FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Konzepte moderner Programmiersprachen (KmPS)

(Wahlfach, Modulnummer 55685)

Wintersemester 2025/2026

Prof. Dr. Andreas Claßen
(zusammen mit Prof. Dr. Heinrich Faßbender)
Fachbereich 5 Elektrotechnik und Informationstechnik
FH Aachen

Lifetime Problematik für Referenzen / Borrowed Data

Referenzierter Wert muss mindestens so lange existieren wie die Referenz selbst.
Referenzen kontrollieren nicht den Speicher, auf den sie verweisen …

Referenzen insbes. als Fkt-Rückgabewerte sowie als `struct`-Members problematisch, da Wert-Owner dann in anderem Scope, d.h. „andere Lebensdauer".
Referenz als Rückgabewert nur möglich, wenn dieser Wert auch per Referenzparameter übernommen …

```
struct MyNumRef { n: &i32 }   // … Codebsp so nicht korrekt / vollständig

fn to_mynumref(i: &i32) -> MyNumRef { MyNumRef { n: &i } } // … so nicht vollständig

fn num_value(nr: &MyNumref) -> &i32 { & nr.n }   // … so nicht korrekt / vollständig
```

Lifetime Annotations für Borrowed Data

… müssen vom Programmierer explizit programmiert werden *genau wie die Datentypen*
… bis auf „einfache Ausnahme-Fälle".

… sind symbolische Namen für (minimale) Lebensdauern der referenzierten Werte
*… d.h. sie **beschreiben** Lebensdauern, ändern/**bestimmen diese aber nicht**!*

```
struct MyNumRef<'a> { n: &'a i32 }
// … d.h. MyNumRef ist eine generische Struct-Definition (ähnlich Template).
// Wenn MyNumRef Wert über Scope-Grenzen transferiert wird, muss Lifetime spezifiziert werden.

fn max_mynumref<'b>(i1: &'b i32, i2: &'b i32) -> MyNumRef<'b>
   { MyNumRef { n: if i1 > i2 { &i1 } else { &i2 } } }

fn mk_mynumref<'a>(i: &'a i32) -> MyNumRef<'a> { MyNumRef { n: i } }

fn unwrap_num_value<'a,'b>(nr: &'a MyNumRef<'b>) -> &'b i32 { & nr.n }
```

Lifetime Annotations für Borrowed Data

… sind generische Parameter, die erst vom Compiler beim Borrow-Checking mit konkreten Lifetime-Werten *(== scopes!)* gefüllt werden

… ähnlich wie Datentyp-Parameter in Templates erst bei der Variantenbildung (Instanziierung) mit konkreten Datentypen besetzt werden

… Borrow Checker prüft die Lifetimes über Scopes hinweg

Check: Lifetime einer Referenz versus Lifetime des Werts entsprechend Lifetime seines Owners.

```
struct MyNumRef<'a> { n: &'a i32 }
// … d.h. MyNumRef ist eine generische Struct-Definition (ähnlich Template).
// Wenn MyNumRef Wert über Scope-Grenzen transferiert wird, muss Lifetime spezifiziert werden.

fn max_mynumref<'b>(i1: &'b i32, i2: &'b i32) -> MyNumRef<'b>
   { MyNumRef { n: if i1 > i2 { &i1 } else { &i2 } } }
```

Weglassen von Lifetime Annotations bei Funktionen

… mittels „Weglass-Regeln“ *(elision)* in bestimmten Fällen möglich.
Wenige Regeln/Fälle, die aber in der Programmierpraxis häufig vorkommen und die aus Borrow Checking Sicht „trivial einfach“ sind, so dass die „sowieso klaren Annotationen“ vom Programmierer nur als „nerviger Overhead“ empfunden würden.

- *Wenn nur ein Referenzparameter der Fkt und Rückgabetyp auch Referenz:*
  Rückgabe-Referenzen ohne Lifetime Annotation bekommt auch diese Lifetime-Annotation.

```
struct MyNumRef<'a> { n: &'a i32 }

fn mk_mynumref(i: &i32) -> MyNumRef   { MyNumRef { n: i } }
```

… oder `MyNumRef<'a>` oder `MyNumRef<'_>`

- *Für Methoden:* Wenn Referenzparameter `&self` oder `&mut self` vorkommt, dann kriegen Rückgabe-Referenzen dessen *implizite* Lifetime-Annotation, falls keine Annotationen spezifiziert.

Smart Pointer: `Rc<T>` und `RefCell<T>`

Für dynamische Datenstrukturen

`Rc<T>`:
Reference Counting eines geteilten Werts vom Typ `T` auf dem Heap.
Ermöglicht das sichere dynamische Borrowing mit Prüfung zur Laufzeit.

Wenn der Zähler auf Null dekrementiert wird, wird der Speicher freigegeben.

`RefCell<T>`:
"Hilfstyp", um Ref-Count-Zyklen bei Datenstrukturen mit Zyklen zu vermeiden.

Quelle: https://de.wikipedia.org/wiki/Rust_(Programmiersprache)

Option Types: `Option<T>`

… vermeiden Speicherzugriffsfehler wegen "nicht erwarteter" Nullpointer
z.B. bei Rückgabewerten von Funktionen.

Ist `enum` mit …
Wert **`Some(T)`** für einen "Nicht-Null" Wert sowie …
Wert **`None`** für einen "zu behandelnden" Null-Wert.

Pattern Matching auf den Rückgabewert mit den Patterns **`Some(T)`** und **`None`**.
Struktur des Werts erzwingt Fallunterscheidung (`if` / Pattern Matching) zwischen `Some(T)` Fall und `None` Fall,
um im `Some(T)` Fall an den eigentlichen Resultatwert vom Typ `T` dran zu kommen.
Ein "unbehandelter" `None` Fall ist also nicht möglich.
Fehlerbehandlung im `None` Fall statt über Exceptions.

Nullpointerwerte in Rust somit nicht benötigt und daher auch nicht erlaubt.

Ggfs "erst mal ohne Fehlerhandling beim Development" (Code für kontrolliertes Fehlerhandling dann später dazu):
Methode `.unwrap()` packt Wert aus. Bei `None` erfolgt hartes Beenden des Programms mit `panic` Error.

```
let val = result.unwrap();
```

Option Types: Beispiel inkl. Pattern Matching

```
fn checked_division(dividend: i32, divisor: i32) -> Option<i32> {
    if divisor == 0 {
        None  // ... failure represented as the `None` variant
    } else {
        Some(dividend / divisor)  // ... result is wrapped in a `Some` variant
    }
}

fn try_division(dividend: i32, divisor: i32) {
    match checked_division(dividend, divisor) {

        None => println!("{} / {} failed!", dividend, divisor),

        Some(quotient) => {
            println!("{} / {} = {}", dividend, divisor, quotient)
        },
    }
}

fn main() {
    try_division(4, 2);
    try_division(1, 0);
}
```

Quelle: https://doc.rust-lang.org/rust-by-example/std/option.html