



# Garbage Collection & Reference Counting

Memory-Management in Rust

Clemens Ruck & Alex Egger  
Summer Term 2017

# Inhalt

## Arten des Memory-Managements

- 1 Probleme bei manuellem Memory-Management
- 2 Garbage Collection
- 3 Rust's Ansatz

## Memory Management in Rust

- 1 Stack-Allokation
- 2 Heap-Allokation
- 3 Reference Counting in Rust
- 4 'unsafe' und Raw Pointer


# Manuelles Memory-Management

Dynamisch Speicher reservieren in C:

```
void *malloc(size_t size);
```

Speicher freigeben:

```
void free(void *ptr);
```

 Es ist leicht zu vergessen, wann und ob Speicher bereits freigegeben wurde, was zu unnötigen Bugs führen kann!

# Probleme - Memory Leaks

```
int main(void) {  
    while(1) {  
  
        // Allocate some amount of memory on the heap.  
        char *c;  
        if(!(c = malloc(20 * sizeof(char)))) {  
            perror("Could not allocate memory on heap.");  
            return 1;  
        }  
  
        // Do something with allocated memory.  
  
        // Memory is never freed, and can never be  
        // reclaimed by the system.  
    }  
    return 0;  
}
```

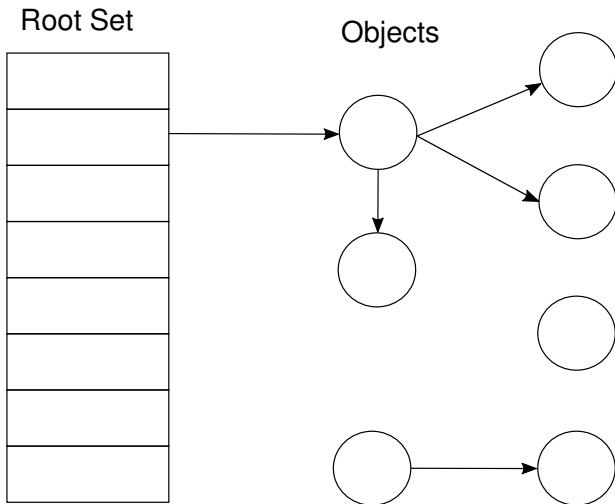
# Probleme - Use after Free

```
int main(void) {  
    // Allocate memory.  
    char *c = malloc(10);  
  
    // Do something and then free the memory.  
    free(c);  
  
    // The memory now doesn't belong to us anymore, so  
    // this will result in a segmentation fault.  
    (*c)++;  
  
    return 0;  
}
```

# Garbage Collection

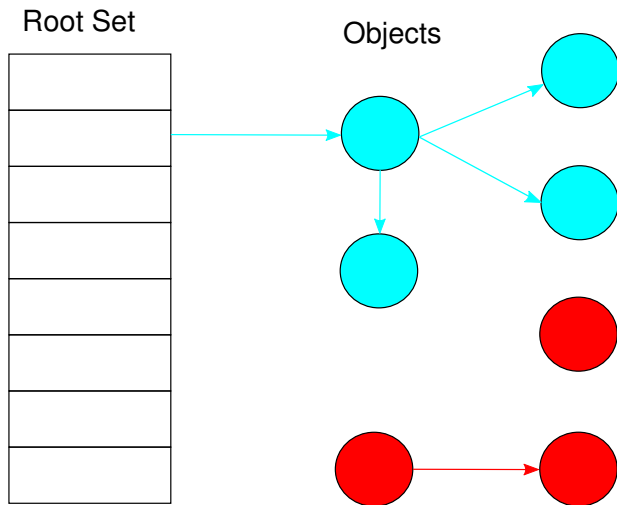
**Garbage Collection** ist eine Form des automatischen Memory-Managements. Hierbei wird versucht Speicher freizugeben, der von Objekten benutzt wird, deren Lebenszeit abgelaufen ist.

# Beispiel - Mark & Sweep



**Figure:** Eine Graphen-Darstellung von lebenden Objekten.

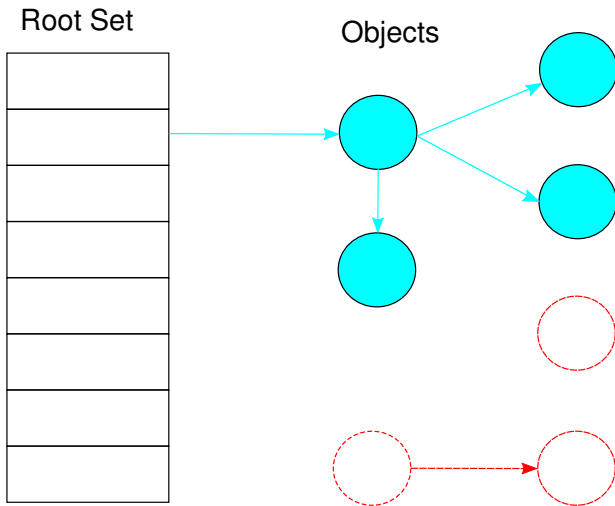
# Beispiel - Mark & Sweep



**Figure:** Die 'Mark'-Phase des Algorithmus.



# Beispiel - Mark & Sweep



**Figure:** Die 'Sweep'-Phase des Algorithmus.

# Memory-Management in Rust

In Rust werden Variablen standardmäßig auf dem Stack allokiert:

```
let x = 12; // Auf dem Stack.
```

Um Speicher auf dem Heap zu reservieren, benutzt man `Box<T>`:

```
let y = Box::new(56); // Auf dem Heap.
```

# Variablen auf dem Stack

Adresse	Name	Wert
0	x	42

```
1  fn main() {  
2      let x = 42;  
3      other();  
4  
5      //...  
6  }  
7  
8  fn other() {  
9      let y = 27;  
10     let z = 99;  
11 }
```

# Variablen auf dem Stack

Adresse	Name	Wert
2	z	99
1	y	27
0	x	42

```
1  fn main() {  
2      let x = 42;  
3      other();  
4  
5      //...  
6  }  
7  
8  fn other() {  
9      let y = 27;  
10     let z = 99;  
11 }
```

# Variablen auf dem Stack

Adresse	Name	Wert
0	x	42

```
1  fn main() {  
2      let x = 42;  
3      other();  
4  
5      ...  
6  }  
7  
8  
9  fn other() {  
10     let y = 27;  
11     let z = 99;  
12 }
```

# Speicherreservierung auf dem Heap

Adresse	Name	Wert
1	y	???
0	x	42

```
1 fn main() {  
2     let x = 42;  
3     let y = Box::new(39);  
4 }
```

# Speicherreservierung auf dem Heap

Adresse	Name	Wert
0xffff		39
...		
1	y	— > ffff
0	x	42

```
1 fn main() {  
2     let x = 42;  
3     let y = Box::new(39);  
4 }
```

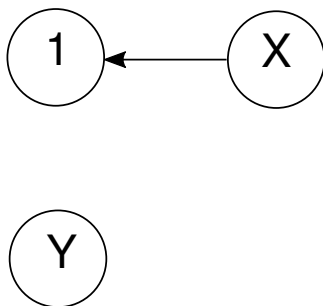
# Vergleich: Heap & Stack

- 1 Den Stack zu verwalten ist trivial
- 2 Den Heap zu verwalten ist aufwendig (z.B. wegen Fragmentierung)
- 3 Standardmäßig auf dem Stack zu arbeiten, lässt einfacher über die Lebenszeit von Variablen nachdenken



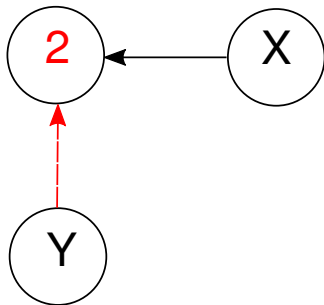
# Reference Counting

Referenzen können als gerichteter Graph dargestellt werden, wobei die Knoten Objekte darstellen und zwischen zwei Knoten eine gerichtete Kante besteht, falls eines eine Referenz auf das andere hält.



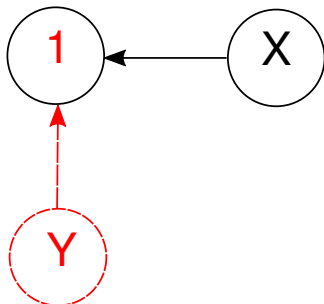
# Reference Counting

Wenn eine neue Referenz auf ein Objekt erstellt wird, wird der **Reference Counter** erhöht.



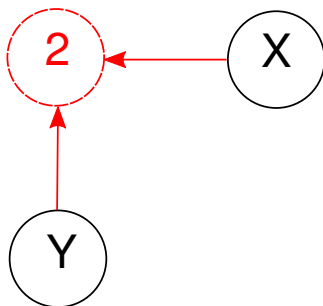
# Reference Counting

Wenn ein Objekt freigegeben wird, werden alle Referenzen die es hält freigegeben und deren jeweilige Referenzzähler dekrementiert.



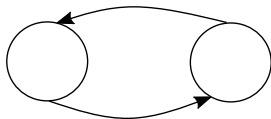
# Reference Counting

Objekte können nur freigegeben werden, wenn ihr **Referenzzähler auf 0** steht! Sonst werden die Referenzen, die noch auf das Objekt zeigen, ungültig.



# Limitierungen

⚠ Zyklische Referenzen können nicht freigegeben werden!



Dieses Problem kann durch den Einsatz eines Incremental Garbage Collectors gelöst werden, der spezifisch auf zyklische Referenzen ausgelegt ist. Eine andere Lösung ist es, zyklische Referenzen in den eigenen Datenstrukturen zu verbieten.

## Rc<T> in Rust


In Rust können wir Reference Counting benutzen, indem wir das `Rc<T>` Struct benutzen.

Um es zu nutzen, muss zuerst `Rc::new` aufgerufen werden:

```
let t = ...;  
let rc_t = Rc::new(t);
```

Um eine neue Referenz zu erstellen und den Referenzzähler zu erhöhen, wird `Rc.clone()` benutzt:

```
let second_rc_t = rc_t.clone();
```

 Um Reference Counting in Szenarien mit multiplen Threads zu benutzen, muss man `Arc<T>` benutzen, was inperformanter ist.

# Reference Counting - Beispiel

```
struct Owner {  
    name: String,  
    // Fields...  
}  
  
struct Car {  
    owner: Rc<Owner>,  
    // Fields...  
}
```

# Reference Counting - Beispiel

```
fn main() {  
    let owner = Rc::new(Owner{  
        name: "Lars",  
        // Fields...  
    });  
  
    let car = Car {  
        owner: owner.clone(),  
        // Fields...  
    }  
  
    let car2 = Car {  
        owner: owner.clone(),  
        // Fields...  
    }  
    // ...  
}
```



# Reference Counting - Beispiel

```
fn main() {  
    // ...  
  
    // Drop die lokale Variable 'owner'  
    drop(owner);  
  
    // Funktionert immer noch, da es immernoc  
    // Referenzen auf owner gibt!  
    println!("{}", car.owner.name);  
}
```

# 'unsafe'

Das `unsafe` Keyword erlaubt:

- ❶ Das Benutzen und Updaten von static mutable Variablen.
- ❷ Das Dereferenzieren von Raw Pointern.
- ❸ Das Aufrufen von als 'unsafe' markierten Funktionen.

# Raw Pointer

Es gibt zwei Typen von Raw Pointern:

1 `*const T`

2 `*mut T`

Raw Pointer sind einfache C-artige Zeiger auf einen Speicherbereich. Dabei unterliegen sie nicht den üblichen Ownership- oder Lifetimeregeln. Sie können nur in "unsafen" Codebereichen dereferenziert werden.

# Example

```
fn main() {  
    let i: u32 = 77;  
  
    // Einen Raw Pointer erstellen,  
    // kann man auch in safe Code.  
    let x = &i as *const u32;  
  
    // Dereferenzieren führt allerdings zu einem Fehler.  
    let y = *x;  
  
    // Wenn der Codeblock als unsafe markiert ist,  
    // ist auch dereferenzieren erlaubt.  
    unsafe {  
        let z = *x;  
    }  
}
```