

RUST


MEILENSTEINE



EINSATZGEBIETE

- Gut geeignet für Low- und High-Level Programmierung!
- Langzeit Support garantiert durch Mozilla und Rust Foundation
- Hardware Programmierung

WERKZEUGE

- **CARGO** Paketmanager: <https://crates.io/> 
- **RUSTC** – Compiler
- **RUSTUP** – Installer und Setuptools: <https://rustup.rs/>

SPRACHEIGENSCHAFTEN

VARIABLEN

```
let x = 7;  
let y = 2.22;  
let z = true;  
  
x = 10; // Error:  
// doppelte Zuweisung von immutable Variable „x“  
  
let mut q = 4;  
q = 10; // passt
```

- Typinferenz: automatisch Typzuweisung
- Variablen werden mit **let** definiert und sind standardmäßig nicht änderbar (immutable)
- Erst durch den Zusatz **mut** werden sie änderbar (mutable)

TYPEN

PRIMITIVE TYPEN

Integer

- **i8, i16, i32, i64**

Fließkommazahlen

- float **f32**, double **f64**

Weitere:

- Boolean **bool** (**true** o. **false**)
- Unicode Char **char**
- String slice **str**

TUPEL (X, Y, ...)

- Beispiel:

- (**i8, bool**)
- (**char, bool, f64**)
- Zugriff mit .0, .1, ...

ARRAY [TYP, Länge]

- Beispiel:

- **[i8; 10]**
- Zugriff mit [0], [1], ...6



FUNKTION

```
fn hello_world() {  
    println!("Hello World!")  
}  
  
fn print_summe(a: i32, b: i32) {  
    println!("Summe ist {}", a + b);  
}  
  
fn main() {  
    hello_world();  
    print_summe(21, 12); // Typinferenz  
}
```

- Erst Parametername und dann Typenbezeichnung
- „main“-Funktion ist der Einstiegspunkt in das Programm

```
fn summe(a: i32, b: i32) -> i32 {  
    a + b  
}  
  
fn teilen(a: i32, b: i32) -> i32 {  
    if b == 0 {  
        return 0; // early return  
    }  
    a / b  
}  
  
fn calc(a: i32, b: i32) -> (i32, i32) {  
    (a + b, a - b)  
}
```

- fn ... () -> Rückgabetype {...}
- Kein **return** nötig, aber möglich!
- Mehrere Werte zurückgeben mit **Tupel**

EXPRESSION ODER STATEMENT

EXPRESSION

Geben **immer** einen Wert zurück!

Literale

- 4 | "Hello World!" | false

Operationen

- 3 + 4 | 7 == 6

Funktionsaufrufe

- summe(3, 4) | calc(7, 2)

...

STATEMENT

Geben **keinen** Wert zurück!

- **let** Deklarationen
- Semikolon macht Expression zu Statement:
 - summe(3, 4);
 - 3 + 4;

IF-ELSE

```
if zahl > 32 {  
    // IF Zweig  
} else if zahl < 32 {  
    // ELSE-IF Zweig  
} else {  
    // ELSE  
}
```

```
let ergebnis = if zahl == 32 { 'U' } else { 'T' }; →
```

- Else Zweig muss vorhanden sein
- IF, ELSE-IF und ELSE Zweig muss den gleichen **TYP** zurückgeben

SCHLEIFEN

```
while numb < 5 {  
    numb += 1;  
}  
  
loop { // unendlose Schleife  
    numb += 1; // --> Error: Integer überläuft  
}
```

```
for i in 1..5 {  
    println!("{}", i);  
}
```

RANGE (start..ende)

- 1..5 -> 1,2,3,4
- 1..=5 -> 1,2,3,4,5

```
let arr = [5, 11, 22];  
for c in &arr {  
    println!("{}", c);  
}
```

- **for**-Schleife ist performanter als **while**-Schleife

SYNTAX:

for variable_name **in** expression { ... }

BORROWING

```
let rosi = "Rosi".to_string();  
  
fn greet(name: &String) {  
    println!("Hello: {}", name);  
}  
  
fn say_goodbye(name: &String) {  
    println!("Goodbye: {}", name);  
}  
  
greet(&rosi);  
say_goodbye(&rosi);
```

```
let mut rosi = "Rosi".to_string();  
  
fn greet(name: &mut String) {  
    println!("Hello: {}", name);  
    *name = "Hansi".to_string();  
}  
  
fn say_goodbye(name: &String) {  
    println!("Goodbye: {}", name);  
}  
  
greet(&mut rosi);  
say_goodbye(& rosi);
```

- **&** in Typposition erzeugt Pointer auf Maschinenebene (wie C-RAW-Pointer)
- **&mut** in Typposition lässt Änderungen über den Pointer zu
 - um den Wert zu ändern muss man auf den Wert hinter dem Pointer zugreifen, was möglichst ist mit ***Expression**

WICHTIG!

Gleichzeitig kann ein Wert entweder
viele immutable Borrows (Aliasing) oder **ein mutable Borrow (Mutability)**
besitzen

ÜBUNGSAUFGABEN

- Zu finden unter <https://github.com/VWeidinger/rust-fosbos-sw/tree/main/aufgaben>.
- Rust Playground um Code zu testen: <https://play.rust-lang.org/>

Wenn du jetzt so richtig Lust hast durchzustarten, kannst du [hier](#) einen offiziell von RUST unterstützte Editor und [hier](#) den Rust Compiler herunterladen.

QUELLEN

INHALT

- <https://github.com/LukasKalbertodt/programmieren-in-rust/blob/master/slides/01-Grundlagen.pdf>
- <https://github.com/LukasKalbertodt/programmieren-in-rust/blob/master/slides/03-Ownership-System.pdf>
- <https://doc.rust-lang.org/stable/rust-by-example/>
- <https://doc.rust-lang.org/stable/book/>
- [https://en.wikipedia.org/wiki/Rust_\(programming_language\)](https://en.wikipedia.org/wiki/Rust_(programming_language))

BILDER UND GRAFIKEN

- https://upload.wikimedia.org/wikipedia/commons/thumb/d/d5/Rust_programming_language_black_logo.svg/1200px-Rust_programming_language_black_logo.svg.png
- <https://crates.io/assets/Cargo-Logo-Small-c39abeb466d747f3be442698662c5260.png>

