PV281: Programování v Rustu

Obsah

1. Týmové projekty
2. Enumy
3. Pattern Matching
4. Struktury
5. Lifetimes
6. Traity
7. Ošetření chyb
8. CLI aplikace
9. Práce se soubory a stdin

Týmové projekty

Zveřejnění projektů

Dnes 2. 10. 2023 v 20:00.
Naleznete je v ISu v Rozpisu `Týmové projekty`.

Spuštění přihlašování

Od pondělí 9. 10. 2023 v 20:00 se může přihlásit 1 člen za tým.

Od středy 11. 10. 2023 v 20:00 se můžou přihlásit i ostatní členové týmu.

Změny v přihlášení na projekt

Do 20. 11. 2023 máte možnost bezproblémově měnit sestavy týmů.

V případě problémů s týmem (během této doby i po ní) můžete kontaktovat koordinátora projektů Petra Wehrenberga.

Vlastní zadání projektu

Do 20. 11. 2023 také můžete navrhovat vlastní zadání projektu.

- Vaše zadání musí být explicitně schváleno vaším cvičícím.
- Vaše zadání musí odpovídat náročností ostatním projektům. Pokud nebude, vrátíme vám ho se zpětnou vazbou a můžete ho vylepšit.

Obhajoby

Nejpozději do konce roku zveřejníme termíny obhajob, které pak **budou probíhat po celé zkouškové** období.

Obhajoba je asi 20minutový online call, kde:

- v rychlosti představíte projekt,
- ukážete demo vaší aplikace
- zodpovíte otázky k demu i kódu vaší aplikace

Novinka

Odevzdání projektu

Projekt odevzdáváte 3 dny před obhajobou.

1. přidáte cvičících do repozitáře s projektem
2. vytvoříte větev `project-submission` a už do ní nepřispíváte

Novinka

Frontend jen v Rustu

Zakazujeme používat frameworky/knihovny z JavaScript/TypeScript ekosystému pro tvorbu frontendu.

Místo toho použijte crates dostupné v Rustu.

Dotazy k projektům?

Enumy

Základní varianta

```
enum Delivery {
    Pickup,
    Parcel,
}

fn main() {
    let delivery = Delivery::Pickup;
}
```

Enumy jako v C

```
// hodnota defaultně začíná nulou
enum Number {
    Zero,
    One,
    Two,
}

// hodnota je ale nastavitelná
enum Color {
    Red = 0xff0000,
    Green = 0x00ff00,
    Blue = 0x0000ff,
}

fn main() {
    // enum můžeme přetypovat na celé číslo
    println!("zero is {}", Number::Zero as i32);
    println!("one is {}", Number::One as i32);

    println!("roses are #{:06x}", Color::Red as i32);
    println!("violets are #{:06x}", Color::Blue as i32);
}
```

Monadická varianta

```
enum Delivery {
    Pickup,
    Parcel(String),
}

fn main() {
    let pickup_delivery = Delivery::Pickup;
    let parcel_delivery = Delivery::Parcel(String::from("Ceska 0, 60200 Brno"));
}
```

Paměťová reprezentace enumu

- Velikost je určena podle největší položky (stejně jako např. `union` v C).
- Kromě toho je zde ještě skrytá položka - diskriminant.
- Velikost diskriminantu je závislá na počtu variant enumu.
- Hodnota diskriminantu určuje aktuální variantu enumu.

Zpracování hodnoty pomocí `if let`

```
fn main() {
    let some_u8_value = Option::Some(0u8);

    if let Some(num) = some_u8_value {
        println!("It was Some! It's inner value was {}!", num);
    }
}
```

V příkladu využíváme důležitý enum `std::option::Option`:

```
enum Option<T> {
    Some(T),
    None,
}
```

Co nedělat

`if let` > `if`

```
fn main() {
    let maybe_value = Some('k');

    if maybe_value.is_some() {
        let value = maybe_value.unwrap();
    }
}
```

Když chci rozbalit 1 variantu enumu, radši použiju `if let`.

`match` > `if let`

```
enum Delivery {
    Pickup,
    Parcel(String),
    Special{ box_id: u128 }
}

fn main() {
    let variable = Delivery::Pickup;

    if let Delivery::Pickup = variable {
        println!("Vyzvednete to u nás v obchodě!");
    } else if let Delivery::Parcel(address) = variable {
        println!("Zboží bude doručeno na adresu: {}!", address);
    } else if let Delivery::Special {box_id} = variable {
        println!("Zboží bude doručeno na do boxu č.: {}!", box_id);
    } else {
        println!("Zatím neimplementovaný způsob doručení!");
    }
}
```

Když chci rozbalit hodně variant enumu, radši použiju `match`.

Pattern matching

Výhody pattern matchingu

1. Kontrola všech variant větvení

To pomáhá i při refaktoringu – nikdy nezpomenete změnit další místa v kódu.

2. Lepší čitelnost

Zpracování hodnoty pattern matchingem

```
fn deliver(delivery: Delivery) {
    match delivery {
        Delivery::Pickup => {
            println!("Vyzvednete to u nás v obchodě!");
        },
        Delivery::Parcel(address) => {
            println!("Zboží bude doručeno na adresu: {}!", address);
        },
        _ => {
            println!("Zatím neimplementovaný způsob doručení!");
        }
    }
}
```

Match různých tvarů enumu

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

fn main() {
    let msg = Message::ChangeColor(0, 160, 255);

    match msg {
        Message::Quit                => println!("The Quit variant has no data to destructure.");
        Message::Move { x, y }       => println!("Move in the x direction {} and in the y direction {}", x, y);
        Message::Write(text)         => println!("Text message: {}", text),
        Message::ChangeColor(r, g, b) => println!("Change the color to red {}, green {}, and blue {}", r, g, b),
    }
}
```

Match & shadowing

```
fn main() {
    let x = Some(5);
    let y = 10;

    match x {
        Some(50) => println!("Got 50"),
        Some(y) => println!("Matched, y = {}", y),
        _ => println!("Default case"),
    }

    println!("at the end: y = {}", y);
}
```

Match guards

```
fn main() {
    let x = Some(10);
    let y = 10;

    match x {
        Some(50) => println!("Got 50"),
        Some(a) if a == y => println!("Got y as a = {}", a),
        Some(y) => println!("Matched, y = {}", y),
        _ => println!("Default case"),
    }

    println!("at the end: y = {}", y);
}
```

Match několika variant

```
fn main() {
    let x = 1;

    match x {
        1 | 2 => println!("one or two"),
        3 => println!("three"),
        _ => println!("anything"),
    }
}
```

Match nad range

```
fn main() {
    let x = 'c';

    // ..= znamená včetně posledního prvku
    match x {
        'a'..='j' => println!("early ASCII letter"),
        'k'..='z' => println!("late ASCII letter"),
        _ => println!("something else"),
    }
}
```

@ binding

```
fn main() {
    let x = 'c';

    // ..= znamená včetně posledního prvku
    match x {
        early @ 'a'..='j' => println!("early ASCII letter: {}", early),
        late @ 'k'..='z' => println!("late ASCII letter: {}", late),
        _ => println!("something else"),
    }
}
```

Match nad tuple

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);

    let (a, b) = match numbers {
        (first, .., last) => {
            println!("Some numbers: {}, {}", first, last);
            (first, last)
        }
    };

    match (a, b) {
        (4, 2) => println!("Everything."),
        (9, _) => println!("Nine."),
        (6, 9) => println!("Nice!"),
        _ => {},
    }
}
```

Struktury

Definice struktury

```
struct Foo {
  tiny: bool,
  normal: u32,
  small: u8,
  long: u64,
  short: u16,
}
```

Zarovnání v paměti

Dle C

```
|tiny|PADDING|normal|small|PADDING|long|short|PADDING|
| 1B |  3B   |  4B  | 1B  |   7B  | 8B | 2B  |  6B   |

# Total of 32 bytes.
```

Položky jsou v deterministickém pořadí, figuruje zarovnání.

```rust
#[repr(C)]
struct MyStruct {}
```

V Rustu

Není deterministické řazení položek, může figurovat zarovnání.

```rust
#[repr(Rust)]
struct MyStruct {}
```

```rust
struct MyStruct {}
```

Alternativní modely

`#[repr(packed)]`

- Nepoužívá zarovnání.
- Vhodné v prostředí s málo pamětí nebo s pomalým síťovým spojením.
- Může velmi zpomalit vykonávání a může dojít k pádu, pokud CPU podporuje pouze zarovnané argumenty.

`#[repr(align(n))]`

- Umožňuje větší zarovnání, než by bylo nutné.
- Pro scénáře, kdy potřebujeme zařidít, aby položky byly v různých *cache lines*. Vyhneme se problému zvanému **false sharing**.
- K false sharingu dochází, když různá CPU sdíli cache line. Oba se ji mohou pokusit změnit současně.

Práce se strukturou

```
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}

fn main() {
    // ...
}
```

Použití struktury

```
struct User {
    // ...
}

fn main() {
    let mut user1 = User {
        email: String::from("someone@example.com"),
        username: String::from("someusername123"),
        active: true,
        sign_in_count: 1,
    };

    user1.email = String::from("anotheremail@example.com");
}
```

Vytvoření z již existující struktury

```
fn main() {
    let user1 = User {
        email: String::from("someone@example.com"),
        username: String::from("someusername123"),
        active: true,
        sign_in_count: 1,
    };

    let user2 = User {
        email: String::from("another@example.com"),
        username: String::from("anotherusername567"),
        ..user1
    };
}
```

Metody struktury

```
impl User {
    fn new(email: String, username: String) -> Self {
        User {
            email,
            username,
            active: true,
            sign_in_count: 1,
        }
    }

    fn is_active(&self) -> bool {
        self.is_active
    }

    fn change_email(&mut self, new_email: String) {
        self.email = new_email;
    }
}

fn main() {
    let mut user = User::new("someone@example.com".to_string(), "someuser123".to_string());
    user.change_email("someone.else@provider.com".to_string());
    println!("{}", user.is_active()); // equivalent to User::is_active(&user)
}
```

Makro Debug

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rectangle = Rectangle {
        width: 30,
        height: 50,
    };

    println!("rectangle is '{:?}'", rectangle);
}
```

```
rectangle is 'Rectangle { width: 30, height: 50 }'
```

Lifetimes

Struktury s referencí

Pokud má struktura obsahovat referenci, tak musíme definovat **lifetime**.

```
struct Extrema<'a> {
    greatest: &'a i32,
    least: &'a i32
}
```

Příklad použití funkce extrema

```
fn find_extrema<'s>(slice: &'s [i32]) -> Extrema<'s> {
    let mut greatest = &slice[0];
    let mut least = &slice[0];

    for i in 1..slice.len() {
        if slice[i] < *least    { least    = &slice[i]; }
        if slice[i] > *greatest { greatest = &slice[i]; }
    }
    Extrema { greatest, least }
}
```

Lifetime

Je konstrukce překladače, která udává dobu platnosti reference.

Dříve bylo nutností ji explicitně definovat, dneska už není moc často třeba. Běžný kód by měl většinou jít napsat i bez specifikace lifetimu.

Lifetime

```
fn main() {
    let i = 3; // Lifetime pro `i` začíná. ──────────────────┐
    //                                                       │
    { //                                                     │
        let borrow1 = &i; // `borrow1` lifetime začíná. ──┐  │
        //                                                │  │
        println!("borrow1: {}", borrow1); //              │  │
    } // `borrow1 koncí. ─────────────────────────────────┘  │
    //                                                       │
    //                                                       │
    { //                                                     │
        let borrow2 = &i; // `borrow2` lifetime začíná. ──┐  │
        //                                                │  │
        println!("borrow2: {}", borrow2); //              │  │
    } // `borrow2` končí. ────────────────────────────────┘  │
    //                                                       │
}   // Lifetime pro `i` končí. ──────────────────────────────┘
```

Explicitní anotace lifetimu

```
&i32        // a reference
&'a i32     // a reference with an explicit lifetime
&'a mut i32 // a mutable reference with an explicit lifetime
```

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Pokud nepoužijeme lifetime, kód nejde přeložit - překladač netuší, jak dlouho má žít návratová hodnota.

Co nebude fungovat

```rust
fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```

```
error[E0515]: cannot return reference to local variable `result`
   |
11 |     result.as_str()
   |     ^^^^^^^^^^^^^^^ returns a reference to data owned by the current function
```

Lifetime s generikou

```
use std::fmt::Display;

fn longest_with_announcement<'a, T: Display>(
    x: &'a str,
    y: &'a str,
    ann: T,
) -> &'a str
{
    println!("Announcement! {}", ann);

    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetime u struktur

Pokud máme ve struktuře referenci, **vždy musíme definovat lifetime**.

```
// Typ `Borrowed` obsahuje referenci na `i32`, reference `i32` musí přežít `Borrowed`.
#[derive(Debug)]
struct Borrowed<'a>(&'a i32);

// Enum, který je buď `i32`, nebo referencí na něj.
#[derive(Debug)]
enum Either<'a> {
    Num(i32),
    Ref(&'a i32),
}

fn main() {
    let x = 18;
    let y = 15;

    let single    = Borrowed(&x);
    let reference = Either::Ref(&x);
    let number    = Either::Num(y);

    println!("x is borrowed in {:?}", single);
    println!("x is borrowed in {:?}", reference);
    println!("y is *not* borrowed in {:?}", number);
}
```

Lifetime elision

Pro běžné příklady určuje lifetime sám překladač dle následujících pravidel:

Pravidlo pro životnost vstupních parametrů

Každý vstupní parametr dostává vlastní lifetime.

Pravidlo pro životnost výstupních parametrů

Pokud má funkce jeden vstupní parametr, všechny výstupy mají stejný lifetime.

Pravidlo pro metody s parametrem self

Pokud má metoda vstupní parametr `&self`, všechny výstupní parametry mají stejný lifetime.

'static

Dává životnost po celý běh programu, hodnota je uložena přímo v binárce programu, a je tedy vždy přístupná.

Hodí se například pro texty chybových hlášek.

Jinak se mu snažíme vyhnout.

Trait

Traity

Zjednodušeně můžeme **trait** považovat za **interface** v jiných programovacích jazycích.

Definujeme pomocí nich společnou funkcionalitu.

Implementace traitu

```
trait Summary {
    fn summarize(&self) -> String;
}

struct NewsArticle {
    headline: String,
    location: String,
    author: String,
    content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author, self.location)
    }
}
```

Implementace traitu pro druhou strukturu

```
trait Summary {
    fn summarize(&self) -> String;
}

struct Tweet {
    username: String,
    content: String,
    reply: bool,
    retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

Výchozí implementace

```
trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}
```

Využití jiných metod ve výchozí implementaci

```
trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        format!("(Read more from {}...)", self.summarize_author())
    }
}
```

Trait jako parametr / návratová hodnota

```
fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

```
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from("of course, as you probably already know, people..."),
        reply: false,
        retweet: false,
    }
}
```

Trait Bound

Syntax `impl Trait` u parametru je syntaktický cukr pro delší zápis, kterému se říká **trait bound**.
Následující bloky kódu jsou ekvivalentní, jen je zápis pomocí trait bound delší a hůře čitelný. Proto v základu doporučujeme používat `impl Trait`.

```
fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

```
fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

Více traitů

```
pub fn notify(item: &(impl Summary + Display)) {
    // ...
}
```

Zápis více traitů pomocí `where`

Použijeme pro situace, kde máme více parametrů s různými kombinacemi traitů.

```
fn some_function<T, U>(t: &T, u: &U) -> i32
where T: Display + Clone,
      U: Clone + Debug
{
    // ...
}
```

Dynamický trait

```
use std::io::Write;

fn say_hello(out: &mut dyn Write) -> std::io::Result<()> {
    out.write_all(b"hello world\n")?;
    out.flush()
}
```

Trait Object

- `dyn Write` představuje jednu variantu polymorfismu, které říkáme **trait object**.
- Slouží k provedení volání přes virtuální tabulku (*vtable*).
- Trait object nemůže být použit jako typ proměnné, reference na něj ale ano.
- Trait object není známý v době překladu, proto obsahuje další informace o typu referenta.
- Rust umožní konverzi `Box<File>` na `Box<dyn Write>`.

Reference na trait object

V jazyce Java je proměnná typu `OutputStream` referencí na libovolný objekt, který implementuje `OutputStream`. Skutečnost, že se jedná o referenci, je samozřejmá.

Obdobně v Rustu je proměnná typu `&dyn Write` referencí na hodnotu, která musí implementovat trait `Write`.

```
let mut buf: Vec<u8> = vec![];
let writer: &mut dyn Write = &mut buf;
```

Lifetime traitu

```rust
// A struct with annotation of lifetimes.
#[derive(Debug)]
struct Borrowed<'a> {
    x: &'a i32,
}

// Annotate lifetimes to impl.
impl<'a> Default for Borrowed<'a> {
    fn default() -> Self {
        Self {
            x: &10,
        }
    }
}

fn main() {
    let b: Borrowed = Default::default();
    println!("b is {:?}", b);
}
```

Subtrait

- Můžeme vytvořit subtrait, který vyžaduje i implementaci nadřazeného traitu.
- Řekneme, že `Creature` je **extension** `Visible`:

```
trait Creature: Visible {
    fn position(&self) -> (i32, i32);
    fn facing(&self) -> Direction;
    // ...
}
```

- Na pořadí implementace nezáleží:

```
impl Creature for Broom { /* ... */ }
impl Visible for Broom { /* ... */ }
```

Ošetření chybových stavů

Typy chyb

1. Chyby, ze kterých se můžeme zotavit.
2. Chyby, po kterých to můžeme zabalit.

Panika

```rust
fn main() {
    panic!("crash and burn");
}
```

Poznámka: v Rustu při panice program sám projde stack a uklidí po sobě veškerá data. Je to za cenu větší binárky. Pokud chcete snížit velikost binárky a nechat úklid na operačním systému, tak můžete udělat následující konfiguraci v `Cargo.toml`:

```toml
[profile.release]
panic = 'abort'
```

Enum `Result`

Funkce, kde může nastat chyba, vrací `Result`. Z něho můžeme vyčíst výsledek, nebo chybu.

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Příklad ošetření chybového stavu

Pokud chceme otevřít soubor, tak může dojít k chybě.
Ošetřit ji můžeme následně:

```
use std::fs::File;

fn main() {
    let file: Result<File, io::Error> = File::open("hello.txt");

    let file: File = match file {
        Ok(value) => value,
        Err(error) => panic!("Problem opening the 'hello.txt' file: {:?}", error),
    };
}
```

Ošetření dílčích chyb

```rust
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let file = File::open("hello.txt");

    let file = match file {
        Ok(ok_file) => ok_file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(created_file) => created_file,
                Err(e) => panic!("Cannot open nor create the file: {:?}", e),
            },
            other_error => {
                panic!("Cannot open the file: {:?}", other_error)
            }
        },
    };
}
```

Vlastní zpracování chyby pomocí `&dyn Error`

```
use std::error::Error;
use std::io::{Write, stderr};

fn print_error(mut err: &dyn Error) {
    let _ = eprintln!("error: {}", err);

    while let Some(source) = err.source() {
        let _ = eprintln!("caused by: {}", source);
        err = source;
    }
}
```

Zpanikaření v případě chyby

```rust
use std::fs::File;

fn main() {
    // Při chybě ukončí program s obecnou chybovou zprávou
    let file = File::open("hello.txt").unwrap();

    // Při chybě ukončí program s naší vlastní chybovou hláškou
    let file = File::open("hello.txt").expect("hello.txt se nepovedlo otevřít");
}
```

Propagace chyb

Operátor `?` použijeme za hodnotou typu `Result`.
Pokud je hodnota `Ok(value)`, výsledkem operace je `value`.
Pokud je hodnota `Err(error)`, funkce skončí s `Err(error)`.

```
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let mut s = String::new();
    File::open("hello.txt")?.read_to_string(&mut s)?;

    Ok(s)
}

fn main() -> Result<(), io::Error> {
    match read_username_from_file() {
        Ok(name) => {
            println!("Found name {name}.");
            Ok(())
        },
        Err(e) => {
            eprintln!("Could not read name!");
            Err(e)
        }
    }
}
```

Extrémní propagace chyb

Source

Práce s chybami různých typů

```
type GenericError = Box<dyn std::error::Error + Send + Sync + 'static>;
type GenericResult<T> = Result<T, GenericError>;
```

`anyhow` crate

Nejpopulárnější knihovna pro zjednodušení práce s chybami.

Je doporučená pro aplikace, pro knihovny doporučujeme se podívat na `thiserror`.

```toml
[dependencies]
anyhow = "1"
```

Práce s chybami pomocí anyhow

Návratový typ

```
fn get_cluster_info() -> anyhow::Result<ClusterMap> {
    let config = std::fs::read_to_string("cluster.json")?;
    let map: ClusterMap = serde_json::from_str(&config)?;
    Ok(map)
}
```

One-of errors

```
return Err(anyhow!("Missing attribute: {}", missing));
```

Context

```
std::fs::read(path).with_context(|| format!("Failed to read from {}", path))?;
```

Práce s CLI

Argumenty příkazové řádky

```
let pattern = std::env::args().nth(1).expect("No pattern given.");
let path = std::env::args().nth(2).expect("No path given.");
```

Uložení argumentů do struktury

```
struct Cli {
    pattern: String,
    path: std::path::PathBuf,
}

fn main() {
    let pattern = std::env::args().nth(1).expect("No pattern given.");
    let path = std::env::args().nth(2).expect("No path given.");

    let args = Cli {
        pattern: pattern,
        path: std::path::PathBuf::from(path),
    };
}
```

`clap` crate

Nejpopulárnější knihovna na zpracování argumentů z CLI.

Závislost se liší podle toho, jestli budeme používat `derive` pattern:

```
[dependencies]
clap = { version = "4.4.6", features = ["derive"] }
```

nebo `builder` pattern:

```
[dependencies]
clap = "4.4.6"
```

Jednodušší zpracování přes clap

```rust
use clap::{ArgAction, Parser};

/// This doc string acts as a help message when the user runs '--help'.
/// The same applies for all doc strings on struct fields.
#[derive(Parser)]
#[clap(version = "1.0", author = "John Smith")]
struct Args {
    /// Sets a custom config file. Could have been an Option<T> with no default too
    #[clap(short = 'c', long = "config", default_value = "default.conf")]
    config: String,

    /// Some input. Because this isn't an Option<T> it is required to be used
    input: String,

    /// A level of verbosity, can be used multiple times
    #[clap(short = 'v', long = "verbose", action = ArgAction::Count)]
    verbose: u8,
}

// Continued on the next slide...
```

Jednodušší zpracování přes clap

```
// ...continued from the previous slide.

fn main() {
    let args: Args = Args::parse();

    println!("Value for config: {}", args.config);
    println!("Using input file: {}", args.input);

    // Vary the output based on how many times the user used the "verbose" flag
    // (i.e. 'myprog -v -v -v' or 'myprog -vvv' vs 'myprog -v')
    match args.verbose {
        0 => println!("No verbose info"),
        1 => println!("Some verbose info"),
        2 => println!("Tons of verbose info"),
        3 | _ => println!("Don't be crazy"),
    }
}
```

Zpracování přes builder pattern

```rust
use clap::{arg, Arg, ArgAction, Command};

fn main() {
    let matches = Command::new("myapp")
        .version("1.0")
        .author("John Smith")
        .about("Does awesome things")
        .arg(arg!(-c --config [FILE] "Sets an optional custom config file").default_value("default.conf"))
        .arg(arg!(<INPUT>            "Sets the required input file to use"))
        .arg(Arg::new("verbosity").short('v').long("verbose").action(ArgAction::Count))
        .get_matches();

    if let Some(i) = matches.get_one::<String>("INPUT") {
        println!("Value for input: {}", i);
    }
    if let Some(c) = matches.get_one::<String>("config") {
        println!("Value for config: {}", c);
    }
    match matches.get_count("verbosity") {
        0 => println!("No verbose info"),
        1 => println!("Some verbose info"),
        2 => println!("Tons of verbose info"),
        _ => println!("Don't be crazy"),
    }
}
```

Práce se soubory a stdin

Standardní vstup

```rust
use std::io;

fn main() {
    let mut input = String::new();

    match io::stdin().read_line(&mut input) {
        Ok(n) => {
            println!("{} bytes read", n);
            println!("{}", input);
        }
        Err(error) => println!("error: {}", error),
    }
}
```

Standardní vstup

```
use std::io;

fn main() {
    let lines = io::stdin().lock().lines();

    for line in lines {
        println!("got a line: {}", line.unwrap());
    }
}
```

Poznámka: `.lock()` vytvoří nad `stdin` *mutex*, kterým se budeme věnovat více v šesté přednášce.

Vytvoření souboru a zápis

```rust
use std::fs::File;
use std::io::prelude::*; // Importuje běžně používané traity

fn main() -> std::io::Result<()> {
    let mut file = File::create("foo.txt")?;

    file.write_all(b"Hello, world!")?;

    Ok(())
}
```

Načtení obsahu ze souboru

```rust
use std::fs::File;
use std::io::prelude::*;

fn main() -> std::io::Result<()> {
    let mut file = File::open("foo.txt")?;
    let mut contents = String::new();

    file.read_to_string(&mut contents)?;
    assert_eq!(contents, "Hello, world!");

    Ok(())
}
```

Práce přes buffer

```rust
use std::fs::File;
use std::io::BufReader;
use std::io::prelude::*;

fn main() -> std::io::Result<()> {
    let file = File::open("foo.txt")?;
    let mut buf_reader = BufReader::new(file);
    let mut contents = String::new();

    buf_reader.read_to_string(&mut contents)?;
    assert_eq!(contents, "Hello, world!");

    Ok(())
}
```

Načtení řádky

```rust
use std::fs::File;
use std::io::{self, prelude::*, BufReader};

fn main() -> io::Result<()> {
    let file = File::open("foo.txt")?;
    let reader = BufReader::new(file);

    for line in reader.lines() {
        println!("{}", line?);
    }

    Ok(())
}
```

Synchronizace na disk

Použijeme, pokud chceme počkat (*blokující volání*), dokud systém nedokončí synchronizaci souborového systému.

```
use std::fs::File;
use std::io::prelude::*;

fn main() -> std::io::Result<()> {
    let mut file = File::create("foo.txt")?;

    file.write_all(b"Hello, world!")?;
    file.sync_all()?;

    Ok(())
}
```

Flush bufferu

Vynucení zápisu z bufferu, Rust jej volá i v metodě traitu `Drop`.

```
use std::io::prelude::*;
use std::io::BufWriter;
use std::fs::File;

fn main() -> std::io::Result<()> {
    let mut buffer = BufWriter::new(File::create("foo.txt")?);

    buffer.write_all(b"some bytes")?;
    buffer.flush()?;

    Ok(())
}
```

Poznámka: `File::flush` nic nedělá a jen vrací `Ok(())`.

Zápis přes buffer

```
use std::fs::File;
use std::io::{BufWriter, Write};

fn main() {
    let data = "Some data!";
    let file = File::create("/tmp/foo").expect("Unable to create file");
    let mut file = BufWriter::new(f);

    file.write_all(data.as_bytes()).expect("Unable to write data");
}
```

Poznámka: dočasný adresář by bylo lepší zjistit nezávisle na platformě pomocí `env::temp_dir()`.

Shrnutí

1. Týmové projekty
2. Enumy
3. Pattern Matching
4. Struktury
5. Lifetimes
6. Traity
7. Ošetření chyb
8. CLI aplikace
9. Práce se soubory a stdin

Dotazy?

Děkuji za pozornost