

Systemnahe Programmierung in Rust

- "The Book" / E/A Projekt / Kap. 12 -

Hubert Högl

Technische Hochschule Augsburg / Informatik
<https://tha.de/~hhoegl>

2024-10-05 13:23:21

- “minigrep” - grep (globally search a regular expression and print)

```
$ cargo run searchstring example-filename.txt
```

- Kommandozeilenargumente (ohne externe Crates)
- Umgebungsvariablen (*environment variables*)
- Standardfehlerausgabe (stderr)
- Praktische Anwendung der Kapitel 7, 8, 9, 10 und 11
- Ein wenig Closures, Iteratoren und Trait Objects
- Code: `material/Beispiele/minigrep/`

E/A Projekt (2)

`std::env::args` : Unicode String

`std::env::args_os` : OSString (nicht unbedingt Unicode)

Simple Programm:

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    dbg!(args);
}
```

Speichern der Argumentwerte in Variablen

```
use std::fs;

fn main() {
    // --abschneiden--
    println!("In Datei {}", file_path);

    let contents = fs::read_to_string(file_path)
        .expect("Etwas ging beim Lesen der Datei schief");

    println!("Mit text:\n{contents}");
}
```

- \$ cargo run the poem.txt
- Funktion main() sollte Aufgaben trennen (Argumente + Datei lesen)
- Keine gute Fehlerbehandlung

- Jede Aufgabe in eine Funktion auslagern
- Konfigurationsvariablen in eine Struktur zusammenfassen
- Bessere Fehlermeldungen
- Fehlerbehandlung konzentriert an eine Ort, statt verstreut im ganzen Programm

⇒ Refaktorisierung!

- `main.rs` + `lib.rs`
- `main.rs`: Ausführung des Programms (kann nicht getestet werden)
- `lib.rs`: Logik des Programms in Funktionen aufgeteilt (Tests möglich)

→ Gute Testbarkeit

```
fn parse_config(args: &[String]) -> (&str, &str) {  
    let query = &args[1];  
    let file_path = &args[2];  
    (query, file_path)  
}
```

Besser: Rückgabe einer Config Struktur

```
struct Config {  
    query: String,  
    file_path: String,  
}  
  
fn parse_config(args: &[String]) -> Config {  
    let query = args[1].clone();  
    let file_path = args[2].clone();  
  
    Config { query, file_path }  
}
```

E/A Projekt (7) / Refactoring

Konstruktor für Config

```
fn main() {  
    let args: Vec<String> = env::args().collect();  
    let config = Config::new(&args);  
    // ...  
}  
  
// ...  
  
impl Config {  
    fn new(args: &[String]) -> Config {  
        let query = args[1].clone();  
        let file_path = args[2].clone();  
        Config { query, file_path }  
    }  
}
```

E/A Projekt (8) / Refactoring

Fehlerbehandlung: Programmierproblem, Nutzungsproblem

```
if args.len() < 3 {  
    panic!("Nicht genügend Argumente");  
}
```

panic! besser für Programmierproblem geeignet, da viele verwirrende Informationen für den Nutzer.

Fehler im Rückgabewert melden, statt panic!:

```
impl Config {  
    fn new(args: &[String]) -> Result<Config, &'static str> {  
        if args.len() < 3 {  
            return Err("Nicht genügend Argumente");  
        }  
  
        let query = args[1].clone();  
        let file_path = args[2].clone();  
  
        Ok(Config { query, file_path })  
    }  
}
```

Aufrufer verwendet z.B. `unwrap_or_else()` mit einer Closure:

```
use std::process;
```

```
fn main() {  
    let args: Vec<String> = env::args().collect();  
  
    let config = Config::new(&args).unwrap_or_else(|err| {  
        println!("Fehler beim Parsen der Argumente: {err}");  
        process::exit(1);  
    });  
    // ...  
}
```

E/A Projekt (10) / Logik in lib.rs

```
fn run(config: Config) {  
    let contents = fs::read_to_string(config.file_path)  
        .expect("Etwas ging beim Lesen der Datei schief");  
  
    println!("Mit text:\n{contents}");  
}
```

Mit Fehlerrückgabe (siehe “?”!)

```
use std::error::Error;
```

```
// ...
```

```
fn run(config: Config) -> Result<(), Box<dyn Error>> {  
    let contents = fs::read_to_string(config.file_path)?;  
  
    println!("Mit text:\n{contents}");  
  
    Ok(())  
}
```

Resultatwert muss in main() behandelt werden:

```
use minigrep::Config;

// ...

if let Err(e) = run(config) {
    println!("Anwendungsfehler: {e}");

    process::exit(1);
}
```

Test Driven Development (TDD)

- 1 Schreibe einen Test, der fehlschlägt, und führe ihn aus, um sicherzustellen, dass er aus dem von dir erwarteten Grund fehlschlägt.
- 2 Schreibe oder modifiziere gerade genug Code, um den neuen Test zu bestehen.
- 3 Refaktoriere den Code, den du gerade hinzugefügt oder geändert hast, und stelle sicher, dass die Tests weiterhin bestanden werden.
- 4 Wiederhole ab Schritt 1!

Wende TDD auf den Code in `lib.rs` an.

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {  
    vec! []  
}
```

Code schlägt zunaechst fehl

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn one_result() {
        let query = "dukt";
        let contents = "\
Rust:
sicher, schnell, produktiv.
Nimm drei.";

        assert_eq!(vec!["sicher, schnell, produktiv."], search(query, contents));
    }
}
```

E/A Projekt (14) / lib.rs und Tests

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}

pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.file_path)?;

    for line in search(&config.query, &contents) {
        println!("{line}");
    }
}
```

- `search_case_insensitive()`
- Neue Testfunktionen: `case_sensitive`, `case_insensitive`
- Neues Feld in Config: `pub case_sensitive: bool`
- In `Config::new()`:

```
let case_sensitive = env::var("CASE_INSENSITIVE").is_err();
```

- Ausführen mit

```
Linux: $ IGNORE_CASE=1 cargo run -- to poem.txt
```

```
Windows (Powershell): PS> $Env:IGNORE_CASE=1; cargo run to poem.txt
```

- Zunächst alle Ausgaben (auch Fehler) in `output.txt`:

```
$ cargo run > output.txt
```

- Trenne `stdout` und `stderr`

```
eprintln!(...);
```

- Damit erscheinen die Fehlermeldungen nicht mehr in `output.txt`.

Das E/A Projekt wird im nächsten Abschnitt 13.3. durch einen Iterator verbessert.