

# 16.

---

Trait Objects, Drop, Smartpointer ...

---

# Vorweg: Raw Pointers

- Referenzen **&T** und **&mut T**
  - Für Maschine nur Pointer
  - *Garantie*: Zeigt auf ein gültiges Objekt von Typ **T**
- Raw Pointer **\*const T** und **\*mut T**
  - Keinerlei Garantien!
  - Dereferenzieren muss in **unsafe {}** Block passieren!
  - Werden wir wohl nie benutzen (außer zur Veranschaulichung in den Slides)

```
// We can create raw pointers in  
// safe Rust  
let p = 0xDEADBEEF as *const u32;  
  
// But dereferencing is considered  
// unsafe. The following code  
// will result in „segfault“  
let x = unsafe { *p };  
println!("{}", x);
```

**\*mut ()** ist Pointer ohne Typinformationen: Primitiv aber oft nützlich!

# Java zu Rust

```
interface Animal { void speak(); }  
class Dog implements Animal { ... }  
class Cat implements Animal { ... }
```

```
Animal choose() {  
    if (userInput()) {  
        return new Dog(...);  
    } else {  
        return new Cat(...);  
    }  
}
```

```
Animal a = choose();  
a.speak();
```

```
trait Animal { fn speak(&self); }  
struct Dog; impl Animal for Dog { ... }  
struct Cat; impl Animal for Cat { ... }
```

```
fn choose() -> Animal {  
    if user_input() {  
        Cat  
    } else {  
        Dog  
    }  
}
```

```
let a = choose(); // type `Animal`??  
a.speak();
```


error: the trait bound  
**Animal: std::marker::Sized**  
is not satisfied

# Monomorphization

- Spezielle Version für jeden Typ
- „Static Dispatch“: **call constant**
  - Funktionspointer von **speak()** in jeder Spezialisierung bekannt
- Nachteile:
  - *Code Bloat*: viel Maschinencode (in der Praxis selten ein Problem)
  - Typ muss zur Compilierzeit bekannt sein

```
fn speak_twice<A: Animal>(a: A) {  
    a.speak();  
    a.speak();  
}
```

```
let (a_cat, a_dog) = ...;  
speak_twice(a_dog);  
speak_twice(a_cat);
```



```
speak_twice<Cat>:    ; Cat version  
    ...  
speak_twice<Dog>:    ; Dog version  
    ...  
main:  
    call speak_twice<Cat>  
    call speak_twice<Dog>
```

# Monomorphization: Vorteile

- *Static Dispatch* schneller als *Dynamic Dispatch* (später mehr Details)
- Ermöglicht Compiler Optimierungen:
  - *Inlining*
    - Code von Funktion an Aufrufstelle kopieren, anstatt aufzurufen
    - Im Beispiel `#[inline(never)]` entfernen, um Effekt im Assembly zu sehen
    - Eine der wichtigsten Optimierungen
  - Weitere spezielle Optimierungen durch Wissen über Typ

→ i.d.R. deutlich schneller!

```
#[inline(never)]
pub fn add_self<T>(x: T) -> T
    where T: Add<Output=T> + Copy
{
    x + x
}

pub fn foo(a: u64, b: f64)
    -> (u64, f64)
{
    (add_self(a), add_self(b))
}
```

[Assembly zum Code](#)

# Dynamic Dispatch: Wie?

- **Ziel:**
  - Funktion, die zur Laufzeit die richtige `speak()` Funktion aufruft
  - Typ zur Kompilierzeit nicht bekannt!
- **Lösung:** Funktionspointer übergeben

```
let my_dog = Dog::new(...);  
  
// these casts don't quite work like this  
let fptr = Dog::speak as fn(*mut ());  
let aptr = &mut my_dog as *mut ();  
speak_twice(aptr, fptr);
```

```
// Types contain data  
struct Dog { name: String }  
struct Cat { legs: u8 }
```

```
fn speak_twice(  
    // pointer to some data  
    animal_data: *mut (),  
    // function pointer  
    speak_fptr: fn(*mut ()),  
) {  
    speak_fptr(animal_data);  
    speak_fptr(animal_data);  
}
```

# Dynamic Dispatch: Wie?

- Was ist mit mehreren Funktionen?  
→ *vtable* und *vptr* (von *virtual function*)
- *Vtable*: Speichert alle Funktionspointer
  - Muss nur einmal angelegt werden
- *Data-Pointer* (**self**): Zeigt auf Objekt

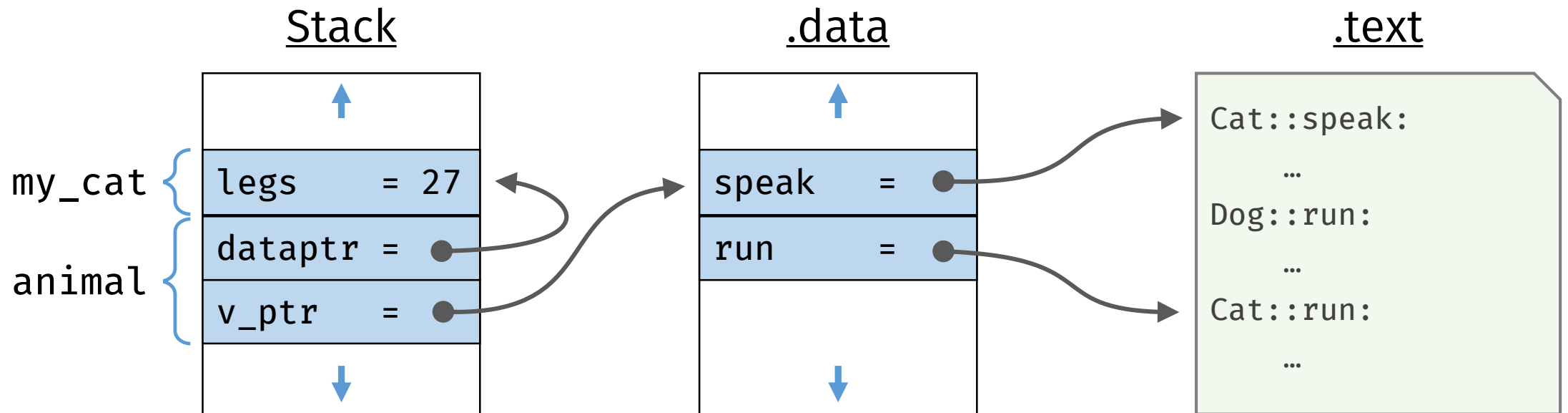
```
fn speak_twice(data: *mut (), vptr: &AnimalVTable) {  
    (vptr.speak)(data);  
    (vptr.speak)(data);  
}
```

```
trait Animal {  
    fn speak(&self);  
    fn run(&self);  
}  
  
struct AnimalVTable {  
    speak: fn(*mut ()),  
    run: fn(*mut ()),  
}  
  
static DOG_VTABLE: ... = ... {  
    speak: Dog::speak as ...,  
    run: Dog::run as ...,  
};  
static CAT_VTABLE: ... = ...;
```

# Trait Objects

- Referenz auf Trait (z.B. `&Animal`)
  - [Wird dargestellt durch `vptr` und `dataptr`](#)
  - Sog. „Fat Pointer“, ähnlich wie `&[T]`
- Nur „`Animal`“ ist ein unsized Typ

```
fn speak_twice(a: &Animal) {  
    a.speak();  
    a.speak();  
}  
  
let my_cat = Cat { legs: 27 };  
let animal: &Animal = &my_cat;  
speak_twice(animal);
```





# Vtable: ein bisschen komplizierter

- Vtable enthält weitere Felder
  - Später mehr Infos zum Destruktor
- [Komplette Dokumentation im Buch](#)

[Beispiel mit Assembly](#)

```
trait Animal {
    fn speak(&self);
    fn run(&self);
}

struct AnimalVTable {
    destructor: fn(*mut ()),
    size: usize, // currently unused
    align: usize, // currently unused
    speak: fn(*mut ()),
    run: fn(*mut ()),
}

static VTABLE_DOG_AS_ANIMAL = ...;
static VTABLE_CAT_AS_ANIMAL = ...;
```

# Trait Objects zurückgeben

```
fn choose() -> &Animal {  
    if user_input() {  
        &Cat { legs: 3 }  
    } else {  
        &Dog::new(...)  
    }  
}
```

error: does not live  
long enough

```
fn choose() -> Box<Animal> {  
    if user_input() {  
        Box::new(Cat { legs: 3 })  
    } else {  
        Box::new(Dog::new(...))  
    }  
}
```

- Trait Objects immer hinter Pointer
- Zurückgeben aus Funktion mit Heap-Allokation (**Box**)
  - DSTs auf dem Stack nur schwierig möglich! (In Rust noch gar nicht)
- Falls endlich viele Optionen: Enum dafür erstellen

# Trait Objects: Einschränkungen

- Trait ist „*object safe*“, wenn:
  - Alle Methoden sind *object safe*; und
  - Trait verlangt nicht **Self: Sized**
- Methode ist *object safe*, wenn:
  - Methode verlangt **Self: Sized**; oder
  - Folgendes muss zutreffen:
    - Keine Typparameter; und
    - Besitzt Receiver-Argument (**&self**); und
    - Benutzt nicht **Self**
- [Mehr Informationen](#)

```
trait Clone {  
    fn clone(&self) -> Self;  
}
```

```
fn recover_type(d: &Clone) {  
    // What's the type of x?  
    let x = d.clone();  
}  
  
let v = Vec::new();  
recover_type(&v);
```

error: the trait `**Clone**` cannot  
be made into an object



# Trait Object mehrerer Traits

```
fn literate_animal(a: &(Animal + io::Read)) { ... }
```

- Funktioniert nicht: Wir bräuchten zwei Vptrs
  - Oder: eine gemeinsame Vtable ([simuliert durch eigenes Trait](#))
  - Theoretisch möglich, aber schwierig (in Rust nicht möglich!)
- **Aber:**

```
fn foo(a: &(Animal + Sync + Send)) { ... } // works
```

- **Sync** und **Send** als spezielle Marker-Traits
- Lifetime Bounds (später mehr)

# Übersicht

## Static Dispatch

```
fn foo<T: Trait>(x: T)
```

- Durch Monomorphization
- Deutlich schneller
  - Auch durch weitere mögliche Optimierungen
- Mehr Möglichkeiten, wenn Compiler den Typ kennt
- In Rust meist Static Dispatch

## Dynamic Dispatch

```
fn foo(x: &Trait)
```

- Durch Vtable und *Type Erasure*
- Fast immer deutlich langsamer
- Konkreter Typ muss nicht zur Kompilierzeit gekannt werden
- Standard in Sprachen wie Java
  - Unter anderem der Grund, warum gewisse Sprachen langsamer sind

# Unsize Types Übersicht

- Oder: Dynamically Sized Types (*DSTs*)
- Built-in:
  - Slice `[T]` (`str` ist newtype um `[u8]`)
  - Trait Objects
- Eigene Typen:
  - Structs mit unsized type als letztes Feld
    - Beispiel: `std::path::Path`
    - So *könnte* auch `str` definiert sein
- Referenz auf Unsize Types immer „Fat Pointer“: Vervollständigen Typ

```
// Not the original definition!  
struct Path([u8]);  
// `Path` is now unsized
```

# Drop Trait

- In C++: Destruktor
- Methode **drop()** wird für jedes Binding aufgerufen, welches den Scope verlässt

```
trait Drop {  
    fn drop(&mut self);  
}
```

```
struct Cat;  
  
impl Drop for Cat {  
    fn drop(&mut self) {  
        // Take this, QM!  
        println!("Cat is dead!");  
    }  
}
```

```
fn main() {  
    let c = Cat;  
    println!("Mhh... ?");  
}
```

```
Mhh... ?  
Cat is dead!
```

# Drop

- Zum Aufräumen:
  - Freigeben von Speicher
  - Schließen von Verbindungen
  - ...
- Methode kann nicht manuell aufgerufen werden
  - Compiler sorgt dafür, dass **drop()** maximal einmal pro Variable aufgerufen wird<sup>1</sup>
  - Manuell droppen mit Funktion **drop()**

```
let mut c = Cat;  
c.drop(); // error
```

```
let c = Cat;  
drop(c); // that's fine
```

<sup>1</sup> Es wird nicht garantiert, dass **drop()** überhaupt aufgerufen wird. Dies ist aber fast immer der Fall. Siehe auch [forget\(\)](#).



# Drop: Reihenfolge

- Von innen nach außen
- Entgegen Initialisierungsreihenfolge
- Move droppt nicht!

```
fn main() {  
    let a = EchoDrop { c: 'a' };  
    let b = EchoDrop { c: 'b' };  
    {  
        let c = EchoDrop { c: 'c' };  
    }  
    let d = EchoDrop { c: 'd' };  
}
```

[Playground](#)

```
struct EchoDrop {  
    c: char,  
}  
  
impl Drop for EchoDrop {  
    fn drop(&mut self) {  
        println!("dropped: {}", self.c);  
    }  
}
```

```
fn main() {  
    let a = EchoDrop { c: 'a' };  
    let b = EchoDrop { c: 'b' };  
}
```

```
dropped: b  
dropped: a
```

# Smartpointer

- **Dumb Pointer:** Nur Adresse, kümmert sich um nichts
- **Smartpointer:** Verwaltet Owner und löscht letztlich den Pointee
- Einfachster Smartpointer: **Box<T>**
  - Genau einen Owner
  - Löscht Pointee wenn Scope zuende
  - Kein Overhead
  - In C++: **unique\_ptr<T>**

## Dumb Pointer (in C)

```
int main(int argc, char** argv) {
    int *p = malloc(sizeof(int));
    *p = 3;
    // pointer does nothing at the end
    // of the scope ... → memory leak
}
```

```
fn main() {
    let b = Box::new(3);
    // When the scope ends, b is
    // dropped. The Drop-impl will
    // drop the value and free
    // the memory.
}
```

# Pseudo Box Implementation

```
struct Box<T> {  
    ptr: *mut T,  
}  
  
impl<T> Box<T> {  
    fn new(t: T) -> Self {  
        unsafe {  
            let ptr = heap::allocate(mem::size_of::<T>());  
            *ptr = t;  
            Box { ptr: ptr }  
        }  
    }  
}
```

```
impl<T> Drop for Box<T> {  
    fn drop(&mut self) {  
        unsafe {  
            mem::drop_in_place(self.ptr);  
            heap::deallocate(self.ptr);  
        }  
    }  
}
```

Dies ist halber Pseudocode  
und nicht die richtige  
Implementierung!

# Reference Counted

- **Rc<T>** (Reference Counted):
  - Mehrere Owner
  - Löscht Pointee, wenn Scope des letzten Owner zuende ist
  - Erlaubt nur lesenden Zugriff
- **Arc<T>** (Atomically RC):
  - Wie **Rc<T>**, aber hat atomaren Ref-Counter (langsamer)
  - Kann an andere Threads geschickt werden

```
let a = Rc::new("hi".to_string());
{
    // This won't clone the string!
    let b = a.clone();

    // a and b reference the same
    // string on the heap
    println!("{}", *b);
}
// the string is not freed yet!
println!("{}", *a);

// When a's scope ends, the string is
// finally freed
```

# Pseudo Implementation

```
struct Shared<T> {
    ref_count: usize,
    data: T,
}

struct Rc<T> {
    ptr: *mut Shared<T>,
}

impl<T> Rc<T> {
    fn new(t: T) -> Self {
        // allocate `Shared` on heap.
        // with ref_count = 1
    }
}
```

```
impl<T> Clone for Rc<T> {
    fn clone(&self) -> Self {
        self.ptr.ref_count += 1;
        Rc { ptr: self.ptr }
    }
}

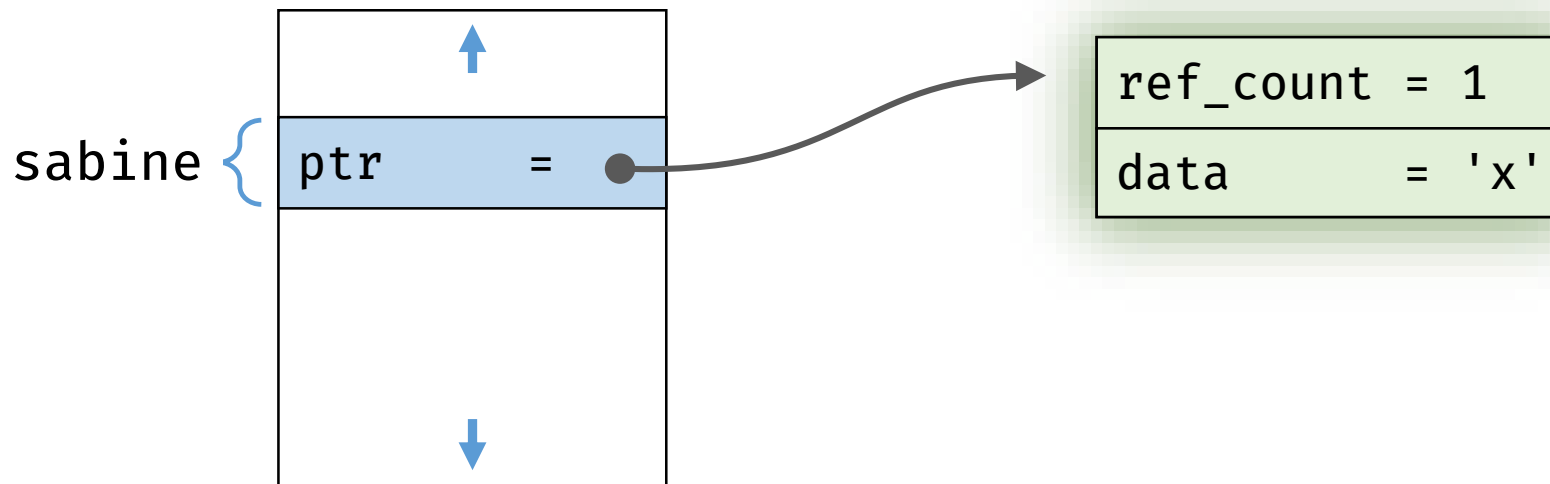
impl<T> Drop for Rc<T> {
    fn drop(&mut self) {
        self.ptr.ref_count -= 1;
        if self.ptr.ref_count == 0 {
            self.deallocate();
        }
    }
}
```

Dies ist halber Pseudocode und nicht die richtige Implementierung!

# Rc: Speicherlayout

```
fn main() {  
    let sabine = Rc::new('x');
```

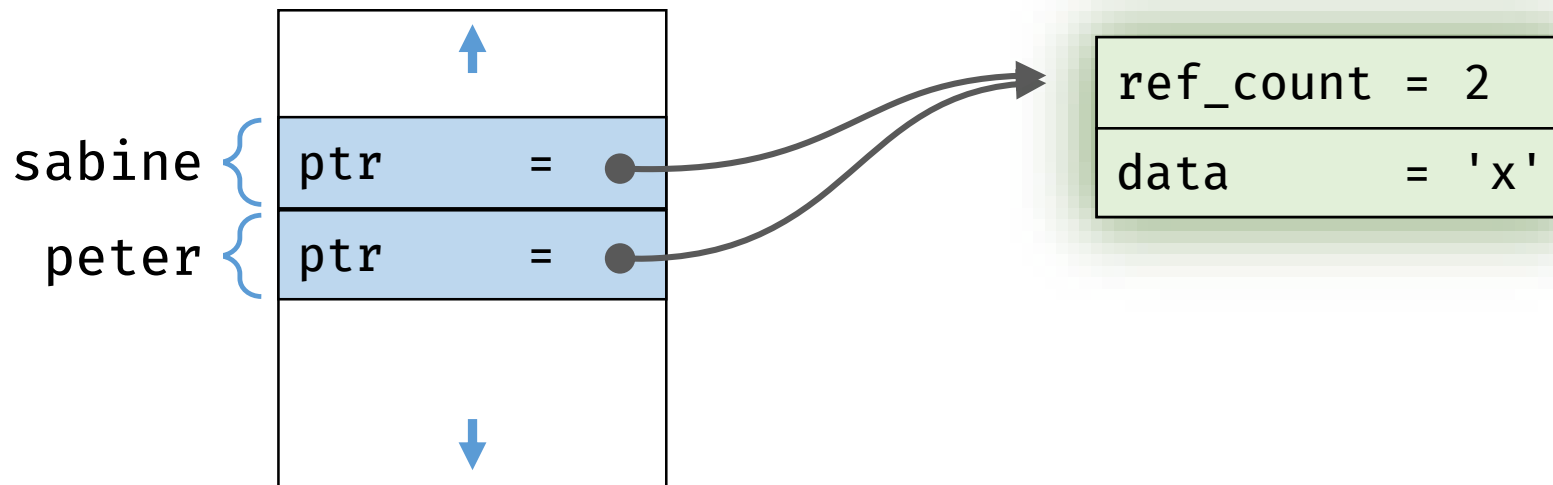
- `new()` → `ref_count = 1`



# Rc: Speicherlayout

```
fn main() {  
    let sabine = Rc::new('x');  
    {  
        let peter = a.clone();  
    }  
}
```

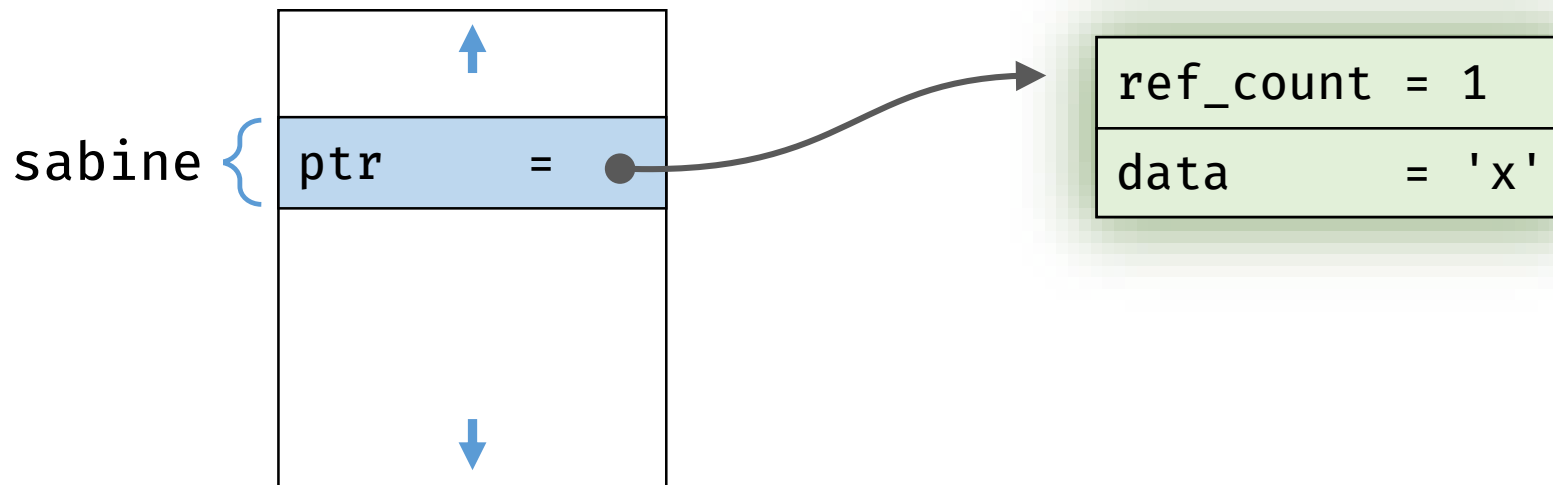
- `new()` → `ref_count = 1`
- `clone()` → `ref_count = 2`



# Rc: Speicherlayout

```
fn main() {  
    let sabine = Rc::new('x');  
    {  
        let peter = a.clone();  
    }  
}
```

- `new()` → `ref_count = 1`
- `clone()` → `ref_count = 2`
- `drop()` → `ref_count = 1`

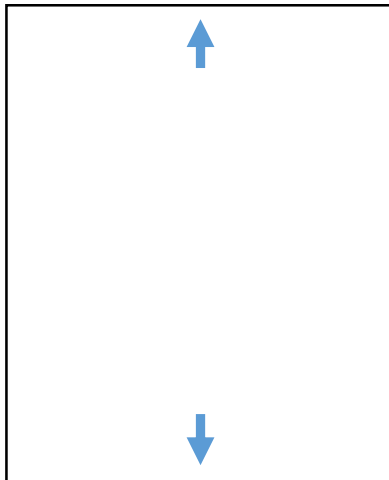




# Rc: Speicherlayout

```
fn main() {  
    let sabine = Rc::new('x');  
    {  
        let peter = a.clone();  
    }  
}
```

- `new()` → `ref_count = 1`
- `clone()` → `ref_count = 2`
- `drop()` → `ref_count = 1`
- `drop()` → `ref_count = 0`  
(dann gelöscht)



# Zyklen und Weak<T>

- Rc-Zyklus führt zu Speicherleak
  - Ref-Count fällt nie auf 0
- Zum Vermeiden von Zyklen: **Weak<T>**
  - Existenz erhöht Ref-Count nicht
  - Kann erfragen, ob Original-Objekt noch existiert
  - Erstellen mit **downgrade()**
  - Optionale Referenz auf Daten mit **upgrade()**
- Beispiel: Baum
  - Childpointer: **Rc**, Parentpointer: **Weak**

```
let weak = {  
    let orig = Rc::new(27);  
    let w = orig.downgrade();  
  
    // returns `Some(...)`  
    w.upgrade();  
  
    w  
}; // orig is dropped here  
  
// returns `None`  
weak.upgrade();
```

# Unsicherheit in einem Thread

„Wenn Aliasing *und* Mutability  
→ Schlimme Dinge!“ — Kapitel 3



Warum?

- Mutation kann alle bisherigen Referenzen invalidieren
- Aliasing schafft und versteckt Abhängigkeiten
- Anderes Beispiel: Iterator Invalidation

```
let mut v = Vec::new();  
v.push(1);
```

```
let x = &v[0];  
println!("{}", x);
```

```
v.push(2);  
v.push(3);
```

```
let y = &v[1];  
println!("{}", x, y);
```

Code kompiliert nicht!

# Moment mal...

- Veränderung über immutable Reference?
  - Interior Mutability

```
impl Display for Foo {  
    fn fmt(&self, f: ...) -> ... {  
        if self.cache.is_none() {  
            self.cache = Some(format!(...));  
        }  
        self.cache.unwrap().fmt(f)  
    }  
}
```

```
impl<T> Clone for Rc<T> {  
    fn clone(&self) -> Self {  
        self.ptr.ref_count += 1;  
        Rc { ptr: self.ptr }  
    }  
}
```

Der Code wird nicht kompilieren!  
Wir haben nur *immutable reference* **&self**!

Wie ist **Rc<T>** überhaupt implementiert?

# Interior Mutability

Von `&Something<T>` zu `&mut T` !

- Shareable mutable containers
  - Erlauben Mutation durch immutable Referenz
  - Mit einem Thread: `Cell<T>` und `RefCell<T>`
  - Mehrere Threads: `Mutex<T>` und `RwLock<T>` (Meist `Mutex<T>`)
- Höchst *unsicher*?!
  - Alle Container stellen intern zur Laufzeit sicher, dass die Regel „*Multiple Reads xor Single Write*“ eingehalten wird!
  - Aufwand von Kompilierzeit zur Laufzeit verschoben

# Cell

(in `std::cell`)

```
fn get(&self) -> T { ... }  
fn set(&self, value: T) { ... }
```

```
fn can_touch_this(x: &Cell<u64>) {  
    x.set(99);  
}  
  
let c = Cell::new(0); // no mut  
c.set(c.get() + 1);  
  
can_touch_this(&c);
```

- Funktioniert nur mit Copy-Typen
- Ermöglicht keine Referenzen auf inneren Wert
- Sicher (bei einem Thread)
  - Teilen mit anderen Threads vom Compiler verboten
- Kein Overhead!

# RefCell

- Muss explizit geborrowed werden (fn borrow\_mut(&self))
- Borrow-Guard zum Verwalten aktiver Borrows
- Guards bieten Zugriff auf Daten
  - Durch **Deref<Target=T>**
- Wenn Regel verletzt: panic!
  - Ist Programmierfehler
  - Warten hilft nicht, da alles in einem Thread

```
let c = RefCell::new(27); // no mut!
{
    // Returns a guard, that implements
    // Deref<Target=T>. As long as the
    // guard exists, any borrow_mut()
    // or borrow() call will fail.
    let mut guard = c.borrow_mut();
    *guard += 1;
    *guard += 1;

    // c.borrow_mut(); // would panic!

    // guard will be dropped here
}

*c.borrow_mut() = 3; // it's fine here
```

# Übersicht

|                              | Shared Ownership Pointer   | Interior Mutability Container   |
|------------------------------|--|---|
| Nur für einen Thread         | <ul style="list-style-type: none"><li>• <code>Rc&lt;T&gt;</code></li><li>• <code>rc::Weak&lt;T&gt;</code></li></ul>    | <ul style="list-style-type: none"><li>• <code>Cell&lt;T&gt;</code> (nur Copy-Typen)</li><li>• <code>RefCell&lt;T&gt;</code></li></ul>               |
| Für mehrere Threads geeignet | <ul style="list-style-type: none"><li>• <code>Arc&lt;T&gt;</code></li><li>• <code>sync::Weak&lt;T&gt;</code></li></ul> | <ul style="list-style-type: none"><li>• <code>Atomic*</code></li><li>• <code>RwLock&lt;T&gt;</code></li><li>• <code>Mutex&lt;T&gt;</code></li></ul> |

Später mehr!

- Häufig benutzt:
  - `Rc<RefCell<T>>`
  - `Arc<Mutex<T>>`

- Warum benutzen?
  - Manchmal unmöglich/aufwändig Regeln zur Kompilierzeit einzuhalten
  - Wenn nur lesender Zugriff möglich (**Rc/Arc**)
  - **Nicht** nutzen, um dem Borrowchecker auszuweichen!



# Andere Vorkommen von I.M.

- Modifikation erlaubt:
  - **File**
  - **TcpStream**
  - ein paar andere...
- Nicht so allgemein wie **RefCell** (und ähnliche), trotzdem eine Art von Interior Mutability

```
// `&File` implements `io::Write`  
fn write_file(f: &File) {  
    // works  
    f.write_all(&[3, 1, 4, 1, 5]);  
}
```

**N**un eine kleine Geschichte über den  
*Garbage Collector* (GC) und *RAII*...

Referenz: [Vortrag „Why I hate garbage collectors“](#)

# Speicherleak finden: C++

Schlechtes, altes C++

```
class Enemy {  
private:  
    EventFilter *filter;  
  
public:  
    Enemy(World *w, EnemyType t) {  
        // we only want to receive specific events  
        this->filter = new EventFilter(t);  
        w.register_event_handler(filter, this);  
    }  
};
```

Wem gehört das?  
Wer muss das freigeben?

**delete** fehlt eventuell!

# ~~GC to the rescue!~~

Java

```
class Enemy {  
  
    private EventFilter filter;  
  
    public Enemy(World w, EnemyType t) {  
        // we only want to receive specific events  
        this.filter = new EventFilter(t);  
        w.register_event_handler(filter, this);  
    }  
};
```

this noch  
in world! ☹️

```
List<Enemy> enemies;  
World w;  
  
if (random())  
    enemies.add(new Enemy(weak));  
  
// remove all dead enemies  
enemies.removeIf(e -> e.isDead());
```

GC löscht **filter!** 😊

## Memory Leak

`delete/free`  
vergessen

## Lingering Object

Referenz noch da,  
aber unbenutzt

## Resource Leak

Vergessen, Resource  
zu schließen



Garbage Collector



?????

# Was ist eine Ressource?

- Klassische Beispiele:
  - Datei
  - Netzwerk-Socket
  - Datenbankverbindung
  - ...
- *Aber:* Auch Speicher!
  - Alles kann als Ressource verwendet werden

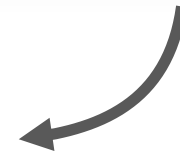


Alles mit „**open()/close()** Semantics“ ist eine Ressource!

# Lösung: RAI

Niemand mag den Namen...  
([nicht mal Stroustrup](#))

Resource **A**cquisition **i**s **I**nitialization



- Originale Grundidee:
  - Constructor *acquires* resource
  - Destructor *releases* resource
- Wichtige Hauptmerkmale:
  - Ein Destruktor ist für das Freigeben von Ressourcen verantwortlich
  - Destruktor (in Rust **drop()**) wird *garantiert* aufgerufen



Kann alle drei Arten von Leaks vermeiden!

# GC und RAI?

- Idee: GC für Memory-Leaks, RAI für alles andere
- Java: `finalize()`?
  - Unzuverlässig!
  - Wird erst aufgerufen, wenn der GC gerade Lust hat

## Why would you ever implement `finalize()`?



275



I've been reading through a lot of the rookie Java questions and I'm bewildered that no one has really made it plain that `finalize()` is for cleaning up resources. I saw someone comment that they use it to clean up resources since the only way to come as close to a guarantee that a `Closeable` (catch) finally.



65

I was not schooled in CS, but I have been programming in Java for a decade now and I have never seen anyone implement `finalize()`.

“`finalize()` is a hint to the JVM that it might be nice to execute your code at an unspecified time. This is good when you want code to mysteriously fail to run. [...]”

[Link](#)



# Nachteile vom GC

- Gibt falsches Gefühl der Sicherheit
  - Löst nicht alle Probleme (Leaks gibt es immer noch)
  - Gaukelt Programmieren vor, sie müssten sich keine Gedanken machen
- Erschwert das Behandeln anderer Probleme
  - Bietet gewisse Möglichkeiten nicht
- Verhindert richtiges Gedankenmodell
  - Über Ownership nachdenken sinnvoll
- *Achja*: Langsamer, „Stop the World“, braucht Runtime, ...

# Zusammenfassung Kapitel

- *Dynamic Dispatch* via Funktionspointer in Vtable
- *Static Dispatch* via Monomorphization
- Drop ist der Destruktor des *RAII* Patterns
  - *RAII* deutlich sinnvoller als *GC*
- Smartpointer: **Box**, **Rc** und **Arc**
- Interior Mutability mit **Cell**, **RefCell**, ...