Dr Dinu Dragan

PARALELNE I DISTRIBUIRANE ARHITEKTURE I JEZICI (ČAS 8)

ŠTA RADIMO DANAS?

- Closure
- Ulaz/izlaz
- Mrežno programiranje

CLOSURES

CLOSURE

- Rust omogućuje pisanje anonimnih funkcija. Šta je to?
- Izrazi sa anonimnim funkcijama se zovu **closure**
- Zamislite da imate strukturu

```
struct City {
    name: String,
    population: i64,
    country: String,
    ...
}
```

- I da želite da pozovete ugrađenu metodu za sortiranje

```
fn sort_cities(cities: &mut Vec<City>) {
    cities.sort();  // error: how do you want them sorted?
}
```

- Neće proći jer **City** ne implementira **std::cmp::Ord**

CLOSURE

- U osnovi trebalo bi implementirati po čemu se vrši sortiranje, tj. po kom redosledu, nešto tipa:

```
/// Helper function for sorting cities by population.
fn city_population_descending(city: &City) -> i64 {
    -city.population
}

fn sort_cities(cities: &mut Vec<City>) {
    cities.sort_by_key(city_population_descending);  // ok
}
```

- Pomoćna funkcija **city_population_descending**, uzima referencu na instancu tipa **City** i izvlači ključ, polje po kojem se želi sortirati (vraća se negativna vrednost jer **sort** metoda sortira po rastućoj vrednosti, a ovde se želi po opadajućoj)
- Metoda **sort_by_key** uzima tu ključ-funkciju (**key-function**) kao parametar

CLOSURE

- Ovako jednostavne funkcije se mogu zameniti **closure** anonimnom funkcijom

```
fn sort_cities(cities: &mut Vec<City>) {
    cities.sort_by_key(|city| -city.population);
}
```

- Ovde je **|city| -city.population** anonimna funkcija, **closure**
- Argument je **city**, dok je povratna vrednost **-city.population**
- Rust dedukuje tip argumenta i tip povratne vrednosti na osnovu toga kako se koristi **closure**
- Primeri **closure** anonimnih funkcija mogu se naći u Iterator osobini, APIju za rad sa nitima i sl.
- Liči na anonimne funkcije u nekim drugim programskih jezicima, ali ima svoje specifičnosti

HVATANJE PROMENLJIVIH

- Anonimne funkcije se koriste za implementaciju situacije koja se baš zove **closure**
- To znači da anonimna funkcija može da koristi kao svoju promenljivu, promenljivu koja pripada funkciji koja poziva anonimnu funkciju

```
/// Sort by any of several different statistics.
fn sort_by_statistic(cities: &mut Vec<City>, stat: Statistic) {
    cities.sort_by_key(|city| -city.get_statistic(stat));
}
```

- Ovo se u Rust-u zove **capturing variables** (**stat is captured**)
- Ovde ima nekoliko caka oko kojih mora da se razmišlja a osnovno je ko je vlasnik vrednosti i šta kada/ako funkcija izađe iz opsega?
- Kod jezika sa GC to nije problem, jer je dovoljno smestiti je na heap, i GC će se pobrinuti da se ona obriše kad je više niko ne koristi, naravno toga nema u Rustu

HVATANJE PROMENLJIVIH – POZAJMLJIVANJE

- U primeru:

```
/// Sort by any of several different statistics.
fn sort_by_statistic(cities: &mut Vec<City>, stat: Statistic) {
    cities.sort_by_key(|city| -city.get_statistic(stat));
}
```

- Kada Rust kreira **closure**, automatski pozajmljuje referencu na **stat**
- **Closure** je podređeno svim pravilima vezanim za pozajmljivanje, kao i pravilima o životnim vekovima
- U ovom slučaju, kako **closure** sadrži referencu na **stat** ne može je nadživeti (da li je ovo ispunjeno u primeru?)
- Rust garantuje sigurnost tako što nameće životni vek (te mu ne treba GC)
- Gde će se smestiti **stat**?

HVATANJE PROMENLJIVIH – KRAĐA

- U sledećem primeru:

```
use std::thread;

fn start_sorting_thread(mut cities: Vec<City>, stat: Statistic)
    -> thread::JoinHandle<Vec<City>>
{
    let key_fn = |city: &City| -> i64 { -city.get_statistic(stat) };

    thread::spawn(|| {
        cities.sort_by_key(key_fn);
        cities
    })
}
```

- **thread::spawn** uzima closure i poziva je u novoj niti izvršavanja
- Obratite pažnju da **| |** znači da **closure** nema argumenata
- Nova nit se izvršava u paraleli sa niti koja ju je pozvala, a pozivalac prestaje da se izvršava tek kada **closure** vrati vrednost

HVATANJE PROMENLJIVIH – KRAĐA

- Obratite pažnju da je **closure** ovde dodeljen promenljivoj
- I u ovom primeru postoji promenljiva **stat** koju **key_fn** referencira
- U ovom primeru Rust ne može da garantuje da se reference koristi na siguran način

```
error[E0373]: closure may outlive the current function, but it borrows `stat`,
              which is owned by the current function
  --> closures_sort_thread.rs:33:18
   |
33 | let key_fn = |city: &City| -> i64 { -city.get_statistic(stat) };
   |              ^^^^^^^^^^^^^^^^^^^^                    ^^^^
   |              |                                  `stat` is borrowed here
   |              may outlive borrowed value `stat`
```

- U suštini i **cities** i **stat** se dele na nesiguran način
- Problem je u tome što ne postoje garancije da će se nova nit završiti pre nego što su **cities** i **stat** obrisane na kraju funkcije

HVATANJE PROMENLJIVIH – KRAĐA

- Rešenje je da se promenljive prebace u **closure**

```
fn start_sorting_thread(mut cities: Vec<City>, stat: Statistic)
    -> thread::JoinHandle<Vec<City>>
{
    let key_fn = move |city: &City| -> i64 { -city.get_statistic(stat) };

    thread::spawn(move || {
        cities.sort_by_key(key_fn);
        cities
    })
}
```

- To se radi pomoću ključne reči **move**
- Ključna reč **move** se dodaje ispred svakog **closure** gde se koristi
- Ova sintaksa kaže Rust-u da se promenljive ne pozajmljuju, već kradu
- Prvi **closure**, **key_fn** preuzima vlasništvo nad **stat**, dok drugi **closure** preuzima vlasništvo i nad **cities** i nad **key_fn**

HVATANJE PROMENLJIVIH – KRAĐA

- U suštini, kao i svugde u Rust-u, vrednosti se ili pozajmljuju ili se pomeraju i važe sva pravila kao i u svim ostalim slučajevima gde se to koristi
- Naravno, ako su u pitanju **Copy** vrednosti, pomeranje radi njihovo kopiranje, tako da će te vrednosti moći da se koriste i nakon poziva **closure** sa **move** naredbom
- Vrednosti koje se ne mogu kopirati, će se pomeriti, nakon čega se one više ne mogu koristiti u datom dosegu (**cities** se više ne može koristiti u primeru nakon poziva **closure**)
- Ako je potrebno koristiti vrednost i nakon prebacivanja u **closure**, onda je potrebno izvršiti kloniranje te vrednosti, te će **closure** ukrasti samo vrednost jednog od klonova

TIP FUNKCIJE I CLOSURE

- U osnovi, funkcija:

```
fn city_population_descending(city: &City) -> i64 {
    -city.population
}
```

- ima jedan argument, **&City**, i jednu povratnu vrednost, **i64**, odatle sledi da je njen tip: **fn(&City) -> i64**
- Pošto ima tip funkcije, sa funkcijama je moguće činiti sve isto što i sa vrednostima
- Mogu se smestiti u promenljivu i posle se funkcija može proslediti kao parametar druge funkcije:

```
let my_key_fn: fn(&City) -> i64 =
    if user.prefs.by_population {
        city_population_descending
    } else {
        city_monster_attack_risk_descending
    };

cities.sort_by_key(my_key_fn);
```

CLOSURE – FUNKCIJE I TIPOVI

- Strukture mogu imati polja tipa funkcije
- Vektori ili nizovi mogu biti tipa funkcije
- Funkcija može preuzeti drugu funkciju kao argument

```
/// Given a list of cities and a test function,
/// return how many cities pass the test.
fn count_selected_cities(cities: &Vec<City>,
                         test_fn: fn(&City) -> bool) -> usize
{
    let mut count = 0;
    for city in cities {
        if test_fn(city) {
            count += 1;
        }
    }
    count
}
```

CLOSURE – FUNKCIJE I TIPOVI

- Funkcija može preuzeti drugu funkciju kao argument

```
/// An example of a test function. Note that the type of
/// this function is `fn(&City) -> bool`, the same as
/// the `test_fn` argument to `count_selected_cities`.
fn has_monster_attacks(city: &City) -> bool {
    city.monster_attack_risk > 0.0
}

// How many cities are at risk for monster attack?
let n = count_selected_cities(&my_cities, has_monster_attacks);
```

- U ovom konkretnom slučaju vrednost tipa funkcije se ponaša isto kao pokazivači na funkcije u C/C++

CLOSURE – FUNKCIJE I TIPOVI

- **Closure** zapravo nema isti tip kao funkcije

```
let limit = preferences.acceptable_monster_risk();
let n = count_selected_cities(
    &my_cities,
    |city| city.monster_attack_risk > limit);  // error: type mismatch
```

- Da bi se koristio **closure**, mora se promeniti zaglavlje funkcije da sada podržava **closure**

```
fn count_selected_cities<F>(cities: &Vec<City>, test_fn: F) -> usize
    where F: Fn(&City) -> bool
{
    let mut count = 0;
    for city in cities {
        if test_fn(city) {
            count += 1;
        }
    }
    count
}
```

CLOSURE – FUNKCIJE I TIPOVI

- Zaglavlje funkcije se samo promenilo, dok je telo ostalo isto
- Ova verzija je sada generička
- Funkcija sada uzima parametar **test_fn** bilo kog tipa **F** sve dok **F** implementira specijalnu osobinu **Fn(&City) -> bool** (**F** nije greška, treba veliko slovo)
- Ovu osobinu automatski implementira svaka funkcija i većina **closure** anonimnih funkcija koje uzima jedan argument, **&City**, i vraća **Boolean** vrednost

```
fn(&City) -> bool    // fn type (functions only)
Fn(&City) -> bool    // Fn trait (both functions and closures)
```

- Ova specijalna sintaksa je ugrađena u jezik
- Znak **->** i povratni tip su opcioni; ako se izostave, povratna vrednost je **()**

CLOSURE – FUNKCIJE I TIPOVI

- Sada funkcija prihvata **count_selected_cities** prihvata i funkcije i **closure** anonimne funkcije

```
count_selected_cities(
    &my_cities,
    has_monster_attacks);  // ok

count_selected_cities(
    &my_cities,
    |city| city.monster_attack_risk > limit);  // also ok
```

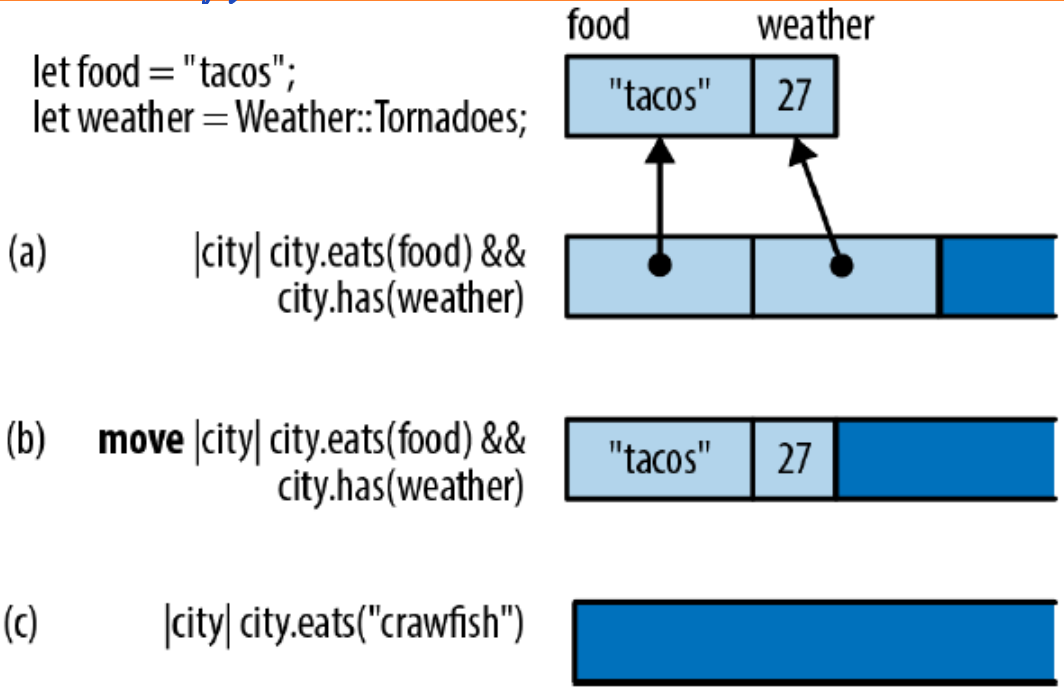
- Zašto ono prvo nije radilo?
- **Closure** može da se poziva, ali nije funkcija (**fn**), **closure** ima sopstveni tip koji nije isti kao i tip funkcije
- Zapravo svaki **closure** predstavlja tip za sebe zato što **closure** može da poseduje podatke koji su ili pozajmljeni ili ukradeni iz dosega unutar koga su definisani (funkcija ima svoj doseg i ima promenljive samo iz tog dosega), pri čemu njihov broj se ne zna unapred

PERFORMANSE CLOSURE ANONIMNIH FUNKCIJA

- Dizajnirane su da budu brze, brže od pokazivača na funkcije, dovoljno brze da se mogu koristiti u vremensko osetljivim situacijama, gde su performanse bitne
- Rust **closure** je sličan C++ lambda funkcijama (brze i kompaktne), samo su sigurnije
- **Closure** funkcije se ne nalaze na hipu (osim ako nisu u Box, Vec, ili drugom kontejneru)
- Kako svaki **closure** ima sopstveni tip, Rust kompajler kada zna tip, može da prevede **closure** i inlajnuje kod na mestu poziva funkcije, zbog toga se **closure** može bezbedno koristiti u petljama (neće uticati na performansu programa)

PERFORMANSE CLOSURE ANONIMNIH FUNKCIJA

- Evo primera:



- U (a) **closure** koristi obe promenljive, memoriji **closure** izgleda kao struktura koja čuva reference na obe promenljive koje koristi
- U (b) **closure** krade promenljive, tako da je slično kao pod (a) samo što su vrednosti sada u **closure**
- U (c) se ne koristi ni jedna promenljiva iz okruženja, pa struktura prazan, a **closure** ne zauzima memoriju uopšte

CLOSURE I BEZBEDNOST

- **Closure** anonimne funkcije u suštini implementiraju 3 vrste osobina
- U zavisnosti od toga kako telo anonimne funkcije upravlja vrednostima, zavisi da li će implementirati jednu, dve ili sve tri osobine
- Da se razumemo, **closure** hvata vrednosti iz sredine u kojoj se poziva i zadržava je, čak i kada se kasnije poziva van te sredine
- Jednom kada je **closure** uhvatilo referencu ili uhvatilo vlasništvo nad nekom vrednošću iz okruženja u kojem je **closure** definisano (čime se određuje da li je bilo šta premešteno u **closure**), kod u telu anonimne funkcije definiše šta se dešava sa referencama ili vrednosti kada se **closure** izvrši kasnije (pri čemu se opet određuje da li se išta izbacuje premešta iz **closure**).
- Telo anonimne funkcije može da uradi nešto od sledećeg: da pomeri uhvaćenu vrednost iz **closure**, da mutira uhvaćenu vrednost, niti da pomeri vrednost, niti da je mutira, ili da, za početak, ne uhvati ništa iz okruženja

CLOSURE I BEZBEDNOST

- Samim tim u zavisnosti šta radi sa datom vrednošću, **closure** implementira sledeće osobine
- **FnOnce** – implementiraju sve **closure** funkcije, jer se svaka od njih poziva, i poziva se makar jednom, ako ne i više puta
  - Ove anonimne funkcije omogućuju pomeranje vrednosti iz njihovog tela (ili njihovo uništavanje)
  - Ove anonimne funkcije implementiraju samo ovu osobinu, jer se mogu pozivati samo jednom (vrednost će biti izbačena iz funkcije, pa će poziv izazvati undefined grešku, što Rust neće dozvoliti)
- **FnMut** – implementiraju **closure** funkcije koje ne dovode do izmeštanja vrednosti iz tela funkcije, ali koje dozvoljavaju izmene nad vrednostima
  - Ove anonimne funkcije se mogu pozivati više nego jednom

CLOSURE I BEZBEDNOST

- **Fn** – implementiraju **closure** funkcije koje niti izmeštaju vrednosti iz svog tela, niti menjaju te vrednosti, kao i funkcije koje ne uzimaju vrednosti iz okruženja
  - Ove anonimne funkcije mogu se pozivati neograničen broj puta i mogu se pozivati u paraleli
  - Ovo anonimne funkcije se mogu kopirati i klonirati

CLOSURE I CALLBACKS

- Veliki broj API poziva se oslanja na poziv funkcija (**callbacks**)
- Standardni način da se to ostvari je nešto slično ovome:

```
App::new()
    .route("/", web::get().to(get_index))
    .route("/gcd", web::post().to(post_gcd))
```

- **get_index** i **post_gcd** su imena funkcija deklarisane negde drugde u programu, ali koje koriste **fn** ključnu reč
- Moguće je staviti umesto funkcija **closure**
- Ali samo zato što **actix-web** napisan tako da primi bilo koju **Fn** implementaciju, pod uslovom da je sigurna za konkurentnu upotrebu (**thread-safe**)

CLOSURE I CALLBACKS

- Implementacija sa **closure**:

```
App::new()
    .route("/", web::get().to(|| {
        HttpResponse::Ok()
            .content_type("text/html")
            .body("<title>GCD Calculator</title>...")
    }))
    .route("/gcd", web::post().to(|form: web::Form<GcdParameters>| {
        HttpResponse::Ok()
            .content_type("text/html")
            .body(format!("The GCD of {} and {} is {}.",
                          form.n, form.m, gcd(form.n, form.m)))
    }))
```

- Ovo se naravno može napraviti i u našem programu, tj. možemo napraviti metode sa funkcijama i **closure** funkcijama kao argumentima

CLOSURE I CALLBACKS

- Simuliraćemo ruter:

```
struct Request {
    method: String,
    url: String,
    headers: HashMap<String, String>,
    body: Vec<u8>
}

struct Response {
    code: u32,
    headers: HashMap<String, String>,
    body: Vec<u8>
}
```

- Posao rutera je da napravi tabelu koja mapira URL adrese na pozive odgovarajućih rutina (zbog jednostavnosti, korisnici mogu da naprave samo jednu rutu koja sadrži samo jednu URL adresu)

CLOSURE I CALLBACKS

- Implementacija jednostavnog rutera:

```
struct BasicRouter<C> where C: Fn(&Request) -> Response {
    routes: HashMap<String, C>
}

impl<C> BasicRouter<C> where C: Fn(&Request) -> Response {
    /// Create an empty router.
    fn new() -> BasicRouter<C> {
        BasicRouter { routes: HashMap::new() }
    }

    /// Add a route to the router.
    fn add_route(&mut self, url: &str, callback: C) {
        self.routes.insert(url.to_string(), callback);
    }
}
```

- Ovde postoji jedna greška, da li je vidite?

CLOSURE I CALLBACKS

- Ako pozovemo ruter prvi put, sve radi:

```
let mut router = BasicRouter::new();
router.add_route("/", |_| get_form_response());
```

- Ali, ako ga ponovo pozovemo:

```
router.add_route("/gcd", |req| get_gcd_response(req));
```

- Dobija se sledeća greška:

```
error[E0308]: mismatched types
  --> closures_bad_router.rs:41:30
   |
41 |     router.add_route("/gcd", |req| get_gcd_response(req));
   |                              ^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |                              expected closure, found a different closure
   |
   = note: expected type `[closure@closures_bad_router.rs:40:27: 40:50]`
              found type `[closure@closures_bad_router.rs:41:30: 41:57]`
note: no two closures, even if identical, have the same type
help: consider boxing your closure and/or using it as a trait object
```

CLOSURE I CALLBACKS

- Greška je samo u tome kako je definisan **BasicRouter** tip:

```
struct BasicRouter<C> where C: Fn(&Request) -> Response {
    routes: HashMap<String, C>
}
```

- Problem je u tome što je napisano da **BasicRouter** tip ima samo jedan tip **C** za **callback** funkciju
- Rešenje je da se podrže svi tipovi kroz boksing i osobine

```
type BoxedCallback = Box<dyn Fn(&Request) -> Response>;

struct BasicRouter {
    routes: HashMap<String, BoxedCallback>
}
```

- Svaki boks može da sadrži različit tip **closure** funkcije, tako da jedna **HashMap** može da sadrži različite vrste **callback** funkcija
- Obratite pažnju da je **C** tip nestao

CLOSURE I CALLBACKS

- Sada se i metode moraju malo modifikovati:

```
impl BasicRouter {
    // Create an empty router.
    fn new() -> BasicRouter {
        BasicRouter { routes: HashMap::new() }
    }

    // Add a route to the router.
    fn add_route<C>(&mut self, url: &str, callback: C)
        where C: Fn(&Request) -> Response + 'static
    {
        self.routes.insert(url.to_string(), Box::new(callback));
    }
}
```

CLOSURE I CALLBACKS

- Sada ruter može da rukuje zahtevima:

```
impl BasicRouter {
    fn handle_request(&self, request: &Request) -> Response {
        match self.routes.get(&request.url) {
            None => not_found_response(),
            Some(callback) => callback(request)
        }
    }
}
```

POKAZIVAČI NA FUNKCIJE

- **Closure** koja ne hvata vrednosti iz okruženja je isto što i pokazivač na funkciju
- **Pokazivač na funkciju** se definiše na sledeći način:

```
fn add_ten(x: u32) -> u32 {
    x + 10
}

let fn_ptr: fn(u32) -> u32 = add_ten;
let eleven = fn_ptr(1); //11
```

- Ako definišite tip funkcije u ili u deklaraciji **closure** ili u njenom vezivanju, kompajler će da je tretira kao pokazivač na funkciju

```
let closure_ptr: fn(u32) -> u32 = |x| x + 1;
let two = closure_ptr(1); // 2
```

- Pokazivači na funkciju zauzimaju jedan **usize**

POKAZIVAČI NA FUNKCIJE

- Pokazivači na funkcije se mogu koristiti za dinamičko preusmeravanje (dynamic dispatch) umesto kompajlerovog sistema (koji je korišćen u slučaju **Box<dyn Fn()>**)

```
struct FnPointerRouter {
    routes: HashMap<String, fn(&Request) -> Response>
}
```

- Ovde sada **HashMap** koristi samo jedan **usize** po stringu i nema boksinga

```
impl FnPointerRouter {
    // Create an empty router.
    fn new() -> FnPointerRouter {
        FnPointerRouter { routes: HashMap::new() }
    }
    // Add a route to the router.
    fn add_route(&mut self, url: &str, callback: fn(&Request) -> Response)
    {
        self.routes.insert(url.to_string(), callback);
    }
}
```

REGULARNI IZRAZI

REGULARNI IZRAZI

- Eksterni **regex** sanduk je Rustova oficijalna biblioteka za implementaciju regularnih izraza
- Omogućuje standardne funkcije za pretraživanje i traženje (i u Unicod i u bajt string formatu)
- Some **regex** je u skladu sa Rust filozofijom, potpuno siguran čak i prilikom pretraživanja nesigurnih izraza unutar nesigurnog teksta (zato je malo ograničeniji od nekih drugih rešenja u drugim programskim jezicima)
- Daćemo samo pregled **regex**, za više konsultujte dokumentaciju
- **regex** se ne nalazi u **std**, održava ga isti tim koji je zadužen za **std**
- Da bi se koristio **regex**, potrebno je staviti sledeći kod unutar **[dependencies]** sekcije **Cargo.toml** fajla projekta:
  regex = "1"

OSNOVNI REGEX IZRAZI

- **regex** vrednost predstavlja izparsiran regularni izraz spreman za upotrebu
- **Regex::new** konstruktor pokušava da izparsira neki string, **&str**, kao regularni izraz, a kao rezultat vraća vrednost tipa **Result**

```
use regex::Regex;

// A semver version number, like 0.2.1.
// May contain a pre-release version suffix, like 0.2.1-alpha.
// (No build metadata suffix, for brevity.)
//
// Note use of r"..." raw string syntax, to avoid backslash blizzard.
let semver = Regex::new(r"(\d+)\.(\d+)\.(\d+)(-[-.[:alnum:]]*)?")?;

// Simple search, with a Boolean result.
let haystack = r#"regex = "0.2.5""#;
assert!(semver.is_match(haystack));
```

OSNOVNI REGEX IZRAZI

- **regex::captures** metoda pretražuje string i traži prvi pogodak i vraća koja sadrži informacije o pogotku za svaku grupu u izrazu:

```
// You can retrieve capture groups:
let captures = semver.captures(haystack)
    .ok_or("semver regex should have matched")?;
assert_eq!(&captures[0], "0.2.5");
assert_eq!(&captures[1], "0");
assert_eq!(&captures[2], "2");
assert_eq!(&captures[3], "5");
```

- Vodite samo računa da indeksiranje nad uhvaćenim izrazom (**captures**) može da izazove paniku ako nije došlo do pogađanja, tako da je bolje koristiti **Captures::get** metodu koja će vratiti **Option**

```
assert_eq!(captures.get(4), None);
assert_eq!(captures.get(3).unwrap().start(), 13);
assert_eq!(captures.get(3).unwrap().end(), 14);
assert_eq!(captures.get(3).unwrap().as_str(), "5");
```

OSNOVNI REGEX IZRAZI

- Iteracija kroz sve pogotke u nekom stringu izgleda ovako:

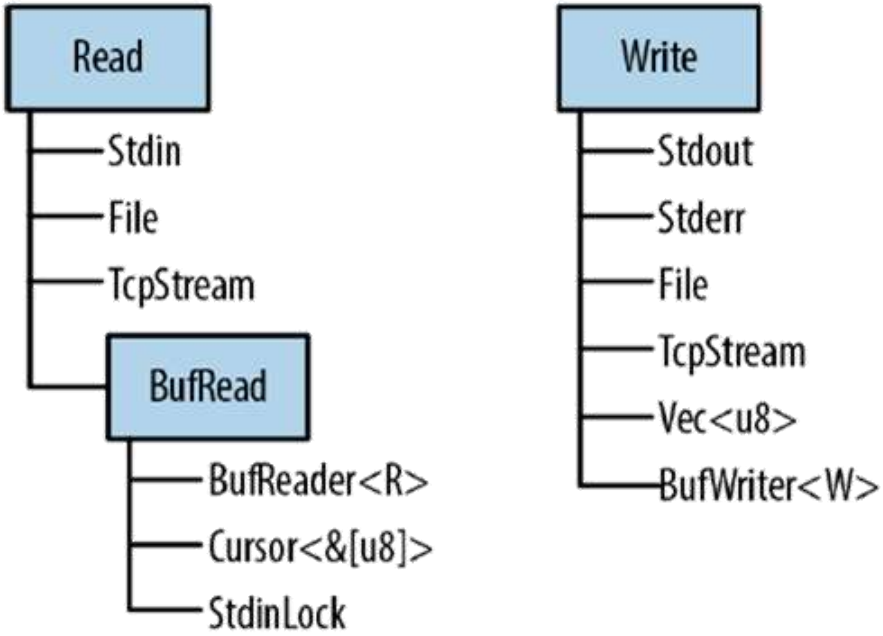
```
let haystack = "In the beginning, there was 1.0.0. \
                For a while, we used 1.0.1-beta, \
                but in the end, we settled on 1.2.4.";

let matches: Vec<&str> = semver.find_iter(haystack)
    .map(|match_| match_.as_str())
    .collect();
assert_eq!(matches, vec!["1.0.0", "1.0.1-beta", "1.2.4"]);
```

- Iterator **find_iter** stvara **Match** vrednost za svaki nepreklapajući pogodak u izrazu počev od početka do kraja stringa
- Metoda **captures_iter** je slična, samo proizvodi **Captures** vrednosti za sve uhvaćene grupe

ULAZ/IZLAZ

ULAZ/IZLAZ

- Rad sa ulazom i izlazom je u Rust-u organizovan oko tri osobine iz standardne biblioteke
  - **Read** – metode za čitanje bajt vrednosti **(readers)**
  - **BufRead** – nadograđuju obične čitače metodama za čitanje bafera poput linija teksta i sl. (**buffered** readers)
  - **Write** – metode za pisanje koje podržavaju pisanje bajt vrednosti i UTF-8 teksta (**writers**)



READERS / WRITERS

- **Readers** su vrednosti iz kojih program može da čita bajt vrednosti
  - **std::fs::File::open(filename)** se koristi za otvaranje fajlova
  - **std::net::TcpStreams** se koristi za preuzimanje podataka preko mreže
  - **std::io::stdin()** se koristi za čitanja iz standardnog ulaznog toka procesa (**process's standard input stream**)
  - **std::io::Cursor<&[u8]>** i **std::io::Cursor<Vec<u8>>** se koriste za čitanje niz bajtova ili vektora bajtova koji se već nalaze u memoriji

READERS / WRITERS

- **Writers** su vrednosti u koje program upisuje bajt vrednosti
  - **std::fs::File::create(filename)** se koristi za otvaranje fajlova
  - **std::net::TcpStreams** se koristi za slanje podataka preko mreže
  - **std::io::stdout()** i **std::io::stderr()** se koriste za pisanje podataka na standardni ulazni tok procesa, tj. terminal
  - **Vec<u8>** je writer čije se **write** metode koriste za proširivanje vektora
  - **std::io::Cursor<Vec<u8>>**, je sličan prethodnom ali omogućava i čitanja i pisanje podataka, kao i pomeranje kursora (**seek**) unutar fajla
  - **std::io::Cursor<&mut [u8]>**, koji je sličan prethodnom samo što ne može da proširuje bafer, jer je on samo isečak već postojećeg niza bajtova

READERS / WRITERS

- Kako je ovo sve poprilično standardno i kako postoje standardni čitači i pisači (**std::io::Read** i **std::io::Write**) lako je napisati generički kod koji radi na više različitih ulaznih i izlaznih kanala
- Primer se nalazi na sledećem slajdu i predstavlja implementaciju **std::io::copy()** metode iz standardne Rust biblioteke
- Pošto je metoda dovoljno generička, može se koristiti za kopiranje podataka iz **File** tipa u **TcpStream** tip, ili iz **Stdin** toka u vektor u memoriji **Vec<u8>** i slično
- Upotreba **std::io** osobina **Read**, **BufRead** i **Write**, zajedno sa **Seek** je toliko uobičajena da postoji poseban prelude module koji sadrži ove osobine: `use std::io::prelude::*;`
- Često se **std::io** modul učitava: `use std::io::{self, Read, Write, ErrorKind};` ovo se radi kako bi se **self** koristio kao alijas na sam **std::io** modul što olakšava upotrebu **std::io::Result** i **std::io::Error** u skraćenom obliku **io::Result** i **io::Error**

READERS / WRITERS


```rust
use std::io::{self, Read, Write, ErrorKind};

const DEFAULT_BUF_SIZE: usize = 8 * 1024;

pub fn copy<R: ?Sized, W: ?Sized>(reader: &mut R, writer: &mut W)
    -> io::Result<u64>
    where R: Read, W: Write
{
    let mut buf = [0; DEFAULT_BUF_SIZE];
    let mut written = 0;
    loop {
        let len = match reader.read(&mut buf) {
            Ok(0) => return Ok(written),
            Ok(len) => len,
            Err(ref e) if e.kind() == ErrorKind::Interrupted => continue,
            Err(e) => return Err(e),
        };
        writer.write_all(&buf[..len])?;
        written += len as u64;
    }
}
```

READERS

- **std::io::Read**
- Rust ima nekoliko metoda koje implementiraju čitanje podataka
- Svi preuzimaju **reader** vrednost kroz **mut** referencu
- **reader.read(&mut buffer)**
  - Metoda za rad sa bajtovima na najnižem nivou apstrakcije, daje punu kontrolu nad transferom bajtova, ali može da bude vrlo komplikovana za implementaciju
  - Čita bajtove iz **reader** vrednosti i smešta ih u bafer, argument mora biti tipa **&mut [u8]**
  - Čita se do **buffer.len()** bajtova
  - Kao rezultat vraća **io::Result<u64>** što je alias for **Result<u64, io::Error>**
  - Rust ima druge metode na višem nivou apstrakcije koje se oslanjaju na ovu metodu i koje imaju predefinisani implementaciju **.read()** i sve implementiraju rukovalac za grešku **ErrorKind::Interrupted**

READERS

- **reader.read_to_end(&mut byte_vec)**
  - Čita sve preostale vrednosti iz **reader** vrednosti i dodaje ih (**append**) na kraj **byte_vec** koji je tipa **Vec<u8>**
  - Vraća **io::Result<usize>**, broj pročitanih bajtova
  - Ne postoji limit na broj pročitanih bajtova (kao ni na broj koji će se smestiti u bafer), tako da ovu metodu treba pažljivo koristiti, naročito sa izvorima koji nisu pouzdani
- **reader.read_to_string(&mut string)**
  - Isto kao i prethodni samo proširuje string
  - Ako string nije validan UTF-8 format, baca grešku **ErrorKind::InvalidData**
- **reader.read_exact(&mut buf)**
  - Pročita tačno toliko bajtova iz **reader** vrednosti koliko je potrebno da se popuni bafer tipa **&[u8]**
  - Ako nema dovoljno bajtova da se popuni bafer, baca grešku **ErrorKind::UnexpectedEof**

READERS

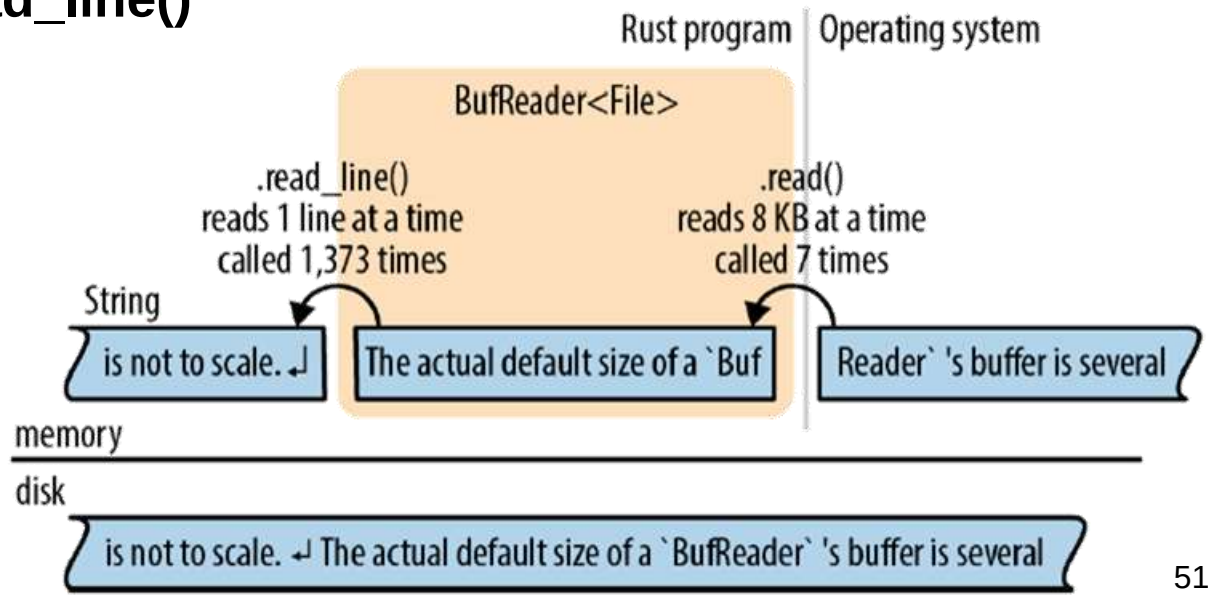
- Pored ovih metoda, **Read** osobina sadrži i tri adaptera koji preuzimaju **Reader** vrednost i pretvaraju je u iterator ili **Reader** drugog tipa
- **reader.bytes()**
  - Vraća iterator nad bajtovima ulaznog toka
  - Povratna vrednost je **io::Result<u8>**, tako da je potrebno proveravati rezultat za svaku pojedinačnu povratnu vrednost
  - Ovo poziva **reader.read()** za svaki pojedinačni bajt, što može biti jako neefikasno ako vrednosti nisu baferovane
- **reader.chain(reader2)**
  - Vraća novu **Reader** vrednost koja će vratiti sve **reader** ulazne vrednosti plus sve **reader2** ulazne vrednosti
- **reader.take(n)**
  - Vraća novi **Reader** koji će čitati isti izvor kao i **reader**, ali limitarn na prvih **n** bajtova izvora

READERS

- **Reader** se ne može zatvoriti, tj. ne postoji metoda za eksplicitno zatvaranja **Reader** vrednosti
- Ono kao deo Rusta ima mogućnost da implementira **Drop** osobinu, što većina **Read** implementacija i implementira

BUFFER READERS

- Zbog efikasnosti, i **Reader** i **Writer** mogu biti baferovani
- Oni imaju bafer (parče memorije) u koji se koristi za čuvanje ulaznih i izlaznih podataka
- Aplikacija pročita podatke iz **BufReader**, npr. pozivom **.read_line()**, a **BufReader** preuzima podatke veću količinu podataka od OS-a
- Veličina bafera iznosi oko nekoliko kilobajta, tako da jedan **read** može da opsluži stotine **.read_line()** poziva (ovo značajno ubrzava stvari, jer se zaobilaze spori OS pozivi)
- Implementira sve što i **Read** plus **BufRead**



BUFFER READERS

- **reader.read_line(&mut line)**
  - Čita red teksta i dodaje **(append)** ga na **line** koji je tipa **String**
  - Vraća **io::Result<usize>**, broj pročitanih bajtova
  - Ako je **Reader** na kraju ulaza, **line** se ne manja i vraća se **Ok(0)**
- **reader.lines()**
  - Vraća iterator nad linijama ulaznog toka
  - Povratna vrednost je **io::Result<String>**, tako da je potrebno proveravati rezultat za svaku pojedinačnu povratnu vrednost
  - Ovo je ono što gotovo uvek želite kada radite sa tekstom
- **reader.read_until(stop_byte, &mut byte_vec), reader.split(stop_byte)**
  - Rade isto kao **read_line()** i **.lines()** ali su bajt orijentisani
  - Vraćaju **Vec<u8>** vektore umesto **String**
  - Delimiter **stop_byte** birate sami

BUFFER READERS

- Primer čitanje linija i traženje određenog stringa

```
use std::io;
use std::io::prelude::*;

fn grep(target: &str) -> io::Result<()> {
    let stdin = io::stdin();
    for line_result in stdin.lock().lines() {
        let line = line_result?;
        if line.contains(target) {
            println!("{}", line);
        }
    }
    Ok(())
}
```

- **io::stdin()** se koristi da napravimo **BufRead** vrednost i da bi došli do ulaznih vrednosti
- Rust štiti **stdin** muteksom, tako da se mora zaključati za siguran konkurentni rad sa **.lock()**

BUFFER READERS

- Metoda se može učiniti i generičkom za čitanje bilo kog fajla s diska

```
fn grep<R>(target: &str, reader: R) -> io::Result<()>
    where R: BufRead
{
    for line_result in reader.lines() {
        let line = line_result?;
        if line.contains(target) {
            println!("{}", line);
        }
    }
    Ok(())
}
```

- Sada može proslediti i **Stdinlock** i baferovan **File**:

```
let stdin = io::stdin();
grep(&target, stdin.lock())?;  // ok

let f = File::open(file)?;
grep(&target, BufReader::new(f))?;  // also ok
```

BUFFER READERS

- **File** nije baferovan po automatizmu
- **File** implementira **Read** ali ne i **BufRead**
- Lako se kreira baferovan reader za **File** (ili za bilo koji drugi nebaferovani reader)
- To se odradi pozivom **BufReader::new(reader)** ili pozivom **BufReader::with_capacity(size, reader)** za bafer željene veličine
- U Rustu je namerno napravljeno da je čitanje fajlova nebaferovano i da se mora eksplicitno preći na baferovanu verziju
- Čitav kod se nalazi u nastavku

BUFFER READERS


```
use std::error::Error;
use std::io::{self, BufReader};
use std::io::prelude::*;
use std::fs::File;
use std::path::PathBuf;

fn grep<R>(target: &str, reader: R) -> io::Result<()>
    where R: BufRead
{
    for line_result in reader.lines() {
        let line = line_result?;
        if line.contains(target) {
            println!("{}", line);
        }
    }
    Ok(())
}
```

BUFFER READERS


```
fn grep_main() -> Result<(), Box<dyn Error>> {
    // Get the command-line arguments. The first argument is the
    // string to search for; the rest are filenames.
    let mut args = std::env::args().skip(1);
    let target = match args.next() {
        Some(s) => s,
        None => Err("usage: grep PATTERN FILE...")?
    };
    let files: Vec<PathBuf> = args.map(PathBuf::from).collect();
    if files.is_empty() {
        let stdin = io::stdin();
        grep(&target, stdin.lock())?;
    } else {
        for file in files {
            let f = File::open(file)?;
            grep(&target, BufReader::new(f))?;
        }
    }

    Ok(())
}
```

BUFFER READERS


```rust
fn main() {
    let result = grep_main();
    if let Err(err) = result {
        eprintln!("{}", err);
        std::process::exit(1);
    }
}
```

BUFFER READERS

- Čitanje (preuzimanje) linije pomoću **.lines()** može da bude problem zato što će napraviti iterator koji rezultuje **Result** vrednošću što može da bude zaista problematično i iritantno kada želite da to pokupite u jedan veliki vektor
- Ovo je ok, li nije baš to što obično želimo

```
// ok, but not what you want
let results: Vec<io::Result<String>> = reader.lines().collect();

// error: can't convert collection of Results to Vec<String>
let lines: Vec<String> = reader.lines().collect();
```

- Alternativa je da se upotrebi **for** petlja i da se svaka povratna vrednost proveri spram greške

```
let mut lines = vec![];
for line_result in reader.lines() {
    lines.push(line_result?);
}
```

BUFFER READERS

- Ipak, može još bolje pomoću same **collect** metode:

```
let lines = reader.lines().collect::<io::Result<Vec<String>>>()?;
```

- Ovo može, zato što postoji implementacije **FromIterator** osobine za **Result** unutar standardne biblioteke

```
impl<T, E, C> FromIterator<Result<T, E>> for Result<C, E>
    where C: FromIterator<T>
{
    ...
}
```

- Sve što treba je da se izvuku vrednosti iz iteratora i da se izgradi kolekcija iz Ok dela rezultata
- Ako se naleti na **Err**, zaustavi se i prosledi se greška dalje
- Kako je **io::Result<Vec<String>>** kolekcija, **.collect()** metoda može da je kreira i popuni na opisan način

WRITERS

- Čitanje se uglavnom realizuje kroz metode, dok je pisanje komplikovanije
- Zbog toga se u većini jednostavnih situacija koriste makroi, **write!()** i **writeln!()**,

```
writeln!(io::stderr(), "error: world not helloable")?;

writeln!(&mut byte_vec, "The greatest common divisor of {:?} is {}",
         numbers, d)?;
```

- Oni su slični **print!() i println!()** makroima

```
println!("Hello, world!");

println!("The greatest common divisor of {:?} is {}",
         numbers, d);

println!();  // print a blank line
```

- Razlika je u tome što potrebno proslediti **writer** i što vraća rezultat tipa **Result**

WRITERS

- **Write** osobina ima sledeće metode
- **writer.write(&buf)**
  - Ispisuje neke od bajtova iz isečka **&buf** na izlazni tok
  - Vraća rezultat tipa **io::Result<usize>** koji ako je ispis bio uspešan (vratio **Ok**), vraća broj ispisanih bajtova koji je manji ili jednak **buf.len()** (koliko će se bajtova ispisati zavisi od samog toka na koji se ispisuje)
  - Ovo je metoda najnižeg nivoa apstrakcije
- **writer.write_all(&buf)**
  - Ispisuje sve bajtove iz isečka **&buf** na izlazni tok
  - Vraća rezultat tipa **Result<()>**
- **writer.flush()**
  - Izbacuje sve baferovane podatke na odgovarajući tok
  - Vraća rezultat tipa **Result<()>**
- Kao i **reader**, **writer** se zatvara automatski kad se desi **drop**

RAD SA FAJLOVIMA

- Do sada su bila dva primera rada sa fajlovima:
- **File::open(filename)**
  - Otvara fajl za čitanje
  - Vraća **io::Result<File>** kao rezultat, nepostojanje fajla izaziva grešku
- **File::create(filename)**
  - Kreira novi fajl za pisanje
  - Ako fajl postoji, onda ga otvara za pisanje, pri čemu se briše stari fajl
- **File** tip se nalazi u modulu za rad sa fajlovima, **std::fs**
- Ako ni **.open()**, ni **.create()** ne rade posao, onda se može koristiti **OpenOptions** preko koje se može definisati kako se želi otvoriti fajl

RAD SA FAJLOVIMA

- Primer sa **OpenOptions**

```
use std::fs::OpenOptions;

let log = OpenOptions::new()
    .append(true)  // if file exists, add to the end
    .open("server.log")?;

let file = OpenOptions::new()
    .write(true)
    .create_new(true)  // fail if file exists
    .open("new_file.txt")?;
```

- Metode **.append()**, **.write()**, **.create_new()**, kao i ostale iz ove grupe su dizajnirane da se koriste u nizu, jer svaka vraća **self**
- Ovaj niz komandi se u Rust-u naziva **builder**
- Kada se fajl otvori, ponaša se kao bilo koji **reader** i **writer**

RAD SA FAJLOVIMA

- Premotavanje (**seeking**) je deo **Seek** osobine, što omogućuje da se premotava fajl (prolazi kroz fajl)
- Omogućava da se pomerite bilo gde u fajlu
- Definisan je na sledeći način:

```
pub trait Seek {
    fn seek(&mut self, pos: SeekFrom) -> io::Result<u64>;
}

pub enum SeekFrom {
    Start(u64),
    End(i64),
    Current(i64)
}
```

- Sva ekspresivnog **seek** funkcije proizilazi iz enumeracije
  - **file.seek(SeekFrom::Start(0))** – pomeranje na početak fajla
  - **file.seek(SeekFrom::Current(-8))** – pomeranje za 8 bajta unazad

DODATCI

- Pored opisanih, postoji još gomila drugih **Reader** i **Writer** tipova
  - **io::stdin()**
  - **io::stdout(), io::stderr()**
  - **Vec<u8>**
  - **Cursor::new(buf)**
  - **std::net::TcpStream**
  - **std::process::Command**
  - **io::sink()**
  - **io::empty()**
  - **io::repeat(byte)**

DODATCI

- Postoje sanduci koji nadograđuju **std::io** i pružaju dodatne funkcije
  - **ReadBytesExt** i **WriteBytesExt** daju podrški za rad sa binarnim ulazom i izlazom
  - **flate2** daje podršku za rad sa **gzipped** podacima
  - s**erde** daje podršku za **serde_json**, serijalizaciju i deserijalizaciju
  - **std::path** i **std::fs** daju podršku za rad sa fajl sistemom (fajlovima i folderima)

MREŽNO PROGRAMIRANJE

MREŽNO PROGRAMIRANJE

- Samo bazično, taman da se shvati kako to radi, ako ste ikada programirali nešto ovog tipa
- Za mrežno programiranje niskog nivoa (**low-level networking code**) upotrebite **std::net** modul
  - Omogućuje multiplatformsku podršku **TCP** i **UDP**
- Za **SSL/TLS** se koristi **native_tls** modul
- Ovi moduli i odgovarajuće osobine i njihove metode se koriste za jednostavno, direktno i blokovsko slanje i čitanje podataka sa mreže
- Jednostavni serveri se vrlo lako mogu implementirati
- Na sledećim slajdovima sledi primer kako se **std::net** koristi za izgradnju jednostavno „eho“ servera

```
use std::net::TcpListener;
use std::io;
use std::thread::spawn;
```

MREŽNO PROGRAMIRANJE

- „eho“ server

```
/// Accept connections forever, spawning a thread for each one.
fn echo_main(addr: &str) -> io::Result<()> {
    let listener = TcpListener::bind(addr)?;
    println!("listening on {}", addr);
    loop {
        // Wait for a client to connect.
        let (mut stream, addr) = listener.accept()?;
        println!("connection received from {}", addr);

        // Spawn a thread to handle this client.
        let mut write_stream = stream.try_clone()?;
        spawn(move || {
            // Echo everything we receive from `stream` back to it.
            io::copy(&mut stream, &mut write_stream)
                .expect("error in client thread: ");
            println!("connection closed");
        });
    }
}
```

MREŽNO PROGRAMIRANJE

- „eho“ server main

```
fn main() {
    echo_main("127.0.0.1:17007").expect("error: ");
}
```

- „eho“ server jednostavno ponovi sve što dobije na ulazu
- Za svaku konekciju se pokreće zasebna nit pomoću **std::thread::spawn()**
- Za efikasno serversko programiranje, potrebno je koristiti asinhronu komunikaciju što ćemo isto učiti kasnije
- Viši komunikacioni protokoli su implementirani sanducima drugih autora
- Npr., **reqwest** sanduk implementira **API** za **HTTP** klijente
- U nastavku sledi primer za dobavljanje dokumenata preko **http:** ili **https:** URLa i njegovo ispisivanje u terminal

MREŽNO PROGRAMIRANJE

- **reqwest** podržava asinhronu komunikaciju i izvršavanje
- Primer http klijenta implementiranog pomoću **reqwest = "0.11"**, sa "blocking„ opcijom

```
use std::error::Error;
use std::io;

fn http_get_main(url: &str) -> Result<(), Box<dyn Error>> {
    // Send the HTTP request and get a response.
    let mut response = reqwest::blocking::get(url)?;
    if !response.status().is_success() {
        Err(format!("{}", response.status()))?;
    }
    // Read the response body and write it to stdout.
    let stdout = io::stdout();
    io::copy(&mut response, &mut stdout.lock())?;

    Ok(())
}
```

MREŽNO PROGRAMIRANJE

- Main klijenta

```
fn main() {
    let args: Vec<String> = std::env::args().collect();
    if args.len() != 2 {
        eprintln!("usage: http-get URL");
        return;
    }

    if let Err(err) = http_get_main(&args[1]) {
        eprintln!("error: {}", err);
    }
}
```

- Vrlo korisni su još i **actix-web** koji sadrži osobine za implementaciju HTTP servera i **websocket** sanduk za implementaciju **WebSocket** protokola