

**14.**

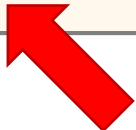
---

Deref & Diverses

---


# Struct Initialisation Syntax

```
struct Point3 { ... }  
  
// usually on multiple lines  
let a = Point3 { x: 3, y: 8, z: 1 };  
  
// we want to use many values from `a`  
let b = Point3 { x: 0, .. a };
```



## .. other\_instance

- Verwendet die restlichen Werte aus **other\_instance**
- Besonders sinnvoll für **Default**

```
struct ServerConfig {  
    port: u16,  
    // ... many more fields!  
}  
  
impl Default for ServerConfig { ... }  
  
// We only want to tweak a couple  
// of values ...  
let my_config = ServerConfig {  
    port: 1337,  
    .. Default::default()   
};
```

# Rvalues & Lvalues

- Eigenschaften einer Expression:
  - Evaluiert zu einem Typen **T**
  - In Kategorie *Rvalue* oder *Lvalue*
- „Lvalues können links von ‚=‘ stehen“
  - Repräsentieren ein Stück Speicher
- Lvalue-Kontext:
  - Linker Teil von (Compound-)Assignment
  - Operand von Borrowing (**&** oder **&mut**)
  - In Verbindung mit **ref** in Pattern (...)
- Alles andere ist Rvalue-Kontext

```
fn foo() -> i32 { 7 }  
let mut a: i32 = 3;  
let r = &mut a;
```

---

```
a      // type of expr: i32  
*r     // type of expr: i32  
3      // type of expr: i32  
foo()  // type of expr: i32
```






---

Lvalue

Rvalue

```
<lvalue-ctx> = <rvalue-ctx>;  
<lvalue-ctx> += <rvalue-ctx>;  
&<lvalue-ctx>;  
&mut <lvalue-ctx>;  
let ref x = <lvalue-ctx>;
```

# Rvalues & Lvalues

	 <b>Rvalue</b> Kontext	<b>Lvalue</b> Kontext
<b>Rvalue</b> Expression	 als Wert	 Promotion
<b>Lvalue</b> Expression	 als Wert	 als Speicherstelle

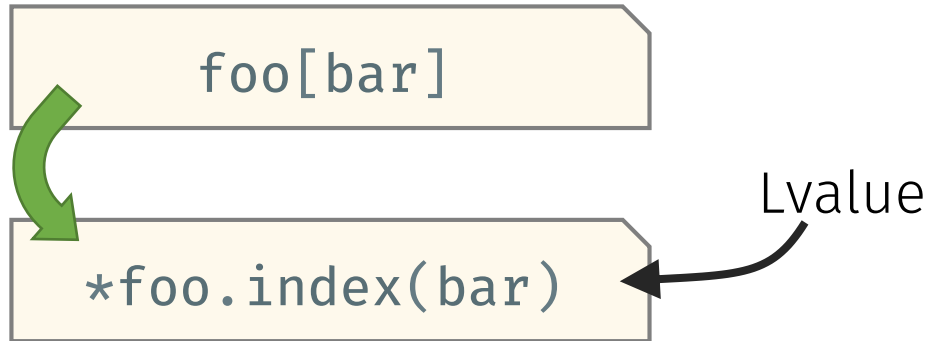
- **Lvalue**: Repräsentiert Speicherstelle
- **Rvalue**: Repräsentiert Wert

## Rvalue Promotion

- Legt temporäre Variable mit dem Wert an
- Nutzt temporäre Variable als Lvalue
- Nicht für Zuweisungen!

```
// Borrowing expects lvalue, but  
// '27' is an rvalue! Rvalue  
// promotion will make it work!  
let r = &mut 27;  
  
// Rvalue promotion will make the  
// above code equivalent to  
let mut _tmp = 27;  
let r = &mut _tmp;
```

# Index Operator



Desugaring enthält Dereferenzierung!

```
let mut v = vec![1, 2, 3];
```

```
// This uses the `IndexMut` trait which  
// works exactly like `Index`, but returns a  
// mutable reference. The compiler chooses  
// between Index/IndexMut automatically!
```

```
v[0] = 8;
```

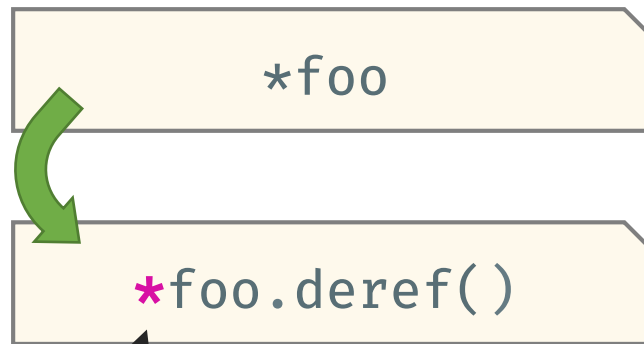
```
trait Index<Idx> {  
    type Output: ?Sized;  
    fn index(&self, index: Idx)  
        -> &Self::Output;  
}
```

```
impl Index<usize> for Vec<T> {  
    type Output = T;  
    fn index(&self, index: usize)  
        -> &T  
    { ... }  
}
```

```
let v = vec![1i32, 2, 3];  
let x: i32 = v[0];    // no ref?!  
let x: &i32 = v.index(0);
```

# Deref

- Überlädt unäres „\*“
- **DerefMut** auch vorhanden
- Gedacht, um auf „inneren“ bzw. „echten“ Wert zuzugreifen



„Rekursionsanker“:  
dereferenziert nur  
echte Referenzen!

```
pub trait Deref {  
    type Target: ?Sized;  
    fn deref(&self) -> &Self::Target;  
}
```

```
pub trait DerefMut {  
    type Target: ?Sized;  
    fn deref_mut(&mut self)  
        -> &mut Self::Target;  
}
```

```
// `String` impls Deref to `str`..  
let a_string = "hi".to_string();  
let s: &str = &*a_string;
```

# Deref Impls

## Container

String	→	str
Vec<T>	→	[T]
Cstring	→	CStr
OsString	→	OsStr
PathBuf	→	Path

## SmartPointer

Box<T>	→	T
Rc<T>	→	T
Arc<T>	→	T
Unique<T>	→	*mut T
Shared<T>	→	*mut T

## Andere Wrapper

NonZero<T>	→	T
Cow<'a, T>	→	T
MutexGuard<T>	→	T
... u.v.m		

- Viele Typen uns noch unbekannt
- Das meiste kommt noch!
- Sinnvoll für **Swagger<T>**

```
impl<T> Deref for Swagger<T> {  
    type Target = T;  
    fn deref(&self) -> &T { &self.0 }  
}
```

# Deref coercions

[Playground](#)

Wenn ein Typ **T** das Trait **Deref<Target=U>** implementiert, kann **&T** automatisch zu **&U** umgewandelt werden

```
fn takes_string_slice(s: &str) { ... }  
  
let s = "bob".to_string(); // : String  
takes_string_slice(&s);    // works!
```

- Automatische Typumwandlung
- Wird angewendet, bis Typ passt
- Auch für Methodenaufrufe!

```
let swag =  
    Swagger::new("bob".to_string());  
  
takes_string_slice(&swag); // works!
```

```
// We can even call methods defined  
// on inner types!  
swag.capacity(); // String::capacity  
swag.chars();    // str::chars
```