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` `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

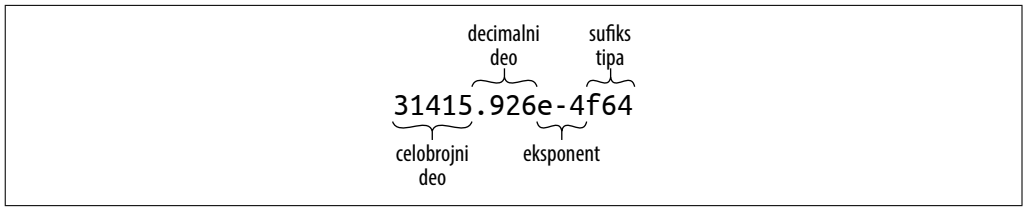
Rust obezbeđuje IEEE tipove sa pokretnim zarezom jednostruke i dvostruke preciznosti. Ovi tipovi uključuju pozitivne i negativne beskonačnosti, različite pozitivne i negativne nulte vrednosti i vrednost *not-a-number* (Tabela 3-8).

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.