Inhaltsverzeichnis

4 Effective Rust

4.1 The Stack and the Heap

Memory management

- stack ist sehr schnell
- Rust nutzt per default den Stack
- Speicher im Stack ist lokal zu einem Funktionsaufruf und in der Größe begrenzt
- Heap ist langsamer
- wird expliziet im Programm angefordert
- Speicher im Heap ist nicht in der Größe beschränkt und global zugreifbar

The Stack

- wenn eine Funktion aufgerufen wird, wird für diese eine bestimmter Speicherbereich auf dem Stack für lokale Variablen reserviert
- dies erfolgt automatisch und der Speicher wird automatisch nach beenden der Funktion wieder frei gegeben
- dies funktioniert sehr schnell
- variablen sind allerdings nur während dem Funktionsaufruf gültig
- Stack funktioniert nach dem LIFO-Prinzip (last in, first out)
- kann nicht genutzt werden, um Speicher an andere Funktionen zu übergeben

The Heap

- genutzt wenn werte länger als der Funktionsaufruf gespeichert werden soll oder wenn Speicher zwischen Funktionen ausgetauscht werden muss
- im gegensatz zum Stack können im Heap "Löscherëntstehen, wenn Speicher freigegeben wird
- Werte im Heap bleiben so lange erhalten, bis der entsprechende Speicherbereich frei gegeben wird
- die Adresse auf dem Heap wird in die lokale variable auf dem Stack gespeichert

- Bsp.: `let x = Box::new(5)`
    - Box belegt Speicher auf dem Heap
    - die Adresse wird in x auf dem Stack gespeichert
    - wenn die Funktion beendet wird und `x` vom Stack gelöscht wird, wird die Funktion *Drop* von Box aufgerufen und der Speicher auf dem Heap freigegeben

Arguments and borrowing

- es können auch Referenzen auf werte im Stack übergeben werden (borrowing)

What do other languages do?

- die meisten anderen Sprachen mit einem Garbage Collector, belegen Speicher auf dem Heap

Which to use?

siehe Datenstrukturen

- Warum benötigt man auch den Heap?
- der Stack arbeitet nur nach dem LIFO Prinzip
- beim Heap arbeitet sehr allgemein und der Speicher kann in zufälliger Reihenfolge belegt und freigegeben werden, was das Management komplexer macht
-

Runtime Efficiency

- den Stack zu verwalten ist sehr einfach
- der ßtack-pointer"muss nur in- bzw. dekrementiert werden
- den Heap zu verwalten ist wesentlich komplizierter

Semantic impact

- Rust ist stark durch das LIFO Prinzip des Stacks beeinflusst $\rightarrow$ automatische Speicherverwaltung
- sollen die stärken des Heap besser ausgenutzt werden, so benötigt man entweder eine besser Unterstützung zu Laufzeit (bspw. Garbage Collector) oder der Programmiere muss sich selbst um korrekte Speichernutzung kümmern, was wiederum durch den Compiler überprüft werden muss (dies bietet der Rust Compiler nicht)

4.2 Testing

- soll eine Bibliothek geschrieben werden, so kann man diese testen, indem man eine Funktion mit dem Attribut `#[test]` versieht in der die Funktionalität getestet wird
- ähnlich wie bei Python `if __name__=='__main__':`
- ausführen des Tests mit `cargo test`
- jeder Test der nicht `panic!` irgendwo auslöst ist erfolgreich
- fällt ein Test negativ aus, so wird auch ein status code in `$?` zurückgegeben
- es ist auch möglich zu testen, ob ein Programm abstürzt
  - Funktion mit dem Attribut `#[should_panic]` versehen
  - Test ist erfolgreich, wenn das Programm abstürzt
- mit `assert_eq!` kann bspw. eine Funktion auf richtige Ergebnisse geprüft werden
- Bsp.: `assert_eq!(4, foo(2));`
  - falls `foo(2)` nicht `4` zurück liefert, so wird `panic!` ausgelöst und das Programm stürzt ab
  - andernfalls passiert nichts und der Test ist bestanden
- weiterhin ist es auch möglich ein eigenes Modul für Tests zu schreiben, um einzelnen Tests zusammenzufassen
- dieses Modul wird nur für den eigentlichen Test mit kompiliert → spart Kompilierzeit und stellt sicher das der Test nicht im Release vorhanden ist
- eine weitere Variante ist eine eigene Test Crate zu schreiben, die die erstellte Bibliothek einbindet → wie man dies bei anderen Sprachen auch tun würde
- diese Test crate bekommt ihren eigenen Ordner

4.3 Conditional Compilation

...

4.4 Dokumentation

rustdoc

- die Rust Distribution enthält ein Tool zur Dokumentation `rustdoc`, welchen ebenfalls von `cargo doc` genutzt wird
- die Doku kann entweder direkt aus dem Quellcode oder aus Markdown Datei erzeugt werden

Documenting source code

- `///` zeigt einen Dokumentations Kommentar an
- Dokumentations Kommentar werden in Markdown geschrieben
- jeder Kommentar muss in einer eigen Zeile stehen und bezieht sich auf die zeile darunter

  → `/// Komentar`
  `let x = 7;`

  → nicht: `let x = 7; /// Kommentar`

Writing documentation comments

...

- es können spezielle Abschnitte definiert werden, bspw. `/// # Examples`
- dies ermöglicht es Beispiele in die Dokumentation zuschreiben, welche durch den Compiler auf ihre Funktionalität überprüft werden können

...

4.5 Iterators

...
...
...

5 Syntax and Semantics

5.1 Variable Bindings

- in vielen anderen Sprachen nur als Variable bezeichnet
- Rust bietet aber noch andere Features
- Definition mit `let`
- linke Seite ist Pattern

  → ermöglicht bspw. `let (x,y) = (1,2)`

- Rust ist statisch typisiert, d.h. Datentypen werden bei der Definition nach einem `:` angegeben und beim kompilieren überprüft

  → `let x: i32 = 5;`

  - "x ist eine Bindung mit dem Typ `i32` und dem Wert `5`"

- Rust verfügt außerdem noch über Typableitung, falls der Kompiler Schlussfolgern kann welchen Typ eine Variable hat, muss dieser nicht explizit mit angegeben werden

  → `let x = 5;`

- per default sind Bindungen **unveränderlich**
- soll eine Bindung veränderlich sein, so muss `mut` benutzt werden

  → `let mut x = 5;`

- Rust legt viel Wert auf Sicherheit, daher sind alle Bindungn per default unveränderlich. Wird ein Wert einer Bindung verändert, der nicht geändert werden soll, so wird dies vom Compiler abgefangen. Soll die Bindung geändert werden, so muss `mut` hinzugefügt werden.
- Bindings **müssen** vor dem benutzen initialisiert werden

5.2 Functions

- jedes Rust Programm hat mindestens eine Funktion

  →
  ```
  fn main(){
  }
  ```

- Parameter werden wie bei `let` angegeben, mit dem Unterschied, dass Datentypen explizit angegeben werden müssen

  → `fn foo(x: i32, y: i32){...}`

- Rust Funktionen haben **genau einen** Rückgabewert
- der Datentyp des Rückgabewertes wird nach einem `->` angegeben
- die letzte Zeile einer Funktion entscheidet, was zurückgegeben wird und hat kein Semikolon am Ende

```
→ fn foo(x: i32, y: i32) -> i32 {
      x + y
  }
```

- Rust ist ausdrucksbasiert und die Verwendung von Semikolons und geschweiften Klammern unterscheidet sich von denen die andere Sprachen verwenden

Expressions vs. Statements

- Ausdrücken haben einen Rückgabewert, Anweisungen nicht
- Rust enthält nur zwei Arten von Anweisungen (Statements), alles andere sind Ausdrücke
- Deklarationsanweisung
  - in anderen Sprache ist `x = y = 5` möglich, in Rust nicht
  - `let x = 5;` ist eine Anweisung und hat keinen Rückgabewert
- Ausdrucksanweisung
  - wandelt einen Ausdruck in eine Anweisung um
  - siehe Rückgabewert einer Funktion
  - mit `x + y;` würde `()` zurückgegeben

Early returns

- keyword `return`

Diverging functions

- Funktionen die nicht zurück kommen

```
fn diverges() -> ! {
    panic!("This function never returns!");
}
```

- `panic!` ist ein Makro, was der gerade ausgeführte Thread mit der übergebenen Nachricht abstürzt
- da diese Funktion abstürzt kann sie nichts zurückgeben und hat den Rückgabetyp `!`

5.3 Primitive Types

- Rust hat einige Primitive Typen definiert
- weitere nützliche Typen werden durch eine Standardbibliothek zur Verfügung gestellt

Booleans

- `bool` mit den Werten `true` und `false`

Char

- eine Unicode Zeichen
- wird in einfachen Anführungszeichen angegeben

  → `let x = 'x';`

- 4 Bytes

5.3.1 Numeric Types

- verschiedene Typen in unterschiedlichen Kategorien
  - signed, unsigned
  - fixed, variable
  - floatingpoint, integer
- zwei Teile: Kategorie und Größe (in Bits)

  → `u16` (unsigned 16 Bit)
- wird bei der Zuweisung nicht explizit ein Datentyp angegeben, kann Rust nicht Schlussfolgern, welcher genommen werden soll, deshalb gibt es folgende defaults:

  → `let x = 42;  // i32`

  → `let x = 1.0; // f64`
- in Rust existieren auch Typen, deren Größe von der Größe der Pointer des jeweiligen Systems abhängt

  → `isize` und `usize`

Arrays

- Liste von Elementen mit fester Größe
- per default **unveränderlich**
- Arrays sind vom Typ `[T;N]` wobei `T` für eine Datentyp steht und `N` die Anzahl der Elemente angibt

  → `let a = [1,2,3]; // a: [i32; 3]`
- Arrays können auch mit einem Initialwert versehen werden

  → `let a=[0;20]; // a: [i32; 3]`

  Array mit 20 Elementen, die alle den Wert `0` haben
- die Größe eines Arrays kann über die Methode `.len()` bestimmt werden
- auf die einzelnen Elemente kann über die Indizes zugegriffen werden

  → `a[2];`
- der korrekte Zugriff auf Arrays wird zur Laufzeit überprüft

Slices

- eine referenzierter Ausschnitt auf einen andere Datenstruktur
- erlaubt einen sicheren Zugriff auf einen bestimmten Teil eines Arrays, ohne diesen zu kopieren
- slices werden mittels einer bereits existierenden Variable erzeugt
- haben eine Länge, können veränderlich sein und verhalten sich wie Arrays

  →

```
let a = [0, 1, 2, 3, 4];

// A slice of a: just the elements 1, 2, and 3
let middle = &a[1..4];

// A slice containing all of the elements in a
let complete = &a[..];
```

Tuples

- geordnete Liste fester Größe

  → `let x = (1,"hello");`

  → `let x: (i32, &str) = (1, "hello");`

- Tupel können Elementen unterschiedlicher Datentypen enthalten

- man keinen einem Tupel ein anderes Tupel zuweisen, falls beide die gleichen Typen und Elementanzahl besitzen

```
let mut x = (1, 2); // x: (i32, i32)
let y = (2, 3); // y: (i32, i32)

x = y;
```

- man kann Tupel mit `let` Destrukturieren

  → `let (x,y,z) = (1,2,3);`

- oder man greift über die Indizes auf die Elemente zu, dabei wird, im Gegensatz zu Arrays, mit `.` auf die Elemente zugegriffen

```
let tuple = (1, 2, 3);

let x = tuple.0;
let y = tuple.1;
let z = tuple.2;
```