

Rust for Embedded Developers

Day 2

Coffee procurement

I loaded some money onto my Campus Card.

You can use it freely in the break room downstairs.

Please remember how much money you spent.

I'll ask you to Twint me back at the end. 😊

Exercise repos on ZHAW GitHub

...should exist now, check <https://github.zhaw.ch>

You can push your solutions there as a backup.

Toward the end of the course, there will be an exercise involving GitHub Actions.

That only works if the repo is in the rust-wbk organisation.

(Individual users don't have access to GitHub Actions by default.)

Language 2

book chapters 6 – 10

- enums & pattern matching
- modules
- collections & error handling
- generics & traits
- lifetimes

Enums & Pattern Matching

book chapter 6

Declaration

```
1  enum IpAddrKind {  
2      V4,  
3      V6,  
4  }
```

Instantiation

```
1 let four = IpAddrKind::V4;  
2 let six = IpAddrKind::V6;
```

Storing Data ?

```
1  enum IpAddrKind {
2      V4,
3      V6,
4  }
5  struct IpAddr {
6      kind: IpAddrKind,
7      address: [u8; 16], // v4 and v6 both fit in here
8  }
9  fn main() {
10     let home = IpAddr {
11         kind: IpAddrKind::V4,
12         address: [127, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
13     };
14     home.address[8]; // ???
15 }
```

Enums are Tagged Unions

```
1  enum IpAddr {
2      V4([u8; 4]),
3      V6([u8; 16]),
4  }
5  fn main() {
6      let home = IpAddr::V4([127, 0, 0, 1]);
7
8      home.0[8]; // ❌ doesn't compile
9  }
```

Flexible Data Modelling

```
1  enum Message {  
2      Quit,                                // ~= unit-like struct  
3      Move { x: i32, y: i32 },             // ~= regular struct  
4      Write(String),                       // ~= tuple struct  
5      ChangeColor(i32, i32, i32),         // ~= tuple struct  
6  }
```

Billion-Dollar Mistake

The C-style pointer mixes two unrelated concepts:

- Indirection
- Optionality (something or nothing)

I call it my billion-dollar mistake. It was the invention of the null reference in 1965.

Tony Hoare

Rust:

Indirection: Reference &

Optionality: ?

The Option Type

```
1  enum Option<T> {  
2      None,  
3      Some(T),  
4  }  
5  main() {  
6      let some_number = Option::Some(5);  
7      let some_char = Some('e');  
8  
9      let absent_number: Option<i32> = None;  
10 }
```

Pattern Matching

book chapter 6.2

The match Expression

```
1  enum Coin {  
2      Penny,  
3      Nickel,  
4      Dime,  
5      Quarter,  
6  }  
7  fn value_in_cents(coin: Coin) -> u8 {  
8      match coin {  
9          Coin::Penny => 1,  
10         Coin::Nickel => 5,  
11         Coin::Dime => 10,  
12         Coin::Quarter => 25,  
13     }  
14 }
```

Patterns that Bind to Values

```
1  enum Message {
2      Quit,
3      Move { x: i32, y: i32 },
4      Write(String),
5      ChangeColor(i32, i32, i32),
6  }
7  fn receive(msg: Message) {
8      match msg {
9          Message::Quit => println!("bye bye!"),
10         Message::Move { x, y } => set_position(x, y),
11         Message::Write(text) => println!("{}", text),
12         Message::ChangeColor(r, g, b) => {
13             set_red(r);
14             set_green(g);
15             set_blue(b);
16         }
17     }
18 }
```

Matching on Option

```
1  fn plus_one(x: Option<i32>) -> Option<i32> {
2      match x {
3          None => None,
4          Some(i) => Some(i + 1),
5      }
6  }
7  fn main() {
8      let five = Some(5);
9      let six = plus_one(five);
10     let none = plus_one(None);
11 }
```

Exhaustive Matching

```
1 fn plus_one(x: Option<i32>) -> Option<i32> {  
2     match x {  
3         Some(i) => Some(i + 1),  
4         // forgot about None!  
5     }  
6 }
```

compiler says:

non-exhaustive patterns: None not covered

Exhaustive Matching

demo

Catch-all Patterns

```
1 let input: i32 = get_user_input();
2 match input {
3     7 => println!("You are lucky! 🍀"),
4     13 => println!("You are unlucky! 😞"),
5     x => println!("squared: {}", x * x),
6 }
```

Catch-all Patterns

```
1 let input: i32 = get_user_input();
2 match input {
3     7 => println!("You are lucky! 🎉"),
4     13 => println!("You are unlucky! 😞"),
5     _ => {}
6 }
```

Boilerplate...

```
1  match option_num {  
2      Some(num) => {  
3          println!("So much typing 🤒");  
4      }  
5      _ => {}  
6  }
```

Matching on a Single Pattern

```
1  let maybe_num: Option<i32> = Some(10);
2
3  if let Some(num) = maybe_num {
4      println!("number detected: {}", num);
5  }
6
7  let Some(num) = maybe_num else {
8      println!("404: number not found");
9      return;
10 }
```

Looping `while` a Pattern Matches

```
1 let mut numbers = vec![1, 2, 3];  
2 while let Some(n) = numbers.pop() {  
3     println!("removed from vec: {}", n);  
4 }
```

Niche optimization

	Optional + borrowed type	size
C	<code>uint8_t* maybe_ptr</code>	<code>size_t</code> (e.g. 8 bytes)
Rust	<code>maybe_ref: Option<&u8></code>	1 byte tag, padding + <code>usize</code> = 2 * <code>usize</code> ??

No! The "niche optimization" exploits unused bit patterns to optimize enums. References cannot be null → null represents `None` for type `Option<&u8>`. Works for custom enums, but without guarantees. → Verify where it matters.

	type	niche optimization	size
	<code>Option<usize></code>	✗	2 * <code>usize</code>
	<code>Option<&u8></code> , <code>Option<NonZeroUsize></code>	✓	<code>usize</code>

Summary

enums and pattern matching

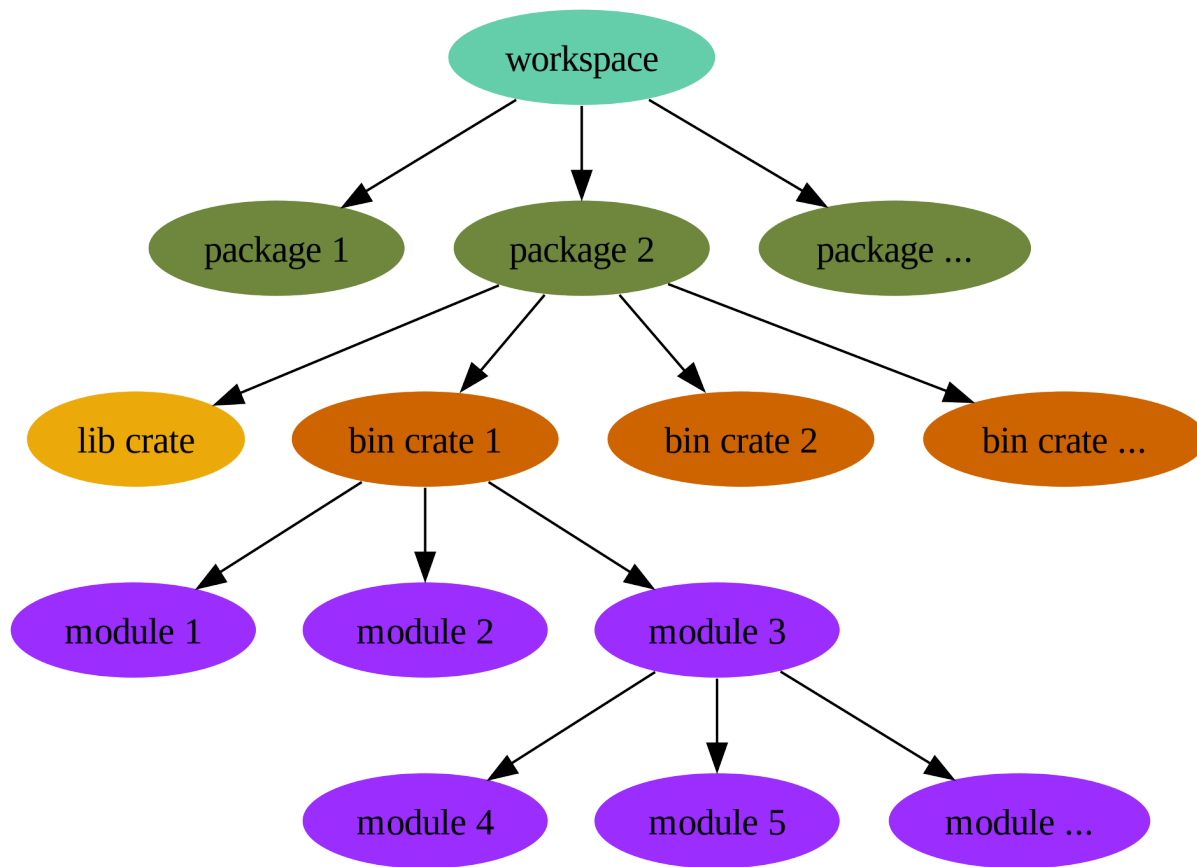
- model alternatives in your data
- prevent invalid data access bugs
- branch over data structures with `match`

Coffee break

Now or later? 

Project Organization

book chapter 7



The Crate

- executable crate: `main.rs` – library crate: `lib.rs`
- the basic compilation unit for `rustc`
- can comprise multiple `.rs` files

Simplification: `main.rs` is the crate!

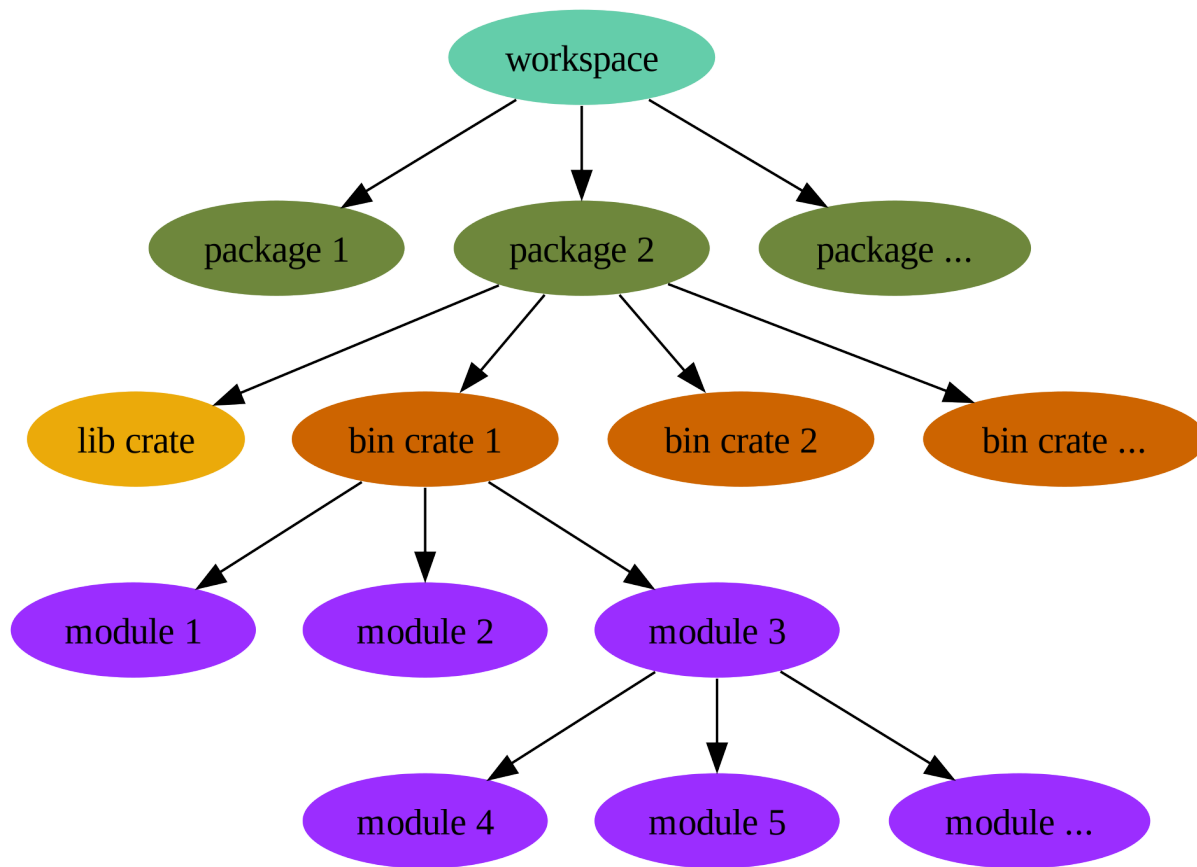
Above the Crate: The Package

- `Cargo.toml`, the basic build unit for `cargo`
- build scripts, e.g. for linking against C-libs
- tests against the public API of a crate
- semantically versioned

Simplification: `Cargo.toml` is the package!

Below the Crate: Modules

- Code organization
- Visibility & Encapsulation
- What we'll focus on next!



Package

```
1  > cargo new foo ; tree foo
2      Created binary (application) `foo` package
3  foo
4  └── Cargo.toml
5  └── src
6      └── main.rs
7
8  2 directories, 2 files
```



Crate entry point

Modules & Visibility

book chapter 7

Declaring a Module

demo

```
1 // main.rs - the root of the crate
2
3 mod inline_module { /* more code... */ }
4
5 mod file_module; // located in file_module.rs
```

The `pub` Keyword

```
1 fn main() {  
2     garden::carrot();  
3     garden::lettuce(); // error  
4 }  
5 mod garden {  
6     pub fn carrot() {}  
7     fn lettuce() {}  
8 }
```

compiler says:

function lettuce is private

Nesting

```
1  fn main() {
2      garden::fruits::orange(); // ✓
3      garden::veggies::lettuce(); // error
4  }
5  mod garden {
6      pub mod fruits {
7          pub fn orange() {}
8      }
9      mod veggies {
10         pub fn lettuce() {}
11     }
12 }
```

compiler says:

```
module veggies is private
```

Paths

```
1  mod garden {
2      pub mod fruits {
3          pub fn orange() {}
4      }
5      fn do_stuff() {
6          // relative path
7          fruits::orange();
8      }
9      mod veggies {
10         fn lettuce() {
11             // absolute path
12             crate::garden::fruits::orange();
13             // walking the module tree backwards
14             super::fruits::orange()
15         }
16     }
17 }
```

Struct Field Visibility

```
1  mod garden {  
2      pub struct Melon {  
3          pub size: u32,  
4          ripeness: u32,  
5      }  
6  }  
7  fn main() {  
8      let m: garden::Melon; // imagine initialization  
9  
10     m.size; // ok  
11     m.ripeness; // error, field is private  
12 }
```

The use Keyword

```
1 mod garden {  
2     pub struct Melon;  
3 }  
4 fn main() {  
5     let m: garden::Melon;  
6  
7     use garden::Melon;  
8     let m: Melon;  
9 }
```

Using External Packages

```
1 # Cargo.toml
2 [dependencies]
3 rand = "0.8.5"
```

```
1 fn main() {
2     let rand_num: i32 = rand::random();
3     println!("your lucky number: {}", rand_num);
4 }
```

Glob Imports

```
1 use std::collections::*;
2
3 fn main() {
4     let hm: HashMap;
5     let bm: BTreeMap;
6 }
7
8 // common pattern for framework-like libraries:
9 use leptos::prelude::*;
```

Re-Exporting with `pub use`

```
1 mod garden {
2     mod fruits {
3         pub struct Melon;
4         pub struct Orange;
5     }
6     pub use fruits::Melon;
7 }
8 fn main() {
9     let m: garden::Melon;
10    // doesn't work, because garden doesn't re-export Orange
11    let o: garden::Orange;
12    // doesn't work, because fruits is private
13    let o: garden::fruits::Melon;
14 }
```

Renaming Imports with `as`

```
1 mod some_lib {  
2     pub struct AnnoyinglyLongIdentifier;  
3 }  
4 mod abbr {  
5     pub use super::some_lib::AnnoyinglyLongIdentifier as Ali;  
6 }  
7 fn main() {  
8     let same_thing: some_lib::AnnoyinglyLongIdentifier = abbr::Ali;  
9 }
```

Summary

modules and visibility

- `main.rs` is the root of the crate.
- Modules can be defined inline or in separate files.
- The module tree is traversed with `::`, `super` and `crate`.
- Items are private unless made `pub`-lic.
- Verbose module path specifiers are avoided with `use`.
- Re-exporting allows fine-grained control over visibility.

Practice

`exercises/day_2.md`

Collections

book chapter 8

Reading from Vectors Safely

```
1 fn main() {
2     let v = vec![1, 2, 3, 4, 5];
3
4     v[10]; // panics immediately! 🤖
5
6     let tenth: Option<&i32> = v.get(9); // safety: 100
7     match tenth {
8         Some(tenth) => println!("The tenth item is {}", tenth),
9         None => println!("There is no tenth item."),
10    }
11 }
```

HashMap

a.k.a. dict , map , hash table, associative array

```
1 use std::collections::HashMap;
2
3 fn main() {
4     let mut scores = HashMap::new();
5
6     scores.insert("Brazil", 1);
7     scores.insert("Germany", 7);
8
9     let germany_score: Option<i32> = scores.get(&"Germany");
10
11     for (key, value) in scores {
12         println!("{}", scored {} points!", key, value);
13     }
14 }
```

`std::collections::*`

- Queue
- Linked-List
- Set
- Heap

Error Handling

book chapter 9

Quick and Dirty

stack unwinding

```
1 fn main() {  
2     let v = vec![1, 2, 3];  
3     let x: i32 = v[100]; // <- panic  
4     let x: Option<i32> = None;  
5     let x: i32 = x.unwrap(); // <- panic  
6     panic!("custom panic message");  
7     todo!(); // shush compiler, I'm not done  
8 }
```

The Result Enum

```
1  enum Result<T, E> {  
2      Ok(T),  
3      Err(E),  
4  }
```

recall the Option type:

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

Error Handling

demo

```

1  #[derive(Debug)]
2  enum AreaError {
3      BadSeparator,
4      BadInteger(std::num::ParseIntError),
5  }
6  fn main() {
7      match calculate_area("12x8") {
8          Ok(area) => println!("area: {area}"),
9          Err(e) => match e { /* compiler ensures all errors are handled */ }, }
10 }
11 fn calculate_area(input: &str) -> Result<usize, AreaError> {
12     let (length, width) = match input.split_once('x') {
13         Some(t) => t,
14         None => return Err(AreaError::BadSeparator),
15     };
16     Ok(parse_number(length)? * parse_number(width)?)
17 }
18 fn parse_number(num: &str) -> Result<usize, AreaError> {
19     match num.parse() {
20         Ok(n) => Ok(n),
21         Err(e) => Err(AreaError::BadInteger(e)), } }

```

Generics

book chapter 10.1

```
1 let b: bool = Some(true).unwrap();  
2 let c: char = Some('x').unwrap();
```

Can we write `unwrap` ourselves?

The Problem: Duplication

```
1  fn my_unwrap_bool(opt: Option<bool>) -> bool {  
2      match opt {  
3          Some(val) => val,  
4          None => panic!(),  
5      }  
6  }  
7  
8  fn my_unwrap_char(opt: Option<char>) -> char {  
9      match opt {  
10         Some(val) => val,  
11         None => panic!(),  
12     }  
13 }
```

C-style type erasure

```
1  struct Option {  
2      bool some;  
3      void* value;  
4  };  
5  void* unwrap(struct Option* opt) {  
6      if (opt->some) {  
7          return opt->value;  
8      }  
9      __builtin_trap(); // None >_  
10 }
```

🐢 requires pointer for fixed memory layout

⚠️ requires user to type-cast `void*` result

The Solution: Generics

```
1 fn my_unwrap<T>(opt: Option<T>) -> T {  
2     match opt {  
3         Some(val) => val,  
4         None => panic!(),  
5     }  
6 }  
7  
8 my_unwrap(Some(true));  
9 my_unwrap(Some('x'));
```

Generics in Structs

```
1 struct Point<T> {  
2     x: T,  
3     y: T,  
4 }  
5  
6 fn main() {  
7     let integer = Point { x: 5, y: 10 };  
8     let float = Point { x: 1.0, y: 4.0 };  
9     let mix = Point { x: 1.0, y: 10 };  
10    //                ^^  
11    // mismatched types: expected float  
12 }
```

Multiple Generic Type Parameters

```
1 struct Point<T, U> {  
2     x: T,  
3     y: U,  
4 }  
5  
6 fn main() {  
7     let mix = Point { x: 1.0, y: 10 };  
8     // inferred type: Point<f64, i32>  
9 }
```

Generics in Enums

```
1  enum Result<T, E> {  
2      Ok(T),  
3      Err(E),  
4  }
```

Generics in Methods

```
1 struct Point<T> {  
2     x: T,  
3     y: T,  
4 }  
5  
6 impl Point<u32> {  
7     fn x(&self) -> &u32 {  
8         &self.x  
9     }  
10 }
```

Generics in Methods

```
1  struct Point<T> {  
2      x: T,  
3      y: T,  
4  }  
5  
6  impl<T> Point<T> {  
7      fn x(&self) -> &T {  
8          &self.x  
9      }  
10 }
```

Performance?

Generics are resolved at compile time.

Generic code is essentially copy-pasted for every type parameter.

There is zero runtime cost to using generics.

(just like C++ templates)

Practice

`exercises/day_2.md`

Traits

book chapter 10.2

What are Traits?

German: Eigenschaft, Merkmal

Traits fulfill the same purpose as interfaces in other languages.

They enable polymorphism by specifying shared behavior.

(Rust does not have OOP-style class inheritance.)

Problem: T is useless

```
1  fn find_largest<T>(list: &[T]) -> &T {  
2      let mut largest = &list[0];  
3  
4      for item in list {  
5          if item > largest {  
6              largest = item;  
7          }  
8      }  
9  
10     largest  
11 }
```

compiler says:

binary operation > cannot be applied to type &T

Define Shared Behavior

```
1 trait Comparable {  
2     fn is_greater_than(&self, other: &Self) -> bool;  
3 }
```

Implementation for Concrete Types

```
1  impl Comparable for i32 {  
2      fn is_greater_than(&self, other: &Self) -> bool {  
3          self > other  
4      }  
5  }  
6  impl Comparable for i64 {  
7      fn is_greater_than(&self, other: &Self) -> bool {  
8          self > other  
9      }  
10 }
```

Constrain Generic Type Parameters

```
1  fn find_largest<T: Comparable>(list: &[T]) -> &T {  
2      let mut largest = &list[0];  
3  
4      for item in list {  
5          if item.is_greater_than(largest) {  
6              largest = item;  
7          }  
8      }  
9  
10     largest  
11 }
```

Default Implementations

```
1 trait Comparable {  
2     fn is_greater_than(&self, other: &Self) -> bool;  
3  
4     fn is_less_than_or_equal(&self, other: &Self) -> bool {  
5         !self.is_greater_than(other)  
6     }  
7 }
```

Multiple Trait Bounds

```
1 fn find_largest<T: Comparable + Debug>(list: &[T]) -> &T {  
2     // ...  
3     println!("found {largest:?}!");  
4     largest  
5 }
```

Where Clauses

```
1 // hard to read
2 fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {
3
4 // much better
5 fn some_function<T, U>(t: &T, u: &U) -> i32
6 where
7     T: Display + Clone,
8     U: Clone + Debug,
9 {
```

More syntax sugar

```
1 fn some_function<T: Clone, U: Clone>(t: T) -> i32 {}  
2  
3 // equivalent, some people prefer this:  
4 fn some_function(t: impl Clone, u: impl Clone) -> i32 {}
```

Blanket Implementations

demo

```
1  fn main() {
2      42.print_10_times();
3      3.14.print_10_times();
4      "hello".print_10_times();
5  }
6
7  trait Print10Times {
8      fn print_10_times(&self);
9  }
10
11 impl<T: std::fmt::Display> Print10Times for T {
12     fn print_10_times(&self) {
13         for _ in 0..9 {
14             print!("{self}, ");
15         }
16         println!("{self}");
17     }
18 }
```

Useful Traits

`Debug` string-representation for debugging

`Clone` & `Copy` can be copied (cheaply)

`Default` has default value (zero, empty string)

`PartialEq` & `Eq` can check for equality (`==` , `!=`)

`PartialOrd` & `Ord` can be ordered (`<` , `>` etc.)

`Hash` can compute hash (for `HashMap` etc.)

not derivable: `Display` , `From` & `TryFrom`

Useful Traits

```
1  #[derive(Debug, Clone, Copy, Default, PartialEq, Eq, PartialOrd, Ord, Hash)]
2  struct Useful {
3      a: i32,
4      b: bool,
5  }
```

Lifetimes

book chapter 10.3

Reminder:

Rust forbids invalid references

```
1  fn main() {  
2      let r;                                // -----+-- 'a  
3                                          //      |  
4      {                                    //      |  
5          let x = 5;                       // -+-- 'b |  
6          r = &x;                         //  |   |  
7      }                                    // -+   |  
8                                          //      |  
9      println!("r: {}", r);               //      |  
10 }
```

'a' and 'b' are the "lifetimes" of r and x .

Lifetime annotations

demo

```
1 fn borrow_left<'left>(left: &'left str, right: &str) -> &'left str {
2     left
3 }
4 fn borrow_right<'right>(left: &str, right: &'right str) -> &'right str {
5     right
6 }
7 fn main() {
8     let borrowed;
9     let left = String::from("left");
10    {
11        let right = String::from("right");
12        borrowed = borrow_left(&left, &right);
13    }
14    println!("{borrowed}");
15 }
```

Merging parameter lifetimes

```
1 fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {  
2     if x.len() > y.len() {  
3         x  
4     } else {  
5         y  
6     }  
7 }
```

Lifetime Annotations

```
1 fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {  
2     if x.len() > y.len() {  
3         x  
4     } else {  
5         y  
6     }  
7 }
```

There is some lifetime `'a`.

Lifetime Annotations

```
1 fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {  
2     if x.len() > y.len() {  
3         x  
4     } else {  
5         y  
6     }  
7 }
```

x lives at least for 'a .

Lifetime Annotations

```
1 fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {  
2     if x.len() > y.len() {  
3         x  
4     } else {  
5         y  
6     }  
7 }
```

y lives at least for 'a .

Lifetime Annotations

```
1 fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {  
2     if x.len() > y.len() {  
3         x  
4     } else {  
5         y  
6     }  
7 }
```

The returned reference lives at least for 'a .

Lifetime Annotations

```
1 fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {  
2     if x.len() > y.len() {  
3         x  
4     } else {  
5         y  
6     }  
7 }
```

In essence:

The lifetime of the returned reference
is the shorter one of `x` and `y` 's lifetimes.

Limitations

```
1  fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
2      if x.len() > y.len() { x } else { y }
3  }
4
5  fn main() {
6      let output;
7      let x = String::from("Hello world");
8      {
9          let y = String::from("Hello");
10         output = longest(&x, &y);
11     }
12     println!("output: {}", output);
13 }
```

compiler says:

y does not live long enough

...but we know this would be OK at runtime.

LT Annotations in Structs

```
1 struct StoringBorrowedData<'number, 'text> {  
2     n: &'number i32,  
3     s: &'text str,  
4 }
```

Lifetime Elision Rules

(slightly simplified)

★ For a single parameter, its lifetime is assigned to all outputs.

```
1 fn foo(single_arg: &str) -> &str {}  
2 // is the same as  
3 fn foo<'a>(single_arg: &'a str) -> &'a str {}
```

★ For methods, the lifetime of `self` is assigned to all outputs.

```
1 impl Whatever {  
2     fn foo(&self, second_arg: &str) -> &str {}  
3     // is the same as  
4     fn foo<'a>(&'a self, second_arg: &str) -> &'a str {}  
5 }
```

The 'static' Lifetime

```
1  static GREETING: &'static str = "hello world";
2
3  fn main() {
4      let greeting: &'static str = "hello world";
5
6      let answer: &'static i32;
7      {
8          let heap_alloc = Box::new(42);
9          answer = Box::leak(heap_alloc); // explicit memory-leak
10     }
11     println!("answer: {}", answer);
12 }
```

Hint for Practice Session

```
1 // a test that ensures an expected panic occurs
2 #[test]
3 #[should_panic]
4 fn expected_panic_occurs() {
5     let v = vec![1, 2, 3];
6     v[10];
7 }
```

Practice

`exercises/day_2.md`