



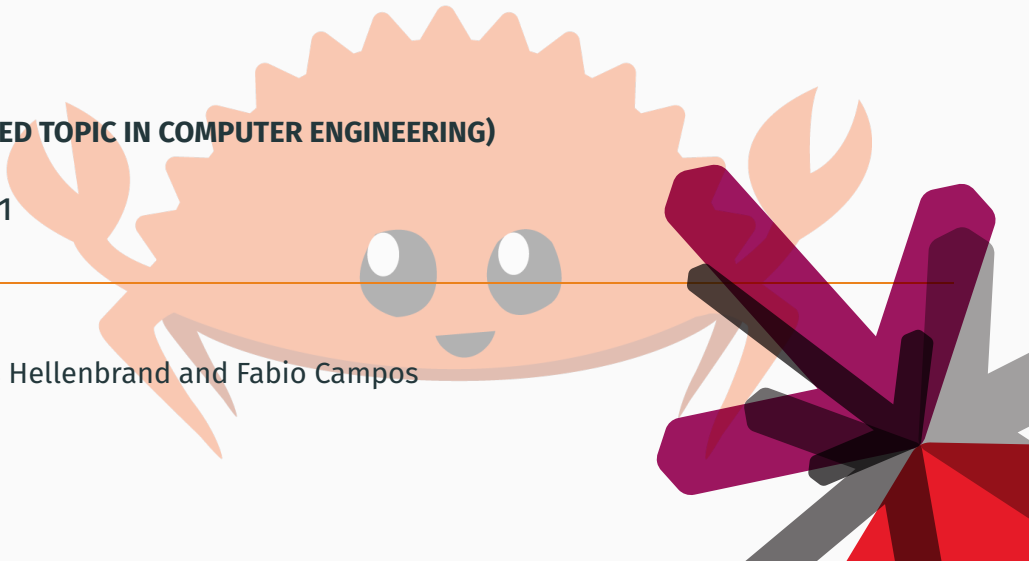
Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

Rust

(SELECTED TOPIC IN COMPUTER ENGINEERING)

LV 7281

Andreas Hellenbrand and Fabio Campos



Agenda

- Smart pointers
- Concurrency

Smart pointers

Raw pointers¹

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

```
1 let mut num = 5;
2 let r1 = &num as *const i32; // create raw pointers in safe code
3 let r2 = &mut num as *mut i32; // create raw pointers in safe code
4
5 let address = 0x012345usize; // even to arbitrary location in memory
6 let r = address as *const i32;
7
8 unsafe { // dereference raw pointers only in unsafe block
9     println!("r1 is: {}", *r1);
10    println!("r2 is: {}", *r2);
11 }
12 let points_at = unsafe { *r1 }; // dereference raw pointers only in unsafe block
13 println!("raw points at {}", points_at);
```

¹code@playground

Drop trait²

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

```
1 struct Microphone;
2
3 impl Drop for Microphone {
4     fn drop(&mut self) {
5         println!("Just drop the mic!");
6     }
7 }
```

```
1 fn main() {
2     let m = Microphone;
3     println!("It's over ...");
4 }
```

```
1 fn main() {
2     let m = Microphone;
3     drop(m);    // just drop it manually
4     println!("It's over ...");
5 }
```

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

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

```
1 fn main() {  
2     let b = Box::new(5);  
3     println!("b = {b}");  
4 }
```

```
1 use crate::List::{Pair, Nil};
2
3 #[derive(Debug)]
4 enum List {
5     Pair(i32, List), // fine?
6     Nil,
7 }
8
9 fn main() {
10     let list = Pair(1, Pair(2, Pair(3, Nil)));
11     println!("{:?}", list);
12 }
```



```
1 use crate::List::{Pair, Nil};
2
3 #[derive(Debug)]
4 enum List {
5     Pair(i32, List), // fine?
6     Nil,
7 }
8
9 fn main() {
10     let list = Pair(1, Pair(2, Pair(3, Nil)));
11     println!("{:?}", list);
12 }
```

error[E0072]: recursive type `List` has infinite size

```
...
5 |     Pair(i32, List), // fine?
  |               ---- recursive without indirection
```

```
1 use crate::List::{Pair, Nil};
2
3 #[derive(Debug)]
4 enum List {
5     Pair(i32, Box<List>),
6     Nil,
7 }
8
9 fn main() {
10     let list = Pair(1, Box::new(Pair(2, Box::new(Pair(3, Box::new(Nil))))));
11     println!("{:?}", list);
12 }
```

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

- 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 }
```

```
1 use crate::List::{Pair, Nil};
2 use std::rc::Rc;
3
4 #[derive(Debug)]
5 enum List {
6     Pair(i32, Rc<List>),
7     Nil,
8 }
9
10 fn main() {
11     let a = Rc::new(Pair(5, Rc::new(Pair(10, Rc::new(Nil)))));
12     println!("count after creating a = {}", Rc::strong_count(&a));
13     let _b = Pair(3, Rc::clone(&a)); // cloning an Rc<T> increases the reference count
14     let _c = Pair(4, Rc::clone(&a));
15     println!("count after creating b & c = {}", Rc::strong_count(&a));
16     println!("{:?}", _c);
17 }
```

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

```
1 use std::thread;
2
3 fn main() {
4     thread::spawn(|| {
5         println!("Hello threaded world!");
6     });
7 }
```

Creating a New Thread with spawn

```
1 use std::thread;
2
3 fn main() {
4     let handle = thread::spawn(|| {
5         println!("Hello threaded world!");
6     }); // returns a JoinHandle
7
8     handle.join(); // wait for the thread to finish
9 }
```

```
1 use std::thread;
2
3 fn main() {
4     let mut handles = Vec::new();
5     for _ in 0..9 {
6         handles.push(thread::spawn(|| {
7             println!("test");
8         }));
9     }
10
11     for handle in handles {
12         handle.join();
13     }
14 }
```


Returning values

```
1 use std::thread;
2
3 fn main() {
4     let handle = thread::spawn(|| {
5         100 + 1
6     }); // returns a JoinHandle
7
8     let result_t = handle.join().unwrap(); //collect result from thread
9     println!("{}", result_t);
10 }
```

```
1 impl<T> JoinHandle<T>() {
2     fn join(self) -> Result<T> { ... }
3 }
```

Passing values

```
1  fn endless(i: u32) -> u32{
2      ... // do something
3      i
4  }
5
6  let input = 101;
7  let handle = thread::spawn(|| {
8      endless(input) // fine?
9  });
10
11 println!("{}", handle.join().unwrap());
```

```
1 fn endless(i: u32) -> u32 {
2     ... // do something
3     i
4 }
5
6 let input = 101;
7 let handle = thread::spawn(|| {
8     endless(input)
9 });
10
11 println!("{}", handle.join().unwrap());
```

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

```
1  fn endless(i: u32) -> u32{
2      ... // do something
3      i
4  }
5
6  let input = 101;
7  let handle = thread::spawn(move || {
8      endless(input)
9  });
10
11 println!("{}", handle.join().unwrap());
```

```
1 use std::thread;
2 use std::sync::Arc;
3
4 fn main() {
5     let s = Arc::new(String::from("Hello, world!")); // use Arc<T>
6     let mut handles = Vec::new();
7     for _ in 0..9 {
8         let for_thread = s.clone();
9         handles.push(thread::spawn(move || {
10             println!("{}", for_thread);
11         }));
12     }
13
14     for handle in handles {
15         handle.join();
16     }
17 }
```

```
1 use std::thread;
2 use std::sync::Arc;
3 use std::sync::Mutex;
4
5 fn main() {
6     let s = Arc::new(Mutex::new(String::from("Hello, world!"))); // use Arc<T> and Mutex<T>
7     let mut handles = Vec::new();
8     for _ in 0..9 {
9         let for_thread = s.clone();
10        handles.push(thread::spawn(move || {
11            for_thread.lock().unwrap().push_str("!");
12        }));
13    }
14
15    for handle in handles {
16        handle.join();
17    }
18    println!("{}", s.lock().unwrap());
19 }
```

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*