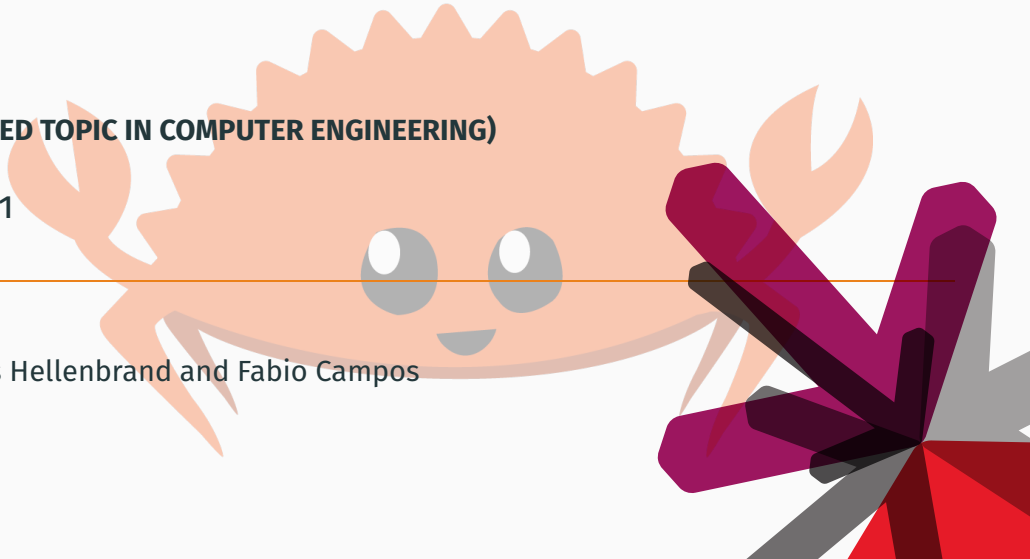
Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

# Rust

## (SELECTED TOPIC IN COMPUTER ENGINEERING)

LV 7281

Andreas Hellenbrand and Fabio Campos



# Agenda

# Smart pointers

# Raw pointers[1]

- allowed ignoring borrowing rules
- not guaranteed to point to valid memory
- allowed to be null
- no automatic cleanup

```
let mut num = 5;
let r1 = &num as *const i32;    // create raw pointers in safe code
let r2 = &mut num as *mut i32;  // create raw pointers in safe code

let address = 0x012345usize;    // even to arbitrary location in memory
let r = address as *const i32;

unsafe {                        // dereference raw pointers only in unsafe block
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
let points_at = unsafe { *r1 }; // dereference raw pointers only in unsafe block
println!("raw points at {}", points_at);
```

[1] *code@playground*

# Drop trait[2]

- *drop()* called for each binding after leaving scope
- releasing memory
- closing files, connections
- ...

```rust
struct Microphone;

impl Drop for Microphone {
    fn drop(&mut self) {
        println!("Just drop the mic!");
    }
}
```

```rust
fn main() {
    let m = Microphone;
    println!("It's over ...");
}
```

```rust
fn main() {
    let m = Microphone;
    drop(m);    // just drop it manually
    println!("It's over ...");
}
```

[2] https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=9e0a8127fa39dc478fff509096aec3a0

# Smart pointers

- act like a pointer
- having additional metadata and capabilities
- common smart pointers

  **Box<T>** for allocating values on the heap

  **Rc<T>** reference counting type, enables multiple ownership

  **Arc<T>** like *Rc<T>* but atomic reference counter

# Box<T>



- allocates on the heap
- exactly one owner
- deletes pointee when scope ends
- no overhead

```
fn main() {
    let b = Box::new(5);
    println!("b = {b}");
}
```

```rust
use crate::List::{Pair, Nil};

#[derive(Debug)]
enum List {
    Pair(i32, List),  // fine?
    Nil,
}

fn main() {
    let list = Pair(1, Pair(2, Pair(3, Nil)));
    println!("{:?}", list);
}
```

# Box<T>



```rust
use crate::List::{Pair, Nil};

#[derive(Debug)]
enum List {
    Pair(i32, List),  // fine?
    Nil,
}

fn main() {
    let list = Pair(1, Pair(2, Pair(3, Nil)));
    println!("{:?}", list);
}
```

```
error[E0072]: recursive type `List` has infinite size
...
5 |     Pair(i32, List),  // fine?
  |               ---- recursive without indirection
```

# Box<T>[3]



```rust
use crate::List::{Pair, Nil};

#[derive(Debug)]
enum List {
    Pair(i32, Box<List>),
    Nil,
}

fn main() {
    let list = Pair(1, Box::new(Pair(2, Box::new(Pair(3, Box::new(Nil))))));
    println!("{:?}", list);
}
```

[3] https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=308d17c9e9d63a0ee901eda8831ef849

# Reference counting Rc<T> 1/2

- allows multiple owners
- keeps track of the number of references
- deletes pointee after last owner leaves scope
- allows read-only access (immutable)
- only for use in single-threaded scenarios

```
1  use crate::List::{Pair, Nil};
2
3  enum List {
4      Pair(i32, Box<List>),
5      Nil,
6  }
7
8  fn main() {
9      let a = Pair(5, Box::new(Pair(10, Box::new(Nil))));
10     let b = Pair(3, Box::new(a));
11     let c = Pair(4, Box::new(a)); // fine?
12 }
```

# Reference counted Rc<T>[4]



```rust
use crate::List::{Pair, Nil};
use std::rc::Rc;

#[derive(Debug)]
enum List {
    Pair(i32, Rc<List>),
    Nil,
}

fn main() {
    let a = Rc::new(Pair(5, Rc::new(Pair(10, Rc::new(Nil)))));
    println!("count after creating a = {}", Rc::strong_count(&a));
    let _b = Pair(3, Rc::clone(&a)); // cloning an Rc<T> increases the reference count
    let _c = Pair(4, Rc::clone(&a));
    println!("count after creating b & c = {}", Rc::strong_count(&a));
    println!("{:?}", _c);
}
```

```
count after creating a = 1
count after creating b & c = 3
Pair(4, Pair(5, Pair(10, Nil)))
```

[4] https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=c5785b1a01ea2038ec35f656a2fb38a9

# Concurrency

# Processes vs threads

**Processes**

- separate memory space
- better isolation
- higher overhead
- slower context switching

**Threads**

- shared memory space
- higher risk of interference
- lower overhead
- faster context switching

# Creating a new thread with spawn

```
use std::thread;

fn main() {
    thread::spawn(|| {
            println!("Hello threaded world!");
    });
}
```

# Creating a New Thread with spawn

```rust
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
            println!("Hello threaded world!");
    });  // returns a JoinHandle

    handle.join();  // wait for the thread to finish
}
```

```rust
use std::thread;

fn main() {
    let mut handles = Vec::new();
    for _ in 0..9 {
        handles.push(thread::spawn(|| {
            println!("test");
        }))
    }

    for handle in handles {
        handle.join();
    }
}
```

# Returning values

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
            100 + 1
    });  // returns a JoinHandle

    let result_t = handle.join().unwrap();  //collect result from thread
    println!("{}", result_t);
}
```

```
impl<T> JoinHandle<T>() {
    fn join(self) -> Result<T> { ... }
}
```

# Passing values

```rust
fn endless(i: u32) -> u32{
    ...  // do something
    i
}

let input = 101;
let handle = thread::spawn(|| {
    endless(input)  // fine?
});

println!("{}", handle.join().unwrap());
```

# Passing values 1/2

```
fn endless(i: u32) -> u32{
    ...  // do something
    i
}

let input = 101;
let handle = thread::spawn(|| {
    endless(input)
});

println!("{}", handle.join().unwrap());
```

```
error[E0373]: closure may outlive the current function, but it borrows `input`
which is owned by the current function
...
```

# Passing values 2/2

```
fn endless(i: u32) -> u32{
    ...  // do something
    i
}

let input = 101;
let handle = thread::spawn(move || {
    endless(input)
});

println!("{}", handle.join().unwrap());
```

# Passing values using Arc<T>



```
use std::thread;
use std::sync::Arc;

fn main() {
    let s = Arc::new(String::from("Hello, world!"));  // use Arc<T>
    let mut handles = Vec::new();
    for _ in 0..9 {
        let for_thread = s.clone();
        handles.push(thread::spawn(move || {
            println!("{}", for_thread);
        }))
    }

    for handle in handles {
        handle.join();
    }
}
```

# Passing values using Arc<T> and Mutex<T>[5] 2/2

```rust
use std::thread;
use std::sync::Arc;
use std::sync::Mutex;

fn main() {
    let s = Arc::new(Mutex::new(String::from("Hello, world!")));  // use Arc<T> and Mutex<T>
    let mut handles = Vec::new();
    for _ in 0..9 {
        let for_thread = s.clone();
        handles.push(thread::spawn(move || {
            for_thread.lock().unwrap().push_str("!");
        }))
    }

    for handle in handles {
        handle.join();
    }
    println!("{}", s.lock().unwrap());
}
```

[5] https://play.rust-lang.org/?version=stable&mode=debug&edition=2015&gist=03ff4a908d2ccaf8fcd37e04c764acc0

# Open topics

- *RefCell<T>*
- communication channels, see *mpsc::channel()*
- *RwLock*, atomics
- sync and send traits
- nice crates, e.g., *crossbeam*, *rayon*