7.

Collections, Match, …

- Vec<T>
- Iteration mit for-Schleife
- Collections
- Konstanten
- Tuple-Structs
- Match
- Pattern

Vec<T>

- Homogene Sequenzdatenstruktur
- Dynamisch wachsendes Array
- Generisch über inneren Typen **T**
  - **`Vec<i32>, Vec<char>`**, …
  - Explizite Annotation seltenst nötig
  - Mehr zu Generics später

```rust
let mut v = Vec::new();

v.push(3.14);
v.push(27.0);
v.push(1337.42);

println!("{}", v[0]); // 3.14

println!("{}", v.len());      // 3
println!("{}", v.capacity()); // 4
```

- Dokumentation: https://doc.rust-lang.org/std/vec/struct.Vec.html

Vec<T>

Vec erstellen

```rust
// Empty vectors
let _ = Vec::new();
let _ = Vec::with_capacity(3);

// Macro to init contents
let _ = vec![1, 2, 3, 4];
let _ = vec![];
let _ = vec![42; 10];

// From slice (both are equivalent)
let arr = [27; 10];
let _ = arr.to_owned();
let _ = arr.to_vec();
```

Vec verändern

```rust
// Stack operations
v.push(3.14);
let top = v.pop();

// With given index (in O(n)!)
v.insert(7, 3.14); // index 7
v.remove(7);

// If order doesn‘t matter: remove
// element at index in O(1)
v.swap_remove(7);

// Remove all elements
v.clear();
```

Durch Datenstruktur iterieren

```rust
let a = vec!['a', 'b', 'c'];
for x in a {
    // x has type `char`
}

// error: moved value!
println!("{}", a.len());
```

```rust
let b = vec!['a', 'b', 'c'];
for x in &b {
    // x has type `&char`
}
```

- `for`-Schleifen wie Funktionen:
  - Können „Argument“ konsumieren
- Arrays können nicht konsumiert werden
  - Funktioniert nicht:

```rust
for x in [1, 2, 3] {}
```

Andere Collections

- Zu finden in `std::collections` (Doku)
- `VecDeque`: effizientes `push()` und `pop()` auf beiden Seiten
- `LinkedList`: Eigentlich immer schlechter als `Vec`
- Maps:
  - `HashMap`
  - `BTreeMap`: andere Implementation, kann manchmal schneller sein
- `HashSet` und `BTreeSet`
- `BinaryHeap`

Konstanten und Statics

```
const PI: f64 = 3.1415926;
const START_POINT: Point = Point {
    x: 10.0,
    y: 7.0,
};

// error: function calls not allowed!
const HOME: Point = Point::origin();

// You rarely need this
static FOO: bool = true;
```

- Explizite Typannotation nötig
- Konstanten
  - Keine feste Adresse
  - In `SCREAMING_SNAKE_CASE`
- Initialisierung nur mit *constant expression*
- Static
  - Fest Adresse im Speicher
  - Sehr selten nötig

Type Alias

```rust
// oh god why
type Zeichenkette = String;

type Grid<T> = Vec<Vec<T>>;
type ByteVec = Vec<u8>;



let zk = Zeichenkette::new();

// This works. It's just an alias,
// not a new type!
let s: String = zk;
```

Tuple Structs

- Zwischen Tuple und Struct
  - Typ hat Name
  - Felder sind anonym
- Sind wie Structs ein Typ!
  - **`impl`**-Block möglich
- Quasi nur sinnvoll für das *new-type* Pattern

```rust
// Special type to manifest semantic
// difference in the type system
struct Seconds(pub f64);

let a = Seconds(3.1);
let b: f64 = a;   // error!
let c: f64 = a.0; // ok

// Can have multiple fields. Note: this
// is pretty much never useful. Rather
// name the fields in a normal struct.
struct Foo(i32, bool);
```

Match

```rust
let count = 3;
let output = match count {
    0 => "zero",
    1 => "one",
    _ => "many",
};
println!("{}", output);

match count {
    1 => {  // blocks allowed
        do_stuff();
    }
    _ => {}
}
```

- Erstmal vergleichbar mit switch-case
  - Aber viel mächtiger
- Kein `break;` notwendig
- Ist auch Expression
- Mehrere `match`-Arme:
  - Links Arm-Pattern
  - Rechts Arm-Body
- Muss alle Fälle abdecken!

Match

```rust
let hour = 14;
let meal = match hour {
    8 | 9 => "breakfast",
    12 ... 15 => "lunch",
    19 ... 22 => "dinner",
    _ => "hungry!",
};
println!("{}", meal);



match 123 {
    n @ 0 ... 100 if is_prime(n) => {},
    _ => println!("not a prime below 100"),
}
```

- Mehrere Pattern mit | kombinieren
- Inklusive Ranges mit `...` Syntax
  - Leider nur in Pattern möglich
- Namen an Pattern binden mit `@`
- Optionale, beliebige Bedingung
  - *Match Guard* genannt

Pattern

Refutable Pattern
(passt nicht immer)
z.B. `(x, true)`

Irrefutable Pattern
(passt immer)
z.B. `(x, y)`

- „Am Ende“: Name, Literal(-range)
  - Spezieller Name: `_` zum ignorieren
  - Wenn Literal in Pattern → Refutable
- Destructuring: Aufbrechen von Strukturen
  - `(x, 3, …)` → Tuple
  - `&x` und `&mut x` → Referenztypen
  - `Foo(x, 3, …)` → Tuple Struct
  - `Foo { x, y: 3, z: bob, … }` → Struct
  - Enum später mehr…

Mit „`..`“ können Felder ignoriert werden!

Match

```
let x = (27, true); // : (i32, bool)
match x {
    (0, false) => {},
    (n, true) => println!("{}", n),
    (m, _) => println!("{}", m),
}


match Point::origin() {
    Point { x: 0.0, y } => println!("On y-axis: {}", y),
    Point { x: 3.0, y: peter } => println!("{}", peter),
    p => println!("Somewhere else: {:?}", p),
}
```

Eigenschaften von Bindings

- Werden vor Namen geschrieben
- `mut`
  - Macht Binding mutable
  - *Wichtig*: Bezieht sich auf das Binding!
- `ref`
  - Bindet Namen nicht an Wert, sondern an Referenz zu Wert
- `ref mut`
  - Bindet Namen an mutable Referenz zum Wert

```
let mut x = 3; // : i32

{
    let ref y = x; // : &i32
}

let ref mut z = x; // : &mut i32
*z = 27;
```

mut Binding vs. mutable Reference

```
// `x` binds to an `i32`, we
// declare the binding mutable,
// thus we can change the `i32`
let (mut x, z) = (3, 4);

// `y` binds to an immutable ref,
// we declare the binding mutable,
// thus we can change the
// reference (not referee!)
let mut y = &x;

*y = 15; // NOT OK!

y = &z; // OK: we change the ref
```

```
// same as on the left
let mut x = 3;


// `y` binds to a mutable ref,
// we don‘t declare the binding
// mutable, thus we can’t change
// the reference (but the
// referee!)
let y = &mut x;

*y = 15; // OK: we change referee

y = &z; // NOT OK!
```

Pattern nutzen

- `let`-Binding: (nur irrefutable!)

```
let Point { x, y: height } = Point::origin();
```

- Funktionsargumente (nur irrefutable!)

```
fn foo((x, y): (i32, i32), &x: &f64) { … }
```

- `for`-Schleife (nur irrefutable!)

```
for &(x, y) in &[(1, 3), (2, 4)] { … }
```

- `match`, `if let`, `while let` (refutable auch erlaubt)

if let/while let

- Syntaxzucker für spezielles Match
- `if let` kann `else`-Zweig haben

```
let mut foo = (false, 3);
if let (true, x) = foo { … }

while let (b, 0...5) = foo { … }
```

```
if let ⟨Pattern⟩ = ⟨Expr⟩ {
    ⟨Body⟩
}
```

=

```
match ⟨Expr⟩ {
    ⟨Pattern⟩ => { ⟨Body⟩ }
    _ => {}
}
```

```
while let ⟨Pattern⟩ = ⟨Expr⟩ {
    ⟨Body⟩
}
```

=

```
loop {
    match ⟨Expr⟩ {
        ⟨Pattern⟩ => { ⟨Body⟩ }
        _ => break,
    }
}
```

Referenz

Pattern

Grammatik (let und match):

GuardedPatternList := ⟨PatternList⟩ [`if` ⟨BoolExpr⟩]
PatternList := ⟨NamedPattern⟩ [`|` ⟨PatternList⟩]
NamedPattern := ⟨Name⟩ `@` ⟨Pattern⟩

LetBinding := `let` ⟨Pattern⟩ `=` ⟨Expr⟩ `;`
MatchArm := ⟨GuardedPatternList⟩ `=>` ⟨Expr⟩

- Patterns werden an vielen Stellen genutzt:
  - `let`-Bindings, Funktionsargument, …
- Match erlaubt mehrere Patterns
- Match erlaubt Guard (arbiträre Bedingung)

Grammatik (Pattern):

Pattern := ⟨Leaf⟩ | ⟨DesRef⟩ | ⟨DesTuple⟩
| ⟨DesStruct⟩ | ⟨DesTupleStruct⟩
| ⟨DesEnum⟩
Leaf := [`ref`] [`mut`] ⟨Name⟩ | ⟨Literal⟩ | ⟨LiteralRange⟩

DesRef := `&` ⟨Pattern⟩ | `&mut` ⟨Pattern⟩
DesTuple := `(` ⟨Pattern⟩`,*` `)`
DesStruct := ⟨StructName⟩
`{` (⟨Name⟩ | ⟨FieldName⟩ `:` ⟨Pattern⟩)`,*` `}`
DesTupleStruct := ⟨StructName⟩ `(` ⟨Pattern⟩`,*` `)`
DesEnum := ⟨EnumName⟩
`::` [⟨DesStruct⟩ | ⟨DesTupleStruct⟩]