Organization

Language?

- German
- English

Rust Experience

1. Never Programmed Rust
2. Dabbled with Rust
3. Wrote project with Rust

Microcontroller Experience

1. Never programmed a microcontroller
2. Programmed a microcontroller with
   - Java
   - Arduino
   - Mindstorms
3. Programmed a microcontroller with
   - Ada
   - C/C++
   - Rust

Please distribute yourself

- Don‘t sit next to someone with your own skill level
- Help each other

Schedule

- Today
  - Rust basics
- Tomorrow
  - more Rust basics
  - Split into Groups
  - Rust on a microcontroller
- Wednesday
  - How to make an LED blink
  - Touchscreen basics
- Thursday
  - Project selection

Final Presentation

- Next week Thursday
- Present your Project
- 15 minutes
  - 10 min Implementation
  - 5 min Showcase

- 10 Groups -> 3 hours
- Stay until the end (12:00)

Rust Basics

The Rust Programming Language

- Syntax based on C
- Standard library similar to C++11
- Libraries (crates) are part of the language
- No nasal demons (undefined behaviour)

- Automatic code formatting (rustfmt)
- Automatic common bug detection (clippy)

- Automatic download and compilation of dependencies

Rust Compiler installation: Step [1 of 13]

Windows

https://win.rustup.rs/

Mac/Linux

curl https://sh.rustup.rs –sSf | sh

Rust Compiler installation: Step [2 of 17]

1. Press Return
2. Wait
3. Press Return

Rust Compiler installation: Step [3 of 15]

- Windows:
  - Restart computer

- Mac/Linux
  - Open a new command line window

~~Step [4 of 21]:~~ just kidding, you‘re done

Rustup

- is a compiler manager
  - installs cross compilers
  - updates compiler
  - installs unstable bleeding edge compilers if requested
  - allows using multiple different compilers on one computer
- opens the Rust documentation in the browser
  - rustup component add rust-docs
  - rustup doc

Hello Rust

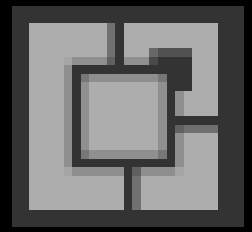
- cargo new –bin my_hello_world
  - All libraries in Rust are named in snake_case
  - No capital letters
  - Don‘t use letters other than a-z, 0-9 or underscore
    - English keyboard users will be thankful
- cd my_hello_world
- cargo run
  - Compiles and executes your project
- You should now see „Hello, world!“ written on your command line

Editor Wars

- ~~My editor is always the better one~~
- All examples shown in vscode
- Use other editors at your own risk and effort
- Full list of editors: http://stackoverflow.com/tags/rust/info

Visual Studio Code Tutorial

- Download from https://code.visualstudio.com/
- Install
- Click the Extensions icon

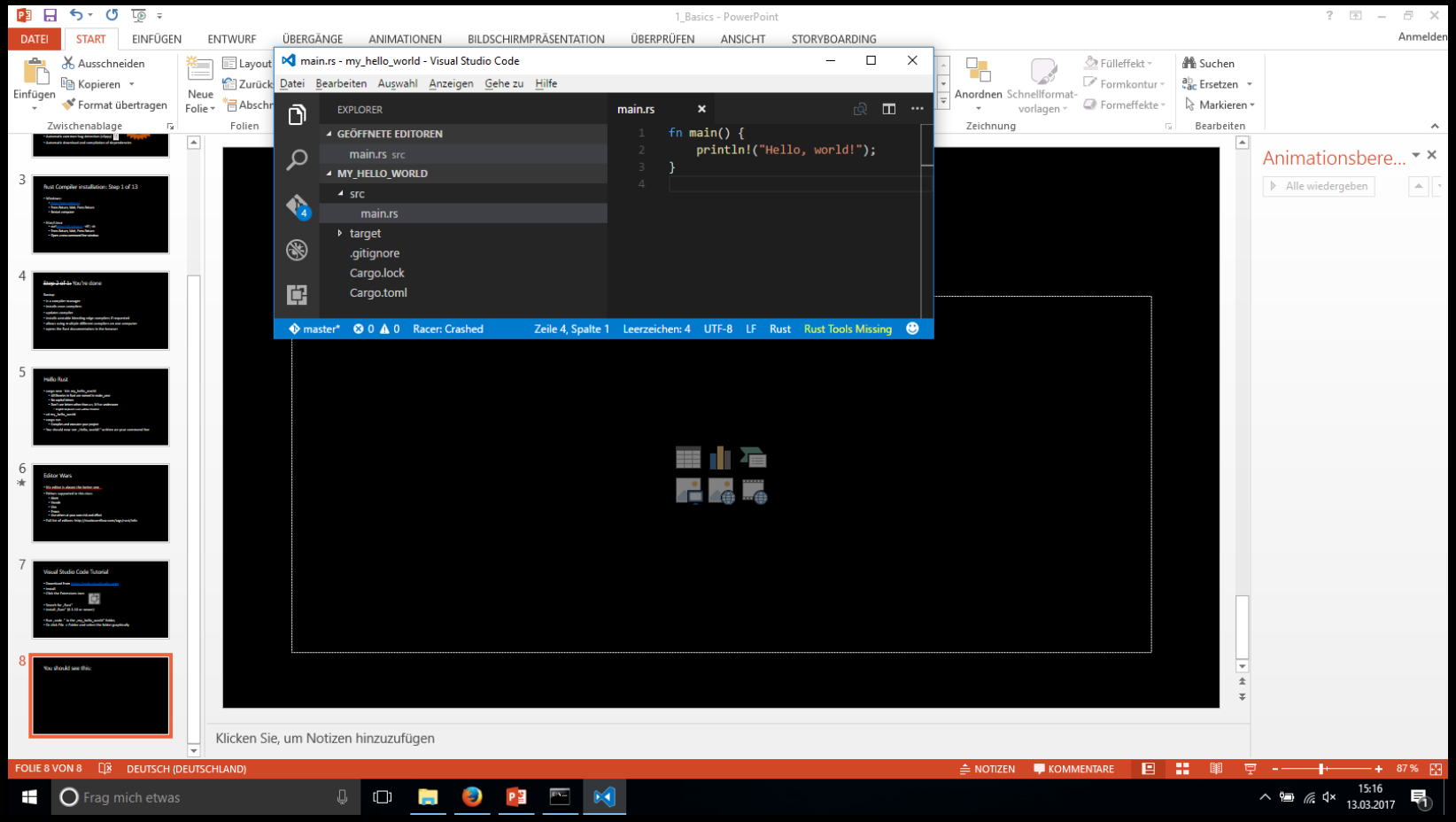


- Search for „Rust“
- Install „Rust“ (0.3.10 or newer)

- File -> Automatically Save
  - Congratulations, you‘ll never have to press Ctrl + S again

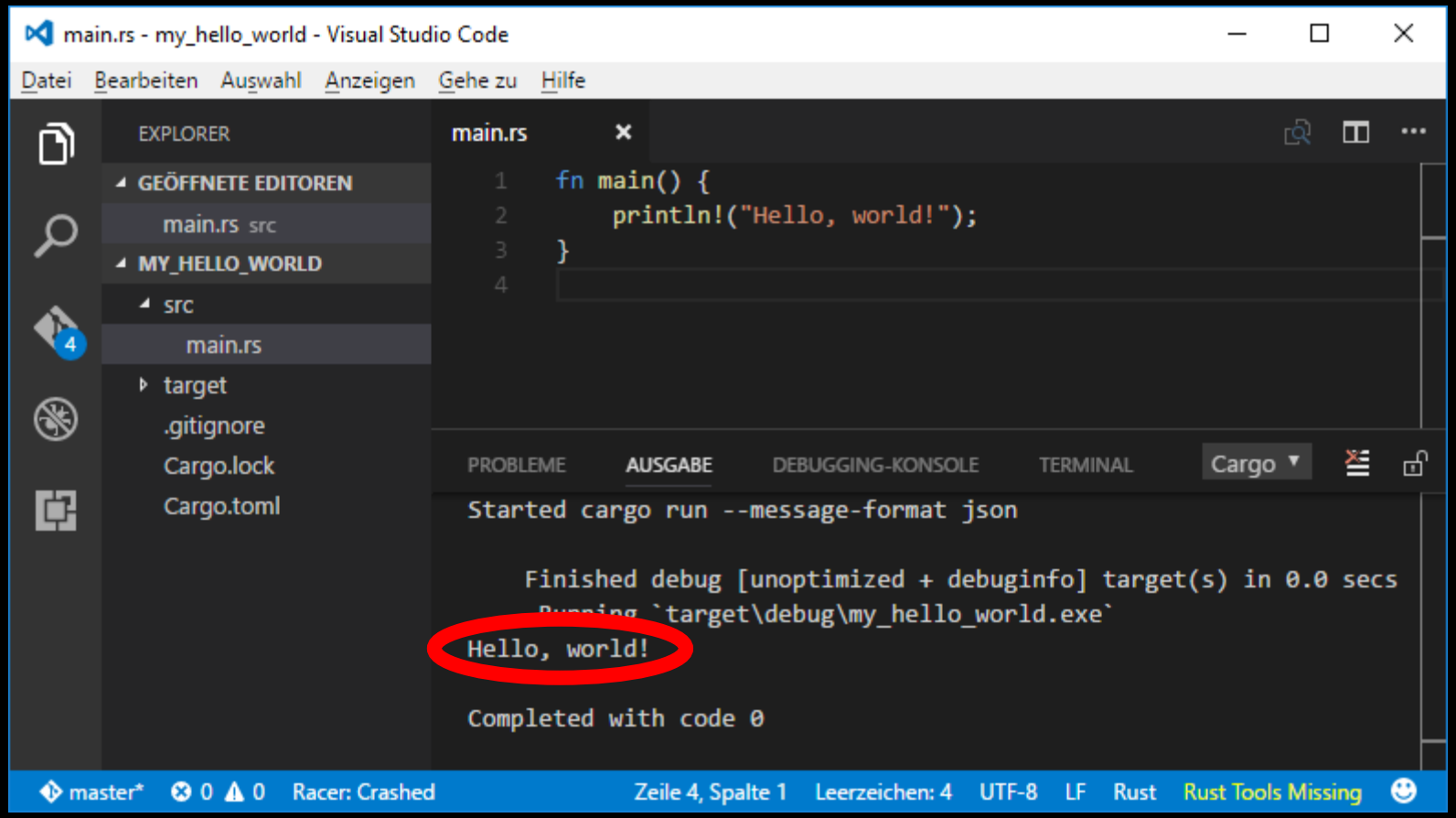
- Run „code .“ in the „my_hello_world“ folder,
- Or click File -> Folder and select the folder graphically

You should see this:



Running from vscode

- Ctrl + Shift + R



What‘s going on?

The starting point of any Rust program is the „main“ function

A Function is denoted by the keyword „fn“

The main function has no arguments

```
fn main() {
    println!("Hello, world!");
}
```

A standard library macro to help you write stuff to the command line

Strings are declared with double quotes

Rust doesn‘t have variables

- It has bindings
- We‘ll get to those later
- Let‘s call them variables for now

Placeholder like %d
in C format strings

```
let x = 5;
println!("My Number: {}", x);
```

- A „let“ binding infers its type where possible

Bindings are immutable by default

- C++ suggests to use „const“ wherever possible.
- Statistics on real code show that most variables are or could be const

- Don‘t accidentally modify a variable you did not intend to modify

```
let x = 5;

E0384: re-assignment of immutable variable `x`
  label: re-assignment of immutable variable
x = 6;
println!("My Number: {}", x);
```

Mutable binding

```rust
fn main() {
    let mut x = 5;
    println!("My Number: {}", x);
    x = 6;
    println!("My Number: {}", x);
}
```

PROBLEME AUSGABE DEBUGGING-KONSOLE TERMINAL Cargo

```
    Finished debug [unoptimized + debuginfo] target(s) in 0.90 secs
     Running `target\debug\my_hello_world.exe`
My Number: 5
My Number: 6

Completed with code 0
```

Variable names

- All variable names are in snake_case
- No capital letters
- No greek letters (yet)
- Cannot start with a number
- No exceptions!
  - Don‘t use other styles in Rust

Interlude – Better error messages

- Language Server Protocol
- rustup update nightly-2017-03-28
  - Living on the edge
- git clone https://github.com/rust-lang-nursery/rls.git
  - Or unpack https://github.com/rust-lang-nursery/rls/archive/master.zip
- „rustup default nightly-2017-03-28“

Interlude – Better error messages

- Install Visual Studio
  - https://www.visualstudio.com/de/vs/community/
- Install CMake
  - Windows: https://cmake.org/download/
    - Add CMake to the system PATH for all users
    - Restart
  - Debian: apt-get install cmake
  - Mac: brew install cmake
- Install openssl(-dev) on mac & linux (windows uses the native ssl)

Interlude – Better error messages

- Run „cargo build“ in the rls folder
- Patience…
- Windows:
  - add C:\Users\user\.rustup\toolchains\nightly-2017-03-28-x86_64-pc-windows-msvc\bin to PATH

Interlude – Better error messages

- File -> Settings -> Settings

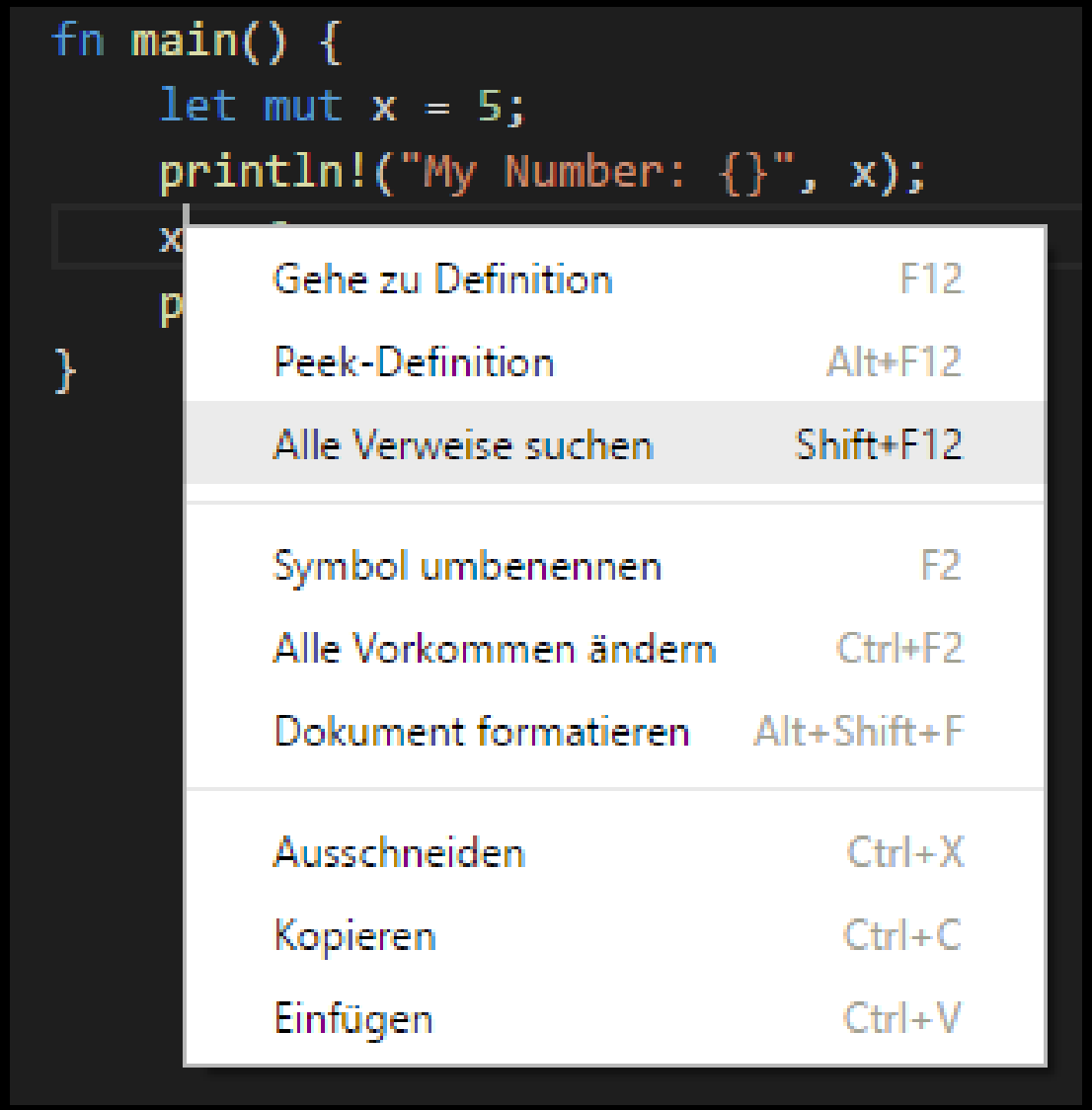
```
{
    "workbench.welcome.enabled": false,
    "files.autoSave": "afterDelay",
    "rust.rls": {
        "executable": "cargo",
        "args": ["+nightly-2017-03-28", "run", "--manifest-path=C:/Users/oliver/rls/Cargo.toml"]
    }
}
```

- Restart vscode

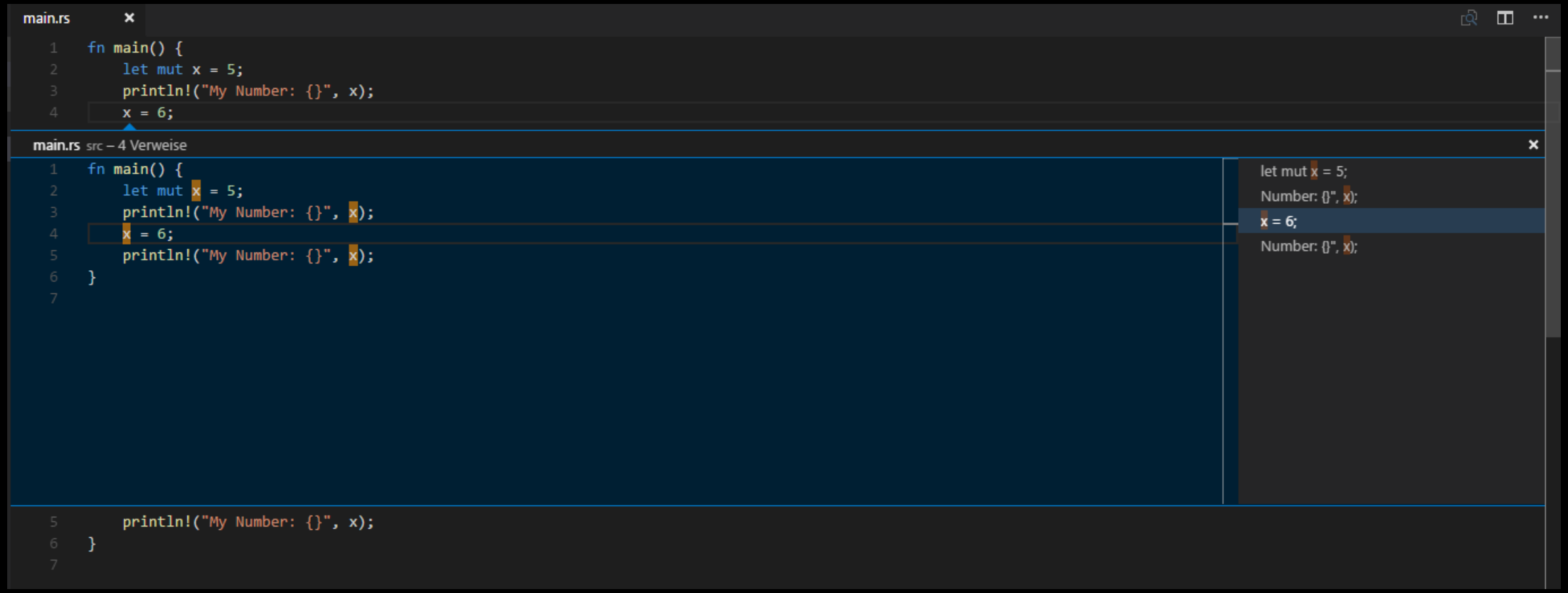
Interlude – Better error messages

- Errors now show up as you type
  - No need to manually press Ctrl + Shift + B anymore
- IDE-Features:
  - Go to Definition
  - Find all References
  - Rename Symbol

Find all References

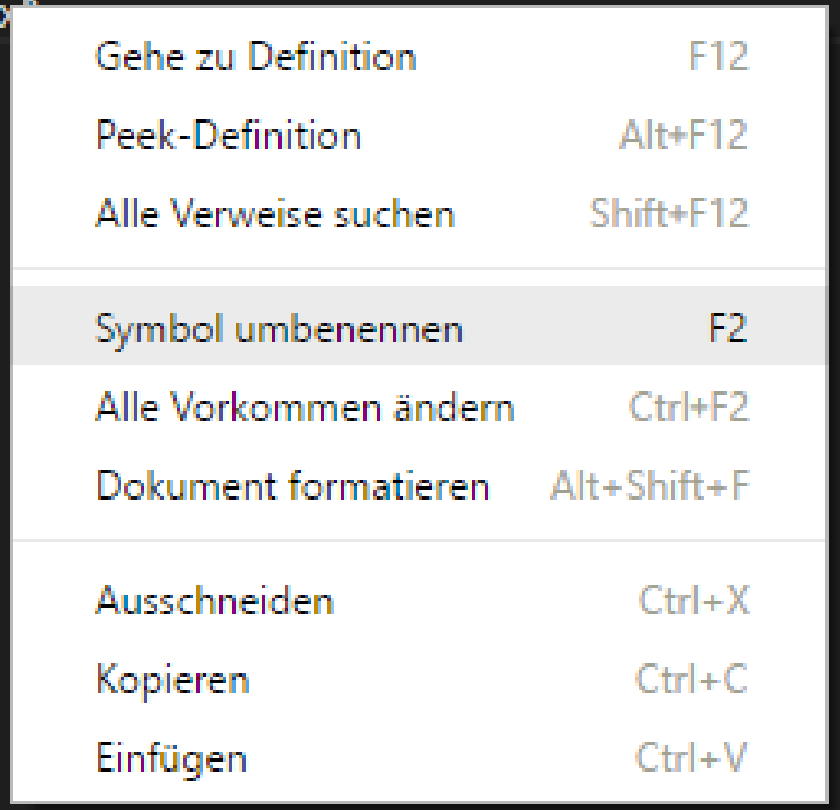


Find all References

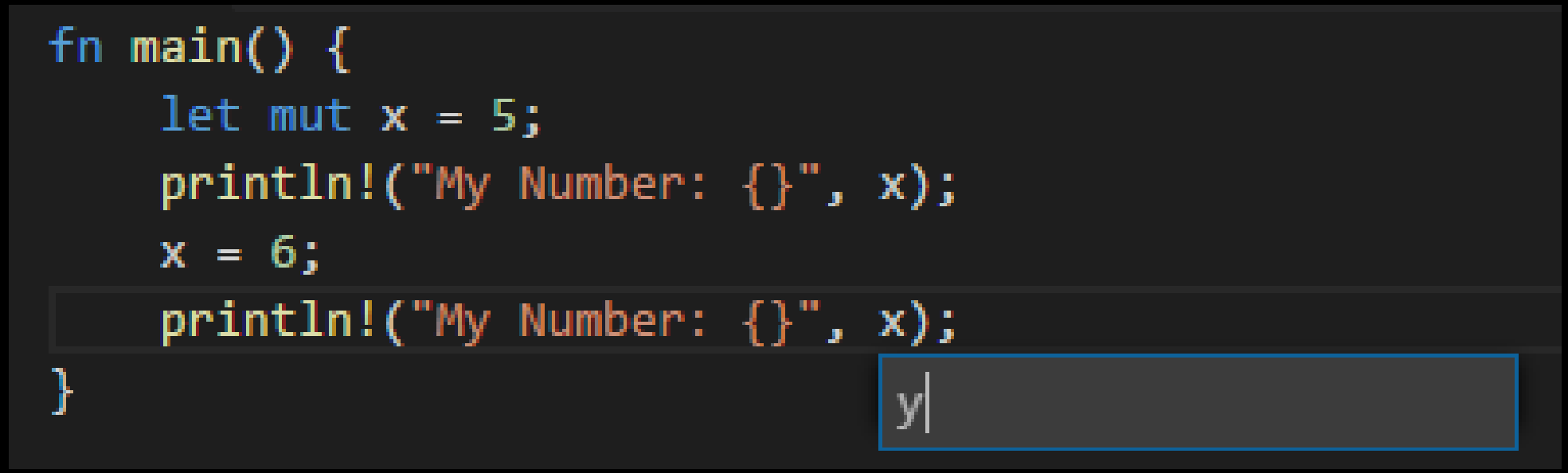


Rename Symbol

```rust
fn main() {
    let mut x = 5;
    println!("My Number: {}", x);
    x = 6;
    println!("My Number: {}", x);
}
```



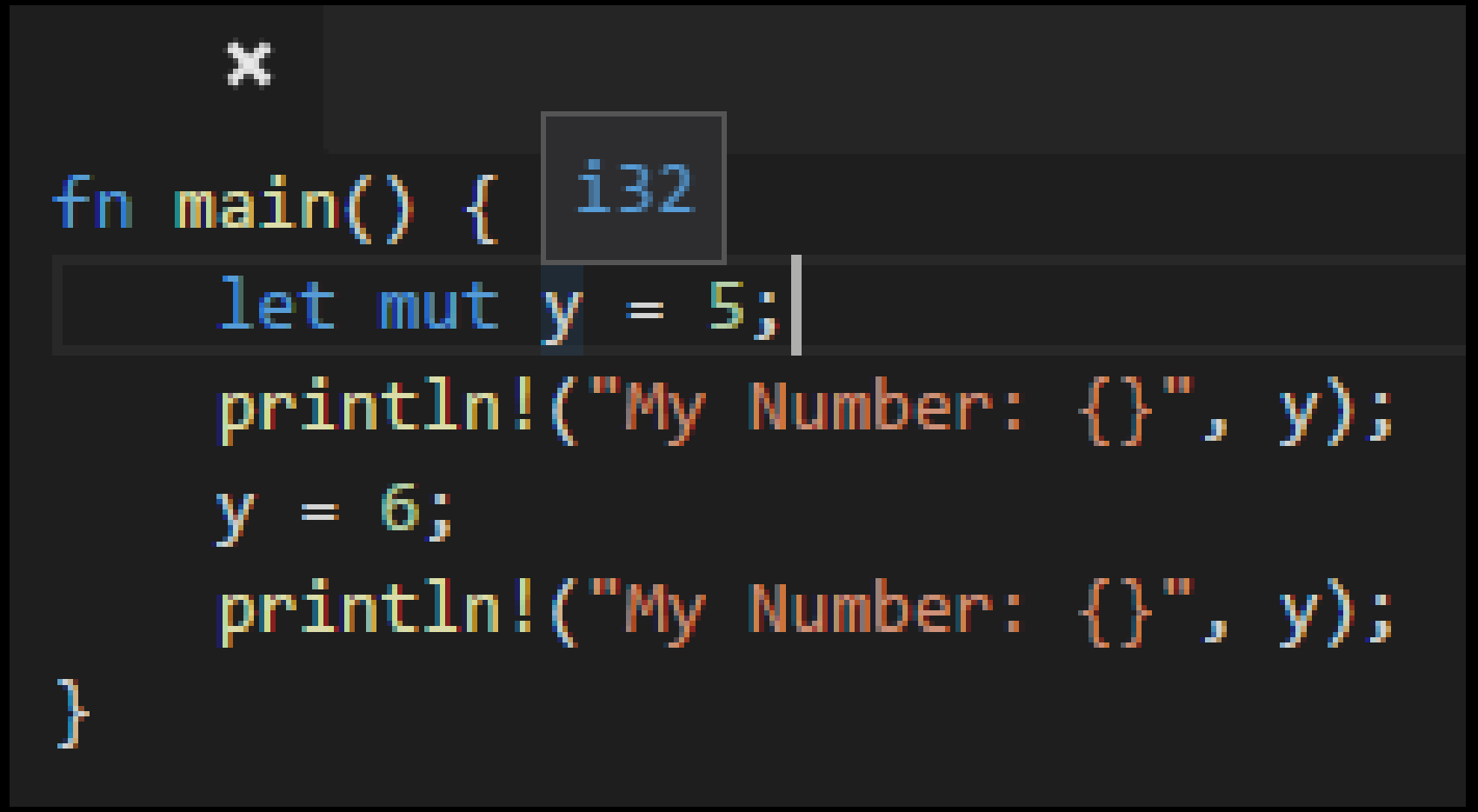
Rename Symbol



Rename Symbol

```rust
fn main() {
    let mut y = 5;
    println!("My Number: {}", y);
    y = 6;
    println!("My Number: {}", y);
}
```

Show type of $anything



Builtin Types

- Signed integers
  - i8, i16, i32, i64, isize(, i128)
- Unsigned integers
  - u8, u16, u32, u64, usize(, u128)
- Floating point
  - f64, f32

Variables with explicit types

```
fn main() {
    let mut y: u64 = 5;
    println!("My Number: {}", y);
    y = 6;
    prin
    y = -3;
}
```

```
[rustc] cannot apply unary operator `-` to type `u64`
```

Functions

A Function is denoted by the keyword „fn“

Function names, like variable names, are camel_case

Arguments are declared like variables, but must have an explicit type

```
fn square(i: i32) -> i32 {
    i * i
}
```

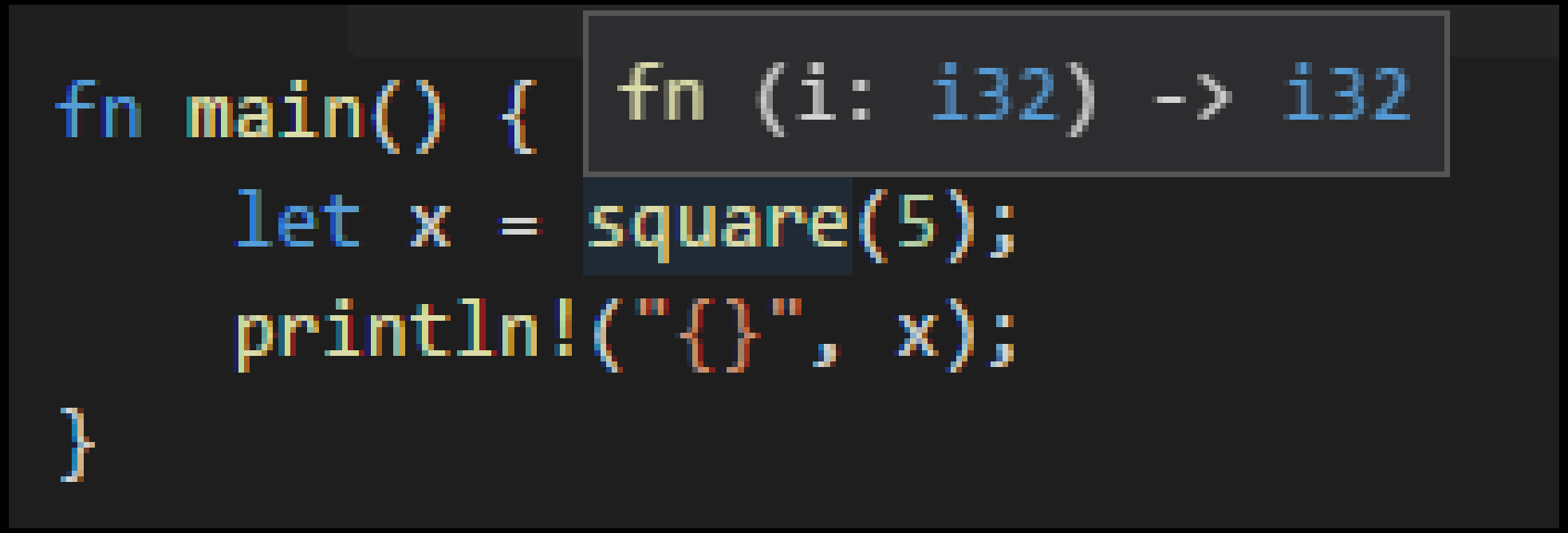
No return statement needed
Functions look like Mathematical functions

The return type is declared after the „arrow“ (->) like in C++11

Function calls

```
fn main() {
    let x = square(5);
    println!("{}", x);
}
```

Inspect function type through tooltip



Control Flow: if

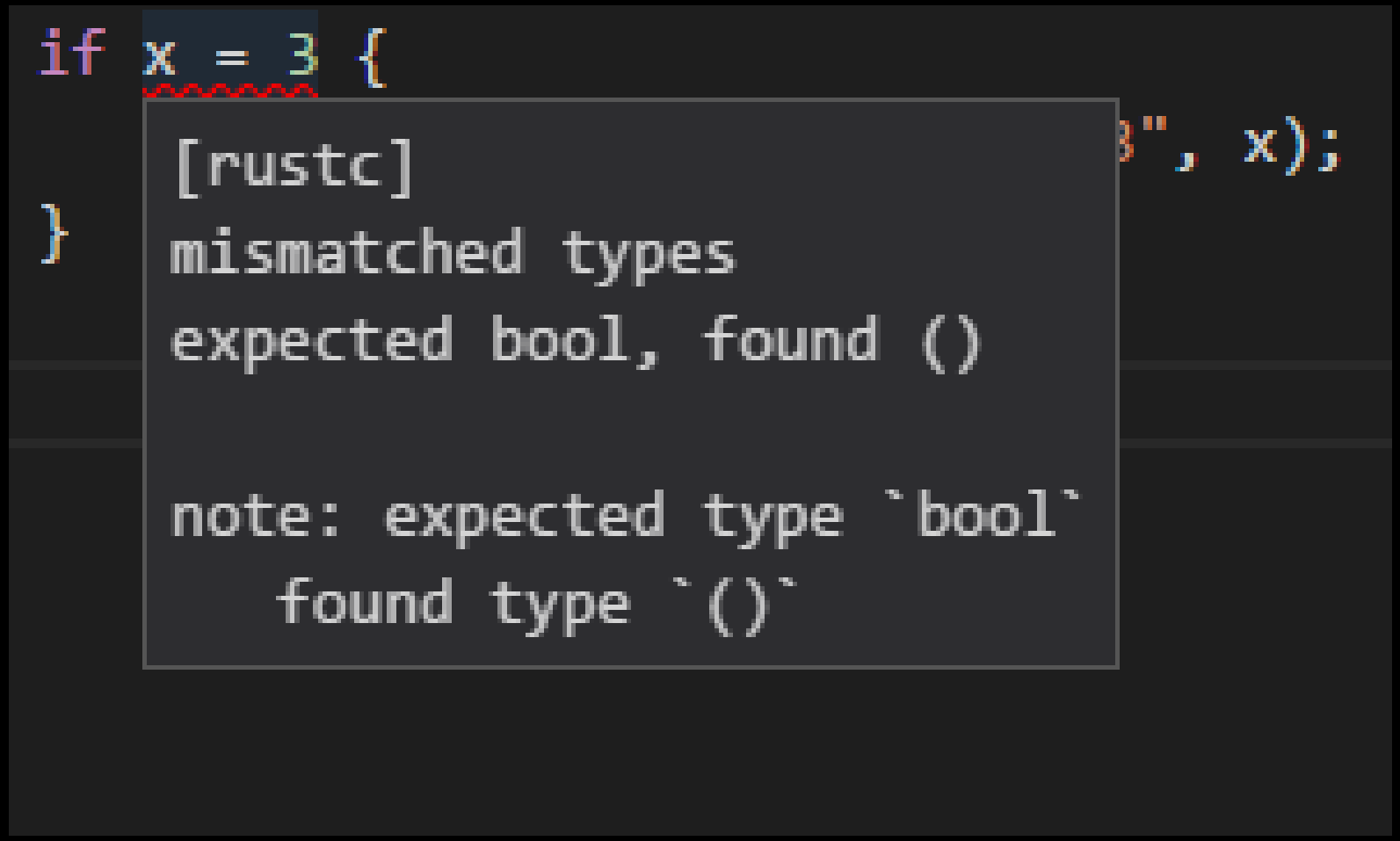
```
let x = 42;
if x < 3 {
    println!("{} is less than 3", x);
}
```

Common C-mistakes

```
fn main() {
    let x = 42;
    if (x < 3) println!("{} is less than 3", x);
}
```

```
[rustc]
expected `{`, found `println`

help: try placing this code inside a block
```

Common C-Mistakes

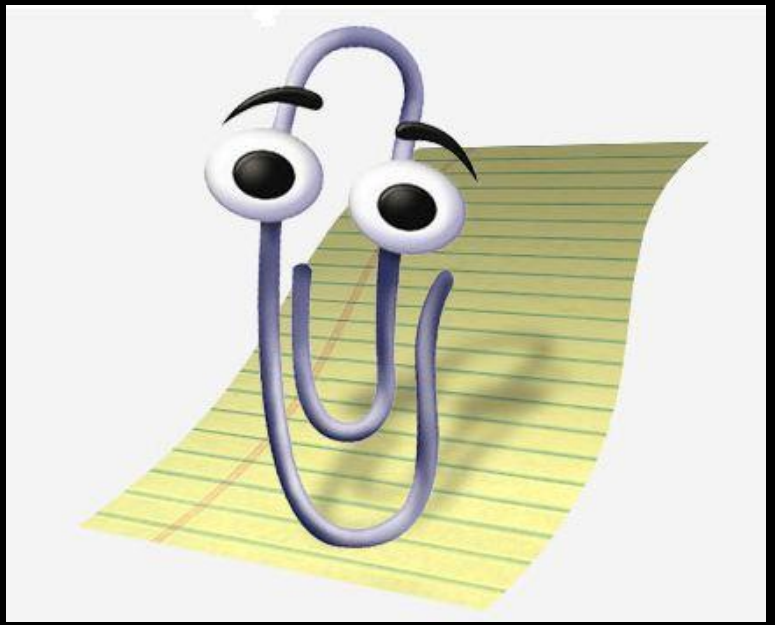


Excercise

- Write recursive fibonacci function
- fib(n) = fib(n – 1) + fib(n – 2)
- fib(0) = 1;
- fib(1) = 1;

Interlude: clippy

- Clippy detects common pitfalls in Rust code
- Complains about
  - „x == x“
  - „x + 0“
  - „return x;“ where „x“ suffices
  - … and many more
- This process is called „linting“
  - Removing knots of wool from sheep



Clippy

- „rustup run nightly-2017-03-28 cargo install clippy --vers 0.0.121“
- main.rs:
  - #![feature(plugin)]
  - #![plugin(clippy)]
- Cargo.toml
  - clippy = "0.0.121„
- Restart vscode
  - patience

Fibonacci improvements

```
[rustc]
unneeded return statement

note: #[warn(needless_return)] on by default
help: remove `return` as shown:
help: for further information visit https://github.com/Manishea
rth/rust-clippy/wiki#needless_return
```

```
[rustc]
this `if` has identical blocks

note: #[warn(if_same_then_else)] on by default
note: same as this
help: for further information visit https://github.com/Manishea
rth/rust-clippy/wiki#if_same_then_else
```

```rust
#![feature(plugin)]
#![plugin(clippy)]

fn main() {
    println!("{}", fib(5));
}

fn fib(n: u64) -> u64 {
    if n == 0 {
        return 1;
    } else if n == 1 {
        return 1;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

Better fibonacci

```
fn fib(n: u64) -> u64 {
    if n == 0 || n == 1 {
        1
    } else {
        fib(n - 1) + fib(n - 2)
    }
}
```

Learn to trust …

- … the compiler
  - Don‘t ignore warnings
  - DO NOT IGNORE WARNINGS
  - Really!
- … clippy
  - Style is important, since **you never hack alone**
    - Your future self will thank you, because they can read the code you wrote today
  - Readability is important
    - „check if the array‘s number of elements is zero“ vs „check if array is empty“
  - Even in Rust you can make logic mistakes
    - Clippy tries to detect the obvious cases so your brain can concentrate on the real issues

Aggregate Types

- Arrays
  - [T; N] is an array of N elements of Type T
  - [value; N] creates an array of length N with each element initialzed to „value“
  - Access elements with the index operation: „my_arr[42]“

- Structs
  - struct Foo { a: i32, b: u8 }
  - Struct names are CamelCase, struct fields are snake_case
  - No semicolon after the struct definition
  - Create values: „let s = Foo { a: -3, b: 42 };“
  - Access fields by name: „s.b = 99;“

Aggregate Types: Tuples

Tuples are anonymous structs

- (i32, u64, [i8; 3]) is a „struct“ with numbered field names
- „let x = (-1, 5, [3, 4, 5]);“ creates a tuple
  - No type declaration necessary
- „fn foo() -> (i32, u64)“
  - Functions returning multiple values at once
- Access fields with their index
  - „let y = x.1;“ takes the second field‘s value

Methods

- Functions that are tied to a concrete type
- Can only implement methods for your types
- Three types of methods:
  - Static (no object to modify)
    - Call with „TypeName::method_name(args)“
  - Mutating
    - Call with „object.method_name(args)“
  - Read only
    - Call with „object.method_name(args)“

```
fn main() {
    let mut foo = Foo::new();
    foo.double();
    foo.double();
    println!("{}", foo.get());
}

struct Foo {
    a: i32,
}

impl Foo {
    fn new() -> Foo {
        Foo {
            a: 99,
        }
    }
    fn double(&mut self) {
        self.a *= 2;
    }
    fn get(&self) -> i32 {
        self.a
    }
}
```

Loops

- loop { action }
  - Infinite loop
  - Runs until „break“ or „return“ statement reached
- for i in iter { action }
  - Iterate over each element in „iter“ and apply „action“
  - Access to current element throug „i“
- while condition { action }
  - Repeat „action“ as long as „condition“ hold

For loops

```
for i in 0..10 {
    println!("{}", i);
}
```

References

- Rust‘s builtin „pointers“ can never
  - Be a nullpointer
  - Point to invalid memory
  - Point to deallocated memory
  - Participate in pointer arithmetic
  - Provide you with ~~nasal demons~~ undefined behaviour
- Validity is proven at compile time
  - No runtime effort
  - No garbage collection
  - Your program is correct wrt memory safety or it won‘t compile

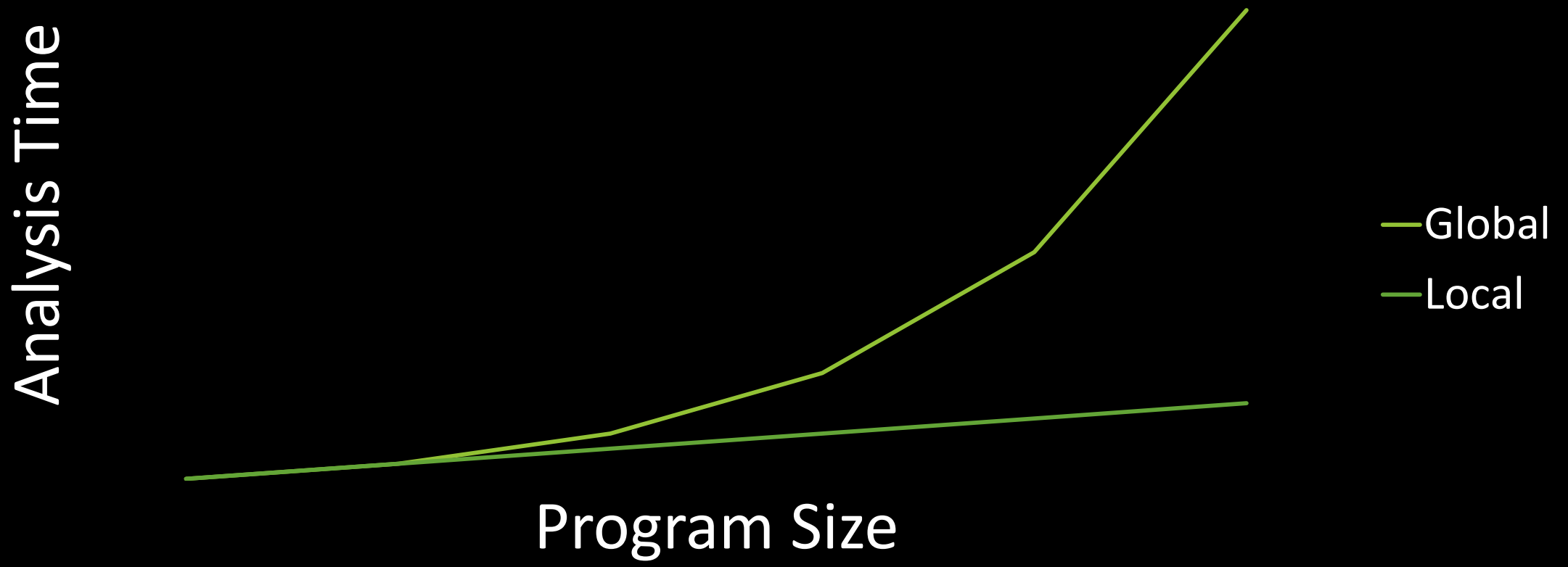
The Borrow Checker

- Proves that references are used correctly
- Stops at function boundaries

```
`z` does not live long enough
`z` dropped here while still borrowed
```

```
let x = 5;
let mut y = &x;
{
    let z = 6;
    y = &z;
}
println!("{}", y);
```

Local vs Global analysis



Local vs Global analysis

- Global Analyses require less programmer interaction
- Local Analyses provide local errors
  - Easier to comprehend
  - Easier to fix
- Global analyses can prove more code as correct
- Local analyses will sometimes declare correct code as „can‘t prove“
  - Still good enough for most purposes

Function boundaries

- „foo“ API states that result points to argument
- „main“ knows that „y“ points to „x“
  - Without looking at „foo“ code

```
fn main() {
    let x = 5;
    let y = foo(&x);
}

fn foo(i: &i32) -> &i32 {
    i
}
```

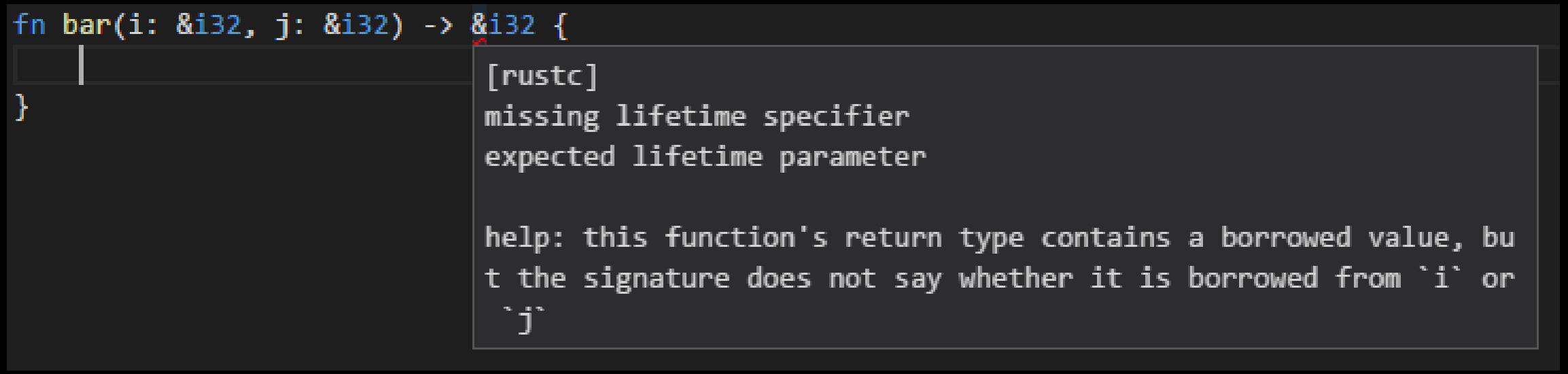
- „bar“ returns reference to local value
- But API states that it returns reference to argument
- Local analysis forbids this mismatch

```
`x` does not live long enough
```

```rust
fn bar(i: &i32) -> &i32 {
    let x = *i;
    &x
}
```

Ambiguities

- Even without code, the API is already ambiguous



Resolving Ambiguities

- Global analysis could solve this without user intervention
  - But would take hours or days depending on the project size
- Explicitly state which value the result depends on

```
fn bar<'a>(i: &'a i32, j: &i32) -> &'a i32 {
    i
}
```

Wrong body

```
fn main() {
    let x = 5;
    let z = 6;
    let y = bar(&x, &z);
}

fn bar<'a>(i: &'a i32, j: &i32) -> &'a i32 {
    j
}
```

Modification of z

```
fn main() {
    let x = 5;
    let mut z = 6;
    let y = bar(&x, &z);
    z = 9;
}

fn bar<'a>(i: &'a i32, j: &i32) -> &'a i32 {
    i
}
```

Modification of x

```
fn main() {
    let mut x = 5;
    let mut z = 6;
    let y = bar(&x, &z);
    z = 9;
    x = 99;
}

fn bar<'a>(i: &'a i32, j: &i32) -> &'a i32 {
    i
}
```

Result can point to either argument

```
fn main() {
    let x = 5;
    let z = 6;
    let y = bar(&x, &z);
}

fn bar<'a>(i: &'a i32, j: &'a i32) -> &'a i32 {
    if i > j {
        i
    } else {
        j
    }
}
```

Lifetimes

- Implicitly exist in C/C++
- By definition, lifetime violations do not exist in C/C++
  - If the programmer violates lifetime rules, undefined behaviour happens
- By definition, hands do not touch chainsaw blades
  - If you violate that rule, undefined behaviour happens

- Rust checks for violations of lifetime rules
  - Sometimes code is marked as violating rules even if it doesn‘t
  - Still better than loosing fingers

Heap Allocations

- Box<T>
  - Allocate a single value on the heap
  - let b = Box::new(42);
  - Access with dereference: let c = *b;
- Vec<T>
  - Allocate multiple values on the heap and change the number of elements dynamically
  - let mut v = vec![42, 43, 44];
  - v.push(99);
  - Access with index syntax: v[3]

Strings

- The „str“ type denotes a utf8 sequence with known length
- Only exists behind references:
  - let s: &str = „Strings can include äöüß and even ☺♪♪♪“;
- Resizable strings exist on the heap:
  - let mut s: String = String::from(„foo“);
  - s += „ bar“;
- Dynamically create strings from values
  - let s = format!(„abc: {}“, 42);
  - Same syntax as command line output

Ownership

- Types with ownership
  - String, Vec, custom types
  - Assignment invalidates the previous variable
- Types without ownership
  - Integers, floats, references
  - Assignment creates a copy

Ownership

```
let x = 5;
let y = x;
let z = x;

let a = "Hello".to_owned();
let b = a;

    [rustc]
    use of moved value: `a`
    value used here after move

    note: move occurs because `a` has type `std::string::String`, w
    hich does not implement the `Copy` trait

let c = a;
```

Explicit copying of owned objects

```
let x = 5;
let y = x;
let z = x;

let a = "Hello".to_owned();
let b = a.clone();
let c = a;
```

Clippy strikes again

```
fn main() {
    let
    let
    let
    let
    let
    let c = a;
}
```

```
[rustc]
6th binding whose name is just one char

note: #[warn(many_single_char_names)] on by default
help: for further information visit https://github.com/Manishearth/rust-clippy/wiki#many_single_char_names
```

Mutation

```
let a = "Hello".to_owned();

[rustc]
cannot borrow immutable local variable `a` as mutable
cannot borrow mutably
a += "!";
```

Rebinding

```
let a = "Hello".to_owned();
let mut b = a;
b += "!";
```

Borrowing

```
let x = 5;
let y = &x;
let z = *y;

let a = "Hello".to_owned();
let b = &a;

[rustc]
cannot move out of borrowed content
cannot move out of borrowed content
let c = *b;
```

Automatic dereferencing

```
let a = "Hello";
let b: &&&&&&&&str = &&&&&&&&a;
let c = b.to_owned();
```

Coercion

```
let a: String = "Hello".to_owned();
let b: &str = &a;
```

Clippy helps

```
fn main() {
    let a: String = "Hello".to_owned();
    let b: &String = &a;
    let c = foo(b);
}

[rustc]
writing `&String` instead of `&str` involves a new object where
 a slice will do. Consider changing the type to `&str`

note: #[warn(ptr_arg)] on by default
help: for further information visit https://github.com/Manishea
rth/rust-clippy/wiki#ptr_arg

fn foo(s: &String) -> usize {
    s.len()
}
```

Otherwise...

```
fn main() {
    let a: String = "Hello".to_owned();
    let b: &String = &a;
    let c = foo(b);

    let d = foo("bar");
}

fn foo(s: &String) -> usize {
    s.len()
}
```

```
[rustc]
mismatched types
expected struct `std::string::String`, found str

note: expected type `&std::string::String`
   found type `&'static str`
```

Solution

```
fn main() {
    let a: String = "Hello".to_owned();
    let b: &String = &a;
    let c = foo(b);
    let d = foo("bar");
}

fn foo(s: &str) -> usize {
    s.len()
}
```

Slices

- str
  - Sequence of utf8 characters
  - Corresponding owned type: String
- [T]
  - Sequence of elements of type T
  - Corresponding owned type: Vec<T>

- Prefer over owned type if only immutable access is needed
- Common interface instead of passing around a pointer and length

Slices

- Documentation of slices is found by searching for „slice“
  - slice (builtin type)
- Reminder:
  - Open documentation with „rustup doc --open“

[T] usage

- len() -> usize
  - Length
- is_empty() -> bool
- let x = &y[5..10];
  - Subslice including everything from the 6th element to the 10th inclusive
- let x = &y[..3];
  - From the start to the 3rd element inclusive
- let x = &y[2..];
  - From the 3rd element to the end
- let a = y[5];
  - Read the 6th element or abort program if index out of bounds

Assignment

- Write a binary search algorithm
- Work on a slice of u8
- https://en.wikipedia.org/wiki/Binary_search_algorithm
- Pring your slices with
  - println!(„{:?}“, slice);

Generics

- **Not** Templates
- **Not** Macros
- Compile a generic without knowing its „real“ types
- Functions and Types can be generic

Generic Function

```rust
fn main() {
    foo("bar");
    foo(42);
    foo(32.3);
}

fn foo<T>(value: T) {
    // don't know anything about T
    // Can't do much with "value"
}
```

Things that can be done with objects of type T

- Move it
- Take References to it
- Return it

Generics are useless?

- Add bounds to your generics

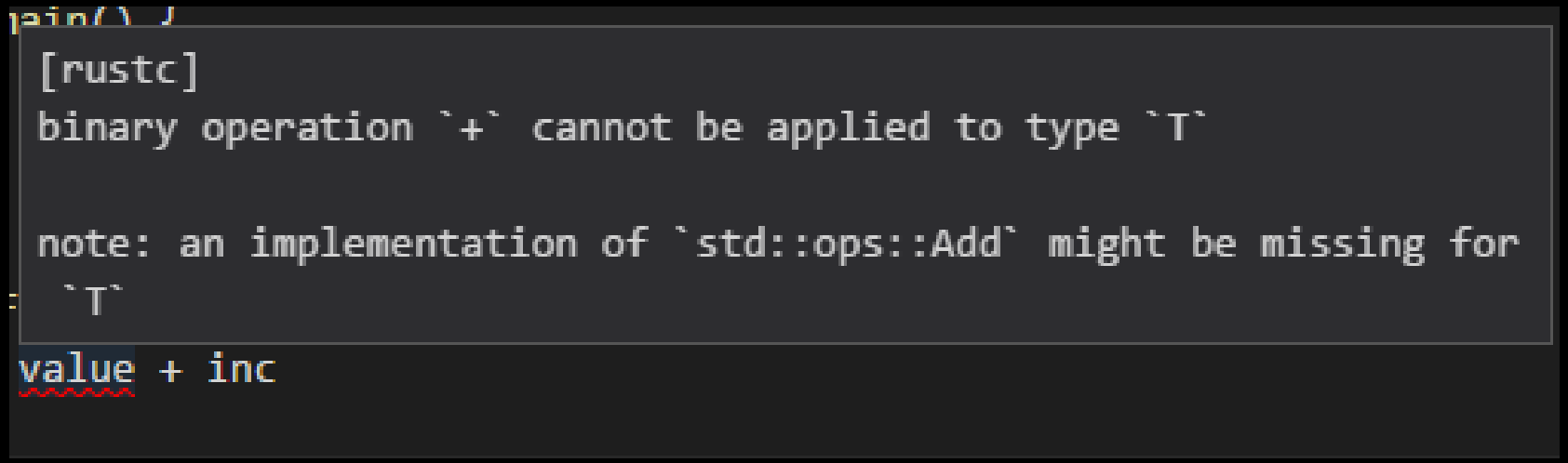
```
fn main() {
    foo("bar", "baa");
    [rustc]
    the trait bound `&str: std::ops::Add` is not satisfied
    the trait `std::ops::Add` is not implemented for `&str`

    note: required by `foo`
    foo(42, 5);
    foo(32.3, 3.14);
}

fn foo<T: std::ops::Add>(value: T, inc: T) -> T::Output {
    value + inc
}
```

Common Traits used as bounds

- std::ops::{Add, Sub, Mul, Div}
- std::cmp::{Eq, Ord}

Assignment

- Rewrite your search function with generics
- [T] instead of [u8]
- Apply bounds as you need them
  - The compiler will tell you which bounds you need



Polymorphism isn‘t inheritance

2 ways to do it in Rust

- Traits
  - Can add new types easily
- Enums
  - Can add new methods easily

Custom Traits

```rust
trait Dog {
    fn bark(&self);
    /// In Centimeter
    fn height(&self) -> f32;
}

struct Chihuahua;

impl Dog for Chihuahua {
    fn bark(&self) {
        println!("Whiff Whiff");
    }
    fn height(&self) -> f32 {
        20.0
    }
}
```

```rust
struct GreatDane;

impl Dog for GreatDane {
    fn bark(&self) {
        println!("Woof");
    }
    fn height(&self) -> f32 {
        75.0
    }
}
```

Enums

```rust
enum Dog {
    Chihuahua,
    GreatDane,
}

impl Dog {
    fn bark(&self) {
        match *self {
            Dog::Chihuahua => println!("Whiff Whiff"),
            Dog::GreatDane => println!("Woof"),
        }
    }
    /// In Centimeter
    fn height(&self) -> f32 {
        match *self {
            Dog::Chihuahua => 20.0,
            Dog::GreatDane => 75.0,
        }
    }
}
```

Accessing members

```rust
enum Dog {
    Chihuahua,
    GreatDane{ size_variation: f32 },
}

impl Dog {
    fn bark(&self) {
        match *self {
            Dog::Chihuahua => println!("Whiff Whiff"),
            Dog::GreatDane{..} => println!("Woof"),
        }
    }
    /// In Centimeter
    fn height(&self) -> f32 {
        match *self {
            Dog::Chihuahua => 20.0,
            Dog::GreatDane{ size_variation } => 75.0 + size_variation,
        }
    }
}
```

```rust
trait Dog {
    fn bark(&self);
    /// In Centimeter
    fn height(&self) -> f32;
}

struct Chihuahua;

impl Dog for Chihuahua {
    fn bark(&self) {
        println!("Whiff Whiff");
    }
    fn height(&self) -> f32 {
        20.0
    }
}

struct GreatDane {
    size_variation: f32,
}

impl Dog for GreatDane {
    fn bark(&self) {
        println!("Woof");
    }
    fn height(&self) -> f32 {
        75.0 + self.size_variation
    }
}
```

Standard library enums

- Option<T>
  - Either a value (Some) or no value (None)
  - Similar to pointer vs null pointer
  - let x = Some(val);
  - let x = None;

- Result<T, E>
  - Either a value (Ok) or an error (Err)
  - let x = Ok(val);
  - let y = Err(err);

Mutable References

- &mut T
  - There can be only one!
- Can coerce to &T
  - But then can‘t modify anymore

Assignment Bubblesort

- fn bubblesort(slice: &mut [u8]);
- https://en.wikipedia.org/wiki/Bubble_sort
- slice.swap(i, j);

Borrowed members

```
struct Foo<‘a> {
    elem: &‘a i32,
}
let i = 5;
let s = Foo { a: &i };
```

The static lifetime

- let x: &‘static str = „foo“;
- References with static lifetime are always valid

```
struct Foo {
    s: &‘static str,
}

let foo = Foo { s: „hello“ };
let s = String::from(„the cake is a lie“);
let bar = Foo { s: &s }; // ERROR
```

Iterators

- for element_ref in vec.iter()
- for (index, element_ref) in vec.iter().enumerate()
- for element_ref in vec.iter().rev()
  - Iterate from the back

- for c in str.chars()
- for c in str.chars().rev()

Owned iteration

```
let vec = vec![„foo“.to_owned(), „bar“.to_owned()];
for s in vec {
    let y: String = s;
}
```

Functional programming

```
let iter = 0..10;
let iter = iter.map(|elem| elem * 2);
for i in iter {
    println!(„{}“, i);
}
```

More functional programming

```
let iter = 0..10;
let iter = iter.filter(|elem| elem % 2 == 0);
for i in iter {
    println!(„{}“, i);
}
```

Closures

```
let mut x = 5;
let f = || {
    x += 3;
    x
};
let a = f();
let b = f();
x = 44; // ERROR
```

Functional programming

```
let x = (0..5).fold(3, |init, i| init + i);
```

Assignment: Iterators

- 5! = 5 * 4 * 3 * 2 * 1
- Implement with iterator methods

- Implement Binomial Coefficient function
- https://en.wikipedia.org/wiki/Binomial_coefficient

Assertions

- assert!(array.is_empty());
- assert_eq!(array.len(), 42);
- assert_ne!(array.len(), 3);
- assert!(cond > 0, „Why is cond not positive?“);

Unsafe code

- Breaking the Rules in a controlled manner
- Create safe abstraction
- Don‘t use unsafe if there‘s a safe alternative

- Rule: „don‘t use chainsaws“
  - Unsafe: using a chainsaw
  - Safe abstraction: experts use the chainsaw for you
    - In programming this means, that you can cut down a tree with a chainsaw safely, without needing an expert every time
    - The expert „pre-uses“ the chainsaw for you

Raw Pointers

- *const T
- *mut T
- Dereferenzierung funktioniert nur in unsafe code

Unsafe example

```
fn foo(slice: &mut [i32]) {
    unsafe {
        let x: *mut i32 = slice.as_ptr();
        let y = x.offset(1);
        assert!(slice.len() > 1);
        println!(„{}“, *y);
    }
}
```

Microcontrollers!

- Split into groups of 2-3
- Pick up one controller per group

Interlude – microcontroller flash tool

Install stlink

- arch: sudo pacman -S stlink
- general linux
  - install libusb-dev 1.0
  - install cmake
  - install a C-Compiler (sorry)
  - git clone https://github.com/texane/stlink.git && cd stlink && make release && cd build/Release && sudo make install
- mac os: brew install stlink
- windows: unzip https://github.com/embed-rs/stm32f7-discovery/blob/master/stlink-1.3.1-win32.zip

Interlude – cross compiler

- arch: sudo pacman -S arm-none-eabi-gcc arm-none-eabi-gdb
- debian/ubuntu: sudo apt-get install gcc-arm-none-eabi gdb-arm-none-eabi
- macOS: brew tap osx-cross/arm && brew install arm-gcc-bin
- windows:
  - download GNU ARM Embedded Toolchain from https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads
  - execute to install
  - ensure installation path is added to 'PATH' variable (might require a reboot)

Interlude – download rust source code

- rustup component add rust-src --toolchain nightly-2017-03-28
- if your rustup does not have the component subcommand
  - rustup self update
- Install cross cargo in parallel:
  - cargo install xargo

Interlude – Get the Demo code

- git clone https://github.com/embed-rs/stm32f7-discovery.git

Interlude - Compiling

1. cd stm32f7_discovery
2. rustup override set nightly
3. xargo build
   - have patience, the first time you run xargo build, the core library and various others need to be built.
4. open another terminal and run st-util
   - Windows: st-util.exe is located in stlink-1.3.1-win32\bin, which was unziped for setup
5. go back to your first terminal
6. run sh gdb.sh
   - run gdb.bat for win
7. The code has now been flashed and is ready to run.
   - Type c (for continue) and observe your controller.

println! on microcontrollers

- run semihosting-enable to enable semihosting support
- http://embed.rs/articles/2016/semi-hosting-rust/
  - Skip over the details for now, the gist is important