esse

ESSE Einführung in Security – VO 07: Sicherheitsimplikationen von Typisierung in Programmiersprachen

Alexander Firbas, Rafael Vrecar, Florian Fankhauser, Christian Schanes

INSO – Industrial Software
Institut für Information Systems Engineering | Fakultät für Informatik | Technische Universität Wien

Agenda

Was macht eine Programmiersprache sicher?

Safety & (Un)Definedness

Verschiedene Typisierungsansätze & Type Safety

Vergleich: C & Rust

Fehlerbehandlung

Speicherverwaltung

Quellen & Literatur

Zusammenfassung

Quiz:

Was sind die Ergebnisse der folgenden Ausdrücke?

(JavaScript) `'2' + 3`

1. `5`
2. `"23"`
3. `TypeError`
4. `undefined`

(PHP) `'2' + 3`

1. `5`
2. `"23"`
3. `TypeError`
4. `undefined`

(JavaScript) `'5' - 3`

1. `2`
2. `"2"`
3. `undefined`
4. `TypeError`

(PHP) `'5' - 3`

1. `2`
2. `"2"`
3. `undefined`
4. `TypeError`

(C) `'5' - 3`

1. `50`
2. `"2"`
3. `undefined`
4. `TypeError`

(Python) '5' - 3

1. 50
2. "2"
3. undefined
4. TypeError

(JavaScript & PHP) `0.3 + 0.5 === 0.8`

1. `true`
2. `false`
3. `TypeError`
4. `undefined`

(JavaScript & PHP) `0.3 + 0.5 == 0.8`

1. `true`
2. `false`
3. `TypeError`
4. `undefined`

(JavaScript) `typeof NaN`

1. `NaN`
2. `number`
3. `undefined`
4. `object`

(JavaScript & PHP) "2" == 2

1. `true`
2. `false`
3. `undefined`
4. `TypeError`

(JavaScript & PHP) "2" === 2

1. `true`
2. `false`
3. `undefined`
4. `TypeError`

(JavaScript) Boolean(null)

1. `true`
2. `false`
3. `undefined`
4. `TypeError`

(JavaScript) [] + []

1. `[]`
2. `""`
3. `undefined`
4. `TypeError`

(JavaScript) `[] + {}`

1. `0`
2. `"[]"`
3. `"[object Object]"`
4. `TypeError`

Was macht eine Programmiersprache sicher?

(Eine) Definition: „Safe Programming Language“

*„In einer **Safe Programming Language** (dt. sicheren Programmiersprache) kann man den Abstraktionen vertrauen, welche die Sprache bietet.*

Eine alternative Charakterisierung, die etwas konkreter ist, lautet:

In einer sicheren Programmiersprache haben die Programme immer eine präzise und wohldefinierte Semantik.“

(Vergleiche Erik Poll, 2023. S13.)

Safety & (Un)Definedness

- dt. Sicherheit & (Un)bestimmtheit
- Programmstatements nur unter bestimmten Umständen sinnvoll
- **Beispiel:** `a[i] = (float) x;` erfordert, dass ...
  - a ein Array ist, welches `float` speichern kann,
  - `i` einen `integer`-Wert innerhalb der Array-Schranken gespeichert hat
- stimmt eine dieser Bedingungen nicht, ist unklar, welche Semantik das Statement haben soll

(Vergleiche Erik Poll, 2023. S14f.)

Unklare Semantik: 2 Ansätze

- Akzeptanz von unsinnigen Statement-Ausführungen mit undefinierter Semantik
  - Programmierer:in muss korrekte Ausführung sicherstellen
  - Verhalten bei undefinierten Fällen abhängig von Compiler, Plattform und gegenwärtigem Speicherinhalt sowie Programmzustand
- Sicherstellung korrekter Statement-Ausführung oder Fehlermeldung
  - Sprache verhindert/detektiert unsinnige Ausführung durch Kompilier- oder Laufzeitprüfungen

(Vergleiche Erik Poll, 2023. S14f.)

Unklare Semantik: sicher vs. unsicher

- 1. Ansatz führt zu *unsicheren*, 2. *sicheren* Programmiersprachen
- sichere Sprachen: immer präzise definierte Semantik oder Fehlermeldung
- **Beispiele unsicherer Sprachen:** C, C++
- **Beispiele sicherer Sprachen:** C# (exkl. Pointer Arithmetik), Haskell, Java (exkl. „native machine code“ calls mit JNI), LISP, ML, Prolog, Rust
- Trend, Sprachen *sicherer* zu gestalten in Go, JavaScript ($\Rightarrow$ TypeScript), PHP, Python, Ruby, Scala
- neuere Systemprogrammiersprachen wie Rust und Swift mit Sicherheitsfokus

(Vergleiche Erik Poll, 2023. S14f.)

Ist *unsicher* immer schlecht?

- *Vorteil* unsicherer Ansatz: Geschwindigkeit, Kompatibilität (i.e., C)
- *Nachteil:* Sicherheitsrisiken, keine Garantie für Programmverhalten
- Sicherheit versus praktische Erwägungen (Geschwindigkeit, Bequemlichkeit)
- Sicherheitsopfer zugunsten praktischer Vorteile oft bereut
- tägliche Probleme: Buffer Overflows in C-Programmen
- Schreiben sicherer Programme ohne Sprachsicherheitsmechanismen ist schwierig(er)!

(Vergleiche Erik Poll, 2023. S14f.)

Beispiel: unsichere Prozeduren

Frage: Warum ist es problematisch, in einem beliebigen Programm eine Prozedur in einer unsicheren Sprache zu verwenden?

```
login(uname,passwd) // Prozedur in unsicherer Sprache
```

- Notwendigkeit genauer Prüfung aller Aufrufstellen zur Garantie von Verhalten
- Modularitätsproblem: Verständnis von Code ohne Kontext schwierig
- Problem bei Bibliotheken oder Betriebssystemaufrufen: Unbekannte Aufrufumstände

(Vergleiche Erik Poll, 2023. S14f.)

Inwiefern spielt Typisierung bei der Sicherheit von Sprachen eine Rolle?

Was kann sie für uns leisten?

Type Safety: Allgemeines

- **Type Safety (dt. Typsicherheit)** als wichtige Sicherheitsform
- Typsysteme legen Einschränkungen auf, um *unsinnige Programm(zuständ)e* zu vermeiden
- Sicherheit durch Vermeidung von Typfehlern bei der Ausführung
- **Typsichere Sprachen:** C#, Go, Haskell, Java, Kotlin, ML, Python, Rust, Swift
- **Typunsichere Sprachen:** C, C++

(Vergleiche Erik Poll, 2023. S18ff.)

Type Safety: Expressivität, potenzielle Probleme

- **Expressivität** (Ausdrucksstärke) und Typüberprüfungen beeinflussen Programmiersicherheit
- Umgang mit `null`-Werten: Laufzeitprüfung vs. Typsystemerweiterungen
  - **Beispiel Kotlin:** Unterscheidung: nullable und non-null Typen
- Fragilität der Typsicherheit durch potenzielle „Schlupflöcher“
  - z.B. **Type Confusion** durch Speicherinterpretation und -manipulation
    - Navigator 3.0
    - Java Card

(Vergleiche Erik Poll, 2023. S18ff.)

Beispiele für verschiedene Typisierungsansätze

- Statische Typisierung
- Dynamische Typisierung

- Starke Typisierung
- Schwache Typisierung

- Duck Typing
- Nominal Typing
- Strukturelles Typing
- Typinferenz
- Generische Typisierung
- Dependently Typed Languages

Statische und Dynamische Typisierung

- **statisch** $\Rightarrow$ Datentyp einer Variable ist zur *Compile-Time* bekannt und kann sich *nicht ändern*.
- **dynamisch** $\Rightarrow$ Datentyp einer Variable wird *zur Laufzeit ermittelt* und *kann sich ändern*.

- Beispiele in Java (statisch) und Python (dynamisch):
  - Java:

```
int nummer = 1; // nummer IMMER Typ int
```

  - Python:

```
nummer = 1 # nummer ist zunaechst Typ int
nummer = "eins" # und schon ist nummer Typ str
```

Starke und Schwache Typisierung

- **stark** $\Rightarrow$ Typen von Werten strikt gehandhabt, Konvertierungen zwischen inkompatiblen Typen müssen *explizit* gemacht werden
- **schwach** $\Rightarrow$ Konvertierungen zwischen inkompatiblen Typen können *implizit* gemacht werden

- Beispiele in Python (stark) und JavaScript (schwach):
  - Python:

```
a = "5" + 10 # TypeError
a = "5" + str(10) # gueltig
```

  - JavaScript:

```
var a = "5" + 10; // a = "510" (String), da 10 zu
String konvertiert
```

Duck Typing, Nominal Typing, Strukturelles Typing

- **Duck Typing:** Eigenschaften & Methoden eines Objekts bestimmen Zugehörigkeit zu Typ: „Wenn es wie eine Ente quakt und wie eine Ente läuft, dann ist es wahrscheinlich eine Ente.“
- **Nominal Typing:** Typenbestimmung basierend auf Namen und expliziten Definitionen. Zwei Typen nur kompatibel, wenn selber Namen oder einer explizit als Untertyp des anderen deklariert
- **Strukturelles Typing:** Typenkompatibilität basiert auf Struktur eines Typs, d.h. Mitgliedern & Methoden, unabhängig von Name

Typinferenz, Generische Typisierung, Dependently Typed Languages

- **Typinferenz:** Fähigkeit vom Compiler/Laufzeitumgebung, Typ automatisch aus Kontext zu erschließen, ohne explizite Angabe
- **Generische Typisierung:** Typen mit Variablen definieren, sodass mit verschiedenen anderen Typen wiederverwendbar
- **Dependently Typed Languages:** Typen können von Werten abhängen, ermöglicht ausdrucksstarke Typsysteme, eng mit Programm-Logik verknüpft

Beispiel für Typisierung in verschiedenen Sprachen

Funktion/Methode **`mylen`** schreiben, die als Parameter **`String`** hat, und dessen *Länge* als **`Integer`** zurückgibt.

mylen in Python

```
def mylen(s: str) -> int:
    return len(s)
```

- Laufzeitprüfung
- Ducktyping
- Typannotationen haben keine bindende Bedeutung

mylen in Java

```
static int mylen(String s) {
    return s.length();
}
```

- Statische Prüfung
- `s == null` $\Rightarrow$ `NullPointerException`

mylen in Kotlin (Variante 1)

```
fun mylen(s : String) : Int {
    return s.length;
}
```

- Statische Prüfung
- *Typsystem* stellt sicher, dass String nicht `null` ist.
- $\Rightarrow$ `NullPointerException` ist *nicht* möglich!

mylen in Kotlin (Variante 2)

```
fun mylen(s : String?) : Int {
    return s?.length ?: 0;
}
```

- Statische Prüfung
- String darf zwar `null` sein, aber Typsystem zwingt uns, Fehlerbehandlung durchzuführen
- $\Rightarrow$ `NullPointerException` ist *nicht* möglich!

Type Safety: Ausblick

- mathematische Beweise der Korrektheit von Typsystemen
- Theorembeweiser stellen Zuverlässigkeit der Verifikation sicher
- Definition der Typsicherheit sowohl informell als auch formal
- Äquivalenzbeweise für defensive und untypisierte Semantiken
- Repräsentationsunabhängigkeit als Beweisziel für Typsicherheit

(Vergleiche Erik Poll, 2023. S18ff.)

Zwischenfazit zu „Safety“ von Programmiersprachen

- Sicherheit in Programmiersprachen vielschichtig, nicht nur Speicher und Typen
- Sicherheitsniveau variiert in verschiedenen Bereichen
- Low-Level-Sprachen mit weiteren Sicherheitsaspekten: Kontrollfluss und Stack
- Wichtigkeit korrekter Argumentanzahl bei Prozeduraufrufen
- ROP-Attacken als Beispiel für Kontrollfluss-Schwachstellen
- Format-String-Angriffe als Schwäche in C

Ein gängiger Fehler: Integer-Überlauf

- Sicherheit muss insbesondere auch bei Basisoperationen wie Arithmetik mitgedacht werden
- unterschiedlicher Umgang mit Integer-Überläufen in Sprachen
- undefiniertes Verhalten bei Überlauf in C
- Java definiert Überlaufverhalten präzise, C# ermöglicht Überlauf-Exceptions
- Überlaufbehandlung in sicherheitskritischen Systemen extrem wichtig
- Ariane V als Beispiel für katastrophalen Überlauf-Fehler

Quiz Part 2: Let's talk about C

Was sind die Ergebnisse der folgenden Ausdrücke?

C-Quiz: Frage 1

```
int main(void) {
    int i = 0;
    return i++ + ++i;
}
```

1. 1
2. 2
3. 3
4. Ich weiß es nicht.

C-Quiz: Frage 2

```
int main(void) {
    int i = 16;
    return (((((i >= i) << i) >> i) <= i));
}
```

1. 0
2. 1
3. 16
4. I don't know.

C-Quiz: Frage 3

```
int main(void) {
    char a = ' ' * 13;
    return a;
}
```

1. 416
2. 160
3. -96
4. Ik weet het niet.

C-Quiz: Frage 4

```
int main(void) {
    char a = 0;
    short int b = 0;
    return sizeof(b) == sizeof(a+b);
}
```

1. 0
2. 1
3. 2
4. Je ne sais pas.

C-Quiz: Frage 5

```
struct S {
    int i;
    char c;
} s;
int main(void) {
    return sizeof(*(&s));
}
```

1. 4
2. 5
3. 8
4. No lo sé.

C-Quiz: Auflösung

1. Auflösung live in der VO! :)

(Vergleiche `https://wordsandbuttons.online/SYTYKC.pdf`)

Fehlerbehandlung anhand des Rückgabewertes in C

- Rückgabewert `null`

```
void *ptr = malloc(size);
if (ptr == NULL) { /* Fehler */ }
```

- negative Rückgabewerte

```
int fd = open("invalidfile", O_RDONLY);
if (fd < 0) { /* Fehler */ }
```

- Rückgabe spezieller Konstanten

```
int c = fgetc(file);
if (c == EOF) { /* Fehler oder end-of-file */ }
```

Weitere Strategien zur Fehlerbehandlung in C

- funktionsabhängige Konventionen

```
char *end;
long i = strtol(str, &end, 10);
if (end == str) { /* Fehler */ }
```

- undefiniertes Verhalten

```
free(ptr); free(ptr);
```

$\Longrightarrow$ Fehlerbehandlung nicht möglich/sinnvoll

- fremder Code: willkürliche weitere Konventionen möglich

Herausfinden, welcher Fehler aufgetreten ist (in C)

- Überprüfung des Rückgabewertes auf spezielle Fehlerwerte
- Überprüfung globaler Fehlercodevariable
  - `errno` in der C-Standardlibrary `libc`
  - *Achtung:* Ist `errno` threadsafe auf aktueller Plattform?
  - *Achtung:* Auch wenn `errno` threadsafe ist: *„signal handlers that call functions that may set errno or modify the floating-point environment must save their original values, and restore them before returning.“ – glibc Dokumentation*
  - *Achtung:* Ist `errno` aktuell?
  - *Achtung:* `errno` nicht universell, siehe z.B.: `ferror`

Herausforderungen korrekter Fehlerbehandlung in C

- erfordert absolute Präzision durch Programmierer:in
- wird nicht durch Typsystem/Compiler sichergestellt
- *Ist ein zusätzlicher Aufwand!*

Wie kann ein modernes Typsystem helfen?

Fallbeispiel Rust – eine *sichere* Programmiersprache

- entwickelt von Mozilla als Antwort auf Jahrzente der unsicheren Systemprogrammierung
- wird in Linux Kernel integriert
- zunehmende Nutzung, Platz 20 im Tiobe Index
- „Zero-Cost Abstractions“ und Performance auf C-Niveau

Language	Time
C	1.00
Rust	1.04
C++	1.56
Java	1.89
⋮	⋮
Python	71.90
Lua	82.91

(Vergleiche Pereira et al. 2017, `https://rust-for-linux.com/`, `https://www.tiobe.com/tiobe-index/`)

Allgemeines zur Fehlerbehandlung in Rust

- keine (optional) überprüften Fehlercodes
- kein (optional überprüfter) globaler State (`errno, ...`)
- keine Exceptions ($\Longrightarrow$ Keine unerwarteten Sprünge im Kontrollfluss)

- stattdessen: Fehler werden mittels `Result`-Typ abgebildet.
  (Ausnahme: `panics`, entstehen z.B.: bei Division durch 0)

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

Fehlerbehandlung in Rust mittels Pattern Matching

- Behandlung von Normal- & Fehlerfall mittels *Pattern Matching*

```
let result: Result<File, io::Error> = File::open("file");
match result {
    Ok(file) => {
        // Do something with file ...
    },
    Err(e) => {
        // Handle error ...
    }
}
```

Herausfinden, welcher Fehler aufgetreten ist (in Rust)

```
match e.kind() {
    io::ErrorKind::NotFound => { /* ... */ },
    io::ErrorKind::PermissionDenied => { /* ... */ },
    _ => {/* any other error kind */}
}
```

- Compiler erzwingt, dass jeder Wert von `e.kind()` durch ein Pattern abgedeckt ist

Resümee zur Fehlerbehandlung in Rust

- genau eine Art, Möglichkeit an Fehlern zu signalisieren
- Fehler können wegen Typsystem nicht ignoriert werden
- keine Fehlerausprägung kann wegen Semantik von `match` übersehen werden
- Panics (=Termination wegen Fehler) sind möglich, aber *niemals undefiniertes Verhalten!*

Microsoft Survey 2019:

70% der Security Bugs sind

Memory Safety Issues

(Vergleiche https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/)

Manuelle Speicherverwaltung

- z.B. C
- `malloc` und `free`
- **Vorteil:** kein Performance-Overhead, *Sicherheitsvorteil:* Löschen des Speichers kann sichergestellt werden (i.e., bei sensiblen Daten kann Speicher direkt gelöscht werden)
- **Nachteil:** katastrophale Fehler möglich

Automatische Speicherverwaltung zur Laufzeit

- z.B. Swift, Java
- **Vorteil:** garantierte Speichersicherheit
- **Nachteil:** konstanter Overhead (z.B. in Swift) und/oder unerwartbarers Pausieren der Ausführung für Garbage Collection (z.B. bei Java)

- **ACHTUNG:** nur weil gute Abstraktion vorhanden, sind nicht automatisch alle Speicherprobleme „gelöst“

Automatische Speicherverwaltung zur Übersetzungszeit

- z.B. Rust
- **Vorteil:** kein (oder minimaler) Performance-Overhead, garantierte Speichersicherheit, Speicher-Bugs werden zur Übersetzungszeit anstatt Laufzeit sichtbar
- **Nachteil:** Programmierung schwieriger
- *konzeptuell:* vor Kompilation werden `malloc`/`free` an den *beweisbar* richtigen Stellen eingefügt
- möglich, da Code strikte Regeln erfüllen muss
- in Rust realisiert durch
  - Ownership
  - Borrowing

Ownership in Rust

1. Jeder Wert hat *genau einen* Owner.

2. Wenn Owner eines Wertes Scope verlässt, wird Wert freigegeben.

Moves in Rust

- *Achtung:* Semantik von Zuweisungen ist fundamental anders als in C, Java, Python ...
- Primitive Typen wie integer, floats (Allgemeiner: Typen mit *Copy-Trait*) werden kopiert
- sonst: Zuweisung `y = x` überträgt `y` Ownership über Wert von `x` und `x` verlässt Scope

```
let x : String = String::from("esse");
let y : String = x;
println!("y = {}", y); // ok
println!("x = {}", x); // compile error
```

- analog erzeugt Parameterübergabe & Funktionswertrückgabe einen Move

Moves in Rust – Beispiel 1

```
fn calc_len(input : String) -> usize {
    return input.len()
}

fn main() {
    let input = String::from("esse");
    let len = calc_len(input);
    println!("'{0}' is {1} chars long", input, len);
    // compile error!
}
```

Moves in Rust – Beispiel 2

```
fn calc_len(input : String) -> (usize, String) {
    return (input.len(), input)
}

fn main() {
    let input = String::from("esse");
    let (len, input) = calc_len(input);
    println!("'{0}' is {1} chars long", input, len);
    // Ok
}
```

Borrowing in Rust

- Variablen in Rust sind entweder ...
  - immutable (`let x : T = ...`)
  - oder mutable (`let mut x : T = ...`)
- Referenz == Pointer auf Wert, aber kein Ownership
- „Borrowing“ ist der Akt der Referenzerstellung
- für einen Wert `x : T` gibt es ...
  - *entweder* beliebig viele lesende Referenzen (`&x : &T`),
  - *oder* höchstens eine schreibende Referenz (`&mut x : &mut T`)
- Referenzen zeigen *immer* auf gültigen Speicher

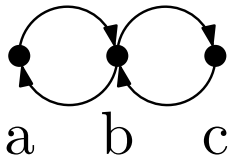
Borrowing in Rust – Beispiel

```
fn calc_len(input : &String) -> usize {
    return (*input).len()
    // Gleiche Bedeutung (Syntax Sugar) haette:
    // return input.len()
}

fn main() {
    let input = String::from("esse");
    let len = calc_len(&input);
    println!("'{0}' is {1} chars long", input, len);
    // Ok
}
```

Grenzen des Borrow-Checkers

- doppelt verkettete Liste



- Implementierung naiv nicht möglich – „Learn Rust With Entirely Too Many Linked Lists“
- im Allgemeinen: nicht jeder sichere Code kann von Borrow-Checker erkannt werden (vgl. Halteproblem).
- Lösung: Sichere Abstraktionen bauen, die auf `unsafe`-Blöcken aufbauen
  - willkürlicher Speicherzugriff
  - Aufruf von Maschinencode ausserhalb Rusts

(Vergleiche `https://rust-unofficial.github.io/too-many-lists/`)

Weiterführende Speicherverwaltung in Rust

„Smart Pointer“ sind ebensolche sicheren Abstraktionen:

- `Rc<T>` = `ReferenceCounting<T>` ermöglicht multiple Ownership von konstanten Werten
- `RefCell<T>` Referenz auf mutablen Speicher, Borrow-Checks werden zur Laufzeit statt statisch ausgeführt
- ...

andere Features, z.B.

- disjunkte Partitionierung von Arrays ermöglicht multiple mutating Referenzen auf Array (wichtig für Threading)

Rust verhindert Pointer auf ungültigen Speicher

- *„Dangling pointers“* – Pointer auf ungültigen Speicher:

```
// In C:
char* get_string() {
    char my_string[] = "esse";
    return my_string;
} // Ok

// In Rust:
fn get_string() -> &String {
    let my_string = String::from("esse");
    return &my_string;
} // compile error: Referenz &my_string kann nicht
  // laenger als owner (my_string) leben
```

$\Longrightarrow$ Keine „use after free“ Sicherheitslücken

Rust verhindert Memory Leaks

- Memory Leaks:

```
// In C:
void do_something() {
    char* wasted_memory = malloc(1024);
} // Ok

// In Rust:
fn do_something() {
    let not_wasted_memory = /* anything */
}
// not_wasted_memory verlaesst Scope,
// bessessener Wert wird hier und nur hier freigegeben
```

$\Longrightarrow$ Auch keine *double free* Fehler möglich

Rust verhindert Buffer Overflows

- Buffer Overflows:

```
// In C:
void buffer_overflow() {
    char buffer[10];
    strcpy(buffer, "This is way too long for the buffer");
}
// Undefiniertes Verhalten!
```

- in Rust: Zur Laufzeit wird immer ein Bounds-Check durchgeführt.
  - Bei ungültigem Zugriff: `panic` = Programmtermination
  - *Kein undefiniertes Verhalten!*

Rust verhindert Auslesen von uninitialisiertem Speicher

```
// In C:
void uninitialized_memory() {
    int x;
    printf("%d\n", x);
}
// Undefiniertes Verhalten!
```

- in Rust: Nur initialisierte Werte können gelesen werden, sonst liegt ein compile error vor.

Rust verhindert Data Races

- „Data Races“: Nebenläufiges, unsynchronisertes Schreiben auf geteilten Speicher mehrerer Prozesse
- nicht möglich, da für jeden Wert maximal eine schreibende Referenz gleichzeitig existieren kann
- nebenläufiges Schreiben nur auf kontrollierte Art möglich, z.B.: mittels `Mutex`

Statische Garantien durch Makro & Typsystem in Rust

Makros in Rust (Ausdrücke mit „!“) ...

- Rust-Code, der zur Compile-Time ausgeführt wird
- anders als der C-Präprozessor (rohe Stringmanipulation)
- erlauben mächtige Überprüfungen zur Übersetzungszeit

```
println!("Pi auf 4 Nachkommastellen = {:.4}",
        std::f64::consts::PI);
```

$\Longrightarrow$ „Pi auf 4 Nachkommastellen = 3.1416“

```
println!("Pi auf 4 Nachkommastellen = {:.4}",
        "Apfel", "Kuchen");
```

$\Longrightarrow$ compile error

Beispiel für fortgeschrittene statische Prüfung in Rust

```
struct User { name: String, id: i64 }
let user = sqlx::query_as!(User,
    "SELECT * FROM User WHERE name = ?",
    name
).fetch_all(&pool).await?;
```

Erklärung zur fortgeschrittenen statischen Prüfung in Rust

- SQLx Library: `https://github.com/launchbadge/sqlx`
- die statische Typisierung und Überprüfung zur Kompilationszeit anhand einer Dev-Datenbank stellt sicher, dass
  - das SQL-Statement valide ist,
  - die DB ein Ergebnis zurückgibt, welches mit der `User`-Struktur verträglich ist
  - das Ergebnis automatisch geparsed wird
  - der Parameter `name` mit der Anfrage verträglich ist

Keine Nullpointer, aber Options in Rust

„I call it my billion-dollar mistake. It was the invention of the null reference in 1965. (. . .)“ – Tony Hoare

- Rust besitzt keine Nullpointer (Ausnahme: „raw pointers“ in `unsafe` Blöcken)
- Möglichkeit, dass Werte fehlen können, wird analog zu Fehlern mittels enum abgebildet:

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

Quellen und weiterführende Literatur 1/3

- Cristina Cifuentes und Gavin Bierman. What is a secure programming language? In *3rd Summit on Advances in Programming Languages (SNAPL 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019. `https://doi.org/10.4230/LIPIcs.SNAPL.2019.3`
- Erik Poll. Language-Based Security. 2023. `http://cs.ru.nl/E.Poll/papers/language_based_security.pdf`
- Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, und João Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN international conference on software language engineering*, Seiten 256–267, 2017. `https://dl.acm.org/doi/pdf/10.1145/3136014.3136031`

Quellen und weiterführende Literatur 2/3

- Steve Klabnik und Carol Nichols. *The Rust programming language*. No Starch Press, 2023. https://doc.rust-lang.org/book/title-page.html
- Too Many Lists. https://rust-unofficial.github.io/too-many-lists/, d. Accessed: 2023-11-10
- Rust For Linux. https://rust-for-linux.com/, b. Accessed: 2023-11-10
- Tiobe Index. https://www.tiobe.com/tiobe-index/, c. Accessed: 2023-11-10

Quellen und weiterführende Literatur 2/2

- Oleksandr Kaleniuk. So You Think You Know C? https://wordsandbuttons.online/SYTYKC.pdf, 2020. Accessed: 2023-11-10
- ZDNET. Microsoft: 70 Percent of all Security Bugs are Memory Safety Issues. https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-is Accessed: 2023-11-10
- PHP: a fractal of bad design. https://eev.ee/blog/2012/04/09/php-a-fractal-of-bad-design/, a. Accessed: 2023-11-10

Zusammenfassung

- Sichere Software zu entwickeln, ist komplex!
- *unsichere* Programmiersprache $\Rightarrow$ einfacher, kompilierendes Programm zu erzeugen; Sicherheit an Entwickler:innen ausgelagert!
- *sichere* Programmiersprache kann Teil der Bürde abnehmen
- verschiedene Ansätze bzgl. **Typisierung**
- Typisierung kann bei Refactoring helfen, insbesondere in großen Projekten
- unterschiedliche Sprachen haben (teilweise) *unintuitive, verschiedene Interpretationen* von Datentypen
- **Rust** als moderne Option; effizient und behandelt Probleme von C *(C ist alt!)*
- trotzdem: *kein* Allheilmittel
- ***Aufpassen! Know Your Tools!***

Vielen Dank!

https://security.inso.tuwien.ac.at/