



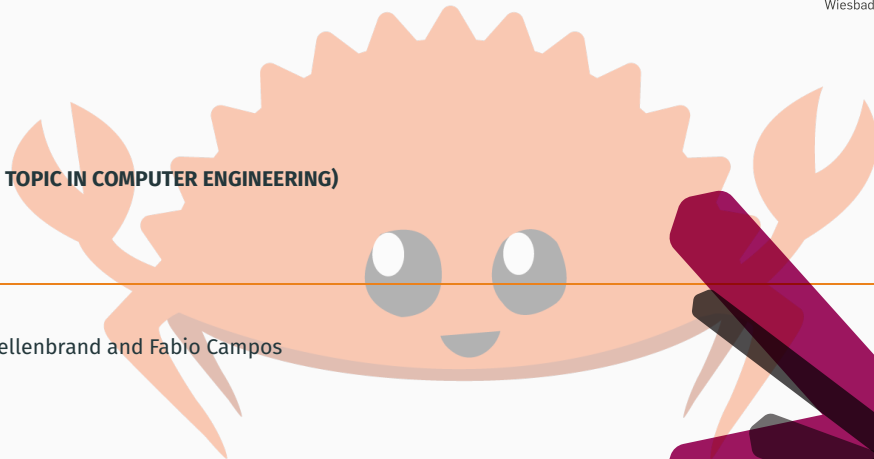
# Rust

(SELECTED TOPIC IN COMPUTER ENGINEERING)

LV 7281

---

Andreas Hellenbrand and Fabio Campos



- Common Collections
- Enumerations

## Common Collections

---

- Sequences

**Vec** a contiguous growable array type

**LinkedList** a doubly-linked list with owned nodes.<sup>1</sup>

**VecDeque** a double-ended queue implemented with a growable ring buffer.

---

<sup>1</sup>Note: it's better to use *Vec* or *VecDeque* as they are more efficient.

- Sequences

**Vec** a contiguous growable array type

**LinkedList** a doubly-linked list with owned nodes.<sup>1</sup>

**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)

---

<sup>1</sup>Note: it's better to use *Vec* or *VecDeque* as they are more efficient.

- Sequences

**Vec** a contiguous growable array type

**LinkedList** a doubly-linked list with owned nodes.<sup>1</sup>

**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

---

<sup>1</sup>Note: it's better to use *Vec* or *VecDeque* as they are more efficient.

- 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

creating a new vector:

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



creating a new vector:

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

updating a vector

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

there are many more functions, check the doc!

creating a new vector:

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

updating a vector

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

there are many more functions, check the doc!

```
1      let first: &i32 = &v[0];
2      let third = &v[2];
3
4      let third: Option<&i32> = v.get(2);  // why two ways?
```

```
1      let v1 = vec![1, 2, 3];
2      for val in v1.iter() {
3          println!("Got: {}", val);
4      }
5
6      for (i, val) = v1.iter().enumerate() {
7          println!("{i}: {val}");
8      }
9
10     for val = v1.into_iter() { // consumes v1
11         println!("{v1}");
12     }
13
```

## HashMap

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

```
1      use std::collection::HashMap;  
2  
3      HashMap<K, V>
```

# Hash Maps - Example

```
1  #[derive(Eq, Hash, PartialEq)]
2  enum Team {
3      A,
4      B,
5  }
6
7  fn main() {
8      let mut points = HashMap::new();
9
10     points.insert(Team::A, 10);
11     points.insert(Team::B, 15);
12
13     for (team, point) in points.iter() {    // iterate all
14         println!("{team:?}: {point}");
15     }
16
17     let point_a = points.entry(Team::A).or_insert(0); // insert if not present
18     *point_a += 1;
19
20     let point_a = points.get(&Team::A);    // access single item
21     if let Some(p) = point_a {
22         println!("A: {p}")
23     }
24 }
```

## Enums & Pattern

---

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

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

```
1  enum IceCream {  
2      Fruit,  
3      Milk,  
4  }
```



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

```
1  enum IceCream {  
2      Fruit,  
3      Milk,  
4  }  
  
1  let cold = IceCream::Fruit;  
2  
3  fn enjoy(ice: IceCream) { }  
4  
5  enjoy(IceCream::Soft);  
6  enjoy(IceCream::Milk);
```

## Enums with appended data (1/2)

Enums can have Data attached:

```
1      enum IceCream {  
2          Fruit(String),  
3          Milk(String),  
4      }  
5      let strawberry = IceCream::Fruit(String::from("Erdbeere"));
```

## Enums with appended data (1/2)

Enums can have Data attached:

```
1      enum IceCream {
2          Fruit(String),
3          Milk(String),
4      }
5      let strawberry = IceCream::Fruit(String::from("Erdbeere"));

1      enum IpAddr {
2          IPv4(u8,u8,u8,u8),
3          IPv6(String),
4      }
```

## Enums with appended data (2/2)

```
1  enum IceCream {  
2      Fruit { f : String , suggar : u8},    // Struct  
3      Milk(String , u8),                    // Tuple  
4      Water,                                // Unit  
5  }
```

Enums can have functions (cmp. *structs*)

```
1  enum IceCream {  
2      Fruit(String),  
3      Milk(String),  
4  }  
5  
6  impl IceCream {  
7      fn enjoy(&self) { ... }  
8  }  
9  
10 let strawberry = IceCream::Fruit(String::from("Erdbeere"));  
11 strawberry.enjoy();
```

## Question

```
1  enum Option<T> {  
2      Some(T),  
3      None,  
4  }
```

## Question

```
1  enum Option<T> {  
2      Some(T),  
3      None,  
4  }  
  
1  let some_num = Some(5);  
2  let some_ice = Some(IceCream::Milk("chocolate".into()));  
3  
4  let nothing : Option<i32> = None;
```

## Question

```
1      enum Option<T> {  
2          Some(T),  
3          None,  
4      }  
  
1      let some_num = Some(5);  
2      let some_ice = Some(IceCream::Milk("chocolate".into()));  
3  
4      let nothing : Option<i32> = None;
```

no *null*!



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

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

1      let y = match five {
2          Option::Some(i) => Some(i + x),
3          Option::None => None,
4      }
```

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

1      let y = match five {
2          Option::Some(i) => Some(i + x),
3          Option::None => None,
4      }
```

*switch-case in usefull*

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

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

*match* is exhaustive -> every option needs to be defined

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

## Guards and Binding

```
1      let x: i8 = 13;
2      match x {
3          i if i < 0 => println!("negative"),
4          2 | 3 | 5 | 7 => println!("prime less than 10"),
5          n @ 10..=19 => println!("10 <= {n} <= 19"),
6          i8::MAX => println!("max i8"),
7          i if (i % 2 == 0) => println!("even"),
8          _ => println!("it is just some number"),
9          // ^ allways needed when guards are used
10     }
```

### Guards and Binding

```
1      let x: i8 = 13;
2      match x {
3          i if i < 0 => println!("negative"),
4          2 | 3 | 5 | 7 => println!("prime less than 10"),
5          n @ 10..=19 => println!("10 <= {n} <= 19"),
6          i8::MAX => println!("max i8"),
7          i if (i % 2 == 0) => println!("even"),
8          _ => println!("it is just some number"),
9          // ^ allways needed when guards are used
10     }
11
```

```
1      let x = Some(13);
2      match x {
3          Some(42) => println!("answer found"),
4          Some(_n) => println!("thanks for the fish"),
5          _ => (),
6      }
```

```
1  match x {  
2      Some(42) => println!("answer found"),  
3      _       => (),  
4  }
```



# if let

```
1  match x {  
2      Some(42) => println!("answer found"),  
3      _       => (),  
4  }
```

```
1  if let Some(42) = x {  
2      println!("answer found");  
3  }
```

- 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