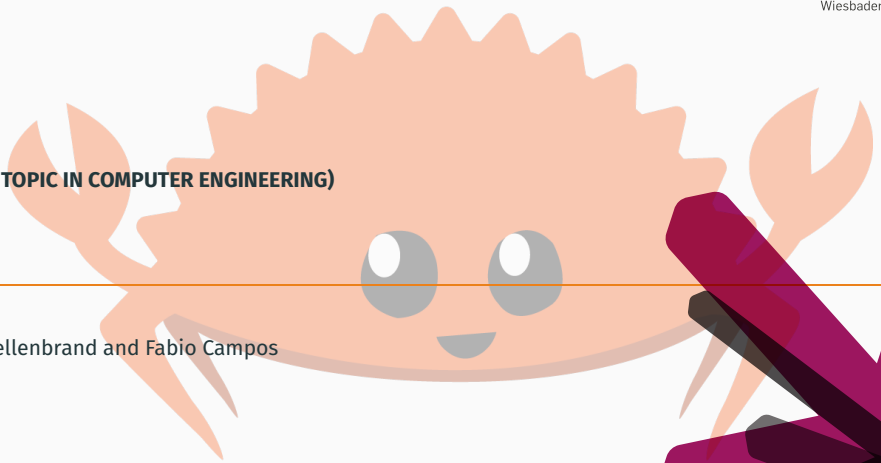
Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim



Rust

(SELECTED TOPIC IN COMPUTER ENGINEERING)

LV 7281

Andreas Hellenbrand and Fabio Campos

Topics

- Common Collections
- Enumerations

Common Collections

Collections

- Sequences

Vec	a contiguous growable array type
LinkedList	a doubly-linked list with owned nodes.[1]
VecDeque	a double-ended queue implemented with a growable ring buffer.

[1] Note: it's better to use `Vec` or `VecDeque` as they are more efficient.

Collections

- Sequences

Vec	a contiguous growable array type
LinkedList	a doubly-linked list with owned nodes.[1]
VecDeque	a double-ended queue implemented with a growable ring buffer.

- Maps

HashMap	key-value pairs
BTreeMap	binary search tree (BST) (choice for a sorted map)

[1] Note: it's better to use `Vec` or `VecDeque` as they are more efficient.

Collections

- Sequences
  - **Vec** a contiguous growable array type
  - **LinkedList** a doubly-linked list with owned nodes.[1]
  - **VecDeque** a double-ended queue implemented with a growable ring buffer.
- Maps
  - **HashMap** key-value pairs
  - **BTreeMap** binary search tree (BST) (choice for a sorted map)
- Sets
  - **HashSet** set of unique values
  - **BTreeSet** set of unique values as BTS

[1] Note: it's better to use `Vec` or `VecDeque` as they are more efficient.

Common Collections

Collections

- Sequences

Vec	**a contiguous growable array type**
LinkedList	a doubly-linked list with owned nodes.
VecDeque	a double-ended queue implemented with a growable ring buffer.

- Maps

HashMap	**key-value pairs**
BTreeMap	binary search tree (BST) (choice for a sorted map)

- Sets

HashSet	set of unique values
BTreeSet	set of unique values as BTS

Vectors

creating a new vector:

```
1           let v: Vec<i32> = Vec::new();
2
3           let v = vec![1,2,3];
```

Vectors

creating a new vector:

```
        let v: Vec<i32> = Vec::new();

        let v = vec![1,2,3];
```

updating a vector

```
        let mut v: Vec<i32> = Vec::new();
        v.push(1);
        v.push(2);
        v.push(3);
        assert_eq!(vec.pop(), Some(3));   // more on Some later
```

there are many more functions, check the doc!

Vectors

creating a new vector:

```
        let v: Vec<i32> = Vec::new();

        let v = vec![1,2,3];
```

updating a vector

```
        let mut v: Vec<i32> = Vec::new();
        v.push(1);
        v.push(2);
        v.push(3);
        assert_eq!(vec.pop(), Some(3));   // more on Some later
```

there are many more functions, check the doc!

```
        let first: &i32 = &v[0];
        let third = &v[2];

        let third: Option<&i32> = v.get(2);  // why two ways?
```

Iterator

```rust
let v1 = vec![1, 2, 3];
for val in v1.iter() {
    println!("Got: {}", val);
}

for (i, val) = v1.iter().enumerate() {
    println!("{i}: {val}");
}

for val = v1.into_iter() {  // consums v1
    println!("{v1}");
}

```

Hash Maps

HashMap

a.k.a. dictionary or key-value-pairs

```
use std::collection::HashMap;

HashMap<K, V>
```

Hash Maps

```rust
#[derive(Eq, Hash, PartialEq)]
enum Team {
    A,
    B,
}

fn main() {
    let mut points = HashMap::new();

    points.insert(Team::A, 10);
    points.insert(Team::B, 15);
    fn main() {
        let mut points = HashMap::new();

        points.insert(Team::A, 10);             // insert
        points.insert(Team::B, 15);

        for (team, point) in points.iter() {    // interate all
            println!("{team:?}: {point}");
        }

        let point_a = points.entry(Team::A).or_insert(0); // insert if not present
        *point_a += 1;

        let point_a = points.get(&Team::A);     // access single item
        if let Some(p) = point_a {
            println!("A: {p}")
        }
    }
}
```

Enums & Pattern

Emuns

- enumerations
- similar to *algebraic data types* from functional languages

Emuns

- enumerations
- similar to *algebraic data types* from functional languages

```
enum IceCream {
    Fruit,
    Milk,
}
```

Emuns

- enumerations
- similar to *algebraic data types* from functional languages

```
enum IceCream {
    Fruit,
    Milk,
}
```

```
let cold = IceCream::Soft;

fn enjoy(ice: IceCream) { }

enjoy(IceCream::Soft);
enjoy(IceCream::Milk);
```

Enums with appended data (1/2)

Enums can have Data attached:

```
enum IceCream {
    Fruit(String),
    Milk(String),
}
let strawberry = IceCream::Fruit(String::from("Erdbeere"));
```

Enums with appended data (1/2)

Enums can have Data attached:

```
enum IceCream {
    Fruit(String),
    Milk(String),
}
let strawberry = IceCream::Fruit(String::from("Erdbeere"));
```

```
enum IpAddr {
    IpV4(u8,u8,u8,u8),
    IpV6(String),
}
```

Enums with appended data (2/2)

```
enum IceCream {
    Fruit { f : String , suggar : u8},   // Struct
    Milk(String , u8),                   // Tuple
    Water,                               // Unit
}
```

Enums - impl

Enums can have functions (cmp. `structs`)

```
enum IceCream {
    Fruit(String),
    Milk(String),
}

impl IceCream {
    fn enjoy(&self) { ... }
}

let strawberry = IceCream::Fruit(String::from("Erdbeere"));
strawberry.enjoy();
```

Question

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

Question

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

```
let some_num = Some(5);
let some_ice = Some(IceCream::Milk("chocolate".into()));

let nothing : Option<i32> = None;
```

Question

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

```
let some_num = Some(5);
let some_ice = Some(IceCream::Milk("chocolate".into()));

let nothing : Option<i32> = None;
```

no `null`!

match

```
let five = Some(5);
let x : i32 = 10;
let y = five + x;           // would this work?
```

match

```
let five = Some(5);
let x : i32 = 10;
let y = five + x;           // would this work?
```

```
let y = match five {
    Option::Some(i) => Some(i + x),
    Option::None => None,
}
```

match

```
let five = Some(5);
let x : i32 = 10;
let y = five + x;           // would this work?
```

```
let y = match five {
    Option::Some(i) => Some(i + x),
    Option::None => None,
}
```

switch-case in usefull

match

```
1           let y = match five {
2               Option::Some(i) => Some(i + x),
3           }                               // would this work?
```

match

```
1           let y = match five {
2               Option::Some(i) => Some(i + x),
3           }                               // would this work?
```

`match` is exhausive -> every option needs to be defined

```
1           let y = match five {
2               Option::Some(i) => Some(i + x),
3               _ => None           // _ catches everything not defined bevor
4           }
```

Guards and Binding

```
let x: i8 = 13;
match x {
    i if i < 0 => println!("negative"),
    2 | 3 | 5 | 7 => println!("prime less than 10"),
    n @ 10..=19 => println!("10 <= {n} <= 19"),
    i8::MAX => println!("max i8"),
    i if (i % 2 == 0) => println!("even"),
    _ => println!("it is just some number"),
    //  ^ allways needed when guards are used
}
```

Guards and Binding

```
        let x: i8 = 13;
        match x {
            i if i < 0 => println!("negative"),
            2 | 3 | 5 | 7 => println!("prime less than 10"),
            n @ 10..=19 => println!("10 <= {n} <= 19"),
            i8::MAX => println!("max i8"),
            i if (i % 2 == 0) => println!("even"),
            _ => println!("it is just some number"),
            //  ^ allways needed when guards are used
        }
```

```
    let x = Some(13);
    match x {
        Some(42) => println!("answer found"),
        Some(_n) => println!("thanks for the fish"),
        _ => (),
    }
```

if let

```
match x {
    Some(42) => println!("answer found"),
    _        => (),
}
```

if let

```
match x {
    Some(42) => println!("answer found"),
    _        => (),
}
```

```
if let Some(42) = x {
    println!("answer found");
}
```

Summary

- Collections
  - Vector
  - HashSet
  - HashMap
- Enumerations
  - custom types to that can be one of a set (of enumerated values)
  - `Option<T>` as a better `null`
  - `match`-pattern