PV281: Programování v Rustu

Obsah

Cargo

Binding & Shadowing

Paměťový model & Borrow checker

Práce s řetězci

Cargo

Vytváří nové projekty

Spravuje závislosti

Spouští testy

Publikuje balíčky

Založení projektu

```
cargo new nazev_projektu
```

V základu se založí projekt pro aplikaci (binary)

Založení knihovny

```
cargo new nazev_projektu --lib
```

Založení projektu v existující složce

```
mkdir nazev_projektu
cd nazev_projektu
cargo init
```

Sestavení projektu

```
cargo build
```

Vytvoří složku target a v ní debug. Program je pomalejší, protože obsahuje více instrukcí. Získáme tím podporu debugování.

Produkční sestavení projektu

```
cargo build --release
```

Takto vždy vytváříme aplikaci pro distubuci klientům nebo pro nasazení na server.

Spuštění testů

```
cargo test
```

Generování dokumentace

```
cargo doc
```

Publikování balíčku

```
cargo publish
```

Spuštění

```
cargo run options -- args
```

Argumenty před -- jsou do Cargo, argumenty za jdou do binárky.

rustfmt

```
rustup component add rustfmt
```

```
cargo fmt
```

clippy

detekuje a případně opravuje další chyby v kódu

```
rustup component add clippy
```

```
cargo clippy
```

To je ke cargo pro dnešek vše, jdeme na kód

Základní program

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

Binding

```
fn main() {
    let x = 5; // toto je binding, v jinych jazycich definice
               // je imutabilní = neměnné = konstantní
    println!("Hodnota x je: {}", x);

    // x += 5 by hodilo chybu
    // x = 10 by taktez hodilo chybu

    let y: u32 = 6;
    println!("Hodnota y je: {}", y);
}
```

Co když potřebujeme změnit hodnotu v proměnné?

Mut binding

```
fn main() {
    let mut x = 5; // toto je binding, v jinych jazycich definice
                   // díky klíčovému slovu mut jej můžeme změnit
    println!("Hodnota x je: {}", x);

    x = 10;
    println!("Hodnota x je: {}", x);


    x += 1; // hned na začátek si raději řekneme, že Rust nemá
            // inkrementaci (žádné x++ nebo ++x)
            // proč? nepoužíváme cykly k iterování
    println!("Hodnota x je: {}", x);
}
```

Shadowing

```
fn main() {
    let x = 5;
    println!("Hodnota x je: {}", x);

    let x = x + 5; // použitím let překryjeme původní proměnnou
                   // nová položka na stacku
    println!("Hodnota x je: {}", x);
}
```

Shadowing

```
fn main() {
    let x = 5;
    println!("Hodnota x je: {}", x);

    let x = x + 5; // použitím let překryjeme původní proměnnou
                   // nová položka na stacku
    println!("Hodnota x je: {}", x);
}
```

Datové typy

Celočíselné typy

Velikost	Znaménkový	Neznaménkový
8 bitů	i8	u8
16 bitů	i16	u16
32 bitů	i32	u32
64 bitů	i64	u64
128 bitů	i128	u128

Zápisy literálů

Velikost	Příklad
desítkové	98_222
šestnáctkové	0xff
osmičkové	0o77
binární	0b1111_0000
bajtové	b'A'

S plovoucí řádovou čárkou (IEEE-754)

Velikost	Typ
32 bitů	f32
64 bitů	f64

výchozí je f64

Přetypování desetinných čísel

```rust
fn main() {
    let x = 2;
    let y: f64 = x as f64;
    let z: i32 = y as i32;

    let u: i32 = y.trunc();
    let v: i32 = y.ceil();
    let w: i32 = y.floor();

    println!("Hodnota x je: {}", x);
    println!("Hodnota y je: {}", y);
    println!("Hodnota z je: {}", z);
}
```

Přetypování boolu

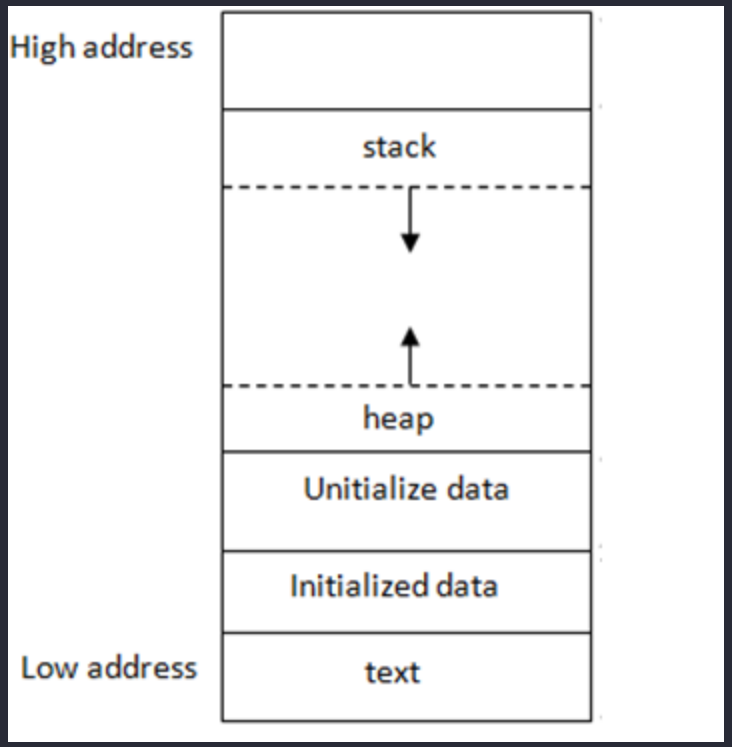
```rust
fn main() {
    let t = true;

    let f: bool = false; // explicitní typ

    let value: i32 = f as i32; // bool je v Rust vždy 0 nebo 1. Nic jiného.
}
```

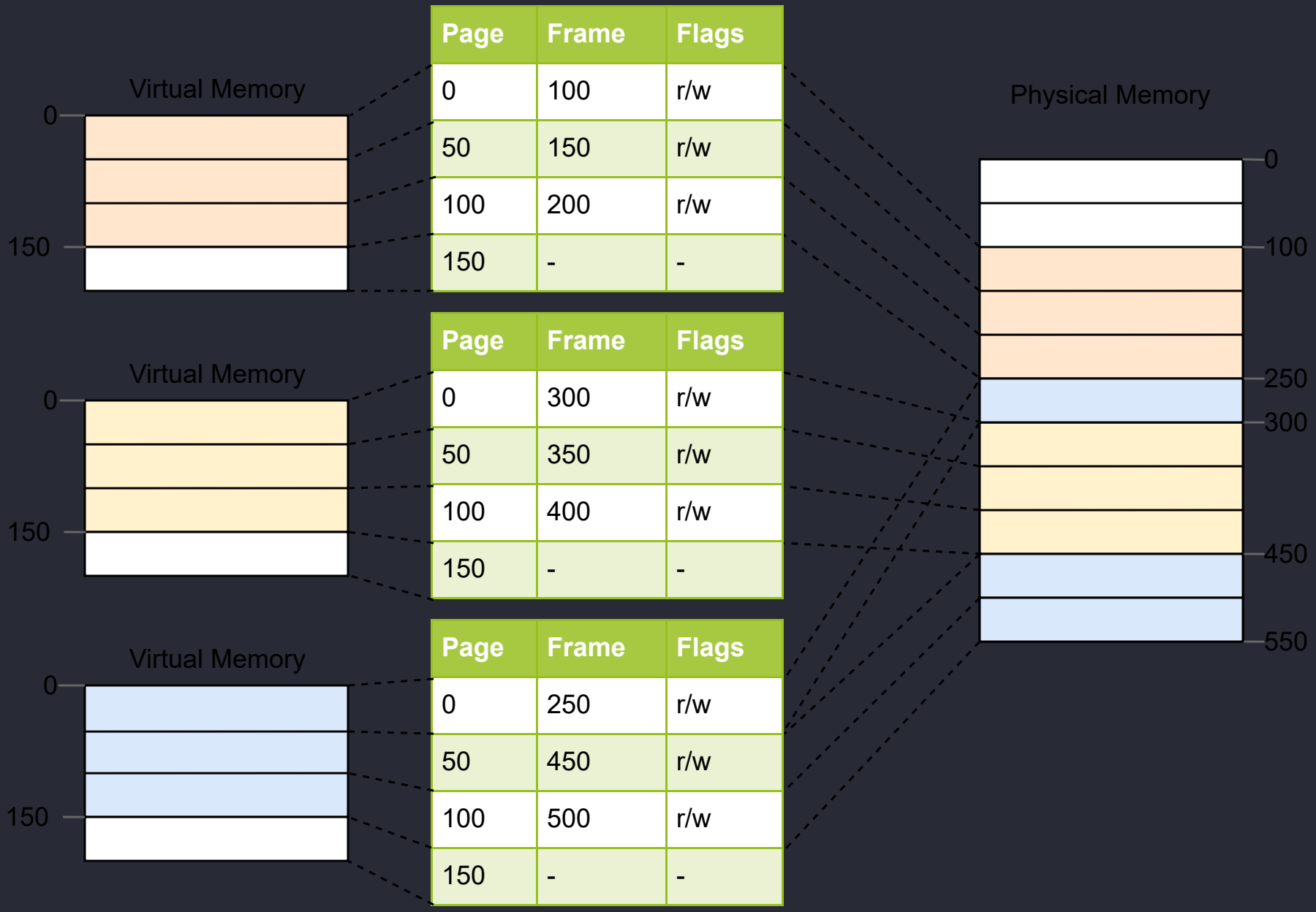
Práce s pamětí

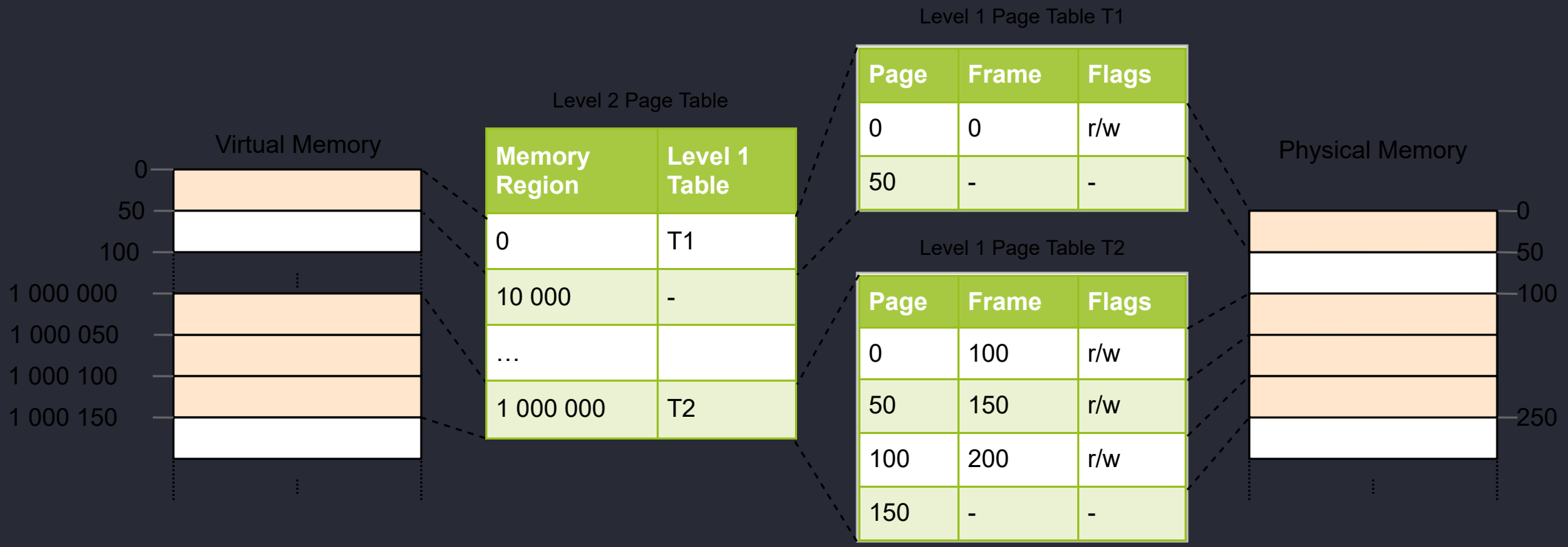
Rozdělení paměti (C programy)

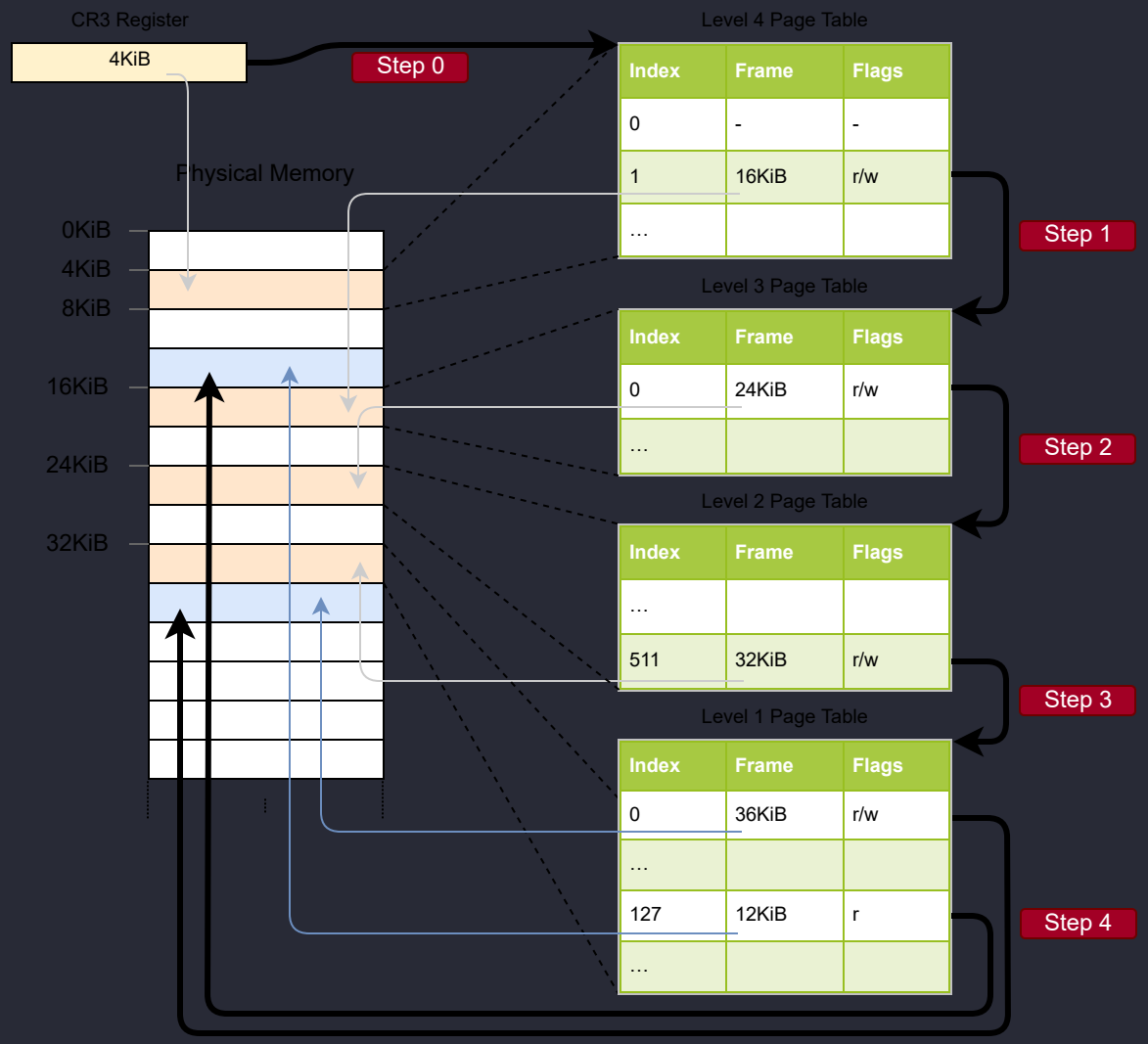


Virtuální paměť

Dnes se řeší praktický jen stránkováním. To je víceúrovňové. Dříve se používala i segmentace a nebo kombinace segmentace se stránkováním.


Virtual Memory
0
150
Page
Frame
Flags
0
100
r/w
50
150
r/w
100
200
r/w
150
-
-
Virtual Memory
0
150
Page
Frame
Flags
0
300
r/w
50
350
r/w
100
400
r/w
150
-
-
Virtual Memory
0
150
Page
Frame
Flags
0
250
r/w
50
450
r/w
100
500
r/w
150
-
-
Physical Memory
0
100
250
300
450
550


Virtual Memory
0
50
100
1 000 000
1 000 050
1 000 100
1 000 150
Level 2 Page Table
Memory Region
Level 1 Table
0
T1
10 000
-
…
1 000 000
T2
Level 1 Page Table T1
Page
Frame
Flags
0
0
r/w
50
-
-
Level 1 Page Table T2
Page
Frame
Flags
0
100
r/w
50
150
r/w
100
200
r/w
150
-
-
Physical Memory
0
50
100
250


CR3 Register
4KiB
Step 0
Physical Memory
0KiB
4KiB
8KiB
16KiB
24KiB
32KiB
Level 4 Page Table
Index Frame Flags
0 - -
1 16KiB r/w
…
Step 1
Level 3 Page Table
Index Frame Flags
0 24KiB r/w
…
Step 2
Level 2 Page Table
Index Frame Flags
…
511 32KiB r/w
Step 3
Level 1 Page Table
Index Frame Flags
0 36KiB r/w
…
127 12KiB r
…
Step 4

Stack

Rust se snaží primárně využívat stack kvůli rychlosti. Stack je vždy samostatný pro každé vlákno.

Jde o lineární paměť určité velikosti.

Data jsou vždy pevné velikosti.

Pro zajímavost:

1. Vlákno v Rustu má defaultně 2MB
2. Windows standardně dává 1MB, Linux oproti tomu 8MB

Velikost vlánka: sys:🧵:DEFAULT_MIN_STACK_SIZE


w:512 h:512

Stack Time Complexity

Push: O(1)

Pop: O(1)

Top: O(1)

Search: O(n)

Heap

Používáme pro data proměnné velikosti. Heap je sdílený mezi vlákny. Často programovací jazyky využívají několik heapů.

Rust dává k dispozici jeden heap ve stable. Nemůžete změnit alokátor. V nightly to jde, ale pokud netvoříte OS, tak to nedělejte.

Heap se běžně implementuje jako linked list nebo tree.

Tree Complexity

Insert, delete and search:
Average case: O(log n),
Worst Case: O(log n) or O(n) based on implementation

Založení proměnné na haldě

```
fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

Ownership

Založený na principu RAII (Resource Aquisition Is Initialization).

Přesun na zásobníku

```
fn main() {
    let x = 5;
    let y = x;
}
```

Přesun na haldě

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;
}
```

Tímto dojde ke zkopírování ukazatele na stejné místo v paměti

Přesun na haldě

Pokud potřebujeme data na haldě zkopírovat, tak musíme klonovat.

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1.clone();

    println!("s1 = {}, s2 = {}", s1, s2);
}
```

Převzetí z vrácení vlastnictví

```
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("The length of '{}' is {}.", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() returns the length of a String

    (s, length)
}
```

Použití reference

Vlastník se nemění, předáváme objekt s tím, že se nebude modifikovat.

```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

Použití reference

Dokud ovšem nepoužijeme mutovatelnou referenci.

```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

Jak funguje kontrola

1. Můžeme vytvořit neomezeně nemutovatelných referencí
2. Můžeme mít pouze jednu mutovatelnou referenci
3. Nekombinujeme mutovatelné a nemutovatelné
4. Odkaz musí být platný

Co nedělat u referencí

```
fn main() {
    let mut s = String::from("hello");

    let r1 = &s; // žádný problém
    let r2 = &s; // taky žádný problém
    let r3 = &mut s; // VELKÝ PROBLÉM

    // důvodem problému je následující řádek
    println!("{}, {}, and {}", r1, r2, r3);
}
```

Jak vyřešit potřebu různých typů referencí?

```
fn main() {
    let mut s = String::from("hello");

    let r1 = &s; // žádný problém
    let r2 = &s; // taky žádný problém
    println!("{} and {}", r1, r2);
    // r1 a r2 se dále nepoužívají, takže zádný problém nebude

    let r3 = &mut s; // a tady to už problém není
    println!("{}", r3);
}
```

Práce se stringy

Problémy práce se stringy

Ve většině jazyků je práce s typem string jednoduchá. V Rustu je vzhledem k paměťovému modelu složitější a je častým zdrojem problémů pro začátečníky.

Pod pokličkou jsou řetězce relativně komplexní ve všech jazycích.

String a &str

String a &str jsou zřetězením znaků UTF-8. Oba používáme a můžeme mezi nimi přecházet.

String je mutovatelný a uložený na haldě.

&str je imutabilní, uložený na zásobníku haldě nebo součást kódu.

Převod mezi string a &str

```
fn main() {
    let hello_str: &str = "hello";

    let hello_string: String = String::from("hello");

    let hello_string_from_str: String = hello_str.to_string();
    let hello_string_from_str: String = "hello".to_string();

    let hello_str_from_string: &str =  &hello_string;
}
```

Slice

```
fn main() {
    let s = String::from("hello world");

    let hello = &s[0..5];
    let world = &s[6..11];
}
```

Konkatenace

```
fn main() {
    let hello_world_concat = ["hello", " world"].concat();

    let hello_world_concat_format = format!("{} {}", "hello", " world");

    // pokud chcete spojovat pluskem, tak vysledek je string a prvni argument musi byt string
}
```

Práce se stringem

```
fn main() {
    let mut my_string = String::new();
    my_string.push_str("hello");
    my_string.push('!');
}
```

Práce se znaky

```
fn main() {
    let pangram: &'static str = "the quick brown fox jumps over the lazy dog";

    // Copy chars into a vector, sort and remove duplicates
    let mut chars: Vec<char> = pangram.chars().collect();
    chars.sort();
    chars.dedup();

     // Create an empty and growable `String`
    let mut string = String::new();
    for c in chars {
        // Insert a char at the end of string
        string.push(c);
        // Insert a string at the end of string
        string.push_str(", ");
    }
```

Byte string

```
fn main() {
    let raw_bytestring = br"\u{211D} is not escaped here";

     // Converting a byte array to `str` can fail
    if let Ok(my_str) = str::from_utf8(raw_bytestring) {
        println!("And the same as text: '{}'", my_str);
    }
}
```

Dotazy?

Děkuji za pozornost