# Sadržaj

**Predgovor** ..... **xv**

**1. Programeri sistema mogu imati lepe stvari** ..... **1**
Rust peuzima vaš teret na sebe 2
Paralelno programiranje je ukroćeno 3
I pored svega Rust je i dalje brz 3
Rust čini saradnju jednostavnijom 4

**2. Kratka tura kroz Rust** ..... **5**
rustup i Cargo 5
Rust funkcije 8
Pisanje i izvođenje jediničnih testova 9
Rukovanje argumentima komandne linije 10
Prikazivanje stranica na vebu 14
Istovremenost 19
Šta je zapravo Mandelbrotov skup 20
Raščlanjivanje para argumenata komandne linije 24
Mapiranje od piksela do kompleksnih brojeva 27
Iscrtavanje skupa 28
Upisivanje datoteka slika 29
Istovremen Mandelbrotov program 30
Izvršavanje Mandelbrot plotera 36
Bezbednost je nevidljiva 37
Fajl sistem i alatke komandne linije 37
Interfejs komandne linije 38
Čitanje i pisanje datoteka 40
Find i Replace 41

**3. Osnovni tipovi** ..... **45**
Numerički tipovi fiksne širine 48
Celobroji tipovi 48
Provera, omotavanje, zasićenje i prekoračenje aritmetike 51
Tipovi sa pokretnim zarezom 53
Tip bool 55

Znakovi 56
Torke 57
Tipovi pokazivača 59
Reference 60
Box 60
Raw pokazivači 61
Nizovi, vektori i isečci 61
Nizovi 61
Vektori 62
Isečci 65
String tipovi 67
String literali 67
Bajt string 68
Stringovi u memoriji 68
String 70
Korišćenje stringova 71
Drugi tipovi nalik stringovima 71
Alijas tipovi 72
Posle osnova 72

**4. Vlasništvo i premeštanja ..... 73**

Vlasništvo 74
Premeštanja 79
Još operacija koje se premeštaju 83
Premeštanja i kontrolni tok 84
Premeštanja i indeksiran sadržaj 85
Copy tipovi: izuzetak za premeštanje 87
Rc i Arc: deljeno vlasništvo 90

**5. Reference ..... 93**

Reference na vrednosti 93
Rad sa referencama 97
Rust reference u odnosu na C++ reference 97
Dodeljivanje referenci 98
Reference na reference 98
Poređenje referenci 99
Reference nikada nisu Null 100
Pozajmljivanje referenci na proizvoljne izraze 100
Reference na isečke i objekte osobina 101
Bezbednost referenci 101
Pozajmljivanje lokalne promenljive 101
Primanje referenci kao argumenta funkcije 104
Prosleđivanje referenci funkcijama 106
Vraćanje referenci 106

Strukture koje sadrže reference 108
Različiti parametri životnog veka 110
Izostavljanje parametara životnog veka 111
Deljenje nasuprot izmena 113
Uzimanje oružja protiv mora objekata 120

**6. Izrazi ..... 123**
Ekspresivan jezik 123
Prednost i asocijativnost 124
Blokovi i tačka-zarezi 126
Deklaracije 128
if i match 129
if let 131
Petlje 132
Kontrola toka u petljama 133
return izrazi 135
Zašto Rust ima petlju 136
Pozivi funkcija i metoda 137
Polja i elementi 138
Operatori za referenciranje 139
Operatori za aritmetiku, nad bitovima, poređenja i logički 140
Dodela 141
Konverzija tipova 141
Zatvaranja 142
Nastavak 143

**7. Obrada grešaka ..... 145**
Panika 145
Odmotavanje 146
Prekid 147
Result 147
Hvatanje grešaka 148
Result alijas tipa 149
Greške pri ispisivanju 150
Greške propagacije 151
Rad sa višestrukim tipovima grešaka 152
Bavljenje greškama koje se „ne mogu desiti“ 154
Ignorisanje grešaka 155
Rukovanje greškama u main() 155
Deklarisanje prilagođenog tipa greške 157
Zašto Results? 158

**8. Kašete i moduli ..... 159**
Kašete 159
Izdanja 162
Izgradite profile 163
Moduli 164
Ugnežđeni moduli 165
Moduli u zasebnim datotekama 166
Putanje i uvozi 168
Standardni uvod 171
Pravljenje use dekleracija javnim 171
Pavljenje polja strukture javnim 172
Statika i konstante 172
Pretvaranje programa u biblioteku 173
Direktorijum src/bin 174
Atributi 176
Testovi i dokumentacija 178
Testovi integracije 181
Dokumentacija 181
Doc testovi 184
Zadavanje zavisnosti 187
Verzije 187
Cargo.lock 188
Objavljivanje kašeta na crates.io 189
Radni prostori 191
Još lepih stvari 192

**9. Strukture ..... 193**
Strukture imenovanih polja 193
Strukture slične torki 196
Strukture slične jedinicama 197
Izgled strukture 197
Definisanje metoda sa impl 198
Prosleđivanje self kao Box, Rc ili Arc 200
Funkcije povezane sa tipom 201
Povezane konstante 202
Generičke strukture 203
Generičke strukture sa parametrima životnog veka 205
Generičke strukture sa konstantnim parametrima 206
Izvođenje zajedničkih osobina za tipove struktura 208
Izmenljivost unutrašnjosti 209

**10. Enumi i obrasci....213**
Enumi 214
Enumi sa podacima 216
Enumi u memoriji 217
Bogate strukture podataka pomoću enuma 218
Generički enumi 220
Obrasci 222
Literali, promenljive i zamenski znakovi u obrascima 225
Obrsci torke i strukture 226
Obrasci niza i isečka 227
Obrasci referenci 228
Čuvari podudaranja 230
Podudaranje višestrukih mogućnosti 231
Vezivanje sa @ obrascima 231
Gde su obrasci dozvoljeni 232
Popunjavanje binarnog stabla 233
Velika slika 235

**11. Osobine i generici....237**
Korišćenje osobina 239
Objekti osobina 240
Generičke funkcije i parametri tipa 242
Šta koristiti 245
Definisanje i implementacija osobina 247
Podrazumevane metode 248
Osobine i tipovi drugih ljudi 250
Self u osobinama 251
Podosobine 253
Funkcije povezane sa tipom 253
Pozivi potpuno kvalifikovanih metoda 255
Osobine koje definišu odnose između tipova 256
Pridruženi tipovi (ili kako rade iteratori) 256
Generičke osobine (ili kako funkcioniše preopterećenje operatora) 260
impl osobina 261
Pridružene konstante 263
Granice obrnutog inženjeringa 264
Osobine kao osnova 267

**12. Preopterećenje operatora....269**
Aritmetički operatori i opratori nad bitovima 270
Unarni operatori 272
Binarni operatori 273
Složeni operatori dodele 274

Poređenje jednakosti 275
Poređenje redoslednosti 278
Index i IndexMut 281
Drugi operatori 283

**13. Korisne osobine ... 285**
Drop 286
Sized 289
Clone 291
Copy 292
Deref i DerefMut 293
Default 296
AsRef i AsMut 298
Borrow i BorrowMut 299
From i Into 301
TryFrom i TryInto 304
ToOwned 305
Borrow i ToOwned na delu: skromna krava 305

**14. Zatvaranja ... 309**
Hvatanje promenljivih 310
Zatvaranja koja pozajmljuju 311
Zatvaranja koja kradu 312
Vrste funkcija i zatvaranja 313
Performanse zatvaranja 315
Zatvaranja i bezbednost 317
Zatvaranja koja ubijaju 317
FnOnce 317
FnMut 319
Copy i Clone za zatvaranja 321
Povratni pozivi 322
Efektivno korišćenje zatvaranja 326

**15. Iteratori ... 329**
Osobine Iterator i IntoIterator 330
Kreiranje iteratora 332
iter i iter_mut metode 332
IntoIterator implementacije 333
from_fn i successors 334
drain metode 336
Drugi izvori iteratora 336
Adapteri iteratora 338
map i filter 338
filter_map i flat_map 340

flatten 342
take i take_while 344
skip i skip_while 345
peekable 345
fuse 347
Reverzibilni iteratori i rev 347
inspect 348
chain 349
enumerate 350
zip 350
by_ref 351
cloned, copied 352
cycle 353
Potošnja iteratora 353
Jednostavna akumulacija: count, sum, product 354
max, min 354
max_by, min_by 355
max_by_key, min_by_key 355
Upoređivanje nizova stavki 356
any i all 356
position, rposition i ExactSizeIterator 357
fold i rfold 357
try_fold i try_rfold 358
nth, nth_back 360
last 360
find, rfind i find_map 361
Izgradnja kolekcija: collect i FromIterator 362
extend osobina 363
partition 364
for_each i try_for_each 365
Implementacija sopstvenih iteratora 366

**16. Kolekcije ..... 371**

Pregled 372
Vec<T> 373
Pristup elementima 373
Iteracija 375
Vektori rasta i smanjenja 376
Pridruživanje 379
Razdvajanje 379
Zamena sadržaja 382
Punjenje 382
Sortiranje i pretraživanje 382
Upoređivanje isečaka 384

Nasumični elementi 384
Rust isključuje greške pri invalidaciji 385
VecDeque<T> 386
BinaryHeap<T> 388
HashMap<K, V> i BTreeMap<K, V> 389
Unosi 392
Iteracija mape 395
HashSet<T> i BTreeSet<T> 395
Postavljanje iteracije 396
Kada su jednake vrednosti različite 396
Operacije nad celim skupom 397
Heširanje 398
Korišćenje prilagođenog algoritma heširanja 399
Posle standardnih kolekcija 401

**17. Stringovi i tekst ... 403**

Unikod pozdina 403
ASCII, Latin-1 i Unicode 403
UTF-8 404
Smer teksta 406
Znakovi (char) 406
Klasifikacija znakova 406
Rad sa ciframa 408
Konverzija velikih i malih slova 408
Konverzije u i iz celih brojeva 409
String i str 409
Kreiranje vrednosti niza 410
Jednostavna inspekcija 411
Dodavanje i umetanje teksta 412
Uklanjanje i zamena teksta 413
Konvencije za pretraživanje i itraciju 414
Obrasci za pretraživanje teksta 414
Traženje i zamena 415
Iteracija nad tekstom 416
Trim 418
Konverzija velikih i malih slova stringova 419
Izvlačenje drugih tipova iz stringova 419
Pretvaranje drugih tipova u nizove 420
Pozajmljivanje kao drugi tipovi nalik tekstu 421
Pristup tekstu kao UTF-8 421
Proizvodnja teksta iz UTF-8 podataka 421
Odlaganje alokacije 422
Stringovi kao generičke kolekcije 425
Formatiranje vrednosti 425

Formatiranje tekstualnih vrednosti 426
Formatiranje brojeva 428
Formatiranje drugih tipova 430
Formatiranje vrednosti za otklanjanje grešaka 430
Formatiranje pokazivača za otklanjanje grešaka 431
Referisanje na argumente po indeksu ili imenu 432
Dinamičke širine i preciznosti 432
Formatiranje vaših sopstvenih tipova 433
Korišćenje jezika za oblikovanje u sopstvenom kodu 435
Regularni izrazi 436
Osnovna upotreba regularnog izraza 437
Odložena izgradnja vrednosti regularnog izraza 438
Normalizacija 439
Obrasci za normalizaciju 440
unicode-normalization 441

**18. Ulaz i izlaz ..... 443**

Čitači i pisači 444
Čitači 445
Baferovani čitači 446
Čitanje redova 448
Prikupljanje redova 450
Pisači 451
Datoteke 452
Pretraživanje 453
Drugi tipovi čitača i pisača 453
Binarni podaci, kompresija i serijalizacija 455
Datoteke i direktorijumi 456
OsStr i Path 456
Path i PathBuf metode 458
Funkcije pristupa sistemu datoteka 460
Čitačnje direktorijuma 461
Karakteristike specifične za platformu 462
Umrežavanje 464

**19. Konkurentnost ..... 467**

Paralelizam grananje-spajanje 468
spawn i join 470
Upravljanje greškom u nizu 472
Deljenje neizmenljivih podataka preko niti 473
Rayon 475
Ponovni pogled Mandelbrotovog skupa 477

Kanali 479
Slanje vrednosti 480
Primanje vrednosti 483
Pokretanje cevovoda 484
Funkcije i performanse kanala 486
Bezbednost niti: Send i Sync 487
Dovođenje skoro svakog iteratora na kanal 490
Posle cevovoda 491
Deljeno izmenljivo stanje 491
Šta je muteks? 492
Mutex<T> 493
mut i Mutex 495
Zašto muteksi nisu uvek dobra ideja 496
Zastoj 497
Otrovani muteksi 497
Multikorisnički kanali koji koriste mutekse 498
Zaključavanja čitanja/pisanja (RwLock<T>) 499
Uslovne promenljive (Condvar) 500
Atomika 501
Globalne promenljive 502
Kako izgleda hakovanje konkurentnog koda u Rustu 505

**20. Asinhrono programiranje ..... 507**
Od sinhronog ka asinhronom 509
Future 510
Asinhrone funkcije i izrazi čekanja 512
Pozivanje asinhronih funkcija iz sinhronog koda: block_on 514
Pokretanje asinhronnih zadataka 517
Asinhroni blokovi 520
Izgradnja asinhronih funkcija iz asinhronih blokova 522
Pokretanje asinhronih zadataka u grupi niti 524
Da li vaša buduća implementacija šalje? 524
Dugotrajna izračunavanja: yield_now i spawn_blocking 527
Upoređivanje asinhronih dizajna 529
Pravi asinhron HTTP klijent 529
Asinhroni klijent i server 530
Tipovi grešaka i rezultata 532
Protokol 532
Primanje korisničkog unosa: asinhroni tokovi 534
Slanje paketa 536
Primanje paketa: više asinhronih tokova 537
Klijentova funkcija main 538
Serverska funkcija main 539
Upravljanje vezama za ćaskanje: asinhroni muteksi 541

Grupna tabela: sinhroni muteksi 543
Grupe za ćaskanje: tokio kanali za emitovanje 544
Primitive budućnosti i izvršioci: kada je budućnost vredna ispitivanja? 547
Pozivanje Wakers: spawn_blocking 549
Implementacija block_on 551
Označavanje 552
Dve životne faze budućnosti 553
Pin pokazivači 556
Unpin osobina 558
Kada je asinhron kôd koristan? 559

**21. Makroi ..... 561**

Osnove makroa 562
Osnove proširenja makroa 563
Nenamerne posledice 565
Ponavljanje 566
Ugrađeni makroi 568
Otklanjanje grešaka u makroima 570
Izgradnja json! makroa 571
Tipovi fragmenata 572
Rekurzija u makroima 575
Korišćenje osobina sa makroima 576
Obim i higijena 578
Uvoz i izvoz makroa 580
Izbegavanje sintaksičkih grešaka tokom podudaranja 582
Više od macro_rules! 583

**22. Nebezbedan kod ..... 585**

Nebezbedno od čega? 586
Nebezbedni blokovi 587
Primer: Efikasan ASCII tip stringa 588
Nebezbedne funkcije 590
Nebezbedan blok ili nebezbedna funkcija? 592
Nedefinisano ponašanje 593
Nebezbedne osobine 595
Sirovi pokazivači 597
Bezbedno dereferenciranje sirovih pokazivača 600
Primer: RefWithFlag 601
Null pokazivači 603
Veličine tipova i poravnanje 603
Aritmetika pokazivača 604
Premeštanje u memoriju i van nje 605
Primer: GapBuffer 609
Bezbednost od panike u nebezbednom kodu 615

Reinterpretacija memorije sa unijama 616
Podudarni sindikati 618
Pozajmljivanje unije 619

**23. Spoljne funkcije ..... 621**
Pronalaženje zajedničkih prikaza podataka 621
Deklarisanje spoljnih funkcija i promenljivih 625
Korišćenje funkcija iz biblioteka 627
Sirov interfejs za libgit2 630
Bezbedan interfejs za libgit2 636
Zaključak 646

**Rečnik termina korišćenih u knjizi ..... 649**

**Indeks ..... 651**

POGLAVLJE 1

# Programeri sistema mogu imati lepe stvari

*U određenim kontekstima – na primer, kontekst na koji Rust cilja – je 10x ili čak 2x brži od konkurencije. Brzina odlučuje o sudbini na tržištu, kao što odlučuje na tržištu hardvera.*
Greydon Hoare

*Svi računari su sada paralelni...*
*Paralelno programiranje **jeste** programiranje.*
Michael McCool i saradnici, *Structured Parallel Programming*

*Greška u paraseru TrueTypa koriščena od američke države koristi za sveopšti nadzor; sav softver je bezbednosno osetljiv.*
Andy Wingo

Odlučili smo da počnemo ovu knjigu sa tri navedena citata. Ali počnimo sa misterijom. Šta radi sledeći C program?

```
int main(int argc, char **argv) {
  unsigned long a[1];
  a[3] = 0x7ffff7b36cebUL;
  return 0;
}
```

Na Džimovom laptopu jutros je ovaj program ispisao:

```
undef: Error: .netrc file is readable by others.
undef: Remove password or make file unreadable by others.
```

Onda se srušio. Ako ga isprobate na svojoj mašini, možda će uraditi nešto drugo. Šta se ovde dešava?

Program je pogrešan. Niz `a` je dugačak samo jedan element, tako da je korišćenje `a[3]`, prema standardu programskog jezika C, *nedefinisano ponašanje*:

> Ponašanje pri korišćenju neprenosive ili pogrešne programske konstrukcije ili pogrešnih podataka, za koje međunarodni standard ne nameće nikakve zahteve

Nedefinisano ponašanje nema samo nepredvidiv rezultat: standard eksplicitno dozvoljava programu da uradi *bilo šta*. U našem slučaju, čuvanje ove određene vrednosti u četvrtom elementu ovog određenog niza narušava stek poziva funkcije pri vraćanju iz `main` funkcije, umesto elegantnog izlaska iz programa kako bi trebalo, skače usred koda iz standardne C

biblioteke za preuzimanje lozinke iz datoteke u korisničkom kućnom direktorijumu. To nije dobro.

C i C++ imaju stotine pravila za izbegavanje nedefinisanog ponašanja. Oni su uglavnom zdravorazumski: ne pristupajte memoriji kojoj ne bi trebalo, ne koristite nedozvoljene aritmetičke operacija, ne delite sa nulom itd. Ali kompajler ne sprovodi ova pravila; nema obavezu da otkriva čak ni eklatantne prekršaje. Zaista, prethodni program se kompajlira bez grešaka ili upozorenja. Odgovornost za izbegavanje nedefinisanog ponašanja u potpunosti pada na vas, programera.

Empirijski gledano, mi programeri nemamo sjajne rezultate u ovom pogledu. Dok je bio student na Univerzitetu Juta, istraživač Peng Li je modifikovao C i C++ kompajlere da bi programi koje je prevodio izvestili da li su izvršili određene oblike nedefinisanog ponašanja. Otkrio je da skoro svi programi to čine, uključujući i one iz dobro cenjenih projekata koji svoj kôd drže po visokim standardima. Pretpostavka da možete da izbegnete nedefinisano ponašanje u C i C++ je kao pretpostavka da možete pobediti u partiji šaha samo zato što znate pravila.

Povremena čudna poruka ili rušenje mogu biti problem kvaliteta, ali nenamerno nedefinisano ponašanje je glavni uzrok bezbednosnih propusta kada je Morrisov crv 1988. god. iskoristio varijaciju tehnike prikazane ranije, za širenje sa jednog računara na drugi na ranom Internetu.

Dakle, C i C++ dovode programere u nezgodnu poziciju: ti jezici su industrijski standardi za programiranje sistema, ali zahtevi koje postavljaju programerima samo garantuju stalna rušenja i bezbednosne probleme. Odgovor na misteriju postavlja veće pitanje: zar ne možemo bolje?

## Rust peuzima vaš teret na sebe

Naš odgovor je uokviren sa naša tri uvodna citata. Treći citat se odnosi na izveštaje da je Stuxnet, kompjuterski crv otkriven u provali u industrijsku kontrolnu opremu 2010. godine, preuzeo je kontrolu nad računarima žrtava koristeći, između mnogih drugu tehniku, nedefinisano ponašanje u kodu koji je parsirao TrueType fontove ugrađene u dokumente za obradu teksta. Možemo biti sigurni da autori tog koda nisu očekivali da će se koristiti na ovaj način, što ilustruje da nisu samo operativni sistemi i serveri ti koji treba da brinu o bezbednosti: svaki softver koji može da obrađuje podatke iz nepouzdanog izvora može biti meta zloupotrebe.

Rust jezik vam daje jednostavno obećanje: ako vaš program prođe provere kompajlera, nema nedefinisanog ponašanja. Viseći pokazivači, double free greške i pozivanja null pokazivača se hvataju u vreme kompajliranja. Reference niza su obezbeđene mešavinom provera vremena kompajliranja i vremena izvođenja, tako da nema prekoračenja bafera: Rust ekvivalent našeg nesrećnog C programa bezbedno javlja poruku o grešci.

Dalje, Rust ima za cilj da bude *bezbedan* i *prijatan za korišćenje*. Da bi dao jače garancije o ponašanju vašeg programa, Rust nameće više ograničenja vašem kodu nego C i C++, a ova ograničenja zahtevaju praksu i iskustvo da biste se navikli. Ali jezik u celini je fleksibilan i izražajan. O tome svedoči obim koda napisanog u Rustu i opseg oblasti primene na koje se primenjuje.

Po našem iskustvu, mogućnost da raspolažemo jezikom kome verujemo da hvata više grešaka podstiče nas da pokušamo sa ambicioznijim projektima. Izmene velikih, složenih programa su manje rizične kada znate da su rešeni problemi upravljanja memorijom i validnosti pokazivača. A otklanjanje grešaka je mnogo jednostavnije kada potencijalne posledice greške ne uključuju oštećenje nepovezanih delova programa.

Naravno, još uvek postoji mnogo grešaka koje Rust ne može da otkrije. Ali u praksi, uklanjanje nedefinisanog ponašanja bitno menja karakter razvoja na bolje.

## Paralelno programiranje je ukroćeno

Istovremenost (eng. concurrency) je notorno teško koristiti pravilno u C i C++. Programeri se obično okreću istovremenosti samo kada se pokaže da jednonitni kôd nije u stanju da postigne performanse koje su im potrebne. Ali drugi uvodni citat tvrdi da je paralelizam previše važan za moderne mašine da bismo ga tretirali kao krajnju instancu.

Kao što se ispostavilo, ista ograničenja koja obezbeđuju bezbednost memorije u Rustu takođe obezbeđuju da Rust programi nemaju trku podataka. Možete slobodno da delite podatke između niti, sve dok se ne menjaju. Podacima koji se menjaju može se pristupiti samo pomoću primitiva za sinhronizaciju. Dostupni su svi tradicionalni alati za konkurentnost: muteksi, promenljive uslova, kanali, atomika itd. Rust jednostavno proverava da li ih pravilno koristite.

Ovo čini Rust odličnim jezikom za iskorišćenje sposobnosti savremenih mašina sa više jezgara. Rust ekosistem nudi biblioteke koje prevazilaze uobičajene paralelne primitive i pomažu vam da ravnomerno rasporedite složena opterećenja na skupove procesora, koristite mehanizme sinhronizacije bez zaključavanja kao što je Read-Copy-Update i još mnogo toga.

## I pored svega Rust je i dalje brz

Ovo je, konačno, naš prvi uvodni citat. Rust deli ambicije koje Bjarne Stroustrup artikuliše za C++ u svom radu „Abstraction and the C++ Machine Model“:

> Generalno, implementacije C++-a poštuju princip nultih troškova: ono što ne koristite, ne plaćate. I dalje: Ono što koristite, ne može biti bolji kôd.

Programiranje sistema se često bavi guranjem mašine do njenih granica. Za video igre, cela mašina treba da bude posvećena stvaranju najboljeg iskustva za igrača. Za veb čitače, efikasnost čitača postavlja gornju granicu onoga što autori sadržaja mogu da urade. U okviru inherentnih ograničenja mašine, što je više moguće memorije i pažnje procesora mora biti

prepušteno samom sadržaju. Isti princip važi i za operativne sisteme: kernel treba da učini resurse mašine dostupnim korisničkim programima, a ne da ih sam troši.

Ali kada kažemo da je Rust „brz“, šta to zaista znači? Može se napisati spor kôd na bilo kom jeziku opšte namene. Preciznije bi bilo reći da, ako ste spremni da investirate u dizajniranje vašeg programa kako biste na najbolji način iskoristili mogućnosti mašine ispod, Rust vas podržava u tom nastojanju. Jezik je dizajniran sa efikasnim podrazumevanim podešavanjima i daje vam mogućnost da kontrolišete kako se memorija koristi i kako se troši pažnja procesora.

## Rust čini saradnju jednostavnijom

Sakrili smo četvrti citat u naslov ovog poglavlja: „Programeri sistema mogu da imaju lepe stvari“. Ovo se odnosi na Rustovu podršku za deljenje koda i ponovnu upotrebu.

Rustov menadžer paketa i alatka za pravljenje, Cargo, olakšava korišćenje biblioteka koje su objavili drugi na Rustovom javnom spremištu paketa, veb sajtu crates.io. Jednostavno dodate naziv biblioteke i potreban broj verzije u datoteku, a Cargo se brine o preuzimanju biblioteke, zajedno sa svim drugim bibliotekama koje koristi, i povezivanju čitave serije. Možete zamisliti Cargo kao Rustov odgovor na NPM ili RubyGems, sa naglaskom na zdravo upravljanje verzijama i ponovljive verzije. Postoje popularne Rust biblioteke koje pružaju sve, od standardne serijalizacije do HTTP klijenata i servera i modernih grafičkih API-ja.

Idući dalje, sam jezik je takođe dizajniran da podrži saradnju: Rustove osobine i generika omogućavaju vam da kreirate biblioteke sa fleksibilnim interfejsima tako da mogu da služe u mnogo različitih konteksta. A Rustova standardna biblioteka pruža osnovni skup osnovnih tipova koji uspostavljaju zajedničke konvencije za uobičajene slučajeve, čineći različite biblioteke lakšim za zajedničko korišćenje.

Sledeće poglavlje ima za cilj da konkretizuje široke tvrdnje koje smo izneli u ovom poglavlju, uz obilazak nekoliko malih Rust programa koji pokazuju snagu jezika.

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 ...
```

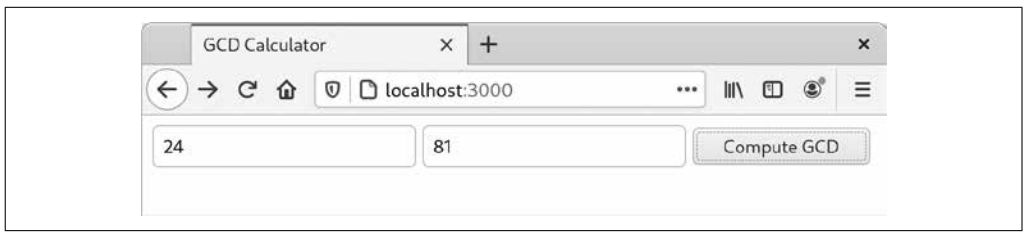
U ovom odeljku smo koristili nekoliko funkcija iz Rustove standardne biblioteke. Ako vas zanima šta je još dostupno, preporučujemo vam da isprobate Rustovu onlajn dokumentaciju. Raspolaže funkciom pretrage uživo koja olakšava istraživanje i čak sadrži veze do izvornog koda. Komanda `rustup` automatski 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))
    });

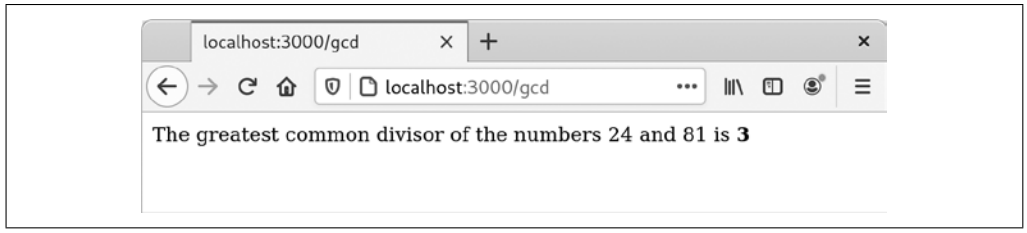
    println!("Serving on http://localhost:3000...");
    server
        .bind("127.0.0.1:3000").expect("error binding server to address")
        .run().expect("error running server");
}
```

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:

POGLAVLJE 3

# Osnovni tipovi

*Na svetu postoji mnogo, mnogo vrsta knjiga, što ima smisla, jer postoji mnogo, mnogo vrsta ljudi i svako od njih želi da čita nešto drugačiju knjigu.*
Lemoni Snicket

U velikoj meri, Rust jezik je dizajniran oko svojih tipova. Njegova podrška za kôd visokih performansi proizilazi iz toga što dozvoljava programerima da izaberu prikaz podataka koji najbolje odgovara situaciji, uz pravi balans između jednostavnosti i cene. Bezbednost Rustove memorije i niti takođe garantuju da počivaju na ispravnosti njegovog sistema tipova, a Rustova fleksibilnost proizilazi iz njegovih generičkih tipova i osobina (eng. traits).

Ovo poglavlje pokriva Rustove osnovne tipove za predstavljanje vrednosti. Ovi tipovi na izvonorm nivou imaju konkretne parnjake na nivou mašine sa predvidljivim troškovima i performansama. Iako Rust ne obećava da će predstavljati stvari upravo onako kako ste tražili, vodi računa i odstupiće od vaših zahteva samo kada je to pouzdano poboljšanje.

U poređenju jezikom dinamičkog tipa kao što je JavaScript ili Python, Rust zahteva od vas da više planirate unapred. Morate navesti tipove argumenata funkcije i povratne vrednosti, polja strukture i nekoliko drugih konstrukcija. Međutim, dve karakteristike Rusta čine da su to manji problemi nego što biste očekivali:

- S obzirom na tipove koje napišete, Rustovo *zaključivanje tipa* će shvatiti većinu ostalog umesto vas. U praksi, često postoji samo jedan tip koji će funkcionisati za datu promenljivu ili izraz; kada je to slučaj, Rust vam dozvoljava da izostavite (*elide*) tip. Na primer, možete da navedete svaki tip u funkciji, ovako:

  ```
  fn build_vector() -> Vec<i16> {
      let mut v: Vec<i16> = Vec<i16>::::new();
      v.push(10i16);
      v.push(20i16);
      v
  }
  ```

  Ali ovo je pretrpano i ponavlja se. S obzirom na povratni tip funkcije, očigledno je da `v` mora biti `Vec<i16>`, vektor 16-bitnih celih brojeva sa predznakom; nijedan drugi tip ne bi funkcionisao. A iz toga sledi da svaki element vektora mora biti `i16`. Ovo je upravo vrsta rasuđivanja koje Rustovo zaključivanje tipa primenjuje, omogućavajući vam da umesto toga napišete:

```
fn build_vector() -> Vec<i16> {
    let mut v = Vec::new();
    v.push(10);
    v.push(20);
    v
}
```

Ove dve definicije su potpuno ekvivalentne, a Rust će generisati isti mašinski kod u svakom slučaju. Zaključivanje tipa vraća dobar deo čitljivosti koje imaju jezici dinamičkog tipa, dok se i dalje hvataju greške u tipu tokom kompajliranja.

- Funkcije mogu biti *generičke*: jedna funkcija može da radi na vrednostima mnogo različitih tipova.

  U Pythonu i JavaScriptu, sve funkcije rade prirodno na ovaj način: funkcija može da radi na bilo kojoj vrednosti koja ima svojstva i metode koje će funkciji trebati. (Ovo je karakteristika koja se često naziva *duck typing*: ako kvače kao patka, to je patka.) Ali upravo ta fleksibilnost otežava tim jezicima da rano otkriju greške u tipu; testiranje je često jedini način da se uhvate takve greške. Rustove generičke funkcije daju jeziku stepen iste fleksibilnosti, dok i dalje hvata sve greške u tipu tokom kompajliranja.

  Uprkos njihovoj fleksibilnosti, generičke funkcije su jednako efikasne kao i njihove negeneričke kolege. Ne postoji inherentna prednost u performansama pisanja, recimo, određene `sum` funkcije za svaki ceo broj u odnosu na pisanje generičke funkcije koja rukuje svim celim brojevima. O generičkim funkcijama ćemo detaljno raspravljati u Poglavlju 11.

Ostatak ovog poglavlja pokriva Rustove tipove odozdo prema gore, počevši od jednostavnih numeričkih tipova kao što su celi brojevi i vrednosti sa pokretnim zarezom, a zatim prelazimo na tipove koji sadrže više podataka: kutije, torke, nizove i stringove.

Evo rezimea vrsta tipova koje ćete videti u Rustu. Tabela 3-1 prikazuje Rustove primitivne tipove, neke veoma uobičajene tipove iz standardne biblioteke i neke primere korisnički definisanih tipova.

*Tabela 3-1. Primeri tipova u Rustu*

| Tip | Opis | Vrednosti |
|---|---|---|
| `i8`, `i16`, `i32`, `i64`, `i128`<br>`u8`, `u16`, `u32`, `u64`, `u128` | Celi brojevi sa i bez predznaka, date širine bitova | `42`, `-5i8`, `0x400u16`, `0o100i16`, `20_922_789_888_000u64`, `b'*'` (`u8` literalni bajt) |
| `isize`, `usize` | Celi brojevi sa i bez pedznaka, iste veličine kao adresa na mašini (32 ili 64 bita) | `137`, `-0b0101_0010isize`, `0xffff_fc00usize` |
| `f32`, `f64` | IEEE brojevi sa pokretnim zarezom, jednostruka i dvostruka preciznost | `1.61803`, `3.14f32`, `6.0221e23f64` |
| `bool` | Boolean | `true`, `false` |
| `char` | Unicode znak, 32 bita širok | `'*'`, `'\n'`, `'字'`, `'\x7f'`, `'u{CA0}'` |

| Tip | Opis | Vrednosti |
|---|---|---|
| `(char, u8, i32)` | Torka: dozvoljeni mešoviti tipovi | `('%', 0x7f, -1)` |
| `()` | „jedinica“ (prazna torka) | `()` |
| `struct S { x: f32, y: f32 }` | Imenovan-polje strkut | `S { x: 120.0, y: 209.0 }` |
| `struct T (i32, char);` | Struktura slična torci | `T(120, 'X')` |
| `struct E;` | Struktura slična jedinici; nema polja | `E` |
| `enum Atend { OnTime, Late(u32) }` | Nabrajanje, algebarski tip podataka | `Attend::Late(5)`, `Attend::OnTime` |
| `Box<Attend>` | Box: poseduje pokazivač na vrednost u hipu | `Box::new(Late(15))` |
| `&i32`, `&mut i32` | Deljene i izmenljive reference: ne posedujući pokazivači koji ne smeju da nadžive svoje referente | `&s.i`, `&mut v` |
| `String` | UTF-8 string, dinamičke veličine | `"ラーメン: ramen".to_string()` |
| `&str` | Referenca na `str`: neposedujući pokazivač na UTF-8 tekst | `"„そば: soba"`, `&s[0..12]` |
| `[f64; 4]`, `[u8; 256]` | Niz, fiksna dužina; svi elementi istog tipa | `[1.0, 0.0, 0.0, 1.0]`, `[b' '; 256]` |
| `Vec<f64>` | Vektor, promenljiva dužina; svi elementi istog tipa | `vec![0.367, 2.718, 7.389]` |
| `&[u8]`,`&mut [u8]` | Referenca na deo: referenca na deo niza ili vektora, koji sadrži pokazivač i dužinu | `&v[10..20]`, `&mut a[..]` |
| `Option<&str>` | Opcionalna vrednost: ili Non (odsutan) ili `Some(v)` (prisutan, sa vrednošću v) | `Some("Dr.")`, `None` |
| `Result<u64, Error>` | Rezultat operacije koja može da ne uspe: ili vrednost uspeha `Ok(v)`, ili greška `Err(e)` | `Ok(4096)`, `Err(Error::last_os_error())` |
| `&dyn Any`, `&mut dyn Read` | Objekat osobine: referenca na bilo koju vrednost koja implementira dati skup metoda | `vrednost kao &dyn Any`, `&mut fajl kao &mut dyn Read` |
| `fn(&str) -> bool` | Pokazivač na funkciju | `str::is_empty` |
| (Tipovi zatvaranja nemaju pisani oblik) | Zatvaranje | `\|a, b\| { a*a + b*b }` |

Većina ovih tipova je obrađena u ovom poglavlju, osim sledećih:

- `struct` tipovi imaju sopstveno Poglavlje 9.
- Nabrojivi tipovi imaju sopstveno Poglavlje 10.
- Objekti osobina su u Poglavlju 11.
- Osnove za `String` i `&str` su opisane ovde, ali pružamo više detalja u Poglavlju 17.
- Funkcije i tipovi zatvaranja dati su u Poglavlju 14.

# Numerički tipovi fiksne širine

Osnova Rustovog sistema tipova je kolekcija numeričkih tipova fiksne širine, izabranih da odgovaraju tipovima koje skoro svi savremeni procesori implementiraju direktno u hardver.

Numerički tipovi fiksne širine mogu da se preliju (prekoračenje vrednosti) ili izgube preciznost, ali su adekvatni za većinu aplikacija i mogu biti hiljadama puta brži od oblika kao što su celi brojevi proizvoljne preciznosti i tačni racionali. Ako su vam potrebne takve vrste numerika, one su podržane u `num` kašeti.

Imena Rustovih numeričkih tipova prate pravilan obrazac, navodeći njihovu širinu u bitovima i predstavljanje koju koriste (Tabela 3-2).

*Tabela 3-2. Rust numerički tipovi*

| Veličina (bitovi) | Ceo broj bez predznaka | Ceo broj sa predznakom | Sa pokretni zareom |
|---|---|---|---|
| 8 | `u8` | `i8` | |
| 16 | `u16` | `i16` | |
| 32 | `u32` | `i32` | `f32` |
| 64 | `u64` | `i64` | `f64` |
| 128 | `u128` | `i128` | |
| Mašinska reč | `usize` | `isize` | |

Ovde, *mašinska reč* je vrednost veličine adrese na mašini na kojoj kôd radi, 32 ili 64 bita.

## Celobroji tipovi

Rustovi tipovi celih brojeva bez predznaka koriste svoj puni opseg da predstavljaju pozitivne vrednosti i nulu (Tabela 3-3).

*Tabela 3-3. Rustovi celi brojevi bez predznaka*

| Tip | Opseg |
|---|---|
| `u8` | 0 do $2^8$–1 (0 do 255) |
| `u16` | 0 do $2^{16}$–1 (0 do 65,535) |
| `u32` | 0 do $2^{32}$–1 (0 do 4,294,967,295) |
| `u64` | 0 do $2^{64}$–1 (0 do 18,446,744,073,709,551,615 ili 18 triliona) |
| `u128` | 0 do $2^{128}$–1 (0 do oko $3{,}4\times10^{38}$) |
| `usize` | 0 na $2^{32}$–1 ili $2^{64}$–1 |

Rustovi tipovi celih brojeva sa predznakom koriste komplementarno predstavljanje, koristeći iste obrazaz bitova kao i odgovarajući brojvi bez predznaka da pokriju opseg pozitivnih i negativnih vrednosti (Tabela 3-4).

*Tabela 3-4. Rustovi celi brojevi sa predznakom*

| Tip | Opseg |
|---|---|
| `i8` | $-2^7$ do $2^7-1$ (–128 do 127) |
| `i16` | $-2^{15}$ do $2^{15}-1$ (–32,768 do 32,767) |
| `i32` | $-2^{31}$ do $2^{31}-1$ (–2,147,483,648 do 2,147,483,647) |
| `i64` | $-2^{63}$ do $2^{63}-1$ (–9,223,372,036,854,775,808 do 9,223,372,036,854,775,807) |
| `i128` | $-2^{127}$ do $2^{127}-1$ (oko $-1{,}7\times10^{38}$ do $+1{,}7\times10^{38}$) |
| `isize` | Ili $-2^{31}$ do $2^{31}-1$, ili $-2^{63}$ do $2^{63}-1$ |

Rust koristi tip `u8` za vrednosti bajtova. Na primer, čitanje podataka iz binarne datoteke ili priključka (eng. socket) daje tok vrednosti `u8`.

Za razliku od C i C++, Rust tretira znakove kao različite od numeričkih tipova: `char` nije `u8`, niti je `u32` (iako dugačak je 32 bita). Rustov tip `char` opisujemo u Znakovi (strana 56).

Tipovi `usize` i `isize` su analogni sa `size_t` i `ptrdiff_t` u C i C++. Njihova preciznost odgovara veličini adresnog prostora na ciljnoj mašini: dugački su 32 bita na 32-bitnim arhitekturama i 64 bita na 64-bitnim arhitekturama. Rust zahteva da indeksi niza budu vrednosti `usize`. Vrednosti koje predstavljaju veličine nizova ili vektora ili brojanje broja elemenata u nekoj strukturi podataka takođe generalno imaju tip `usize`.

Celobrojni literali u Rustu mogu imati sufiks koji označava njihov tip: `42u8` je `u8` vrednost, a `1729isize` je `isize`. Ako celobrojnom literalu nedostaje sufiks tip, Rust odlaže određivanje njegovog tipa sve dok ne pronađe vrednost koja se koristi na način da je smesti: uskladištena u promenljivoj određenog tipa, prosleđena funkciji koja očekuje određeni tip, upoređena sa drugom vrednošću određenog tipa, ili nešto slično. Na kraju, ako više tipova može da funkcioniše, Rust podrazumevano postavlja `i32` ako je to jedna od mogućnosti. U protivnom, Rust prijavljuje dvosmislenost kao grešku.

Prefiksi `0x`, `0o` i `0b` označavaju heksadecimalne, oktalne i binarne literale.

Da biste dugačke brojeve učinili čitljivijim, možete da ubacite donje crte među cifre. Na primer, najveću vrednost `u32` možete zapisati kao `4_294_967_295`. Tačan položaj donjih crta nije značajan, tako da možete da razbijete heksadecimalne ili binarne brojeve u grupe od četiri cifre, a ne tri, kao u `0xffff_ffff`, ili da odvojite sufiks tip od cifara, kao u `127_u8`. Neki primeri celobrojnih literala su ilustrovani u Tabeli 3-6.

*Tabela 3-5. Primeri celobrojnih literala*

| Literal | Tip | Decimalna vrednost |
|---|---|---|
| `116i8` | `i8` | 116 |
| `0xcafeu32` | `u32` | 51966 |
| `0b0010_1010` | Izvedeno | 42 |
| `0o106` | Izvedeno | 70 |

Iako su numerički tipovi i tip `char` različiti, Rust pruža *literale bajta*, literale slične znakovima za `u8` vrednosti : `b'X'` predstavlja ASCII kôd za znak `X`, kao vrednost `u8`. Na primer, pošto je ASCII kôd za `A` 65, literali `b'A'` i `65u8` su potpuno ekvivalentni. U literalima bajta mogu se pojaviti samo ASCII znakovi.

Postoji nekoliko znakova koje ne možete jednostavno staviti iza jednostrukog navodnika, jer bi to bilo sintaktički dvosmisleno ili teško čitljivo. Znakovi u Tabeli 3-6 mogu se pisati samo pomoću rezervne notacije, uvedene obrnutom kosom crtom.

*Tabela 3-6. Znakovi za koje je potrebna rezervna notacija*

| Znak | Bajt literal | Numerički ekvivalent |
|---|---|---|
| Jednostruki navodnik, ' | `b'\''` | `39u8` |
| Obrnuta kosa crta, \ | `b'\\'` | `92u8` |
| Novi red | `b'\n'` | `10u8` |
| Povratak na početak reda | `b'\r'` | `13u8` |
| Tab | `b'\t'` | `9u8` |

Za znakove koje je teško napisati ili pročitati, možete umesto njih napisati njihov kôd u heksadecimalnom obliku. Literal bajta u obliku `b'kHH'`, gde je `HH` bilo koji dvocifreni heksadecimalni broj, predstavlja bajt čija je vrednost `HH`. Na primer, možete napisati literal bajta za ASCII „escape“ kontrolni znak kao `b'x1b'`, pošto je ASCII kôd za „escape“ 27, ili 1B u heksadecimalnom. Pošto su literali bajta samo još jedna oznaka za `u8` vrednosti, razmislite da li bi jednostavan numerički literal mogao biti čitljiviji: verovatno ima smisla koristiti `b'\x1b'` umesto jednostavnog `27` samo kada želite da naglasite da vrednost predstavlja ASCII kôd.

Možete da konvertujete iz jednog celobrojnog tipa u drugi pomoću operatora `as`. Objašnjavamo kako konverzije funkcionišu u Konverzija tipova (strana 141), ali evo nekoliko primera:

```
assert_eq!(   10_i8  as u16,    10_u16); // u opsegu
assert_eq!( 2525_u16 as i16,  2525_i16); // u opsegu

assert_eq!(   -1_i16 as i32,    -1_i32); // sa znakom proširen
assert_eq!(65535_u16 as i32, 65535_i32); // sa nulom proširen
```

```
// Konverzije koje izlaze iz opsega stvaraju vrednosti
// koje su ekivavalent početnom modulu 2^N,
// gde je N širina odredišta u bitima.
// To se neki put naziva "odsecanje."
assert_eq!( 1000_i16 as  u8,   232_u8);
assert_eq!(65535_u32 as i16,    -1_i16);

assert_eq!(   -1_i8  as u8,    255_u8);
assert_eq!(  255_u8  as i8,     -1_i8);
```

Standardna biblioteka obezbeđuje neke operacije kao metode nad celim brojevima. Na primer:

```
assert_eq!(2_u16.pow(4), 16);           // eksponent
assert_eq!((-4_i32).abs(), 4);          // apsolutna vrednost
assert_eq!(0b101101_u8.count_ones(), 4); // prebrojavanje
```

Ove možete pronaći u dokumentaciji na mreži. Imajte na umu, međutim, da dokumentacija sadrži odvojene stranice za sam tip pod „`i32` (primitivni tip)“ i za modul posvećen tom tipu (potražite „`std::i32`“).

U stvarnom kodu, obično nećete morati da pišete sufikse tipa kao što smo uradili ovde, jer će kontekst odrediti tip. Međutim, kada ne odredi, poruke o grešci mogu biti iznenađujuće. Na primer, sledeće se ne kompajlira:

```
println!("{}", (-4).abs());
```

Rust se žali:

```
error: can't call method `abs` on ambiguous numeric type `{integer}`
```

Ovo može biti malo zbunjujuće: svi celi brojevi sa predznakom imaju metodu `abs`, pa u čemu je problem? Iz tehničkih razloga, Rust želi da zna tačno koju vrednost celobrojni tip ima pre nego što pozove sopstvene metode tipa. Podrazumevana vrednost `i32` se primenjuje samo ako je tip i dalje dvosmislen nakon što su svi pozivi metoda razrešeni, tako da je prekasno da se reši. Rešenje je da navedete koji tip nameravate, bilo sa sufiksom ili korišćenjem funkcije određenog tipa:

```
println!("{}", (-4_i32).abs());
println!("{}", i32::abs(-4));
```

Imajte na umu da pozivi metoda imaju veći prioritet od operatora unarnog prefiksa, pa budite pažljivi kada primenjujete metode na negirane vrednosti. Bez zagrada oko `-4_i32` u prvoj izjavi, `-4_i32.abs()` bi primenio metodu `abs` na pozitivnu vrednost `4`, proizvodeći pozitivno `4`, a zatim to negirali, proizvodeći `-4`.

## Provera, omotavanje, zasićenje i prekoračenje aritmetike

Kada se celobrojna aritmetička operacija prekorači, Rust se uspaniči pri izgradnji progama u procsu otklanjanje grešaka (debag). U verziji izgrađenog programa (eng. release build), operacija *omotavanja* (eng. wraps around): proizvodi vrednost ekvivalentnu matematički ispravnom rezultatu po modulu opsega vrednosti. (Ni u jednom slučaju prekoračenje nije nedefinisano ponašanje, kao što je to u C i C++.)

Na primer, sledeći kôd se uspaniči u debag izgradnji:

```
let mut i = 1;
loop {
    i *= 10; // panika: pokušaj množenja sa prekoračenjem
             // (ali samo u debag izgradnji!)
}
```

U verziji izgrađenog programa, ovo množenje se omotava u negativan broj, a petlja se izvodi neograničeno.

Kada ovo podrazumevano ponašanje nije ono što vam treba, celobrojni tipovi pružaju metode koje vam omogućavaju da navedete tačno šta želite. Na primer, sledeće paniči u svakoj verziji izgradnje:

```
let mut i: i32 = 1;
loop {
    // panika: prekoračenje množenja (u svakoj izgradnji)
    i = i.checked_mul(10).expect("prekoračenje množenja");
}
```

Ove celobrojne aritmetičke metode spadaju u četiri opšte kategorije:

- *Proverene* operacije vraćaju `Option` rezultata: `Some(v)` ako se matematički tačan rezultat može predstaviti kao vrednost tog tip, ili `None` ako ne može. Na primer:

```
// Zbir 10 i 20 može da se predstavi kao u8.
assert_eq!(10_u8.checked_add(20), Some(30));

// Nažalost suma 100 i 200 ne može.
assert_eq!(100_u8.checked_add(200), None);

// Radi dodavanje; paniči ako je prekoračenje
let sum = x.checked_add(y).unwrap();

// Čudno, deljenje sa predznakom može prekoračiti, u jednom posebnom slučaju.
// Sa predznakom n-bit tip može biti -2^(n-1), ali ne 2^(n-1).
assert_eq!((-128_i8).checked_div(-1), None);
```

- *Operacije omotavanja* vraćaju vrednost koja je ekvivalentna matematički ispravnom rezultatu po modulu opsega vrednosti:

```
// Prvi proizvod se može predstaviti kao u16
// drugi ne može, tako da dobijamo 250000 modulo 2^16.
assert_eq!(100_u16.wrapping_mul(200), 20000);
assert_eq!(500_u16.wrapping_mul(500), 53392);

// Operacije na tipovima sa predznakom mogu biti omotane na negativne vrednosti.
assert_eq!(500_i16.wrapping_mul(500), -12144);

// U shift (pomeranje) operacijama na bitovima, shift rastojanje
// je omotano da upadne unutar veličene vrednosti
// Tako da shift od 17 bitova u 16-bitnom tipu je shift od 1
assert_eq!(5_i16.wrapping_shl(17), 10);
```

Kao što je objašnjeno, ovako se ponašaju obični aritmetički operatori u verzijama izgradnje. Prednost ovih metoda je u tome što se ponašaju na isti način u svim izgradnjama.

- Operacije *zasićenja* vraćaju reprezentabilnu vrednost koja je najbliža matematički ispravnom rezultatu. Drugim rečima, rezultat se „lepi“ na maksimalne i minimalne vrednosti koje tip može da predstavlja:

```
assert_eq!(32760_i16.saturating_add(10), 32767);
assert_eq!((-32760_i16).saturating_sub(10), -32768);
```

- Ne postoje metode zasićenja deljenja, ostatka ili pomeranja po bitu.
- Operacije *prekoračenja* vraćaju torke `(result, overflowed)`, gde je `result` ono što bi verzija omotavajuće funkcije vratila, i `overflowed` je `bool` koji pokazuje da li je došlo do prekoračenja:

```
assert_eq!(255_u8.overflowing_sub(2), (253, false));
assert_eq!(255_u8.overflowing_add(2), (1, true));
```

`overfloving_shl` i `overfloving_shr` malo odstupaju od obrasca: vraćaju tačno za `overflowed` samo ako je rastojanje pomeranja bilo veliko kao ili veće od širine bita samog tipa. Stvarno primenjeni shift je traženi shift po modulu širine bita tipa:

```
// Shift od 17 bita je prevelika za `u16`, a 17 modulo 16 je 1.
assert_eq!(5_u16.overflowing_shl(17), (10, true));
```

Nazivi operacija koje prate prefiksi `checked_`, `wrapping`, `saturating_` ili `overflowing_` prikazani su u Tabeli 3-7.

*Tabela 3-7. Nazivi operacija*

| Operacija | Sufiks imena | Primer |
|---|---|---|
| Sabiranje | `add` | `100_i8.checked_add(27) == Some(127)` |
| Oduzimanje | `sub` | `10_u8.checked_sub(11) == None` |
| Množenje | `mul` | `128_u8.saturating_mul(3) == 255` |
| Deljenje | `div` | `64_u16.wrapping_div(8) == 8` |
| Ostatak | `rem` | `(-32768_i16).wrapping_rem(-1) == 0` |
| Negacija | `neg` | `(-128_i8).checked_neg() == None` |
| Apsolutna vrednost | `abs` | `(-32768_i16).wrapping_abs() == -32768` |
| Eksponent | `pow` | `3_u8.checked_pov(4) == Some(81)` |
| Pomak po bitovima levo | `shl` | `10_u32.wrapping_shl(34) == 40` |
| Pomak po bitovima desno | `shr` | `40_u64.wrapping_shr(66) == 10` |

## Tipovi sa pokretnim zarezom

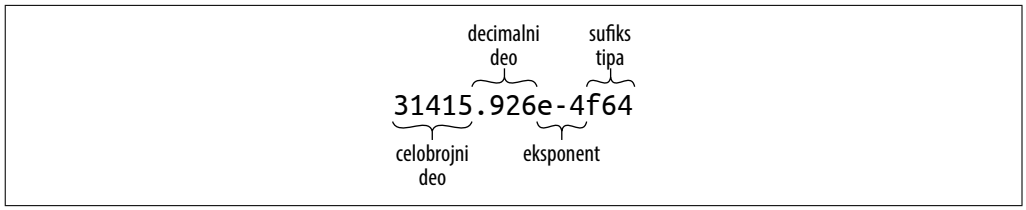
Rust obezbeđuje IEEE tipove sa pokretnim zarezom jednostruke i dvostruke preciznosti. Ovi tipovi uključuju pozitivne i negativne beskonačnosti, različite pozitivne i negativne nulte vrednosti i vrednost *not-a-number* (Tabela 3-8).

*Tabela 3-8. IEEE tipovi sa pokretnim zarezom jednostruke i dvostruke preciznosti*

| Tip | Preciznost | Opseg |
|---|---|---|
| `f32` | IEEE jednostruka preciznost (najmanje 6 decimalnih cifara) | Oko $-3{,}4 \times 10^{38}$ do $+3{,}4 \times 10^{38}$ |
| `f64` | IEEE dvostruka preciznost (najmanje 15 decimalnih cifara) | Oko $-1{,}8 \times 10^{308}$ do $+1{,}8 \times 10^{308}$ |

Rustovi `f32` i `f64` odgovaraju tipovima `float` i `double` u C i C++ (u implementacijama koje podržavaju IEEE pokretni zarez) kao i Java (koja uvek koristi IEEE pokretni zarez).

Literali sa pokretnim zarezom imaju opšti oblik dijagramiran na Slici 3-1.




*Slika 3-1. Literal sa pokretnim zarezom*

Svaki deo broja sa pokretnim zarezom posle celog dela je opcion, ali najmanje jedan deo decimala, eksponenta ili sufiksa tipa mora da bude prisutan da bi se razlikovao od celobrojnog literala. Decimalni deo se može sastojati od usamljene decimale, tako da je `5.` važeća konstanta sa pokretnim zarezom.

Ako literalu sa pokretnim zarezom nedostaje sufiks tipa, Rust proverava kontekst da bi video kako se vrednosti koriste, kao što to čini za celobrojne literale. Ako na kraju utvrdi da bi bilo koji tip sa pokretnim zarezom mogao da stane, on podrazumevano bira `f64`.

Za potrebe zaključivanja tipa, Rust tretira literale sa celim brojem i literale sa pokretnim zarezom kao različite klase: nikada neće zaključiti tip sa pokretnim zarezom za celobrojni literal, ili obrnuto. Tabela 3-9 prikazuje neke primere literala sa pokretnim zarezom.

*Tabela 3-9. Primeri literala sa pokretnim zarezom*

| Literal | Tip | Matematička vrednost |
|---|---|---|
| `-1,5625` | Izvedeno | $-(1\ ^{9}/_{16})$ |
| `2.` | Izvedeno | 2 |
| `0.25` | Izvedeno | ¼ |
| `1e4` | Izvedeno | 10,000 |
| `40f32` | `f32` | 40 |
| `9.109_383_56e-31f64` | `f64` | Oko $9{,}10938356 \times 10^{-31}$ |

Tipovi `f32` i `f64` imaju povezane konstante za posebne vrednosti koje zahteva IEEE kao što su `INFINITY`, `NEG_INFINITY` (negativna beskonačnost), `NAN` (vrednost koja nije broj), i `MIN` i `MAX` (najveća i najmanja konačna vrednost):

```
assert!((-1. / f32::INFINITY).is_sign_negative());
assert_eq!(-f32::MIN, f32::MAX);
```

Tipovi `f32` i `f64` pružaju potpuni komplet metoda za matematičke proračune; na primer, `2f64.skrt()` je kvadratni koren dvostruke preciznosti od dva. Neki primeri:

```
assert_eq!(5f32.sqrt() * 5f32.sqrt(), 5.); // tačno 5.0, po IEEE
assert_eq!((-1.01f64).floor(), -2.0);
```

Opet, pozivi metoda imaju veći prioritet od operatora prefiksa, pa proverite da ste ispravno stavili u zagrade pozive metoda za negirane vrednosti.

Moduli `std::f32::consts` i `std::f64::consts` obezbeđuju različite često korišćene matematičke konstante kao što su `E`, `PI`, i kvadratni koren od dva.

Kada pretražujete dokumentaciju, imajte na umu da postoje stranice za oba tipa, pod nazivom „`f32` (primitivni tip)“ i „`f64` (primitivni tip)“, i moduli za svaki tip, `std::f32` i `std::f64`.

Kao i kod celih brojeva, obično nećete morati da pišete sufikse tipa za literale sa pokretnim zarezom u stvarnom kodu, ali kada to uradite, stavljanje tipa na literal ili funkciju će biti dovoljno:

```
println!("{}", (2.0_f64).sqrt());
println!("{}", f64::sqrt(2.0));
```

Za razliku od C i C++, Rust ne obavlja skoro nikakve numeričke konverzije implicitno. Ako funkcija očekuje argument `f64`, greška je proslediti vrednost `i32` kao argument. U stvari, Rust neće čak implicitno pretvoriti `i16` vrednost u `i32` vrednost, iako je svaka `i16` vrednost takođe `i32` vrednost. Ali uvek možete da zadate *eksplicitne* konverzije koristeći `as` operator: `i as f64`, ili `x as i32`.

Nedostatak implicitnih konverzija ponekad čini Rust izraz opširnijim nego što bi bio odgovarajući C ili C++ kôd. Međutim, implicitne konverzije celih brojeva imaju dobro utvrđenu evidenciju o izazivanju grešaka i bezbednosnih rupa, posebno kada celi brojevi o kojima je reč predstavljaju veličinu nečega u memoriji, a dolazi do neočekivanog prekoračenja. Prema našem iskustvu, čin pisanja numeričkih konverzija u Rustu nas je upozorio na probleme koje bismo inače propustili.

Objašnjavamo kako se konverzije tačno ponašaju u „Konverzija tipova“ na strani (141).

# Tip bool

Rustov Boolean (logički) tip, `bool`, ima uobičajene dve vrednosti za takve tipove, `true` i `false`. Operatori poređenja poput `==` i `<` proizvode `bool` rezultate: vrednost `2 < 5` je `tačno`.

Mnogi jezici su popustljivi prema korišćenju vrednosti drugih tipova u kontekstima koji zahtevaju Bulovu vrednost: C i C++ implicitno pretvaraju znakove, cele brojeve, brojeve sa po-

kretnim zarezom i pokazivače u Bulove vrednosti, tako da se mogu direktno koristiti kao uslov u izjavi `if` ili `while`. Python dozvoljava stringove, liste, rečnike, pa čak i skupove u Bulovim kontekstima, tretirajući takve vrednosti kao istinite ako nisu prazne. Rust je, međutim, veoma strog: kontrolne strukture poput `if` i `while` zahtevaju da njihovi uslovi budu `bool` izrazi, kao i logički operatori `&&` i `||`. Morate napisati `if x != 0 { ... }`, a ne samo `if x { ... }`.

Rustov `as` operator može da konvertuje `bool` vrednosti u celobrojne tipove:

```
assert_eq!(false as i32, 0);
assert_eq!(true  as i32, 1);
```

Međutim, `as` neće konvertovati u drugom pravcu, iz numeričkih tipova u `bool`. Umesto toga, morate napisati eksplicitno poređenje kao što je `x != 0`.

Iako `bool` treba samo jedan bit, Rust koristi ceo bajt za `bool` vrednost u memoriji, tako da možete da kreirate pokazivač na njega.

## Znakovi

Rustov znakovni tip `char` predstavlja jedan Unicode znak, kao 32-bitnu vrednost.

Rust koristi tip `char` za pojedinačne samostalne znakove, ali koristi UTF-8 kodiranje za stringove i tokove teksta. Dakle, `String` predstavlja svoj tekst kao niz UTF-8 bajtova, a ne kao niz znakova.

Literali znakova su znakovi zatvoreni u jednostrukim navodnicima, kao što su `'8'` ili `'!'`. Možete koristiti pun opseg Unicoda: `'錆'` je `char` literal koji predstavlja japanski kanji za *sabi* (rđa, eng. rust).

Kao i kod literala bajta, za nekoliko znakova je obavezna obrnuta kosa crta (Tabela 3-10).

*Tabela 3-10. Znakovi za koje je potrebna obrnuta kosa crta*

| Znak | Rust literal znaka |
|---|---|
| Jednostruki navodnik, ' | `'\''` |
| Obrnuta kosa crta, \ | `'\\'` |
| Novi red | `'\n'` |
| Odlazak na početak reda | `'\r'` |
| Tab | `'\t'` |

Ako želite, možete da napišete Unicode kodnu tačku znaka heksadecimalno:

- Ako je kodna tačka znaka u opsegu U+0000 do U+007F (to jest, ako je izvučena iz ASCII skupa znakova), onda znak možete napisati kao `'\xHH'`, gde je `HH` dvocifreni heksadecimalni broj. Na primer, literali `'*'` i `'x2A'` su ekvivalentni, jer je kodna tačka znaka `*` je 42, ili 2A heksadecimaln.

- Možete da napišete bilo koji Unicode znak kao `'\u{HHHHHH}'`, gde je `HHHHHH` heksadecimalni broj dužine do šest cifara, sa donjom crtom koja je dozvoljena za grupisanje kao i obično. Na primer, literal `'\u{CA0}'` predstavlja znak „ಠ“, Kannada znak koji se koristi u Unicode kao znak neodobravanja, „ಠ_ಠ“. Isti literal može se jednostavno napisati kao `'ಠ'`.

`char` uvek sadrži Unicode kodnu tačku u opsegu od 0x0000 do 0xD7FF, ili od 0xE000 do 0x10FFFF. `char` nikada nije polovina surogatnog para (to jest, kodna tačka u opsegu od 0xD800 do 0xDFFF), ili vrednost izvan Unicode kodnog prostora (to jest, veća od 0x10FFFF). Rust koristi sistem tipova i dinamičke provere da bi osigurao da su vrednosti `char` uvek u dozvoljenom opsegu.

Rust nikada implicitno ne konvertuje između `char` i bilo kog drugog tipa. Možete da koristite operator konverzije `as` da biste pretvorili `char` u celobrojni tip; za tipove manje od 32 bita, gornji bitovi vrednosti znaka su odsečeni:

```
assert_eq!('*' as i32, 42);
assert_eq!('‡≤†' as u16, 0xca0);
assert_eq!('‡≤†' as i8, -0x60); // U+0CA0 odsečen na 8 bita, sa predznakom
```

Idući u drugom pravcu, `u8` je jedini tip koji će `as` operator konvertovati u `char`: Rustova namera je da `as` obavlja samo jeftine, nepogrešive konverzije, ali svaki celobrojni tip osim `u8` uključuje vrednosti koje nisu dozvoljene Unicode kodne tačke, tako da bi te konverzije zahtevale provere tokom izvršavanja (eng. run-time). Umesto toga, funkcija standardne biblioteke `std::char::from_u32` uzima bilo koju vrednost `u32` i vraća `Option<char>`: ako `u32` nije dozvoljena Unicode kodna tačka, onda `from_u32` vraća `None`; u suprotnom, vraća `Some(c)`, gde je `c` rezultat `char`.

Standardna biblioteka pruža neke korisne metode za znakove, koje možete potražiti u onlajn dokumentaciji pod „`char` (primitivni tip)“, i modul „`std::char`“. Na primer:

```
assert_eq!('*'.is_alphabetic(), false);
assert_eq!('Ⓔ≤'.is_alphabetic(), true);
assert_eq!('8'.to_digit(10), Some(8));
assert_eq!('‡≤†'.len_utf8(), 3);
assert_eq!(std::char::from_digit(2, 10), Some('2'));
```

Prirodno, pojedinačni samostalni znakovi nisu toliko zanimljivi kao stringovi i tokovi teksta. Rustov standardni `String` tip i rukovanje tekstom uopšteno ćemo opisati u String tipovi na strani 67.

# Torke

*Torka* (eng. tuple) je par, ili trostruka, četvorostruka, petostruka, itd. (dakle, *n-torka*, ili *torka*), vrednost različitih tipova. Možete napisati torku kao niz elemenata, odvojenih zarezima i okruženih zagradama. Na primer, `("Brazil", 1985)` je torka čiji je prvi element statički dodeljen string, a čiji je drugi ceo broj; njegov tip je `(&str, i32)`. S obzirom na vrednost torke `t`, možete pristupiti njenim elementima kao `t.0`, `t.1`, itd.

U određenoj meri, torke liče na nizove: oba tipa predstavljaju uređeni niz vrednosti. Mnogi programski jezici spajaju ili kombinuju ova dva koncepta, ali u Rustu, oni su potpuno odvojeni. Kao prvo, svaki element torke može imati drugačiji tip, dok elementi niza moraju biti istog tipa. Dalje, torke dozvoljavaju samo konstante kao indekse, kao što je `t.4`. Ne možete napisati `t.i` ili `t[i]` da biste dobili `i`-ti element.

Rust kôd često koristi torke tipove za vraćanje više vrednosti iz funkcije. Na primer, metoda `split_at` na delovima stringa, koja deli string na dve polovine i vraća ih obe, deklarisana je ovako:

```
fn split_at(&self, mid: usize) -> (&str, &str);
```

Povratni tip `(&str, &str)` je skup od dva isečka niza. Možete koristiti sintaksu podudaranja obrzaca da dodelite svaki element povratne vrednosti različitoj promenljivoj:

```
let text = "I see the eigenvalue in thine eye";
let (head, tail) = text.split_at(21);
assert_eq!(head, "I see the eigenvalue ");
assert_eq!(tail, "in thine eye");
```

Ovo je čitljivije od ekvivalentnog:

```
let text = "I see the eigenvalue in thine eye";
let temp = text.split_at(21);
let head = temp.0;
let tail = temp.1;
assert_eq!(head, "I see the eigenvalue ");
assert_eq!(tail, "in thine eye");
```

Takođe ćete videti torke koji se koriste kao neka vrsta minimalno dramskog tipa strukture. Na primer, u Mandelbrotovom programu u Poglavlju 2, morali smo da prosledimo širinu i visinu slike funkcijama koje je crtaju i zapisuju je na disk. Mogli bismo da deklarišemo strukturu sa članovima `width` i `height`, ali to je prilično teška notacija za nešto tako očigledno, pa smo samo koristili torku:

```
/// Upisuje bafer `pixels`, čije su dimenzije date sa `bounds`,
/// u fajl `filename`.
fn write_image(filename: &str, pixels: &[u8], bounds: (usize, usize))
    -> Result<(), std::io::Error>
{ ... }
```

Tip parametra `bounds` je `(usize, usize)`, torka od dve vrednosti `usize`. Doduše, mogli bismo da napišemo odvojene parametre `width` i `height`, a mašinski kôd bi bio otprilike isti u svakom slučaju. To je pitanje jasnoće. Mi o veličini razmišljamo kao o jednoj vrednosti, a ne o dve, a korišćenje torki nam omogućava da napišemo šta mislimo.

Drugi uobičajeno korišćeni tip torke je nulta torka (eng. zero-tuple) `()`. Ovo se tradicionalno naziva *jedinični tip* jer ima samo jednu vrednost, koju pišemo `()`. Rust koristi tip jedinice gde nema smislene vrednosti, ali kontekst ipak zahteva neku vrstu tipa.

Na primer, funkcija koja ne vraća nikakvu vrednost ima tip vraćanja `()`. Funkcija `std::mem::swap` standardne biblioteke nema značajnu povratnu vrednost; samo razmenjuje vrednosti svoja dva argumenta. Deklaracija za `std::mem::swap` glasi:

```
fn swap<T>(x: &mut T, y: &mut T);
```

`<T>` znači da je `swap` *generički*: možete ga koristiti za reference na vrednosti bilo kog tipa `T`. Ali signatura u potpunosti izostavlja `swap` tip povratka, što je skraćenica za vraćanje tipa jedinice:

```
fn swap<T>(x: &mut T, y: &mut T) -> ();
```

Slično, primer `write_image` koji smo ranije spomenuli ima povratni tip `Result<(), std::io::Error>`, što znači da funkcija vraća `std::io::Error` vrednost ako nešto krene naopako, ali ne vraća vrednost nakon uspeha.

Ako želite, možete da uključite zarez posle poslednjeg elementa torke: tipovi `(&str, i32,)` i `(&str, i32)` su ekvivalentni, kao i izrazi `("Brazil", 1985,)` i `("Brazil", 1985)`. Rust dosledno dozvoljava dodatni zarez na kraju svuda gde se koriste zarezi: argumenti funkcije, nizovi, definicije strukture i enuma itd. Ovo može izgledati čudno čoveku, ali olakšava čitanje razlika kada se unosi dodaju i uklanjaju na kraju liste.

Radi doslednosti, postoje čak i torke koje sadrže jednu vrednost. Literal `("usamljena srca",)` je torka koja sadrži jedan string; njegov tip je `(&str,)`. Ovde je zarez posle vrednosti neophodan da bi se razlikovala jednostruka torka od jednostavnog izraza u zagradi.

## Tipovi pokazivača

Rust ima nekoliko tipova koji predstavljaju memorijske adrese.

Ovo je velika razlika između Rusta i većine jezika sa prikupljanjem smeća (eng. garbage collection). U Javi, ako `class Rectangle` sadrži polje `Vector2D upperLeft;`, onda je `upperLeft` referenca na drugi posebno kreiran `Vector2D` objekat. Objekti nikada fizički ne sadrže druge objekte u Javi.

Rust je drugačiji. Jezik je dizajniran da pomogne da se alokacija svedu na minimum. Vrednosti se podrazumevano ugnežđavaju. Vrednost `((0, 0), (1440, 900))` se čuva kao četiri susedna cela broja. Ako je smestite u lokalnu promenljivu, imate lokalnu promenljivu široku četiri cela broja. Ništa nije alocirano na hip (eng. heap) memoriju.

Ovo je odlično za efikasnost memorije, ali kao posledica toga, kada su Rust programu potrebne vrednosti da bi ukazivale na druge vrednosti, on mora eksplicitno da koristi pokazivačke tipove. Dobra vest je da su tipovi pokazivača koji se koriste u bezbednom Rustu ograničeni da eliminišu nedefinisano ponašanje, tako da je pokazivače mnogo lakše pravilno koristiti u Rustu nego u C++.

Ovde ćemo razgovarati o tri tipa pokazivača: reference, kutije i nebezbedni pokazivači.