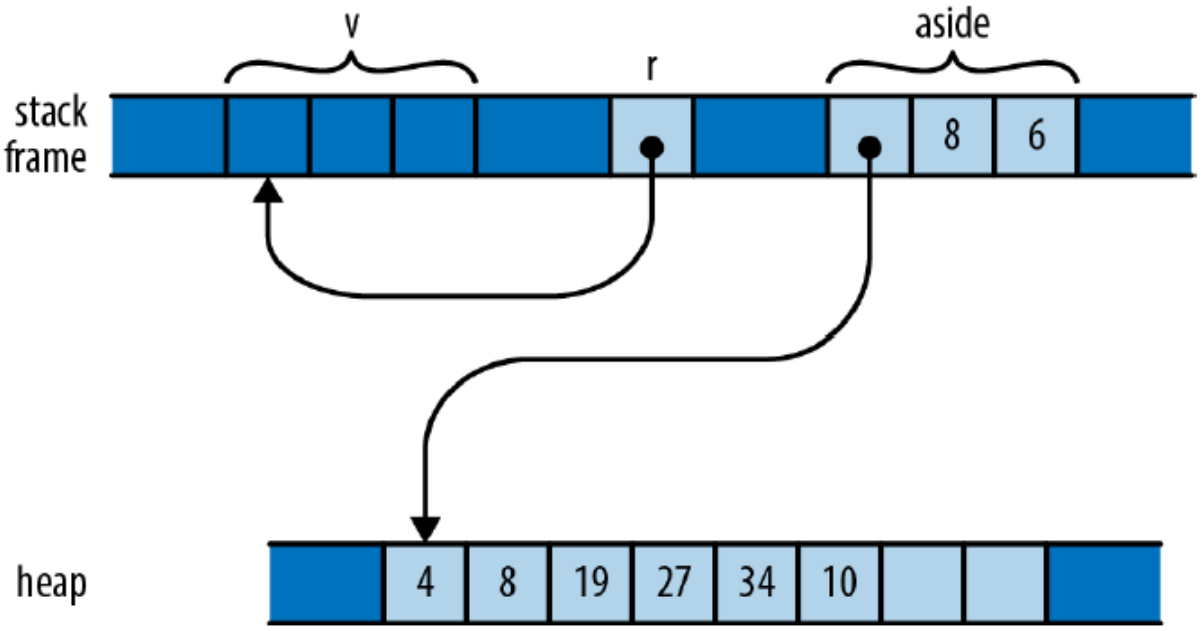
```
fn f<'a, 'b>(r: &'a i32, s: &'b i32) -> &'a i32 { r } // looser
```

OGRANIČENJA U RADU SA REFERENCAMA

- Rust neće dozvoliti pomeranja (**move**) nad resursom nad kojim je pre toga napravljeno pozajmljivanje

```
let v = vec![4, 8, 19, 27, 34, 10];
let r = &v;
let aside = v;  // move vector to aside
r[0];           // bad: uses `v`, which is now uninitialized
```

- U ovom slučaju se dešava sledeća stvar u memoriji



OGRANIČENJA U RADU SA REFERENCAMA

- Stvara sledeću grešku

```
error[E0505]: cannot move out of `v` because it is borrowed
  --> references_sharing_vs_mutation_1.rs:10:9
   |
9  |     let r = &v;
   |             - borrow of `v` occurs here
10 |     let aside = v;  // move vector to aside
   |         ^^^^^ move out of `v` occurs here
```

- Malo promene dovodi kod u red

```
let v = vec![4, 8, 19, 27, 34, 10];
{
    let r = &v;
    r[0];       // ok: vector is still there
}
let aside = v;
```

OGRANIČENJA U RADU SA REFERENCAMA

- Rust će dozvoliti promenljivo pozajmljivanje referecne na vektor (borrow a mutable reference to the vector) i pozajmljivanje deljenje reference na njegove elemente (shared reference to its elements) pod uslovom da se njihovi životni vekovi ne poklope
- Npr.

```
fn extend(vec: &mut Vec<f64>, slice: &[f64]) {
    for elt in slice {
        vec.push(*elt);
    }
}
```

- Ovo može:

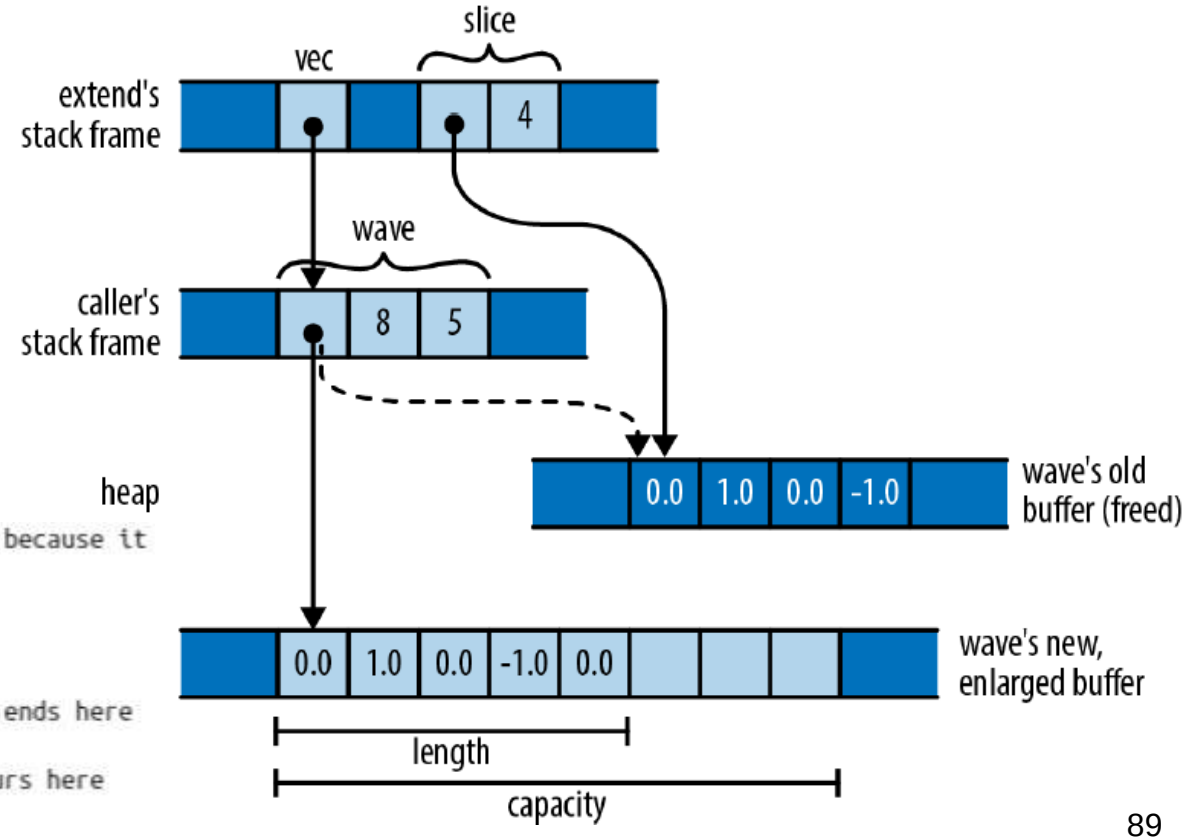
```
let mut wave = Vec::new();
let head = vec![0.0, 1.0];
let tail = [0.0, -1.0];

extend(&mut wave, &head);    // extend wave with another vector
extend(&mut wave, &tail);    // extend wave with an array
```

OGRANIČENJA U RADU SA REFERENCAMA

- Ali problem nastaje ovde:

```
extend(&mut wave, &wave);
```

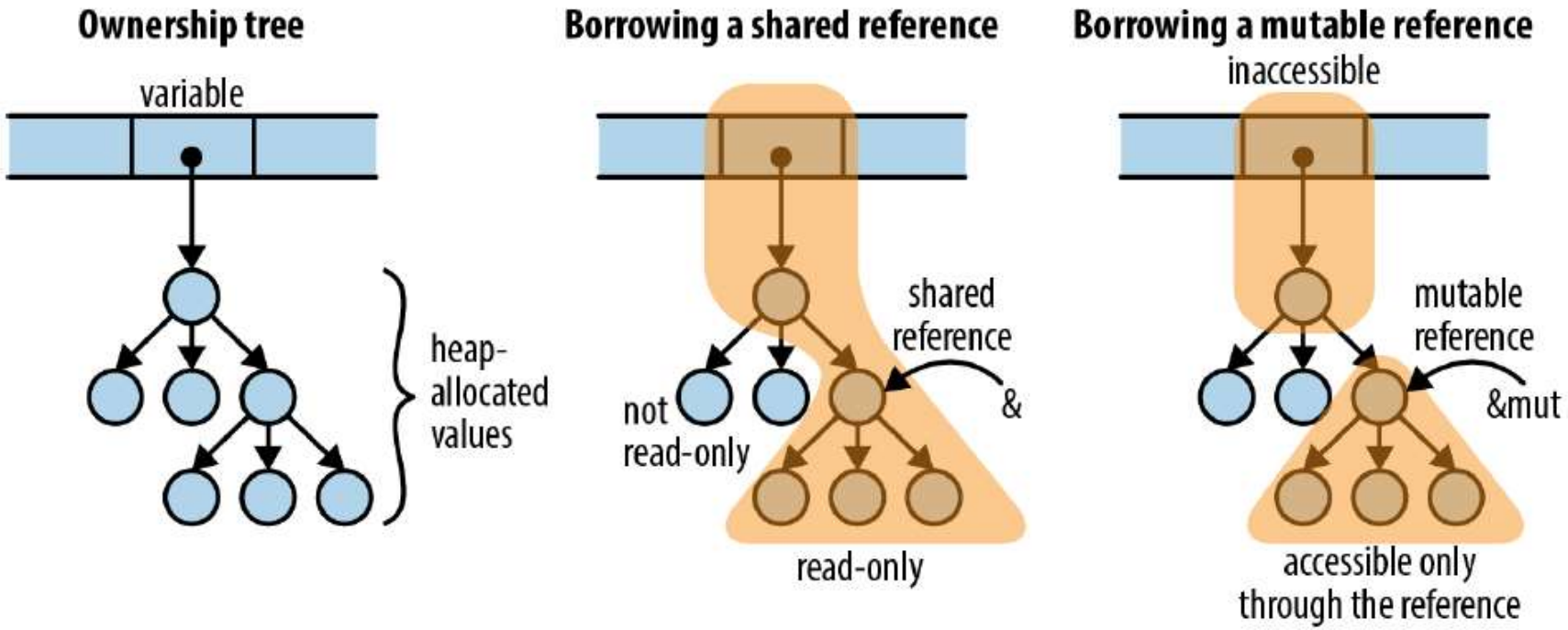
- Proširujemo vektor isečkom samog vektora, zašto je to problem?
- Memorija u jednom trenutku može da izgleda ovako:
- Problem sa ovim bagom je što se ne dešava uvek!




```
error[E0502]: cannot borrow `wave` as immutable because it
borrowed as mutable
 --> references_sharing_vs_mutation_2.rs:9:24
  |
9 |     extend(&mut wave, &wave);
  |                ----   ^^^^- mutable borrow ends here
  |                |      |
  |                |      immutable borrow occurs here
  |                mutable borrow occurs here
```

OGRANIČENJA U RADU SA REFERENCAMA

- U primeru se aktivira Rust pravilo da se deljenim referencama pristupa samo **read-only**, dok je **write** pristup ekskluzivan



OGRANIČENJA U RADU SA REFERENCAMA

- Sličan primer sa strukturom i fajlovima:

```
struct File {
    descriptor: i32
}

fn new_file(d: i32) -> File {
    File { descriptor: d }
}

fn clone_from(this: &mut File, rhs: &File) {
    close(this.descriptor);
    this.descriptor = dup(rhs.descriptor);
}
```

- Sledeći kod pravi haos:

```
let mut f = new_file(open("foo.txt", ...));
...
clone_from(&mut f, &f);
```

OGRANIČENJA U RADU SA REFERENCAMA

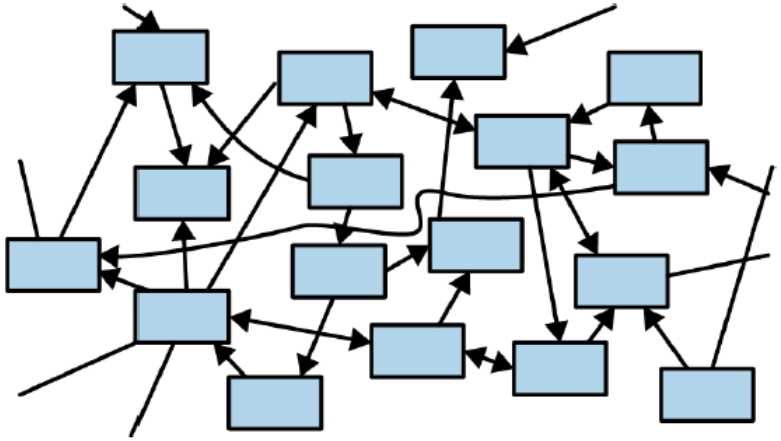
- Rust će naravno to uhvatiti prilikom kompajliranja i sprečiti:

```
error[E0502]: cannot borrow `f` as immutable because it is also
borrowed as mutable
  --> references_self_assignment.rs:18:25
   |
18 |     clone_from(&mut f, &f);
   |                -       ^- mutable borrow ends here
   |                |       |
   |                |       immutable borrow occurs here
   |                mutable borrow occurs here
```

- Za razliku od C++, Rust detektuje dva klasična baga koji mogu nastati u C++

OGRANIČENJA U RADU SA REFERENCAMA

- Rust je dizajniran da spreči stvaranja tzv. mora objekata:



- Stablo vrednosti: