Konstanten

- `const` & **nie** mit `mut` verwenden
- explizit mit **Datentyp** & **Wert** initialisieren

```rust
const TEXT: &'static str = "ABC";
```

- Static Variablen können **nicht** mut sein **&** Referenzen nutzen
- `static` Unterschied zu `const`: potenziel. mut

```rust
static HELLO: &str = "Hello, world!";
fn main() {println!("{}", HELLO);}
```

Datentypen

Length	Int. Signed	Int. Unsigned
8-bit	`i8` (-128-127)	`u8` (0-255)
16-bit	`i16` (-32k-32k)	`u16` (0-65535)
32-bit	`i32`	`u32`
64-bit	`i64`	`u64`
128-bit	`i128`	`u128`
arch	`isize`	`usize`

i/u-size = pointer-sized un-/signed integer type

Floating-Point-Numbers: `f32`, `f64`

Weitere: `bool`, `char`

Tuple

- **feste** # Elemente, **versch.** Typen möglich
- Zugriff via **Index** oder Tuple neuen Variablen zuweisen

```rust
let t:(i32,&str,bool)=(1, "B", true);
let b = t.1; // "B"
let (x, y, z) = t;
```

Unit Value ()

- genutzt, wenn **nichts** passieren soll oder wenn **nichts** returnt wird:

```rust
match dice_roll {
    6 => println!("You won!"),
    _ => (),
}
```

Tuple Structs

- Attribute **nicht** benannt sind (nur Datentyp)

```rust
struct TuSt(u8, u8, u8);
fn main() {
    let mut tuple = TuSt(5, 5, 5);
    tuple.0 = 255;
}
```

Structs

- Auf Variablen zugreifen: `Objekt.Attribut`
- Referencen bei Attributen ⟹ Lifetime Specifier
- Attribute verändern ⟹ Variable `mut`

Structs in Structs

```rust
struct Author {
    name: String,
}
struct Book {
    title: String,
    author: Author,
}
fn main() {
    let author = Author::new
        (String::from("..."));
    let book = Book::new
        (String::from("..."),
        author.clone());
}
```

Methoden `impl`

- `&self` / `&mut self` (`&mut self` Elem. `mut`)
- `self.` auf Attribute zugreifen
- an Struct **gebunden** & **gleicher** Namen
- mit `impl` markiert
- **mehrere** Implem. für **dasselbe** `struct` möglich (alle gleicher Name)

Objekte einer Klasse vergleichen:

```rust
struct Rect {
    w: u32,
    h: u32,
}
impl Rect {
    fn hold(&self,other:&Rect)->bool{
        self.w >= other.w && self.h
            >= other.h
    }
}
```

Input/Output

```rust
let mut input = String::new();
let result = io::stdin().read_line
    (&mut input); //Typ: Result
```

Mit `for` Impl. für **spez. Datentyp**:

```rust
struct Person {
    name: String,
}
trait CanRun {
    fn run(&self);
}
impl CanRun for Person {
    fn run(&self) {...}
}
```

Default Implementations

- direkt in **trait** angeben

```rust
pub trait Licensed {
    fn info(&self) -> String {
        String::from("...")
    }
}
```

Getters und Setter

- Setter ⟹ `&mut self` & Variable `mut`
- Rückgabewert `&mut Self` großschreiben, da Datentyp

```rust
fn set_name(&mut self, new_name:
String) -> &mut Self {
    self.name = new_name;
    self
}
```

Associated Functions

- **kein** Parameter `&self`
- Typischer Nutzen: **Konstruktoren**
- Aufbau: `Structname::Funktion(Parameter)`

Konstruktoren

- neue Objekte `Structname::new(Parameter)`

```rust
struct Per {
    name: String,
}
impl Per {
    fn new(name: String) -> Per{
        Per { name }
    }
}
fn main() {
    let p = Per::new("A".to_string());
}
```

Konstruktor als alleinstehende Funktion:

```rust
fn create_person(name: String, age:
u8) -> Person {
    Person { name, age }
}
```

Struct update syntax

- nur neue Attribute angegeben
- **Achtung**: alle neuen Attribute kommen **vor** `..` (auch sie im Struct danach)
- **alte** Objekt ⟹ **invalide**, da Ownership transf.
- prim. Typ kopiert, sonst .clone()

```rust
let my_struct1 = Student {
    id: 13,
    name: String::from("Bob"),
};
let my_struct2 = Student {
    id: 24,
    ..my_struct1
};
```

Komplexe Datentypen mit `.clone()`:

```rust
let my_struct2 = Student {
    ...
    ..my_struct1.clone()
};
```

Cell

- mutable memory location
- hat mut Wert auch wenn Cell in nicht-mut Variable gespeichert wird
- für Datentyp, die Copy-Trait implem.
- **keine** mut Referenzen auf inneren Wert!

RefCell

- wie Cell, nur nutz Lifetimes, um temp., exklusiv., mut Zugriff auf Wert in Cell haben
- Borrow rules werden zur Runtime gecheckt, nicht wie bei der Cell zur Compile-Time
- Vorteil zu Cell: Kann Datentypen halten, die Copy nicht implemn. oder größer sind

Funktionen

Funktion → Variablen ordnen, als Parameter übergeben & bei Funktion entgegennehmen:

```rust
fn main() {
    let function = less;
    let res = other(20,function);
}
fn less(num: &i32) -> bool {
    *num < 10
}
fn other(num: i32, func: fn(&i32) ->
bool) -> bool {
    func(&num)
}
```

Closure

- **anonyme Funktion** (d.h. ohne Name)
- Variablen zuordnen oder an Funkt. übergeben
- Closures Parameter in `|...|`

```rust
|param1:type, param2:type,...| {...}
```

Variable zuordnen:

(Type Annotation & geschwei. Kl. nicht notw.)

```rust
fn add_one_v1(x: u32) -> u32 {x + 1}
fn main() {
    let add_one_v2 = |x: u32| -> u32
        { x + 1 };
    let add_one_v3 = |x| { x + 1 };
    let add_one_v4 = |x| x + 1;
}
```

Closure **kann** auf Variablen **außerhalb** eigenen Scopes zugreifen:

```rust
let x = 10;
let my_closure = |input: i32| {
    x + input // kein Error
};
println!("closure output: {}",
my_closure(5));
```

Fn - Immutable borrow

- Borrows **immutable** values from external scope (kann Werte nur auslesen, **nicht** verändern)
- Häufig der **Standard**-Closure

```rust
let x = 10;
let my_closure = |input| {x + input};
```

FnOnce - Move (Copy/Clone)

- erhält **Ownership** Variablen äußeren Scope
- Closure kann nur **1x** ausgeführt werden (da Ownership äußeren Wertes nicht mehr da ist)
- mit `move` gekennzeichnet

```rust
let mut v = vec![1, 2, 3];
let mut c = move |input|v.push(input);
c(10);
```

FnMut - Mutable borrow

- Borrows **mutable** values from external scope
- mit `mut` gekennzeichnet

```rust
let mut x = 5;
let mut my_closure = |input| x +=
input;
my_closure(10);
```

Inline Functions

- Aufbau: `| Parameter | { Code }`

```rust
let hello = |name: &str| {...};
```

Iterator

- konsumiert via `sum()` oder `collect()`
- `.iter()` erzeugt Iterator aus z.B. Vektor (Iterator erhält **kein** Ownership. Wenn das möchte ⟹ `.into_iter()` ⟹ originale Variable invalid)
- `.next()` führt Itertor weiter

Spezieller Datentyp mit `.collect` erzeugen:

```rust
fn list_of_results() ->
Vec<Result<i32, DivisionError>> {
    let numbers = vec![27, 297,
    38502, 81];
    numbers
        .into_iter()
        .map(|n| divide(n, 27))
        .collect::<Vec<Result<i32,
        DivisionError>>>()
}
```

Map

- `.iter()` / `iter_mut()` convert z.B. Vek.→Iter.
- `| ... |` greift auf jedes Element im Iter. zu
- `{ ... }` Code für jedes Element im Iter. aus
- `.map(| ... | { ... })`

```rust
let v: Vec<i32> = [1, 2, 3]
    .into_iter()
    .map(|x| x + 1)
    .rev()
    .collect();
```

- `.last()` ⟹ damit Iterator konsumiert wird

```rust
stack.iter_mut()
     .map(|value| { *value += 1 })
     .last();
```

Fold

- Aufbau: `.fold(acc-Start, | acc, x| Operation)`

Zählt wie viele Elemente in Stack:

```rust
stack.iter()
     .fold(0, |count, _| count + 1);
```

println!()

- Debug: `"{:?}"` (braucht Debug-Trait)
- Pretty Print: `"{:#?}"` (braucht Debug-Trait)
- Adresse in Speicher von Pointer: `"{:p}"`

Filter

Gibt Elemente weiter, die Kriterien erfüllen:

```rust
let v1 = vec![1, 2, 3, 4, 5, 6];
let v2: Vec<_> = v1
    .iter()
    .filter(|item|(*item % 2 != 0))
    .map(|item| item) // nicht notw.
    .collect(); // [1, 3, 5]
```

Arrays

- **feste** Größe (für flexible Größe ⟹ Vektoren!)
- alle **gleicher** Datentyp

```rust
let a: [i32; 5] = [1, 2, 3, 4, 5];
let first = a[0]; // 1
let slice = &a[1..3]; // [2, 3]
```

If-Else-Statements

```rust
if number < 5 {
    ...
} else {
    ...
} // kein Semikolon!
```

Traits

- wie Interfaces
- nutzen `impl` für tatsäch. Implem.
- erzwingen Implem. Methoden
- Methoden Parameter `&self` / `&mut self`

```rust
struct Person {...}
trait FullName {
    fn name(&self) -> String;
}
impl FullName for Person {
    fn name(&self) -> String {...}
}
```

Wichtige Traits:

- **PartialOrd** - Ordnen
- **Copy** - primär **einfache** Datentypen
- **Clone** - primär **komplexe** Datentypen
- **Debug** - Ausgabe via `{:?}`
- **Deref** - z.B. `*v`

Standard-Traits:

```rust
#[derive(Debug,Clone,Copy,Deref)]
struct Counter {...}
```

Trait inheritance

Traits können von anderen Traits erben

Wenn Kind-Trait für Typ implem. wird, muss Eltern-Trait auch implem. werden.

```rust
trait Animal {...}
trait Flyable: Animal {...}
```

Trait bounds

```rust
struct Node<T: Debug> {
    elem: T,
    next: Link<T>,
}
```

Bei `impl`:

```rust
impl<T:Display+PartialOrd>Pair<T>{...}
```

Trait imple., wenn Typ schon andere Trait imple.:

```rust
impl<T: Display> ToString for T {...}
```

Bei Return-Typ (darf nur **1** Datentyp returnen!)

```rust
fn get_string() -> impl Clone + Debug
{...}
```

Bei mehreren Generics & mehrere Datentypen mit `where` organisieren:

```rust
fn some_function<T, U>(t: &T, u: &U) -
> i32
where
    T: Display + Clone,
    U: Clone + Debug,
{
```

Bei Traits:

```rust
trait Stack<T : Debug + PartialEq +
Clone> {...}
```

OOP

Abstraction: big picture without knowing all details of code, z.B. traits & impls

Encapsulation: bundling of data with methods that work on the data, z.B. modul., traits, structs

Inheritance: type-based or code-based inheritance, z.B. trait

Polymorphism. child classes can be used were the parent class(es) can be used, z.B. Generics, Trait Bounds

Trait Bounds für Datentypen

- für spezielle Datentypen, nicht Generics

```rust
fn func(item: &(impl Summary +
Display)) {...}
fn func<T: Summary + Display>(item:
&T) {...}
```

- Beide Möglichkeiten **äquivalent**.
- nimmt Referenz entgegen.
- Wenn keine Referenz:

```rust
fn some_func(item: impl SomeTrait +
OtherTrait) -> bool {...}
```

Default-Implementierung

Gilt solange nicht durch andere Implementierung überschrieben:

```rust
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("...")
    }
}
```

Loop

- Endlosschleife: `loop{}`
- `break`, `continue`

`break` kann Rückgabewert haben:

```rust
let result = loop {
    counter += 1;
    if counter == 10 {
        break counter * 2;
    }
};
```

Break aus labeled loop:

```rust
'counting_up: loop {
    loop {
        if count == 2 {
            break 'counting_up;
        }
    }
}
```

For-Scheifen

Mit Vektor:

```rust
let v: Vec<char> = vec!['B','o','b'];
for c in &v {
    print!("{}", c);
}
```

Mit Vektor als Iterator und Dereferenzieren:

```rust
let mut v1 = vec![1, 2, 3, 4, 5];
for item in v1.iter_mut() {
    *item += 1;
}
```

Mit Array:

```rust
let values: [i32; 5] = [1, 2, 3, 4];
for x in values.iter() {
    println!("{}", x);
}
```

While

```rust
while number != 0 {
    number -= 1;
}
```

HashMaps

- `HashMap<K, V>`, K = Key, V = Value
- Keys & Values jeweils **selben** Datentyp
- Um Werte hinzuzufügen ⟹ `mut`

```rust
let mut s: HashMap<String, f32> =
HashMap::new();
scores.insert(String::from("A"), 1.3);
```

Wichtige Funktionen:

- `.insert(*key*,*value*)`
- `.get(*key*)` (gibt Option zurück)
- `.contains_key(*key*)`
- `.entry(*key*).or_insert(*value*)` testet, ob Key exi. (returnt ihn, falls der Fall), sonst Value hinzugefügen

```rust
scores.entry(String::from("Bob")).or_i
nsert(2.0);
```

Refutability

Refutable:

- `if let Some (x) = a_value`
- `while let`

Irrefutable (keine unbekannte Zustände):

- `match`
- `let x = 5`
- `For-Loops`

Any

- **Nach** `dyn` kommt **Trait**
- Für Typ eines Trait-Objekts
- → Dynamic Dispatch
- Referenz `&dny Any` wird genutzt, um Datentyp Objektes herausfinden

```rust
fn test(s: &dyn Any) -> bool {
    TypeId::of::<String>() ==
        s.type_id()
}
```

Structs in Box wrappen

```rust
struct ErrorOne;
impl Error for ErrorOne {}
struct ErrorTwo;
impl Error for ErrorTwo {}
fn func(input: u8) -> Result<String,
Box<dyn Error>> {
    match input {
        0 => Err(Box::new(ErrorOne)),
        1 => Err(Box::new(ErrorTwo)),
        _ => Ok("ABC".to_string()),
    }
}
```

Options

- Wert in `Some()` **entpacken**!

Wichtige Funktionen:

- `.is_some()` und `.is_none()`
- `.take()` nimmt Ownership, hinterlässt None (wenn Wert schon None, passiert nix)

Error Handling

- `.is_ok()` & `.is_err()` für Check
- `error.kind()` :

```rust
match error.kind() {
    ErrorKind::NotFound => ...,
    other_error => panic!(".."),
};
```

Error bei Funktion zurückgeben:

Achtung: Auch "guten" Fall in `Ok()` wrappen!

```rust
fn func(n: String)->Result<String,
String> {
    if n.is_empty() {
        Err("was empty".into())
    } else {
        Ok("not empty".into())
    }
}
```

Recoverable Errors

- Vom Typ: `Result<T, E>`

Unrecoverable Errors

- Verursachen Makro: `panic!`

Questionmark Operator

- Wenn Err/None ⟹ return Err/None, sonst unpacken
- Nur in Funk. nutzen, die Option/Result return

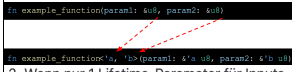
```rust
let o = result?;
match result {
    Err(e) => return Err(e),
    Ok(o) => o
}
```

Möglichkeiten zu unwrappen

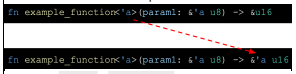
- `.unwrap()` - gibt Wert zurück oder panic
- `.expect(*Fehlermeldung*)` - gibt Wert zurück oder panic & Fehlermeldung
- `.unwrap_or(*Wert*)` - Wert zurück oder hier definierter Wert
- `.unwrap_or_else(*Closure*)` - gibt Wert zurück oder Wert mit **Closure** berechnet
- Question-Mark-Operator

Lifetime Elision Rules

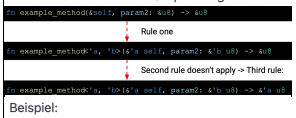
1. Für jeden Parameter, der eine Referenz ist, werden Lifetimes bestimmt



2. Wenn nur 1 Lifetime-Parameter für Inputs existiert ⟹ alle Outputs diese Lifetime



3. Wenn `&self` / `&mut self` in Parametern ⟹ Lifetime von self allen Outputs zugewiesen



Beispiel:

```rust
fn get_name1() -> &str {
    "John"
} // error: missing lifetime specifier
fn get_name2() -> &'static str {
    "John"
} // works
fn get_name3() -> String {
    "John".to_string()
} // works
```

Ownership

1. **normal** übergeben ⟹ Funktion übernimmt **Ownership** & Variable geht **out of scope**.

```rust
fn take_ownership(s: String) { ... }
fn main() {
    let s = String::from("Hello");
    take_ownership(s);
}
```

2. **normal** übergeben & Wert **zurückzugeben** & Variable **zuzuordnen**

```rust
fn give_back(s: String) -> String
    {...}
fn main() {
    let s1 = String::from("Hello");
    let s2 = give_back(s1);
}
```

3. **Referenz** übergeben ⟹ Funktion **kein** Ownership & gibt zurück (mut. Refere. `&mut`)

```rust
fn take_ref(s: &str) {...}
fn main() {
    let my_s = String::from("Hello!");
    take_ref(&my_s);
}
```

Ownership-Regeln

- Jeder Wert hat 1 Inhaber
- Nur 1 Inhaber zur selben Zeit
- Wenn Inhaber out of scope ⟹ Wert wird gelöscht

Dereferencing

```rust
let mut val = 42;
let val_ref = &mut val;
*val_ref = 27;
println!("{:?}", val); // 27
```

Enums

Zugriff auf Elemente: `Enum::Element`

```rust
#[derive(Debug)]
enum StudyProgram {
    INF,
    MINF,
}
fn main() {
    println!("{:?}",
    StudyProgram::MINF);
}
```

Enums & Structs

```rust
enum Mess {
    Move { x: u8, y: u8 }, // Struct
    Echo(String), // TupleStruct
    Color(u8, u8), // TupleStruct
    Quit,  // Normales Element
}
let message = [
    Mess::Move {x: 10, y: 30},
    Mess::Echo(String::from("y")),
    Mess::Color(200, 255),
    Mess::Quit,
];
```

Enum in Struct:

```rust
enum Program {
    INF,
    MINF,
}
struct Student {
    id: u32,
    name: String,
    program: Program,
}
fn main() {
    let my_struct = Student {
        id: 13,
        name: String::from("Bob"),
        program: Program::INF,
    };
}
```

Structs in Enums:

```rust
enum Person {
    Lecturer {
        name: String,
    },
    Student {
        name: String,
        program: Program,
    },
}
```

'static Lifetime

`'static` = Laufzeit Programms (wenn **außerhalb** main-Funktion deklariert)

```rust
static NUM: i32 = 18;
```

Anonymous Lifetime

`'_` genutzt, um Lifetime **autom. festzustellen**

Generics

```rust
struct Point<T, U> {
    x: T,
    y: U,
}
```

Achtung: Generic zu jedem Zeitpunkt nur **1 Datentyp**

Generic Method

Generic implementation:

```rust
impl<T> Point<T> {
    fn swap(self) -> Point<T> {
        Point::<T> {
            x: self.y,
            y: self.x,
        }
    }
}
fn main() {
    let p_int = Point {x:50, y:100};
    Point::swap(p_int));
}
```

Match

- **Alle** Möglichkeiten **verarbeiten**

Mit `other`:

```rust
match user_level {
    0 => println!("..."),
    other => println!("{}", other),
}
```

Mit Wildcard (**Achtung**: Wert Wildcard kann danach nicht nutzen):

```rust
match user_level {
    0 => println!("..."),
    _ => println!("..."),
}
```

Mit Ranges:

```rust
match input {
    0..=9 => println!("{}", input),
    _ => println!("..."),
}
```

Mit Bedingungen bei Attributen des Structs:

```rust
match point {
    Point { x, y: 0 } => println!("
    {}", x),
    Point { x, y } => println!("{}
    {}", x, y),
}
```

Der 2. Ast ist der Catch-All-Ast.

Mit Ranges und `|` (= **or**):

```rust
match hour {
    11..=14 | 15..=17 => println!
            ("..."),
    9..=18 => println!("..."),
    _ => println!("..."),
}
```

Mit Guards (= if-Statement):

```rust
match input {
    Yo(x,_) if x%2 == 0 =>
        println!("{}",x),
    Yo(x,y) => println!("{},{}",x,y),
}
```

Man kann Guards auch so nutzen:

```rust
match value {
    x if x > 0 => ...,
    x if x == 0 => ...,
    x if x < 0 => ...,
}
```

Mit Rückgabewert:

```rust
let fluffy_name = match fluffy {
    Pet::Dog { name } => name,
    Pet::Cat { name } => name,
};
```

Match & Enum

```rust
fn status(status: Program) {
    match status {
        Program::INF => ...,
        Program::MINF => ...,
        other => ...,
    }
}
fn main() {
    status(Program::INF);
    status(Program::MINF);
    status(Program::Other);
}
```

Bindings

Achtung: Wert von Ranges kann man danach **nicht** mehr nutzen!

```rust
match reg_ref {
    Point{x:0..=100,y:0,} => ...,
    _ => (),
}
```

⟹ Wert mit `@` einer neuen Variable zuordnen:

```rust
match reg_ref {
    Point{x:x_new@0..=100,y:0,} =>...,
    _ => (), }
```

if let

- muss **nicht** alle Möglichkeiten verarbeiten

```rust
if let 0..=12 = hour {
    ...
} else if let 13..=24 = hour {
    ...
} else { ... }
```

while let

```rust
let mut counter = 0;
while let 0 | 1 | 2 | 3 = counter {
    println!("{}", counter);
    counter += 1;
}
```

Ref

- **unpackt Option automatisch**
- Wert ist **borrowed**, nicht **moved** (kann später genutzt werden)
- Match-statements konsumieren alles ⟹ `ref` verhindert das

```rust
let name = Some(String::from("Bob"));
match name {
    Some(ref n) => ...,
    _ => ...,
}
```

Smart Points

- ähnl. wie Pointer, nur mit eig. Fähigkeiten
- Haben **Ownership** über Daten
- z.B.: Vektoren, Strings, Boxen, Structs mit Deref-Trait...

Box

- selbst auf **Stack**, Daten auf **Heap**
- hat **bekannte** Größe (genutzt für Daten, deren Größe sonst nicht at compile time bekannt sind)
- vermeidet **unendliche Rekursion**
- vermeidet, dass große Daten kopiert werden, wenn Ownership wechselt
- Mit `*` auf Wert zugreifen

```rust
let age = Box::new(22);
let twice_age = *age * 2;
```

Erzwingt, dass Typ in Box Debug-Trait implem.:

```rust
fn create_box<T: Debug>(data: T) ->
Box<T> {Box::new(data)}
```

```rust
pub enum List {
    Cons(i32, Box<List>),
    Nil,
}
```

Vektoren

- Daten alle **gleichen** Datentyp
- Auf **Heap** gespeichert, Größe **veränderbar** (wenn `mut`!)
- Zugriff auf Elemente via **Referenzen**

3 verschiedene Wege Vektoren initial.:

```rust
let mut v1: Vec<i32> = Vec::new();
let mut v2 = Vec::<i32>::new();
let mut v3: Vec<char> = vec!['R',
    'u', 's', 't'];
let mut v4: Vec<String> = vec!
    [String::from("Hello"),
    String::from("World")];
```

Zugriff auf Elemente via Referenzen:

```rust
let v1 = vec!['R', 'u', 's', 't'];
println!("{:?}", &v1[2]); // 's'
```

Modules

```rust
mod my_module {
    pub fn transformer() {...}
}
mod tests {
    use super::my_module::transformer;
}
```

Veränderbar via Pointer:

```rust
for element in v.iter_mut() {
    *element *= 2;
}
```

Wichtige Funktionen:

- `.push(*element*)`
- `.remove(*index*)` (panic, Index nicht exist.)
- `.get(*index*)` (return **Option**)

Achtung: `push()` für **Werte** & `append()` für andere **Arrays**

Vektoren & Enums

Versch. Datentypen via Enum in Vek. speichern:

```rust
enum Cell {
    Int(i32),
    Float(f64),
    Text(String),
}
let row = vec![
    Cell::Int(3),
    Cell::Text(String::from("blue")),
    Cell::Float(10.12),
];
```

String

- **hat** Ownership über Elemente
- nutzt **keine** Indizes (da nicht klar, ob auf Bytes oder Chars zugreifen, aber man kann iterieren)
- **heap** allocated, **growable**, unbekannte Größe Compile Time
- **mutable** borrows (`&mut String`) oder **read-only** (`&str`)
- kann man einfach via `&` (deref coercion) oder `as_str()` zu `&str` machen

3 Wege Strings initial.:

```rust
let mut s1 = String::new();
let mut s2 = "...".to_string();
let mut s3 = String::from("...");
```

```rust
let mut s1 = String::from("A"); // A
s1.push('B'); // AB
s1.push_str("CD"); // ABCD
```

Strings mit `format!()` konkatenieren:

```rust
let s1 = String::from("Rust");
let s2 = String::from("Lecture");
let s = format!("{s1}{s2}");
```

Mit `.chars()` kann über Buchstaben iterieren:

```rust
let word = String::from("ABC");
for c in word.chars() {...}
```

UTF-8 Strings auf 3 Arten interpretiert werden:

- Reihe von **Bytes** im Speicher
- Reihe von **Chars** im Speicher
- Grapheme clusters (Zeichen auf Bildschirm)

String Slices `&str`

- **read-only, immutable Reference**

(mutable Referenz ⟹ `&mut String`)

- **kein** Ownership
- kann Indizes nutzen
- initialisieren: `let s = "Hello, world!";`
- **nicht** dynamisch, d.h. Kapazität kann sich **nicht** ändern
- mit `.to_string()` zu String machbar

```rust
let s = String::from("hello world");
let hello = &s[0..5];
let world = &s[6..11];
```

Dynamic Dispatch `dyn`

- man nimmt Typ mit Trait, aber spez. Typ nicht bekannt at compile time ⟹ dyn. festgestellt
- Return-Typ in Box wrappen ⟹ auch Form dyn. Dispatch

```rust
trait Animal {...}
struct Cat {}
struct Dog {}
fn animal_talk(a: &dyn Animal) {...}
```

Static Dispatch

- Datentypen sind at compile time bekannt
- Rust macht aus allg. Version mit Generic spezif. Version mit spez. Datentypen

```rust
fn func<T:Foo>(x:T){x.method();}
```

Reference Counter RC

- Daten >1 Inhaber, zählt sie & gibt die Daten frei, wenn keine Inhaber mehr da

```rust
struct Su {}
enum Plan {Mer(Rc<Su>),}
...
let sun = Rc::new(Su {});
let mer=Plan::Mer(Rc::clone(&su));
drop(mer);
println!("{}",Rc::strong_count(&su));
```