Dr Dinu Dragan

PARALELNE I DISTRIBUIRANE ARHITEKTURE I JEZICI (ČAS 7)

ŠTA RADIMO DANAS?

- Operator overloading
- Utility osobine

REIMPLEMENTACIJA OPERATORA

OPERATOR OVERLOADING

- Rust podržava operator overloading tako da možete napraviti kompleksne tipove a kasnije ih koristiti sa osnovnim operatorima
- Npr.

```rust
#[derive(Clone, Copy, Debug)]
struct Complex<T> {
    /// Real portion of the complex number
    re: T,

    /// Imaginary portion of the complex number
    im: T,
}
```

- I onda posle mogu da se sabiraju, množe, oduzimaju
- Sve svodi na implementaciju odgovarajućih osobina
- Lista osobina koja se za to koristi nalazi se na sledećem slajdu

OPERATOR OVERLOADING

Category	Trait	Operator
Unary operators	`std::ops::Neg`	`-x`
	`std::ops::Not`	`!x`
Arithmetic operators	`std::ops::Add`	`x + y`
	`std::ops::Sub`	`x - y`
	`std::ops::Mul`	`x * y`
	`std::ops::Div`	`x / y`
	`std::ops::Rem`	`x % y`
Bitwise operators	`std::ops::BitAnd`	`x & y`
	`std::ops::BitOr`	`x | y`
	`std::ops::BitXor`	`x ^ y`
	`std::ops::Shl`	`x << y`
	`std::ops::Shr`	`x >> y`
Compound assignment arithmetic operators	`std::ops::AddAssign`	`x += y`
	`std::ops::SubAssign`	`x -= y`
	`std::ops::MulAssign`	`x *= y`
	`std::ops::DivAssign`	`x /= y`
	`std::ops::RemAssign`	`x %= y`

OPERATOR OVERLOADING

Category	Trait	Operator
Compound assignment arithmetic operators	`std::ops::AddAssign`	`x += y`
	`std::ops::SubAssign`	`x -= y`
	`std::ops::MulAssign`	`x *= y`
	`std::ops::DivAssign`	`x /= y`
	`std::ops::RemAssign`	`x %= y`
Compound assignment bitwise operators	`std::ops::BitAndAssign`	`x &= y`
	`std::ops::BitOrAssign`	`x |= y`
	`std::ops::BitXorAssign`	`x ^= y`
	`std::ops::ShlAssign`	`x <<= y`
	`std::ops::ShrAssign`	`x >>= y`
Comparison	`std::cmp::PartialEq`	`x == y, x != y`
	`std::cmp::PartialOrd`	`x < y, x <= y, x > y, x >= y`
Indexing	`std::ops::Index`	`x[y], &x[y]`
	`std::ops::IndexMut`	`x[y] = z, &mut x[y]`

OVERLOADING – ARITMETIČKI OPERATORI

- U Rust-u **a+b** je zapravo skraćeno od **a.add(b)**, što je metoda **std::ops::Add** osobine (iz standardne biblioteke)
- Slično: **a * b** je skraćeno za **a.mul(b)** iz **std::ops::Mul**, **std::ops::Neg** pokriva **–** operator i tako dalje…
- Da biste koristili add metodu, a ne operator, mora se data osobina dovesti u doseg

```
use std::ops::Add;

assert_eq!(4.125f32.add(5.75), 9.875);
assert_eq!(10.add(20), 10 + 20);
```

- Metoda ima sledeću definiciju:

```
trait Add<Rhs = Self> {
    type Output;
    fn add(self, rhs: Rhs) -> Self::Output;
}
```

OVERLOADING – ARITMETIČKI OPERATORI

- Ako bismo želeli da sabiramo **Complex<i32>** moramo implementirati **Add<Complex<i32>>**

```
use std::ops::Add;

impl Add for Complex<i32> {
    type Output = Complex<i32>;
    fn add(self, rhs: Self) -> Self {
        Complex {
            re: self.re + rhs.re,
            im: self.im + rhs.im,
        }
    }
}
```

- Naravno, besmisleno je implementirati Add za svaki tip zasebno

```
#[derive(Clone, Copy, Debug)]
struct Complex<T> {
    /// Real portion of the complex number
    re: T,

    /// Imaginary portion of the complex number
    im: T,
}
```

OVERLOADING – ARITMETIČKI OPERATORI

- Naravno, besmisleno je implementirati Add za svaki tip zasebno, zato se implementira **Add<Complex<T>>**

```
use std::ops::Add;

impl<T> Add for Complex<T>
where
    T: Add<Output = T>,
{
    type Output = Self;
    fn add(self, rhs: Self) -> Self {
        Complex {
            re: self.re + rhs.re,
            im: self.im + rhs.im,
        }
    }
}
```

- Obratite pažnju da ovakva implementacija ne dozvoljava sabiranje kompleksnih brojeva baziranih na različitim tipovima

OVERLOADING – ARITMETIČKI OPERATORI

- Da bi se omogućilo sabiranje kompleksnih brojeva baziranih na različitim tipovima, mora se malo izmeniti reimplementacija **Add** metode

```
use std::ops::Add;

impl<L, R> Add<Complex<R>> for Complex<L>
where
    L: Add<R>,
{
    type Output = Complex<L::Output>;
    fn add(self, rhs: Complex<R>) -> Self::Output {
        Complex {
            re: self.re + rhs.re,
            im: self.im + rhs.im,
        }
    }
}
```

OVERLOADING – UNARNI OPERATORI

- Rust implementira dva unarna operator:

Trait name	Expression	Equivalent expression
std::ops::Neg	-x	x.neg()
std::ops::Not	!x	x.not()

- Za sve označene brojeve koristi se **Neg** osobina, dok se **Not** osobina koristi za cele neoznačene brojeve i boolean
- Definicije ovih osobina su jednostavne:

```
trait Neg {
    type Output;
    fn neg(self) -> Self::Output;
}

trait Not {
    type Output;
    fn not(self) -> Self::Output;
}
```

OVERLOADING – UNARNI OPERATORI

- Negacija kompleksnog broja, predstavlja jednostavnu negaciju oba njegova dela

```
use std::ops::Neg;

impl<T> Neg for Complex<T>
where
    T: Neg<Output = T>,
{
    type Output = Complex<T>;
    fn neg(self) -> Complex<T> {
        Complex {
            re: -self.re,
            im: -self.im,
        }
    }
}
```

OVERLOADING – BINARNI OPERATORI

- Binarni operatori su implementirani kroz sledeće osobine:

Category	Trait name	Expression	Equivalent expression
Arithmetic operators	`std::ops::Add`	`x + y`	`x.add(y)`
	`std::ops::Sub`	`x - y`	`x.sub(y)`
	`std::ops::Mul`	`x * y`	`x.mul(y)`
	`std::ops::Div`	`x / y`	`x.div(y)`
	`std::ops::Rem`	`x % y`	`x.rem(y)`
Bitwise operators	`std::ops::BitAnd`	`x & y`	`x.bitand(y)`
	`std::ops::BitOr`	`x | y`	`x.bitor(y)`
	`std::ops::BitXor`	`x ^ y`	`x.bitxor(y)`
	`std::ops::Shl`	`x << y`	`x.shl(y)`
	`std::ops::Shr`	`x >> y`	`x.shr(y)`

- Svi tipovi koji implementiraju brojeve, implementiraju i osobine vezane za aritmetičke operatore, dok celobrojni tipovi i boolean implementiraju binarne operatore

OVERLOADING – BINARNI OPERATORI

- Sve osobine koje implementiraju binarne operatore imaju isti opšti oblik kao npr. **std::ops::BitXor** koji implementira ^ operator

```
trait BitXor<Rhs = Self> {
    type Output;
    fn bitxor(self, rhs: Rhs) -> Self::Output;
}
```

OVERLOADING – SLOŽENI OPERATORI

- Pod složenim operatorima podrazumevaju se operatori koji se kombinuju sa operatorom dodele **+=**, **&=** i sl.
  - Tu postoje dva operanda nad kojima se primenjuje operacija a rezultat se smešta u levi operand
  - Povratna vrednost ovog operanda u Rust-u je uvek ( )
- U mnogim programskim jezicima **x +=y** je skraćeno od **x = x + y**, ali ne i u Rust-u
- U Rust-u je **x +=y** skraćeno od **x.add_assign(y)**, gde **add_assign** metoda **std::ops::AddAssign** osobine

```
trait AddAssign<Rhs = Self> {
    fn add_assign(&mut self, rhs: Rhs);
}
```

- Sve osobine koje implementiraju ove složene operatore nalaze se u tabeli na sledećem slajdu

OVERLOADING – BINARNI OPERATORI

- Sve osobine koje implementiraju ove složene operatore :

Category	Trait name	Expression	Equivalent expression
Arithmetic operators	std::ops::AddAssign	x += y	x.add_assign(y)
	std::ops::SubAssign	x -= y	x.sub_assign(y)
	std::ops::MulAssign	x *= y	x.mul_assign(y)
	std::ops::DivAssign	x /= y	x.div_assign(y)
	std::ops::RemAssign	x %= y	x.rem_assign(y)
Bitwise operators	std::ops::BitAndAssign	x &= y	x.bitand_assign(y)
	std::ops::BitOrAssign	x |= y	x.bitor_assign(y)
	std::ops::BitXorAssign	x ^= y	x.bitxor_assign(y)
	std::ops::ShlAssign	x <<= y	x.shl_assign(y)
	std::ops::ShrAssign	x >>= y	x.shr_assign(y)

- Svi tipovi koji implementiraju brojeve, implementiraju i osobine vezane za aritmetičke operatore, dok celobrojni tipovi i boolean implementiraju binarne operatore

OVERLOADING – BINARNI OPERATORI

- Generička implementacija AddAssign za kompleksni broj:

```
use std::ops::AddAssign;

impl<T> AddAssign for Complex<T>
where
    T: AddAssign<T>,
{
    fn add_assign(&mut self, rhs: Complex<T>) {
        self.re += rhs.re;
        self.im += rhs.im;
    }
}
```

- Implementacija **std::ops::Add** ne podrazumeva automatsku implementaciju **std::ops::AddAssign**

OVERLOADING – OPERATORI != I ==

- Rust **!=** i **==** operatori su skraćeni pozivi za **ne** i **qe** metode iz **std::cmp::PartialEq** osobine

```
assert_eq!(x == y, x.eq(&y));
assert_eq!(x != y, x.ne(&y));
```

- Definicija **std::cmp::PartialEq** osobine:

```
trait PartialEq<Rhs = Self>
where
    Rhs: ?Sized,
{
    fn eq(&self, other: &Rhs) -> bool;
    fn ne(&self, other: &Rhs) -> bool {
        !self.eq(other)
    }
}
```

- Pošto **ne** ima predefinisanu implementaciju, dovoljno je implementirati samo **eq**

OVERLOADING – OPERATORI != I ==

- Pošto **ne** ima predefinisanu implementaciju, dovoljno je implementirati samo **eq**

```
impl<T: PartialEq> PartialEq for Complex<T> {
    fn eq(&self, other: &Complex<T>) -> bool {
        self.re == other.re && self.im == other.im
    }
}
```

- Ako se pretpostavi da je implementirano i množenje:

```
let x = Complex { re: 5, im: 2 };
let y = Complex { re: 2, im: 5 };
assert_eq!(x * y, Complex { re: 0, im: 29 });
```

- Implementacija qe metode iz primera je vrlo uobičajena (poređenje odgovarajućeg polja levog operanda sa odgovarajućim poljem desnog operanda), te Rust omogućuje da se ovde automatski izgeneriše

```
#[derive(Clone, Copy, Debug, PartialEq)]
struct Complex<T> {
    ...
}
```

OVERLOADING – OPERATORI != I ==

- **PartialEq** uzima operande putem reference, za razliku od ostalih koji ih uzimaju vrednošću
- To znači da poređenje dovodi do pozajmljivanja
- Kod ove osobine, postoji mala relaksacija, jer se sada mogu porediti tipovi koji nisu Sized (npr. stringovi čija dužina može varirati), što je omogućeno upotrebom **?Sized**

```
where
    Rhs: ?Sized,
```

- To znači da mogu postojati implementacije sledećeg tipa: **PartialEq<str>** ili **PartialEq<[T]>**

OVERLOADING – OPERATORI != I ==

- Zašto se osobine zove PartialEq?
- Zbog toga što Rust implementira f32 i f64 u skladu sa IEEE standardom, te je jednakost samo parcijalno tačna
- U matematičkom smislu, nešto je strogo jednako ako su zadovoljene tri stvari:
  - Ako je **x == y** tačno, onda i **y == x** mora biti tačno
  - Ako je **x == y** i **y == z**, onda mora biti i **x == z**
  - **x == x** je uvek tačno
- Zbog implementacija brojeva sa pokretnim zarezom, ovo poslednje nije uvek tačno
- Po standardu 0.0 / 0.0 je NaN (not-a-number) i on nije ničemu jednako, pa ni samom sebi
- Ako se piše generički kod, onda mora da se koristi PartialEq

OVERLOADING – OPERATORI != I ==

- Ako se želi potpuna jednokost, onda se mora koristiti **Eq** osobina, koja proširuje **PartialEq**

```
trait Eq: PartialEq<Self> {}
```

- Svi tipovi koji implementiraju **PartialEq** implementiraju i **Eq**, osim **f32** i **f64**
- **Eq** ne dodaje nikakve nove metode
- Ako naš tip implementira **PartialEq** a želimo da implementira i **Eq**, mora se eskplicitno implementirati **Eq** iako se ne dodaju nove metode

```
impl<T: Eq> Eq for Complex<T> {}
```

- Naravno, može i Rust to da uradi za nas, ali se mora eksplicitno navesti

```
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
struct Complex<T> {
    ...
}
```

OVERLOADING – OPERATORI POREĐENJA

- Implementacija operatora **<**, **>**, **<=** i **>=** se nalazi u osobini **std::cmp::PartialOrd**

```
trait PartialOrd<Rhs = Self>: PartialEq<Rhs>
where
    Rhs: ?Sized,
{
    fn partial_cmp(&self, other: &Rhs) -> Option<Ordering>;

    fn lt(&self, other: &Rhs) -> bool { ... }
    fn le(&self, other: &Rhs) -> bool { ... }
    fn gt(&self, other: &Rhs) -> bool { ... }
    fn ge(&self, other: &Rhs) -> bool { ... }
}
```

- **PartialOrd** proširuje **PartialEq** te se može primeniti samo nad tipovima koji implementiraju **PartialEq**
- Svaka funkcija uključuje predefinisanu implementaciju, osim **partial_cmp** koju morate implementirati sami

OVERLOADING – OPERATORI POREĐENJA

- Funkcija **partial_cmp** vraća **Some(o)**, gde **o** može imati bilo koju vrednost enumeracije **Ordering**

```
enum Ordering {
    Less,       // self < other
    Equal,      // self == other
    Greater,    // self > other
}
```

- Kada funkcija **partial_cmp** vrati **None**, to znači da ne postoji uređenost između **self** i **other** (ne postoji relacija na osnovu koje bi se odredilo da li su jednaki, veći ili manji)
- Od svih Rust tipova jedino tipovi sa pokretnim zarezom mogu vratiti **None** (šta mislite zašto?)

OVERLOADING – OPERATORI POREĐENJA

- Što se sam implementacije operatora tiče, svaki od relacionih operatora predstavlja skraćeni zapis poziva metode levog operadna sa parametrom koji predstavlja desni operand

Expression	Equivalent method call	Default definition
`x < y`	`x.lt(y)`	`x.partial_cmp(&y) == Some(Less)`
`x > y`	`x.gt(y)`	`x.partial_cmp(&y) == Some(Greater)`
`x <= y`	`x.le(y)`	`matches!(x.partial_cmp(&y), Some(Less) | Some(Equal)`
`x >= y`	`x.ge(y)`	`matches!(x.partial_cmp(&y), Some(Greater) | Some(Equal)`

- Podrazumeva se da je **std::cmp::PartialOrd** i **std::cmp::Ordering** u dosegu

OVERLOADING – OPERATORI POREĐENJA

- Kada su vrednosti dva tipa uvek uređena, tj. kada postoji sigurno definisana relacija između njih, koristi se **std::cmp::Ord**

```
trait Ord: Eq + PartialOrd<Self> {
    fn cmp(&self, other: &Self) -> Ordering;
}
```

- U osnovi proširuje **PartialOrd**
- Funkcija **cmp** kao rezultat vraća **Ordering** umesto **Option<Ordering>**
- Svi tipovi koji implementiraju **PartialOrd** implementiraju i **Ord**, osim **f32** i **f64**
- Primer reimplementacije relacionih operatora se ne može uvesti za kompleksne brojeve, barem ne smislen primer (zašto?)

OVERLOADING – OPERATORI POREĐENJA

- Primer, struktura koja reprezentuje interval **(lower, upper]**:

```
#[derive(Debug, PartialEq)]
struct Interval<T> {
    lower: T, // inclusive
    upper: T, // exclusive
}
```

- Poređenje radi po principu:
  - jedan interval je manji od drugog ako je u potpunosti ispred drugog, bez preklapanja
  - jedan interval je veći od drugog ako je u potpunosti iza drugog, bez preklapanja
  - ako dolazi do preklapanja, onda nisu uređeni (nema relacije)
  - ako su isti, onda su jednaki

OVERLOADING – OPERATORI POREĐENJA

- Implementacija:

```
use std::cmp::{Ordering, PartialOrd};

impl<T: PartialOrd> PartialOrd<Interval<T>> for Interval<T> {
    fn partial_cmp(&self, other: &Interval<T>) -> Option<Ordering> {
        if self == other {
            Some(Ordering::Equal)
        } else if self.lower >= other.upper {
            Some(Ordering::Greater)
        } else if self.upper <= other.lower {
            Some(Ordering::Less)
        } else {
            None
        }
    }
}
```

OVERLOADING – INDEKSI

- Moguće je tipu pridodati i indeks, tj. definisati kako se tip ponaša u izrazima tipa **x[i]** kroz implementaciju osobina **std::ops::Index** i **std::ops::IndexMut**
- Nizovi ovu osobinu implementiraju direktno, ali kod ostalih tipova je zapis **x[i]** skraćeno od ***x.index(i)** gde je **index** metoda osobine **std::ops::Index**
- Ako se pak indeks koristi za dodelu vrednosti ili mutabilno pozajmljivanje, onda je **x[i]** skraćeno od ***x.index_mut(i)** gde je **index_mut** metoda osobine **std::ops::IndexMut**

```
trait Index<Idx> {
    type Output: ?Sized;
    fn index(&self, index: Idx) -> &Self::Output;
}

trait IndexMut<Idx>: Index<Idx> {
    fn index_mut(&mut self, index: Idx) -> &mut Self::Output;
}
```

OVERLOADING – INDEKSI

- Tip povratne vrednosti **index** metode se preuzima kroz parametar
- Na primer, moguće je implementirati isečak samo jednim brojem, jer isečak implementira **Index<usize>**, ali i indeksirati ga rasponom od 2 broja, jer implementira i **Index<Range<usize>>**

```
*a.index(std::ops::Range { start: i, end: j })
```

- **IndexMut** proširuje osobinu **Index** sa **index_mut** metodom koja uzima mutabilnu referencu na **self**, a vraća mutabilnu referencu na izlaznu vrednost
- Rust automatski poziva **index_mut** u situacijama kada to izraz zahteva mutabilno indeksiranje
- Ograničenje uvedeno namerno je da se uvek vraća mutabilna referenca na neku vrednost, tako da prilikom dodele vrednosti, to mora isto biti referenca

OVERLOADING – INDEKSI

- Primer: dvodimenzionalna slika koja omogućuje indeksiranje tipa:

```
image[row][column] = ...;
```

- U suprotnom, bi moralo ovako:

```
pixels[row * bounds.0 + column] = ...;
```

- Potrebna je struktura koja proširuje vektor:

```
struct Image<P> {
    width: usize,
    pixels: Vec<P>,
}

impl<P: Default + Copy> Image<P> {
    /// Create a new image of the given size.
    fn new(width: usize, height: usize) -> Image<P> {
        Image {
            width,
            pixels: vec![P::default(); width * height],
        }
    }
}
```

OVERLOADING – INDEKSI

- Implementacija **Index** i **IndexMut**

```
impl<P> std::ops::Index<usize> for Image<P> {
    type Output = [P];
    fn index(&self, row: usize) -> &[P] {
        let start = row * self.width;
        &self.pixels[start..start + self.width]
    }
}

impl<P> std::ops::IndexMut<usize> for Image<P> {
    fn index_mut(&mut self, row: usize) -> &mut [P] {
        let start = row * self.width;
        &mut self.pixels[start..start + self.width]
    }
}
```

- Kada se indeksira **Image** vraća se nazad isečak piksela, ideksiranje isečka, vraća individualni piksel

OVERLOADING – DRUGI OPERATORI

- Ne mogu se reimplementirati svi Rust operatori: **?**, **||**, **&&** operatori
- Rade na podršci za reimplementaciju **?**

UTILITY TRAITS

UTILITY TRAITS

- Rust sadrži nekoliko vrlo korisnih osobina iz standardne biblioteke koje se mogu koristiti, eventualno proširivati kako bi se pisao Rust idiomatski kod (kod u duhu jezika)
- To su osobine za proširivanje jezika (Language extension traits) – dodatna reimplementacija određenih osobina za sopstvene tipove (**Drop**, **Deref** i **DerefMut**, osobine za konverziju **From** i **Into**)
- Marker osobine (Marker traits) – koriste se za vezivanje promenljivih generičkog tipa za neka ograničenja (**Sized** i **Copy**)
- Osobine javnog rečnika (Public vocabulary traits) – koriste se za javne interfejse kada se želi obezbediti interoperabilnost bez preke potrebe za dodatnim kodom (**Default**, **AsRef**, **AsMut**, **Borrow**, **BorrowMut**, **TryFrom**, **TryInto**, **ToOwned**)

UTILITY TRAITS

Trait	Description
`Drop`	Destructors. Cleanup code that Rust runs automatically whenever a value is dropped.
`Sized`	Marker trait for types with a fixed size known at compile time, as opposed to types (such as slices) that are dynamically sized.
`Clone`	Types that support cloning values.
`Copy`	Marker trait for types that can be cloned simply by making a byte-for-byte copy of the memory containing the value.
`Deref` and `DerefMut`	Traits for smart pointer types.
`Default`	Types that have a sensible "default value."
`AsRef` and `AsMut`	Conversion traits for borrowing one type of reference from another.
`Borrow` and `BorrowMut`	Conversion traits, like `AsRef`/`AsMut`, but additionally guaranteeing consistent hashing, ordering, and equality.
`From` and `Into`	Conversion traits for transforming one type of value into another.
`TryFrom` and `TryInto`	Conversion traits for transforming one type of value into another, for transformations that might fail.
`ToOwned`	Conversion trait for converting a reference to an owned value.

DROP

- Kada vlasnik izađe iz dosega, Rust oslobađa tu vrednost iz memorije (takođe i sve ostale vrednosti na koju se vlasnik referiše, kao i bilo koje druge sistemske resurse) – ova akcija se naziva drop (the value is dropped)
- U većini slučajeva Rust realizuje drop automatski
- Rust poseduje mehanizam kroz koji se može prilagoditi kako Rust radi to brisanje
- Pretpostaviti da postoji sledeća struktura:

```
struct Appellation {
    name: String,
    nicknames: Vec<String>
}
```

- U osnovi Rust može da rukuje brisanjem bilo koje instance ove strukture, jer zna kako da briše vrednosti sa Heapa koje koriste string i vektor

DROP

- Ipak, ako se želi prilagoditi kako Rust radi drop, može se implementirati **drop** metoda iz **std::ops::Drop** osobine

```
trait Drop {
    fn drop(&mut self);
}
```

- Ovo je slično tome da ste implementirali dekstruktor u C++ ili finalizer u nekim drugim jezicima
- Kada Rust briše vrednost, ako je implementirana, pozvaće se **drop** funkcija pre nego što se realizuje predefinisano ponašanje za brisanje
- Ova metoda se samo implicitno poziva, eksplicitni poziv **drop** metode izazvaće grešku
- Kako se metoda poziva pre brisanja, sama vrednost je i dalje inicijalizovana i dozvoljeno je koristiti sve vrednosti (polja, elemente) iz date vrednosti

DROP

- Implementacija **drop** metode za **Appellation** strukturu bi izgledalo ovako

```
impl Drop for Appellation {
    fn drop(&mut self) {
        print!("Dropping {}", self.name);
        if !self.nicknames.is_empty() {
            print!(" (AKA {})", self.nicknames.join(", "));
        }
        println!("");
    }
}
```

DROP

- **Appellation** struktura bi se mogla uporediti na sledeći način:

```
{
    let mut a = Appellation {
        name: "Zeus".to_string(),
        nicknames: vec!["cloud collector".to_string(),
                        "king of the gods".to_string()]
    };

    println!("before assignment");
    a = Appellation { name: "Hera".to_string(), nicknames: vec![] };
    println!("at end of block");
}
```

- Kada se druga vrednost pridoda promenljivoj a, prva se briše, zatim kada a izađe izvan ospega, briše se i druga vrednost, pa se ispisuje sledeće

```
before assignment
Dropping Zeus (AKA cloud collector, king of the gods)
at end of block
Dropping Hera
```

DROP

- **Drop** se najčešće se koristi u situacijama kada koristite neke resurse za koje Rust ne zna, ili ne može da upravlja njima
- Rust-ova standardna biblioteka koristi interno sledeći tip za reprezentaciju deskriptora sistemskih datoteka:

```
struct FileDesc {
    fd: c_int,
}
```

- **c_int** je alijas za **i32**
- **fd** polje **FileDesc** strukture je samo identifikator datoteka koju treba zatvoriti kada program završi sa njom
- **Drop** implementacija u standardnoj biblioteci za **FileDesc** izgleda:

```
impl Drop for FileDesc {
    fn drop(&mut self) {
        let _ = unsafe { libc::close(self.fd) };
    }
}
```

DROP

- **Drop** implementacija u standardnoj biblioteci za **FileDesc** izgleda:

```
impl Drop for FileDesc {
    fn drop(&mut self) {
        let _ = unsafe { libc::close(self.fd) };
    }
}
```

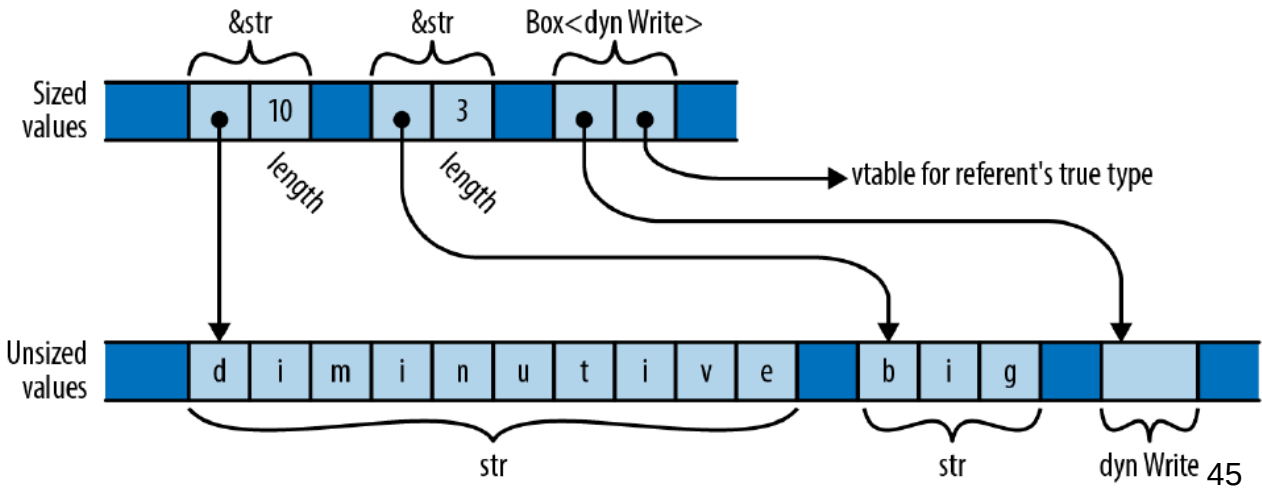
- libc::close je Rust ime za close funkcije C biblioteke
- Rust može da poziva C funkcije samo unutar unsafe blokova, pa se zato koristi
- Ako tip implementira **Drop**, ne može da implementira **Copy** osobinu

SIZED

- **Sized tip** je tip čije su sve vrednosti uvek iste veličine
- Većina Rust tipova je sized, čak i enum
- Vec<T> je zapravo sized, jer njegov deo na steku uvek poznate veličine (bager alociran na heapu je varijabilne veličine)
- Svi sized tipovi implementiraju **std::marker::Sized** osobinu
- Ova osobina nema ni metoda, funkcija
- Rust automatski implementira ovu osobinu za svaki tip koji zadovoljava, ne može se implementirati eskplicitno
- Jedina upotreba je za vezivanje tipova kada se želi naglasiti da u nekoj generičkoj implementaciji tip mora biti sized, npr. **T: Sized**
- Ovakva vrsta osobine se naziva Marker Trait, jer se koriste za markiranje (označavanje) da nešto ima željenu karakteristiku

SIZED

- Rust ima i nekoliko tipova koji su unsized, tj. koji nisu fiksne veličine
- Isečak tipa string, **str** (bez &) nije fiksne veličine
- String literali su rezličitih veličina
- Isečci iz tipova poput [T] nisu fiksne dužine
- Slično je i sa dyn tipovima
- Rust ne može da smesti unsized vrednosti u promenljive, već se oni koriste kroz reference, koji su uvek složeni pokazivači (**fat pointers**) koji imaju dva dela, jedan je pokazivač, a drugi veličina onoga na šta se pokazuje



SIZED

- Zbog toga što je primena Unsized tipova ograničena, većina generičkih promenljivih je ograničena na Sized tipove
- U suštini Sized je podrazumevana vrednost tipa u Rust-u
- Kada napišete:
  struct S<T> { ... }
  Rust podrazumeva da je napisano:
  struct S<T: Sized> { ... }
- Ako se ne želi ograničiti T na taj način, potrebno je to eksplicitno naglasiti:
  struct S<T: ?Sized> { ... }
- **T: ?Sized** znači **“not necessarily Sized”**
- Ako napišete **struct S<T: ?Sized> { b: Box<T> }**, Rust će dozvoliti da se napiše i **S<str>** i **S<dyn Write>**

CLONE

- **std::clone::Clone** osobine se koristi kod tipova koji mogu da naprave kopiju sebe
- Definicija osobine:

```
trait Clone: Sized {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source: &Self) {
        *self = source.clone()
    }
}
```

- Metoda **clone** treba da konstruiše nezavisnu kopiju sebe (**self**) i vrati je nazad
- Sama **Clone** osobina proširuje **Sized** osobinu i **clone** ne može da vrati unsized vrednost
- Samo kloniranje je vrlo skupa operacija, jer zahteva i kloniranje svega ostalog vezanog za tip koji se klonira

CLONE

- Zbog cene kloniranja, Rust insistira na tome da se **clone** poziva eksplicitno
- Kloniranje je jedino jeftino kada se radi nad brojivim referencama, **Rc<T>** i **Arc<T>**
- **clone_from** metoda modifikuje **self** u kopiju izvora (**source**), tj. kopira izvor i onda ga pomera u ***self**
- Ako implementacija **clone** metode podrazumeve kloniranje svakog polja ili elementa tipa koji se klonira i konstruisanje novih vrednosti od kloniranih, onda je nije potrebno to implementirati, tj. podrazumevana implementacija je dovoljna i Rust će to sam implementirati
  - Potrebno je samo dodati #[derive(Clone)] iznad definicije tipa
- Gotovo svaki tip u standardnoj biblioteci (a da je smisleno kopirati ga), implementira **Clone**, čak i **String**, **Vec<T>**, **HashMap**

COPY

- Jednostavni tipovi koji ne sadrže posebne resurse mogu biti **Copy** tipovi za koje dodela vrednosti vrši kopiranje umesto pomeranja
- Tip je **Copy** tip ako implementira **std::marker::Copy** marker osobinu

```
trait Copy: Clone { }
```

- Implementacija je jednostavna:

```
impl Copy for MyType { }
```

- Ova osobina ima posebno značenje za sam jezik, te je dozvoljena samo za tipove kojima je potrebno plitko (shallow) bajt-za-bajt kopiranje
- Tipovi koji poseduje bilo koje druge resurse, poput heap memorije, kopiranje nije dozvoljeno
- Rust će automatski dodeliti Copy osobinu tipu, potrebno je samo dodati #[derive(Copy)] iznad definicije tipa

DEREF I DEREFMUT

- Kako će se ***** i **.** operatori za dereferenciranje ponašati, može se definisati reimplementacijom **std::ops::Deref** i **std::ops::DerefMut** osobina
- Ugrađeni tipovi poput **Box<T>** i **Rc<T>** implementiraju ove operatore u skladu sa prirodom jezika i ponašanjem ugrađenih Rust tipova pokazivača
  - Ako imamo promenljivu **b** tipa **Box<Complex>**, onda ***b** se odnosi na vrednost tipa **Complex** na koju **b** pokazuje, a **b.re** se odnosi njenu komponentu
- Ako je kontekst takav da se dodeljuje ili pozajmljuje promenljiva referenca na neku vrednost, onda Rust koristi **DerefMut** osobinu (“dereference mutably”), ako je dovoljan read-only pristup, koristi se **Deref** osobina

DEREF I DEREFMUT

- Osobine imaju sledeću definiciju:

```
trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}

trait DerefMut: Deref {
    fn deref_mut(&mut self) -> &mut Self::Target;
}
```

- Metode **deref** i **deref_mut** uzimaju **&Self** referencu i vraćaju **&Self::Target** referencu
- **Target** mora biti nešto što **Self** sadrži, poseduje ili se odnosi na; npr. za **Box<Complex>** tip od **Target** je **Complex**
- **DerefMut** proširuje **Deref**
- Kako metoda vraća referencu sa životnim vekom koji je isti kao i životni vek **&self**, **self** ostaje pozajmljena sve dok i referenca živi

DEREF I DEREFMUT

- Rust koristi ove osobine da izvrši automatsku konverziju iz **&Self** u **&Self::Target**
- U prevodu ako bi ubacivanje **deref** poziva omogućilo da se izbegne **type mismatch**, Rust automatski ubacuje poziv
- Naravno, ubacivanje deref_mut omogućava da se izbegne **type mismatch** za promenljive reference
- Ovo se naziva **deref coercions** (nametnuto dereferenciranje), jer se jednom tipu nameće da se ponaša kao drugi tip (Rust ovo može da uradi i više puta za redom ako je potrebno)
- Npr. primena **split_at** direktno nad **Rc<String>** je moguća, jer se **&Rc<String>** dereferencira u **&String**, koji se dereferencira u **&str**, koji poseduje **split_at** metodu

DEREF I DEREFMUT

- Na primer, pretpostaviti sledeću strukturu:

```
struct Selector<T> {
    /// Elements available in this `Selector`.
    elements: Vec<T>,

    /// The index of the "current" element in `elements`. A `Selector`
    /// behaves like a pointer to the current element.
    current: usize
}
```

- Da bi se ona ponašala kako stoji u komentaru, potrebno je implementirati **Deref** i **DerefMut** za datu strukturu

DEREF I DEREFMUT

- Da bi se ona ponašala kako stoji u komentaru, potrebno je implementirati **Deref** i **DerefMut** za datu strukturu

```
use std::ops::{Deref, DerefMut};

impl<T> Deref for Selector<T> {
    type Target = T;
    fn deref(&self) -> &T {
        &self.elements[self.current]
    }
}

impl<T> DerefMut for Selector<T> {
    fn deref_mut(&mut self) -> &mut T {
        &mut self.elements[self.current]
    }
}
```

DEREF I DEREFMUT

- Sa tom implementacijom **Deref** i **DerefMut** osobina **Selektor** se može koristiti na sledeći način:

```
let mut s = Selector { elements: vec!['x', 'y', 'z'],
                       current: 2 };

// Because `Selector` implements `Deref`, we can use the `*` operator to
// refer to its current element.
assert_eq!(*s, 'z');

// Assert that 'z' is alphabetic, using a method of `char` directly on a
// `Selector`, via deref coercion.
assert!(s.is_alphabetic());

// Change the 'z' to a 'w', by assigning to the `Selector`'s referent.
*s = 'w';

assert_eq!(s.elements, ['x', 'y', 'w']);
```

DEFAULT

- Kod nekih tipova postoje očigledne predefinisane početne vrednosti: prazan string, prazan vektor, nula kod brojeva, None kod Option i sl.
- Ovakvi tipovi mogu da implementiraju **std::default::Default** osobinu

```
trait Default {
    fn default() -> Self;
}
```

- Implementacija bi imala sledeći izgled:

```
impl Default for String {
    fn default() -> String {
        String::new()
    }
}
```

- **default** metoda jednostavno vraća novu vrednost tipa **Self**
- Sve kolekciju u Rust-u, **Vec**, **HashMap**, **BinaryHeap** i dr. implementiraju **Default** kod koje **default** metoda vraća praznu vrednost

DEFAULT

- Druga uobičajena primena **Default** je da definiše podrazumevane vrednosti za polja neke strukture (tako da ona ne moraju da se popune, već će se popuniti podrazumevanim vrednostima)
- Primer, glium sanduk implementira metoda za rad sa OpenGL-om, njena struktura **glium::DrawParameters** ima 24 polja koja kontrolišu različite stvari vezane za to kako bi OpenGL trebao nešto da renderuje
- Sva ta polja imaju predefinisane vrednosti, te se prilikom kreiranja instance **DrawParameters** strukture prosleđuju samo vrednosti za polja koja su različita, dok se vrednosti ostalih polja postave na predefinisane vrednosti pozivom **default** metode

```
let params = glium::DrawParameters {
    line_width: Some(0.02),
    point_size: Some(0.02),
    .. Default::default()
};

target.draw(..., &params).unwrap();
```

DEFAULT

- Ako tip **T** implementira **Default**, onda Rust automatski implementira **Default** za **Rc<T>**, **Arc<T>**, **Box<T>**, **Cell<T>**, **RefCell<T>**, **Cow<T>**, **Mutex<T>** i **RwLock<T>**
- Predefinisana vrednost za **Rc<T>** je referenca na predefinisanu vrednost za tip **T**
- Ako svi elementi n-torke implementiraju **Default** onda i n-torka implementira **Default** koja će naravno sadržati predefinisane vrednosti svih elemenata n-torke
- Rust ne implementira **Default** eksplicitno za strukturu čak i ako svako polje strukture implementira **Default**
- Ako sva polja implementiraju **Default** onda je moguće implementirati **Default** automatski za datu strukturu upotrebom **#[derive(Default)]**

ASREF I ASMUT

- **AsRef** i **AsMut** osobine omogućuje implementaciju efikasnog pozajmljivanja za dati tip, prvo je naravno za read-only, drugo je promenljivo pozajmljivanje

```
trait AsRef<T: ?Sized> {
    fn as_ref(&self) -> &T;
}

trait AsMut<T: ?Sized> {
    fn as_mut(&mut self) -> &mut T;
}
```

- **AsRef<T>** znači da se od tipe **T** može efikasno pozajmiti **&T**
- **Vec<T>** implementira **AsRef<[T]>** i **String** implementira **AsRef<str>**, ali implementira i **AsRef<[u8]>** te omogućava efikasno pozajmljivanje sadržaja stringa kao niz bajtova
- **AsRef** se obično koristi kako bi se napravilo da funkcije budu fleksibilnije u tome koje vrste argumenata prihvataju

ASREF I ASMUT

- Na primer, std::fs::File::open funkcija se deklariše na sledeći način:

```
fn open<P: AsRef<Path>>(path: P) -> Result<File>
```

- **Open** funkcija tu želi referencu na putanju koja reprezentuje putanju u fajl sistemu, ali je ovakav zapis čini fleksibilnom u smisli da će prihvatiti bilo šta što implementira **AsRef<Path>** a to podrazumeva i **String** i **str**, ali i **OsString** i **OsStr** (koji su string implementacije sistemskih naziva), i **PathBuf** i **Path** (koje implementiraju putanju)
- Zato je moguće proslediti i string u pozivu **open** funkcije

```
let dot_emacs = std::fs::File::open("/home/jimb/.emacs")?;
```

BORROW I BORROWMUT

- Radi slično kao i **AsRef**, omogućuje efikasno pozajmljivanje, ali uz više restrikcija
- Deo je **std::borrow::Borrow** osobine
- Ako tip **T** implementira **Borrow<T>**, onda **borrow** metoda efikasno pozajmljuje **&T**
- **Borrow** se treba implementirati samo u situacijama kada se &T hešira i poredi na isti način kao i vrednost od koje se pozajmljuje
- Zbog toga je **Borrow** najkorisnija kada se koristi za ključeve u heš tabelama ili stablima ili kada se koristi za vrednosti koje će biti heširane ili upoređivane za neke druge svrhe
- Ako se pozajmljuje od **String** putem **AsRef**, onda će, na primer, sva tri pozajmljivanja, **AsRef<str>**, **AsRef<[u8]>**, **AsRef<Path>** imati različite heš vrednosti

BORROW I BORROWMUT

- Definicija izgleda slično **AsRef** definiciji

```
trait Borrow<Borrowed: ?Sized> {
    fn borrow(&self) -> &Borrowed;
}
```

- **Borrow** je dizajniran za tačno određenu situaciju sa generičkim heš tabelama i drugim asocijativnim tipovima
- Najbolje je objasniti na primeru, pretpostaviti da postoji heš tabale koja mapira string na broj, **std::collections::HashMap<String, i32>**
- Ključ heš tabele je string
- Metoda za traženje zapisa u tabeli mogla bi da ima sledeću izgled:

```
impl<K, V> HashMap<K, V> where K: Eq + Hash
{
    fn get(&self, key: K) -> Option<&V> { ... }
}
```

BORROW I BORROWMUT

- Problem u toj implementaciji što je tip ključa, **K**, u ovom slučaju, **String**, pa se ovde String mora prenositi po vrednosti, što je besmisleno trošenje resursa (mora se klonirati svaki put)
- Zapravo je potrebna samo referenca na ključ

```
impl<K, V> HashMap<K, V> where K: Eq + Hash
{
    fn get(&self, key: &K) -> Option<&V> { ... }
}
```

- Ovo je bolje, ali sada kada se želi proslediti String kao ključ, mora se proslediti njegova referenca, što je malo nakaradno kada se radi sa neposrednim operandima

```
hashtable.get(&"twenty-two".to_string())
```

- U osnovi dovoljno bi bilo poslati bilo šta što se može heširati, tako da bi i **&str** bilo dovoljno u ovom slučaju, ali za to treba **Borrow**

BORROW I BORROWMUT

- Ovako to izgleda u standardnoj biblioteci:

```
impl<K, V> HashMap<K, V> where K: Eq + Hash
{
    fn get<Q: ?Sized>(&self, key: &Q) -> Option<&V>
        where K: Borrow<Q>,
              Q: Eq + Hash
    { ... }
}
```

- Ovo ovde znači da se može pozajmiti **&Q** kao ključ onoga što se traži, ali i da postoji garancija da će se **&Q** heširati isto kao **Q**
- String implementira **Borrow<str>** i **Borrow<String>**, tako da ovaj kod omogućuje da se prosledi ili **&String** ili **&str** kao ključ
- **Vec<T>** i **[T: N]** implementiraju **Borrow<[T]>**
- Svaki string-like tip dozvoljava pozajmljivanje svog isečka
- Takođe, svaki tip može da pozajmi samog sebe, **T: Borrow<T>**

BORROW I BORROWMUT

- **BorrowMut** zapravo služi za implementaciju promenljivog pozajmljivanja
- Svaki **&mut T** takođe implementira **Borrow<T>** i vreća referencu **&T**
- **BorrowMut** osobina analogna je **Borrow** osobini

```
trait BorrowMut<Borrowed: ?Sized>: Borrow<Borrowed> {
    fn borrow_mut(&mut self) -> &mut Borrowed;
}
```

- Sva ograničenja vezana za **Borrow** važe i kod **BorrowMut**

FROM I INTO

- Osobine **std::convert::From** i **std::convert::Into** reprezentuju konverzije koje konzumiraju vrednost jednog tipa i vraćaju vrednost drugog tipa
- Obe osobine preuzimaju vlasništvo nad argumentom, transformišu ga i predaju vlasništvo pozivaocu
- Definicije su simetrične

```
trait Into<T>: Sized {
    fn into(self) -> T;
}

trait From<T>: Sized {
    fn from(other: T) -> Self;
}
```

- Rust automatski implementira trivijalne transformacije svakog tipa u samog sebe: svaki tip **T** implementira **From<T>** i **Into<T>**

FROM I INTO

- Iako u osnovi predstavljaju implementaciju iste stvari samo u različitim smerovima, u suštini se koriste različito
- **Into** se uobičajeno koristi kako bi se namestilo da funkcije budu fleksibilnije u vrsti argumenata koje primaju, npr.

```
use std::net::Ipv4Addr;
fn ping<A>(address: A) -> std::io::Result<bool>
    where A: Into<Ipv4Addr>
{
    let ipv4_address = address.into();
    ...
}
```

- Što omogućuje da **ping** prima kao argument sve tipove koji implementiraju **Into<Ipv4Addr>**, u osnovi u32 ili [u8;4]
- Sve što **ping** zna o adresi je da ona implementira **Into<Ipv4Addr>** ali nije potrebno navesti tip kod poziva **into**, jer samo jedan tip tu odgovara, te će Rust sam odrediti tu tip

FROM I INTO

- Sada sve ove verzije **ping** funkcije prolaze

```
println!("{:?}", ping(Ipv4Addr::new(23, 21, 68, 141))); // pass an Ipv4Addr
println!("{:?}", ping([66, 146, 219, 98]));             // pass a [u8; 4]
println!("{:?}", ping(0xd076eb94_u32));                 // pass a u32
```

FROM I INTO

- **From** osobina ima drugačiju ulogu, ona se koristi kao generički konstruktor koja će napraviti instancu tipa na osnovu neke druge vrednosti
- Na primer, da Ipv4Addr ne bi imao dve metode, **from_array** i **from_u32**, ono jednostavno implementira **From<[u8;4]>** i **From<u32>**, što omogućava da se napiše sledeće:

```
let addr1 = Ipv4Addr::from([66, 146, 219, 98]);
let addr2 = Ipv4Addr::from(0xd076eb94_u32);
```

- Na osnovu tipa argumenta (**type inference**) se odlučuje koja će se implementacija pozivati
- Uz adekvatnu implementaciju **From** osobine, standardna Rust biblioteka automatski generiše **Into** osobinu

FROM I INTO

- Pošto **from** i **into** metode preuzimaju vlasništvo nad svojim argumentima, sama konverzija može iskoristiti resurse preuzetih argumenata kako bi konstruisale konvertovanu vrednost
- Npr.

```
let text = "Beautiful Soup".to_string();
let bytes: Vec<u8> = text.into();
```

- Implementacija **Into<Vec<u8>>** metode u **String** tipu, jednostavno uzima bafer iz stringa i menja mu namenu, i tako promenjene namene ga vraća kao bafer koji sadrži elemente vektora
- Ovo je način da se neki ograničeni tip pretvori u neki fleksibilniji tip a da se pri tome ne umanjuju ograničenja osnovnog tipa (**text** više nije inicijalizovan posle pomeranja, tako da to što mi radimo sa baferom više nema veze sa samim tekstom)

FROM I INTO

- Za razliku od **AsRef**/**AsMut**, **From**/**Into** nije namenjeno jednostavnim konverzijama, već omogućuju da raditi unutar njih i kompleksne, zahtevne operacije
- Na primer, **std::collections::BinaryHeap<T>** implementira **From<Vec<T>>**, koji poredi i preuređuje redosled elemenata kolekcije u skladu sa zahtevima algoritma kojeg implementira
- **From** i **Into** se koriste u situcijama kada se očekuje da konverzija uvek uspe

TRYFROM I TRYINTO

- **TryFrom** i **TryInto** osobine se koriste u situacijama kada konverzija može da ne uspe
- Tako na primer, Rust ne implementira **From<i64>** za **i32**, već **TryFrom<i64>** (koji je problem sa konverzijom iz i64 u i32?)
- Kao i kod **From/Into**, tako i kod **TryFrom/TryInto**, ako ste implementirali **TryFrom**, Rust može automatski da generiše implementaciju za **TryInto**
- Definicija je samo malo kompleksnija nego za **From/Into**

```
pub trait TryFrom<T>: Sized {
    type Error;
    fn try_from(value: T) -> Result<Self, Self::Error>;
}

pub trait TryInto<T>: Sized {
    type Error;
    fn try_into(self) -> Result<T, Self::Error>;
}
```

TRYFROM I TRYINTO

- Kao što se može videti, **try_into** metoda vraća **Result** tako da možemo odlučiti šta da radimo kada konverzija ne uspe
- U slučaju konverzije većeg broja u manji, nešto veće od i32 u i32:

```
use std::convert::TryInto;
// Saturate on overflow, rather than wrapping
let smaller: i32 = huge.try_into().unwrap_or(i32::MAX);
```

- Slično je i za slučaj kada mora da se rukuje konverzijom negativnih brojeva

```
let smaller: i32 = huge.try_into().unwrap_or_else(|_|{
    if huge >= 0 {
        i32::MAX
    } else {
        i32::MIN
    }
});
```

TRYFROM I TRYINTO

- Koliko će biti kompleksna poruka o grešci, tj. **Error** tip, zavisi od konkretne implementacije
- Standardna biblioteka koristi praznu strukturu
- Konverzije kompleksnijih tipova zahteva više informacije:

```
impl TryInto<LinearShift> for Transform {
    type Error = TransformError;

    fn try_into(self) -> Result<LinearShift, Self::Error> {
        if !self.normalized() {
            return Err(TransformError::NotNormalized);
        }
        ...
    }
}
```

- U osnovi **TryFrom/TryInto** predstavlja kompleksniju verziju **From/Into**

TOOWNED

- **std::borrow::ToOwned** osobina omogućuje da se referenca pretvori u vrednosti koja je u vlasništvu (**owned value**)

```
trait ToOwned {
    type Owned: Borrow<Self>;
    fn to_owned(&self) -> Self::Owned;
}
```

- Za razliku od **clone**, koja mora da vrati tačno **Self**, **to_owned** može da vrati bilo šta što može da pozajmi **&Self**
- **Owned** tip mora da implementira **Borrow<Self>**
- Može se pozajmiti **&[T]** iz **Vec<T>**, tako da **[T]** može implementirati **ToOwned<Owned=Vec<T>>**, sve dok **T** implementira **Clone**, tako da se elementi isečka mogu kopirati u vektor