Gymnázium Brno, třída Kapitána Jaroše, p. o.

Programovací jazyk Rust jako náhrada za C++

Závěrečná práce

Vedoucí práce:
Mgr. Marek Blaha

Adam Hrnčárek

Brno 2023

Poděkování

Chtěl bych poděkovat Rust komunitě za skvělé učební materiály, díky kterým Rustu rozumím. Také bych chtěl poděkovat YouTuberovi TheCherno, jehož C++ série mě naučila všechny koncepty uvedené v této práci. Na závěr bych rád poděkoval svému profesoru informatiky a vedoucímu práce Marku Blahovi.

Čestné prohlášení

Prohlašuji, že jsem práci **Programovací jazyk Rust jako náhrada za C++** vypracoval samostatně a veškeré použité prameny a informace uvádím v seznamu použité literatury. Souhlasím, aby moje práce byla zveřejněna v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách ve znění pozdějších předpisů a v souladu s platnou Směrnicí o zveřejňování závěrečných prací.

V Brně dne 13. června 2023 .....

podpis

Abstract

This thesis introduces the Rust programming language and some of it's advantages compared to C++. The thesis contains examples of Rust syntax and assumes no previous knowledge of the language.

Specifically, this thesis focues on major changes in the style of programming (working with strings and errors, object oriented programming).

Key words: The Rust Programming Language, C++, language comparison, object oriented programming, working with strings, software reliability and safety

Abstrakt

Tato závěrečná práce představuje programovací jazyk Rust a některé jeho výhody oproti jazyku C++. Práce obsahuje ukázky syntaxu a nepředpokládá žádnou předchozí znalost Rustu.

Specificky se práce soustředí na zásadnější změny ve stylu programování (práce s řetězci, zpracování chyb, objektově orientované programování).

Klíčová slova: Programovací Jazyk Rust, C++, porovnání jazyků, objektově orientované programování, práce s řetězci, softwarová spolehlivost a bezpečnost

Obsah

Seznam obrázků

1 Úvod a cíl práce

1.1 Úvod do problematiky

Při vývoji aplikací je nutné si vybrat programovací jazyk, který budeme na vývoj používat. Pokud nám záleží na tom, aby byl výsledný produkt co nejrychlejší, je historicky naše volba jasná: C++. Jazyk C++ je ale hodně komplikovaný a nemá jednoduchá řešení pro některé problémy moderních vývojářů.

Jedním z těchto problémů je bezpečnost při paralelním programování. Právě proto vznikl Rust. Tento poměrně nový jazyk s sebou přináší velké množství inovací, přesto se ale jedná o jazyk s rychlostí podobnou C++.

V posledních letech začíná Rust vytlačovat C++ v mnoha oblastech vývoje, především díky svým bezpečnostním zásadám.

1.2 Cíl práce

Cílem této práce je představit Rust běžnému uživateli C++ a představit některé jeho výhody oproti C++[1]. Práce se bude věnovat hlavně:

- syntaxi a obvyklým návrhovým vzorům
- objektově orientovanému programování
- práci s řetězci
- bezpečnosti programů
- zpracování chyb
- paralelnímu programování

Cílem práce *není* porovnávání rychlosti C++ a Rustu, jelikož jsou zhruba stejně rychlé.

1.3 Předpoklady

Je předpokládána znalost jazyka C++, základních konceptů programování pro více vláken, objektově orientovaného programování.

[1] Nikoliv tedy oproti jazykům, jako je například TypeScript. Ty sice Rust také nahrazuje, ale z důvodu své rychlosti, ne jiných výhod.

2 Základní syntax

Základní syntax jazyka Rust se tolik neliší od jazyka C++. Klíčová slova `for`, `while`, `if` apod. jsou stejná. V této kapitole shrnu pouze základní informace, nebudu zacházet do hloubky. Rust syntax ale nevyužívá přetěžování operátorů do takové míry jako C++, nemělo by se tedy stát, že v příkladech bude něco příliš nečekaného (jako například spojování iterátorů pomocí operátoru `|` v C++).

2.1 Proměnné

Na rozdíl od deklarování proměnných jejich typem (popřípadě slovem `auto`) používá Rust klíčové slovo `let`. *Proměnné v Rustu jsou defaultně konstantní*, aby šla hodnota proměnné měnit, musíme v deklaraci přidat klíčové slovo `mut`.

Rust má i klíčové slovo `const`, které ale slouží stejnému účelu jako `constexpr` v jazyce C++ (deklarování compile-time konstanty). Pokud chceme vytvořit statickou proměnnou (=globální), musíme použít klíčové slovo `static`, popřípadě `static mut`. Jedním slovem není možné deklarovat více proměnných.

Většinou je vhodné uvést při deklaraci i počáteční hodnotu. U `let` se ale nejedná o chybu při kompilaci (chyba se ale objeví, pokud použijeme neinicializovanou proměnnou). Hodnotu přiřazujeme klasickým použitím znaku `=`.

Pokud jsme pomocí `let` nadefinovali proměnnou, můžeme opětovným použitím `let` nadefinovat *jinou proměnnou* se stejným názvem. Tomuto se říká stínování (angl. shadowing). (The Book, sekce 3.1)

V Rustu často není potřeba udávat typ vytvářené proměnné – velmi dobře si umí typy proměnných odvodit z hodnot, popřípadě jejich dalšího používání. Toto ale platí pouze pro deklarace pomocí slova `let` – u ostatních je typová anotace povinná. Pokud si kompilátor neumí typ odvodit, je nutné typ explicitně uvést. Toho docílíme pomocí dnes častějšího „dvojtečkového syntaxu“. (The Book, sekce 3.2)

Celá deklarace proměnné tedy vypadá takto:

```
let mut variable_name: i32 = 5;
```

Obr. 1: Ukázka deklarace nekonstantní proměnné typu `i32` (`int` v C++)

2.2 Funkce

Stejně jako u proměnných nezačínají deklarace funkcí jejich návratovým typem, ale klíčovým slovem `fn`. Následuje syntax velmi podobný C++, až na „dvojtečkový syntax“ deklarace typů. Za uzavírací závorkou následuje šipka (`->`) a za ní návratový typ. Pokud funkce nevrací nic (typ `void` v C++), můžeme za šipku napsat „prázdný typ“ `()` nebo šipku vůbec nepsat.

V Rustu je povinné zároveň s deklarací uvést i definici (tělo) funkce[2] To je obsaženo ve složených závorkách, stejně jako v C++.

Jednou z méně intuitivních věcí pro začátečníky jsou takzvané „implicit returns“. Pokud nedáme na konec výrazu středník, bude tento výraz vrácen jako hodnota. Toto se používá jako vrácení úplně na posledním řádku funkce. Jinak se používá klíčové slovo `return`, stejně jako v C++. (The Book, sekce 3.3)

Takto by vypadala definice funkce, která sečte dvě čísla a vrátí výsledek:

```
fn sum(first: &i32, second: &i32) -> i32 {
    first + second // implicit return
}
```

Obr. 2: Ukázka definice funkce

2.3 Pattern matching

Rust má velmi silný pattern matching. Pomocí něj je možné „vytahovat“ hodnoty z enumů, datových struktur a n-tic (anglicky tuple). Vždy je nutné použít klíčové slovo `let`. (The Book, sekce 18.1)

```
let (number, text) = (12, "Hello world!");
```

Obr. 3: Ukázka jednoduchého pattern matchingu

Po tomto řádku můžeme používat proměnnou `number`, která je typu `i32` a proměnnou `text` typu `&str` (práci se stringy je věnována vlastní kapitola).

Rust má také klíčové slovo `match`:

```
let is_two_or_three: bool = match number {
    2 | 3 => true,
    12 => false,
    _ => false,
};
```

Obr. 4: Ukázka pattern matchingu pomocí `match`

V tomto případě používáme `match` na proměnnou `number`. Pokud je hodnota 2 nebo 3, vrátí `match` hodnotu `true`. Znak `_` znamená „vše ostatní“, rameno s hodnotou 12 je tedy zbytečné. Výsledek se zde uloží do proměnné, v jednotlivých ramenech (anglicky match arms) lze ale napsat libovolný kód. Pokud `match` nepočítá s každou možností, kód se nezkompiluje. (The Rust Reference, sekce 8.2.16)

V tomto případě se jedná o zbytečnou komplikaci kódu. `match` a pattern matching obecně je ale velmi užitečný pří implementování komplexnější logiky.

[2] Pokud se tedy nejedná o deklaraci funkce v Trait definici nebo v `extern` bloku.

2.4 Smyčky a kontrolní bloky

Smyčky a kontrolní bloky shodné s C++

Klíčová slova `while` a `if`, `else if`, `else` fungují stejně jako v jazyce C++. Jediným rozdílem je, že závorky kolem výrazů za klíčovými slovy nejsou povinné (a nepoužívají se).

Dalším detailem je využitelnost pattern matchingu. Ke zjištění struktury můžeme použít slova `if let` a `while let`. Pokud se match podaří, výraz vrátí `true` a proměnná je k dispozici k použití v daném bloku. Toto se často vyskytuje při práci s možnými chybami. Následuje ukázka použití `if let` bloku. V proměnné `calculation_result` je uložen enum `Option<i32>` – jedná se o návratovou hodnotu nějaké funkce, která *může* (ale nemusí) vrátit výsledek. Pokud výsledek existuje (proměnná má hodnotu `Some`), vypíšeme jej do konzole.

```
let calculation_result = Some(5);
if let Some(num) = calculation_result {
    // print result to standard output
    println!("Success! The result is {}", num);
} else {
    println!("Calculation not successful");
}
```

Obr. 5: Ukázka `if let` výrazu

Smyčka for

Smyčka `for` funguje jinak než v C++. Používá se na cyklení přes tzv. iterátory. Iterátory jako takové jsou návrhovým vzorem, který se částečně vyskytuje i v C++. V Rustu jsou ale daleko mocnější – jedná se o nejpoužívanější návrhový vzor tohoto jazyka. „Klasický“ for loop přes čísla můžeme napsat pomocí tzv. rozsahu (anglicky range). Jeho syntax je následující:

```
for i in 0..10 {
    println!("{}", i);
}
```

Obr. 6: Ukázka for smyčky přes Range objekt

Tato smyčka vypíše celá čísla od 0 do 9 včetně. Pokud bychom chtěli vypsat i hodnotu 10, museli bychom range napsat takto: `0..=10`. Můžeme si všimnout, že proměnná `i` se postupně nastaví na každou hodnotu iterátoru. Lze zde použít i pattern matching. (The Rust Reference, sekce 8.2.13)

Smyčka loop

V Rustu se nachází ještě jeden typ smyčky. Jedná se o nekonečnou smyčku, vytvořenou pomocí klíčového slova `loop`. Funguje stejně jako `while (true)` v C++.

2.5 Makra

Makra se syntakticky liší od C++. Mají více typů, nejčastěji se ale používají tzv. function-like makra. Volání těchto maker vypadá stejně jako volání funkci, akorát je za názvem znak `!`. Tato makra, na rozdíl od funkcí, mohou brát libovolný počet parametrů. Asi nejčastěji používané makro je `println!`, které řeší tisknutí do konzole bez toho, aniž bychom se museli starat o konvertování proměnných na řetězec.

Existují také derivační a atributová makra. Derivační se používají při OOP, atributová přidávají nějaký compile-time atribut k symbolu (funkci, proměnné,...). (The Book, sekce 19.5)

Deklarace maker je složitější, protože pracují přímo s abstraktním syntaktickým stromem svého vstupu a modifikovaný strom vrací. Do vytváření maker tato práce nezachází.

2.6 Importování

Používání symbolů (funkcí, proměnných, objektů, ...) je mnohem jednodušší než v C++. Rust nepotřebuje znát symbol před jeho použitím (jde tedy například použít funkci, i když je její definice až níž v souboru). Pro importování symbolu nám stačí použít klíčové slovo `use`, za kterým napíšeme cestu k symbolu pomocí namespace operátorů `::`. Nepotřebujeme žádné hlavičkové soubory, stačí nám pouze soubory s kódem. (Rust STD, keyword.use) Při externím linkování se ale deklarováním symbolů a funkcí bez těla nevyhneme. (Rust STD, keyword.extern)

V následujícím příkladu importujeme typ `LinkedList` z modulu `collections` knihovny `std`. Dále voláme statickou metodu `new` tohoto typu. Můžeme si také všimnout, že nikde neuvádíme typ proměnné. Ukládáme do ní „nový“ spojový seznam – hlavní typ je tedy `LinkedList`, ale neuvádíme ani v něm ukládaný typ. Je totiž odvozen z toho, že později ukládáme do listu hodnotu typu `i32`. Plný typ proměnné `list` je tedy `LinkedList<i32>`.

```
use std::collections::LinkedList;

fn main() {
    let mut list = LinkedList::new();
    list.push_back(0);
}
```

Obr. 7: Ukázka importování a používání namespace operátoru

2.7 Typy

Názvy většiny základních typů se liší od těch v C++. Jediným stejným typem je `bool`.

Čísla

Rust nemá typy `short`, `long` apod. Místo toho má sadu typů začínající na `i`. Dále následuje velikost typu v bitech od 8 až po 128. Typy celých čísel tedy jsou: `i8`, `i16`, `i32`, `i64` a `i128`. Pokud nám jde pouze o kladná čísla (`unsigned`), stačí nahradit `i` za `u`. Například tedy `u32`.

Výhodou tohoto způsobu je, že jsme si vždy vědomi jejich velikosti (například typ `int` může mít různou velikost podle platformy). Jejich ekvivalentem by tedy byly spíš typy jako `int32_t`, dostupné od C++11.

Pro desetinná čísla existují typy `f32` a `f64`. Čísla opět reprezentují jejich velikost v paměti. (The Book, sekce 3.2)

Pole

Pole jsou podobná objektu `std::array`. Jejich typ je `[T; N]`, kde T je typ ukládaný v poli a N je délka pole. Když pole vytváříme, musíme vždy uvést počáteční hodnotu. Do pole můžeme indexovat stejně jako v C++ pomocí `[]`. (The Book, sekce 3.2)

Rust má také dynamické pole. Jeho název je z `std::vector` zkrácen na `Vec`. Vytvářet vektory můžeme buď voláním jednoho z konstruktorů, nebo voláním makra `vec![]`, které bere počáteční hodnoty. (The Book, sekce 8.1)

```
let mut array: [i32; 3] = [0; 3];
array[0] = 1;
let vector1: Vec<i32> = Vec::new();
let vector2: Vec<i32> = Vec::with_capacity(10);
let mut vector = vec![1, 2, 3];
vector.push(4);
```

Obr. 8: Ukázka vytváření pole a vektoru

V ukázce nastavujeme hodnotu proměnné `array` na `[0; 3]`, což je pole délky 3 vyplněné nulami.

Samotný typ u proměnné `array` by šlo vynechat (byl by implicitní). U vektorů je ale *nutné* uvést typy, protože z konstruktoru nejde poznat, jaký typ do vektoru ukládáme. Kdybychom je ale dále nějak používali, nebylo by nutné typ uvádět.

Reference

Když chceme předat hodnotu proměnné bez jejího kopírování nebo umožnit jiné části programu přepsat její hodnotu, použijeme reference. Rust sice má i ukazatele, ale kvůli

problémům s bezpečností se prakticky nepoužívají (jdou plně používat pouze v tzv. unsafe blocích).

Stejně jako u proměnných máme dva typy referencí – `&T` a `&mut T` (kde T je typ proměné). `&T` nám neumožňuje jakkoliv modifikovat proměnnou na kterou ukazuje. Když chceme poslat referenci, musíme to explicitně uvést (viz ukázka níže). Reference umožňující modifikaci jdou dělat pouze na modifikovatelné proměnné. (The Book, sekce 4.2)

Pokud chceme referenci na blok paměti, pošleme tzv. řez (anglicky slice). Jedná se o speciální objekt, který si pamatuje ukazatel na začátek a délku úseku. Jeho zápis je `&[T]` a `&mut [T]`. Získáme jej pomocí operátoru `[]`, ve kterém ale neuvedeme index, ale range. (The Book, sekce 4.3)

```
fn change_number(number: &mut u64) {
    *number = 0;
}

fn main() {
    let mut number = 5;
    change_number(&mut number);
    println!("{}", number); // 0
    let mut vector = vec![1, 2, 3];
    let number_slice: &mut [i32] = &mut vector[1..];
    number_slice[1] = 0;
    println!("{}", vector[2]); // 0
}
```

Obr. 9: Ukázka použití referencí a řezů

V ukázce vytváříme modifikovatelnou proměnnou `number`, posíláme její referenci funkci, ta změní její hodnotu na 0. Následně tiskneme hodnotu proměnné do `stdout`. Typ proměnné je `u64`. Explicitně jsme jej neuvedli, ale kompilátor si to odvodil z toho, že referenci na proměnnou posíláme funkci, která bere referenci na `u64`.

Dále vytváříme vektor o třech prvcích. Referenci na všechny *stávající*[3] prvky vektoru s indexem 1 a dále ukládáme do proměnné `number_slice`. Používáme `1..` – zprava nekonečný range (když neuvedeme jeden krajní prvek, bude range tím směrem zasahovat do nekonečna). Měníme druhý prvek range (tedy třetí prvek vektoru), ten následně tiskneme (pomocí vektoru, ne řezu).

[3] Kdybychom do vektoru poté přidali nový prvek, řez by měl stále délku 2.

Řetězce

Situace s řetězci je dost podobná C++. Existují dva typy: `&str` a `String`. Protože Rust používá UTF-8, velikost typu `char` jsou 4 byty. (Rust STD, primitive.char) Pokud chceme typ reprezentující jeden byte (a tedy i paměť do které se řetězce ukládají), použijeme `u8`. Pokud chceme používat ASCII, můžeme používat „byte string“ (`b""`). Jeho charaktery můžeme psát jako `b'a'` apod. (Rust By Example, sekce 19.3)

`&str` je reference na konstantní řetězec, podobně jako `const char *`. Na pozadí se jedná o `&[u8]`, kde je vždy uložena *validní* UTF-8 sekvence.

`String`, stejně jako `std::string` je dynamicky alokovaný řetězec. Funguje stejně jako v C++, akorát jeho buffer je `Vec<u8>` (tj. dynamické pole bytů) a jeho velikost je 24 bytů (nikoliv 32). (The Book, sekce 8.2)

Vytvářet řetězce můžeme více způsoby:

```
let hello1: &str = "Hello world!";
let hello2: String = "Hello world!".to_string();
let hello3: String = String::from("Hello world!");
```

Obr. 10: Ukázka vytváření řetězců

Vlastní typy

Podobně jako C++, Rust má klíčové slovo `struct`, které umožňuje vytvářet vlastní typy. Ve složených závorkách se nachází *pouze členské proměnné* oddělené čárkami. (The Book, sekce 5.1)

Příklad struktury `Person`:

```
struct Person {
    name: String,
    age: u8,
}
```

Obr. 11: Ukázka vytvoření struktury

Pokud chceme přidat metody, uděláme tak v tzv. `impl` bloku. Můžeme zde vytvářet funkce, které jako první parametr berou `self`, `&self` nebo `&mut self`. Abychom měli přístup k datům ve struktuře, musíme to udělat právě přes jednu z těchto možností. Pokud funkce nebere žádnou z nich, jedná se o „statickou metodu“[4]. Když použijeme přímo `self`, ztratí volající přístup k objektu (Rust implicitně nekopíruje, ale přesunuje hodnoty). (The Book, sekce 5.3)

Nemáme speciální syntax pro konstruktor. Místo toho existuje konvence, že každá struktura má statickou metodu `new`, která vrací vytvořenou strukturu.

[4] Všechny funkce nadefinované v `impl` bloku se nazývají *asociované funkce* (asociované s typem). Protože první parametr takové asociované funkce není `self`, nejedná se o metodu.

Příklad funkcí na struktuře `Person`:

```
impl Person {
    fn new(name: &str, age: u8) -> Person {
        Person {
            name: name.to_string(),
            age, // equivalent to `age: age`
        }
    }

    fn add_year(&mut self) {
        self.age += 1;
    }
}
```

Obr. 12: Ukázka `impl` bloku

Rust má také enumy. Oproti C++ ale mohou držet i další hodnoty, a to buď jako n-tici, nebo jako strukturu. Můžeme jim také přidávat asociované funkce pomocí `impl` bloků. (The Book, sekce 6.1) Příklad na obr. 13.

2.8 Generiky

Na spoustě míst lze použít generiky – kompilátor dosadí přesný typ až při kompilaci, stejně jako v C++. Používání generik je poněkud jednodušší (není potřeba psát `template<typename T>`, ale na správném místě stačí pouze `<T>`), zato ale častěji a komplikovaněji používané. Ke generikám můžeme (a většinou musíme) přidat i tzv. trait bound – typ T musí implementovat určitý trait (popř. jejich kombinaci), tj. musí mít určitou funkcionalitu. Více v kapitole o OOP.

2.9 Move sémantika

Když v C++ nebereme proměnou přes referenci, její hodnota se zkopíruje. V Rustu tomu tak není. Používá totiž tzv. move sémantiku, tj. hodnota se „přesune“ do volané funkce a dále k ní nejde přistupovat. Pokud se jedná o primitiv (například číslo), hodnota se kopíruje (protože kopie je triviální) – toto tedy platí pouze pro komplexnější typy. Viz obr. 14. (Rust By Example, sekce 15.2)

```rust
enum FileError {
    // An `enum` variant may either be `unit-like`,
    NotFound,
    // like tuple structs,
    TooBig(u64),
    // or c-like structures.
    DifferentOwner { user: String },
}

impl FileError {
    fn description(&self) -> String {
        match self {
            FileError::NotFound => "File not found".to_string(),
            FileError::TooBig(max_size) => {
                format!("File too big, max size is {}", max_size)
            }
            FileError::DifferentOwner { user } => {
                format!("Different owner: {}", user)
            }
        }
    }
}
```

Obr. 13: Ukázka enumů

```rust
let a = vec![1, 2, 3];
let b: Vec<i32> = a; // move `a` into `b`
for elem in a {
    println!("{}", elem);
}
```

```
C:\_Adam\zaverecka\rust\examples>cargo build
   Compiling samples v0.1.0 (C:\_Adam\zaverecka\rust\examples)
error[E0382]: use of moved value: `a`
   --> src\main.rs:151:21
    |
149 |         let a = vec![1, 2, 3];
    |             - move occurs because `a` has type `Vec<i32>`, which does not implement the `Copy` trait
150 |         let b: Vec<i32> = a; // move `a` into `b`
    |                           - value moved here
151 |         for elem in a {
    |                     ^ value used here after move
    |
help: consider cloning the value if the performance cost is acceptable
    |
150 |         let b: Vec<i32> = a.clone(); // move `a` into `b`
    |                            ++++++++

For more information about this error, try `rustc --explain E0382`.
error: could not compile `samples` (bin "samples") due to previous error
```

Obr. 14: Ukázka move sémantiky

3 OOP a Rust

Objektově orientované programování se hodně liší od C++. Podle některých definic OOP není Rust ani objektově orientovaným jazykem (The Book, kapitola 17).

Již jsme viděli, že je možné vytvořit strukturu a dát jí nějaké metody. Objekty tedy máme, ale jaké jsou vztahy mezi nimi? *Rust nemá objektovou dědičnost* – jeden struct nemůže dědit z jiného. Žádné klíčové slovo class ale neexistuje.

3.1 Trait systém

Rust odděluje funkcionalitu od dat. Data jsou definována přímo ve struktuře. Funkcionalita spojená pouze s daným typem je také definována přímo na něm. Když ale chceme nadefinovat sdílenou funkcionalitu mezi typy, použijeme klíčové slovo `trait`.

Trait je něco jako interface v dalších objektově orientovaných jazycích. Definuje asociované funkce (jejich hlavičky, případně i defaultní implementaci), nikoliv však proměnné. (The Book, sekce 10.2)

Výhodou je, že můžeme implementovat námi vytvořený trait pro cizí strukturu a obráceně. Když tedy potřebujeme sdílenou funkcionalitu u typu, který není náš, *můžeme mu ji přidat*. Například existuje trait `FromStr`, který definuje, jak máme z řetězce vytvořit daný typ. Když tedy chceme umět parsovat nějaký náš typ, stačí nám implementovat tento trait (nikoliv přetížením `std::basic_istream& operator>>`). (Rust STD, str/trait.FromStr) (C++ reference, io/basic_istream/operator_gtgt)

Díky tomuto také najdeme každou funkci, která má co dělat s daným typem, přímo jako její metodu (toto platí i u primitivních typů jako `i32`). Náš našeptávač je tedy o dost víc nápomocný než u C++.

Příklad vytvoření traitu a jeho implementace viz obr. 15

3.2 Trait bound a generiky

Pokud chceme v generické funkci použít nějakou funkcionalitu, musíme vědět, že typ ji má (program se jinak ani nezkompiluje). Toho docílíme tak, že na generický typ dáme tzv. trait bound. To znamená, že aby mohl tento typ být použit, musí implementovat daný trait (může jich být i víc). (The Book, sekce 10.2)

Tohoto lze docílit buď klíčovým slovem `where`, nebo dvojtečkovým syntaxem přímo v parametru. Pro každý pevně daný typ se vygeneruje *jiná* funkce, kde je za generiku dosazena pevná hodnota (The Book, sekce 10.1). Ukázka viz obr. 16

3.3 Trait objekty a dynamic dispatch

Generiky ale mají jeden problém: můžeme je použít, když máme hodnoty pouze daného typu. Co když ale máme kolekci ukazatelů na objekty, které všechny implementují daný trait[5]? Použijeme tzv. trait objekty.

[5] ukazatelů proto, že každý typ může mít jinou velikost

```rust
use std::f64::consts::PI;

trait Shape {
    fn area(&self) -> f64;
}

struct Rectangle {
    width: f64,
    height: f64,
}

struct Circle {
    radius: f64,
}

impl Shape for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
}

impl Shape for Circle {
    fn area(&self) -> f64 {
        self.radius * self.radius * PI
    }
}
```

Obr. 15: Ukázka používání traitu

Pokud používáme trait objekty, potřebujeme nějak zjistit, jakou implementaci máme zavolat. Tomuto se říká dynamic dispatch – dynamicky za běhu programu zjišťujeme, kde se nachází funkce, kterou máme zavolat. Proto také reference na trait objekty označujeme slovem `dyn`. (The Book, sekce 17.2) Příklad funkce ekvivalentní s generickými na obr. 17

3.4 Derivační makra

Některé traity nemusíme definovat manuálně, může to za nás udělat derivační makro. Stačí nám pouze nad definici struktury napsat `#[derive(TraitName)]` (do závorek můžeme napsat více názvů, které oddělíme čárkou). (The Book, sekce 5.2)

Tuto funkcionalitu nemá každý trait. Mezi ty, které ji mají patří například `Debug` (umožňující tisknutí pro debugování), `Clone` (hluboké kopírování), `PartialEq` (částečné porovnávání) a další. (The Book, Appendix C)

```
fn print_area_generic1<T: Shape>(shape: &T) {
    println!("This shape has an area of {:.2}", shape.area());
}

fn print_area_generic2<T>(shape: &T)
where
    T: Shape,
{
    println!("This shape has an area of {:.2}", shape.area());
}
```

Obr. 16: Ukázka generických funkcí s trait boundem

```
fn print_area_tobj(shape: &dyn Shape) {
    println!("This shape has an area of {:.2}", shape.area());
}
```

Obr. 17: Ukázka trait objektů

3.5 Nejpoužívanější traity

- `Display` – tisknutí ve formátovaných řetězcích
- `Debug` – tisknutí struktury objektu ve formátovaných řetězcích
- `Iterator` – všechny iterátory jej implementují, díky němu lze použít tzv. builder pattern na iterátorech
- `Error` – požadavky pro chyby (vypisovatelnost,...)
- `PartialEq` – částečné porovnání (`==`) (nezaručena ekvivalence)
- `PartialOrd` – částečné srovnávání (`< > <= >=`)
- `Copy` – kopírovací sémantika
- `Clone` – hluboká kopie (včetně obsahu ukazatelů), „duplikace“
- `Sized` – velikost dané proměnné je možné určit při kompilaci
- `Sync` – typy, jejichž reference je bezpečné sdílet mezi vlákny
- `Send` – bezpečnost posílání typů mezi vlákny
- `FromStr` – parsování hodnoty z řetězce
- `IntoIterator` – převedení na iterátor
- `From<T>` – převedení z hodnoty T
- `Into<T>` – prevedení *na* hodnotu T

Každý operátor má také vlastní trait. Díky tomu není potřeba žádný zvláštní syntax pro přetěžování operátorů – stačí pouze implementovat daný trait (např. `Add` pro sčítání). (The Book, sekce 19.2)

3.6 Generické implementace – automatické rozšiřování

Pomocí generik můžeme také implementovat trait pro každý typ splňující generiku. Toto je velmi užitečné, například knihovna `itertools` dělá právě to – rozšiřuje `Iterator` trait o další metody (Docs.rs, itertools, src/itertools/lib.rs). Také toho využívají traity `From` a `Into` – pokud implementujeme `From`, druhá implementace se vytvoří automaticky (pokud umíme vytvořit `T` z `E`, umíme konvertovat `E` na `T`) (Rust STD, convert/trait.From).

```
trait Average: Iterator<Item = usize> {
    fn average(mut self) -> f64
    where
        Self: Sized,
    {
        let mut sum = 0;
        let mut count: usize = 0;
        for value in self {
            sum += value;
            count += 1;
        }
        // will be NaN if the iterator is empty
        sum as f64 / count as f64
    }
}

impl<T: Iterator<Item = usize>> Average for T {}

fn main() {
    let v = vec![1, 2, 3];
    println!("{}", v.into_iter().average());
}
```

Obr. 18: Ukázka rozšíření funkcionality pomocí generických implementací

V ukázce definujeme tzv. supertrait `Average`. Supertrait je to proto, že aby jej nějaký typ implementoval, musí už implementovat trait `Iterator`, jehož asociovaný typ `Item` je `usize` (toto je řečeno za dvojtečkou po názvu supertraitu) – iteruje tedy prvky typu `usize`.

`Average` definuje jedinou funkci `average`, která konzumuje `self`. Protože nebereme `self` přes referenci, musíme vědět jeho velikost při kompilaci (proto `Sized`). Tato

funkce má defaultní implementaci, která spočítá průměr prvků v iterátoru. Ve funkci iterujeme `self` (je to uričtě iterátor), proto taky potřebujeme `self` měnit (a je tedy označené `mut`). Typ obou použitých proměnných je `usize`, musíme je tedy převést na `f64` pomocí klíčového slova `as`.

Na označeném řádku implementujeme `Average` pro všechny iterátory, jejichž prvek je `usize`. V `main` funkci tiskneme průměr čísel ve vektoru `v`. Typ proměnné `v` je `Vec<usize>` (kompilátor vidí, že jiná možnost není).

4 Práce s řetězci

Rust pracuje s UTF-8 řetězci, nikoliv s ASCII. Z tohoto plynou komplikace, jelikož ne všechny hodnoty jsou platné. Například pokud se pokusíme tisknout neplatnou UTF-8 sekvenci, program spadne.

4.1 UTF-8

Jeden kódový bod (znak) má 1–4 byty. Jeho velikost je dána pozicí první nuly v prvním bytu. Některé kódové body označují pouze diakritiku – je nutné spojit znaky do větších celků (angličtina jim říká „grapheme clusters“). Standardní knihovna tuto spojovací funkcionalitu nemá (kódovací tabulky jsou hodně velké). Pokud je tato funkcionalita žádaná, je nutné použít externí knihovnu. (The Book, sekce 8.2)

4.2 Indexování

Problém nastává, když indexujeme do řetězce. Mělo by se jednat o konstantní operaci, ale to není možné, když máme různě dlouhé znaky. Mohli bychom indexovat přímo do bytů, ale ne vše, co nám takové indexování vrátí, je validní znak. Proto *nejde číselně indexovat do řetězců.*

Můžeme ale indexovat pomocí range, což nám vrátí `&str`. *Pokud ale naše indexace nezačne a neskončí na hranicích kódových bodů, program spadne.* Implementace různých metod na řetězcích je tedy poněkud komplikovanější. Práci nám poněkud usnadňují iterátory, například `chars()` nebo `bytes()`. (The Book, sekce 8.2)

4.3 Formátování a spojování

Spojovat mnoho řetězců dohromady taky není jednoduché. Můžeme použít `+` operátor a přičíst `&str` k typu `String`, ale to může (spolu s konverzemi na řetězec) být náročné na vypsání. Proto používáme makro `format!()`. Formátování v Rustu má poměrně jednoduchý princip: kam chceme vložit hodnotu, tam dáme `"{}"`[6]. Toto použije implementaci traitu `Display`. Pokud chceme vypsat strukturu proměnné, můžeme napsat `"{:?}"`, což používá `Debug` trait (který je derivovatelný pomocí makra). (The Book, sekce 8.2)

Proměnné následně uvedeme za formátovací řetězec, podobně jako například u funkce `printf`. Alternativně můžeme název proměnné uvést přímo do složených závorek. Zde také můžeme uvést formátovací možnosti, například již dříve jsme použili `"{:.2}"`, což znamená, že zaokrouhlujeme na 2 desetinná místa.

[6] Uvozovky na začátku a na konci jsou pouze pro ilustrační funkci (jedná se o řetězec), když píšeme `{}` v řetězci, uvozovky nepíšeme.

4.4 Tisknutí do standardního výstupu

Můžeme použít makra `println!()` a `print!()`. První z maker je častěji používané, protože na konec tisknutého řetězce automaticky přidá nový řádek. Do maker dáváme formátovací řetězce. Existuje také makro `eprintln!()`, které tiskne do standardního erroru. Tato makra byla často i v předhozích kódových ukázkách. Syntax tisknutí je poměrně jednodušší než u streamů v C++, hlavně pro uživatele jiných jazyků jako například Python (f-stringy).

Následuje ukázka konceptů, o kterých tato kapitola pojednává. Na konci tiskneme proměnnou `greeting` po jednotlivých znacích a `debug` přímo přes formátovací řetězec. Pro ilustraci jsou proměnné doplněny o typy, nic by se ale jejich vynecháním nezměnilo.

```
let name: &str = ”John”;
let greeting: String = format!(”Hi {}!”, name); // Hi John!
let some_numbers: Vec<i32> = vec![1, 2, 3];
let debug: String = format!(”{some_numbers:?}”); // [1, 2, 3]
for c in greeting.chars() {
    // print char by char
    print!(”{c}”);
}
println!(); // end with newline
println!(”{debug}”);
```

Obr. 19: Ukázka používání řetězců

5 Bezpečnost programů

Hlavní důvod, proč Rust nahrazuje C++, je jeho lepší bezpečnost. C++ jako takové nepatří mezi nejbezpečnější jazyky – je poměrně jednoduché v něm vytvořit memory leak a celkově je zde hodně věcí, na které si musí programátor dávat pozor.

5.1 Paměť

Asi nejunikátnější z věcí, které Rust přináší, je tzv. *vlastnictví* (angl. ownership). Historicky se k udržování bezpečnosti paměti používaly dva různé způsoby: garbage collection (již nepotřebná paměť se „sama uklidí“) a manuální alokace/dealokace. Garbage collection je pomalejší, ale zato zaručeně bezpečný. Díky konceptu vlastnictví *Rust garantuje paměťovou bezpečnost* bez zpomalení garbage collectorem. (The Book, kapitola 4)

Pravidla vlastnictví

Tento koncept se řídí poměrně jednoduchými pravidly (The Book, sekce 4.1):

- Každá hodnota má *právě jednoho* vlastníka
- Když skončí životnost vlastníka, končí i životnost hodnoty (a tedy se hodnota bezpečně vyčistí/dealokuje apod.)

Toto také souvisí s principem RAII (Resource Acquisition Is Initialization), známým i z C++ (C++ reference, language/raii). Když tedy vznikne proměnná, vzniká zároveň s ní i hodnota (např. se alokuje na haldě).

Jelikož se jedná o zcela unikátní koncept, je nutné si zvyknout na jiný mentální model. Pojďme se znovu podívat na příklad move sémantiky (Obr. 14) z dřívější kapitoly, tentokrát z pohledu vlastnictví:

```
let a = vec![1, 2, 3];
let b: Vec<i32> = a; // move `a` into `b`
for elem in a {
    println!("{}", elem);
}
```

Obr. 20: Připomenutí ukázky move sémantiky

Ze začátku vlastní vektor (neboli hodnotu) proměnná `a`. Na dalším řádku ale *předáváme vlastnictví vektoru proměnné `b`*. Když tedy později chceme použít proměnnou `a`, nejde to, protože *`a` již vektor nevlastní*.

Půjčování

Vlastnictví představuje nový mentální model a vysvětluje move sémantiku, ale jak do toho zapadají reference? Zde přichází koncept *půjčování* (angl. borrowing). Jeho pravidla kontroluje část kompilátoru jménem *borrow checker*.

Když vytváříme referenci, „půjčujeme si hodnotu“. „Dívat“ na hodnotu se může libovolný počet proměnných, problém by ale nastal, kdyby někdo hodnotu měnil. Provádět změny proto může pouze jedna proměnná a to pouze v případě, že nikdo jiný nemá hodnotu půjčenou na prohlížení. Formálněji V *jakýkoliv čas jde mít buď jednu modifikovatelnou referenci (`&mut T`), nebo libovolný počet nemodifikovatelných referencí (`&T`).* (The Book, sekce 4.2)

```
let mut value = 5;
let reference = &value;
let mut mutable_reference = &mut value;
println!("{}", reference);
```

```
$ cargo build
   Compiling samples v0.1.0 (F:\_Adam\_Jaroska vyssi\Informatika\zaverecka\rust\examples)
error[E0502]: cannot borrow `value` as mutable because it is also borrowed as immutable
   --> src\main.rs:260:37
    |
259 |         let reference = &value;
    |                         ------ immutable borrow occurs here
260 |         let mut mutable_reference = &mut value;
    |                                     ^^^^^^^^^^ mutable borrow occurs here
261 |         println!("{}", reference);
    |                         --------- immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `samples` (bin "samples") due to previous error
```

Obr. 21: Ukázka půjčování

V ukázce vytváříme nejdřív konstantní referenci a potom modifikovatelnou referenci. Tiskneme ale pomocí první z nich – obě tedy existují, což je problém. Kdybychom prohodili pořadí vytváření referencí nebo tiskli pomocí modifikovatelné reference (nebo přímo proměnné s hodnotou), nejednalo by se o chybu.

Druhým pravidlem půjčování je, že *všechny refrence musí být validní*. Občas (například když funkce vrací referenci) jsou k zajištění tohoto pravidla nutné tzv. životnostní anotace – do těch ale nebudeme zabíhat.

Příklad 2. pravidla můžeme vidět na obr. 22. Zde měníme, na co reference ukazuje (používáme tedy reference více jako ukazatele v C++). Problémem ale je, že proměnná, na kterou se odkazujeme (kterou jsme si „půjčili“) již neexistuje, tedy je naše reference již nevalidní. Kdybychom referenci dále nepoužili, nejednalo by se o chybu.

```rust
let value = 5;
let mut reference: &u64 = &value;
{
    let another = 30;
    reference = &another;
}
println!("{}", reference);
```

```
$ cargo build
   Compiling samples v0.1.0 (F:\_Adam\_Jaroska vyssi\Informatika\zaverecka\rust\examples)
error[E0597]: `another` does not live long enough
   --> src\main.rs:270:25
    |
269 |             let another = 30;
    |                 ------- binding `another` declared here
270 |             reference = &another;
    |                         ^^^^^^^^ borrowed value does not live long enough
271 |         }
    |         - `another` dropped here while still borrowed
272 |         println!("{}", reference);
    |                        --------- borrow later used here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `samples` (bin "samples") due to previous error
```

Obr. 22: Ukázka nevalidních refrencí

Chytré ukazatele a halda

Zatím jsme (kromě vektorů) alokovali pouze na zásobníku. Jak ale alokujeme na haldě? Použijeme tzv. chytré ukazatele. Mohli bychom alokovat i manuálně, ale to by vyžadovalo `unsafe` bloky (o těch později) a bylo potenciálně nebezpečné. Proto v Rustu *nikdy* nealokujeme manuálně, vždy používáme chytré ukazatele.

Chytré ukazatele v rámci svého destruktoru dealokují i svá data (k modifikaci destruktoru manuálně implementujeme `Drop` trait). Nejčastěji používané chytré ukazatele jsou `Box<T>`, `Rc<T>`[7] a `Weak<T>`. Jejich ekvivalentem jsou `std::unique_ptr<T>`, `std::shared_ptr<T>` a `std::weak_ptr<T>`. Existují ale i další chytré ukazatele. (The Book, kapitola 15)

Důležitý ukazatel je `RefCell<T>`. Umožňuje nám totiž implementovat kód, který by jinak nebyl možný bez `unsafe` bloků. Umí dynamicky, během běhu programu, plnit úkol borrow checkeru. Používáme jej, když by kompilátoru nebylo jasné, že jsme splnili pravidla půjčování.

I s pomocí chytrých ukazatelů může vzniknout referenční cyklus a vytvořit memory leak!

Ukázka na obr. 23 by měla být jednoznačná. Pomocí `use` nejdříve importujeme `Rc<T>` a `Weak<T>`. `Box<T>` importovat nemusíme, protože je dostupný vždy. Vytváříme číslici 5 alokovanou na haldě. Do další proměnné ukládáme `Rc` se stejnou hodnotou jako

[7] Zkratka znamená „reference counted“

ta, na kterou ukazuje `Box`. Dále z `Rc` vytváříme jednu silnou a jednu slabou referenci. Na posledních řádcích vypisujeme počet referencí sdíleného ukazatele. Typy jsou přidány pouze pro pohodlí čtenáře, na funkčnosti programu nic nemění.

```
use std::rc::{Rc, Weak};

fn main() {
    let boxed_value: Box<i32> = Box::new(5);
    let rc_value: Rc<i32> = Rc::new(*boxed_value);
    let strong_reference: Rc<i32> = Rc::clone(&rc_value);
    let weak_reference: Weak<i32> = Rc::downgrade(&rc_value);
    println!("{}", Rc::strong_count(&rc_value)); // 2
    println!("{}", Rc::weak_count(&rc_value)); // 1
}
```

Obr. 23: UKázka chytrých ukazatelů

Unsafe bloky

S pamětí souvisí i klíčové slovo **`unsafe`**. V blocích označených tímto slovem můžeme, kromě klasického „safe“ Rustu, provádět následujících 5 věcí:

1. *dereferencovat ukazatele*
2. volat unsafe metody
3. *číst nebo modifikovat statickou měnitelnou proměnnou* (**`static mut`**)
4. implementovat unsafe trait
5. přistoupit k hodnotám unionu (abychom mohli interagovat s uniony jazyka C)

Unsafe kód se často nepoužívá, ale jedná se o způsob jak zvolnit pravidla Rustu, například pro vytvoření lepší API nebo zlepšení výkonu. Některé objekty ve standardní knihovně (třeba chytré ukazatele) unsafe bloky používají. (The Book, sekce 19.1)

5.2 Zpracování chyb

V C++ jsme zvyklí vrhat chybové objekty pomocí **`throw`** a chytat je pomocí **`catch`**. Tento přístup je špatně sledovatelný, jelikož bez pohledu na dokumentaci nevíme, která funkce hází jakou chybu. Také můžeme mít problémy, když chybu zachytí nějaká centrálnější funkce a nevíme, v jakém stavu je náš program. Rust proto má zcela jiný přístup k práci s chybami.

Nenávratné chyby, aneb panika!

Rust má více typů chyb (The Book, kapitola 9). První z nich jsou tzv. nenávratné chyby. Těmto chybám se říká panika (angl. panic). Vyvolat je můžeme explicitně pomocí makra `panic!` (do argumentu makra dáme formátový řetězec, který se zobrazí jako chybová hláška), nebo nějakou akcí, jako je například dělení nulou nebo špatná indexace do řetězce (a tedy vznik neplatné UTF-8 sekvence). (The Book, sekce 9.1) Panika se používá, když se nemáme jak vyrovnat s chybou.

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

```
$ cargo run --bin very_stable
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `target\debug\very_stable.exe`
thread 'main' panicked at 'crash and burn', src\bin\very_stable.rs:2:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
error: process didn't exit successfully: `target\debug\very_stable.exe` (exit code: 101)
```

Obr. 24: Ukázka paniky
Zdroj: (The Book, sekce 9.1)

Obnovitelné chyby

Druhým typem chyb jsou obnovitelné chyby. Funkce, ve kterých se může nějaká taková chyba stát nevracejí přímo návratový typ `T`, ale vracejí buď `Option<T>`, nebo `Result<T, E>`, kde `E` je typ chyby. Tyto dva typy jsou enumy. (The Book, sekce 9.2)

Enum `Option<T>` má dvě varianty: `Some(T)` a `None`. Používá se, když nevíme, zda funkce vrátí hodnotu, nebo ne. *Tento typ je také náhrada za hodnotu* `null`, *která v Rustu neexistuje.*

`Result<T, E>` má rovněž dvě varianty: `Ok(T)` a `Err(E)`. Tento typ použijeme v případě, že někde může nastat chyba, kterou chceme sdělit volajícímu. Typ `E` může být cokoliv – enum (například u I/O operací), řetězec,...

Práce s chybovými hodnotami

Pokud chceme získat typ T, musíme jej dostat z vrácené hodnoty. Díky tomu jsme, jakožto volající, *nuceni* něco s potenciální chybou udělat. Samozřejmě můžeme vždy použít `match`, ale díky velkému množství pomocných metod to většinou není potřeba. Jelikož jsou metody na obou typech funkčně ekvivalentní, budeme se dále zabírat pouze typem `Result`.

Asi nejlehčí, co můžeme udělat, je použít metodu `unwrap`. Tato metoda nám vrátí typ `T`. Pokud se ale jedná o chybovou variantu, spustí se panika a program spadne. Další metody, jako například `map` a `and_then` nám dovolují řetězit funkce vracející `Result`.

Pokud chceme hodnotu `T` nebo nějakou výchozí hodnotu, máme metodu `unwrap_or`. (Rust STD, result)

```
// match
let result: Result<i32, String> = Ok(200);
match &result {
    Ok(code) => println!("Code: {}", code),
    Err(error) => println!("Error: {}", error),
}
// unwrap
let code = result.unwrap();
println!("Code: {}", code);
// map
let mut mode: Result<i32, String> = Ok(4);
mode = mode.map(|c| c + 1);
// unwrap_or
let mode = mode.unwrap_or(-1);
```

Obr. 25: Ukázka zpracování chyb

Operátor ?

Protože často chceme ve funkci řešit pouze případ bez chyby a předat chybu volajícímu, vznikl operátor `?`. Pokud naše funkce vrací stejný typ, jako námi volaná funkce[8], můžeme za závorky přidat `?`. Tento otazník vrátí chybovou hodnotu, pokud ji dostane; jinak pokračuje dál. Výhodou je, že při programování funkce se staráme pouze o bezchybovou cestu. (The Book, sekce 9.2)

Otazník můžeme použít i v `main` funkci, pokud změníme její návratový typ z implicitního `()` (prázdný typ) na `Result<(), E>` a na jejím konci vrátíme `Ok(())`. Pokud v tomto případě `main` vrátí `Err` variantu, vypíše se.

Na obr. 26 můžeme vidět příklad používající `?`.

5.3 Paralelní programování

Paralelní programování v jazyce Rust se velmi liší od C++. Kód se totiž nezkompiluje, dokud není bezpečný. Rust této vlastnosti říká *fearless concurrency* (The Book, kapitola 16).

[8] ? ve skutečnosti umí provádět i lehké typové konverze

```rust
fn error_fn() -> Result<i32, String> {
    Ok(5)
}

fn do_stuff(num: i32) -> Result<i32, String> {
    let result = error_fn()?;
    if num == result {
        return Err(format!("{} is equal to {}", num, result));
    }
    Ok(result + num)
}

fn main() -> Result<(), String> {
    let code = do_stuff(5)?;
    println!("Code: {}", code);
    Ok(())
}
```

```
$ cargo run --bin complicated
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `target\debug\complicated.exe`
Error: "5 is equal to 5"
error: process didn't exit successfully: `target\debug\complicated.exe` (exit code: 1)
```

Obr. 26: Ukázka použití `?`

Syntax vytváření vláken

Vlákna se, podobně jako v C++, vytvářejí pomocí `thread::spawn`. Objekt, který funkce vrací má metodu `join`, která vrací `Result` (buď s návratovou hodnotou vlákna, nebo s chybou, se kterou vlákno spadlo).

Funkce `thread::spawn` bere parametr, který implementuje trait `FnOnce` (tedy je funkcí, kterou jde zavolat aspoň jednou). Buď zde můžeme napsat přímo název funkce, nebo můžeme vytvořit tzv. closure (anonymní funkci).

Syntax vytváření anonymní funkce je poměrně složitý. Mezi svislé čáry (|) napíšeme parametry. Za čárami následuje tělo, většinou ohraničené složenými závorkami (pokud se nejedná o jediný výraz). Při vytváření funkcí pro jiná vlákna před první svislou čáru píšeme klíčové slovo **`move`**, které říká, že všechny vnější proměnné, které anonymní funkce používá, se mají přesunout do funkce. Toto je nutné, protože jinak kompilátor nemůže poznat, zda jsou reference validní po celou dobu běhu vlákna. (The Book, sekce 13.1)

Na obr. 27 můžeme vidět ukázku vytvoření vlákna, které píše do standardního výstupu.

```rust
use std::thread;

fn main() {
    let value = 5;
    let handle = thread::spawn(move || {
        println!("Hello from a thread!");
        println!("Value: {}", value);
    });
    handle.join().unwrap();
    println!("Finished!");
}
```

Obr. 27: Ukázka vytvoření vlákna

Atomické proměnné

Rust má atomické typy jako `AtomicU32`, podobně jako C++ (`atomic_uint32_t`). Operace s těmito hodnotami berou parametr typu `Ordering`, jejichž hodnoty jsou stejné jako řazení z C++20 (`std::memory_order`) (Rust STD, sync/atomic).

Značící traity

Abychom mohli mezi vlákny posílat hodnoty, musejí implementovat `Send`. Toto zaručuje, že se více vláken nedostane k hodnotě, která na to není připravena. Například se jedná o ukazatel `Rc<T>`, který nepoužívá atomické proměnné pro udržování referenčních počtů. Pokud všechny části typu `T` implementují `Send`, implementuje jej i typ `T`. (Rust STD, marker/trait.Send)

Pro přístup z více vláken je potřeba trait `Sync`. Typ `T` implementuje `Sync`, pokud `&T` implementuje `Send`. (Rust STD, marker/trait.Sync)

Tyto dva traity umožňují kompilátoru zajistit bezpečnost jen pomocí generik.

Chytré ukazatele

Pokud chceme z více vláken modifikovat jednu proměnnou, musíme použít vzájemné vyloučení (angl. mutual exclusion, zkráceně mutex). Na rozdíl od C++ se ale nejedná o „hodnotu“, ale o chytrý ukazatel. Metoda `lock` pak vrací hodnotu[9]. Abychom ale mohli použít hodnotu `Mutex<T>`, musíme ji přesunout do vlákna. Když ji ale přesuneme, ztratíme možnost ji poslat do dalšího – je tedy potřeba jej zabalit do sdíleného ukazatele.

Tento ukazatel je `Arc<T>`[10]. Má stejnou funkcionalitu jako `Rc<T>`, pouze používá atomické proměnné k udržování počtů referencí. (The Book, sekce 16.3)

[9] ve skutečnosti vrací `Result`. O možnost `Err` se jedná tehdy, když nějaké vlákno spadlo, zatímco mělo vlastnictví zámku.

[10] atomically reference counted

Na následujícím obrázku je krátký program, který vytvoří 3 vlákna, z nichž každé přičte jedničku k hodnotě uložené v proměnné `data`. Kdybychom nepoužili `Arc<T>` a `Mutex<T>`, *kód by se nezkompiloval.*

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let mut data = Arc::new(Mutex::new(0));
    let mut handles = vec![];
    for x in 0..3 {
        let data_ref = Arc::clone(&data);
        let handle = thread::spawn(move || {
            let mut data = data_ref.lock().unwrap();
            *data += 1;
        });
        handles.push(handle);
    }
    for handle in handles {
        handle.join().unwrap();
    }
    let data: u64 = *data.lock().unwrap();
    println!("{}", data); // 3
}
```

Obr. 28: Ukázka používání měnění proměnné z více vláken

Message passing

Existuje i druhý typ paralelního programování, kde se používá posílání „zpráv“ (hodnot) mezi vlákny. Protože standardní knihovna C++ tento způsob nepodporuje (nemá kanály se synchronizací (C++ reference)), nebudeme do tohoto způsobu moc zabývat. V Rustu jde použít primitiv `mpsc` (multiple producer, single consumer). (The Book, sekce 16.2)

Uváznutí

Rust sice garantuje synchronizaci a správné posílání mezi vlákny, ale *negarantuje, že vlákna nebudou čekat na sebe navzájem* (uváznutí, nebo také deadlock). (The Book, sekce 16.3)

6 Závěr

Cílem práce bylo představit jazyk Rust. Tento cíl byl splněn. Práce by šla rozšířit představením většího projektu v Rustu, kde by se prakticky využily získané znalosti a lépe projevila schopnost jazyka elegantně řešit časté problémy jako například zpracování chyb.

Rust je dobrou náhradou za C++, jdou v něm vytvářet bezpečnější programy a zároveň je uživatelsky přívětivější díky lepším nástrojům. Celosvětová komunita má rovněž tento trend, spousta velkých společností volí pro nové projekty exkluzivně Rust místo C++ (například Microsoft, Amazon,...). Doporučuji si tedy minimálně s tímto jazykem pohrát – třeba mu přijdete na chuť.

Také si myslím, že se jedná o skvělý učící nástroj pro paralelní programování – kompilátor totož hlídá bezpečnost a „správnost“ programu (nestane se tedy, že někde zapomeneme na `Mutex` objekt). Osobně jsem dokázal za cca 30 minut vytvořit komplikovanější program, aniž bych měl dřívější zkušenost s paralelním programováním.

7 Literatura

[C++ reference] *C++ reference* [online]. [cit. 2023-06-07]. Dostupné z WWW: <https://en.cppreference.com/w/cpp>.

[Docs.rs] *Docs.rs* [online]. [cit. 2023-06-08]. Dostupné z WWW: <https://docs.rs/>.

[Rust By Example] *Rust By Example* [online]. [cit. 2023-06-07]. Dostupné z WWW: <https://doc.rust-lang.org/rust-by-example/>.

[Rust STD] *The Rust Standard Library* [online]. [cit. 2023-06-07]. Dostupné z WWW: <https://doc.rust-lang.org/std/>.

[The Book] KLABNIK, Steve a Carol NICHOLS. *The Rust programming language* [online]. 2nd Edition. San Francisco: No Starch Press, 2023 [cit. 2023-06-06]. ISBN 9781718503106. Dostupné z WWW: <https://doc.rust-lang.org/book/>.

[The Rust Reference] *The Rust Reference* [online]. [cit. 2023-06-07]. Dostupné z WWW: <https://doc.rust-lang.org/reference/>.