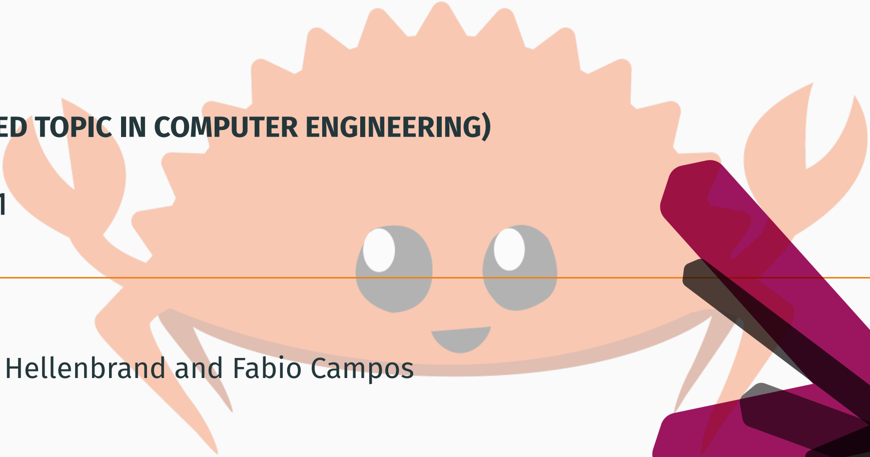
Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim



Rust

(SELECTED TOPIC IN COMPUTER ENGINEERING)

LV 7281

Andreas Hellenbrand and Fabio Campos

Agenda

Error Handling

Different types of errors[1] 1/2

1. dereferencing null pointer
2. server does not respond
3. (function) contract violation
4. config file has invalid format
5. division by zero
6. array out of bounds
7. file not found
8. user entered letters as phone number

[1]stronlgy based on *https://github.com/LukasKalbertodt/programmieren-in-rust*

Different types of errors 2/2

Unrecoverable Errors $\approx$ Bugs

- dereferencing null pointer
- (function) contract violation
- division by zero
- array out of bounds

Recoverable errors

- config file has invalid format
- file not found
- server does not respond
- user entered letters as phone number

Bugs

- unexpected and usually not treatable
- lead to unpredictable status
- **Solution:** abort
- in Rust: *panic!()* $\rightarrow$ abort thread

```rust
fn main() {
    let x = 101;
    println!("Hello world of panic!");
    panic!("goodbye, x = {}", x);
}
```

```
thread 'main' panicked at src/main.rs:4:5:
goodbye, x = 101
note: run with 'RUST_BACKTRACE=1' environment variable to display a backtrace
```

Panic

```rust
#[allow(unconditional_panic)]
fn main() {
    let vec = vec![1, 2];
    vec[101];
}
```

```
$ rustc panic.rs
$ ./panic
thread 'main' panicked at panic.rs:4:5:
index out of bounds: the len is 2 but the index is 101
note: run with 'RUST_BACKTRACE=1' environment variable to display a backtrace
```

```
$  RUST_BACKTRACE=1 ./panic
thread 'main' panicked at panic.rs:4:5:
index out of bounds: the len is 2 but the index is 101
stack backtrace:
   0: rust_begin_unwind
             at /rustc/7cf61ebde7b22796c69757901dd346d0fe70bd97/library/std/src/panicking.rs:647:5
   1: core::panicking::panic_fmt
             at /rustc/7cf61ebde7b22796c69757901dd346d0fe70bd97/library/core/src/panicking.rs:72:14
   2: core::panicking::panic_bounds_check
             at /rustc/7cf61ebde7b22796c69757901dd346d0fe70bd97/library/core/src/panicking.rs:208:5
   3: panic::main
   4: core::ops::function::FnOnce::call_once
note: Some details are omitted, run with 'RUST_BACKTRACE=full' for a verbose backtrace.
```

Where to panic?

- out of bounds
- overflows and underflows
- *unimplemented!()*
- *unreachable!()*
- Asserts $\rightarrow$ e.g. function contracts
- Deadlocks (if detected)
- . . .

Unwinding

- clears stack before terminating $\rightarrow$ unwinding
- by climbing up the stack
- *drops* all local objects ($\approx$ destructor, more in 2 weeks)
- can take quite a lot of time
- to deactivate (panic='abort' @cargo profile or *crate panic_abort*)

Recoverable errors

- expected
- due to invalid state of the environment
- can be handled
- in Rust: no exceptions, done using return values:
    - *Result<T, E>*
    - *Option<T>*
- error cannot be ignored $\rightarrow$ safer
- correct result must first be ”unpacked”

Result<T, E> example

```rust
use std::fs::File;

fn main() {
    let file_result = File::open("hello.txt");
    match file_result {
        Ok(file) => {
            // do something with the file
        },
        Err(error) => {
            panic!("Error: {}", error);
        }
    }
}
```

```rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}

impl File { // fake impl of File
    fn open(name: &str) -> Result<Self, String>
    { ... }
    ...
}
```

Propagating Errors

```rust
fn copy_file(from: &str, to: &str) -> Result<(), String> {
    match File::open(from) {
        Ok(file) => {
            ...
            Ok(())
        }
        Err(e) => {
            // not my beer!
            Err(e)
        }
    }
}
```

Which type of error?

- String … a good choice?
- enums significantly better!
    - defining error cases
    - methods for output
- error type *()* $\rightarrow$ *Option<T>*

```rust
fn copy_file(from: &str, to: &str) -> Result<(), FileError> {
    ...
}
enum FileError {
    NoFile,
    NoPermission,
    MaxDescriptors,
}
impl FileError {
    fn description(&self) -> String {
        ...
    }
}

```

Option

Option<T> has two variants:

- *None*: failure or lack of value
- *Some(value)*: tuple struct wraps value

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

```rust
fn take_101th(vec: Vec<i32>) -> Option<i32> {
    if vec.len() < 101 {
        None
    } else {
        Some(vec[4])
    }
}

fn main() {
    let vec = vec![1, 2];
    let big_vec = vec![0; 200];
    println!("{:?}, {:?}", take_101th(vec), take_101th(big_vec));
}

```

Shortcuts for Panic on Error

It converts recoverable error into bug

```rust
fn take_101th(vec: Vec<i32>) -> Option<i32> {
    if vec.len() < 101 {
        None
    } else {
        Some(vec[4])
    }
}
fn main() {
    let vec = vec![1, 2];
    let big_vec = vec![42; 200];
    println!("{:?}, {:?}", take_101th(big_vec.clone()), take_101th(big_vec).unwrap());
    println!("{:?}", take_101th(vec.clone()).unwrap()); // fine?
    println!("{:?}", take_101th(vec).expect("length of vector should be >= 101")); // fine?
}
```

Open topics

- *try!()* and the *?* operator $\rightarrow$ see next exercise
- the bottom type *!*
- ...

Lifetimes

Working with references

Reference always points to valid object

- no use after drop/free
- scope of reference < scope of referenced value
- scope of variables $\rightarrow$ stack (LIFO)

```rust
fn return_smt() -> &u32 {
    let x = 101u32;
    &x  // fine?
}

let r = {
    let x = 101u32;
    &x  // fine?
};
```

Rust compiler

the rust compiler ensures:

- no reference longer than referenced value
- aliasing xor mutability
- analysis based on
  - own function body
  - own signature
  - signature of called functions

```rust
fn foo(i: &u8) -> &u8 { ... }

let r = {
    let x = 3;
    foo(&x) // fine?
}
```

Rust compiler

the rust compiler ensures:

- no reference longer than referenced value
- aliasing xor mutability
- analysis based on
    - own function body
    - own signature
    - signature of called functions

```rust
fn foo(i: &u8, j: &u8) -> &u8 {
    i
}
```

```rust
fn foo(i: &u8, j: &u8) -> &u8 {
    j
}
```

```rust
static STATIC_X: u8 = 101;
fn foo(i: &u8, j: &u8) -> &u8 {
    &STATIC_X
}
```

```rust
fn foo(i: &u8, j: &u8) -> &u8 { ... }
let y = 101;
let r = {
    let x = 3;
    foo(&x, &y) // fine?
};
```

Rust compiler

- full analysis impossible without function body
- rust aims for safety
- $\rightsquigarrow$ necessary information required in signature

```rust
fn foo(i: &u8, j: &u8) -> &u8 { ... }
```

- How long could the value live behind the returned reference?
    - as long as the value behind *i*
    - as long as the value behind *j*
    - or static

Lifetime annotation syntax

```rust
&i32        // a reference
&'a i32     // a reference with an explicit lifetime
&'a mut i32 // a mutable reference with an explicit lifetime

fn foo<'a>(i: &'a u8, j: &u8) -> &'a u8 { i }

fn foo<'a>(i: &u8, j: &'a u8) -> &'a u8 { j }

static STATIC_X: u8 = 101;
fn foo(i: &u8, j: &u8) -> &'static u8 { &STATIC_X }
```

Lifetime elision 1/3

1st rule compiler assigns a lifetime parameter to each parameter that's a reference

```rust
fn foo(i: &u8, j: &u8) -> &u8 { ... }
```

$\downarrow$

```rust
fn foo<'a, 'b>(i: &'a u8, j: &'b u8) -> &u8 { ... }
```

Lifetime elision 2/3

2nd rule assuming a single input lifetime parameter, that lifetime is assigned to all output lifetime parameters

```
fn foo(i: &u8) -> &u8 { ... }
```

$\downarrow$

```
fn foo<'a>(i: &'a u8) -> &'a u8 { ... }
```

Lifetime elision 3/3

3rd rule assuming multiple input lifetime parameters, but one &*self*, the lifetime of *self* is assigned to all output lifetime parameters

```rust
fn foo(&self, e: &u8) -> &u8 { ... }
```

$\downarrow$

```rust
fn foo<'a>(&'a self, e: &u8) -> &'a u8 { ... }
```

Lifetime downgrade

```rust
use rand::Rng; // for generating random

fn foo<'a>(i: &'a u8, j: &'a u8) -> &'a u8 {
    let mut rng = rand::thread_rng(); // for generating random
    let n1: bool = rng.gen(); // for generating a random boolean
    if n1 {
        i
    } else {
        j
    }
}

static STATIC_X: u8 = 101;
fn main() {
    let r;
    {
        let x = 100;
        r = foo(&STATIC_X, &x);
    }
    println!("{}", r); // fine?
}

```

References within other types

```rust
struct MyRefType { // fine?
    r: &u8,
}

fn foo(i: &u8, j: &u8) -> MyRefType {
    MyRefType { r: i }
}

fn main() {
    let a: MyRefType = {
        let x = 101;
        foo(&x, &x); // fine?
    }
}

```

```rust
struct MyRefType<'a> {
    r: &'a u8,
}

fn foo<'a>(i: &'a u8, j: &u8) -> MyRefType<'a> {
    MyRefType { r: i }
}

```