



*Dr Dinu Dragan*



# PARALELNE I DISTRIBUIRANE ARHITEKTURE I JEZICI (ČAS 2)

# ŠTA RADIMO DANAS?



*Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici*

*O NASTAV*

- Ključne reči (rečnik jednog programskog jezika)
- Rust sintaksa
- Promenljive
- Tipovi podataka

## KLJUČNE REČI

# KLJUČNE REČI RUST PROGRAMSKOG JEZIKA



*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

- Rust ima ključne reči koje su rezervisane isključivo za sam Rust jezik, tj. ne mogu se koristiti za pravljenje identifikatora
- Šta su identifikatori?
- Zašto su oni važni?
- Koja su pravila za rad sa identifikatorima?
- Ako se naziv promenljive sastoji iz više reči, u Rustu su reči u identifikatoru odvojene donjom crtom  
**moja\_promenljiva**
- Rust ključne reči su rezervisane za trenutnu upotrebu ili za buduću upotrebu (?)



# KLJUČNE REČI TRENUTNO U UPOTREBI

*Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici*

<b>as</b>	perform primitive casting, disambiguate the specific trait containing an item, or rename items in use statements
<b>async</b>	return a Future instead of blocking the current thread
<b>await</b>	suspend execution until the result of a Future is ready
<b>break</b>	exit a loop immediatel
<b>const</b>	define constant items or constant raw pointers
<b>continue</b>	continue to the next loop iteration
<b>crate</b>	in a module path, refers to the crate root
<b>dyn</b>	dynamic dispatch to a trait object
<b>else</b>	fallback for if and if let control flow constructs
<b>enum</b>	define an enumeration
<b>extern</b>	link an external function or variable

# KLJUČNE REČI TRENUTNO U UPOTREBI



*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

<b>false</b>	Boolean false literal
<b>fn</b>	define a function or the function pointer type
<b>for</b>	loop over items from an iterator, implement a trait, or specify a higher-ranked lifetime
<b>if</b>	branch based on the result of a conditional expression
<b>impl</b>	implement inherent or trait functionality
<b>in</b>	part of for loop syntax
<b>let</b>	bind a variable
<b>loop</b>	loop unconditionally
<b>match</b>	match a value to patterns
<b>mod</b>	define a module
<b>move</b>	make a closure take ownership of all its captures



# KLJUČNE REČI TRENUTNO U UPOTREBI

*Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici*

<b>mut</b>	denote mutability in references, raw pointers, or pattern bindings
<b>pub</b>	denote public visibility in struct fields, impl blocks, or modules
<b>ref</b>	bind by reference
<b>return</b>	return from function
<b>Self</b>	a type alias for the type we are defining or implementing
<b>self</b>	method subject or current module
<b>static</b>	global variable or lifetime lasting the entire program execution
<b>struct</b>	define a structure
<b>super</b>	parent module of the current module
<b>trait</b>	define a trait
<b>true</b>	Boolean true literal



# KLJUČNE REČI TRENUTNO U UPOTREBI

*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

<b>type</b>	define a type alias or associated type
<b>union</b>	define a union; is only a keyword when used in a union declaration
<b>unsafe</b>	denote unsafe code, functions, traits, or implementations
<b>use</b>	bring symbols into scope
<b>where</b>	denote clauses that constrain a type
<b>while</b>	loop conditionally based on the result of an expression





# KLJUČNE REČI ZA BUDUĆU UPOTREBU

*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

- **abstract**
- **become**
- **box**
- **do**
- **final**
- **macro**
- **override**
- **priv**
- **try**
- **typeof**
- **unsized**
- **virtual**
- **yield**
- Ostavljeno za buduće upotrebe ako ikada zatreba ...



# RAW IDENTIFIERS (SIROVI IDENTIFIKATORI)

*Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici*

- Raw identifikatori su deo sintakse koja vam omogućava da koristite ključne reči tamo gde one inače ne bi bile dozvoljene
- Raw identifikator se koristi tako što se na ključnu reč doda prefiks **r#**
- Na primer, ako se želi napraviti funkcija koja se zove **match**, to nije moguće, jer je **match** ključna reč u programsku jeziku Rust
- Može se koristiti ako se ispred naziva doda raw identifikator, **r#match**, jedino što se i u definiciji identifikatora i u pozivu mora koristiti **r#**
- Raw identifikatori omogućavaju da se koristi bilo koja reč za identifikator, čak i ako je ta reč rezervisana ključna reč
- Ovo daje više slobode i omogućava integraciju sa programima napisanim na jeziku gde ove reči nisu ključne reči
- Omogućuju da se koriste biblioteke napisane u drugačijem izdanju Rusta (try nije ključna reč u Rust verziji iz 2015. godine, ali jeste u verziji iz 2018. godine)

## REČNIK RUST SINTAKSE



- Kao i svi programski jezici, Rust ima složenu sintaksu čiji rečnik uključuje operatore i druge simbole koji se pojavljuju sami ili u kontekstu putanja, generičkih karakteristika, granica osobina, makroa, atributa, komentara, torki i zagrada
- Tabela B-1 iz Rust knjige sadrži listu svih operatora u Rust programskom jeziku, primer kako se operator koristi, kratko objašnjenje i da li je taj operator može da se overloduje
- Ako operator može da se overloduje, navedena je relevantna osobina koja se koristi za overloading tog operatora

# OPERATORI



*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

Operator	Example	Explanation	Overloadable?
!	ident!(...), ident!{...}, ident![...]	Macro expansion	
!	!expr	Bitwise or logical complement	Not
!=	expr != expr	Nonequality comparison	PartialEq
%	expr % expr	Arithmetic remainder	Rem
%=	var %= expr	Arithmetic remainder and assignment	RemAssign
&	&expr, &mut expr	Borrow	
&	&type, &mut type, &'a type, &'a mut type	Borrowed pointer type	
&	expr & expr	Bitwise AND	BitAnd
&=	var &= expr	Bitwise AND and assignment	BitAndAssign
&&	expr && expr	Short-circuiting logical AND	
*	expr * expr	Arithmetic multiplication	Mul

# OPERATORI



*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

<code>*=</code>	<code>var *= expr</code>	Arithmetic multiplication and assignment	MulAssign
<code>*</code>	<code>*expr</code>	Dereference	Deref
<code>*</code>	<code>*const type, *mut type</code>	Raw pointer	
<code>+</code>	<code>trait + trait, 'a + trait</code>	Compound type constraint	
<code>+</code>	<code>expr + expr</code>	Arithmetic addition	Add
<code>+=</code>	<code>var += expr</code>	Arithmetic addition and assignment	AddAssign
<code>,</code>	<code>expr, expr</code>	Argument and element separator	
<code>-</code>	<code>- expr</code>	Arithmetic negation	Neg
<code>-</code>	<code>expr - expr</code>	Arithmetic subtraction	Sub
<code>-=</code>	<code>var -= expr</code>	Arithmetic subtraction and assignment	SubAssign
<code>-&gt;</code>	<code>fn(...) -&gt; type, [...]  -&gt; type</code>	Function and closure return type	
<code>.</code>	<code>expr.ident</code>	Member access	

# OPERATORI



*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

<code>..</code>	<code>.., expr.., ..expr,</code> <code>expr..expr</code>	Right-exclusive range literal	PartialOrd
<code>..=</code>	<code>..=expr,</code> <code>expr..=expr</code>	Right-inclusive range literal	PartialOrd
<code>..</code>	<code>..expr</code>	Struct literal update syntax	
<code>..</code>	<code>variant(x, ..),</code> <code>struct_type { x,</code> <code>.. }</code>	"And the rest" pattern binding	
<code>...</code>	<code>expr...expr</code>	(Deprecated, use <code>..=</code> instead) In a pattern: inclusive range pattern	
<code>/</code>	<code>expr / expr</code>	Arithmetic division	Div
<code>/=</code>	<code>var /= expr</code>	Arithmetic division and assignment	DivAssign

# OPERATORI



*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

:	pat: type, ident: type	Constraints	
:	ident: expr	Struct field initializer	
:	'a: loop {...}	Loop label	
;	expr;	Statement and item terminator	
;	[...; len]	Part of fixed-size array syntax	
<<	expr << expr	Left-shift	Shl
<<=	var <<= expr	Left-shift and assignment	ShlAssign
=	var = expr, ident = type	Assignment/equivalence	
==	expr == expr	Equality comparison	PartialEq
=>	pat => expr	Part of match arm syntax	
>	expr > expr	Greater than comparison	PartialOrd
>=	expr >= expr	Greater than or equal to comparison	PartialOrd
>>	expr >> expr	Right-shift	Shr
>>=	var >>= expr	Right-shift and assignment	ShrAssign



# OPERATORI



*Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici*

@	ident @ pat	Pattern binding	
^	expr ^ expr	Bitwise exclusive OR	BitXor
^=	var ^= expr	Bitwise exclusive OR and assignment	BitXorAssign
	pat   pat	Pattern alternatives	
	expr   expr	Bitwise OR	BitOr
=	var  = expr	Bitwise OR and assignment	BitOrAssign
	expr    expr	Short-circuiting logical OR	
?	expr?	Error propagation	



# NEOPERATORSKI SIMBOLI

*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

Symbol	Explanation
'ident	Named lifetime or loop label
...u8, ...i32, ...f64, ...usize, etc.	Numeric literal of specific type
"..."	String literal
r"...", r#"..."#, r##"..."##, etc.	Raw string literal, escape characters not processed
b"..."	Byte string literal; constructs an array of bytes instead of a string
br"...", br#"..."#, br##"..."##, etc.	Raw byte string literal, combination of raw and byte string literal
'...'	Character literal
b'...'	ASCII byte literal
...  expr	Closure
!	Always empty bottom type for diverging functions
-	"Ignored" pattern binding; also used to make integer literals readable

# SIMBOLI ZA RAD SA PUTANJAMA



*Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici*

Symbol	Explanation
<code>ident::ident</code>	Namespace path
<code>::path</code>	Path relative to the crate root (i.e., an explicitly absolute path)
<code>self::path</code>	Path relative to the current module (i.e., an explicitly relative path).
<code>super::path</code>	Path relative to the parent of the current module
<code>type::ident, &lt;type as trait&gt;::ident</code>	Associated constants, functions, and types
<code>&lt;type&gt;::....</code>	Associated item for a type that cannot be directly named (e.g., <code>&lt;&amp;T&gt;::....</code> , <code>&lt;[T]&gt;::....</code> , etc.)
<code>trait::method(...)</code>	Disambiguating a method call by naming the trait that defines it
<code>type::method(...)</code>	Disambiguating a method call by naming the type for which it's defined
<code>&lt;type as trait&gt;::method(...)</code>	Disambiguating a method call by naming the trait and type

# GENERIČKI SIMBOLI



*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

Symbol	Explanation
<code>path&lt;...&gt;</code>	Specifies parameters to generic type in a type (e.g., <code>Vec&lt;u8&gt;</code> )
<code>path::&lt;...&gt;</code> , <code>method::&lt;...&gt;</code>	Specifies parameters to generic type, function, or method in an expression; often referred to as turbofish (e.g., <code>"42".parse::&lt;i32&gt;()</code> )
<code>fn ident&lt;...&gt; ...</code>	Define generic function
<code>struct ident&lt;...&gt;</code> <code>...</code>	Define generic structure
<code>enum ident&lt;...&gt; ...</code>	Define generic enumeration
<code>impl&lt;...&gt; ...</code>	Define generic implementation
<code>for&lt;...&gt; type</code>	Higher-ranked lifetime bounds
<code>type&lt;ident=type&gt;</code>	A generic type where one or more associated types have specific assignments (e.g., <code>Iterator&lt;Item=T&gt;</code> )



# CONSTRAIN SIMBOLI

*Dragan de Dinnu - Paralelne i distribuirane arhitekture i jezici*

Symbol	Explanation
<code>T: U</code>	Generic parameter T constrained to types that implement U
<code>T: 'a</code>	Generic type T must outlive lifetime 'a (meaning the type cannot transitively contain any references with lifetimes shorter than 'a)
<code>T: 'static</code>	Generic type T contains no borrowed references other than 'static ones
<code>'b: 'a</code>	Generic lifetime 'b must outlive lifetime 'a
<code>T: ?Sized</code>	Allow generic type parameter to be a dynamically sized type
<code>'a + trait, trait + trait</code>	Compound type constraint



# SIMBOLI ZA RAD SA MAKROIMA

*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

Symbol	Explanation
#[meta]	Outer attribute
#![meta]	Inner attribute
\$ident	Macro substitution
\$ident:kind	Macro capture
\$(...)...	Macro repetition
ident!(...), ident!{...}, ident![...]	Macro invocation



# SIMBOLI ZA KOMENTARE

*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

Symbol	Explanation
//	Line comment
//!	Inner line doc comment
///	Outer line doc comment
/*...*/	Block comment
/*!...*/	Inner block doc comment
/**...*/	Outer block doc comment



# SIMBOLI ZA RAD SA SLOGOVIMA (TUPLE)

*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

Symbol	Explanation
()	Empty tuple (aka unit), both literal and type
(expr)	Parenthesized expression
(expr,)	Single-element tuple expression
(type,)	Single-element tuple type
(expr, ...)	Tuple expression
(type, ...)	Tuple type
expr(expr, ...)	Function call expression; also used to initialize tuple structs and tuple enum variants
expr.0, expr.1, etc.	Tuple indexing



# SIMBOLI VITIČASTIH ZAGRADA



*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

Context	Explanation
{...}	Block expression
Type {...}	struct literal

# SIMBOLI UGLASTIH ZAGRADA



*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

Context	Explanation
<code>[...]</code>	Array literal
<code>[expr; len]</code>	Array literal containing <code>len</code> copies of <code>expr</code>
<code>[type; len]</code>	Array type containing <code>len</code> instances of <code>type</code>
<code>expr[expr]</code>	Collection indexing. Overloadable ( <code>Index</code> , <code>IndexMut</code> )
<code>expr[..]</code> , <code>expr[a..]</code> , <code>expr[..b]</code> , <code>expr[a..b]</code>	Collection indexing pretending to be collection slicing, using <code>Range</code> , <code>RangeFrom</code> , <code>RangeTo</code> , or <code>RangeFull</code> as the “index”

## PROMENLJIVE



**let** **<ime\_p>**[:tip] [=vrednost];

- Podrazumevano je da je promenljiva u Rustu nepromenljive, **immutable**, vrednosti

**let x = 5;**    ili    **let x:i32 = 5;**    ili    **let x;**    ili    **let x:i32;**

- Šta se dešava sa **let x;** (**kako nepromenljivo a nema vredost?**)
- Zašto odmah **immutable** promenljive?
- Dozvoljeno je i:

**let x; ... x = 5;**

- Kako zna koji je tip?
- Da bi promenljiva od starta bila promenljive vrednosti potrebno je koristiti naredbu **mut**

**let mut x = 5;**

# IMMUTABLE VS. MUTABLE

*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

- Nepromenljiva promenljiva, molim?
- Koje su prednosti nepromenljivih promenljivi (osim užasa)?
  - Sigurnost – ne može doći do slučajne ili pogrešne promene vrednosti promenljive
  - Konkurentnost – mogu se bezbedno deliti između više niti, bez potrebe za sinhronizacijom ili zaključavanjem
  - Performanse – omogućuju Rust kompajleru određene oblike optimizacije, kao što je propagacija konstanti (šta je to?)
- U osnovi, Rust je obrnuo C++ mantru da sve što ne menja vrednost treba da bude konstanta!
- U suštini, to znači da sada moramo dobro promisliti o prirodi promenljive koje unosimo u kod



- Šta je to?
- Identifikatori kojima se dodeljuje vrednost koja se ne može menjati
- Kada i zašto koristimo konstante?
- Ne može se dodati **mut** nikada
- Umesto **let** koristi se **const**
- Mora se eksplicitno navesti kog je tipa konstanta

**CONST X:i32 = 5;**

- Može i rezultat izraza, pod uslovom da je vrednost konstantna i odgovarajućeg tipa:

**const SEKUNDE\_U\_SATU:u32 = 60 \* 60;**

- Konstante važe sve vreme tokom izvršavanja programa, u okviru opsega u kome su deklarisanе



# PREKLAPANJE PROMENLJIVE (SHADOWING)

*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

- Mogućnost da se napravi nova promenljiva sa istim imenom (ali drugačijim tipom ako to želimo)
- Svaki novi **let** sa istim identifikatorom stvara novu promenljivu tj. zauzima novu memoriju
- U Rust terminologiji to se naziva **shadowing** i kaže se da je prva promenljiva preklopljena ili **shadowed**

```
fn main() {  
    let x = 5;  
  
    let x = x + 1;  
  
    {  
        let x = x * 2;  
        println!("The value of x in the inner scope is: {x}");  
    }  
  
    println!("The value of x is: {x}");  
}
```



# PREKLAPANJE PROMENLJIVE (SHADOWING)

*Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici*

- Kada se desi preklapanje, svaki put kada se upotrebi naziv promenljive, koristi se promenljiva koja je poslednja definisana
- Preklapanje prestaje kada promenljiva sa kojom se vrši preklapanje izađe van opsega
- Nije isto što i **mut**, jer je pored promene vrednosti moguće da se prilikom preklapanja promeni tip promenljive

```
fn main() {  
    let spaces = "  ";  
    let spaces = spaces.len();  
}
```

```
fn main() {  
    let mut spaces = "  ";  
    spaces = spaces.len();  
}
```

**Error**

- Prvo imamo string, posle imamo ceo broj
- Zašto je ovo uopšte moguće? Koji mehanizam stoji iza svega?
- Ipak, preklapanje promenljivih na ovaj način može biti opasno!
- Zašto?





- Šta su to tipovi podataka? Šta nam oni govore?
- Rust jezik je u velikoj meri dizajniran oko svojih tipova.
- Njegova podrška za kod visokih performansi proizilazi iz toga što dozvoljava programerima da izaberu prikaz podataka koji najbolje odgovara situaciji, uz pravi balans između jednostavnosti i cene
- Rust-ova garancija bezbednosti u radu sa memorijom i nitima izvršavanja počiva na ispravnosti njegovih tipova, a fleksibilnost proizilazi iz njegovih generičkih tipova i osobina
- Ovi tipovi na nivou koda imaju konkretne parnjake na nivou računara sa predvidljivim troškovima i performansama.
- Iako Rust ne garantuje da će se tipovi direktno mapirati, vodi se računa da su odstupanja minimalna i samo onda kada to predstavlja pouzdano poboljšanje



- Kako Rust ima striktan statički sistem tipova podataka, moraju se navesti tipovi argumenata funkcije i povratne vrednosti, polja strukture i nekoliko drugih konstrukcija
  - Dve karakteristike Rusta čine njegovu striktnost manje problematičnom za kodiranje nego što bi se očekivalo
1. Rust ima mogućnost da sam dedukuje koji tip se koristi (**type inference**)
    - U praksi, često postoji samo jedan tip koji će raditi za datu promenljivu ili izraz
    - Rust omogućava da se izostavi (**elide**) ili izbací tip u većini slučajeva

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

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



## 1. Rust ima mogućnost da sam dedukuje koji tip se koristi (**type inference**)

- Kada je neophodno navesti tip, tj. Rust ne može da ga sam dedukuje, onda se on poprilično jasno buni

```
$ cargo build
  Compiling no_type_annotations v0.1.0 (file:///projects/no_type_annotations)
error[E0282]: type annotations needed
  --> src/main.rs:2:9
   |
2 |   let guess = "42".parse().expect("Not a number!");
   |   ^^^^^ consider giving `guess` a type
```

For more information about this error, try `rustc --explain E0282`.  
error: could not compile `no\_type\_annotations` due to previous error

- Navođenje tipa u tom slučaju rešava problem

```
#![allow(unused)]
fn main() {
let guess: u32 = "42".parse().expect("Not a number!");
}
```



## 2. Funkcije mogu biti generičke

- jedna funkcija može raditi na vrednostima mnogo različitih tipova
  - funkcija može da radi na bilo kojoj vrednosti koja ima svojstva i metode koje će funkciji trebati (**duck typing**)
  - upravo ova fleksibilnost otežava jezicima sa dinamičkim tipovima da rano otkriju greške; 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 fleksibilnosti, generičke funkcije su jednako efikasne kao i njihove negeneričke verzije; nema inherentne prednosti u performansama pisanja, recimo, specifične funkcije sume za svaki ceo broj u odnosu na pisanje generičke koja rukuje svim celim brojevima
- Rust tipovi podataka mogu se podeliti na skalarne i kompleksne (**compound**)

## SKALARI

# SKALARNI TIPOVI



*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

- Rust ima četiri osnovna skalarna tipa: cele brojeve (integer), brojeve sa decimalnim zarezom (floating-point numbers), karaktere (character) i logički tip (boolean)

Type	Description	Values
<code>i8, i16, i32, i64, i128</code> <code>u8, u16, u32, u64, u128</code>	Signed and unsigned integers, of given bit width	42, -5i8, 0x400u16, 0o100i16, 20_922_789_888_000u64, b'*' (u8 byte literal)
<code>isize, usize</code>	Signed and unsigned integers, the same size as an address on the machine (32 or 64 bits)	137, -0b0101_0010isize, 0xffff_fc00usize
<code>f32, f64</code>	IEEE floating-point numbers, single and double precision	1.61803, 3.14f32, 6.0221e23f64
<code>bool</code>	Boolean	true, false
<code>char</code>	Unicode character, 32 bits wide	'*', '\n', '字', '\x7f', '\u{CA0}'

# SKALARNI TIPOVI



*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

- Osnova Rustovog sistema tipova je kolekcija numeričkih tipova fiksne širine, izabranih da odgovaraju tipovima koje skoro svi savremeni procesori implementiraju (implementirano direktno u hardveru)
- Numerički tipovi fiksne širine mogu da izađu izvan opsega ili da dovedu do gubitka preciznosti
- Numerički tipovi fiksne širine su adekvatni za većinu aplikacija i mogu biti hiljadama puta brži od reprezentacija kao što su celi brojevi proizvoljne preciznosti i tačni izrazi (exact rationals)

- Ako su vam potrebne takve vrste numeričkih reprezentacija, one su podržane u **num** Cargo sanduku (paketu)

Size (bits)	Unsigned integer	Signed integer	Floating-point
8	u8	i8	
16	u16	i16	
32	u32	i32	f32
64	u64	i64	f64
128	u128	i128	
Machine word	usize	isize	



# INTEGER TIPOVI

*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

- Rust ima označene i neoznačene brojeve

Type	Range
u8	0 to $2^8-1$ (0 to 255)
u16	0 to $2^{16}-1$ (0 to 65,535)
u32	0 to $2^{32}-1$ (0 to 4,294,967,295)
u64	0 to $2^{64}-1$ (0 to 18,446,744,073,709,551,615, or 18 quintillion)
u128	0 to $2^{128}-1$ (0 to around $3.4 \times 10^{38}$ )
usize	0 to either $2^{32}-1$ or $2^{64}-1$

Type	Range
i8	$-2^7$ to $2^7-1$ (-128 to 127)
i16	$-2^{15}$ to $2^{15}-1$ (-32,768 to 32,767)
i32	$-2^{31}$ to $2^{31}-1$ (-2,147,483,648 to 2,147,483,647)
i64	$-2^{63}$ to $2^{63}-1$ (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
i128	$-2^{127}$ to $2^{127}-1$ (roughly $-1.7 \times 10^{38}$ to $+1.7 \times 10^{38}$ )
isize	Either $-2^{31}$ to $2^{31}-1$ , or $-2^{63}$ to $2^{63}-1$





# INTEGER TIPOVI

*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

- **u8** se koristi za rad sa bajtovima
- **char** u Rustu nije **u8**, niti **u32** (zašto uopšte razmatramo u32?)
- **usize** i **isize** odgovaraju **size\_t** i **ptrdiff\_t** iz C/C++
  - Njihova preciznost odgovara veličini adresnog prostora na računaru na kojem se koriste: dugački su 32 bita na 32-bitnim arhitekturama i 64 bita na 64-bitnim arhitekturama
  - **usize** ne može biti negativna i generalno se koristi za memorijske adrese, pozicije, indekse, dužine (ili veličine!)
  - **isize** veličina može biti negativna i generalno se koristi za pomeranja adresa, pozicija, indeksa ili dužina
  - Rust zahteva da indeksi niza budu **usize** vrednosti (zašto?)
  - Vrednosti koje predstavljaju veličine nizova/vektora ili koje broje koliko elemenata ima u nekoj strukturi podataka takođe generalno imaju tip **usize**
- Integer literali u Rustu mogu imati sufiks koji označava njihov tip
  - **42u8**, **42i64**, **1729isize**, ... (čemu to služi?)



# INTEGER TIPOVI

*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

- Integer literali u Rustu mogu imati prefiks (čemu to služi?)

Number literals	Example
Decimal	98_222
Hex	0xff
Octal	0o77
Binary	0b1111_0000
Byte (u8 only)	b'A'

- Da bi se dugački brojevi učinili čitljivijim, mogu se umetnuti donje crte među cifre
  - 4\_294\_967\_295 (isto što i 4294967295)
- Konverzija iz jednog tipa u drugi (pa i iz integera u integer) se radi sa operatorom **as** (zašto? kako je u drugim progr. jezicima?)
  - **10\_i8 as u16, 2525\_u16 as i16, 65535\_u16 as i32**
- Standardna biblioteka ima metode za rad integerom (**pow**, **abs**, ...)



# PREKORAČENJE OPSEGA

*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

- Kada dođe do prekoračenja opsega, Rust reaguje
- U debug bildu dolazi do nečega što se zove **panika!** tj. Rust kompajler krene da paniči (to nije jedini slučaj kad Rust kompajler paniči u debug režimu ...)

```
let mut i = 1;
loop {
    i *= 10; // panic: attempt to multiply with overflow
             // (but only in debug builds!)
}
```

- U riliz (release) bildu, Rust radi vraping (**wrapping**) korišćenjem tehnike potpunog komplementa. Šta je to?
- Program neće paničiti, ali će promenljiva imati neočekivanu vrednost
- Oslanjanje na vreping celog broja smatra se greškom

# PREKORAČENJE OPSEGA



*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

- Rust ima čitav mehanizam za rukovanjem prekoračenjima, tako što se umesto poziva običnih operatora pozivaju metode koje odgovaraju običnim operatorima, ali koje vrše provere prekoračenja opsega i onda vraćaju različite rezultate u zavisnosti sa svojom prirodom
- To je familija sledećih metoda

Operation	Name suffix	Example
Addition	add	<code>100_i8.checked_add(27) == Some(127)</code>
Subtraction	sub	<code>10_u8.checked_sub(11) == None</code>
Multiplication	mul	<code>128_u8.saturating_mul(3) == 255</code>
Division	div	<code>64_u16.wrapping_div(8) == 8</code>
Remainder	rem	<code>(-32768_i16).wrapping_rem(-1) == 0</code>
Negation	neg	<code>(-128_i8).checked_neg() == None</code>
Absolute value	abs	<code>(-32768_i16).wrapping_abs() == -32768</code>
Exponentiation	pow	<code>3_u8.checked_pow(4) == Some(81)</code>
Bitwise left shift	shl	<code>10_u32.wrapping_shl(34) == 40</code>
Bitwise right shift	shr	<code>40_u64.wrapping_shr(66) == 10</code>



# PREKORAČENJE OPSEGA

*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

- Metode za upravljanje prekoračenjem mogu se podeliti u 4 grupe
- **Checked** operacije vraćaju rezultat tipa **Option** (malo kasnije više reči o tome), gde je **Some(v)** vrednost koja je matematički tačna (ako nije došlo do prekoračenja) ili **None** ako je došlo do prekoračenja i ne može se napisati adekvatna vrednost

```
// The sum of 10 and 20 can be represented as a u8.  
assert_eq!(10_u8.checked_add(20), Some(30));
```

```
// Unfortunately, the sum of 100 and 200 cannot.  
assert_eq!(100_u8.checked_add(200), None);
```

```
// Do the addition; panic if it overflows.  
let sum = x.checked_add(y).unwrap();
```

```
// Oddly, signed division can overflow too, in one particular case.  
// A signed n-bit type can represent -2n-1, but not 2n-1.  
assert_eq!((-128_i8).checked_div(-1), None);
```



# PREKORAČENJE OPSEGA

*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

- **Wrapping** operacije vraćaju tačan rezultat ili obmotan rezultat primenom mehanizma potpunog komplementa
- Ovo obezbeđuje da se metoda ponaša na isti način i u debug i u riliz bildu

```
// The first product can be represented as a u16;  
// the second cannot, so we get 250000 modulo 216.  
assert_eq!(100_u16.wrapping_mul(200), 20000);  
assert_eq!(500_u16.wrapping_mul(500), 53392);
```

```
// Operations on signed types may wrap to negative values.  
assert_eq!(500_i16.wrapping_mul(500), -12144);
```

```
// In bitwise shift operations, the shift distance  
// is wrapped to fall within the size of the value.  
// So a shift of 17 bits in a 16-bit type is a shift  
// of 1.  
assert_eq!(5_i16.wrapping_shl(17), 10);
```

# PREKORAČENJE OPSEGA



*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

- **Saturating** operacije vraćaju rezultat najbliži tačnoj vrednosti, tj. ako ne dođe do prekoračenja, vraćaju tačan rezultat, a ako dođe do prekoračenja, vraća najveću moguću vrednost za dati tip
- Ne postoje verzije ovih metoda za deljenje (**div**), ostatak (**rem**) ili pomeranje (**shl** i **shr**)

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

- Zašto ne postoje?



- **Overflowing** operacije vraćaju uređen par (**tuple**) (**result**, **overflowed**)  
gde **result** sadrži wrapping verziju rezultata, a **overflowes** sadrži logičku vrednost (**boolean**) koji kaže da li je došlo do prekoračanja ili ne

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

- **overflowing\_shl** i **overflowing\_shr** malo odstupaju od šablona: oni vraćaju **true** za **overflowed** samo ako je rastojanje pomeranja bilo veliko kao ili veće od širine samog tipa u bitima

```
// A shift of 17 bits is too large for `u16`, and 17 modulo 16 is 1.  
assert_eq!(5_u16.overflowing_shl(17), (10, true));
```





# TIPOVI SA POKRETNIM ZAREZOM

*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

- Rust ima dva primitivna tipa za brojeve sa pokretnim zarezom
- To su f32 i f64, koji su veličine 32 bita i 64 bita, respektivno
- Podrazumevani tip je f64, jer je na modernim računarima otprilike iste brzine kao f32, ali je sposoban za veću preciznost

Type	Precision	Range
f32	IEEE single precision (at least 6 decimal digits)	Roughly $-3.4 \times 10^{38}$ to $+3.4 \times 10^{38}$
f64	IEEE double precision (at least 15 decimal digits)	Roughly $-1.8 \times 10^{308}$ to $+1.8 \times 10^{308}$

- Svi tipovi sa pokretnim zarezom su označeni (**signed**)
- Tipovi sa pokretnim zarezom su reprezentovani u skladu sa IEEE-754 standard

fractional part      type suffix  
31415.926e-4f64  
integer part      exponent



# TIPOVI SA POKRETNIM ZAREZOM

*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

- Ako se ne navede literal za tip sa pokretnim zarezom, Rust sam pokušava da ga odredi
- Rust tretira celobrojne tipove i tipove sa pokretnim zarezom kao odvojene klase (neće ih pobrkati)
- Rust gotovo da nema implicitnih konverzija (voditi računa o poklapanju tipova – gde to imamo?)
- Nedostatak implicitne konverzije čini Rust malo rogobatnim za pisanje (ali je pokazano da implicitne konverzija često dovodi do grešaka i sigurnosnih rupa/propusta)
- Podsetnik, eksplicitne konverzije se realizuju primenom operatora **as**  
**i as f64; x as i32**

# BOOLEAN



*Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici*

- Rust je veoma striktan po pitanju **boolean** vrednosti: kontrolne strukture kao što su **if** i **while** zahtevaju da njihovi uslovi budu **boolean** izrazi, kao i logički operatori kratkog spoja **&&** i **||**
- Mora **if x != 0 { ... }**, ne može **if x { ... }**
- **true** ili **false**
- Boolean tip se može konvertovati u integer

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

- Obrnuto ne važi
- **Boolean** je reprezentovan sa jednim bajtom, pa se može napraviti pokazivač na njega



- Rust **char** tip predstavlja jedan Unicode karakter veličine 32 bita
- Rust koristi tip **char** za pojedinačne izolovane znakove, ali koristi **UTF-8** kodiranje za stringove i tokove teksta
- String predstavlja svoj tekst kao niz UTF-8 bajtova, a ne kao **char** niz
- **'8'**, **'!'**, **'鑄'**

Character	Rust character literal
Single quote, '	'\''
Backslash, \	'\\'
Newline	'\n'
Carriage return	'\r'
Tab	'\t'

- Karakter se može zapisati i u heksadecimalnom i decimalnom kodu  
**'\*' isto što '\x2A' ili '\u{42}'**



- Rust nikada ne pretvara implicitno char u bilo koji drugi tip
- Operator **as** se može koristiti za pretvaranje iz **char** tipa u **int** tip

```
assert_eq!('*' as i32, 42);  
assert_eq!('ð' as u16, 0xca0);  
assert_eq!('ð' as i8, -0x60); // U+0CA0 truncated to eight bits, signed
```

- Standardna biblioteka poseduje korisne metode za rad sa karakterima

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



## **VIŠEDIMENZIONALNI TIPOVI PODATAKA**



# VIŠEDIMENZIONALNI TIPOVI PODATAKA

*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

Type	Description	Values
<code>(char, u8, i32)</code>	Tuple: mixed types allowed	<code>('%', 0x7f, -1)</code>
<code>()</code>	“Unit” (empty tuple)	<code>()</code>
<code>struct S { x: f32, y: f32 }</code>	Named-field struct	<code>S { x: 120.0, y: 209.0 }</code>
<code>struct T (i32, char);</code>	Tuple-like struct	<code>T(120, 'X')</code>
<code>struct E;</code>	Unit-like struct; has no fields	<code>E</code>
<code>enum Attend { OnTime, Late(u32) }</code>	Enumeration, algebraic data type	<code>Attend::Late(5), Attend::OnTime</code>
<code>Box&lt;Attend&gt;</code>	Box: owning pointer to value in heap	<code>Box::new(Late(15))</code>
<code>&amp;i32, &amp;mut i32</code>	Shared and mutable references: non-owning pointers that must not outlive their referent	<code>&amp;s.y, &amp;mut v</code>
<code>String</code>	UTF-8 string, dynamically sized	<code>"ラーメン: ramen".to_string()</code>
<code>&amp;str</code>	Reference to <code>str</code> : non-owning pointer to UTF-8 text	<code>"そば: soba", &amp;s[0..12]</code>
<code>[f64; 4], [u8; 256]</code>	Array, fixed length; elements all of same type	<code>[1.0, 0.0, 0.0, 1.0], [b' '; 256]</code>
<code>Vec&lt;f64&gt;</code>	Vector, varying length; elements all of same type	<code>vec![0.367, 2.718, 7.389]</code>





# KOMPLEKSNI TIPOVI PODATAKA

*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

Type	Description	Values
<code>&amp;[u8], &amp;mut [u8]</code>	Reference to slice: reference to a portion of an array or vector, comprising pointer and length	<code>&amp;v[10..20]</code> , <code>&amp;mut a[..]</code>
<code>Option&lt;&amp;str&gt;</code>	Optional value: either <code>None</code> (absent) or <code>Some(v)</code> (present, with value <code>v</code> )	<code>Some("Dr.")</code> , <code>None</code>
<code>Result&lt;u64, Error&gt;</code>	Result of operation that may fail: either a success value <code>Ok(v)</code> , or an error <code>Err(e)</code>	<code>Ok(4096)</code> , <code>Err(Error::last_os_error())</code>
<code>&amp;dyn Any, &amp;mut dyn Read</code>	Trait object: reference to any value that implements a given set of methods	value as <code>&amp;dyn Any</code> , &mut file as <code>&amp;mut dyn Read</code>
<code>fn(&amp;str) -&gt; bool</code>	Pointer to function	<code>str::is_empty</code>
(Closure types have no written form)	Closure	<code> a, b  { a*a + b*b }</code>



# TUPLES (N-TORKE)

*Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici*

- Tuples predstavlja n-torke vrednosti
- Zapisuje se kao niz vrednosti (različitih tipova) odvojenih zarezom unutar zagrade
- Predstavlja generalan način da se više vrednosti različitog tipa grupiše u jednu celinu

```
fn main() {  
    let tup: (i32, f64, u8) = (500, 6.4, 1);  
}
```

- Promenljiva **tup** predstavlja sve 3 vrednosti
- **Tuple** ima fiksnu veličinu, jednom kad mu se definiše veličina, ne može se menjati (ni smanjivati, ni povećavati)
- Za pristup pojedinačnom elementu n-torke koriste se konstante vrednosti (ne mogu se koristiti iteratori)



# TUPLES (N-TORKE)

*Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici*

- Za pristup pojedinačnom elementu n-torke koriste se konstante vrednosti (ne mogu se koristiti iteratori)

```
fn main() {  
    let x: (i32, f64, u8) = (500, 6.4, 1);  
  
    let five_hundred = x.0;  
  
    let six_point_four = x.1;  
  
    let one = x.2;  
}
```

- Vrednosti n-torke se mogu i razdvojiti upotrebom **pattern matching** mehanizma

```
fn main() {  
    let tup = (500, 6.4, 1);  
  
    let (x, y, z) = tup;  
  
    println!("The value of y is: {y}");  
}
```



# TUPLES (N-TORKE)

*Dragan da Dinu - Paralelne i distribuirane arhitekture i jezici*

- N-torka bez ikakvih vrednosti ima posebno ime, unit
- Ova vrednost i njen odgovarajući tip su predstavljeni kao prazna zagrada () i predstavljaju praznu vrednost ili tip prazne povratne vrednosti
- Izrazi implicitno vraćaju unit vrednost ako ne vraćaju vrednost
- Rust kod često koristi tuple tip za vraćanje više vrednosti iz funkcije

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

- I onda:

```
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");
```

- Tuple tip se pre koristi kako je dato u prethodnom primeru



# TUPLES (N-TORKE)

*Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici*

- Nego:

```
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");
```



- Rust poseduje nekoliko tipova za čuvanje memorijskih adresa
- Čitav Rust je dizajniran oko toga da minimizuje alokaciju memorije, što znači da uglavnom, kad god to ima smisla, gura podatke na stek
- Podaci se predefinisano ugnježdavaju, npr.  $((0, 0), (1440, 900))$  će biti sačuvano kao 4 povezane celobrojne vrednosti
  - Ako se čuvaju u lokalnoj promenljivoj, onda je ona dužine 4 celobrojne vrednost (i32)
  - Ništa ne ide na hip (heap)
- Za razliku od Java ili sličnog jezika (sa GC), upotreba pokazivača je eksplicitna, tj. mora se eksplicitno naglasiti da radimo sa pokazivačima (slično C/C++ jeziku)
- Sa druge strane, upotreba pokazivača je ograničena tako da se obezbedi siguran rad sa memorijom, te je rad sa njima bezbednija



- Reference predstavljaju Rust-ov osnovni pokazivački tip
- **&String** (referenca na string) ili **&i32** (referenca na i32)
- Ako je **r** pokazivač, onda se do referencirane vrednosti dolazi sa **\*r**
- **&** i **\*** u mnogome liče na operatore iz C/C++ programskih jezika; memorija se ne oslobađa automatski kada referenca izađe iz opsega
- U Rustu nema null vrednosti, tj. u safe režimu rada Rusta to je nemoguće
- Takođe, u Rustu se prati vlasništvo nad vrednošću (ne nad referencom) i prati se doseg/život te vrednosti (kad ona prestaje da postoji)
- Stvaranje reference na neku promenljivu dovodi do nečega što se naziva pozajmljivanje (borrow) – ovaj mehanizam je toliko bitan za Rust i njegov mehanizam rada sa memorijom da će biti predmet skoro čitavog jednog predavanja



- Rast referenca može biti nepromenljiva (**immutable**) ili promenljiva (**mutable**)
- **&T** je **immutable**, deljena referenca. Može istovremeno postojati mnogo nepromenljivih zajedničkih referenci na datu vrednost, ali one su samo za čitanje (read-only): modifikovanje vrednosti na koju ukazuju je zabranjeno, kao što je slučaj sa `const T*` u C programskom jeziku
- **&mut T** je **mutable**, ekskluzivna referenca. Mogu se i čitati i menjati vrednosti na koju referenca pokazuje, ali sve dok ta promenljiva referenca postoji, u isto vreme ne može postojati niti jedna druga (promenljiva ili nepromenljiva) referenca na tu vrednost
  - Za vreme postojanja ove reference, vrednosti se može pristupiti samo preko nje
- Rast koristi ovu dihotomiju između deljenih i promenljivih referenci da bi primenio pravilo „single writer or multiple readers “





- Najjednostavniji način da se neka vrednost alokira na hipu je da se koristi **Box::new** sintaksa

```
let t = (12, "eggs");  
let b = Box::new(t); // allocate a tuple in the heap
```

- Tip promenljive **t** je (**i32, &str**), tako da će tip promenljive **b** biti **Box<(i32, &str)>**
- Poziv **Box::new** alokira dovoljno memorije da se tuple smesti na hip
- Kada **b** izađe izvan opsega memorija će se odmah osloboditi osim ako se **b** ne pomeri (**move**)
- Pomeranja su takođe esencijalna za to kako Rast brine o memoriji i vrednostima, pa ćemo i o tome naknadno učiti



- Rast takođe ima sirove tipove pokazivača **\*mut T** i **\*const T**
- Sirovi pokazivači su zaista kao pokazivači u C++
- Korišćenje sirovog pokazivača nije bezbedno, jer Rast ne čini nikakav napor da prati na šta ukazuje. Na primer, neobrađeni pokazivači mogu biti null, ili mogu ukazivati na memoriju koja je oslobođena ili koja sada sadrži vrednost drugog tipa. Sve klasične greške pokazivača C++-a su ponuđene za vaše uživanje.
- Međutim, oni se mogu dereferencirati samo unutar **unsafe** bloka
- **Unsafe** blok je Rastov mehanizam da se omoguće napredne jezičke funkcija čija bezbednost zavisi od vas
- Ako vaš kod nema unsafe blokova (ili ako je kod u njima ispravno napisan), onda sigurnosne garancije Rasta i dalje važe



# INDEKSIRANE STRUKTURE

*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

- Rast ima 3 tipa koja reprezentuju indeksirane strukture, tj. sekvence vrednosti u memoriji
- **Tip  $[T; N]$**  predstavlja niz od  **$N$  vrednosti**, svaka **tipa  $T$** . Veličina niza je konstanta određena u vreme kompajliranja i deo je tipa; ne možete dodati nove elemente ili smanjiti niz.
- **Tip  $\text{Vec}\langle T \rangle$** , nazvan vektor elemenata **tipa  $T$** , je dinamički alocirana sekvenca vrednosti **tipa  $T$**  koja se može povećati. Elementi vektora žive na hipu, tako da se veličina vektora može menjati po želji: mogu se ubaciti novi elementi, dodavati novi vektori, brisanje elemenata i tako dalje.
- **Tipovi  $\&[T]$  i  $\&\text{mut } [T]$** , koji se nazivaju deljeni isečak elemenata tipa  $T$  i promenljivi isečak elemenata tipa  $T$ , su reference na niz elemenata koji su deo neke druge vrednosti, kao što je niz ili vektor



- Možete zamisliti isečak kao pokazivač na njegov prvi element, zajedno sa brojem elemenata kojima možete pristupiti počevši od te tačke
- Promenljivi isečak **&mut [T]** omogućava čitanje i menjanje elemenata, ali se ne može deliti
- Deljeni isečak **&[T]** omogućava deljenje pristupa između nekoliko čitača, ali ne dozvoljava izmene elemenata
- Za promenljivu **v** (bilo kog od ova 3 tipa) važi da je broj elemenata u **v** jednak **v.len()**
- **i**-tom elementu se pristupa preko indeksa pozicije **v[i]**; prvi element ima poziciju 0, a poslednji **v.len() – 1**; Rust uvek proverava da li se **i** nalazi u datom opsegu
- Dužina **v** može biti nula, u kom slučaju će svaki pokušaj indeksiranja izazvati paniku.
- Indeks mora biti **usize** tipa (ni jedna druga nije dopuštena)

- Postoje različiti načini da se definiše niz i njegove vrednosti

```
let lazy_caterer: [u32; 6] = [1, 2, 4, 7, 11, 16];  
let taxonomy = ["Animalia", "Arthropoda", "Insecta"];
```

```
assert_eq!(lazy_caterer[3], 7);  
assert_eq!(taxonomy.len(), 3);
```

---

```
let mut sieve = [true; 10000];  
for i in 2..100 {  
    if sieve[i] {  
        let mut j = i * i;  
        while j < 10000 {  
            sieve[j] = false;  
            j += i;  
        }  
    }  
}
```

```
assert!(sieve[211]);  
assert!(!sieve[9876]);
```

- Pristup pojedinačnom elementu niza vrši se preko indeksa i operatora **[]**

```
fn main() {  
    let a = [1, 2, 3, 4, 5];  
  
    let first = a[0];  
    let second = a[1];  
}
```

- Dužina niza je deo njegovog tipa i fiksna je u vreme kompajliranja
- Ako je **n** promenljiva, ne može se napisati **[true; n]** da bi se dobio niz od n elemenata
- Kada je potreban niz čija dužina varira tokom vremena izvršavanja (to je često potrebno), umesto niza se koristi vektor
- Rast ima ugrađene metode za rad sa nizovima

```
let mut chaos = [3, 5, 4, 1, 2];  
chaos.sort();  
assert_eq!(chaos, [1, 2, 3, 4, 5]);
```



- Vektor je zapravo niz čiji se broj elemenata može menjati
- Kreira se kao **Vect<T>**, gde je **T** tip od koga se pravi vektor
- Elementi zauzimaju kontinualnu memoriju, idu jedan pored drugog
- Vektor se može kreirati na sledeći način:

```
fn main() {  
    let v: Vec<i32> = Vec::new();  
}
```

- Ako se u startu ne dodaju nikakve vrednosti, neophodno je ubaciti tip u definiciji, u suprotnom ako se vektor inicijalizuje, moguće je dedukovati tip elemenata vektora i može se iskoristiti makro za kreiranje vektora

```
fn main() {  
    let v = vec![1, 2, 3];  
}
```

- Kog će ovo biti tipa?





- Vektori se kreiraju pomoću mehanizma **generic**, više o tome kasnije
- Vektor se može izgraditi i tako što se određena vrednost ponovi određen broj puta

```
fn new_pixel_buffer(rows: usize, cols: usize) -> Vec<u8> {  
    vec![0; rows * cols]  
}
```

- Kad ispred **v** stoji **mut**?
- Elementi se dodaju u vektor pomoću **push** metode

```
fn main() {  
    let mut v = Vec::new();  
  
    v.push(5);  
    v.push(6);  
    v.push(7);  
    v.push(8);  
}
```

- Zašto ovde ne treba tip? Čime će biti popunjen ovaj vektor?



- Elementima vektora se može pristupiti na 2 načina, ili preko indeksa ili putem **get** metode

```
fn main() {  
    let v = vec![1, 2, 3, 4, 5];  
  
    let third: &i32 = &v[2];  
    println!("The third element is {}", third);  
  
    let third: Option<&i32> = v.get(2);  
    match third {  
        Some(third) => println!("The third element is {}", third),  
        None => println!("There is no third element."),  
    }  
}
```

- Korišćenje **&** i **[ ]** vraća referencu na element sa datim indeksom
- Kada se koristi **get** sa indeksom koji je prosleđen kao argument, vraća se **Option** koji se kombinuje sa **match**
- Ovi pristupi su tu zbog rukovanja greškom kod prekoračenja opsega



```
fn main() {  
    let v = vec![1, 2, 3, 4, 5];  
  
    let does_not_exist = &v[100];  
    let does_not_exist = v.get(100);  
}
```

- Prvi pristup elementu vektor, `[ ]`, će dovesti do toga da kod paniči i najbolje ga je koristiti kad želimo da program padne u slučaju prekoračenja opsega (kada je ovo ilegalno, tj. ne postoji normalna situacija u kojoj se ovo može desiti)
- Drugi pristup, sa **get** metodom, vraća **None** ako je došlo do prekoračenja opsega i omogućuje način da se rukuje situacijom kada dolazi do prekoračenja (koristi se kada je prekoračenje normalno, tj. može se legitimno desiti tokom izvršavanja)
- Ima ovde još par tvikova, ali više o tome kad vidimo šta radi pozajmljivanje u Rastu



- **Vec** je esencijalni tip za Rast — koristi se skoro svuda gde je potrebna lista dinamičke veličine — tako da postoje mnoge druge metode koje konstruišu nove vektore ili proširuju postojeće
- **Vec<T>** se sastoji od tri vrednosti:
  - pokazivača na bafer u hipu koji vektor zauzima i poseduje;
  - broj elemenata koje bafer ima kapacitet da uskladišti;
  - broj elemenata koji se sada zapravo nalaze u vektoru (drugim rečima, njegova dužina)
- Kada bafer dostigne svoj kapacitet, dodavanje novog elementa u vektor podrazumeva dodeljivanje većeg bafera, kopiranje sadašnjeg sadržaja u njega, ažuriranje pokazivača i kapaciteta vektora da opiše novi bafer i konačno oslobađanje starog
- Iako je esencijalan tip za Rast, nije ugrađen u sam jezik



- Ako se broj elemenata vektora zna unapred, onda se umesto **Vec::new** može pozvati **Vec::with\_capacity**, ovo će unapred napraviti vektor dovoljno velike veličine
- I dalje je moguće dodavati nove elemente u vektor i prevazići inicijalnu procenu potrebne veličine

```
let mut v = Vec::with_capacity(2);
assert_eq!(v.len(), 0);
assert_eq!(v.capacity(), 2);

v.push(1);
v.push(2);
assert_eq!(v.len(), 2);
assert_eq!(v.capacity(), 2);

v.push(3);
assert_eq!(v.len(), 3);
// Typically prints "capacity is now 4":
println!("capacity is now {}", v.capacity());
```

- Metoda **capacity** vraća info koliki je kapacitet vektora



- Novi elementi se pored dodavanja na kraj vektora, mogu i ubaciti na bilo koju poziciju u vektoru
- Elementi se mogu i obrisati
- Ovo dovodi do pomeranja elemenata u levo ili desno
- U zavisnosti od veličine vektora, ovo može biti zahtevna operacija. Zašto?

```
let mut v = vec![10, 20, 30, 40, 50];
```

```
// Make the element at index 3 be 35.
```

```
v.insert(3, 35);
```

```
assert_eq!(v, [10, 20, 30, 35, 40, 50]);
```

```
// Remove the element at index 1.
```

```
v.remove(1);
```

```
assert_eq!(v, [10, 30, 35, 40, 50]);
```



- Moguće je odraditi i izvlačenje poslednjeg elementa vektora upotrebom **pop** metode
- Upotreba **push** i **pop**, esencijalno omogućuje korišćenje vektora kao da je **stek**
- Ova metoda kao rezultat vraća **Option** koji ima vrednost **None** ako je vektor prazan ili **Some(v)** ako je poslednji element **v**

```
let mut v = vec!["Snow Puff", "Glass Gem"];  
assert_eq!(v.pop(), Some("Glass Gem"));  
assert_eq!(v.pop(), Some("Snow Puff"));  
assert_eq!(v.pop(), None);
```



- Moguće je kretati se kroz vektor upotrebom iteratora
- Read-only pristup

```
fn main() {  
    let v = vec![100, 32, 57];  
    for i in &v {  
        println!("{}", i);  
    }  
}
```

- Write pristup

```
fn main() {  
    let mut v = vec![100, 32, 57];  
    for i in &mut v {  
        *i += 50;  
    }  
}
```



# ISEČCI (SLICES)



*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

- Isečci omogućavaju da se referencira neprekinut (kontinualni) niz elemenata u nekoj kolekciji, a ne cela kolekcija
- Može se zamisliti kao region u vektoru ili nizu
- Kako se radi o mogućem pozamašnom broju elemenata, **slice** se uvek prenosi po referenci
- **slice** se sastoji iz dva dela: reference na prvi element u isečku i broj elemenata u isečku
- Reference na niz ili vektor se automatski konvertuju u **slice** celog niza i **slice** celog vektora

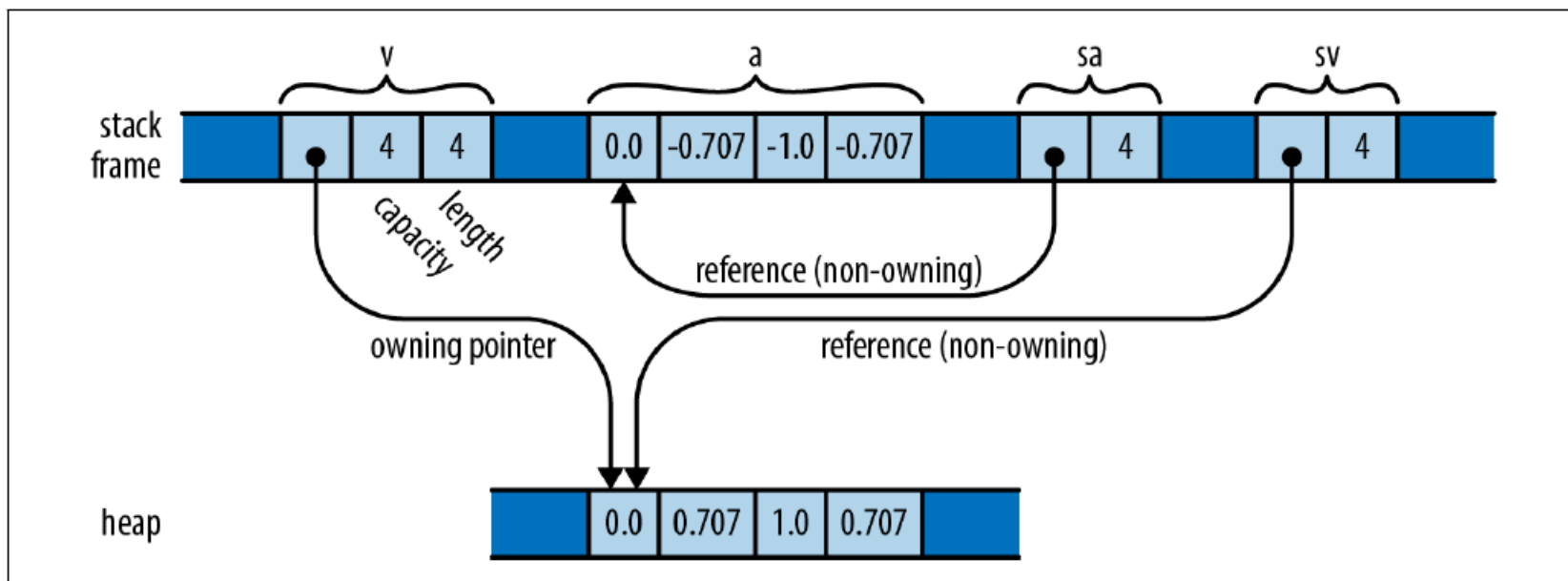
```
let v: Vec<f64> = vec![0.0, 0.707, 1.0, 0.707];  
let a: [f64; 4] = [0.0, -0.707, -1.0, -0.707];  
  
let sv: &[f64] = &v;  
let sa: &[f64] = &a;
```

# ISEČCI (SLICES)



*Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici*

- Kako memorija izgleda:



- Ono što je ključno je da **slice** ne poseduju datu memoriju



# ISEČCI (SLICES)

*Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici*

- **slice** je odličan izbor kada se želi napisati metoda koja radi i nad nizovima i nad vektorima

```
fn print(n: &[f64]) {  
    for elt in n {  
        println!("{}", elt);  
    }  
}
```

```
print(&a); // works on arrays  
print(&v); // works on vectors
```

- Upravo zbog toga, većina metoda koje rade nad nizovima i vektorima, zapravo su ista metoda koja radi nad slicom



# ISEČCI (SLICES)

*Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici*

- Možete dobiti referencu na isečak niza ili vektora ili deo postojećeg isečka tako što ćete ga indeksirati opsegom

```
print(&v[0..2]);    // print the first two elements of v
print(&a[2..]);      // print elements of a starting with a[2]
print(&sv[1..3]);    // print v[1] and v[2]
```

- Pošto se isečci skoro uvek pojavljuju iza referenci, tipovi kao što su **&[T]** ili **&str** često se nazivaju prosto isečci (**slices**), koristeći kraće ime za češći koncept

# STRINGOVI



*Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici*

- Rust ima više vrste stringova, string literale, byte stringove, i string u memoriji (niz Unikod karaktera)



# STRING LITERALI

*Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici*

- Šta je string literal?
- String literali su zatvoreni dvostrukim navodnicima

```
let speech = "\"Ouch!\" said the well.\n";  
  
println!("In the room the women come and go,  
    Singing of Mount Abora");  
  
println!("It was a bright, cold day in April, and \  
    there were four of us-\  
    more or less.");
```

- Sirovi stringovi (raw strings)

```
let default_win_install_path = r"C:\Program Files\Gorillas";  
  
let pattern = Regex::new(r"\d+(\.\d+)*");
```



- Počinje sa sufiksom **b**

```
let method = b"GET";  
assert_eq!(method, &[b'G', b'E', b'T']);
```

- Niz bajtova fiksne dužine (vrednosti od 0 do 255)
- Može biti i **immutable** i **mutable**, nije **unikod** string
- Zašta ih koristimo?
- Za rad sa tekstom koji sadrži iskučivo ASCII znakove



# STRING U MEMORIJI

*Dragan da Dinu - Paralelne i distribuirane arhitekture i jezici*

- Rast stringovi su nizovi Unicode znakova
- Ne čuvaju se u memoriji kao nizovi znakova, već kao UTF-8 kod, promenljive širine (<https://en.wikipedia.org/wiki/UTF-8>)
- Svaki ASCII karakter u nizu se čuva u jednom bajtu, dok ostali znakovi zauzimaju više bajtova
- String se čuva u baferu čija se veličina može promeniti i koji sadrži **UTF-8** tekst
- Bafer se čuva u hipu, tako da se veličina može promeniti po potrebi ili na zahtev
- String je zapravo implementiran kao **Vec<u8>** koji čuva dobro formatiran **UTF-8**

```
let noodles = "noodles".to_string();
```





# STRING U MEMORIJI

*Dragan da Dinu – Paralelne i distribuirane arhitekture i jezici*

- `&str` („stir“ ili „string slice“, string isečak) je referenca na sekvencu UTF-8 teksta čiji je vlasnik neko drugi – isečak „pozajmljuje“ taj tekst

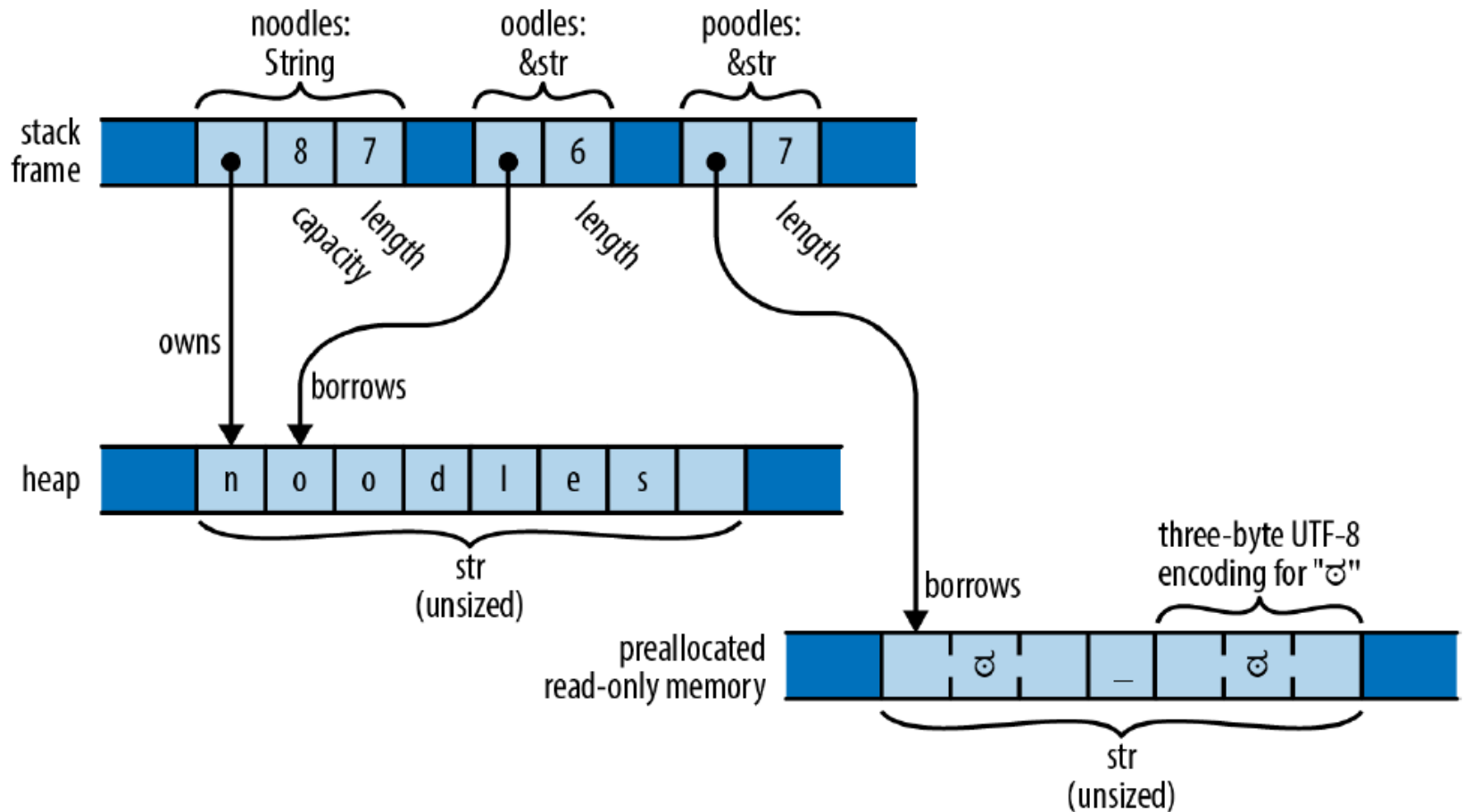
```
let noodles = "noodles".to_string();
let oodles = &noodles[1..];
let poodles = "໐_໐";
```
- Isečak teksta je pointer koji pored adrese sadrži i podatak o dužini isečka
- **&str** je ništa više do `&[u8]` koji je sadrži dobro formiran **UTF-8**
- String literal je **&str** koji se odnosi na unapred dodeljeni tekst, nalazi se u read-only memoriji zajedno sa mašinskim kodom programa
- **poodles** je string literal, koji ukazuje na sedam bajtova koji se kreiraju kada program počne da se izvršava i koji traju dok i program traje



# STRING U MEMORIJI

*Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici*

- String, &str i str





# STRING U MEMORIJI

*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

- Voditi računa **String** i **&str** metoda **.len( )** vraća dužinu teksta u bajtovima, ne u broju karaktera
- **&str** se ne može modifikovati, umesto toga koristiti **String**
- **String** se kreira
  - direktno iz literala primenom metode **from**
  - pretvaranjem literala u **String** primenom metode **to\_string**
  - primenom **format!** makroa
  - metode kombinovanje više stringova u novi string poput **.concat()** ili **.join()**

```
let error_message = "too many pets".to_string();
```

```
let hello = String::from("Hello, world!");
```

```
format!("test");
```

```
format!("hello {}", "world!");
```

```
format!("x = {}, y = {}", 10, y = 30);
```



# STRING U MEMORIJI

*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

- **String** je zapravo analogan **Vec<T>**

	<b>Vec&lt;T&gt;</b>	<b>String</b>
Automatically frees buffers	Yes	Yes
Growable	Yes	Yes
<code>::new()</code> and <code>::with_capacity()</code> type-associated functions	Yes	Yes
<code>.reserve()</code> and <code>.capacity()</code> methods	Yes	Yes
<code>.push()</code> and <code>.pop()</code> methods	Yes	Yes
Range syntax <code>v[start..stop]</code>	Yes, returns <code>&amp;[T]</code>	Yes, returns <code>&amp;str</code>
Automatic conversion	<code>&amp;Vec&lt;T&gt;</code> to <code>&amp;[T]</code>	<code>&amp;String</code> to <code>&amp;str</code>
Inherits methods	From <code>&amp;[T]</code>	From <code>&amp;str</code>

- Kao i **Vec**, **String** ima svoj bafer dodeljen na hipu
- Kada string promenljiva izađe van opsega, bafer se automatski oslobađa, osim ako string nije pomeren (**moved**)
- **String** ili **&str** ?



# STRING U MEMORIJI

*Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici*

- **String** podržava **==** i **!=**, kao **<**, **<=**, **>** i **>=**
- Korisne metode za rad sa stringovima nalaze se u **std::str** modulu
  - **.contains()**, **.replace()**, **.trim()**, **.split()** su samo neke od njih
- Zbog prirode UTF-8 koda poređenje stringova, kao i njihovo leksičko sortiranje ne daje uvek očekivane rezultate (ako stignemo, bavićemo se malo ovim)



# TIP SINONIMI (TYPE ALIASES)

*Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici*

- Kao u C/C++ ključna reč **type** se može iskoristi za definisanje „novih tipova,” tj. sinonima ili aliasa

```
type Bytes = Vec<u8>;
```

```
fn decode(data: &Bytes) {  
    ...  
}
```



# OVO NISU SVI TIPOVI

*Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici*

- Pored nabrojanih tipova postoje još:
  - structs
  - enums
  - traits
  - closures
  - collection types
  - ...
- O njima malo kasnije

