RUSTikales Rust for beginners

Plan for today

Plan for today

1. Recap

Plan for today

1. Recap
2. Arrays and Vectors

1. Recap

1. Recap

- Rust offers many basic types
  - Integer
  - Floating Point numbers
  - boolean
  - char

1. Recap

- Rust offers many basic types
- Integer types in Rust: i8, u8, i16, u16, ..., i128, u128
  - number → How many bits does the value have?
  - u is unsigned, i is signed → Negative numbers

1. Recap

- Rust offers many basic types
- Integer types in Rust: i8, u8, i16, u16, ..., i128, u128
- let declares an immutable variable
- let mut declares a mutable variable

```
fn main() {
    let a: i32 = 0;
    a = 5;
    let mut b: u8 = 12;
    b = -29;
}
```

1. Recap

- Rust offers many basic types
- Integer types in Rust: i8, u8, i16, u16, ..., i128, u128
- let declares an immutable variable
- let mut declares a mutable variable

```
fn main() {
    let a: i32 = 0;
    a = 5;
    let mut b: u8 = 12;
    b = -29;
}
```

Immutable → Can‘t re-assign to a

1. Recap

- Rust offers many basic types
- Integer types in Rust: i8, u8, i16, u16, ..., i128, u128
- let declares an immutable variable
- let mut declares a mutable variable

```
fn main() {
    let a: i32 = 0;
    a = 5;
    let mut b: u8 = 12;
    b = -29;
}
```

Mutable → Can re-assign to b

1. Recap

- Rust offers many basic types
- Integer types in Rust: i8, u8, i16, u16, ..., i128, u128
- let declares an immutable variable
- let mut declares a mutable variable

0/3

```
fn main() {
    let a: i32 = 0;
    a = 5;
    let mut b: u8 = 12;
    b = -29;
}
```

Does this work?

1. Recap

- Rust offers many basic types
- Integer types in Rust: i8, u8, i16, u16, ..., i128, u128
- let declares an immutable variable
- let mut declares a mutable variable

0/3

```
fn main() {
    let a: i32 = 0;
    a = 5;
    let mut b: u8 = 12;
    b = -29;
}
```

Does this work?
No → u8 is unsigned, can‘t be negative

1. Recap

2/3

```rust
fn main() {
    let a = 330;
    let b = a + 26;
    let c = b as u8;
    println!("c = {}", c);
}
```

1. Recap

What does the program print in the console?

2/3

```
fn main() {
    let a = 330;
    let b = a + 26;
    let c = b as u8;
    println!("c = {}", c);
}
```

1. Recap

What does the program print in the console?

2/3

```
fn main() {
    let a = 330;
    let b = a + 26;
    let c = b as u8;
    println!("c = {}", c);
}
```

a and b are both i32

1. Recap

What does the program print in the console?

2/3

```
fn main() {
    let a = 330;
    let b = a + 26;
    let c = b as u8;
    println!("c = {}", c);
}
```

a and b are both i32
→ b contains 356

1. Recap

What does the program print in the console?

2/3

```
fn main() {
    let a = 330;
    let b = a + 26;
    let c = b as u8;
    println!("c = {}", c);
}
```

a and b are both i32
→ b contains 356
→ 356 does not fit into an u8!

1. Recap

What does the program print in the console?

2/3

```
fn main() {
    let a = 330;
    let b = a + 26;
    let c = b as u8;
    println!("c = {}", c);
}
```

```
Runnin
c = 100
```

a and b are both i32
→ b contains 356
→ 356 does not fit into an u8!
→ Overflow, c contains 100

1. Recap

- Important to know:
  - Rust only performs most sanity checks (e.g. Arithmetic Overflow) in Debug mode, not Release mode

1. Recap

- Important to know:
  - Rust only performs most sanity checks (e.g. Arithmetic Overflow) in Debug mode, not Release mode
  - Default type for integers is i32

1. Recap

- Important to know:
  - Rust only performs most sanity checks (e.g. Arithmetic Overflow) in Debug mode, not Release mode
  - Default type for integers is i32
  - Integer division truncates the result (rounds toward 0)
    - Use floats (f32, f64) if decimal digits are important

1. Recap

- Important to know:
  - Rust only performs most sanity checks (e.g. Arithmetic Overflow) in Debug mode, not Release mode
  - Default type for integers is i32
  - Integer division truncates the result (rounds toward 0)
  - Type implementations come with additional constants to make life easier

```rust
fn main() {
    let min_i8: i8 = i8::MIN;
    let max_u128: u128 = u128::MAX;
    let u64_bits: u32 = u64::BITS;
}
```

1. Recap

- Important to know:
  - Rust only performs most sanity checks (e.g. Arithmetic Overflow) in Debug mode, not Release mode
  - Default type for integers is i32
  - Integer division truncates the result (rounds toward 0)
  - Type implementations come with additional constants to make life easier

Special constant variable defined in another module

```
fn main() {
    let min_i8: i8 = i8::MIN;
    let max_u128: u128 = u128::MAX;
    let u64_bits: u32 = u64::BITS;
}
```

1. Recap

- Important to know:
  - Rust only performs most sanity checks (e.g. Arithmetic Overflow) in Debug mode, not Release mode
  - Default type for integers is i32
  - Integer division truncates the result (rounds toward 0)
  - Type implementations come with additional constants to make life easier
  - Type limits are bounds, there's no way of storing more than that
    - If your bottle can only hold 2 liters of water, you can't fill it with 10 liters → It overflows

1. Recap

- Important to know:
  - Rust only performs most sanity checks (e.g. Arithmetic Overflow) in Debug mode, not Release mode
  - Default type for integers is i32
  - Integer division truncates the result (rounds toward 0)
  - Type implementations come with additional constants to make life easier
  - Type limits are bounds, there's no way of storing more than that
    - If your bottle can only hold 2 liters of water, you can't fill it with 10 liters → It overflows
    - Overflows in Programming work slightly different:
      - Instead of being stuck at 2 liters, it wraps around → Our bottle would contain 10 % 2 = 0 liters of water!
      - There's no physical space for the other bits

2. Arrays and Vectors

2. Arrays and Vectors

- Arrays and Vectors are collections of values

2. Arrays and Vectors

- Arrays and Vectors are collections of values

```
fn main() {
    let x1: i32 = 0;
    let x2: i32 = 17;
    let x3: i32 = 39;
    let x4: i32 = -129;
    let x5: i32 = 41;
}
```

2. Arrays and Vectors

- Arrays and Vectors are collections of values

```
fn main() {
    let x1: i32 = 0;
    let x2: i32 = 17;
    let x3: i32 = 39;
    let x4: i32 = -129;
    let x5: i32 = 41;
}
```

A lot of effort, a lot of manual management!

2. Arrays and Vectors

- Arrays and Vectors are collections of values

```
fn main() {
    let x1: i32 = 0;
    let x2: i32 = 17;
    let x3: i32 = 39;
    let x4: i32 = -129;
    let x5: i32 = 41;
}
```

A lot of effort, a lot of manual management!

```
println!("{}", x1);
println!("{}", x2);
println!("{}", x3);
println!("{}", x4);
println!("{}", x5);
```

2. Arrays and Vectors

- Arrays and Vectors are collections of values

```rust
fn main() {
    let x1: i32 = 0;
    let x2: i32 = 17;
    let x3: i32 = 39;
    let x4: i32 = -129;
    let x5: i32 = 41;
}
```

A lot of effort, a lot of manual management!
Now imagine having 1000 x's!!

```rust
println!("{}", x1);
println!("{}", x2);
println!("{}", x3);
println!("{}", x4);
println!("{}", x5);
```

2. Arrays and Vectors

- Arrays and Vectors are collections of values

```rust
fn main() {
    let x1: i32 = 0;
    let x2: i32 = 17;
    let x3: i32 = 39;
    let x4: i32 = -129;
    let x5: i32 = 41;
}
```

```rust
let xs: [i32; 5] = [0, 17, 39, -129, 41];
```

Introducing: The Array!

2. Arrays and Vectors

- Arrays and Vectors are collections of values

```
fn main() {
    let x1: i32 = 0;
    let x2: i32 = 17;
    let x3: i32 = 39;
    let x4: i32 = -129;
    let x5: i32 = 41;
}
```

5 elements of type i32

```
let xs: [i32; 5] = [0, 17, 39, -129, 41];
```

Introducing: The Array!

2. Arrays and Vectors

- Arrays and Vectors are collections of values

```rust
fn main() {
    let x1: i32 = 0;
    let x2: i32 = 17;
    let x3: i32 = 39;
    let x4: i32 = -129;
    let x5: i32 = 41;
}
```

Create array using [elem1, elem2, ...]

```rust
let xs: [i32; 5] = [0, 17, 39, -129, 41];
```

Introducing: The Array!

2. Arrays and Vectors

- Arrays and Vectors are collections of values

```rust
fn main() {
    let x1: i32 = 0;
    let x2: i32 = 17;
    let x3: i32 = 39;
    let x4: i32 = -129;
    let x5: i32 = 41;
}
```

```rust
let xy: [[i32; 3]; 3] = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];
```

2. Arrays and Vectors

- Arrays and Vectors are collections of values

```
fn main() {
    let x1: i32 = 0;
    let x2: i32 = 17;
    let x3: i32 = 39;
    let x4: i32 = -129;
    let x5: i32 = 41;
}
```

Arrays can be nested

```
let xy: [[i32; 3]; 3] = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];
```

2. Arrays and Vectors

- Arrays and Vectors are collections of values

```
fn main() {
    let x1: i32 = 0;
    let x2: i32 = 17;
    let x3: i32 = 39;
    let x4: i32 = -129;
    let x5: i32 = 41;
}
```

```
let xy: [[i32; 3]; 3] = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];
```

Read: An array of arrays of i32

2. Arrays and Vectors

- Arrays and Vectors are collections of values

```
fn main() {
    let x1: i32 = 0;
    let x2: i32 = 17;
    let x3: i32 = 39;
    let x4: i32 = -129;
    let x5: i32 = 41;
}
```

```
let xy: [[i32; 3]; 3] = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];
```

3 arrays...

2. Arrays and Vectors

- Arrays and Vectors are collections of values

```
fn main() {
    let x1: i32 = 0;
    let x2: i32 = 17;
    let x3: i32 = 39;
    let x4: i32 = -129;
    let x5: i32 = 41;
}
```

```
let xy: [[i32; 3]; 3] = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];
```

3 arrays...
of 3 values each

2. Arrays and Vectors

- Arrays and Vectors are collections of values

```rust
fn main() {
    let x1: i32 = 0;
    let x2: i32 = 17;
    let x3: i32 = 39;
    let x4: i32 = -129;
    let x5: i32 = 41;
}
```

```rust
let xy: [[i32; 3]; 3] = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];
```

3 arrays...
of 3 values each...
all of type i32

2. Arrays and Vectors

- Arrays and Vectors are collections of values

```rust
fn main() {
    let x1: i32 = 0;
    let x2: i32 = 17;
    let x3: i32 = 39;
    let x4: i32 = -129;
    let x5: i32 = 41;
}
```

```rust
let xy: [[i32; 3]; 3] = [
    [1, 2, 3],
    [4, 5, 6u8],
    [7, 8, 9]
];
```

3 arrays...
of 3 values each...
all of type i32...
Static typecheck catches that!

2. Arrays and Vectors

- Arrays and Vectors are collections of values
- Collections of values with the same type

2. Arrays and Vectors

- Arrays and Vectors are collections of values
- Collections of values with the same type

```
let arr: [u8; 2] = [5.0, 190];
```

2. Arrays and Vectors

- Arrays and Vectors are collections of values
- Collections of values with the same type

```
mismatched types
expected `u8`, found floating-point number
compiler diagnostic)
5 (bits: 0x4014000000000000)
View Problem (Alt+F8)  No quick fixes available
[5.0, 190];
```

```
let arr: [u8; 2] = [5.0, 190];
```

2. Arrays and Vectors

- Arrays and Vectors are collections of values
- Collections of values with the same type
- Arrays have a fixed size, Vectors are resizable

2. Arrays and Vectors

- Arrays and Vectors are collections of values
- Collections of values with the same type
- Arrays have a fixed size, Vectors are resizable
  - Array size is specified when declaring them

```
let xs: [i32; 5] = [0, 17, 39, -129, 41];
```

```
let ys: [i32; 2] = [x1, x2];
```

2. Arrays and Vectors

- Arrays and Vectors are collections of values
- Collections of values with the same type
- Arrays have a fixed size, Vectors are resizable
  - Array size is specified when declaring them

```
let xs: [i32; 5] = [0, 17, 39, -129, 41];
```

Specify directly

```
let ys: [i32; 2] = [x1, x2];
```

2. Arrays and Vectors

- Arrays and Vectors are collections of values
- Collections of values with the same type
- Arrays have a fixed size, Vectors are resizable
  - Array size is specified when declaring them

```
let xs: [i32; 5] = [0, 17, 39, -129, 41];
```

```
let ys: [i32; 2] = [x1, x2];
```

Compiler knows the size of this array, so it can infer the type

2. Arrays and Vectors

- Arrays and Vectors are collections of values
- Collections of values with the same type
- Arrays have a fixed size, Vectors are resizable
  - Array size is specified when declaring them

```
let xs: [i32; 5] = [0, 17, 39, -129, 41, 19];
```

The compiler will complain if the size doesn‘t match

```
mismatched types
expected an array with a fixed size of 5 elements, found one
with 6 elements rustc(Click for full compiler diagnostic)
```

2. Arrays and Vectors

- Arrays and Vectors are collections of values
- Collections of values with the same type
- Arrays have a fixed size, Vectors are resizable
  - Array size is specified when declaring them

```
let big: [i32; 1000] = [12; 1000];
```

2. Arrays and Vectors

- Arrays and Vectors are collections of values
- Collections of values with the same type
- Arrays have a fixed size, Vectors are resizable
  - Array size is specified when declaring them

```
let big: [i32; 1000] = [12; 1000];
```

Might take a while to specify 1000 elements...

2. Arrays and Vectors

- Arrays and Vectors are collections of values
- Collections of values with the same type
- Arrays have a fixed size, Vectors are resizable
  - Array size is specified when declaring them

```
let big: [i32; 1000] = [12; 1000];
```

Short form: Creates an array of size 1000, and sets each element to 12

2. Arrays and Vectors

- Arrays and Vectors are collections of values
- Collections of values with the same type
- Arrays have a fixed size, Vectors are resizable
  - Array size is specified when declaring them

```
let big: [i32; 1000] = [12, 1000];
```

Watch out!

2. Arrays and Vectors

- Arrays and Vectors are collections of values
- Collections of values with the same type
- Arrays have a fixed size, Vectors are resizable
  - Array size is specified when declaring them
  - Vectors have an internal capacity, and are resized automatically when adding too many elements

2. Arrays and Vectors

- Arrays and Vectors are collections of values
- Collections of values with the same type
- Arrays have a fixed size, Vectors are resizable
  - Array size is specified when declaring them
  - Vectors have an internal capacity, and are resized automatically when adding too many elements

```
let vec: Vec<i32> = Vec::new();
```

2. Arrays and Vectors

- Arrays and Vectors are collections of values
- Collections of values with the same type
- Arrays have a fixed size, Vectors are resizable
  - Array size is specified when declaring them
  - Vectors have an internal capacity, and are resized automatically when adding too many elements

```
let vec: Vec<i32> = Vec::new();
```

Vector of i32
(Note the <> instead of [])

2. Arrays and Vectors

- Arrays and Vectors are collections of values
- Collections of values with the same type
- Arrays have a fixed size, Vectors are resizable
  - Array size is specified when declaring them
  - Vectors have an internal capacity, and are resized automatically when adding too many elements

```
let vec: Vec<i32> = Vec::new();
```

Structs and functions will be covered later,
but this creates a new Vector for us

2. Arrays and Vectors

- Arrays and Vectors are collections of values
- Collections of values with the same type
- Arrays have a fixed size, Vectors are resizable
  - Array size is specified when declaring them
  - Vectors have an internal capacity, and are resized automatically when adding too many elements

```
let vec: Vec<i32> = Vec::new();
```

This Vector has a capacity of 0
→ resized the first time you push an element

2. Arrays and Vectors

- Arrays and Vectors are collections of values
- Collections of values with the same type
- Arrays have a fixed size, Vectors are resizable
  - Array size is specified when declaring them
  - Vectors have an internal capacity, and are resized automatically when adding too many elements

```
let vec: Vec<i32> = Vec::new();
```

This Vector has a capacity of 0
→ resized the first time you push an element

3/3 Why? Seems inefficient...

2. Arrays and Vectors

- Arrays have a fixed size, Vectors are resizable
- Array elements are located on the Stack, Vector elements are on the Heap
  - Vec::new() has a capacity of 0, which means it can't store any elements
    → No heap allocation necessary, more efficient
    → Only when you add an element, you need Heap memory

2. Arrays and Vectors

- Arrays have a fixed size, Vectors are resizable
- Array elements are located on the Stack, Vector elements are on the Heap
  - Vec::new() has a capacity of 0, which means it can't store any elements
    → No heap allocation necessary, more efficient
    → Only when you add an element, you need Heap memory
  - For something to be on the Stack, its size has to be known at compile time
    → Arrays are fixed size, can't add or remove elements

2. Arrays and Vectors

- Arrays have a fixed size, Vectors are resizable
- Array elements are located on the Stack, Vector elements are on the Heap
- Generally: Stack access is quicker than Heap access

2. Arrays and Vectors

- Arrays have a fixed size, Vectors are resizable
- Array elements are located on the Stack, Vector elements are on the Heap
- Generally: Stack access is quicker than Heap access
  - Heap needs to be allocated
  - Heap may have Pointer-Overhead
  - Heap needs to be de-allocated

2. Arrays and Vectors

- Arrays have a fixed size, Vectors are resizable
- Array elements are located on the Stack, Vector elements are on the Heap
- Generally: Stack access is quicker than Heap access
  - Heap needs to be allocated and de-allocated
  - For the Stack, those [de-]allocations are builtin and have 0 overhead
    → More on that when we talk about functions

2. Arrays and Vectors

- Arrays have a fixed size, Vectors are resizable
- Array elements are located on the Stack, Vector elements are on the Heap
- Generally: Stack access is quicker than Heap access
  - Heap needs to be allocated and de-allocated
  - For the Stack, those [de-]allocations are builtin and have 0 overhead
- If you know the size of your collection, and never add or remove any elements, it's always better to use an Array instead of a Vector

2. Arrays and Vectors

```rust
fn stack_vs_heap() {
    let array: [i32; 3] = [10, 45, 90];
    let mut vector: Vec<i32> = Vec::new();
    vector.extend(iter: &array);
}
```

2. Arrays and Vectors

```rust
fn stack_vs_heap() {
    let array: [i32; 3] = [10, 45, 90];
    let mut vector: Vec<i32> = Vec::new();
    vector.extend(iter: &array);
}
```

Stack		
array	values	???
		???
		???
vector	content	???
	capacity	???
	length	???

2. Arrays and Vectors

```rust
fn stack_vs_heap() {
    let array: [i32; 3] = [10, 45, 90];
    let mut vector: Vec<i32> = Vec::new();
    vector.extend(iter: &array);
}
```

Stack		
array	values	10
		45
		90
vector	content	???
	capacity	???
	length	???

2. Arrays and Vectors

```rust
fn stack_vs_heap() {
    let array: [i32; 3] = [10, 45, 90];
    let mut vector: Vec<i32> = Vec::new();
    vector.extend(iter: &array);
}
```

Stack		
array	values	10
		45
		90
vector	content	???
	capacity	0
	length	0

2. Arrays and Vectors

```rust
fn stack_vs_heap() {
    let array: [i32; 3] = [10, 45, 90];
    let mut vector: Vec<i32> = Vec::new();
    vector.extend(iter: &array);
}
```

Stack		
array	values	10
		45
		90
vector	content	0xabc0
	capacity	4
	length	3

Heap	
0xabc0	10
0xabc4	45
0xabc8	90
...	...

Intermission - The vec![] macro

- Using Vec::push(), we can add elements to a Vector

Intermission - The vec![] macro

- Using Vec::push(), we can add elements to a Vector

```rust
fn vec_showcase() {
    let mut vec: Vec<i32> = Vec::new();
    vec.push(1);
    vec.push(2);
}
```

Intermission - The vec![] macro

- Using Vec::push(), we can add elements to a Vector

```rust
fn vec_showcase() {
    let mut vec: Vec<i32> = Vec::new();
    vec.push(1);
    vec.push(2);
}
```

Methods will be covered later, but this puts a 1 and a 2 into the vector

Intermission - The vec![] macro

- Using Vec::push(), we can add elements to a Vector

```rust
fn vec_showcase() {
    let mut vec: Vec<i32> = Vec::new();
    vec.push(1);
    vec.push(2);
}
```

Because we modify the vector, the variable has to be mutable

Intermission - The vec![] macro

- Imagine you want to declare a Vector with 5 elements

```rust
fn vec_showcase() {
    let mut vec: Vec<i32> = Vec::new();
    vec.push(1);
    vec.push(2);
    vec.push(3);
    vec.push(4);
    vec.push(5);
}
```

Intermission - The vec![] macro

- Imagine you want to declare a Vector with 5 elements

```rust
fn vec_showcase() {
    let mut vec: Vec<i32> = Vec::new();
    vec.push(1);
    vec.push(2);
    vec.push(3);
    vec.push(4);
    vec.push(5);
}
```

6 lines of code! That‘s a lot of work!
Now imagine declaring it with 10 elements!

Intermission - The vec![] macro

- Imagine you want to declare a Vector with 5 elements
- Introducing: The vec![] macro

Intermission - The vec![] macro

- Imagine you want to declare a Vector with 5 elements
- Introducing: The vec![] macro
- Thanks to Rust's Trait System, Arrays and Vectors share a lot of similarities
  - Getting and setting elements...
  - Checking if an Array/Vector contains an element...

Intermission - The vec![] macro

- Imagine you want to declare a Vector with 5 elements
- Introducing: The vec![] macro
- Thanks to Rust's Trait System, Arrays and Vectors share a lot of similarities
- By using this macro, we add declaration to that list

Intermission - The vec![] macro

- Imagine you want to declare a Vector with 5 elements
- Introducing: The vec![] macro
- Thanks to Rust's Trait System, Arrays and Vectors share a lot of similarities
- By using this macro, we add declaration to that list

```
fn vec_showcase() {
    let vec: Vec<i32> = vec![1, 2, 3, 4, 5];
}
```

Intermission - The vec![] macro

- Imagine you want to declare a Vector with 5 elements
- Introducing: The vec![] macro
- Thanks to Rust's Trait System, Arrays and Vectors share a lot of similarities
- By using this macro, we add declaration to that list

```
fn vec_showcase() {
    let vec: Vec<i32> = vec![1, 2, 3, 4, 5];
}
```

Intermission - The vec![] macro

- Imagine you want to declare a Vector with 5 elements
- Introducing: The vec![] macro
- Thanks to Rust's Trait System, Arrays and Vectors share a lot of similarities
- By using this macro, we add declaration to that list

```
fn vec_showcase() {
    let vec: Vec<i32> = vec![1, 2, 3, 4, 5];
}
```

Fill with those values!

Create a Vector

Intermission - The vec![] macro

- Imagine you want to declare a Vector with 5 elements
- Introducing: The vec![] macro
- Thanks to Rust's Trait System, Arrays and Vectors share a lot of similarities
- By using this macro, we add declaration to that list

```
fn vec_showcase() {
    let vec: Vec<i32> = vec![5; 1000];
}
```

Intermission - The vec![] macro

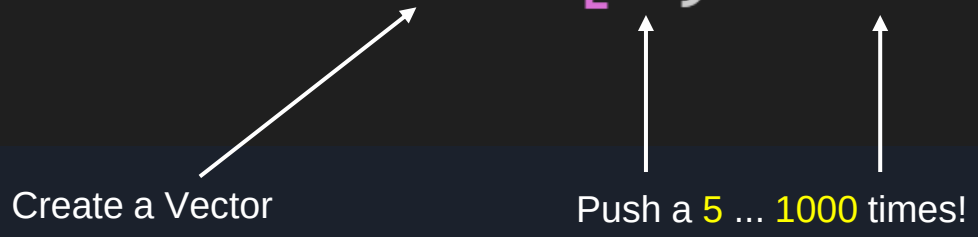
- Imagine you want to declare a Vector with 5 elements
- Introducing: The vec![] macro
- Thanks to Rust's Trait System, Arrays and Vectors share a lot of similarities
- By using this macro, we add declaration to that list

```rust
fn vec_showcase() {
    let vec: Vec<i32> = vec![5; 1000];
}
```

Intermission - The vec![] macro

- Imagine you want to declare a Vector with 5 elements
- Introducing: The vec![] macro
- Thanks to Rust's Trait System, Arrays and Vectors share a lot of similarities
- By using this macro, we add declaration to that list

```
fn vec_showcase() {
    let vec: Vec<i32> = vec![5; 1000];
}
```



Intermission - The vec![] macro

- Imagine you want to declare a Vector with 5 elements
- Introducing: The vec![] macro
- Thanks to Rust's Trait System, Arrays and Vectors share a lot of similarities
- By using this macro, we add declaration to that list
- vec![] has one benefit over Array declarations → The size argument can be dynamic!

```
let size: usize = 10;
let vec: Vec<i32> = vec![5; size];
let array: [i32; size] = [5; size];
```

Intermission - The vec![] macro

- Imagine you want to declare a Vector with 5 elements
- Introducing: The vec![] macro
- Thanks to Rust's Trait System, Arrays and Vectors share a lot of similarities
- By using this macro, we add declaration to that list
- vec![] has one benefit over Array declarations → The size argument can be dynamic!

```
let size: usize = 10;
let vec: Vec<i32> = vec![5; size];
let array: [i32; size] = [5; size];
```

size is evaluated at runtime

Intermission - The vec![] macro

- Imagine you want to declare a Vector with 5 elements
- Introducing: The vec![] macro
- Thanks to Rust's Trait System, Arrays and Vectors share a lot of similarities
- By using this macro, we add declaration to that list
- vec![] has one benefit over Array declarations → The size argument can be dynamic!

```
let size: usize = 10;
let vec: Vec<i32> = vec![5; size];
let array: [i32; size] = [5; size];
```

size is evaluated at runtime

needs to be known at compile time for Arrays

Intermission - The vec![] macro

- Imagine you want to declare a Vector with 5 elements
- Introducing: The vec![] macro
- Thanks to Rust's Trait System, Arrays and Vectors share a lot of similarities
- By using this macro, we add declaration to that list
- vec![] has one benefit over Array declarations → The size argument can be dynamic!

```
let size: usize = 10;
...
let vec: Vec<i32> = vec![5; size];
let array: [i32; size] = [5; size];
```

size is evaluated at runtime

Doesn't matter for vec![]

needs to be known at compile time for Arrays

2. Arrays and Vectors

- Thanks to Traits, the Syntax for getting and setting elements is the same for Arrays and Vectors

2. Arrays and Vectors

- Thanks to Traits, the Syntax for getting and setting elements is the same for Arrays and Vectors
- Indices are always of type usize

2. Arrays and Vectors

- Thanks to Traits, the Syntax for getting and setting elements is the same for Arrays and Vectors
- Indices are always of type usize
- Arrays and Vectors are zero-indexed

2. Arrays and Vectors

- Arrays and Vectors are zero-indexed

```
let array: [i32; 3] = [10, 45, 90];
```

2. Arrays and Vectors

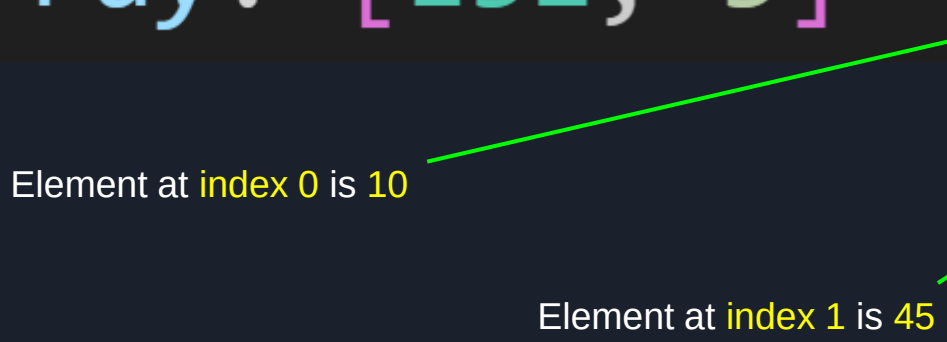
- Arrays and Vectors are zero-indexed

```
let array: [i32; 3] = [10, 45, 90];
```

2. Arrays and Vectors

- Arrays and Vectors are zero-indexed

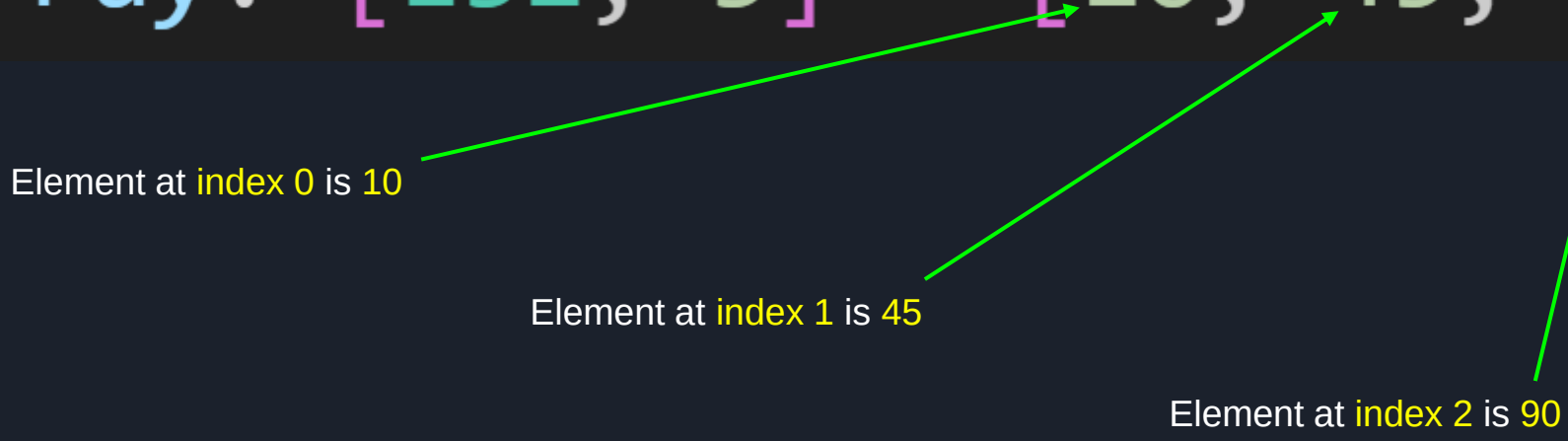
```
let array: [i32; 3] = [10, 45, 90];
```



2. Arrays and Vectors

- Arrays and Vectors are zero-indexed

```
let array: [i32; 3] = [10, 45, 90];
```



2. Arrays and Vectors

```rust
let arr: [i32; 5] = [10, 20, 30, 40, 50];
let element: i32 = arr[1];
println!("arr element = {}", element);

let vec: Vec<i32> = vec![10, 20, 30, 40, 50];
let element: i32 = vec[1];
println!("vec element = {}", element);
```

2. Arrays and Vectors

```rust
let arr: [i32; 5] = [10, 20, 30, 40, 50];
let element: i32 = arr[1];
println!("arr element = {}", element);

let vec: Vec<i32> = vec![10, 20, 30, 40, 50];
let element: i32 = vec[1];
println!("vec element = {}", element);
```

Get element at index 1

Get element at index 1

2. Arrays and Vectors

1/3

```rust
let arr: [i32; 5] = [10, 20, 30, 40, 50];
let element: i32 = arr[1];
println!("arr element = {}", element);
```

What does this print?

```rust
let vec: Vec<i32> = vec![10, 20, 30, 40, 50];
let element: i32 = vec[1];
println!("vec element = {}", element);
```

What does this print?

2. Arrays and Vectors

1/3

```
let arr: [i32; 5] = [10, 20, 30, 40, 50];
let element: i32 = arr[1];
println!("arr element = {}", element);
```

What does this print?

```
Running `cal
arr element = 20
vec element = 20
```

```
let vec: Vec<i32> = vec
let element: i32 = vec[
println!("vec element = {}", element);
```

What does this print?

2. Arrays and Vectors

```rust
fn set() {
    let arr: [i32; 5] = [10, 20, 30, 40, 50];
    arr[3] = 60;

    let vec: Vec<i32> = vec![10, 20, 30, 40, 50];
    vec[3] = 60;
}
```

2. Arrays and Vectors

```rust
fn set() {
    let arr: [i32; 5] = [10, 20, 30, 40, 50];
    arr[3] = 60; Set the element at index 3 to 60

    let vec: Vec<i32> = vec![10, 20, 30, 40, 50];
    vec[3] = 60; Set the element at index 3 to 60
}
```

2. Arrays and Vectors

```rust
fn set() {
    let arr: [i32; 5] = [10, 20, 30, 40, 50];
    arr[3] = 60; Doesn‘t work!

    let vec: Vec<i32> = vec![10, 20, 30, 40, 50];
    vec[3] = 60; Doesn‘t work!
}
```

2. Arrays and Vectors

```rust
fn set() {
    let arr: [i32; 5] = [10, 20, 30, 40, 50];
    arr[3] = 60;

    let vec: Vec<i32> = vec![10, 20, 30, 40, 50];
    vec[3] = 60;
}
```

Variables are immutable, can‘t modify them!

2. Arrays and Vectors

```rust
fn set() {
    let mut arr: [i32; 5] = [10, 20, 30, 40, 50];
    arr[3] = 60;

    let mut vec: Vec<i32> = vec![10, 20, 30, 40, 50];
    vec[3] = 60;
}
```

2. Arrays and Vectors

```
fn set() {
    let mut arr: [i32; 5] = [10, 20, 30, 60, 50];
    arr[3] = 60;

    let mut vec: Vec<i32> = vec![10, 20, 30, 60, 50];
    vec[3] = 60;
}
```

2. Arrays and Vectors

2/3

```
let mut arr: [i32; 5] = [10, 20, 30, 40, 50];
arr[3] = 100;
arr[4] = arr[1] + arr[2];
arr[3] = arr[0] + arr[3];
println!("{:?}", arr);
```

2. Arrays and Vectors

2/3

```
let mut arr: [i32; 5] = [10, 20, 30, 40, 50];
arr[3] = 100;
arr[4] = arr[1] + arr[2];
arr[3] = arr[0] + arr[3];
println!("{:?}", arr);
```

What does the array look like here?

2. Arrays and Vectors

2/3

```
let mut arr: [i32; 5] = [10, 20, 30, 100, 50];
arr[3] = 100;
arr[4] = arr[1] + arr[2];
arr[3] = arr[0] + arr[3];
println!("{:?}", arr);
```

What does the array look like here?

2. Arrays and Vectors

2/3

```
let mut arr: [i32; 5] = [10, 20, 30, 100, 50];
arr[3] = 100;
arr[4] = 20 + arr[2];
arr[3] = arr[0] + arr[3];
println!("{:?}", arr);
```

What does the array look like here?

2. Arrays and Vectors

2/3

```
let mut arr: [i32; 5] = [10, 20, 30, 100, 50];
arr[3] = 100;
arr[4] = 20 + 30;
arr[3] = arr[0] + arr[3];
println!("{:?}", arr);
```

What does the array look like here?

2. Arrays and Vectors

2/3

```
let mut arr: [i32; 5] = [10, 20, 30, 100, 50];
arr[3] = 100;
arr[4] = 20 + 30;
arr[3] = arr[0] + arr[3];
println!("{:?}", arr);
```

What does the array look like here?

2. Arrays and Vectors

2/3

```
let mut arr: [i32; 5] = [10, 20, 30, 110, 50];
arr[3] = 100;
arr[4] = arr[1] + arr[2];
arr[3] = 10 + 100;
println!("{:?}", arr);
```

What does the array look like here?

2. Arrays and Vectors

2/3

```
let mut arr: [i32; 5] = [10, 20, 30, 110, 50];
arr[3] = 100;
arr[4] = arr[1] + arr[2];
arr[3] = arr[0] + arr[3];
println!("{:?}", arr);
```

Result:

```
[10, 20, 30, 110, 50]
```

2. Arrays and Vectors

- Arrays and Vectors come with many useful methods

2. Arrays and Vectors

- Arrays and Vectors come with many useful methods

```rust
let mut arr: [i32; 1] = [1];
arr.is_empty();
arr.fill(5);
arr.contains(&5);
arr.len();
arr.sort();
// ... and many more
```

2. Arrays and Vectors

- Arrays and Vectors come with many useful methods

```
let mut arr: [i32; 1] = [1];
arr.is_empty();
arr.fill(5);
arr.contains(&5);
arr.len();
arr.sort();
// ... and many more
```

Are there any elements in our array?

2. Arrays and Vectors

- Arrays and Vectors come with many useful methods

```
let mut arr: [i32; 1] = [1];
arr.is_empty();
arr.fill(5);
arr.contains(&5);
arr.len();
arr.sort();
// ... and many more
```

Are there any elements in our array?

Set every element in the array to the provided value

2. Arrays and Vectors

- Arrays and Vectors come with many useful methods

```
let mut arr: [i32; 1] = [1];
arr.is_empty();
arr.fill(5);
arr.contains(&5);
arr.len();
arr.sort();
// ... and many more
```

Are there any elements in our array?

Set every element in the array to the provided value

Does the array contain the given value?

2. Arrays and Vectors

- Arrays and Vectors come with many useful methods

```rust
let mut arr: [i32; 1] = [1];
arr.is_empty();
arr.fill(5);
arr.contains(&5);
arr.len();
arr.sort();
// ... and many more
```

Are there any elements in our array?

Set every element in the array to the provided value

Does the array contain the given value?

Get the size of our array (here: 1)

2. Arrays and Vectors

- Arrays and Vectors come with many useful methods

```
let mut arr: [i32; 1] = [1];
arr.is_empty();
arr.fill(5);
arr.contains(&5);
arr.len();
arr.sort();
// ... and many more
```

Are there any elements in our array?

Set every element in the array to the provided value

Does the array contain the given value?

Get the size of our array (here: 1)

Sort the array (in place)

2. Arrays and Vectors

- Arrays and Vectors come with many useful methods

```
let mut vec: Vec<i32> = vec![1];
// Vectors can do everything arrays can do
vec.extend(iter: &arr);
vec.push(5);
vec.remove(index: 0);
vec.insert(index: 0, element: 5);
// ... and many more
```

2. Arrays and Vectors

- Arrays and Vectors come with many useful methods

```rust
let mut vec: Vec<i32> = vec![1];
// Vectors can do everything arrays can do
vec.extend(iter: &arr);
vec.push(5);
vec.remove(index: 0);
vec.insert(index: 0, element: 5);
// ... and many more
```

Push all elements of a collection into the vector

2. Arrays and Vectors

- Arrays and Vectors come with many useful methods

```rust
let mut vec: Vec<i32> = vec![1];
// Vectors can do everything arrays can do
vec.extend(iter: &arr);
vec.push(5);
vec.remove(index: 0);
vec.insert(index: 0, element: 5);
// ... and many more
```

Push all elements of a collection into the vector

Push a single element into the vector
→ inserts it at the end

2. Arrays and Vectors

- Arrays and Vectors come with many useful methods

```rust
let mut vec: Vec<i32> = vec![1];
// Vectors can do everything arrays can do
vec.extend(iter: &arr);
vec.push(5);
vec.remove(index: 0);
vec.insert(index: 0, element: 5);
// ... and many more
```

Push all elements of a collection into the vector

Push a single element into the vector

Remove the element at a given index
→ Shifts all values after it to the left

2. Arrays and Vectors

- Arrays and Vectors come with many useful methods

```rust
let mut vec: Vec<i32> = vec![1];
// Vectors can do everything arrays can do
vec.extend(iter: &arr);
vec.push(5);
vec.remove(index: 0);
vec.insert(index: 0, element: 5);
// ... and many more
```

Push all elements of a collection into the vector

Push a single element into the vector

Remove the element at a given index

Insert an element at a given index
→ Shifts all values after it to the right

Intermission - Exercise

- Time for exercises!

Intermission - Exercise

2/3

```rust
fn main() {
    let size: usize = 5;
    let mut vec: Vec<usize> = vec![42, 31, 1, 19, 3];
    let mut value: usize = vec[size - 1];
    vec[value] = 2;
    value = vec[vec[value]];
    vec[0] = vec[value];
    println!("value: {}", value);
    println!("vec: {:?}", vec);
}
```

Intermission - Exercise

2/3

```rust
fn main() {
    let size: usize = 5;
    let mut vec: Vec<usize> = vec![42, 31, 1, 19, 3];
    let mut value: usize = vec[size - 1];
    vec[value] = 2;
    value = vec[vec[value]];
    vec[0] = vec[value];
    println!("value: {}", value);
    println!("vec: {:?}", vec);
}
```

This code compiles.
What does it print in the end?

Intermission - Exercise

2/3

```rust
fn main() {
    let size: usize = 5;
    let mut vec: Vec<usize> = vec![42, 31, 1, 19, 3];
    let mut value: usize = vec[size - 1];
    vec[value] = 2;
    value = vec[vec[value]];
    vec[0] = vec[value];
    println!("value: {}", value);
    println!("vec: {:?}", vec);
}
```

This code compiles.
What does it print in the end?

→ value = vec[4] = 3

Intermission - Exercise

2/3

```rust
fn main() {
    let size: usize = 5;
    let mut vec: Vec<usize> = vec![42, 31, 1, 19, 3];
    let mut value: usize = vec[size - 1];
    vec[value] = 2;
    value = vec[vec[value]];
    vec[0] = vec[value];
    println!("value: {}", value);
    println!("vec: {:?}", vec);
}
```

This code compiles.
What does it print in the end?

→ value = 3 → vec[3] = 2

Intermission - Exercise

2/3

```rust
fn main() {
    let size: usize = 5;
    let mut vec: Vec<usize> = vec![42, 31, 1, 2, 3];
    let mut value: usize = vec[size - 1];
    vec[value] = 2;
    value = vec[vec[value]];
    vec[0] = vec[value];
    println!("value: {}", value);
    println!("vec: {:?}", vec);
}
```

This code compiles.
What does it print in the end?

→ value = 3 → vec[3] = 2

Intermission - Exercise

2/3

```rust
fn main() {
    let size: usize = 5;
    let mut vec: Vec<usize> = vec![42, 31, 1, 2, 3];
    let mut value: usize = vec[size - 1];
    vec[value] = 2;
    value = vec[vec[value]];
    vec[0] = vec[value];
    println!("value: {}", value);
    println!("vec: {:?}", vec);
}
```

This code compiles.
What does it print in the end?

→ value = vec[vec[value]]

Intermission - Exercise

2/3

```rust
fn main() {
    let size: usize = 5;
    let mut vec: Vec<usize> = vec![42, 31, 1, 2, 3];
    let mut value: usize = vec[size - 1];
    vec[value] = 2;
    value = vec[vec[value]];
    vec[0] = vec[value];
    println!("value: {}", value);
    println!("vec: {:?}", vec);
}
```

This code compiles.
What does it print in the end?

→ value = vec[vec[value]]
→ vec[value] = 2

Intermission - Exercise

2/3

```rust
fn main() {
    let size: usize = 5;
    let mut vec: Vec<usize> = vec![42, 31, 1, 2, 3];
    let mut value: usize = vec[size - 1];
    vec[value] = 2;
    value = vec[vec[value]];
    vec[0] = vec[value];
    println!("value: {}", value);
    println!("vec: {:?}", vec);
}
```

This code compiles.
What does it print in the end?

→ value = vec[vec[value]]
→ vec[value] = 2
→ value = vec[2] = 1

Intermission - Exercise

2/3

This code compiles.
What does it print in the end?

```rust
fn main() {
    let size: usize = 5;
    let mut vec: Vec<usize> = vec![42, 31, 1, 2, 3];
    let mut value: usize = vec[size - 1];
    vec[value] = 2;
    value = vec[vec[value]];
    vec[0] = vec[value]; → value = 1
    println!("value: {}", value);
    println!("vec: {:?}", vec);
}
```

Intermission - Exercise

2/3

```rust
fn main() {
    let size: usize = 5;
    let mut vec: Vec<usize> = vec![42, 31, 1, 2, 3];
    let mut value: usize = vec[size - 1];
    vec[value] = 2;
    value = vec[vec[value]];
    vec[0] = vec[value]; → value = 1 → vec[0] = vec[1]
    println!("value: {}", value);
    println!("vec: {:?}", vec);
}
```

This code compiles.
What does it print in the end?

Intermission - Exercise

2/3

```rust
fn main() {
    let size: usize = 5;
    let mut vec: Vec<usize> = vec![42, 31, 1, 2, 3];
    let mut value: usize = vec[size - 1];
    vec[value] = 2;
    value = vec[vec[value]];
    vec[0] = vec[value]; → value = 1 → vec[0] = vec[1] = 31
    println!("value: {}", value);
    println!("vec: {:?}", vec);
}
```

This code compiles.
What does it print in the end?

Intermission - Exercise

2/3

```rust
fn main() {
    let size: usize = 5;
    let mut vec: Vec<usize> = vec![31, 31, 1, 2, 3];
    let mut value: usize = vec[size - 1];
    vec[value] = 2;
    value = vec[vec[value]];
    vec[0] = vec[value];
    println!("value: {}", value);
    println!("vec: {:?}", vec);
}
```

This code compiles.
What does it print in the end?

→ vec[0] = 31

Intermission - Exercise

2/3

```rust
fn main() {
    let size: usize = 5;
    let mut vec: Vec<usize> = vec![31, 31, 1, 2, 3];
    let mut value: usize = vec[size - 1];
    vec[value] = 2;
    value = vec[vec[value]];
    vec[0] = vec[value];
    println!("value: {}", value);
    println!("vec: {:?}", vec);
}
```

This code compiles.
What does it print in the end?

→ value = 1

```
value: 1
vec: [31, 31, 1, 2, 3]
```

Intermission - Exercise

1/3

```
fn main() {
    let mut arr = [12, 29, 37];
    let size = arr.len();
    arr[0] = size;
    println!("{:?}", arr);
}
```

Intermission - Exercise

1/3

```rust
fn main() {
    let mut arr = [12, 29, 37];
    let size = arr.len();
    arr[0] = size;
    println!("{:?}", arr);
}
```

Does this compile?
If yes, what does it print?
What type does arr have?

Intermission - Exercise

1/3

```rust
fn main() {
    let mut arr = [12, 29, 37];
    let size = arr.len();
    arr[0] = size;
    println!("{:?}", arr);
}
```

Does this compile?
If yes, what does it print?
What type does arr have?

It does compile!

Intermission - Exercise


```rust
fn main() {
    let mut arr = [12, 29, 37];
    let size: usize = arr.len();
    arr[0] = size;
    println!("{:?}", arr);
}
```

Does this compile?
If yes, what does it print?
What type does arr have?

len() returns a usize
→ size has type usize

Intermission - Exercise

1/3

```rust
fn main() {
    let mut arr = [12, 29, 37];
    let size: usize = arr.len();
    arr[0] = size;
    println!("{:?}", arr);
}
```

Does this compile?
If yes, what does it print?
What type does arr have?

at least the first element of arr is of type usize
→ all elements are usize

Intermission - Exercise

1/3

```rust
fn main() {
    let mut arr: [usize; 3] = [12, 29, 37];
    let size: usize = arr.len();
    arr[0] = size;
    println!("{:?}", arr);
}
```

Does this compile?
If yes, what does it print?
What type does arr have?

at least the first element of arr is of type usize
→ all elements are usize

Intermission - Exercise

1/3

```rust
fn main() {
    let mut arr: [usize; 3] = [12, 29, 37];
    let size: usize = arr.len();
    arr[0] = size;
    println!("{:?}", arr);
}
```

```
[3, 29, 37]
```

Does this compile?
If yes, what does it print?
What type does arr have?

at least the first element of arr is of type usize
→ all elements are usize

3. Next time

- Control Flow
- Doing the same thing over and over again
  - loop
  - while
  - for