POGLAVLJE 2

Kratka tura kroz Rust

Rust postavlja pred autore knjige poput ove izazov: ono što jeziku daje karakter nije neka specifična, neverovatna osobina koju možemo da pokažemo na prvoj stranici, već način na koji su delovi dizajnirani da neometano rade zajedno u službi ciljeva koje smo postavili u prethodnom poglavlju: bezbedno, efikasno programiranje sistema. Svaki deo jezika je najbolje opravdan u kontekstu svih ostalih.

Dakle, umesto da se bavimo pojednim jezičkim karakteristikama, pripremili smo nekoliko malih, ali kompletnih programa, od kojih svaki uvodi neke karakteristike jezika, u kontekstu:

- Za zagrevanje, imamo program koji radi jednostavnu kalkulaciju svojih argumenata komandne linije, sa jediničnim testovima. Ovo pokazuje Rustove osnovne tipove i uvodi *osobine*.
- Dalje, gradimo veb server. Koristićemo biblioteku treće strane da odradimo detalje HTTP-a i uvedemo rukovanje stringovima, zatvaranja i rukovanje greškama.
- Naš treći program iscrtava prelep fraktal, distribuirajući proračun u više niti radi brzine. Sadrži primer generičke funkcije, ilustruje kako se rukuje nečim poput bafera piksela i pokazuje Rustovu podršku za istovremenost (konkurentnost).
- Konačno, prikazujemo robustan alat na komandnoj liniji koji obrađuje datoteke pomoću regularnih izraza. Ovo predstavlja mogućnosti Rust standardne biblioteke za rad sa datotekama i najčešće korišćenu biblioteku regularnih izraza treće strane.

Rustovo obećanje (eng. primise) da će sprečiti nedefinisano ponašanje sa minimalnim učinkom na performanse utiče na dizajn svakog dela sistema, od standardnih struktura podataka poput vektora i stringova do načina na koji Rust programi koriste biblioteke trećih strana. Detalji o tome kako se tim upravlja su obuhvaćeni u celoj knjizi. Ali za sada želimo da vam pokažemo da je Rust sposoban i prijatan jezik za upotrebu.

Prvo, naravno, morate da instalirate Rust na računar.

rustup i Cargo

Najbolji način da instalirate Rust je da koristite `rustup`. Idite na *https://rustup.rs* i pratite uputstva.

Možete, alternativno, da odete na *Rust veb lokaciju (https://www.rust-lang.org)* da dobijete unapred napravljene pakete za Linux, macOS i Windows. Rust je takođe uključen u neke distribucije operativnog sistema. Više volimo `rustup` jer je to alatka za upravljanje Rust instalacijama, kao što je RVM za Ruby ili NVM za Node. Na primer, kada bude objavljena nova verzija Rusta, moći ćete da uradite nadogradnju sa nula klikova tako što ćete uneti `rustup update`.

U svakom slučaju, kada završite instalaciju, trebalo bi da imate tri nove komande dostupne na komandnoj liniji:

```
$ cargo --version
cargo 1.56.0 (4ed5d137b 2021-10-04)
$ rustc --version
rustc 1.56.0 (09c42c458 2021-10-18)
$ rustdoc --version
rustdoc 1.56.0 (09c42c458 2021-10-18)
```

Ovde, `$` je komandna linija; na Windowsu, to bi bilo `C:\>` ili nešto slično. U ovom transkriptu pokrećemo tri komande koje smo instalirali, tražeći od svake da prijavi koja je verzija. Pogledajmo svaku komandu redom:

- `cargo` je Rustov kompilacijski menadžer, menadžer paketa i alat opšte namene. Možete da koristite Cargo da započnete novi projekat, izgradite i pokrenete svoj program i upravljate spoljnim bibliotekama od kojih zavisi vaš kod.
- `rustc` je Rust kompajler. Obično pustimo Cargo da pozove kompajler umesto nas, ali ponekad je korisno da ga pokrenemo direktno.
- `rustdoc` je alatka za Rust dokumentaciju. Ako napišete dokumentaciju u komentarima odgovarajućeg oblika u izvornom kodu vašeg programa, `rustdoc` može da napravi lepo formatiran HTML od njih. Poput `rustc`, obično puštamo Cargo da izvrši `rustdoc` za nas.

Kao pogodnost, Cargo može da napravi novi Rust paket za nas, sa nekim standardnim metapodacima uređenim na odgovarajući način:

```
$ cargo new hello
     Created binary (application) `hello` package
```

Ova komanda kreira novi direktorijum paketa pod nazivom *hello*, spreman za pravljenje izvršnog fajla komandne linije.

Gledajući u direktorijum najvišeg nivoa paketa:

```
$ cd hello
$ ls -la
total 24
drwxrwxr-x.  4 jimb jimb 4096 Sep 22 21:09 .
drwx------. 62 jimb jimb 4096 Sep 22 21:09 ..
drwxrwxr-x.  6 jimb jimb 4096 Sep 22 21:09 .git
-rw-rw-r--.  1 jimb jimb    7 Sep 22 21:09 .gitignore
-rw-rw-r--.  1 jimb jimb   88 Sep 22 21:09 Cargo.toml
drwxrwxr-x.  2 jimb jimb 4096 Sep 22 21:09 src
```

Možemo da vidimo da je Cargo napravio datoteku *Cargo.toml* za čuvanje metapodataka za paket. Trenutno ova datoteka ne sadrži mnogo:

```
[package]
name = "hello"
version = "0.1.0"
edition = "2021"

# Za više detalja pogledajte na
# https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
```

Ako su našem programu ikada potrebne zavisnosti od drugih biblioteka, možemo ih snimiti u ovu datoteku, a Cargo će se pobrinuti za preuzimanje, izgradnju i ažuriranje tih biblioteka umesto nas. Detaljno ćemo pokriti datoteku *Cargo.toml* u Poglavlju 8.

Cargo je podesio naš paket za korišćenje sa `git` sistemom kontrole verzija, kreirajući *.git* poddirektorijum metapodataka i *.gitignore* datoteku. Možete reći Cargu da preskoči ovaj korak tako što ćete proslediti `--vcs none` u `cargo new` na komandnoj liniji.

Poddirektorijum *src* sadrži stvarni Rust kôd:

```
$ cd src
$ ls -l
total 4
-rw-rw-r--. 1 jimb jimb 45 Sep 22 21:09 main.rs
```

Izgleda da je Cargo počeo da piše program u naše ime. Datoteka *main.rs* sadrži tekst:

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

U Rustu, ne morate čak ni da pišete svoj „Hello, world!“ program. A ovo je šablon za novi Rust program: dva fajla, ukupno trinaest redova.

Možemo da pozovemo komandu `cargo run` iz bilo kog direktorijuma u paketu da bismo napravili i pokrenuli naš program:

```
$ cargo run
   Compiling hello v0.1.0 (/home/jimb/rust/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.28s
     Running `/home/jimb/rust/hello/target/debug/hello`
Hello, world!
```

Ovde, Cargo je pozvao Rust kompajler, `rustc`, a zatim pokrenuo izvršni fajl koji je proizveo. Cargo postavlja izvršni fajl u poddirektorijum *target* na vrhu paketa:

```
$ ls -l ../target/debug
total 580
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 build
```

```
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 deps
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 examples
-rwxrwxr-x. 1 jimb jimb 576632 Sep 22 21:37 hello
-rw-rw-r--. 1 jimb jimb    198 Sep 22 21:37 hello.d
drwxrwxr-x. 2 jimb jimb     68 Sep 22 21:37 incremental
$ ../target/debug/hello
Hello, world!
```

Kada završimo, Cargo može umesto nas da očisti generisane datoteke:

```
$ cargo clean
$ ../target/debug/hello
bash: ../target/debug/hello: No such file or directory
```

Rust funkcije

Rustova sintaksa je namerno neoriginalna. Ako ste upoznati sa C, C++, Javom ili JavaScriptom, verovatno možete pronaći put kroz opštu strukturu Rust programa. Evo funkcije koja izračunava najveći zajednički delilac dva cela broja, koristeći Euklidov algoritam (*https://oreil.ly/DFpib*). Ovaj kôd možete dodati na kraj *src/main.rs*:

```
fn gcd(mut n: u64, mut m: u64) -> u64 {
    assert!(n != 0 && m != 0);
    while m != 0 {
        if m < n {
            let t = m;
            m = n;
            n = t;
        }
        m = m % n;
    }
    n
}
```

Rezervisana reč `fn` (izgovara se kao englesko fun, što znači. zabava) označava funkciju. Ovde definišemo funkciju pod nazivom `gcd`, koja uzima dva parametra `n` i `m`, od kojih je svaki tipa `u64`, 64-bitni ceo broj bez predznaka. Token `->` prethodi povratnom tipu: naša funkcija vraća vrednost `u64`. Uvlaka sa četiri razmaka je standardni Rust stil.

Imena Rustovog mašinskog celog broja odražavaju njihovu veličinu i predznak: `i32` je 32-bitni ceo broj sa predznakom; `u8` je 8-bitni ceo broj bez predznaka (koristi se za bajt vrednosti) i tako dalje. Tipovi `isize` i `usize` sadrže cele brojeve veličine pokazivača, sa i bez predznaka, duge 32 bita na 32-bitnim platformama i 64 bita na 64-bitnim platformama. Rust takođe ima dva tipa sa pomičnim zarezom, `f32` i `f64`, koji su IEEE tipovi sa pokretnim zarezom jednostruke i dvostruke preciznosti, kao što su `float` i `double` u C i C++.

Podrazumevano, kada se promenljiva inicijalizuje, njena vrednost se ne može promeniti, ali stavljanjem rezervisane reči `mut` (izgovara se kao englesko mute, skraćenica za *mutable*, izmenjivo) ispred parametara `n` i `m` omogućavaju našem telu funkcije da im dodeli vrednost.

U praksi, većini varijabli se ne dodeljuju; `mut` rezervisana reč može biti od pomoći prilikom čitanja koda.

Telo funkcije počinje pozivom `assert!` makroa, proveravajući da nijedan argument nije nula. `!` znak označava ovo kao pozivanje makroa, a ne poziv funkcije. Poput makroa `assert` u C i C++, Rustov `assert!` proverava da li je njegov argument tačan, a ako nije, završava program sa korisnom porukom koja sadrži izvornu lokaciju netačnog argumenta; ova vrsta naglog prekida naziva se *panika*. Za razliku od C i C++, u kojima se tvrdnje (eng. assertions) mogu preskočiti, Rust uvek proverava tvrdnje bez obzira na to kako je program kompajliran. Takođe postoji `debug_assert!` makro, čije se tvrdnje preskaču kada se program kompajlira radi brzine.

Srce naše funkcije je `while` petlja koja sadrži `if` naredbu i dodelu. Za razliku od C i C++, Rust ne zahteva zagrade oko uslovnih izraza, ali zahteva vitičaste zagrade oko izjava koje kontrolišu.

Izraz `let` deklariše lokalnu promenljivu, kao `t` u našoj funkciji. Ne moramo da ispisujemo tip `t`, sve dok Rust može da zaključi iz toga kako se promenljiva koristi. U našoj funkciji, jedini tip koji funkcioniše za `t` je `u64`, koji odgovara `i` za `m` i `n`. Rust zaključuje o tipovima unutar tela funkcije: morate ispisati tipove parametara funkcije i povratne vrednosti, kao što smo ranije radili. Ako želimo da napišemo tip `t`, mogli bismo da napišemo:

```
let t: u64 = m;
```

Rust ima naredbu `return`, ali funkciji `gcd` ona nije potrebna. Ako se telo funkcije završava izrazom koji *nije* praćen tačkom i zarezom, to je povratna vrednost funkcije. U stvari, svaki blok okružen vitičastim zagradama može da funkcioniše kao izraz. Na primer, ovo je izraz koji ispisuje poruku i zatim daje `x.cos()` kao svoju vrednost:

```
{
    println!("evaluating cos x");
    x.cos()
}
```

Uobičajeno je za Rust koristiti ovaj obrazac za dobijanje vrednosti funkcije i korišćenje `return` iskaze samo za eksplicitne rane povratke iz sredine funkcija.

Pisanje i izvođenje jediničnih testova

Rust ima jednostavnu podršku za testiranje ugrađenu u jezik. Da bismo testirali našu `gcd` funkciju, možemo dodati ovaj kod na kraj *src/main.rs*:

```
#[test]
fn test_gcd() {
    assert_eq!(gcd(14, 15), 1);

    assert_eq!(gcd(2 * 3 * 5 * 11 * 17,
                   3 * 7 * 11 * 13 * 19),
               3 * 11);
}
```

Ovde definišemo funkciju pod nazivom `test_gcd`, koja poziva `gcd` i proverava da li vraća ispravne vrednosti. `#[test]` na vrhu definicije označava `test_gcd` kao test funkciju, koja se preskače u normalnim kompilacijama, ali se uključuje i poziva automatski ako pokrenemo naš program sa `cargo test` komandom. Možemo imati test funkcije razbacane po našem kodu, postavljene pored koda koji isputujemo, a `cargo test` će ih automatski prikupiti i pokrenuti ih sve.

Oznaka `#[test]` je primer *atributa*. Atributi su otvoreni sistem za označavanje funkcija i drugih deklaracija sa dodatnim informacijama, kao što su atributi u C++ i C#, ili napomene (eng. annotations) u Javi. Koriste se za kontrolu upozorenja kompajlera i provere stilova koda, uslovno uključuju kôd (kao `#ifdef` u C i C++), govore Rustu kako da komunicira sa kodom napisanim na drugim jezicima, itd. Videćemo više primera atributa kako napredujemo kroz tekst.

Sa našim `gcd` i `test_gcd` definicijama dodatim u paket *hello* koji smo kreirali na početku poglavlja, a naš trenutni direktorijum negde u podstablu paketa, možemo pokrenuti testove na sledeći način:

```
$ cargo test
   Compiling hello v0.1.0 (/home/jimb/rust/hello)
    Finished test [unoptimized + debuginfo] target(s) in 0.35s
     Running unittests (/home/jimb/rust/hello/target/debug/deps/hello-2375...)

running 1 test
test test_gcd ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Rukovanje argumentima komandne linije

Da bi naš program uzeo niz brojeva kao argumente komandne linije i ispisao njihov najveći zajednički delilac, možemo da zamenimo `main` funkciju u *src/main.rs* sa sledećim:

```
use std::str::FromStr;
use std::env;

fn main() {
    let mut numbers = Vec::new();

    for arg in env::args().skip(1) {
        numbers.push(u64::from_str(&arg)
                     .expect("error parsing argument"));
    }

    if numbers.len() == 0 {
        eprintln!("Usage: gcd NUMBER ...");
        std::process::exit(1);
    }
```

```
    let mut d = numbers[0];
    for m in &numbers[1..] {
        d = gcd(d, *m);
    }

    println!("Naiveći zajednički delilac {:?} is {}",
             numbers, d);
}
```

Ovo je veliki blok koda, pa hajde da ga pogledamo deo po deo:

```
use std::str::FromStr;
use std::env;
```

Prva `use` deklaracija dovodi standardnu biblioteku *osobina* `FromStr` u opseg. Osobina (eng. trait) je skup metoda koje tipovi mogu implementirati. Svaki tip koji implementira osobinu `FromStr` ima metodu `from_str` koja pokušava da raščlani vrednost tog tipa iz stringa. Tip `u64` implementira `FromStr`, a mi ćemo pozvati `u64::from_str` da raščlanimo naše argumente komandne linije. Mada ne koristimo ime `FromStr` negde drugde u programu, osobina mora biti u opsegu da bi koristila svoje metode. Detaljno ćemo pokriti osobine u Poglavlju 11.

Druga `use` deklaracija donosi `std::env` modul, koji obezbeđuje nekoliko korisnih funkcija i tipova za interakciju sa izvršnim okruženjem, uključujući `args` funkciju, koja nam daje pristup argumentima komandne linije programa.

Prelazimo na `main` funkciju programa:

```
fn main() {
```

Naša `main` funkcija ne vraća vrednost, tako da možemo jednostavno da izostavimo `->` i tip vraćanja koji bi normalno pratio listu parametara.

```
    let mut numbers = Vec::new();
```

Deklarišemo izmenljivu (eng. mutable) lokalnu promenljivu `numbers` i inicijalizujemo je na prazan vektor. `Vec` je Rustov tip vektora koji može rasti, analogan C++-ovom `std::vector`, Python listi ili JavaScript nizu. Iako su vektori dizajnirani da se dinamički povećavaju i skupljaju, i dalje moramo da označimo promenljivu `mut` za Rust da bismo mogli da dodamo brojeve na njegov kraj.

Tip `numbers` je `Vec<u64>`, vektor vrednosti `u64`, ali kao i ranije nije potrebno da to napišemo. Rust će to odraditi za nas, delom zato što ono što dodajmo u vektor vrednosti `u64`, ali i zato što elemente vektora prosleđujemo u `gcd`, koji prihvata samo `u64` vrednosti.

```
    for arg in env::args().skip(1) {
```

Ovde koristimo `for` petlju za obradu naših argumenata komandne linije, postavljajući promenljivu `arg` za svaki argument redom i izvršavajući telo petlje.

`std::env` funkcija `args` modula vraća *iterator*, vrednost koja proizvodi svaki argument na zahtev, i ukazuje kada završimo. Iteratori su sveprisutni u Rustu; standardna biblioteka uključuje druge iteratore koji proizvode elemente vektora, niz datoteke, poruke primljene na

komunikacioni kanal i skoro sve ostalo što ima smisla za petlju. Rustovi iteratori su veoma efikasni: kompajler je obično u stanju da ih prevede u isti kôd kao što je ručno kodirana petlja. Pokazaćemo kako ovo funkcioniše i dati primere u Poglavlju 15.

Pored njihove upotrebe sa `for` petljama, iteratori nude širok izbor metoda koje možete direktno koristiti. Na primer, prva vrednost koju proizvodi iterator koju vraća `args` je uvek ime programa koji se pokreće. Želimo da to preskočimo, pa pozivamo metodu iteratora `skip` da bismo proizveli novi iterator koji izostavlja tu prvu vrednost.

```
numbers.push(u64::from_str(&arg)
             .expect("error parsing argument"));
```

Ovde pozivamo `u64::from_str` da pokušamo da raščlanimo naš argument komandne linije `arg` kao 64-bitni ceo broj bez predznaka. Umesto metode koju pozivamo na neku vrednost `u64` koju imamo pri ruci, `u64::from_str` je funkcija povezana sa tipom `u64`, slična statičkoj metodi u C++ ili Javi. Funkcija `from_str` ne vraća direktno `u64`, već vrednost `Result` koja pokazuje da li je raščlanjivanje uspelo ili nije uspelo. Vrednost `Result` je jedna od dve varijante:

- Vrednost `Ok(v)`, ukazuje da je raščlanjivanje uspelo i `v` je proizvedena vrednost
- Vrednost `Err(e)`, ukazuje da raščlanjivanje nije uspelo i `e` je vrednost greške koja j objašnjava

Funkcije koje rade bilo šta što bi moglo da ne uspe, kao što je unos ili izlaz ili druga interakcija sa operativnim sistemom, mogu da vrate `Result` tipove čije `Ok` varijante nose uspešne rezultate – broj prenetih bajtova, otvorena datoteka i tako dalje – i čije `Err` varijante nose kôd greške koji pokazuje šta je pošlo naopako. Za razliku od većine modernih jezika, Rust nema izuzetaka: sve greške se obrađuju pomoću `Result` ili panike, kao što je navedeno u Poglavlju 7.

Koristimo `Resultovu` metodu `expect` da proverimo uspeh naše raščlanjivanja. Ako je rezultat `Err(e)`, `expect` ispisuje poruku koja sadrži opis `e` i odmah izlazi iz programa. Međutim, ako je rezultat `Ok(v)`, `expect` jednostavno vraća sam `v`, koji smo konačno u mogućnosti da dodamo na kraj našeg vektora brojeva.

```
if numbers.len() == 0 {
    eprintln!("Usage: gcd NUMBER ...");
    std::process::exit(1);
}
```

Ne postoji najveći zajednički delilac praznog skupa brojeva, pa proveravamo da li naš vektor ima bar jedan element i izlazimo iz programa sa greškom ako nema. Koristimo `eprintln!` makro da zapišemo našu poruku o grešci u standardni izlazni tok greške.

```
let mut d = numbers[0];
for m in &numbers[1..] {
    d = gcd(d, *m);
}
```

Ova petlja koristi `d` kao svoju radnu vrednost, ažurirajući je da ostane najveći zajednički delilac svih brojeva koje smo do sada obrađivali. Kao i ranije, moramo označiti `d` kao izmenljivu promenljivu da bismo mogli da mu menjamo vrednost u petlji.

Petlja `for` ima dva iznenađujuća bita. Prvo smo napisali `for m in &numbers[1..];` čemu služi operator `&`? Drugo, napisali smo `gcd(d, *m)`; čemu služi `*` u `*m`? Ova dva detalja su komplementarna jedan drugom.

Do ove tačke, naš kôd je radio samo na jednostavnim vrednostima kao što su celi brojevi koji se uklapaju u blokove memorije fiksne veličine. Ali sada ćemo ići preko vektora, koji mogu biti bilo koje veličine, moguće i veoma veliki. Rust je oprezan kada rukuje takvim vrednostima: želi da ostavi programeru kontrolu nad potrošnjom memorije, stavljajući do znanja koliko dugo je svaka vrednost živi, a istovremeno osigurava da se memorija odmah oslobodi kada više nije potrebna.

Dakle, kada idemo kroz poetlju, želimo da kažemo Rustu da *vlasništvo* vektora treba da ostane sa `numbers`; mi samo *pozajmljujemo* njegove elemente za petlju. `&` operator u `&numbers[1..]` pozajmljuje *referencu* na elemente vektora od drugog pa nadalje. `for` petlja vrši iteraciju preko referenciranih elemenata, dozvoljavajući da `m` pozajmi svaki element uzastopno. `*` operator u `*m` *dereferencira* `m`, dajući vrednost na koju se odnosi; ovo je sledeći `u64` koji želimo da prosledimo u `gcd`. Konačno, pošto `numbers` poseduje vektor, Rust ga automatski oslobađa kada `numbers` izađe van opsega na kraju `main`.

Rustova pravila za vlasništvo i reference su ključna za Rustovo upravljanje memorijom i bezbednu paralelnost; detaljno ih razmatramo u Poglavlju 4 i Poglavlju 5. Moraćete da dobro razumete ta pravilima da biste se osećali udobno u Rustu, ali za ovaj uvodni obilazak, sve što treba da znate je da `&x` pozajmljuje referencu na `x`, i da `*r` je vrednost na koju se odnosi referenca `r`.

Nastavljamo šetnju kroz program:

```
    println!("Najveći zajednički delilac za {:?} je {}",
             numbers, d);
```

Nakon prolaska kroz eliminate od `numbers`, program ispisuje rezultate u standardni izlazni tok. Makro `println!` uzima string šablon, zamenjuje formatirane verzije preostalih argumenata sa `{...}` formama onako kako se pojavljuju u stringu šablona i upisuje rezultat u standardni izlazni tok.

Za razliku od C i C++, koji zahtevaju da `main` vrati nulu ako je program uspešno završio, ili izlazni status različit od nule ako je nešto pošlo naopako, Rust pretpostavlja da ako se `main` uopšte vrati, program je uspešno završen. Samo eksplicitnim pozivanjem funkcija kao što su `expect` ili `std::process::exit` možemo prouzrokovati da se program završi sa kodom statusa greške.

Naredba `cargo run` nam omogućava da prosledimo argumente našem programu, tako da možemo da isprobamo naše rukovanje komandnom linijom:

```
$ cargo run 42 56
   Compiling hello v0.1.0 (/home/jimb/rust/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.22s
     Running `/home/jimb/rust/hello/target/debug/hello 42 56`
Najveći zajednički delilac za [42, 56] is 14
$ cargo run 799459 28823 27347
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `/home/jimb/rust/hello/target/debug/hello 799459 28823 27347`
Najveći zajednički delilac za [799459, 28823, 27347] is 41
$ cargo run 83
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `/home/jimb/rust/hello/target/debug/hello 83`
Najveći zajednički delilac za [83] is 83
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `/home/jimb/rust/hello/target/debug/hello`
Usage: gcd NUMBER ...
```

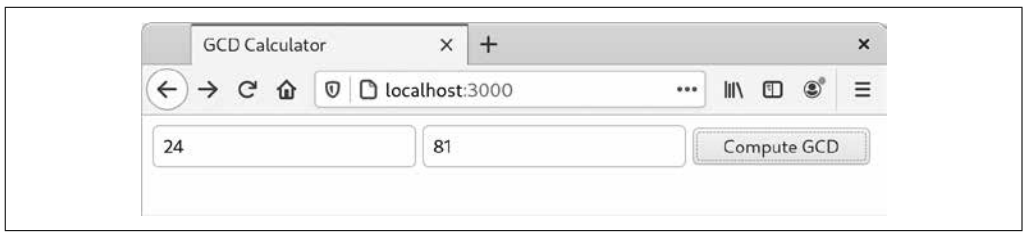
U ovom odeljku smo koristili nekoliko funkcija iz Rustove standardne biblioteke. Ako vas zanima šta je još dostupno, preporučujemo vam da isprobate Rustovu onlajn dokumentaciju. Raspolaže funkciom pretrage uživo koja olakšava istraživanje i čak sadrži veze do izvornog koda. Komanda `rustup` automatski instalira kopiju na vaš računar kada instalirate sam Rust. Dokumentaciju standardne biblioteke možete pogledati na veb lokaciji Rust (*https://oreil.ly/CGsB5*), ili u svom veb čitaču pomoću komande :

```
$ rustup doc --std
```

Prikazivanje stranica na vebu

Jedna od prednosti Rusta je kolekcija besplatno dostupnih bibliotečkih paketa objavljenih na veb lokaciji *crates.io*. Komanda `cargo` olakšava vašem kodu da koristi paket crates.io: ona će preuzeti pravu verziju paketa, izgraditi ga i ažurirati prema zahtevu. Rust paket, bilo da je biblioteka ili izvršni fajl, naziva se *crate* (gajba, kašeta); Cargo i crates.io oba izvode svoja imena iz ovog termina.

Da bismo pokazali kako ovo funkcioniše, sastavićemo jednostavan veb server koristeći `actix-web` veb frejmvork kašetu, `serde` kašetu za serijalizaciju i razne druge kašete od kojih zavise. Kao što je prikazano na Slici 2-1, naša veb lokacija će zatražiti od korisnika dva broja i izračunaće njihov najveći zajednički delilac (eng. greatest common divisor, GCD).



Slika 2-1. Veb stranica koja nudi izračunavanje najvećeg zajedničkog delioca (GCD)

Prvo, Cargo će napraviti novi paket za nas, pod nazivom `actix-gcd`:

```
$ cargo new actix-gcd
     Created binary (application) `actix-gcd` package
$ cd actix-gcd
```

Zatim ćemo urediti datoteku *Cargo.toml* našeg novog projekta da bismo naveli pakete koje želimo da koristimo; njegov sadržaj treba da bude sledeći:

```
[package]
name = "actix-gcd"
version = "0.1.0"
edition = "2021"

# Za više detalja pogledajte na
# https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
actix-web = "1.0.8"
serde = { version = "1.0", features = ["derive"] }
```

Svaki red u `[dependencies]` odeljku *Cargo.toml* daje naziv kašete na crates.io i verziju te kašete koju bismo želeli da koristimo. U ovom slučaju, želimo verziju `1.0.8` kašete `actix-web` i verziju `1.0` kašete `serde`. Možda postoje novije verzije ovih kašeta na crates.io od ovih prikazanih ovde, ali imenovanjem specifičnih verzija na kojima smo testirali ovaj kôd možemo osigurati da će kôd nastaviti da se kompajlira čak i kada budu objavljene nove verzije paketa. O upravljanju verzijama ćemo detaljnije razgovarati u Poglavlju 8.

Kašete mogu imati opcione karakteristike: delove interfejsa ili implementacije koji nisu potrebni svim korisnicima, ali koje ipak ima smisla uključiti u tu kašetu. `serde` kašeta nudi divno sažet način za rukovanje podacima iz veb obrazaca, ali prema dokumentaciji `serde`, dostupan je samo ako odaberemo `derive` kašete, tako da je zahtevamo u našoj datoteci *Cargo.toml* kao što je prikazano.

Imajte na umu da treba da imenujemo samo one kašete koje ćemo direktno koristiti; `cargo` brine o donošenju svih drugih kašeta koje su potrebne.

Za našu prvu iteraciju, učinićemo veb server jednostavnim: on će služiti samo stranici koja od korisnika traži brojeve za računanje. U *actix-gcd/src/main.rs* smestićemo sledeći tekst:

```
use actix_web::{web, App, HttpResponse, HttpServer};

fn main() {
    let server = HttpServer::new(|| {
        App::new()
            .route("/", web::get().to(get_index))
    });

    println!("Serving on http://localhost:3000...");
    server
        .bind("127.0.0.1:3000").expect("error binding server to address")
```

```
        .run().expect("error running server");
}

fn get_index() -> HttpResponse {
    HttpResponse::Ok()
        .content_type("text/html")
        .body(
            r#"
                <title>GCD Calculator</title>
                <form action="/gcd" method="post">
                <input type="text" name="n"/>
                <input type="text" name="m"/>
                <button type="submit">Compute GCD</button>
                </form>
            "#,
        )
}
```

Počinjemo sa `use` deklaracijom da bismo olakšali pristup nekim definicijama `actix-web` kašte. Kada napišemo `use actix-web::{...}`, svako od imena navedenih unutar vitičastih zagrada postaje direktno upotrebljivo u našem kodu; umesto da moramo da napišemo puno ime `actix-web::HttpResponse` svaki put kada ga koristimo, možemo ga jednostavno nazvati `HttpResponse`. (Uskoro ćemo stići do kašete `serde`.)

Naša `main` funkcija je jednostavna: poziva `HttpServer::new` da kreira server koji odgovara na zahteve za jednu putanju, `"/"`; ispisuje poruku koja nas podseća kako da se povežemo sa njom; a zatim ga postavlja da sluša na TCP portu 3000 na lokalnoj mašini.

Argument koji prosleđujemo `HttpServer::new` je Rust *closure* izraz `|| { App::new() ...`. Zatvaranje (eng. closure) je vrednost koja se može pozvati kao da je funkcija. Ovo zatvaranje ne uzima argumente, ali ako uzme, njihova imena bi bila između `||` vertikalnih crta. `{ ... }` je telo zatvaranja. Kada pokrenemo naš server, Actix pokreće skup niti za obradu dolaznih zahteva. Svaka nit poziva naše zatvaranje da bi dobila novu kopiju vrednosti `App` koja joj govori kako da usmeri i rukuje zahtevima.

Zatvaranje poziva `App::new` da kreira novu, praznu `App`, a zatim poziva njen `route` metodu da doda jednu rutu za putanju `"/"`. Rukovalac obezbeđen za tu rutu, `web::get().to(get_index)`, tretira HTTP `GET` zahteve pozivanjem funkcije `get_index`. Metoda `route` vraća istu `App` na kojoj je pozvana, sada poboljšanu novom rutom. Pošto na kraju tela zatvaranja nema tačke i zareza, `App` je povratna vrednost zatvaranja, spremna da upotrebi nit `HttpServer`.

Funkcija `get_index` gradi vrednost `HttpResponse` koja predstavlja odgovor na HTTP `GET /` zahtev. `HttpResponse::Ok()` predstavlja HTTP `200 OK` status, što ukazuje da je zahtev uspeo. Njegove metode `content_type` i `body` pozivamo da bismo popunili detalje odgovora; svaki poziv vraća `HttpResponse` na koji je primenjen, sa izvršenim izmenama. Konačno, povratna vrednost iz `body` služi kao povratna vrednost `get_index`.

Pošto tekst odgovora sadrži mnogo dvostrukih navodnika, pišemo ga koristeći Rust sintaksu „neobrađenog (eng. raw) stringa“: slovo `r`, nula ili više heš oznaka (`#` znak), dvostruki navod-

nik, a zatim sadržaj stringa, koji se završava drugim dvostrukim navodnikom praćen istim brojem heš oznaka. Bilo koji znak se može pojaviti unutar neobrađenog stringa bez znakova izlaz (\), uključujući dvostruke navodnike; u stvari, nijedna izlazna sekvenca kao što je \" se ne prepoznaje. Uvek možemo da obezbedimo da se string završava tamo gde nameravamo tako što ćemo koristiti više heš oznaka oko navodnika nego što se ikad pojavljuje u tekstu.

Napisavši *main.rs*, možemo koristiti komandu `cargo run` da uradimo sve što je potrebno da se izvrši: preuzimanje potrebnih kašeta, kompajliranje, izgradnja (eng. building) sopstvenog programa, povezivanje svega zajedno i pokretanje:

```
$ cargo run
    Updating crates.io index
 Downloading crates ...
  Downloaded serde v1.0.100
  Downloaded actix-web v1.0.8
  Downloaded serde_derive v1.0.100
...
   Compiling serde_json v1.0.40
   Compiling actix-router v0.1.5
   Compiling actix-http v0.2.10
   Compiling awc v0.2.7
   Compiling actix-web v1.0.8
   Compiling gcd v0.1.0 (/home/jimb/rust/actix-gcd)
    Finished dev [unoptimized + debuginfo] target(s) in 1m 24s
     Running `/home/jimb/rust/actix-gcd/target/debug/actix-gcd
Serving on http://localhost:3000...
```

U ovom trenutku možemo da posetimo datu URL adresu u našem čitaču i da vidimo stranicu prikazanu ranije na Slici 2-1.

Nažalost, klik na Compute GCD ne radi ništa, osim odvođenja našeg čitača na praznu stranicu. Hajde da to popravimo, dodavanjem druge rute u naš `App` za obradu `POST` zahteva iz našeg obrasca.

Konačno je vreme da upotrebimo `serde` kašetu koji smo naveli u našoj datoteci *Cargo.toml*: pruža zgodan alat koji će nam pomoći da obradimo podatke obrasca. Prvo ćemo morati da dodamo sledeću `use` direktivu na vrh *src/main.rs*:

```
use serde::Deserialize;
```

Rust programeri obično skupljaju sve svoje `use` deklaracije zajedno na vrhu datoteke, ali to nije striktno neophodno: Rust dozvoljava da se deklaracije pojavljuju bilo kojim redosledom, sve dok se pojavljuju na odgovarajućem nivo gnežđenja.

Dalje, hajde da definišemo tip strukture koji predstavlja vrednosti koje očekujemo od našeg obrasca:

```
#[derive(Deserialize)]
struct GcdParameters {
    n: u64,
    m: u64,
}
```

Ovo definiše novi tip pod nazivom `GcdParameters` koji ima dva polja, `n` i `m`, od kojih je svako `u64`; tip argumenta koji naša `gcd` funkcija očekuje.

Napomena iznad definicije `struct` je atribut, kao što je atribut `#[test]` koji smo ranije koristili za označavanje testnih funkcija. Postavljanje atributa `#[derive(Deserialize)]` iznad definicije tipa govori `serde` kašeti da ispita tip kada se program kompajlira i automatski generiše kôd da raščlani vrednost ovog tipa iz podataka u formatu koji HTML obrasci koriste za `POST` zahteve. U stvari, taj atribut je dovoljan da vam omogući da raščlanite vrednost `GcdParameters` iz gotovo bilo koje vrste strukturiranih podataka: JSON, YAML, TOML ili bilo kog od brojnih drugih tekstualnih i binarnih formata. `serde` kašete takođe obezbeđuje `Serialize` atribut koji generiše kôd koji radi obrnuto, uzimajući Rust vrednosti i ispisujući ih u strukturisanom formatu.

Sa ovom definicijom, možemo prilično lako napisati našu funkciju rukovaoca:

```
fn post_gcd(form: web::Form) -> HttpResponse {
    if form.n == 0 || form.m == 0 {
        return HttpResponse::BadRequest()
            .content_type("text/html")
            .body("Računanje NZD(GCD) sa nulom je dosadno.");
    }

    let response =
        format!("Najveći zajednički delilac za brojeve {} i {} \
                 je <b>{}</b>\n",
                form.n, form.m, gcd(form.n, form.m));

    HttpResponse::Ok()
        .content_type("text/html")
        .body(response)
}
```

Da bi funkcija služila kao rukovalac Actix zahteva, svi njeni argumenti moraju imati tipove koje Actix zna da izdvoji iz HTTP zahteva. Naša `post_gcd` funkcija uzima jedan argument, `form`, čiji je tip `veb::Form<GcdParameters>`. Actix zna kako da izdvoji vrednost bilo kog tipa `veb::Form<T>` iz HTTP zahteva ako, i samo ako, `T` može da se deserializuje iz podataka HTML oblika `POST`. Pošto smo atribut `#[derive(Deserialize)]` postavili u našu definiciju tipa `GcdParameters`, Actix može da ga deserializuje iz podataka obrasca, tako da rukovaoci zahteva mogu očekivati `web ::Form<GcdParameters>` vrednost kao parametar. Ovi odnosi između tipova i funkcija se razrešavaju u vreme kompajliranja; ako napišete funkciju rukovaoca sa tipom argumenta koji Actix ne zna kako da rukuje, Rust kompajler vas odmah obaveštava o grešci.

Gledajući unutar `post_gcd`, funkcija prvo vraća HTTP `400 BAD REQUEST` grešku ako je bilo koji parametar nula, pošto će naša `gcd` funkcija paničiti ako jesu. Zatim konstruiše odgovor na zahtev koristeći makro `format!`. `format!` makro je isti kao makro `println!`, samo što umesto da upiše tekst u standardni izlaz, on ga vraća kao string. Kada dobije tekst odgovora, `post_gcd` ga umotava u HTTP `200 OK` odgovor, postavlja njegov tip sadržaja i vraća ga da bude isporučen pošiljaocu.

Takođe moramo da registrujemo `post_gcd` kao rukovalac obrasca. Zamenićemo našu `main` funkciju sa ovom verzijom:

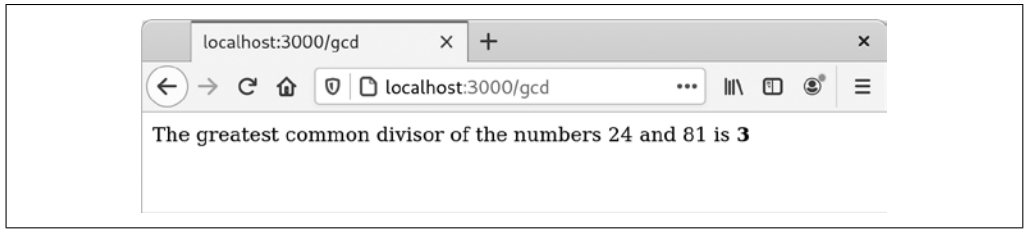
```
fn main() {
    let server = HttpServer::new(|| {
        App::new()
            .route("/", web::get().to(get_index))
            .route("/gcd", web::post().to(post_gcd))
    });

    println!("Serving on http://localhost:3000...");
    server
        .bind("127.0.0.1:3000").expect("error binding server to address")
        .run().expect("error running server");
}
```

Jedina promena je ta što smo dodali još jedan poziv `route`, uspostavljajući `web::post().to(post_gcd)` kao rukovalac za putanju `"/gcd"`.

Poslednji preostali deo je `gcd` funkcija koju smo ranije napisali, koja se nalazi u datoteci *actix-gcd/src/main.rs*. Sa tim možete prekinuti sve servere koje ste možda ostavili da rade i ponovo izgraditi i ponovo pokrenuti program:

```
$ cargo run
   Compiling actix-gcd v0.1.0 (/home/jimb/rust/actix-gcd)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/actix-gcd`
Serving on http://localhost:3000...
```

Ovog puta, ako posetite *http://localhost:3000*, unesete neke brojeve i pritisnite na dugme Compute GCD, trebalo bi da vidite neke rezultate (Slika 2-2).



Slika 2-2. Veb stranica koja prikazuje rezultate izračunavanja GCD-a

Istovremenost

Jedna od Rustovih velikih prednosti je njegova podrška za istovremeno (eng. concurrent) programiranje. Ista pravila koja obezbeđuju da Rust programi nemaju memorijske greške takođe obezbeđuju da niti mogu da dele memoriju samo na načine koji izbegavaju trke podataka (eng. data races.). Na primer: