RUST

1. GETTING STARTED

1.2 Hello world

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

Komande za pokretanje programa

```
 rustc main.rs
 .\main.exe
Hello, world!
```

Anatomija Rust programa

```
fn main() {
}
```

Main funkcija se uvek prva izvršava u Rust programu . Telo funkcije se nalazi u vitičastim zagradama.Unutar njega se nalazi print funkcija. Ono što primećujemo je da Rust indentuje sa 4 razmaka a ne sa tabom. Drugo, Println! Je makro, oni imaju drugačija pravila od običnih funkcija. Četvrto, izraz završavamo sa; i da sledeći može da počne. Uglavnom sve linije koda u Rustu završavaju sa ;.

Kompajlirnje I Runovanje su odvojeni koraci

Prvo moramo da kompajliramo kod i za to koristimo komandu **rustc** i prosledjujemo fajl koji hoćemo da se kompjlira - rustc main.rs. Ako smo uspešno kompajlirali, videcemo 3 fajla main.exe (executable file) main.pdb (debugging extenzija), main.rs (source file). Kompajlirani fajl **run**ujemo sa komandom .\main.exe. I izvršiće program.

Rust je a head of time compiled jezik, što znači da ako kompajliramo program, možemo nekom drugom da damo executable I on može da runuje bez da instalira rust, što nije slučaj sa drugim jezicima npr pythonom. Kompjliranje samo sa rustc komandom je ok za manje programe ali za veće nam treba Cargo tooling.

1.3 Hello Cargo

Cargo je build sistem I package manager. On hendluje bildovanje projekata, download biblioteka (dependecija) I gradjenje sopstvenih biblioteka. Cargo dolazi instaliran sa Rustom cargo –version.

Kreiranje projekta sa Cargom $ cargo new hello_cargo, cd hello_cargo. Cargo je definisao 2 fajla I jedan folder za nas. Cargo.toml fajl I src folder sa main.rs fajlom. Takodje je inicijalizovao git repozitori sa .gitignore fajlom. Git fajlovi nece biti generisani sa komandom cargo new u postojecem git repozitorijumu. Možemo da overrajdujejmo ovo ponašanje sa komandom cargo new --vcs=git. Sa –vcs flagom biramo da ne koristimo version control.

Cargo.toml

```
[package]

name = "hello_cargo"

version = "0.1.0"

edition = "2021"
```

TOML (Tom's Obvious, Minimal Language) format, koji je kofigurise Cargo.

Prva linija [package] je section heading I navodi statement koji definišu paket.

Linija [dependencies] definise spoljašnje dependencije.

U Rustu, paketi koda nazivamo Crates.

Cargo očekuje da nasi source fajlovi budu uskladišteni u src folderu. Top level folder namenjen je readme fajlu, licence informacije, konfiguracioni fajlovi itd…

BUILDING AND RUNNING CARGO PROJECT

Komanda Cargo build:

cargo build

```
  Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)

   Finished dev [unoptimized + debuginfo] target(s) in 2.85 secs
```

Ova komanda kreira executable file u target\debug\hello_cargo.exe, u našem a ne u trenutnom folderu. Zato sto debug build defaultni build, cargo stavlja binary u debug folder. Executable možemo da pokrenemo sa ovom komandom:

```
.\target\debug\hello_cargo.exe
```

Kada pokrenemo Cargo build po prvi put on takodje kreira cargo.lock u top level folderu. On void računa o tačnim verzijama dependencija u našem projektu. Cargo automatski menadžuje ovaj content za nas.

Možemo da koristimo komandu **Cargo Run** koja će istovremeno kompajlirati i runuvati naš kod.

```
cargo run
   Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running `target/debug/hello_cargo`
Hello, world!
```

Cargo run je mnogo uobičajeniji način za ranovanje našeg koda.

Ukoliko promenimo source kod cargo će rebildovati kod I dobićemo drugačiju poruku:

```
cargo run
  Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
   Finished dev [unoptimized + debuginfo] target(s) in 0.33 secs
    Running `target/debug/hello_cargo`
Hello, world!
```

Cargo check je komanda koja brzo proverava da li se nas kod kompajlira ali ne proizvodi executable.

```
cargo check
  Checking hello_cargo v0.1.0 (file:///projects/hello_cargo)
   Finished dev [unoptimized + debuginfo] target(s) in 0.32 sec
```

Check je zgodan zato što je mnogo brži od builda zato što ne proizvodi executable. Na ovaj način možemo da ubrzamo proces našeg rada zato što konstantno proveravamo da li se naš kod kompjlira. Na kraju kad smo spremni da napravimo executable runujemo build.

Upotreba Cargo-a:

- Kreiramo projekte sa **cargo new**
- Buildujemo projekte sa **cargo build**
- Buildujemo I runujemo projekte sa **cargo run**
- Proveravamo da li se kod kompajlira sa **cargo check** bez da proizvodimo executable.
- Cargo čuva rezultate u target/build folderu

BUILDING FOR RELEASE

Kada je program spreman za release koristimo komandu **build –release** da ga kompajliramo sa optimizacijom. Ova komanda će stvoriti exectuble u target/release folderu umesto target/debug folderu. Optimizacija čini naš program bržim ali je I vreme kompajliranja duže. Zato postoje 2 različita profila: jedan za development kada rebildujemo brzo I često I jedan za build finalnog programa koji ćemo predati useru I koji se neće rebildovati ali ce biti veoma brz.

CARGO KONVENCIJE

Za jednostavne programe dovoljno je da koristimo **rustc** ali čim program počne da raste treba da koristimo cargo. Iako Hello Cargo program deluje jednostavno, on obuhvata sav potreban tooling koji nam treba.

Ako radimo na nekom postojećem projektu, možemo da proverimo kod koristeci Git, kloniramo taj projekat prebacimo se u njegov folder I bildujemo:

```
git clone example.org/someproject
cd someproject
cargo build
```

Za više informacija pogledati dokumentaciju.

SUMMARY

Rustup koristimo da instaliramo poslednju stablinu verziju Rusta

Updajtejemo noviju verziju

2.Programming a gusessing game

SETTING UP PROJECT

```
cargo new guessing_game
cd guessing_game
```

PROCCESSING A GUESS

```
use std::io;

fn main() {
    println!("Guess the number!");

   println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {guess}");
}
```

Po defaultu Rust ima set itema definisanih u standarr librariu koje ubacuje u kod svakog programa, to se zove **prelude** I možemo naći u dokumentaciji. Ako tip koji nam treba nije u preludu treba explicitno da ga ubacimo u scope programa koristeći use statement. Use::std::io biblioteka pruža niz fitchera, uključujući I onaj koji može da prima inpute od usera.

```
println!("Guess the number!");
 println!("Please input your guess.");
```

Println je makro koji printuje string na ekran

ČUVANJE VREDNOSTI U PROMENLJIVAMA

```
let mut guess = String::new();

let apples = 5;
```

Promenljivoj apples dodeljujemo vrednost 5. U Rustu su vrednosti imutabilne po defoltu. To znači da je **let mut guess** promenljiva mutabilna I njoj dodeljuemo String::new() funkciju - konstruktor- koji vraća instancu tipa string koji se nalazi u std biblioteci, i on je rastuci UTF-encoded bit texta. Sintaksa :: u preimeru ::new označava da je **new** funkcija dodele tipa String. Associated funkcija ili funkcija dodele je ona koja se implementira na tip, u ovom slučaju na String. Ova new funkcija stvara novi prazan string.

let mut guess = String::new(); ustvari kreira mutabilnu promenljivu koja se vezuje za novu praznu instancu Stringa.

Receiving User Input

Za user input koristimo metodu iz std biblioteke:

```
io::stdin()
    .read_line(&mut guess)
```

Iz io modula pozivamo stdin() funkciju.

Stdin funkcija vraca instancu std::io::Stdin tip koji handluje input u terminalu.

Sledeća linija .read_line(&mut guess) poziva metodu read_line koja čita input usera. Funkciji read_line prosledjujemo &mut guess da kazemo kakav string storuje input. Posao read_line funkcije je da apenduje standard user input na string, tako da prosledjujemo taj string kao argument. String argument mora da bude mutabilan kako bi mogli da ga menjamo. & znak znači da se radi o referenci.

HENDLOVANJE POTENCIJALNOG NEUSPEHA

.expect("Failed to read line");

To smo mogli da napišemo I na ovaj način:

io::stdin().read_line(&mut guess).expect("Failed to read line");

Napisati sve to u jednoj liniji je predugačko, tako da je bolje svaku metodu pisati na zasebnoj liniji koja poseduje ou sintaksu .method_name(). Metoda read_line prima string kao argument ali takodje vraća I Result kao vrednost. Result je enumeracija (enum), koji može posedovati različita stanja – variant. Svrha ovog Result-a je da enkodujemo error-handling informaciju.

Variant's Result-a su Ok I Err. Ok varianta sugeriše da je operacija uspesna I da je unutar ok-a uspešno generisana vrednost. Err varianta znači da je operacija fejlovala I Err sadrži informaciju kako ili zašto je operacija fejlovala. Vrednosti Result tipa, kao vrednosti bilo kog tipa, imaju vrednosti, definisane nad njima. Instanca Result-a ima expect metodu koju možemo da pozovemo nad njom. Ako je ova instanca Result-a Err vrednost, expect ce krešovati program, I ispisati poruku koju smo prosledili kao argument expectu. Ako read_line metoda vrati Err, to je verovatno rezultat erora koji dolazi od undrlaying operating sistema. Ako je instanca Resulta Ok vrednost, expect će vratiti vrednot koji drži ok I vratiti samo tu vrednot da možemo da je koristimo. U ovom slučaju, to je broj bajtova userovog inputa - usize. Ako ne pozovemo expect, program će se kompajlirati ali cemo dobiti warning:

```
$ cargo build
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
warning: unused `Result` that must be used
 --> src/main.rs:10:5
  |
10 |     io::stdin().read_line(&mut guess);
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: this `Result` may be an `Err` variant, which should be handled
  = note: `#[warn(unused_must_use)]` on by default

warning: `guessing_game` (bin "guessing_game") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.59s
```

Rust nas upozorava da nismo iskoristili vrednost Rusulta i vraćen from read_line funkcije indikujući da nismo hendlovali potencijalni error. Najbolji način je da napišemo error-handling code, ali mi u ovom slučaju želimo samo da krešujemo program.

PRINTING VALUES WITH PRINTLN! PLACEHOLDERS

```
use std::io;

fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {guess}");
}
```

println!("You guessed: {guess}"); Ova linija printa string koji je user uneo u konzolu. Ono što se nalazi izmedju vitičastih zagrada je ustvari placeholder za stvarnu vrednost. Kada printamo rezultat evaluacije nekog izraza, smeštamo prazne vitičaste zagrade u format string, I potom pratimo izraze podeljene zarezima sa vitičastim zagradama koje sadrže placeholder vrednosti, npr:

```
let x = 5;

let y = 10;

println!("x = {x} and y + 2 = {}", y + 2);
```

Ovaj kod će isprintitati x = 5 i y + 2 = 12.

TESTIRANJE PRVOG DELA

```
$ cargo run
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
   Finished dev [unoptimized + debuginfo] target(s) in 6.44s
    Running `target/debug/guessing_game`
Guess the number!
Please input your guess.
6
You guessed: 6
```

Dobijamo input koji smo uneli sa tastature.

GENERISANJE TAJNOG BROJA

Tajni broj treba da bude različit svaki put tako da ćemo korisiti random broj izmedju 1 I 100.Rust još uvek nema randomizer brojeva u std biblioteci, tako da ćemo morati da koristimo neki drugi crate.

KORIŠĆENJE CRATE-A ZA DOBIJANJE VIŠE FUNKCIONALNOSTI

Crate je kolekcija Rust source code fajlova. Projekat koji mi gradimo je binary crate i to znači da je executable. U ovom slučaju ćemo koristi rand crate I to je library crate, namenjen da se izvršava u drugim programima. Pre nego što napišemo program koji koristi rand, moramo da rekonfigurišemo naš Cargo.toml fajl I dodamo rand crate u dependeci sekciju.

```
[dependencies]
rand = "0.8.5"
```

“0.8.5.” specifajer se odnosi na verziju I naziva se Sematnic versioning (Sem ver). Ovo je skraćeno od ^0.8.5, što znači vece od verzije 0.8.5 ali manje od verzije 0.9.0. Sada kada izbildujemo projekat, Cargo downloaduje naše dependencije:

```
$ cargo build
   Updating crates.io index
 Downloaded rand v0.8.5
 Downloaded libc v0.2.127
 Downloaded getrandom v0.2.7
 Downloaded cfg-if v1.0.0
```

```
Downloaded ppv-lite86 v0.2.16
Downloaded rand_chacha v0.3.1
Downloaded rand_core v0.6.3
 Compiling libc v0.2.127
 Compiling getrandom v0.2.7
 Compiling cfg-if v1.0.0
 Compiling ppv-lite86 v0.2.16
 Compiling rand_core v0.6.3
 Compiling rand_chacha v0.3.1
 Compiling rand v0.8.5
 Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
  Finished dev [unoptimized + debuginfo] target(s) in 2.53s
```

Kada uključujemo spoljašnje dependncije, Cargo fetchuje poslednje verzije svih dependencija iz registrija koji su kopije podataka koji se nalaze u Crates.io. Tamo možemo da nadjemo open source projekte koji su nam potrebni za naš projekat. Iako u ovom slučaju nama treba samo rand crate, cargo ce fetchovati I druge dependencije potrebne za rand I potom ih kompajlirati I runuvati program.

OSIGURANJE OD PONOVNIH REBILDOVA SA CARGO.LOCK FAJLOM

Cargo ima mehanizam koji obezbedjuje da rebilduje isti artefakt svaki put kada bildujemo kod. To znači da ukoliko se neki Crate koji koristimo u medjuvremenu promeni, cargo.lock nas osigurava da koristimo verziju Crate-a iz prvog builda, sve dok explicitno ne promenimo verziju tog Crate-a.

UPDEJTOVANJE CRATE-A SA NOVOM VERZIJOM

Kada hoćemo da updejtujemo Crate koristimo opciju **update** koja ce ignorisati **lock** fajl i updejtovati poslednju verziju Crate-a u Toml fajlu, zatim ce tu novu verziju zapisati Cargo.lock fajlu.

```
$ cargo update
   Updating crates.io index
   Updating rand v0.8.5 -> v0.8.6
```

```
[dependencies]
rand = "0.9.0"
```

GENERATING A RANDOM NUMBER

```
use std::io;
use rand::Rng;

fn main() {
   println!("Guess the number!");

   let secret_number = rand::thread_rng().gen_range(1..=100);

   println!("The secret number is: {secret_number}");

   println!("Please input your guess.");

   let mut guess = String::new();

   io::stdin()
      .read_line(&mut guess)
      .expect("Failed to read line");

   println!("You guessed: {guess}"); }
```

Prvo dodajemo liniju use rand::Rng; Rng trait definiše metode koje random number generator implementira, i taj trait moramo da dovedemo u scope našeg koda. Potom dodajemo 2 linije koda:U prvoj pozivamo rand::thread_rng funkciju koja nam daje odredjenu random number generator funkciju koju ćemo da koristimo, jednu koja je lokalna za trenutni thread koji nam obezbedjuje naš OS. Potom pozivamo gen_range metodu na random_number generatoru. Ova metoda je definisana Rng traitom koji je doveden u scope sa use rand::Rng; stejtmentom. Gen_range metoda uzima range expresion kao argument i generiše random broj u rangeu. Vrsta range izraza koji ovde koristimo ima formu start..=end, i inkluzivna je na gornjim i donjim granicama tako da moramo da preciziramo 1.=1000 da bi tražili broj izmedju 1 i 100. Da bismo znali koji Crate ima koje metode i koje nam trebaju možemo da koristimo komandu cargo doc –open, koji će nam bildovati dokumentaciju na osnovu naših dependencija lokalno i otvoriti u browseru. Druga nova linija koju smo uveli printa broj.

```
$ cargo run
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
   Finished dev [unoptimized + debuginfo] target(s) in 2.53s
    Running `target/debug/guessing_game`
Guess the number!
The secret number is: 7
Please input your guess.
4
You guessed: 4

$ cargo run
   Finished dev [unoptimized + debuginfo] target(s) in 0.02s
    Running `target/debug/guessing_game`
Guess the number!
The secret number is: 83
Please input your guess.
5
You guessed: 5
```

COMPARING THE GUESS TO THE SECRET NUMBER

Sad kad imamo input od usera i random broj, možemo da ih uporedimo.

```
use rand::Rng;
use std::cmp::Ordering;
use std::io;

fn main() {
   // --snip--
```

```
    println!("You guessed: {guess}");

    match guess.cmp(&secret_number) {
        Ordering::Less => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal => println!("You win!"),
    }
}
```

Prvo uvodimo novi use statement unoseći tip std::cmp::Ordering u scope našeg koda iz std biblioteke. Ordering tip je enum i ima variantse Less. Greater i Equal. To su 3 potencijale varijante poredjenja. Potom dodajemo 5 novih linija koje koriste Ordering type. Cmp metoda poredi 2 vrednosti. Uzima refernece na vrednosti koje želimo da poredimo: ovde konkretno guess i secret_number. Potom vraća variantse Ordering enuma. Koristimo **match** izraz da dolučimo šta da radimo na osnovu variante koju vraca Ordering iz **cmp** poziva sa vrednostima guess i secret_number. Match izraz se sastoji od arms-a. Arm se sastoji od paterna sa kojim se metchuje, i koda koji treba da se izvrši ako se vrednost prosledjena matchu poklapa sa paternom arma. Rust uzima vrednost proslednjenu match-u i i prolazi kroz svaki arm patern. Na našem mprimeru match izraz radi na sledeći način: Ako je user ukucao 50 a random generisani broj je 38. Nas kod poredi 50 i 38 a cmp metoda će vratitit Ordering::Greater. Match izraz dobija Ordering::Greater vrednost i počinje da proverava svaki arm patern. Ignoriše paterne sa kojima se ne poklapa. Asocijativni kod sa Ordering::Greate je da printuje Too big. Match izraz se završava posle prvog uspešnog poklapanja. Veoma je bitno da matchu prosledimo i iste tipove tako da stringove moramo parsovati u intove da bi mogli da ih poredimo.

```
// --snip--

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    let guess: u32 = guess.trim().parse().expect("Please type a number!");
```

```
    println!("You guessed: {guess}");

    match guess.cmp(&secret_number) {
        Ordering::Less => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal => println!("You win!"),
    }
```

Dodali smo sledeću liniju u kod:

```
let guess: u32 = guess.trim().parse().expect("Please type a number!");
```

Rust nam dozvoljova da shadow-ujemo **guess** vrednost. Ova pogodnost se često koristi kada hoćemo da konvertujemo jedan tip u drugi. Mi bajndujemo ovu novu vrednost sa izrazom **guess.trim().parse()**. Guess se u ovom slučaju vezuje za originalnu guess promenljivu koja sadrži input kao string. Trim metoda uklanja white spejsove na početku i na kraju, što je neophodno da bismo mogli da poredimo string sa tipom u32, koji sadrži samo numeričke podatke. User mora da ispostuje read_line funkciju i inputira broj. Kada na primer unese 5 i pritisne enter, guess izgleda ovako: 5\n. \n predstavlja novu liniju.Na windowsu, pritisak entera rezultira sa \r\n. Te dodatke rešava trim metoda i dobijamo samo broj 5. Parse metoda pretvara string u novi tip. Ovde string konvertujemo u broj. Moramo da naznačimo konkretan tip sa let guess: u32. u32 je pogodan tip za male pozitivne brojeve.

ALLOWING MULTIPLE GUESSES WITH LOOPING

Loop keyword kreieira infinite loop. Dodajemo loop da bi user imao više šansi za pogodak.

```
// --snip--

    println!("The secret number is: {secret_number}");

    loop {
        println!("Please input your guess.");
```

```
    // --snip--

    match guess.cmp(&secret_number) {
        Ordering::Less => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal => println!("You win!"),
    }
}
```

Sve od guess input prompta pa nadalje smo prebacili u loop. Ovu petlju možemo prekinuti sa ctr+c ali se ona prekida i kad user unese pogrešan odgovor.

```
$ cargo run
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
   Finished dev [unoptimized + debuginfo] target(s) in 1.50s
    Running `target/debug/guessing_game`
Guess the number!
The secret number is: 59
Please input your guess.
45
You guessed: 45
Too small!
Please input your guess.
60
You guessed: 60
Too big!
Please input your guess.
59
You guessed: 59
You win!
```

```
Please input your guess.
quit
thread 'main' panicked at 'Please type a number!: ParseIntError { kind: InvalidDigit }', src/main.rs:28:47
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Tajping quit će prekinuti igru kao i pogrešan odgovor.

QUITTING AFTER A CORRECT GUESS

```
    // --snip--


        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
    }
}
```

Dodavanjem break linije izlazimo iz petlje kad user pogodi tačan broj. Izlaz iz petlje je i izlaz iz programa, pošto je petlja poslednji deo main funkcije.

HANDLING INVALID INPUT

Loša je praksa prekidati igru kad user pogreši broj, tako taj deo treba da ignorujemo da bi user mogao da ponastavi da pogadja.

```
io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");


    let guess: u32 = match guess.trim().parse() {
```

```
        Ok(num) => num,
        Err(_) => continue,
    };

    println!("You guessed: {guess}");

    // --snip—
```

Prebacujemo se sa expect poziva na match expression da bismo sa krešovanja na error prešli na handle on error. Parse metoda vraća Result enum sa variantsima Ok i Err i ovde ponovo koristimo match izraz. Ako parse() uspe da pretvori string u broj, vratiće se Ok i rezultatni broj. Taj ok će se metchovati sa paternom prvog arma i match izraz će vratiti samo num vrednost koji je parse metoda proizvela i smestice je u ok vrednost. Taj broj će se potom smestiti u novu guess promenljivu koju kreiramo. Ako parse() ne uspe da pretvori string u broj vratiće Err vrednost sa više informacija o erroru. Err vrednost se metchuje sa Err(_) paternom drugog arma. Donja crta je catchcall _ vrednost. U ovom primeru mi hoćemo da metchujemo sve err vrednosti bez obzira na informaciju koju nose. I tako će program izvršiti kod drugog arma continue koji govori programu da nastavi u sledeću iteraciju loopa i pitati za novi guess. Na ovaj način efikasno ignorujemo sve errore na koje parse() može da naidje. Sada bi u programu sve trebalo da radi.

```
$ cargo run
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
   Finished dev [unoptimized + debuginfo] target(s) in 4.45s
    Running `target/debug/guessing_game`
Guess the number!
The secret number is: 61
Please input your guess.
10
You guessed: 10
Too small!
Please input your guess.
99
You guessed: 99
Too big!
```

```
Please input your guess.
foo
Please input your guess.
61
You guessed: 61
You win!
```

Na kraju samo treba da izbrišemo liniju koja printuje secret number, i dobijamo ceo kod.

```
use rand::Rng;
use std::cmp::Ordering;
use std::io;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1..=100);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin()
            .read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };
```

```
        println!("You guessed: {guess}");

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
    }
}
```

3.COMMON PROGRAMMING CONCEPTS

3.1 PROMENLJIVE I MUTABILNOST

Po defoltu promeljive su imutabilne. Na ovaj način se postiže bezbednost i konkurentnost. Promenljive možemo napraviti mutabilnim. Ako dodelimo vrednost nekoj promenljivoj vezanoj za neko ime, vrednot te ptomenljive više ne možemo da menjamo.

```
fn main() {
    let x = 5;
    println!("The value of x is: {x}");
    x = 6;
    println!("The value of x is: {x}");
}
```

Kada runujemo ovaj program, dobićemo error:

```
$ cargo run
  Compiling variables v0.1.0 (file:///projects/variables)
error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:4:5
```

```
  |
2 |     let x = 5;
  |         -
  |         |
  |         first assignment to `x`
  |         help: consider making this binding mutable: `mut x`
3 |     println!("The value of x is: {x}");
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable

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

Ovako nam kompjler pomaže da ndjemo greške u programu. Dobili smo poruku cannot assign twice to immutable variable `x`. Rust kompajler garantuje da se promenljiva neće promeniti. Ako dodamo mut keyword, promenljiva postaje mutabilna i njenu vrednost možemo menjati.

```
fn main() {
    let mut x = 5;
    println!("The value of x is: {x}");
    x = 6;
    println!("The value of x is: {x}");
}
```

Kada runujemo ovaj program dobijemo rezultat:

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished dev [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/variables`
The value of x is: 5
The value of x is: 6
```

Na nama je da odlučimo da li ćemo koristiti mutabilnu ili imutabilnu vrednost.

KONSTANTE

Kao i konstante i promenljive su vezane za ime ime i ne mogu da se menjaju ali postoje neke razlike: Sa konstantama ne možemo da koristimo mut keyword. Konstante nisu samo imutabilne po defoltu one su uvek imutabilne. Konstante koriste const keyword i vrednosti moraju biti anotirane. Konstante mogu biti deklarisane u bilo kojem scopeu, najčešśće na globalu. Konstante mogu biti vezane jedino za constant izraz a ne za rezultat same vrednosti koje mogu biti sračunate samo u runtimeu: const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3; Velika slova i donja crta su Rustova naming notacija za const. Kompajleru je lakše da izračuna vrednost ovog izraza. Radije nego da prosledimo konkretnu vrednost koja bi bila 10,800. Konstante su validne za celukupno vreme trajanja programa u scopeu u kom su deklarisane. Nije loša praksa imati jedno mesto u kodu u kom se menja hardkodovana vrednost.

SHADOWING

U Rustu možemo da deklarišemo promenljivu sa istim imenom kao prethodna promenljiva. To znači da je prva promenljiva shadow-ovana od strane druge promenljive, tako da kompajler sada gleda drugu promenljivu. Promenljiva traja do kraja scopea ili dok je neka druga ne overshadowuje.

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

    let x = x + 1;

    {
        let x = x * 2;
        println!("The value of x in the inner scope is: {x}");
    }

    println!("The value of x is: {x}");
}
```

Program ce prvo odstampati 12 pa 6:

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
```

```
    Running `target/debug/variables`
The value of x in the inner scope is: 12
The value of x is: 6
```

Shadowing se razlikuje od mut promenljivih zato što ćemo dobiti compile-time error ako izostavimo let keyword. Druga razlika je da sa upotrebom let keyworda mi kreiramo novu promenljivu i reuusujemo ime a pri tom možemo i da promenimo tip promenljive.

```
let spaces = "   ";
   let spaces = spaces.len();
```

Prva promenljiva je string druga broj.

Ukoliko upotrebimo mut na ovoj promenljivoj dobićemo compile-time-error

```
   let mut spaces = "   ";
   spaces = spaces.len();
```

The error says we're not allowed to mutate a variable's type:

```
$ cargo run
  Compiling variables v0.1.0 (file:///projects/variables)
error[E0308]: mismatched types
 --> src/main.rs:3:14
  |
2 |     let mut spaces = "   ";
  |                      ----- expected due to this value
3 |     spaces = spaces.len();
  |              ^^^^^^^^^^^^ expected `&str`, found `usize`

For more information about this error, try `rustc --explain E0308`.
error: could not compile `variables` due to previous error
```

3.2 DATA TYPES

Postoje 2 subseta tipova: skalarni i compound (složeni). Rust je staticly typed jezik tako da mora da zna tipove svih promenljivih u compile tajmu. Kompajler uglavnom može da zaključi o kom tipu se radi na osnovu vrednosti i načina korišćenja. U komplikovanijim slučajevima treba dodati anotaciju:

```
let guess: u32 = "42".parse().expect("Not a number!");
```

Ako ne dodamo type anotaciju :u32 dobićemo error koji kaže da je kompajleru potrebno više informacija:

```
$ cargo build
  Compiling no_type_annotations v0.1.0 (file:///projects/no_type_annotations)
error[E0282]: type annotations needed
 --> src/main.rs:2:9
  |
2 |     let guess = "42".parse().expect("Not a number!");
  |         ^^^^^
  |
help: consider giving `guess` an explicit type
  |
2 |     let guess: _ = "42".parse().expect("Not a number!");
  |              +++

For more information about this error, try `rustc --explain E0282`.
error: could not compile `no_type_annotations` due to previous error
```

SKALARNI TIPOVI

Skalarni tipovi predstavljaju pojedinačnu vrednost. Rust ima primarne skalarne tipove: integere, floating pointe, numbere, booleane i karaktere.

INTEGER TIP

Integer je ceo broj. Postoje unsigned i signed integers.

Length	Signed	Unsigned
8-bit	i8	u8
16-bit	i16	u16
32-bit	i32	u32
64-bit	i64	u64
128-bit	i128	u128
arch	isize	usize

Svaka varijanta moze biti signed ili unsigned i ima explicitnu veličinu u bajtovima. Unsigned integers imaju samo pozitivnu vrednost. Unsigned imaju samo pozitivnu vrednost jer nemaju i zato nemaju potrebu da imaju znak ispred. Svaka signed varijanta može da storuje brojeve od –(2^{n-1}) do 2^{n-1} inkluzivno, gde je n broj bitova koje varijanta koristi. Tako da i8 čuva brojeve od -(27) to 27 – 1, što je jednako -128 do 127. Unsigned varijante čuvaju brojeve od 0 do 2^n – 1, tako da u8 storuje brojeve od 0 do 2^8 – 1, sto je jednako 0 do 255. Isize i uszie tipovi zavise od arhitekture naših racunara. Integer literale možemo napisati na sledeće načine:

Number literals	Example
Decimal	98_222
Hex	0xff
Octal	0o77
Binary	0b1111_0000
Byte (u8 only)	b'A'

Možemo koristiti i donju crtu radi boljeg prikaza 1_000 je isto što i 1000. Kao defaultni tip Rust koristi i32. Isize i usize koristimo uglavnom za kolekcije.

INTEGER OVERFLOW

Ako npr imamo tip u8 koji storuje brojeve od 0 do 255, ako pokušamo tom tipu da dodamo vrednost 256 dogodice će se int overflow. Ovo izaziva panic u runtimeu. Ako kompjliramo sa –release flagom, Rust ne radi proveru overflowa nego izvršava 2 komplementarna wrapovanja. Za vrednosti veće od maximuma Rust wrapuje vrednost i pretvara u minimalni broj tog tipa. Za tip u8 za br 256 bice 0, za 257 bice 1. itd..

Program neće panicovati ali nećemo imati vrednosti brojeva koje zelimo. Wrapping ponašanje možemo smatrati errorom. Za hendlovanje overflowa možemo koristiti neke od metoda iz std biblioteke :

- Wrap u svim modovima wrapping_* metode kao npr: wrapping_add.
- Return none vrednost ako koristimo checked_* metode.
- Vraćamo boolean ako koristimo overflowing_* metode.
- Zasitimo metode sa min ili max vrednošću sa saturating_* metodama.

FLOATING POINT TIPOVI

Rust ima 2 primitivna tipa za floating-point brojeve, koji su brojevi za decimalne brojeve. Ti tipovi su f32 i f64, koji su 32 i 64 bita respektivno. Defoltni tip je f64 zato što na modernim CPUovima iste brzine kao f32 ali pruža veću preciznost. Svi floating point tipovi su signed.

```
fn main() {
   let x = 2.0; // f64


   let y: f32 = 3.0; // f32
}
```

Floating point brojevi su reprezentovani po IEEE-754 standardu. F32 je single precision float a f64 double. Floating brojevi dozvoljavaju sve matematičke operacije.

```
fn main() {
   // addition
   let sum = 5 + 10;


   // subtraction
   let difference = 95.5 - 4.3;


   // multiplication
   let product = 4 * 30;


   // division
   let quotient = 56.7 / 32.2;
   let truncated = -5 / 3; // Results in -1
```

```
    // remainder
    let remainder = 43 % 5;
}
```

BOOLEAN TIP

```
fn main() {
    let t = true;

    let f: bool = false; // with explicit type annotation
}
```

KARAKTER TIP

Ovo je Rustov najprimitivniji alfabetski tip.

```
fn main() {
    let c = 'z';
    let z: char = 'ℤ'; // with explicit type annotation
    let heart_eyed_cat = '😻';
}
```

Char literali koristi jednostruke navodnike naspram string literala koji koriste dvostruke. Rustov char tip je 4 bita veličine i reprezentuje Unicode skalarnu vrednost. Što znači da može da reprezentuje mnogo više nego ASCII. Validne vrednosti char-a u Rustu su i Japanska, koranska, kineska slova, emotikoni i zero-width space slova. Unicode skalarne vrednosti su u rasponu od U+0000 do U+D7FF i U+E000 do U+10FFFF inkluzivno.

COMPOUND TIPOVI

Compound tipovi mogu da grupišu više tipova u jedan tip. Postoje 2 primitivna compound tipa: Tupple i Array.

Tupple je opšti način grupisanja više vrednosti sa različitim tipovima u jedan složeni tip. Oni imaju fiksnu dužinu i jednom definisani ne mogu da se povećavaju ili smanjuju. Tupple definišemo sa vrednostima razdvojenim zarezom, kojoj na svakoj poziciji odgovara odredjeni tip, koji ne moraju da budu isti.

```
fn main() {

   let tup: (i32, f64, u8) = (500, 6.4, 1);

}
```

Promenljiva tup vezuje se za ceo tupple, zato što se za tupple smatra da je jedan ceo pojedinačni cpmpound element. Da pristupimo pojedinačnoj vrednosti tuppla možemo da koristimo matching patern da destruktuirešemo vrednost tupla.

```
fn main() {

   let tup = (500, 6.4, 1);


   let (x, y, z) = tup;


   println!("The value of y is: {y}");

}
```

Elementu možemo da pristupimo i sa tačkom, koju prati index vrednosti kojoj želimo da pristupimo.

```
fn main() {

   let x: (i32, f64, u8) = (500, 6.4, 1);


   let five_hundred = x.0;


   let six_point_four = x.1;


   let one = x.2;

}
```

Tupple koji ne poseduje nijednu vrednost se zove **unit.** Izrazi implicitno vraćaju unit vrednost ako ne vraćaju nijednu drugu vrednost.

NIZOVI

Još jedan način da napravimo kolekciju su nizovi ali za razliku od tupplova, nizovi moraju biti istog tipa. Nizovi u Rustu imaju fixnu veličinu.

```
fn main() {
   let a = [1, 2, 3, 4, 5];
}
```

Nizovi su korisni ako hoćemo da se naši podaci alociraju na stack a ne na heap ili kad želimo da uvek imamo fiksan broj elemenata. Nizovi nisu fleksibilni kao vektori. Nizovi su korisni kada znamo koliko nam tačno treba lemenata u kolekciji, npr meseci u godini:

```
let months = ["January", "February", "March", "April", "May", "June", "July",
          "August", "September", "October", "November", "December"];
```

Sintaksa za niz:

```
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

Možemo napisati i ovako ako hoćemo da svaki element ima istu vrednost:

```
let a = [3; 5];
```

to je kevivalentno:

```
let a = [3, 3, 3, 3, 3];
```

Pristupanje elementu niza:

Niz je parče koda fiksne veličine na stacku tako da elementima niza možemo pristupati preko indeksa.

```
fn main() {
   let a = [1, 2, 3, 4, 5];

   let first = a[0];
   let second = a[1];
}
```

Nepravilno pristupanje elementima niza:

```
use std::io;

fn main() {
    let a = [1, 2, 3, 4, 5];

    println!("Please enter an array index.");

    let mut index = String::new();

    io::stdin()
        .read_line(&mut index)
        .expect("Failed to read line");

    let index: usize = index
        .trim()
        .parse()
        .expect("Index entered was not a number");

    let element = a[index];

    println!("The value of the element at index {index} is: {element}");
}
```

Ovaj kod se pravilno kompajlira. Ako ranujemo ovaj program sa cargo run, I prosledimo vrednost 0, 1, 2, 3 ili 4, isprintaće se tačna vrednost za index tog niza. Medjutim ukoliko prosledimo vrednost izvan niza, npr. 10, konzola će ispisati :

```
thread 'main' panicked at 'index out of bounds: the len is 5 but the index is 10', src/main.rs:19:19
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Program je rezultirao sa runtime erorom sa razlogom korišćenja netacne vrednoti za indeksing operaciju.

Ako je index veći ili jednak od dužine niza, Rust će panikovati. Ova provera mora da se odradi u runtajmu zato što kompajler ne zna koju će vrednost user da prosledi. Ovo je primer Rustovog memory safety principa u akciji. U mnogim low level jezicima ovakva provera ne postoji, I kad prosledimo netačan index, možemo da pristupimo netačnoj memoriji. Rust nas stiti od ovog errora tako što odmah izlazi iz programa umesto da pristupi netačnoj memoriji.

3.3 FUNKCIJE

Keyword za funkcije je fn, main funkcija je ulazna tačka za mnoge programe, koja nam dozvoljava da deklarišemo nove funkcije. Rust koristi snake case notaciju za funckiju I promenljive tako da one počinju malim slovom I odvojene su donjom crtom.

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

    another_function();
}

fn another_function() {
    println!("Another function.");
}
```

Funkciju možemo da deklarišemo I posle maina, bitno je samo da je u scopeu u kojem ga caller vidi. Funkcije mogu da primaju parametre ili argumente koje su konkretne vrednosti.

```
fn main() {
    another_function(5);
}

fn another_function(x: i32) {
    println!("The value of x is: {x}");
}
```

Kada pokrenemo program sa cargo run, dobijemo:

```
$ cargo run
  Compiling functions v0.1.0 (file:///projects/functions)
   Finished dev [unoptimized + debuginfo] target(s) in 1.21s
    Running `target/debug/functions`
The value of x is: 5
```

U funkcijama moramo deklarisati tip parametra. To je namerna odluka u dizajnu Rusta i zahtevajući anotacije tipa u definiciji funkcije nam omogućava da kompajler više ne traži tip u ostatku koda I stoga ne moramo više ad ga definišemo. Kompajler takodje može I da nam pruži odgavarajuće error poruke ako zna koje tipove funkcije da očekuje. Višestruke parametre razdvajamo zarezima.

```
fn main() {
    print_labeled_measurement(5, 'h');
}

fn print_labeled_measurement(value: i32, unit_label: char) {
    println!("The measurement is: {value}{unit_label}");
}
```

Kada pokrenemo ovaj kod, dobijemo:

```
$ cargo run
  Compiling functions v0.1.0 (file:///projects/functions)
   Finished dev [unoptimized + debuginfo] target(s) in 0.31s
    Running `target/debug/functions`
The measurement is: 5h
```

STATEMENTI I IZRAZI

Tela funkcije se sastoje od niza **izjava, statementa,** koje se opciono završavaju **izrazom**.

- Izjave su instrukcije koje izvršavaju neku radnju I ne vraćaju vrednost.
- Izrazi se evaluiraju do rezultante vrednosti.
-

Primer izjave, statementa:

```
fn main() {
   let y = 6;
}
```

Statementi ne vraćaju vrednosti I samim tim ne možemo dodeliti let vrednosti drugoj promenljivoj, dobićemo error.

```
fn main() {
   let x = (let y = 6);
}
```

Statement let y=6 ne vraća vrednost, tako da x nema za čega da se binduje. Ovo je razlika u odnosu na C i Ruby, gde dodela vraća vrednost dodele i tamo možemo da napišemo x = y = 6, tako da i x i y imaju vrednost 6.

Izrazi vraćaju vrednost: izraz 5 + 6 evaluira vrednosti 11. Izrazi mogu biti delovi statementa: let y = 6 je izraz koji evaluira vrednost 6. Poziv funkcije izraz. Poziv **macroa** je izraz. Kreiranje novog blok scopea sa vitičastim zagradama je izraz.

```
fn main() {
   let y = {
      let x = 3;
      x + 1
   };

   println!("The value of y is: {y}");
}
```

Blok u ovom slučaju evaluira vrednost 4.

```
{
    let x = 3;
    x + 1
}
```

Vrednost se vezuje za y kao deo let statementa. Linija koda x + 1 nema tačku zarez na kraju. Izrazi ne uključuju završnicu sa tačka zarezom. Ako dodamo tačku zarez na kraj izraza, pretvaramo ga u statement I onda neće vratiti vrednost.

FUNKCIJE SA POVRATNOM VREDNOSTI

Funkcije mogu da vrate vrednost jedino kodu koji ih poziva. Povratne vrednosti ne imenujemo ali moramo deklarisati njihov tip nakon strelice ->. U Rustu, povratne vrednost funkcije je sinonim za vrednost konačnog izraza u bloku tela funkcije. Iz funkcije se možemo vratiti i ranije upotrebom **return** keyworda I navodjenjem vrednosti ali većina funkcija implicitno vraća poslednji izraz.

```
fn five() -> i32 {
    5
}

fn main() {
    let x = five();

    println!("The value of x is: {x}");
}
```

U ovoj funckiji nema poziva funkcija, makroa pa čak ni naredbi, samo broj 5, sam po sebi. Ovo je savršeno validna funkcija u Rustu. Tip povratne vrednosti funkcije je i32.

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/functions`
```

```
The value of x is: 5
```

Pet je povratna vrednost funkcije tipa i32. Postoje dve bitne stvari: let x = five(); pokazuje da koristimo povratnu vrednost funkcije da inicijalizujemo promenljivu. Ta linija je ista kao da smo napisali: let x = 5. Drugo, funkcija zvana 5 nema parametre i definiše tip povratne vrednosti ali je telo funkcije sačinjeno samo od broja 5 bez tačke zareza na kraju zato što hoćemo da vratimo tu vrednost.

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {x}");
}

fn plus_one(x: i32) -> i32 {
    x + 1
}
```

Ako runujemo ovaj kod, konzola će ispisati vrednost 6 ali ako stavimo tačku zarez na kraj linije x + 1 i promenimo je iz izraza u statement, dobićemo error.

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {x}");
}

fn plus_one(x: i32) -> i32 {
    x + 1;
}
```

Dobijamo sledeći error:

```
$ cargo run
  Compiling functions v0.1.0 (file:///projects/functions)
error[E0308]: mismatched types
 --> src/main.rs:7:24
  |
7 | fn plus_one(x: i32) -> i32 {
  |    --------            ^^^ expected `i32`, found `()`
  |    |
  |    implicitly returns `()` as its body has no tail or `return` expression
8 |     x + 1;
  |          - help: remove this semicolon to return this value
```

Error pokazuje da se tipovi ne podudaraju. Definicija funkcije kaže da će vrati vrednost i32 ali statement ne evaluira vrednost izražen zagradama (), unit tip. Stoga se ništa ne vraća.

3.4 COMMENTS

Programeri nastoje da načine svoj kod što čitljivijim ali je ponekad potrebno dodatno objašnjenje. To se moze učiniti komentarima.

```
// So we're doing something complicated here, long enough that we need
// multiple lines of comments to do it! Whew! Hopefully, this comment will
// explain what's going on.
```

Ili na istoj liniji kao kod.

```
fn main() {
    let lucky_number = 7; // I'm feeling lucky today
}
```

Ali najčešće je iznad linije koda na koji se odnosi:

```
fn main() {
   // I'm feeling lucky today
   let lucky_number = 7;
}
```

3.5 CONTROL FLOW

Najčešći konstrukti koji vam omogućavaju da kontrolišete tok izvršavanja Rust koda su **if** izrazi i petlje.

IF IZRAZI

If izraz nam dozvoljava da granamo naš kod u zavisnosti od uslova. Navedemo uslov I kažemo ako je ovaj uslov ispunjen, pokrenite ovaj blok koda I obrnuto.

```
fn main() {
   let number = 3;

   if number < 5 {
      println!("condition was true");
   } else {
      println!("condition was false");
   }
}
```

Blokovi koda pridruženi if izrazima se ponekad zovu arms. Opciono možemo da uključimo I else izraz I damo alternativu bloku koda ako uslov evaluira false. Ako ne pružimo else izraz a uslov evaluira false, program će preskočiti if blok I preći na narednu liniju koda.

```
$ cargo run
  Compiling branches v0.1.0 (file:///projects/branches)
   Finished dev [unoptimized + debuginfo] target(s) in 0.31s
    Running `target/debug/branches`
condition was true
```

Ako promenimo uslov:

```
Let number = 7;
```

I pokrenemo kod, dobijamo:

```
$ cargo run
  Compiling branches v0.1.0 (file:///projects/branches)
   Finished dev [unoptimized + debuginfo] target(s) in 0.31s
    Running `target/debug/branches`
condition was false
```

Uslov mora biti bool, inače dobijamo error.

```
fn main() {
   let number = 3;

   if number {
      println!("number was three");
   }
}
```

Pošto nemamo bool uslov, dobijamo sledeci eror:

```
$ cargo run
  Compiling branches v0.1.0 (file:///projects/branches)
error[E0308]: mismatched types
 --> src/main.rs:4:8
  |
4 |     if number {
  |        ^^^^^^ expected `bool`, found integer

For more information about this error, try `rustc --explain E0308`.
error: could not compile `branches` due to previous error
```

Eror indikuje da Rust očekuje bool a dobio je integer. Moramo eksplicitno da obezbedimo if sa booleanom kao uslov.

HENDLOVANJE VISESTRUKIH USLOVA SA ELSE IF

```
fn main() {
    let number = 6;

    if number % 4 == 0 {
        println!("number is divisible by 4");
    } else if number % 3 == 0 {
        println!("number is divisible by 3");
    } else if number % 2 == 0 {
        println!("number is divisible by 2");
    } else {
        println!("number is not divisible by 4, 3, or 2");
    }
}
```

Kada pokrenemo program dobijamo:

```
$ cargo run
  Compiling branches v0.1.0 (file:///projects/branches)
   Finished dev [unoptimized + debuginfo] target(s) in 0.31s
    Running `target/debug/branches`
number is divisible by 3
```

I pored toga što je 6 deljivo sa 2, Rust izvršava samo blok koji važi za prvi true uslov. Korišćenje previše else if uslova može da uneredi kod zato je dobra praksa refaktorisati kod.

KORIŠĆENJE IF U LET STATEMENTU

Zato što je if izraz možemo ga koristiti sa desne strane let statementa da dodelimo ishod promenljivoj.

```
fn main() {
   let condition = true;
   let number = if condition { 5 } else { 6 };

   println!("The value of number is: {number}");
}
```

Kada pokrenemo kod dobijamo:

```
$ cargo run
  Compiling branches v0.1.0 (file:///projects/branches)
   Finished dev [unoptimized + debuginfo] target(s) in 0.30s
    Running `target/debug/branches`
The value of number is: 5
```

Blokovi koda se evaluiraju prema poslednjem izrazu u njima, I brojevi su sami po sebi izrazi. U ovom slučaju, vrednost celog if izraza zavisi od bloka koda koji izvršava. To znači da vrednosti koje imaju potencijal da budu rezultat svakog arma if bloka moraju biti istog tipa. Ako se tipovi razlikuju, dobićemo eror:

```
fn main() {
    let condition = true;

    let number = if condition { 5 } else { "six" };

    println!("The value of number is: {number}");
}
```

Kada pokrenemo kod, dobijamo eror:

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
error[E0308]: `if` and `else` have incompatible types
 --> src/main.rs:4:44
  |
4 |     let number = if condition { 5 } else { "six" };
  |                                -        ^^^^^ expected integer, found `&str`
  |                                |
  |                                expected because of this

For more information about this error, try `rustc --explain E0308`.
error: could not compile `branches` due to previous error
```

Ovo ne radi zato što promeljive u if else bloku moraju imati jedan tip…

PONAVLJANJE SA PETLJAMA

Rust ima 3 vrste petlji: loop, for, while…

PONAVLJANJE KODA SA LOOP PETLJOM

Loop keyword govori Rustu da izvršava blok koda iznova I iznova dok mu izričito kažemo da prestane.

```
fn main() {
    loop {
        println!("again!");
    }
}
```

Ovaj program će ispisivati iznova I iznova again! Dok ga ne zaustavimo ručno sa ctr + c ili sa break keywordom.

VRAĆANJE VREDNOSTI IZ PETLJE

Jedna od upotrebi petlje je da ponovo pokušamo operaciju za koju znamo da može da fejluje, npr. Provera da li je thread završio svoj posao. Takodje ćemo možda morati da prosledimo rezultat operacije iz petlje u ostatak koda. Da bismo to uradili, vrednost koju želimo da vratimo iz petlje moramo da definišemo u izrazu iza break-a.

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

    let result = loop {
        counter += 1;

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

    println!("The result is {result}");
}
```

U ovom slučaju ćemo iz petlje vratiti vrednost 20, koja će biti zapisana u promenljivoj result.

RAZDVAJANJE VIŠESTRUKIH PETLJI

Ako imamo petlje unutar petlji, **break** I **continue** se primenjuju na najuntrašnjajiju petlju. Opciono možemo da stavimo label na petlju na koju kasnije možemo da primenimo **break** I **continue** da se ne bi odnosilo na najdublju petlju. Labeli petlje moraju početi jednim navodnikom.

```
fn main() {
    let mut count = 0;
    'counting_up: loop {
        println!("count = {count}");
        let mut remaining = 10;

        loop {
            println!("remaining = {remaining}");
            if remaining == 9 {
                break;
            }
            if count == 2 {
                break 'counting_up;
            }
            remaining -= 1;
        }

        count += 1;
    }
    println!("End count = {count}");
}
```

Spoljašnja petlja ima naziv ‘counting_up I inkrementovaće vrednost od 0 do 2. Unutrašnje petlja bez oznake se dekrementuje od 10 do 9. Prvi break koji se ne odnosi na label će izaći iz unutrašnje petlje samo. Break koji se odnosi na ‘counting_up label će izaći samo iz spoljašnje petlje. Ovaj kod će printati:

```
$ cargo run
  Compiling loops v0.1.0 (file:///projects/loops)
   Finished dev [unoptimized + debuginfo] target(s) in 0.58s
    Running `target/debug/loops`
count = 0
remaining = 10
remaining = 9
count = 1
remaining = 10
remaining = 9
count = 2
remaining = 10
End count = 2
```

KONDICIONALNE WHILE PETLJE

Program često mora da proceni stanje unutar petlje. Dok je uslov tačan petlja se pokreće. Kada uslov prestane da bude istinit program poziva break zaustavljajući petlju. Ovakvo ponašanje možemo primeniti I sa loop if else break paternom ali Rust već ima mplementiran konstrukt koji se zove while petlja I koji se veoma često koristi.

```
fn main() {
   let mut number = 3;

   while number != 0 {
      println!("{number}!");

      number -= 1;
   }
```

```
    println!("LIFTOFF!!!");
}
```

Ovaj konstrukt eliminiše mnogo nestovanja koje bi nam bilo neophodno kada bi koristili if else i break. Dok je uslov tačan, kod se izvršava, u suprotno izlazi iz petlje.

PROLAZAK KROZ KOLEKCIJU SA FOR PETLJOM

Kroz kolekciju možemo proći I sa sa while petljom:

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    let mut index = 0;

    while index < 5 {
        println!("the value is: {}", a[index]);

        index += 1;
    }
}
```

Kada pokrenemo program dobićemo sledeće:

```
$ cargo run
  Compiling loops v0.1.0 (file:///projects/loops)
    Finished dev [unoptimized + debuginfo] target(s) in 0.32s
     Running `target/debug/loops`
the value is: 10
the value is: 20
the value is: 30
the value is: 40
the value is: 50
```

Terminal će ispisati svih 5 vrednosti zato što je 5 uslov. Da ima 6 elemenata u kolekciji, šesti element ne bi ispisao. Medjutim ovaj pristup je sklon greškama jer može izazvati panic u programu ako vrednost indeksa ili testni uslovi nisu tačni. Ako bi kolekcija imla 4 elemenata a ne bi promenili uslov

while index < 4 program bi panicovao. Takodje je ovaj pristup spor jer kompajler dodaje runtime kod da izvrsi uslovnu proveru da li je index unutar granica niza na svakoj iteraciji kroz petlju. Mnogo je sažetija alternativa koristi for petlju I izvršiti neki kod za svaku stavku u kolekciji.

```
fn main() {
   let a = [10, 20, 30, 40, 50];

   for element in a {
      println!("the value is: {element}");
   }
}
```

Sada ćemo dobiti isti output kao u prethodnom primeru ali smo sada povećali bezbednost koda I eliminisali šansu za bug koji bi mogao da bude rezultat izlaska iz granica niza ili nedostizanja dovoljno daleko da bi se pristupilo nekim itemima niza. Ako koristimo for petlju ne bi morali da brinemo o promeni nekog dela koda ukoliko promenimo broj elemenata niza.

Sigurnost I sažetost for petlju čine najčešćim konstruktom petlje u Rustu. Čak I u situacijama u kojima želimo da pokrenemo kod odredjeni broj puta kao u primeru odbrojavanja koje koriste while petlju, većina Rust programera bi koristila for petlju. Način da se to uradi bio bi korišćenje opsega koji obezbedjuje standardna biblioteka I koja generiše sve brojeve u nizu počevši od jednog broja I završavajući pre drugog broja.

```
fn main() {
   for number in (1..4).rev() {
      println!("{number}!");
   }
   println!("LIFTOFF!!!");
}
```

Rev metodu koristimo da preokrenemo smer prolaska kroz niz.

4. RAZUMEVANJE OWNERSHIPA

Ownership je Rustova najedinstvenija karakteristika ima duboke implikacije na ostatak jezika. Omogućava Rustu da garantuje bezbednost memorije bez potrbe za Garbage collectorom.

4.1 ŠTA JE OWNERSHIP ?

Ownership je skup pravila koje regulišu kako Rust program upravlja memorijom. Neki jezici imaju Garbage collector koji konstantno traži memoriju koja se više ne koristi dok program radi, u nekim jezicima moramo sami da eksplicitno dodelimo I oslobodimo memoriju. Rust koristi treći pristup, memorijom se upravlja kroz sistem vlasništva sa skupom pravila koje proverava kompajler. Ako se neko pravilo prekrši, program se neće kompajlirati.

STACK I HEAP

Stack skladišti vrednosti onim redosledom kojim ih dobija I uklanja vrednosti suprotnim redosledom. Ovo se zove **last in first out**. Dodavanje podataka se naziva guranje na stack a uklanjanje iskakanje sa stacka. Svi podaci koji se čuvaju na stecku moraju imati poznatu fiksnu veličinu. Podaci sa nepoznatom veličinom u vreme kompajliranja ili veličinom koja se može promeniti čuvaju se na heapu.

Heap je manje organizovan: kada stavimo podatke na heap, zahtevamo odredjenu količinu prostora. Alokator memorije pronalazi prazno mesto na heap-u koje je dovoljno veliko, označava ga kao upotrebi I vraća pokazivač, koji je adresa te lokacije. Ovaj proces se zove alokacija na heap ili samo alokacija (guranje vrednosti na stack se ne smatra alokacijom). Pošto je pokazivač na heapu poznate fiksne veličine, možemo da sačuvamo pokazivač na stacku, ali kad želimo stvarne podatke moramo vratiti pokazivač.

Guranje na stack je brže od alokacije na heap jer alokator nikad ne mora da traži mesto za skladištenje novih podataka; ta lokacija je uvek na vrhu stacka. Uporedno, alokacija prostora na heapu zahteva više posla jer alokator prvo mora da pronadje dovoljno velik prostor da zadrži podatke, a zatim da izvrši knjigovodstvo da bi se pripremio za sledeću alokaciju.

Pristup podacima na heapu je sporiji od pristupa podacima na stacku zato što moramo da pratimo pokazivač da bi tamo stigli. Savremeni procesori su brži ako manje barataju u memoriji. Procesor brže radi sa podacima ako su bliže jedni drugima (na stacku).

Kada pozovemo funkciju, vrednosti prosledjene funkciji, uključujuči I potencijonalno pokazivače na podatke na heapu I lokalne promenljive na heapu se guraju na stack. Kada se funkcija završi, te vrednosti iskaču iz stacka.

Ownership rešava problem, koji delovi koda koriste koje podatke na heapu, minimizira količinu dupliranih podataka heapu I čisti neiskorišćene podatke na heapu kako ne bismo ostali bez prostora.

PRAVILA OWNERSHIPA

- Svaka vrednost u Rustu ima svog vlasnika.
- Istovremeno može biti samo jedan vlasnik.
- Kada owner izadje iz scopea vrednost se ispusta.

SCOPE PROMENLJIVE

Scope je obim u nekom programu za koji je stavka validna.

```
{                   // s is not valid here, it's not yet declared
    let s = "hello";   // s is valid from this point forward

    // do stuff with s
}
```

STRING TIP

Da bismo ilustrovali primer vlasništva potreban nam je složeniji tip koji se čuva na heapu. String literali su zgodni ali nepromenljivi. Ne možemo znati svaku vrednost stringa kada napišemo kod. Npr ako želimo da uzmemo korisnički unos i sačuvamo ga. Tu nam pomaže String tip koji se čuva na heapu, i kao takav može da uskladišti količinu teksta koja nam je nepoznata uvreme kompajliranja. String možemo kreirati od string literala koristeći from funkciju na sledeći način:

```
let s = String::from("hello");
```

Sa dvostrukom dvotačkom uvodimo String tip u nas namespace. Ovaj String se može mutirati.

```
let mut s = String::from("hello");

s.push_str(", world!"); // push_str() appends a literal to a Strin

println!("{}", s); // This will print `hello, world!`
```

MEMORIJA I ALOKACIJA

U string literalu znamo sadržaj u vreme kompajliranja tako da je tekst hardkodovan direktno u izvršni fajl. Zbog toga su string literali brzi i efikasni. Ova svojstva dolaze samo iz imutabilnosti string literala. Nažalost ne možemo staviti komad memorije u binari za svaki komad teksta čija je viličina nepoznata u compile timeu i čija se veličina možda promeni za vreme rada programa.

Da bismo string tipu podržali mutabilni rastući deo texta moramo da alociramo deo memorije na heapu, nepoznatom u compile timeu. To znači:

- Memorija se mora tražiti od alokatora memorije u runtimeu.
- Moramo naći način da vratimo memoriju alokatoru kada završimo sa Stringom

Prvi deo radimo mi kada pozovemo String::from, njegova implementacija zahteva memoriju koja joj je potrebna. Drugi deo je teži pošto nemamo Garbage collector koji prati memoriju a ne želimo manuelno da oslobadjamo memoriju. Rust ima drugačiji pristup. Memorija se automatski vraća kada promenljiva koja je poseduje izadje iz scopea.

```
{
    let s = String::from("hello"); // s is valid from this point forward


    // do stuff with s
  }
```

Kada izadjemo iz opsega Rust poziva drop finkciju.

PROMENLJIVE I PODACI U INTERAKCIJI SA MOVE-OM

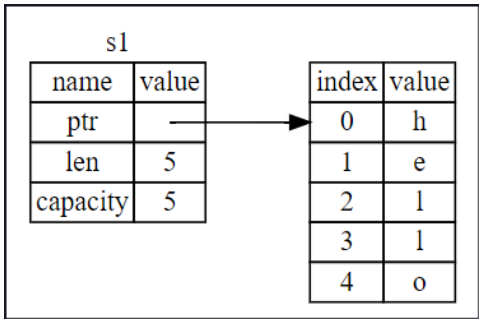
Više promenjlivih može da komunicira sa istim podacima na različite načine u Rustu.

```
Let x = 5;

Let y=x;
```

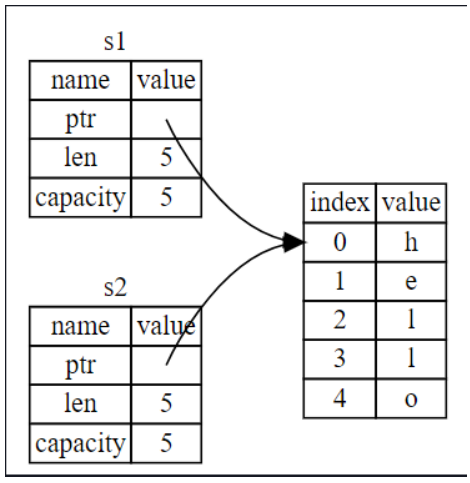
Obe vrednosti su jednake 5. Prvo bajndujemo vrednost 5 za promenljivu x zatim kopiramo vrednost x i dodelimo je y. Integeri su jednostavne vrednosti sa fiksnom veličinom i obe vrednosti 5 su gurnute na stack.

```
let s1 = String::from("hello");

    let s2 = s1;
```

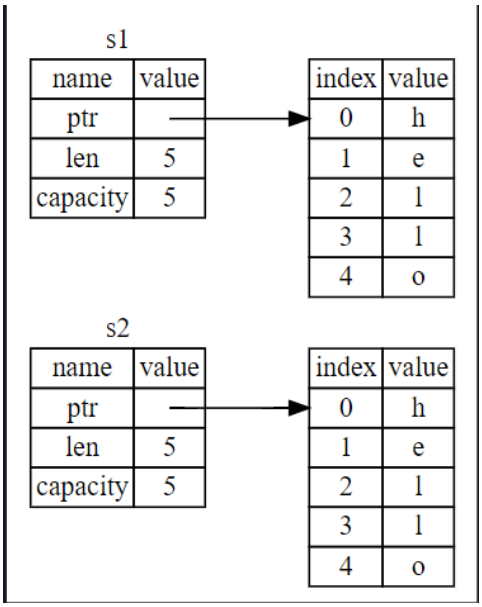
String se sastoji iz 3 dela: pokazivača na memoriju koja sadrži sadržaj stringa, dužine i kapaciteta. Ova grupa podataka se čuva na stacku. Sa desne strane je memorija na heapu koja sadrži sadržaj.



Dužina je koliko memorije u bajtovima sadržaj stringa trenutno koristi. Kapacitet je ukupna količina memorije u bajtovima koju je string primio od alokatora. Kada promenljivoj s2 dodelimo s1, kopiraju se string podaci, što znači da se kopiraju pokazivač, dužina i kapacitet koji su na stacku. Ne kopiramo podatke koji se nalaze na heapu na koju se pokazivač odnosi.



Ukoliko bi se kopirali podaci sa heapa ta operacija bi bila veoma skupa u smislu performansi tokom izvršavanja, ukoliko bi podaci na heapu bili veliki.



Ako oba pokazivača pokazuju na istu lokaciju u memoriji, nastaje problem zato što kad promenljive izadju iz scopea, obe će pokušati da oslobode istu memoriju. Ovo se naziva i **double free error** i jedan je od bugova **memory safetija.** Oslobadjanje memorije dvaput može dovesti do korupcije u memoriji i potencijalnih bezbednosnih ranjivosti. Da bi se osigurala bezbednost memorije, posle linije

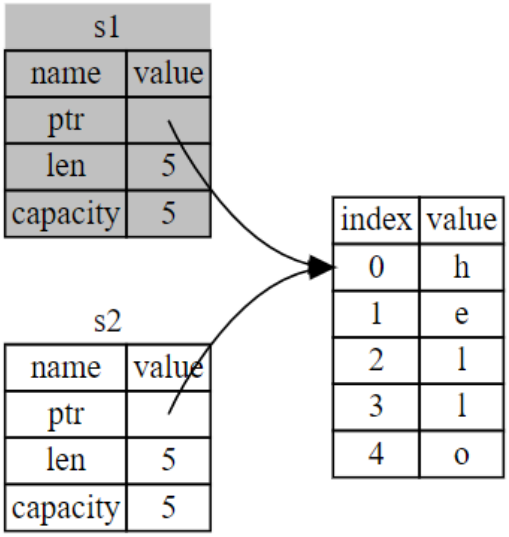
let s2=s1, Rust smatra promenljivu s1 nevažećom. Tako da Rust više ništa ne mora da oslobadja kada s1 izadje iz scopea.

```
let s1 = String::from("hello");
 let s2 = s1;

 println!("{}, world!", s1);
```

Ovaj kod će dovesti do sledećeg erora:

```
$ cargo run
  Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:28
  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ 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)
help: consider cloning the value if the performance cost is acceptable
  |
3 |     let s2 = s1.clone();
  |                ++++++++
```

Zato što Rust invalidira prvu promenljivu, ne radi se o shallow copy-u nego o move-u, tako da možemo reći da je promenljiva s1 pomerena u s2.



Ovo rešava nas problem oslobadjanja memorije pošto samo s2 ostaje validna i promenljiva, i kad ona izadje iz scopea, oslobodiće memoriju. Rustov izbor dizajna koji se podrazumeva: Rust nikada neće kreirati duboke kopije naših promenljivih, stoga za svako automatsko kopiranje se može pretpostaviti da je jeftino u smislu performansi tokom izvršenja.

PROMENLJIVE I PODACI U INTERAKCIJI SA KLONOM

Ako želimo da napravimo duboku kopiju String podataka sa heapa, a ne samo podatke sa stacka, možemo koristiti clone metodu.

```
let s1 = String::from("hello");

let s2 = s1.clone();


println!("s1 = {}, s2 = {}", s1, s2);
```

U ovom primeru se kopiraju podaci sa heapa. Kada vidimo poziv za kloniranje znamo da se izvršava neki proizvoljni kod i da taj kod može biti skup. To je vizuelni pokazatelj da se dešava nešto drugačije.

STACK-ONLY PODACI: COPY

```
let x = 5;

let y = x;


println!("x = {}, y = {}", x, y);
```

U ovom primeru, kod je validan i promenljiva neće biti pomerana u y zato što se integeri koji imaju fiksnu veličinu u runtimeu čuvaju na stacku i njihove kopije se brzo prave. Tako da nema razloga da invalidiramo x posle kreiranja promenljive y. Ovde nema razlike izmedju duboke i plitke kopije, samim tim nema potrebe za clone metodom koja bi uradila obično plitko kopiranje, tako da je možemo izostaviti.

Rust ima posenbu Copy anotaciju koja je osobina koju možemo da primenimo na tipove koji se čuvaju na stacku, kao što su integeri. Ako tip implementira copy osobinu, promenljive koje ih koriste se ne move-uju nego se pravi trivijalna kopija. što ih čini i dalje validnim nakon dodeljivanja drugoj promenljivoj.

Rust nam neće dozvoliti da označimo tip sa copy anotacijom ako bilo koji njegov deo implementira **drop** osobinu. Ako tipu treba nešto posebno da se desi kada promenljiva izadje iz scopea a mi dodelimo Copy osobinu tom tipu, dobićemo compile time eror.

Copy osobinu implementira grupa skalarnih tipova:

- Svi integer tipovi.
- Svi boolean tipovi.
- Svi floating point tipovi.
- Karakter tip
- Tuplovi koji implementiraju copy osobinu

OWNERSHIP I FUNKCIJE

Mehanika prosledjivanje vrednosti funkciji je slična dodeli vrednosti promenljivoj. Prosledjivanje promenljive funkciji će pomeriti ili kopirati baš kao što to čini dodeljivanje.

```
fn main() {
    let s = String::from("hello");  // s comes into scope

    takes_ownership(s);             // s's value moves into the function...
                                    // ... and so is no longer valid here

    let x = 5;                      // x comes into scope

    makes_copy(x);                  // x would move into the function,
                                    // but i32 is Copy, so it's okay to still
                                    // use x afterward

} // Here, x goes out of scope, then s. But because s's value was moved, nothing
  // special happens.

fn takes_ownership(some_string: String) { // some_string comes into scope
    println!("{}", some_string);
} // Here, some_string goes out of scope and `drop` is called. The backing
  // memory is freed.

fn makes_copy(some_integer: i32) { // some_integer comes into scope
    println!("{}", some_integer);
} // Here, some_integer goes out of scope. Nothing special happens.
```

Ako bismo koristili promenljivu s posle poziva finkcije takes_ownership, dobili bismo compile time eror. Ove statične provere nas štite od grešaka.

POVRATNE VREDNOSTI I SCOPE

Povratne vrednosti takodje mogu preneti vlasništvo.

```
fn main() {
   let s1 = gives_ownership();        // gives_ownership moves its return
                                      // value into s1

   let s2 = String::from("hello");    // s2 comes into scope

   let s3 = takes_and_gives_back(s2); // s2 is moved into
                                      // takes_and_gives_back, which also
                                      // moves its return value into s3
} // Here, s3 goes out of scope and is dropped. s2 was moved, so nothing
  // happens. s1 goes out of scope and is dropped.

fn gives_ownership() -> String {           // gives_ownership will move its
                                           // return value into the function
                                           // that calls it

   let some_string = String::from("yours"); // some_string comes into scope

   some_string                             // some_string is returned and
                                           // moves out to the calling
                                           // function
}

// This function takes a String and returns one
fn takes_and_gives_back(a_string: String) -> String { // a_string comes into
                                                      // scope
```

```
    a_string  // a_string is returned and moves out to the calling function
}
```

Vlasništvo nad promenljivom prati isti obrazac: dodeljivanje vrednosti drugoj promenljivoj je pomera. Kad promenljiva koja uključuje podatke na heaup izadje iz scopea, vrednost će biti očišćena automatskim pozivom drop funkcije, osim ako vlasništvo nad podacima nije premešteno u drugu promenljivu.

Rust nam omogućava da vratimo više vrednosti koristeći tuple.

```
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("The length of '{}' is {}.", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() returns the length of a String

    (s, length)
}
```

Rust ima funkciju za korišćenje vrednosti bez prenosa vlasništva, koja se zove referenca.

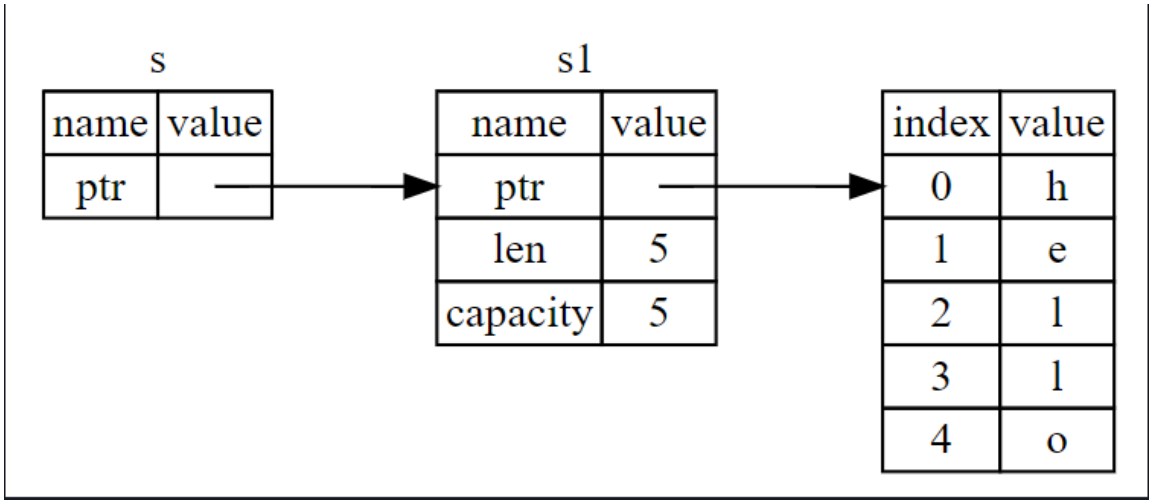
4.2 REFERENCE I POZAJMLJIVANJA

Referenca je kao pointer ka nekoj adresi na kojoj možemo da pristupimo vrednosti na toj adresi koju poseduje neka druga promenljiva. Za razliku od pointera, referenca garantuje da ukazuje na vazeću vrednost odredjenog tipa tokom trajanja te reference.

Ovo je primer kako možemo da koristimo referencu u calculate length funkciji, umesto da preuzmemo vrednost promenljive u parametru:

```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

Reference nam dozvoljavaju da se referišemo na neku vrednost bez da preuzimamo ownership nad njom.



Isto tako mozemo da deklarišemo funkciju čiji parametar ima referencu na tip, a prilikom poziva te funkcije moramo da prosledimo argument koji je referenca na samu vrednost.

```
let s1 = String::from("hello");


   let len = calculate_length(&s1);
fn calculate_length(s: &String) -> usize { // s is a reference to a String
   s.len()
} // Here, s goes out of scope. But because it does not have ownership of what
  // it refers to, it is not dropped.The length of 'hello' is 5.
```

Nakon izvršenja ove funkcije ne poziva se Drop funkcija zato što referenca ne poseduje vrednost promenljive.

Reference su kao I promenljive imutabilne po default, tako da ne možemo da ih prosledjujemo funkcijama koje menjaju neki state.

```
fn main() {
   let s = String::from("hello");


   change(&s);
}


fn change(some_string: &String) {
   some_string.push_str(", world");
}
// baciće error zato sto je promenljiva kao I referenca imutabilna…
```

MUTABILNE REFERENCE

Prethodnu funkciju možemo da ispravimo tako što ćemo proslediti mutabilnu reference na tip kao parameter prilikom deklaracije, I mutablinu reference na samu vrednost kao argument prilikom poziva te funkcije:

```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

Mutabilne reference imaju jednu veliku restrikciju a to je da možemo imati samo jednu mutablinu reference na samu vrednost.

```
let mut s = String::from("hello");

    let r1 = &mut s;
    let r2 = &mut s;

    println!("{}, {}", r1, r2);
// baciće error zato što imamo 2 reference na istu vrednost.
```

Prednost ovog ograničenja je da Rust na ovaj način može da spreči trku vrednosti u kompajl tajmu.

Trke podataka se dogadjaju kada:

Dva ili više pokazivača istovremeno pokazuju na istu vrednost.

Bar jedan od pokazivača se koristi za pisanje u samu vrednost.

Ne postoje mehanizmi za sinhronizaciju pristupa vrednostima.

Trke podataka daju undefined ponašanje i veoma ih je teko dijagnostifikovati u run tajmu, tako da Rust odbija da kompajlira ovakovo ponašanje.

Visestruke mutabilne reference su moguce jedino ako nisu u istom scopeu.

```
let mut s = String::from("hello");


  {
     let r1 = &mut s;
  } // r1 goes out of scope here, so we can make a new reference with no problems.


  let r2 = &mut s;
```

// ovo je validno I kompajliraće se.

Slično pravilo se odnosi I na kombinovanje promenljivih I nepromenljivih reference.

```
let mut s = String::from("hello");


  let r1 = &s; // no problem
  let r2 = &s; // no problem
  let r3 = &mut s; // BIG PROBLEM


  println!("{}, {}, and {}", r1, r2, r3);
```

//baca error.

Za razliku od višestrukih mutabilnih referenci koje menjaju stanje promenljive, imutabilnoh reference možemo imati više u istom scopeu zato što one služe samo za čitanje vrednosti iz promenljive.

Sledeći primer je moguć što mutabilne i imutabilne reference nisu u istom bloku:

```
let mut s = String::from("hello");

   let r1 = &s; // no problem
   let r2 = &s; // no problem
   println!("{} and {}", r1, r2);
   // variables r1 and r2 will not be used after this point

   let r3 = &mut s; // no problem
   println!("{}", r3);
```

Rust ne dozvoljava Labave reference.

Pravila reference:

U jednom trenutku možemo imati samo jednu mutablinu ili neodredjeni broj imutabilnih reference.

Reference uvek moraju biti validne.

THE SLICE TYPE

Slice tip nam omogućava da referenciramo na neprekidnu sekvencu elemenata u kolekciji, bez potrebe da referenciramo na celu kolekciju. Slice je na neki način referenca tako da ne preuzima ownership nad podacima.

STRING SLICE

String slice je referenca na deo stringa:

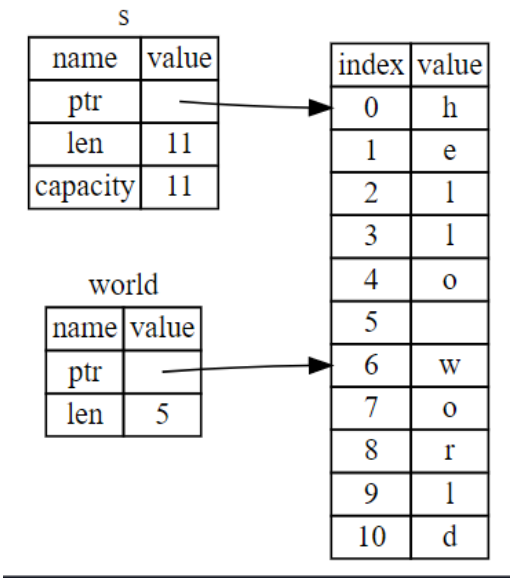
```
  let s = String::from("hello world");
 let hello = &s[0..5];
 let world = &s[6..11];
```

Poslednji bajt je pozicija indeksa – 1; &s[6..11]

Rust koristi range sintaksu sa uglastim zagradama.

Takodje ako počinjemo od prvog indeksa, možemo da izostavimo 0, &s[..5];

Isto tako možemo da izostavimo ako string obuhvata posledji bajt, tu poziciju možemo da izostavimo. &s[5..]



Ako range obuhvata ceo string, možemo da zapišemo I ovako &s[..]

Ukoliko npr. želimo da napišemo funkciju koja će deliti string po razmaku, ta funkcija bi izgledala ovako:

```
Let s = from::String(‘’Hello World’’);


fn first_word(s: &String) -> &str {
   let bytes = s.as_bytes();
for (i, &item) in bytes.iter().enumerate() {
      if item == b' ' {
         return &s[0..i]; //petlja vraca od prvog indeksa do razmaka.
      }
   }
&s[..]  // funkcija vraca vrednost koji je referenca na prvi slajs.
}
```

U petlji nalazimo kraj prve reči tako što nalazimo index bajta koji sadrži razmak, I u petlji vraćamo range od prvog indeksa do indeksa razmaka. Na taj način kada pozovemo funkciju first word, ona će vratiti prvu reč.

Na isti način možemo da dobijemo i drugu reč.

Na ovaj način se postiže validan API koji ne dozvoljava tamperovanje sa stringom jer će se kompajler pobrinuti da reference na string ostanu validne.

```
fn main() {
   let mut s = String::from("hello world");
 let word = first_word(&s);
s.clear(); // error!
println!("the first word is: {}", word);
}
```

Eror se dogadja zato što imamo imutablinu refencu na string u promenljivoj – word,

A kasnije u metodi s.clear() pokušavamo da izvršimo mutabilno pozajmljivanje, a ne možemo imati imutabilnu I mutabilnu reference na vrednost I jednom bloku u isto vreme. Takodje ako bi izvršili s.clear() metodu nad stringom, prethodni slajs više ne bi imao na šta da se referiše.

STRING LITERALS AS SLICES

String literali se čuvaju u binariju.

Let s = ‘‘Hello, World‘‘

Tip string literal s ovde je &s – to je slice koji pokazuje na specifičnu poziciju u binariju, I zato su string literali imutabilni. &str je imutabilna referenca.

STRING SLICES AS PARAMETERS

Ako uzmemo prethodne primere funkcija koje vracaju slajsove, možemo I da modifikujemo njihov zapis:

```
fn first_word(s: &String) -> &str
```

takodje mozemo da zapisemo i na sledeci nacin:

```
fn first_word(s: &str) -> &str
```

String slice možemo da prosledimo direktno a ako imamo String, možemo da prosledimo slice stringa ili referencu na string.

OTHER SLICES

Slajsovi rade I na tipovima koji nisu stringovi, npr niz intova32:

let a = [1, 2, 3, 4, 5];

let slice = &a[1..3];

assert_eq!(slice, &[2, 3]);

SUMMARY

Koncepti ownership, Borrowinga I Slajsova omogućavaju memoru safety u Rustu u compile tajmu.

5. STRUCTS

Struct ili struktura je je custom tip podataka, koji nam dozvoljava da grupišemo i imenujemo skup podataka koji tvore jednu smislenu celinu. Struct je nalik objektu.

5.1. DEFINISANJE I INSTANCIRANJE STRUCTOVA

Structovi su slični tuplovima ali za razliku od njih u structovima imenujemo svaki podatak, tako da je jasno na šta se tip odnosi. Na taj način su structovi fleksibilniji od tuplova, zato što ne moramo da brinemo o redusledu podataka u njima ili reference ka njihovim vrednostima. Struct sintaksa:

```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}
```

Da bismo korisitli struct nakon deklaracije, potrebno je da ga instanciramo i dodelimo konkretne vrednosti za polja. Struct je kao generalni template za tip a instance popunjavaju konkretne podatke za taj tip.

```
fn main() {
    let user1 = User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    };
}
```

Vrednostima nekih polja u structu pristupamo sa dot . notacijom. Ukoliko je referenca na struct mutabilna, možemo sa dot notacijom da izmenimo vrednost polja.

```
fn main() {
    let mut user1 = User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    };

    user1.email = String::from("anotheremail@example.com");
}
```

Cela instance mora biti mutabilna da bi se promenilo pojedinacno polje.

Struct instance možemo da konstruišemo I kao izraz u funkciji koja vraca Struct:

```
fn build_user(email: String, username: String) -> User {
    User {
        active: true,
        username: username,
        email: email,
        sign_in_count: 1,
    }
}
```

USING THE FIELD INIT SHORTHAND

Prethodni konstruktor možemo da napišemo kraće sa obzirom da vrednosti username i email polja imaju istu vrednost kao I sama polja.

```
fn build_user(email: String, username: String) -> User {
    User {
        active: true,
        username,
        email,
        sign_in_count: 1,
    }
}
```

KREIRANJE INSTANCI OD DRUGIH INSTANCI SA STRUCT UPDATE SINTAKSOM

Sa struct update sintaksom možemo da kreiramo novu instancu structu koristeći neku drugu instancu structa:

```
let user2 = User {
        active: user1.active,
        username: user1.username,
        email: String::from("another@example.com"),
        sign_in_count: user1.sign_in_count,
    };
```

Update sintaksu jos krace mozemo da zapisemo sa .. sintaksom:

```
fn main() {
   // --snip--


   let user2 = User {
      email: String::from("another@example.com"),
      ..user1
   };
}
```

Osim email polja, sva druga polja su preuzeta od user1 reference na struct.

Struct update sintaksa koristi = za dodelu, I to je zato što preuzima podatke.

To u ovom primeru znači da više ne možemo da koristimo user1 instancu posto ga je preuzela user2 instanca, zbog Stringa u username polju. Da smo referenci user2 dodelili nove vrednosti za polja usernamei email koji su Stringovi a preuzeli polja active i sign_in_account, onda bi user1 idalje bio validan, zato sto ta 2 polja implemetiraju Copy trait, osnosno su prosti tipovi.

KORIŠĆENJE TUPL STRUCTOVA SA NEIMENOVANIM POLJIMA ZA KREIRANJE RAZLIČITIH TIPOVA

Tupl struktovi imaju dodatno značenje koje daje ime struktu ali nemaju imena povezana sa njihovim poljima, umesto toga imaju samo tipove umesto polja. Tupl structovi su korisni kada želimo da damo ime celom tuplu I učinimo da je tupl drugačiji tip od drugih tuplova, i kada bi imenovanje polja u običnom structu bilo suvišno i opširno. Primer definisanja Tupl Structa:

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);


fn main() {
   let black = Color(0, 0, 0);
   let origin = Point(0, 0, 0);
}
```

Black I origin su vrednosti različitog tipa, zato što su instance različitih tupl structova. Svaki struct koji definišemo je sopstveni tip iako polja u njemu mogu da se podudaraju sa poljima drugih structova.

UNIT-LIKE STRUCTS BEZ POLJA

Unit-like struktovi nemaju polja. Oni mogu da budu korisni kada želimo da implementiramo trait na neki tip ali nemamo podatke koje želimo da sačuvamo u samom tipu. Primer definisanja takvog structa:

```
struct AlwaysEqual;

fn main() {
    let subject = AlwaysEqual;
}
```

Pretpostavljamo da ćemo kasnije implementirati ponašanje za taj tip, npr. da je svaka instanca **every_equal** uvek jednaka svakoj instanci nekog drugog tipa da bi np. Imali poznate rezultate za testiranje.

OWNERSHIP OF STRUCT DATA

Ukoliko structovi poseduju kompleksne tipove tipa Stringa, structovi poseduju vrednosti I one su validne za veme trajanja structa.

U structovima je moguće čuvati reference ka podacima koje čuva neko drugi ali nam za to treba upotreba Life-timea. Life-time je feauture Rusta koji omogućava da je podatak na koji se referencira struct, validan za vreme trajanja structa. Ukoliko želimo da storujemo referencu u structu bez upotrebe lifetime, dobićemo error kao u sledećem primeru:

```
struct User {
    active: bool,
    username: &str,    // kompajler ce se buniti da treba lifetime specifier
    email: &str,          // kompajler ce se buniti da treba lifetime specifier
    sign_in_count: u64,
}


fn main() {
    let user1 = User {
        active: true,
        username: "someusername123",
```

```
        email: "someone@example.com",
        sign_in_count: 1,
    };
}
```

5.2 PRIMER KORIŠĆENJA STRUCTA

Koristićemo primer programa koji izračunava površinu pravougaonika.

```
fn main() {
    let width1 = 30;
    let height1 = 50;

    println!(
        "The area of the rectangle is {} square pixels.",
        area(width1, height1)
    );
}

fn area(width: u32, height: u32) -> u32 {
    width * height
}
```

Kada pokrenemo program dobijamo:

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.42s
     Running `target/debug/rectangles`
The area of the rectangle is 1500 square pixels.
```

Ovaj kod uspeva da izračuna površinu pravougaonika pozivanjem funkcije area sa svakom dimenzijom ali ovaj kod može biti jasniji i čitljiviji.

```
fn area(width: u32, height: u32) -> u32
```

Funkcija površine treba da izračuna površinu jednog pravougaonika ali funkcija koju smo napisali ima 2 parametra i nigde u našem programu nije jasno da li su ti parametri povezani. Bilo bi čitljivije i funkcionalnije grupisati širinu i visinu zajedno. To možemo da učinimo korišćenjem **tuplova**.

REFAKTORIZACIJA SA TUPLOVIMA

```
fn main() {
    let rect1 = (30, 50);

    println!(
        "The area of the rectangle is {} square pixels.",
        area(rect1)
    );
}

fn area(dimensions: (u32, u32)) -> u32 {
    dimensions.0 * dimensions.1
}
```

Na jedan način ovaj kod je malo bolji jer nam tuplovi dodaju malo strukture i sad prosledjujemo samo jedan argument. Na drugi način, ova verzija je manje čitljiva: tuplovi neimenuju svoje elemente tako da moramo da indeksiramo do njegovi elemenata. Što čini našu kalkulaciju manje očiglednom.

Mešanje širine i visine

REFAKTORISANJE SA STRUCTOVIMA

```
struct Rectangle {
    width: u32,
    height: u32,
}
```

```
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        area(&rect1)
    );
}

fn area(rectangle: &Rectangle) -> u32 {
    rectangle.width * rectangle.height
}
```

Ovde smo definisali struct Rectangle sa poljima width i height tipa u32, potom smo u mainu kreirali instancu tog structa koji ima širinu 30 i visinu 50. Naša area funkcija je sada definisina sa jednim parametrom koji smo nazvali Rectangle, čiji je tip imutabilna referenca instance struct Rectangle. Ne želimo da preuzmemo vlasništvo nad structom, već samo da pozajmimo njene vrednosti. Na ovaj način main zadržava svoje vlasništvo i može da nastavi d koristi rect1 i zato koristimo & u potpisu funkcije i prilikom njenog poziva. Funkcija area pristupa poljima width i height instance Rectangle. Naš potpis area funkcije sad tačno govori šta želimo; izračunaj površinu pravougaonika koristeći width i height polja. Ovo dokazuje da su visina i širina medjusobno povezana koja imaju deskriptivna imena a ne indekse tupla, što je na kraju mnogo čitljivije.

DODAVANJE KORISNIH FUNKCIONALNOSTI SA DERIVED TRAIT-OVIMA

Bilo bi korisno da možemo da odprintamo instancu pravougaonika dok otklanjamo greške u našem kodu i vidimo vrednosti za sva njegova polja.

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!("rect1 is {}", rect1);
}
```

Ovaj kod medjutim neće raditi. Kada se kompajlira dobijamo eror:

```
error[E0277]: `Rectangle` doesn't implement `std::fmt::Display`
```

Println makro može da uradi mnogo vrsta formatiranja a po defaultu vitičaste zagrade govore println-u da koristi Display formatiranje, gde je output namenjen za direktno end user korišćenje. Primitivni tipovi implementiraju Display po defaultu zato što postoji samo jedan način da prikažemo primitivnu vrednost. Sa structom je način prikazivanja koji koristi println! Drugačiji pošto postoji više mogućnosti: da li da se isprintaju zarezi, vitičaste zagrade, da li da se prikažu sva polja... Zbog ove dvosmislenosti Structovi ne implementiraju Display formatiranje. U eroru dobijamo poruku koja nam može koristiti:

```
  = help: the trait `std::fmt::Display` is not implemented for `Rectangle`
  = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-print) instead
```

Kada stavimo :? Specifajer unutar vitičastih zagrada govorimo println! Makrou da hoćemo da koristimo Debug format. Debug trait nam omogućava da odprintamo struct na način nama koristan tako da možemo da vidimo vrednosti structa dok debagujemo kod. Kada kompajliramo ovaj kod idalje ćemo imati eror:

```
error[E0277]: `Rectangle` doesn't implement `Debug`
```

Kompajler nam opet daje korisnu poruku:

```
= help: the trait `Debug` is not implemented for `Rectangle`
  = note: add `#[derive(Debug)]` to `Rectangle` or manually `impl Debug for Rectangle`
```

Rust uključuje funkcionalnost za štampanje informacija o otklananju grešaka ali moramo da se opredelimo da funkcionalnost eksplicitno učinimo dostupnom za naš Struct. Neposredno pre definicije structa moramo da dodamo atribut #[derive(Debug)]

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!("rect1 is {:?}", rect1);
}
```

Sad kada pokrenemo kod nećemo dobiti eror i dobićemo sledeći output:

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/rectangles`
rect1 is Rectangle { width: 30, height: 50 }
```

Kada koristimo malo veći struct možemo koristiti {:#?} za lepši ispis:

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/rectangles`
```

```
rect1 is Rectangle {
    width: 30,
    height: 50,
}
```

Drugi način da odštamapamo vrednost koristeći Debug format je korišćenjem dbg! Makroa. Koji preuzima ownership nad izrazom, u suprotnosti nad println! Makroom koji preuzima referencu, štampa fajl i broj reda gde je taj dbg! poziv u našem kodu zajedno sa rezultantom tog izraza i vraća ownership te vrednosti. Poziv dbg! makroa štampa standard eror console stream (stderr), u suprotnosti sa println! Koji štampa standard otput console stream (stdout).

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let scale = 2;
    let rect1 = Rectangle {
        width: dbg!(30 * scale),
        height: 50,
    };

    dbg!(&rect1);
}
```

Možemo stavit dbg! makro oko izraza 30 x scale i zato što dbg! makro vraća ownership vrednosti izraza, polje width će dobiti istu vrednost kao da nismo imali dbg! poziv. Ne želimo da dbg! preuzme ownership za rect1 u sledećem pozivu. Ouput ovog primera izgleda ovako:

```
$ cargo run
  Compiling rectangles v0.1.0 (file:///projects/rectangles)
   Finished dev [unoptimized + debuginfo] target(s) in 0.61s
    Running `target/debug/rectangles`
[src/main.rs:10] 30 * scale = 60
[src/main.rs:14] &rect1 = Rectangle {
   width: 60,
   height: 50,
}
```

Vidimo je prvi deo outputa došao iz linije 10 gde debagujemo izraz 30 x scale i njegova rezultantna vrednost je 60. Na liniji 14 dbg! poziv štampa output za vrednost &rect1, koji je Rectangle struct. Output koristi prety Debuging format Rectangle tipa. Dbg! makro nam je veoma koristan kad pokušavamo da otkrijemo šta naš kod radi.

SINTAKSA METODE

Metode su slične funkcijama ali se za razliku odnjih definišu u konstekstu structa, enuma ili trait objekta i njihov prvi parametar je uvek **self** koji se odnosi na instancu structa za koji se metoda poziva.

DEFINISANJE METODA

```
#[derive(Debug)]
struct Rectangle {
   width: u32,
   height: u32,
}

impl Rectangle {
   fn area(&self) -> u32 {
      self.width * self.height
   }
}
```

```
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

Da bismo definisali funkciju u kontekstu pravougaonika, pokrećemo impl (implementacioni) blok za Rectangle. Sve unutar ovog bloka biće povezano sa tipom Rectangle. Zatim pomeramo Area funkciju unutar impl bloka i menjamo da prvi parametar bude **self** u potpisu i svuda u telu funkcije. U mainu gde pozivamo **area** funkciju i prosledjujemo **rect1** kao argument možemo umesto toga da koristimo sintaksu metode i pozovemo **area** metodu na **Rectangle** instanci. Metoda sintakse ide posle instance: dodajemo tačku praćenu imenom metode, zagradma i svim argumentima.

U potpisu **area** funkcije smo koristili **&self** umesto **rectangle: rectangle**. Ustvari **&self** je skraćeno od

self: &Self. U impl bloku tip U okviru impl blokam tip Self je pseudonim za tip za koji je impl blok namenjen.metode moraju imati parametar self tipa Self za svoj prvi parametar, tako da nam Rust dozvoljava da ovo skratimo samo na **self** na mestu prvog parametra**.** I dalje moramo koristiti & ispred skraćenice self da bismo naznačili da ova metoda pozajmljuje instancu Self. Metode mogu da preuzmu vlasništvo nad self-om, pozajem self imutabilno ili pozajme self mutabilno.

Glavni razlog za korišćenje metoda umesto funkcija su organizacioni razlozi. Stavili smo sve stvari koje možemo da uradimo sa instancom tipa u jedan blok umesto da nateramo buduće korisnike našeg koda da traže mogućnosti Rectangle structa po različitim mestima u biblioteci koju im dostavimo.

Metodi možemo dati isto ime kao i jedno os polja structa, npr width.

```
impl Rectangle {
    fn width(&self) -> bool {
        self.width > 0
    }
```

```
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    if rect1.width() {
        println!("The rectangle has a nonzero width; it is {}", rect1.width);
    }
}
```

Ovde biramo da width metoda vrati true ako je vrednost u instancinom width polju veće od nula i false ako je nula. U main funkciji kada napišemo rect1.width sa zagradama, Rust zna da se odnosi na width metodu. Ako ne koristimo zagrade Rust zna da se odnosi na polje width.

Često kada metodi damo isto ime kao polju želimo samo da vratimo vrednost polja i ništa više. Te metode se zovu geteri i Rust ih ne implementira automatski za struct polja. Geteri su korisni zato što možemo da polje privatnim a metodu public, i na taj način omogućimo pristup tom polju samo za čitanje kao deo javnog API-a tog tipa.

GDE JE - OPERATOR ?

U jeziku C i C++ postoje 2 raličita operatera koji se koriste za poziv funkcije. Koristimo tačku . ako pozivamo metodu na objektu direktno ili -> ako pozivamo metodu na pokazivaču ka objektu i potrebno je da prvo dereferenciramo pokazivač. Drugim rečima, objekat je pokazivač, object -> something() je slično kao object.something().

Rust nema ekvivalent -> operateru, ali umesto toga ima feature koji se zove automotsko referenciranje i dereferenciranje. Poziv funkcije je jedno od retkih mesta gde se primenjuje ovo ponašanje.

Kada pozovemo object.something(), Rust automatski dodaje &, &mut ili * tako da objekat prepoznaje potpis funkcije. Drugim rečima, sledeći zapis je isti:

```
p1.distance(&p2);

(&p1).distance(&p2);
```

Prvi primer deluje puno čitlivije i on predstavlja automatsko referenciranje, i ovakvo ponašanje radi zato što metode imaju čist reciver a to je tip self. Rust može da odgonetne da li metoda čita (&self) mutira(&mut self) ili konzumira (self).

METODE SA VIŠE PARAMETARA

```
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
    let rect2 = Rectangle {
        width: 10,
        height: 40,
    };
    let rect3 = Rectangle {
        width: 60,
        height: 45,
    };

    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}
```

U ovom primeru hoćemo da instanca Rectangla prihvati drugu instancu Recangla i vrati true ako drugi Rectangle može kompletno da paše sa self (prvi Rectangle), u suprotnom treba da vrati false. Output će izgledati ovako zato što su dimenzije rect2 manje od dimenzija rect1 ali su dimenzije rect3 šire od rect1.

```
Can rect1 hold rect2? true
Can rect1 hold rect3? false
```

Znamo da želimo da definišemo metodu , tako da će biti unutar impl Reactangle bloka. Ime metode će biti can_hold, i ona će imutabilno pozajmiti drugi Rectangle kao parametar. Možemo reći kog će tipa biti parametar gledajući kod koji poziva ovu metodu: rect1.can_hold(&rect2), prosledjuje &rect2, koji je imutabilna pozajmica za rect2, instancu Rectangla. Ovo ima smisla zato što samo treba da pročitamo rect2 i želimo da main zadrži vlasništvo nad rect2 tako da možemo ponovo da je koristimo posle poziva can_hold metode. Povratna vrednost can_hold metode će biti Boolean a implementacija će proveravati da li su širina i visina selfa veće od Rectangla.

```
impl Rectangle {
    fn area(&self) -> u32 {
```

```
        self.width * self.height
    }

    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

Kada dodamo još jednu can_hold metodu u impl blok i runujemo naš kod sa main funkcijom, dobićemo željeni ispis. Metode mogu uzeti više parametara koje dodajemo u potpis funkcije nakon self parametra.

POVEZANE FUNKCIJE

Sve funkcije definisane u impl bloku se zovi povezane funkcije zato što su povezane sa tipom posle impl. Možemo definisati povezane funkcije koje nemaju self za prvi parametar, i samim tim nisu metode, zato što im ne treba instanca tipa da bi se sa njima radilo. Već smo koristili jednu takvu funkciju i to je **String::from** funkcija koja je definisana na tipu string.

Povezane funkcije koje nisu metode se često koriste kao konstruktori koji će vratiti novu instancu structa. One se često zovu **new,** ali new nije specijalno ime i nije ugradjeno u jezik. Npr mogli bismo da obezbedimo asocijativnu funkciju sa nazivom **square** koja bi imala jedan parametar za dimenzije visinu i širinu i samim tim bi bilo lakše kreiranje **square Rectangle** radije nego da moramo navedemo iste vrednosti dvaput.

```
impl Rectangle {
    fn square(size: u32) -> Self {
        Self {
            width: size,
            height: size,
        }
    }
}
```

Self ključna reč u retutn tipu je pseudonim za tip koji se pojavljuje iza ključne reči **impl** koja je u ovom slučaju Rectangle.

Da pismo pozvali asocijativnu funkciju koristimo :: sa imenom structa; let sq=Rectangle::square(3). Ova funkcija je nameSpaceovana od strane structa, :: sintaksa se koristi i za asocijativne funkcije i nameSpaceove kreirane od strane modula.

VIŠESTRUKI IMPL BLOKOVI

Svaki struct može da ima višestruki impl blok:

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

Nema potrebe da ovu sintaksu da delimo na višestruke impl blokove ali ovo je validna sintaksa.

SUMMARY

Strukture nam dozvoljavaju da kreiramo custom tipove koji su značajni za naš domen Korišćenjem struktura možemo da povezujemo delove podataka. U impl možemo da definišemo funkcije koje su povezane sa našim tipom, a metode su vrsta asocijativnih funkcija koje nam dozvoljavaju da definišemo ponašanje instanci koji naši **structovi** imaju.