15.

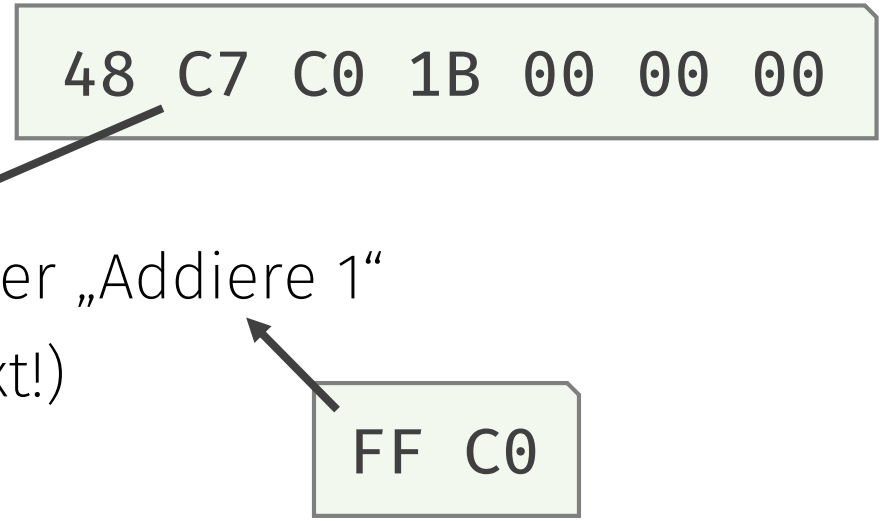
Low Level: Speicher, Assembly, ...

Motivation für Low Level

- Hilft, Rust Konzepte und Designentscheidungen zu verstehen
- Hilft, besseren Code in C/C++/Rust/… zu schreiben
  - Schneller/effizienter
  - In C und C++: Sicherer (in Rust müssen wir uns darum keine Sorgen machen ;-))
- Sinnvoll für andere Uni-Kurse
  - „Programmiersprache C++“, „Info C“, „Betriebssysteme“, …
- Assembly verstehen immer noch wichtig!
- Geschwindigkeit von Programmen/Sprachen besser verstehen

Maschinencode und Instructions

- *Maschinencode*: Reihe von *Instructions*
- *Instruction*: Primitiver Befehl für CPU
  - *Beispiel*: „Lade den Wert 27 an diese Stelle“ oder „Addiere 1“
  - Wird sehr kompakt kodiert (binär, nicht als Text!)
    - In x86_64: 1 bis 15 Bytes
  - Wird von CPU nacheinander ausgeführt (…)
- Hängt von CPU-Architektur ab
  - **x86_64:** Zurzeit quasi alle Desktop/Notebook CPUs (in diesen Slides genutzt)
  - **ARM**: Smartphones, Raspberry Pi und viele mehr…
  - …




```
55 48 89 E5 31 C0 48 85 FF 48 8D 47 FF 48 8D 4F
FE 48 F7 E1 48 0F A4 C2 3F 48 8D 44 3A FF 5D C3
```

Assembly

- Darstellung von *Maschinencode* als Text (für Menschen)
- Grundsätzliche zwei Syntaxen:
  - AT&T und Intel
  - Wir nutzen Intel-Syntax in den Slides
- Wir betrachten immer Assembly
  - **Aber**: „Alles ist binär kodiert“ im Hinterkopf behalten

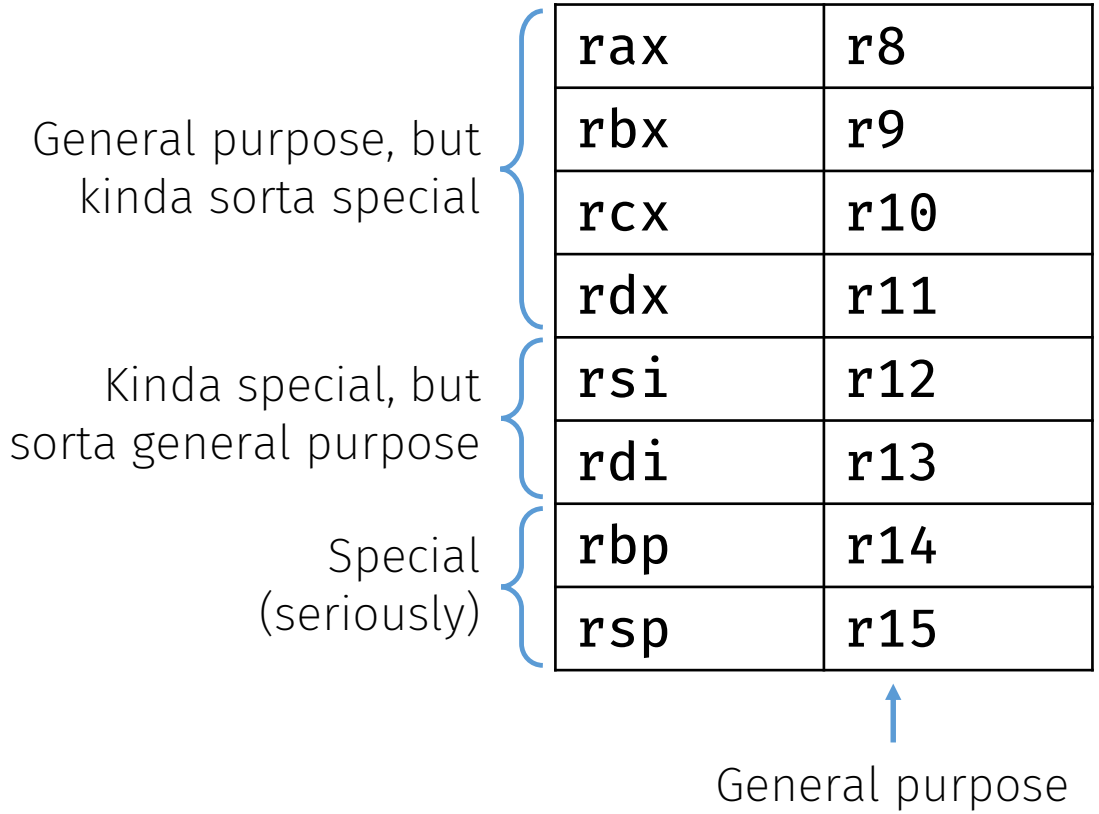
```
triangle:
        push    rbp
        mov     rbp, rsp
        xor     eax, eax
        test    rdi, rdi
        je      .LBB0_2
        lea     rax, [rdi - 1]
        lea     rcx, [rdi - 2]
        mul     rcx
        shld    rdx, rax, 63
        lea     rax, [rdx + rdi - 1]
.LBB0_2:
        pop     rbp
        ret
```

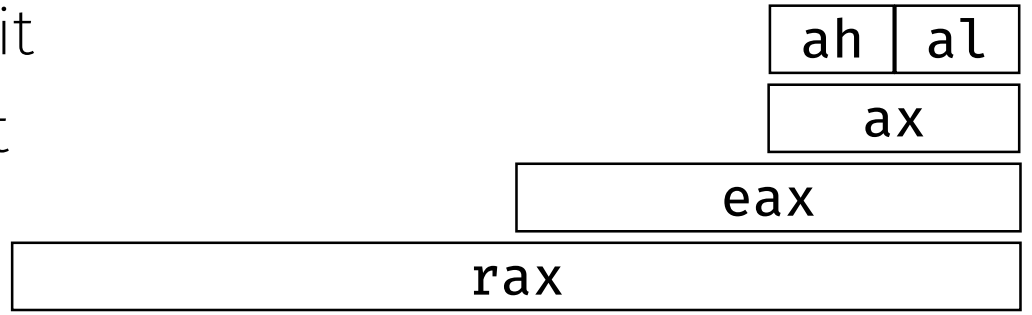
„Assembilieren“ ↓ ↑ „**De**assembilieren“

```
55 48 89 E5 31 C0 48 85 FF 48 8D 47 FF 48 8D 4F
FE 48 F7 E1 48 0F A4 C2 3F 48 8D 44 3A FF 5D C3
```

Register

- Speicher direkt in der CPU
- Extrem schneller Zugriff
- Für x86_64:
  - 16 Register
  - 64 Bit groß
- Teile der Register haben Namen:
  - `e??` bzw. `r??d` → unteren 32 Bit
  - `??` bzw. `r??w` → unteren 16 Bit
  - `wat` bzw. `r??b` → unteren 8 Bit





Beispiel-Instructions

- *Zuerst*: Art der Operation
- *Danach*: Argumente
  - Mit Komma getrennt
  - Register oder „*Intermediates*“
- Argumentreihenfolge manchmal merkwürdig
  - `mov` *`ziel, quelle`*

```
inc rax
```

```
FF C0
```

- Inkrementiert den Wert in **`rax`**

```
mov rax, 27
```

```
48 C7 C0 1B 00 00 00
```

- Füllt **`rax`** mit dem Wert 27

```
add rax, rbx
```

```
48 01 D8
```

- Addiere **`rbx`** auf **`rax`** auf (Ergebnis also in **`rax`**)

Sprünge

- Ausführung springt zu einer Adresse
  - Instruktion an dieser Adresse wird als nächstes ausgeführt
  - Addresse relativ zur jetztigen Instruktion
  - In Assembly werden Sprungziele benannt

```
main:                           ; comments start with ';'
    mov   rax, 27               ; load 27 into rax
    jmp   .loop_forever         ; unconditional jump
    mov   rbx, rax              ; would load the value of rax into
                                ; rbx, but won‘t be executed!

.loop_forever:
    jmp   .loop_forever         ; hihihi
```

Flags & Arten von Sprüngen

ZF	zero
SF	sign
CF	carry
…	

- Bedingungslose Sprünge: **`jmp`**
- *Flags*: Werden von Instructions gesetzt und gelesen
  - **`cmp rax, rbx`**: Subtrahiert **`rax`** von **`rbx`** und setzt *ZF*=1 wenn Ergebnis 0
- Bedingter Sprung:
  - **`jz`** (jump zero), **`je`** (jump equal): Sprung wenn *ZF* == 1
  - **`jnz`** (jump not zero), **`jne`** (jump not equal): Sprung wenn *ZF* == 0
  - Weitere: **`jb`** (below), **`jnb`** (not below), **`ja`** (above), **`jna`** (not above), …
- Dynamischer Sprung
  - **`jmp rax`**: Springt an die Adresse, die in **`rax`** gespeichert ist

Do-While-Schleife

```rust
let mut i = 0;    // will be in rax
let mut sum = 0;  // will be in rbx
do {  // there is no do-while in Rust!
    sum += i;
    i += 1;
} while (i != 10);
```

```
main:
    mov   rax, 0          ; load 0 into rax
    mov   rbx, 0          ; load 0 into rbx
.start_loop:
    add   rbx, rax        ; rbx += rax
    inc   rax             ; rax += 1
    cmp   rax, 10         ; compare (sets ZF=1 if equal)
    jne   .start_loop     ; if ZF==0 jump!

    nop                   ; Nothing to do anymore. Just for fun:
                          ; nop (no operation) does nothing :-)
```

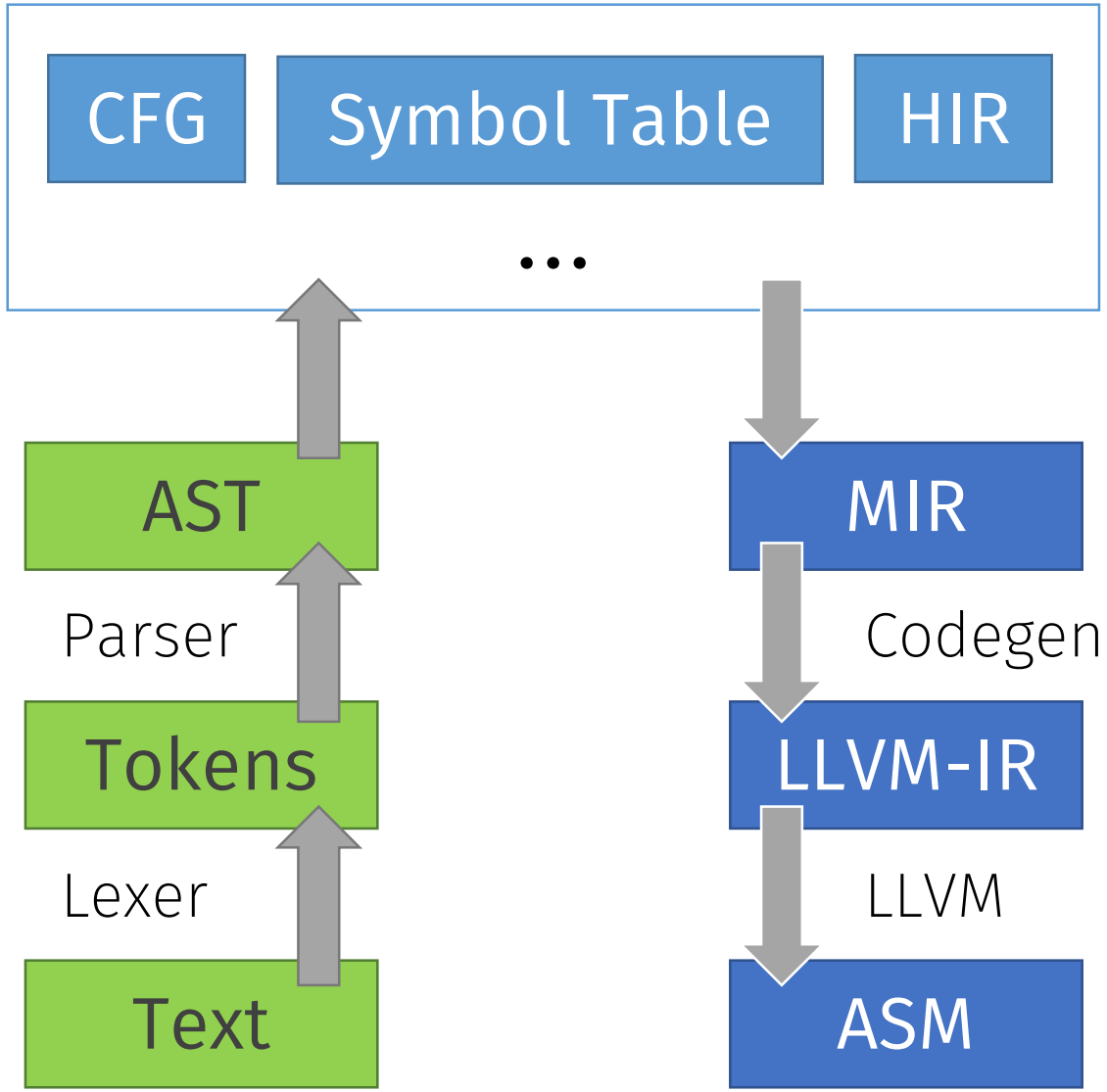
While-Schleife

```asm
main:
    mov   rax, 0          ; load 0 into rax
    mov   rbx, 0          ; load 0 into rbx
.start_loop:
    cmp   rax, 10         ; compare (sets ZF=1 if equal)
    je    .end            ; if ZF==1 jump!
    add   rbx, rax        ; rbx += rax
    add   rax, 1          ; rax += 1
    jmp   .start_loop     ; jump back up again

.end:
    nop                   ;-)
```

```rust
let mut i = 0;    // will be in rax
let mut sum = 0;  // will be in rbx
while i != 10 {
    sum += i;
    i += 1;
}
```

Von Rust zu Maschinencode



- **IR*: Intermediate Representation
- *LLVM-IR* ≈ CPU-unabhängiger Maschinencode
- *LLVM*
  - Von *LLVM-IR* zu Maschinencode
  - Unterstützt **viele** CPUs
  - Optimiert Code (!!!)
  - Wird auch von clang (C++) benutzt

Demo

Komische Instruktionen?

```
xor rax, rax
```

- Setzt **`rax`** auf 0 (diese Instruction ist kürzer als **`mov rax, 0`**)

```
lea rax, [eine Rechnung]
```

- Schreibt Ergebnis der Rechnung in **`rax`**
  - CPU kann gewisse, einfache Rechnungen so schneller ausführen, als mit mehreren **`mov`**, **`add`**, … Instruktionen

```
mul rdi
```

- Rechenoperationen mit einem Operanden nutzen meist **`rax`** als ersten Operanden

Mehr Speicher!

- Systemspeicher (RAM)
  - Deutliche langsamer als Register (Faktor ≈150, später mehr)
  - Deutlich größer (mehrere GB)

```
mov rax, qword ptr [0x1234]
```

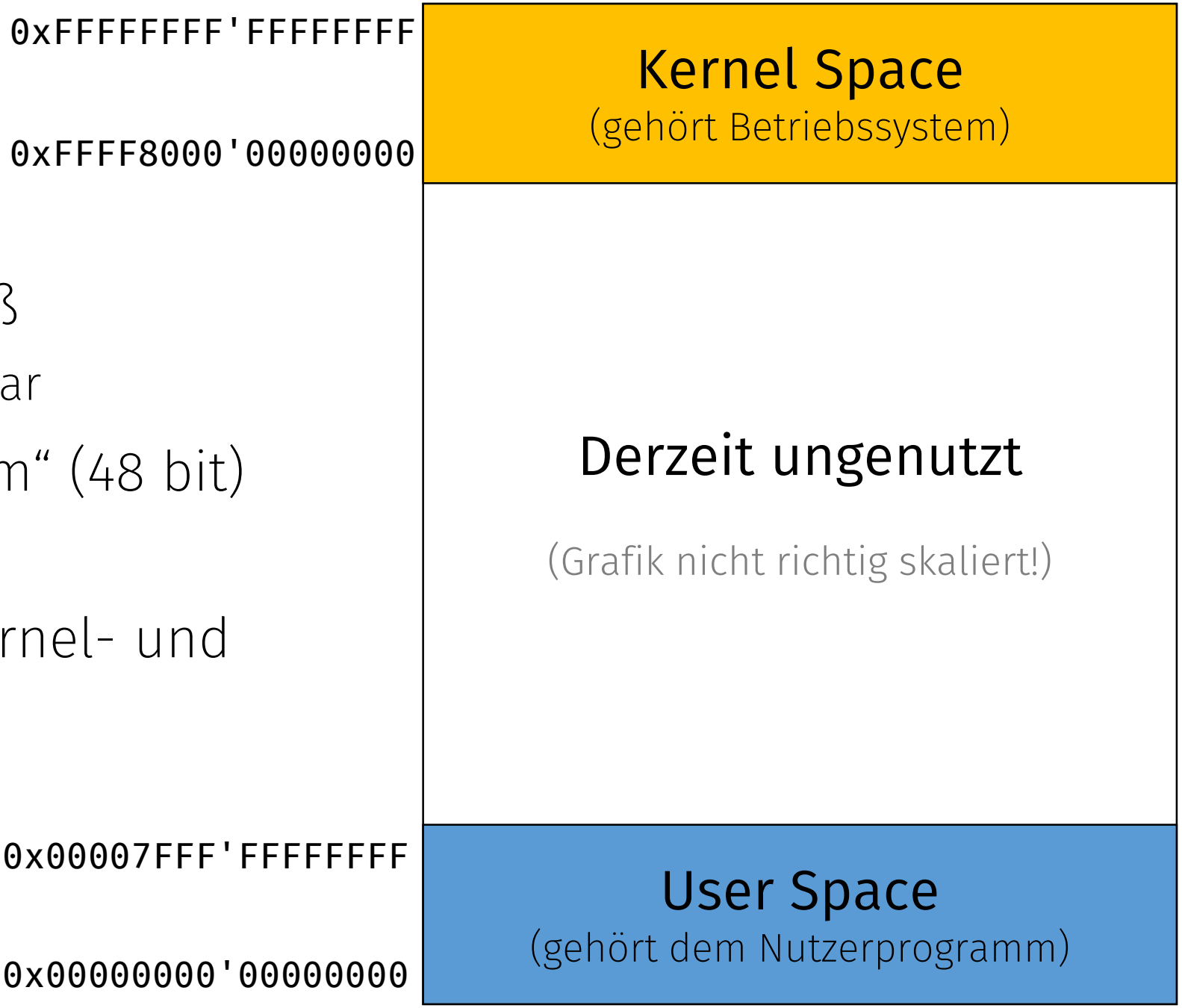
```
mov rax, qword ptr [7 + rbx + rcx * 4]
```

- Lädt Wert von der Adresse **`0x1234`** in **`rax`**
- Kompliziertere Adressberechnungen möglich
  - Mit gewissen Einschränkungen!

- Virtuelle Adressen: Wir sind der einzige Prozess!
  - Werden vom Prozessor (MMU) zu physikalischen Adressen umgewandelt

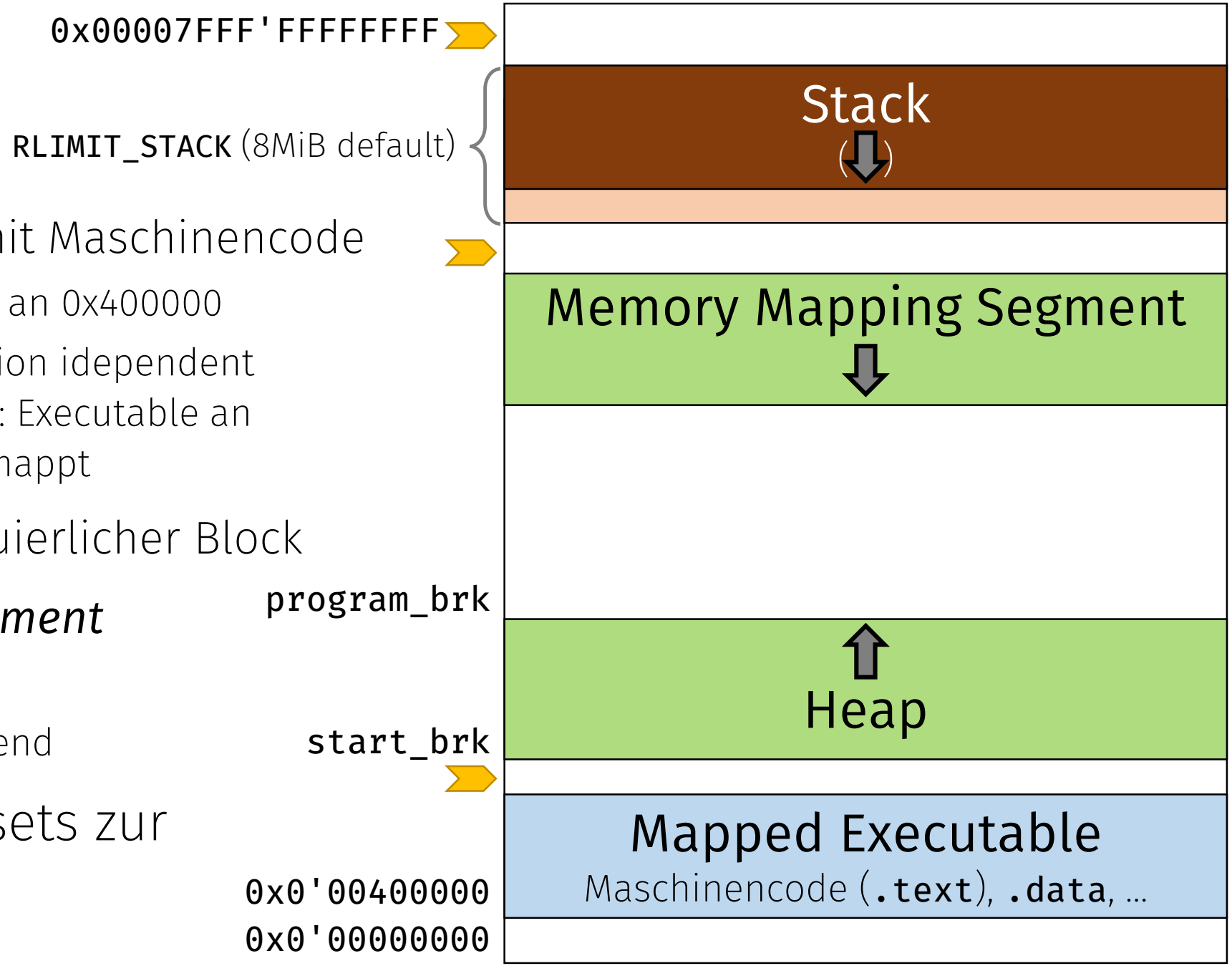
Adressen

- Theoretisch 64 Bit groß
  - 16 ExaBytes ansprechbar
- Zurzeit: „canonical form“ (48 bit)
  - 256 TB ansprechbar
- Unterteilt zwischen Kernel- und User-Space



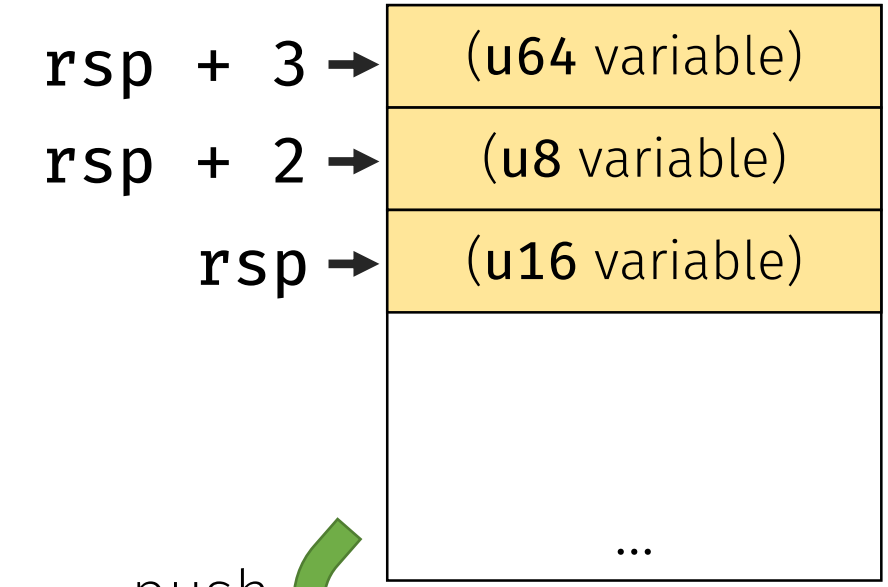
User Space

- *Mapped Executable* mit Maschinencode
  - Normalerweise immer an 0x400000
  - Rust nutzt –fPIE (position idependent executable) linker flag: Executable an zufällige Addresse gemappt
- *Heap & Stack*: Kontinuierlicher Block
- *Memory Mapping Segment*
  - Mehrere Blöcke
  - Nicht zusammenhängend
- *ASLR*: Zufällige Offsets zur Sicherheit (➡)



Stack

- LIFO Datenstruktur
  - Nur auf Einfügen/Löschen bezogen, Lesen überall möglich!
- Verwaltung durch Stackpointer
  - Zeigt auf neustes/unterstes Byte im Stack
  - Wird in Register **`rsp`** gespeichert




```
; push u64 on stack
sub rsp, 8
mov qword ptr [rsp], 42
```

=

```
; shortcut
push 42
```

```
mov rax, qword ptr [rsp]
add rsp, 8
```

=

```
; shortcut
pop rax
```

Heap

- Unterschied zwischen *Heap* und *Memory Mapped Area* für Programmierer selten wichtig
- Daher: Vereinfachung
- Heap := {*Heap*, *Memory Mapped Area*}
  - „Man kann Blöcke fester Größe vom Betriebssystem anfragen“

Stack	Heap
• Verwaltung *einfach*: nur ein Pointer • Hinzufügen/Löschen strikt nach LIFO • Lesen überall	• Hinzufügen/Löschen in beliebiger Reihenfolge • *Komplizierter*: Blöcke und Löcher verwalten

Funktionen und der Stack

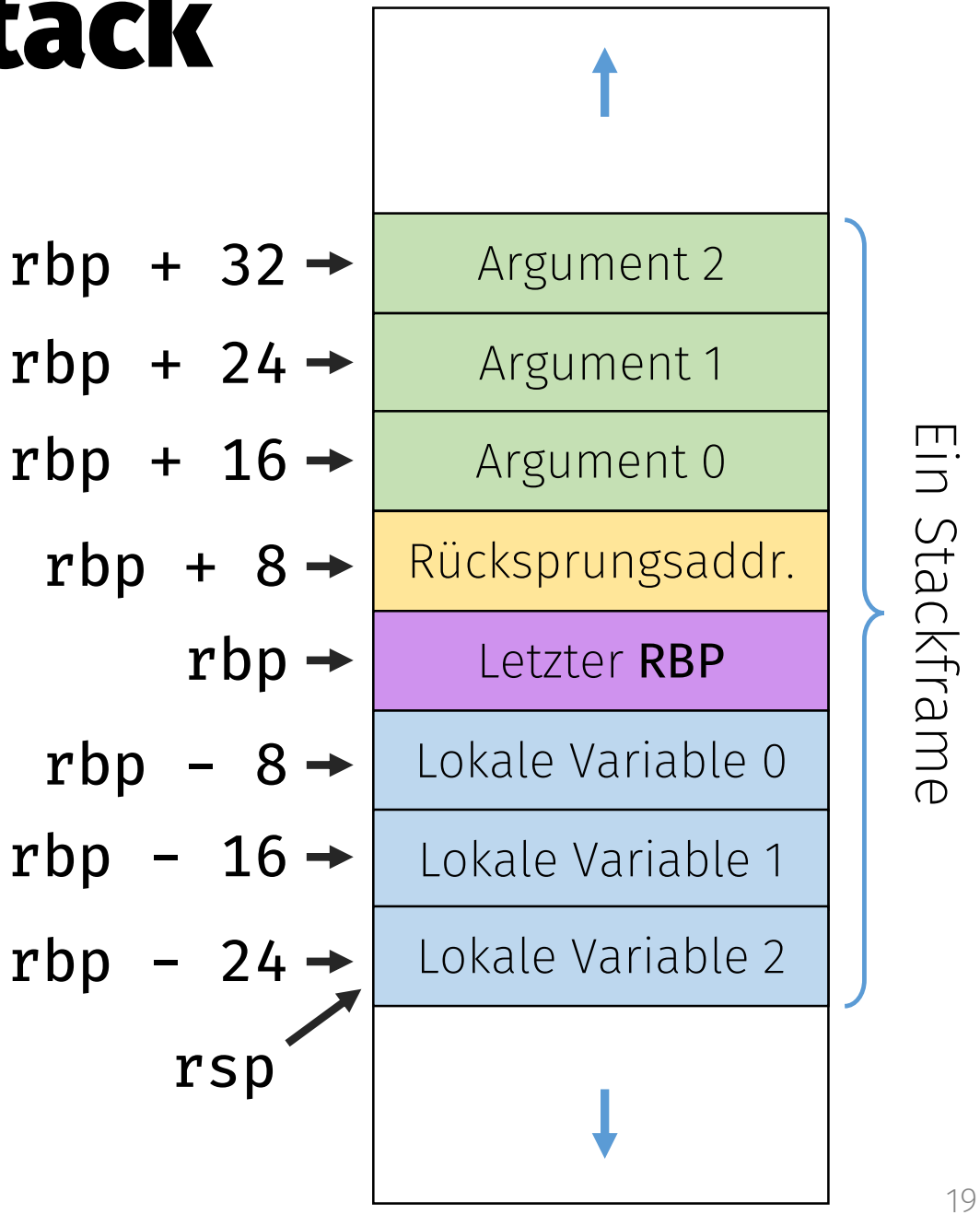
- Funktionen?
  - Idee: `jmp`
- Aber wo speichern wir …
  - … Funktionsargumente?
  - … lokale Variablen?
  - … Rückgabewert?
- Möglichkeit: Alles auf dem Heap
  - Nachteil: sehr langsam!
    - Mit Betriebssystem reden, Speicherfragmentierung, Cache Locality, …
- Stack funktioniert wunderbar!

Wir brauchen ein Protokoll (feste Regeln)!

➔ Calling Conventions

Funktionen und der Stack

- Jede Funktion besitzt sog. Stackframe
  - Argumente, lokale Variablen, …
- Register **`rbp`**: „Base Pointer“
  - Zugriff auf Argumente und Variable durch **`rbp`**
    - Beispiel: `mov rax, qword ptr [rbp - 8]`
  - Nicht unbedingt nötig, aber sinnvoll:
    - Identifizierung von Stackframes
    - Stack Traces
- Rücksprungsadresse nötig
  - Kann von mehreren „Eltern“ aufgerufen werden

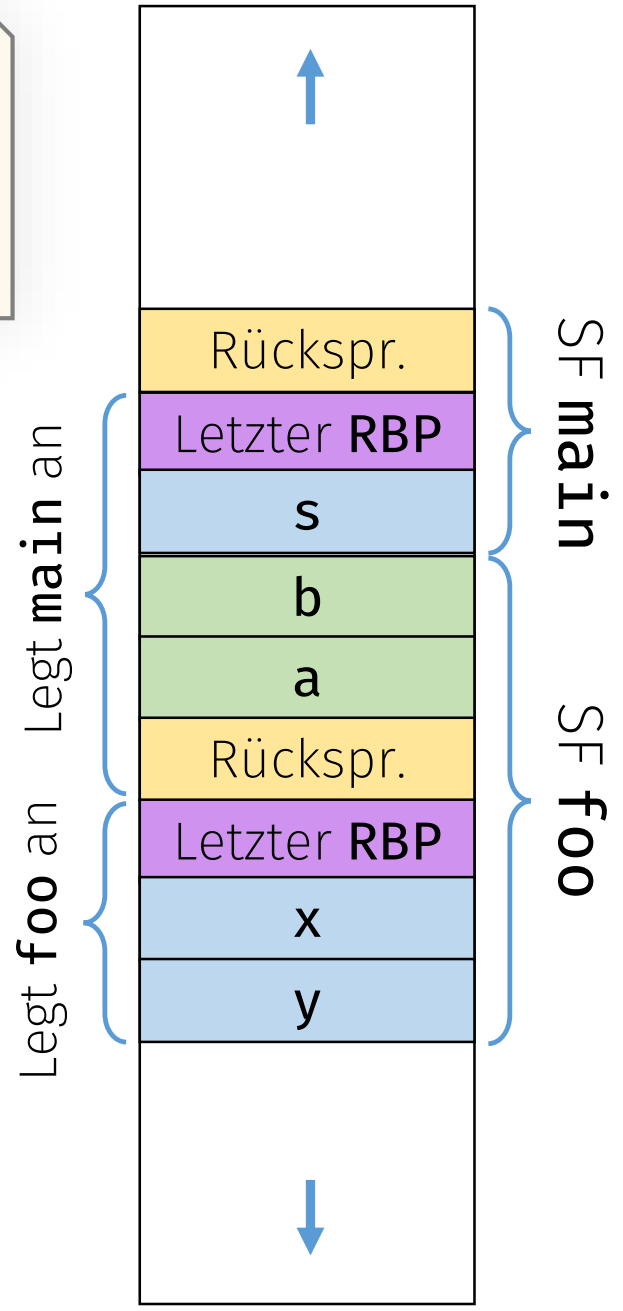


Beispiel

```rust
fn main() {
    let s = 5u64;
    foo(10, 11);
}
```

```rust
fn foo(a: u64, b: u64) {
    let x = a;
    let y = b;
}
```

- Obere Hälfte vom Stackframe:
  - Argumente und Rücksprungsaddresse
  - Wird von aufrufender Funktion angelegt und gefüllt
- Untere Hälfte vom Stackframe:
  - Alter **`rbp`** und lokale Variablen
  - Wird von Funktion selber angelegt und gefüllt

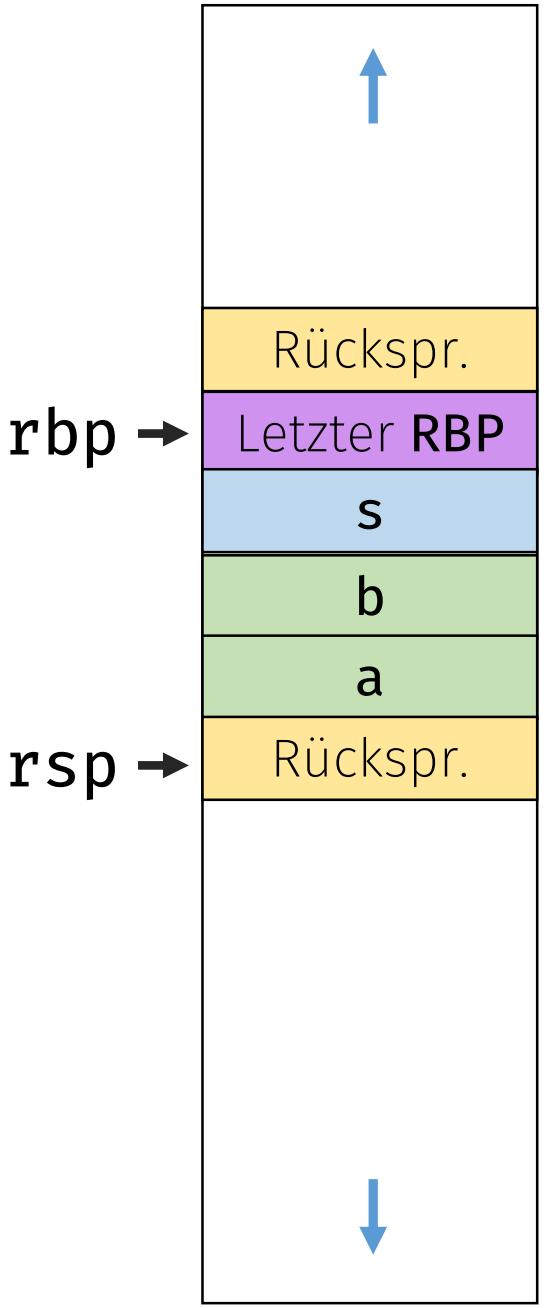


Beispiel

```rust
fn main() {
    let s = 5u64;
    foo(10, 11);
}
```

```rust
fn foo(a: u64, b: u64) {
    let x = a;
    let y = b;
}
```

```
foo:
    push rbp            ; put old rbp on stack
```

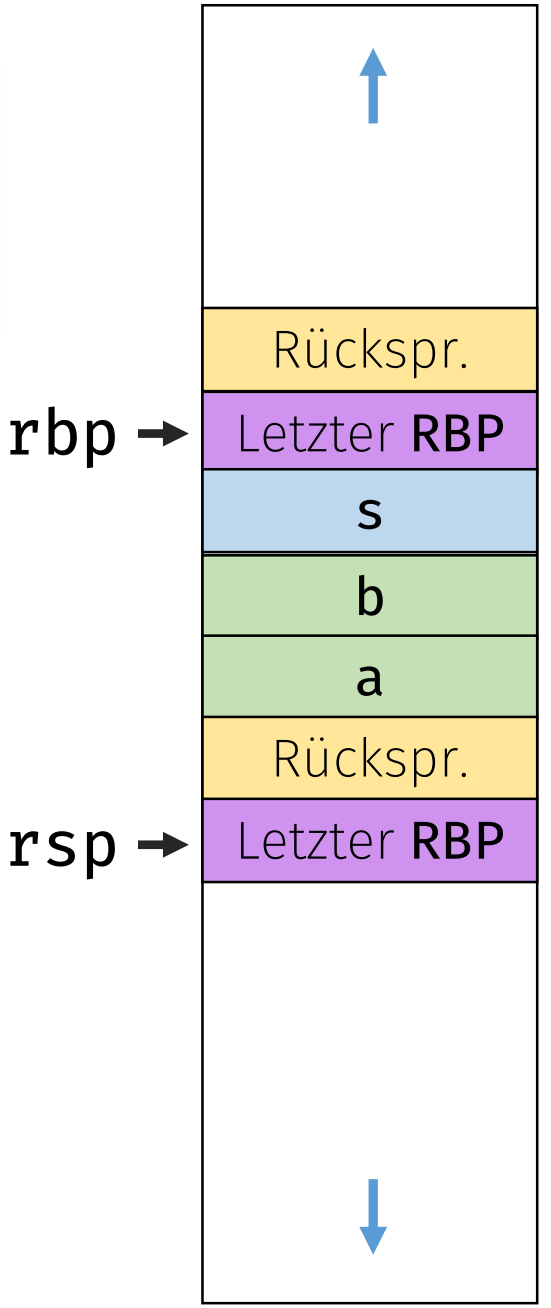


Beispiel

```rust
fn main() {
    let s = 5u64;
    foo(10, 11);
}
```

```rust
fn foo(a: u64, b: u64) {
    let x = a;
    let y = b;
}
```

```
foo:
    push rbp            ; put old rbp on stack
    mov  rbp, rsp       ; new rbp is where rsp points now
```

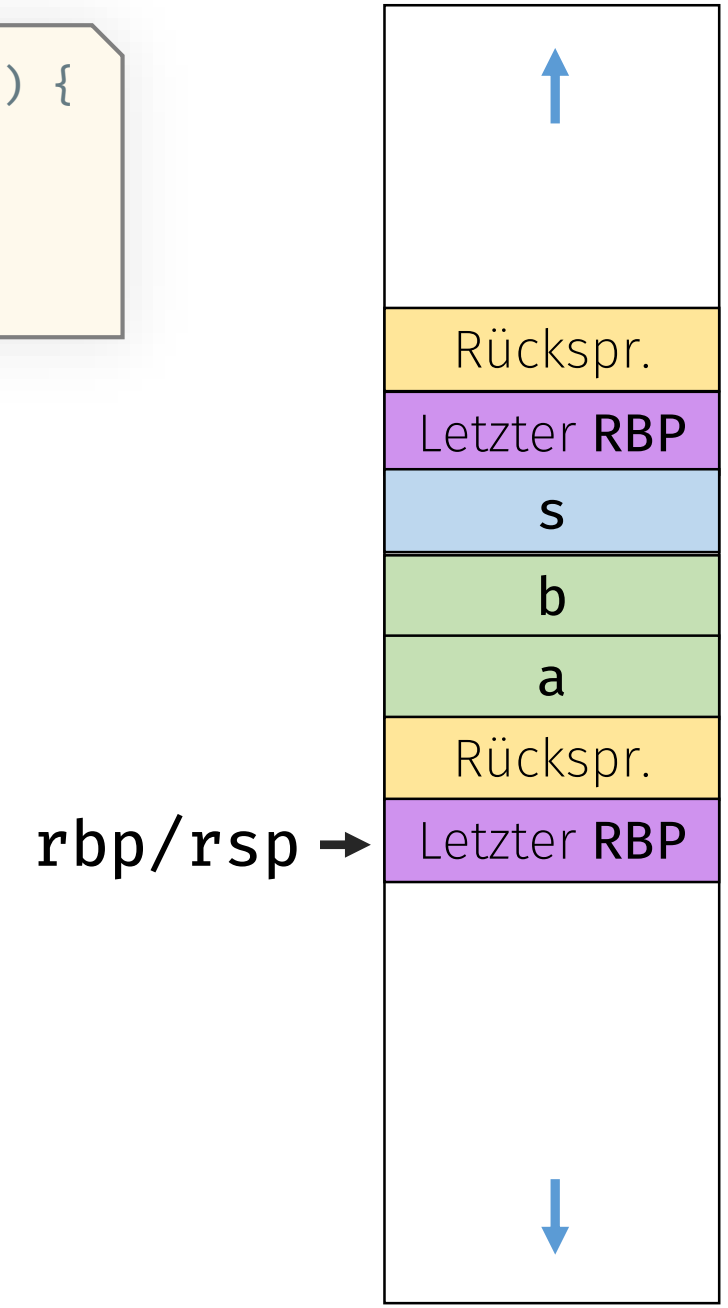


Beispiel

```rust
fn main() {
    let s = 5u64;
    foo(10, 11);
}
```

```rust
fn foo(a: u64, b: u64) {
    let x = a;
    let y = b;
}
```

```
foo:
    push rbp            ; put old rbp on stack
    mov  rbp, rsp       ; new rbp is where rsp points now
    sub  rsp, 16        ; We need 16 bytes for local vars
```

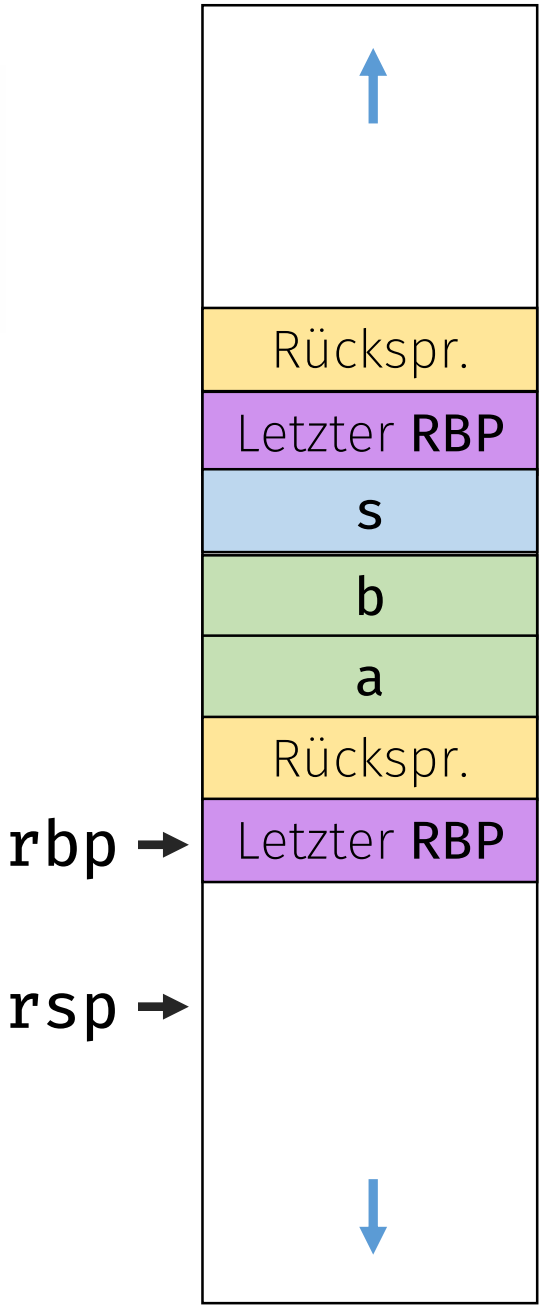


Beispiel

```rust
fn main() {
    let s = 5u64;
    foo(10, 11);
}
```

```rust
fn foo(a: u64, b: u64) {
    let x = a;
    let y = b;
}
```

```
foo:
    push rbp            ; put old rbp on stack
    mov  rbp, rsp       ; new rbp is where rsp points now
    sub  rsp, 16        ; We need 16 bytes for local vars
```



Beispiel

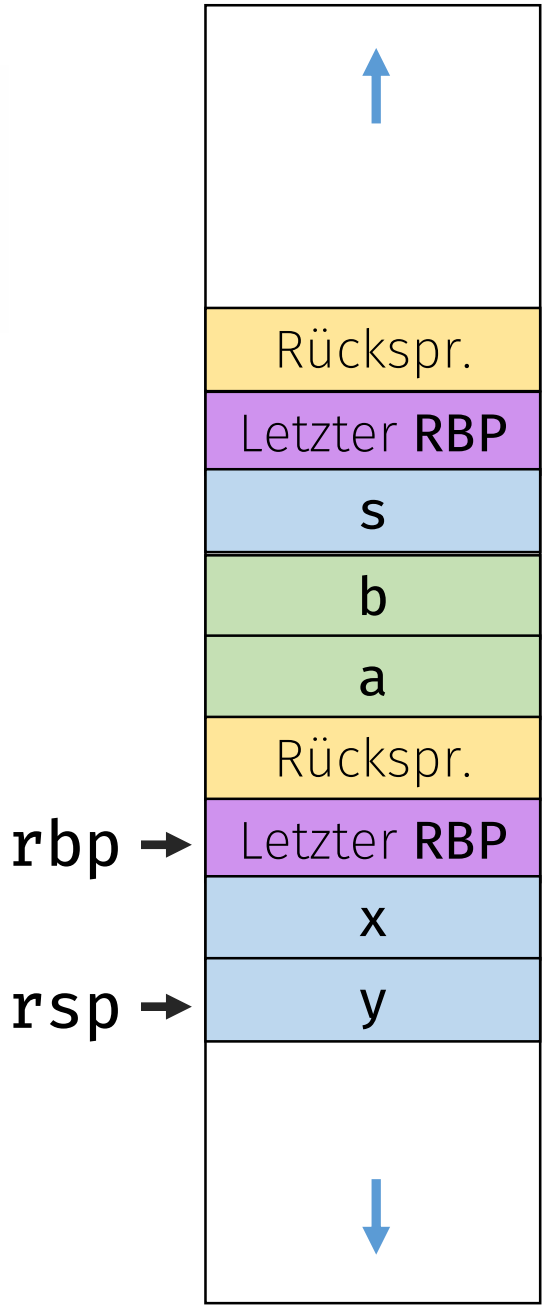
```rust
fn main() {
    let s = 5u64;
    foo(10, 11);
}
```

```rust
fn foo(a: u64, b: u64) {
    let x = a;
    let y = b;
}
```

```
foo:
    push rbp            ; put old rbp on stack
    mov  rbp, rsp       ; new rbp is where rsp points now
    sub  rsp, 16        ; We need 16 bytes for local vars

    ; copy from arguments to local variables
    mov  qword ptr [rbp - 8], qword ptr [rbp + 16]
    mov  qword ptr [rbp - 16], qword ptr [rbp + 24]

    add rsp, 16             ; throw away local variables
```



Beispiel

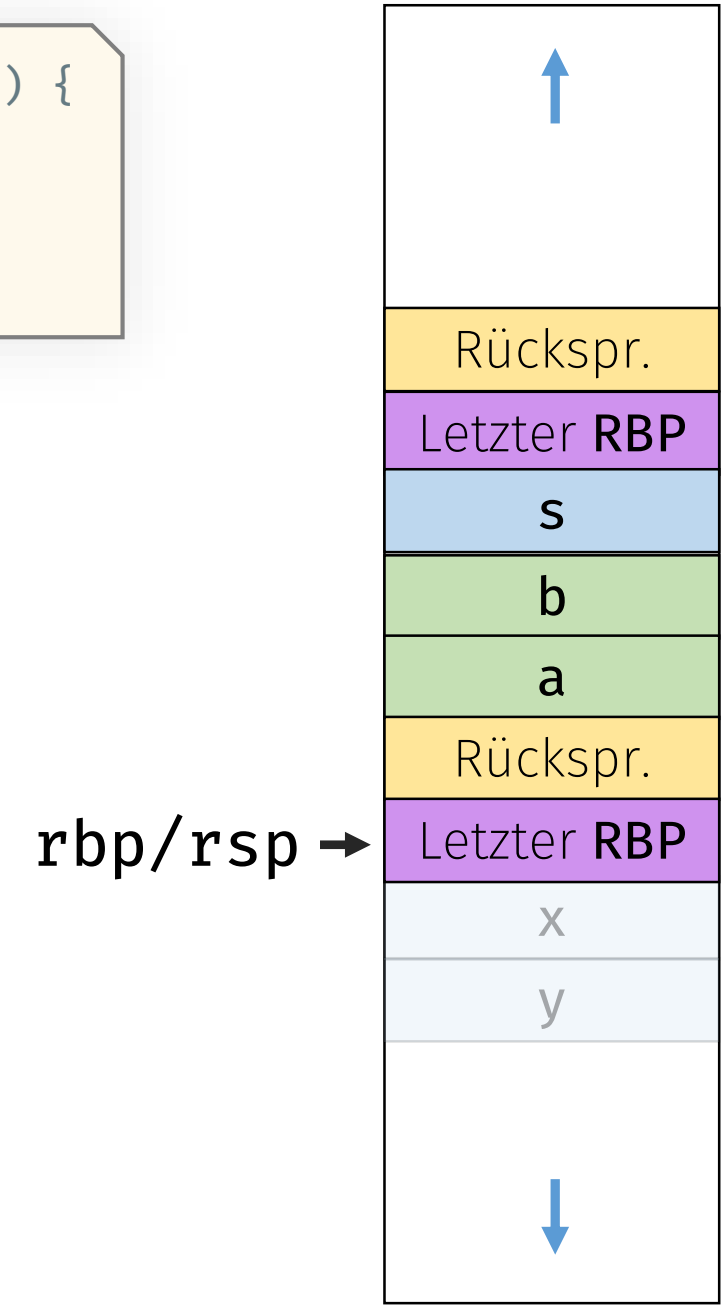
```rust
fn main() {
    let s = 5u64;
    foo(10, 11);
}
```

```rust
fn foo(a: u64, b: u64) {
    let x = a;
    let y = b;
}
```

```
foo:
    push rbp            ; put old rbp on stack
    mov  rbp, rsp       ; new rbp is where rsp points now
    sub  rsp, 16        ; We need 16 bytes for local vars

    ; copy from arguments to local variables
    mov  qword ptr [rbp - 8], qword ptr [rbp + 16]
    mov  qword ptr [rbp - 16], qword ptr [rbp + 24]

    add rsp, 16             ; throw away local variables
    pop rbp                 ; restore rbp
```



Beispiel

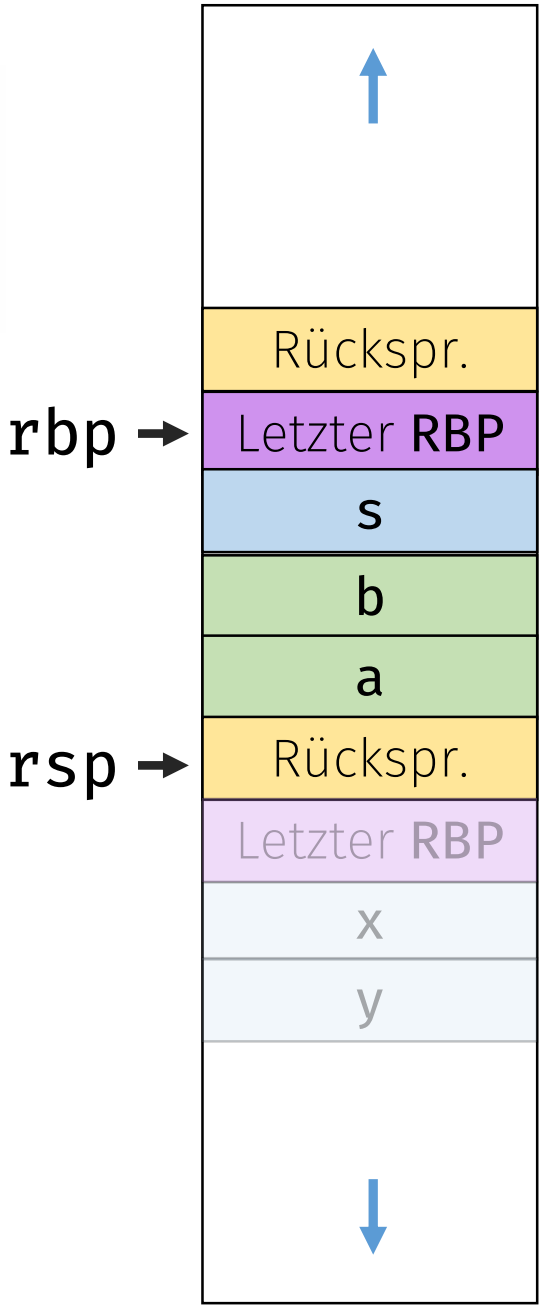
```rust
fn main() {
    let s = 5u64;
    foo(10, 11);
}
```

```rust
fn foo(a: u64, b: u64) {
    let x = a;
    let y = b;
}
```

```
foo:
    push rbp            ; put old rbp on stack
    mov  rbp, rsp       ; new rbp is where rsp points now
    sub  rsp, 16        ; We need 16 bytes for local vars

    ; copy from arguments to local variables
    mov  qword ptr [rbp - 8], qword ptr [rbp + 16]
    mov  qword ptr [rbp - 16], qword ptr [rbp + 24]

    add rsp, 16             ; throw away local variables
    pop rbp                 ; restore rbp
```



Beispiel

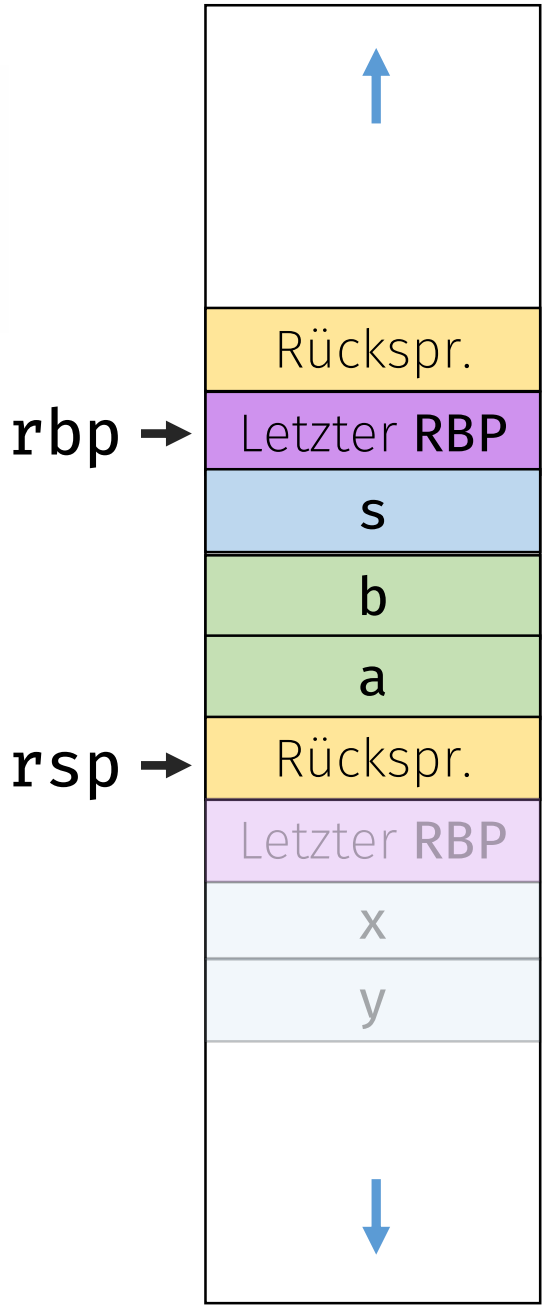
```rust
fn main() {
    let s = 5u64;
    foo(10, 11);
}
```

```rust
fn foo(a: u64, b: u64) {
    let x = a;
    let y = b;
}
```

```
foo:
    push rbp            ; put old rbp on stack
    mov  rbp, rsp       ; new rbp is where rsp points now
    sub  rsp, 16        ; We need 16 bytes for local vars

    ; copy from arguments to local variables
    mov  qword ptr [rbp - 8], qword ptr [rbp + 16]
    mov  qword ptr [rbp - 16], qword ptr [rbp + 24]

    add rsp, 16             ; throw away local variables
    pop rbp                 ; restore rbp
    pop rcx                 ; return address on top of stack
    jmp rcx                 ; jump to return address
```



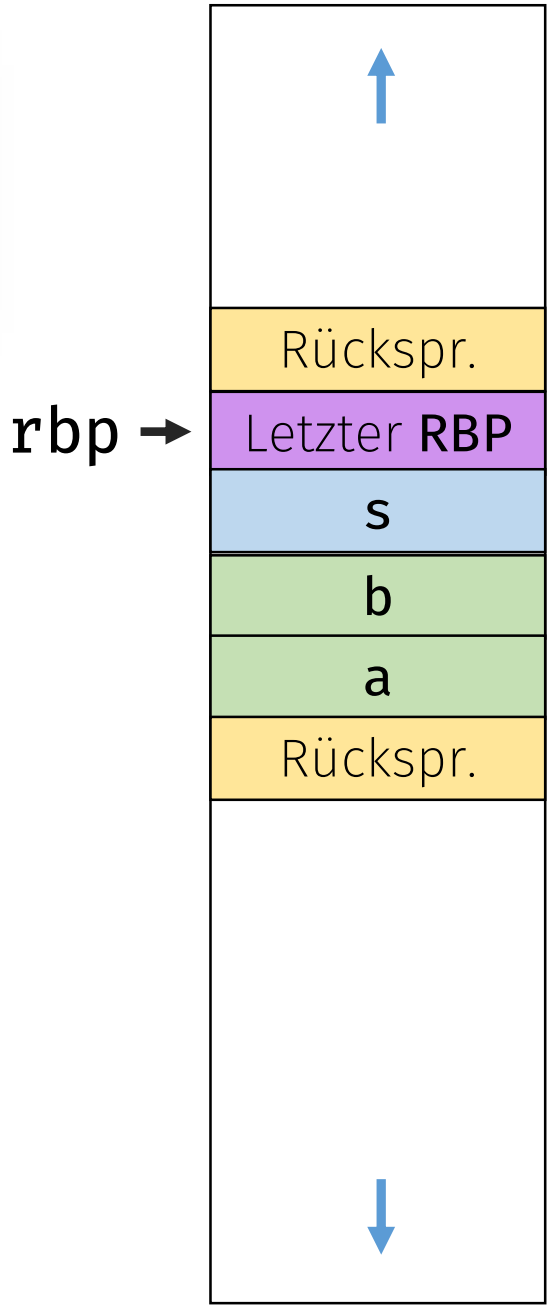
Beispiel

```rust
fn main() {
    let s = 5u64;
    foo(10, 11);
}
```

```rust
fn foo(a: u64, b: u64) {
    let x = a;
    let y = b;
}
```

```
main:
    push rbp            ; standard entry procedure
    mov  rbp, rsp
    sub  rsp, 8         ; 8 bytes local variables
    mov  qword ptr [rpb - 8], 5     ; init variable s

    ; prepare function call
    sub  rsp, 16                    ; 24 bytes for arguments
    mov  qword ptr [rsp], 10        ; argument a
    mov  qword ptr [rsp - 8], 11    ; argument b
    push .after_call                ; pushes address of code
    jmp  foo
.after_call:
```



Call und Ret Instructions

```
push .after_call
jmp  foo
```

=

```
call foo
```

```
pop rcx
jmp rcx
```

=

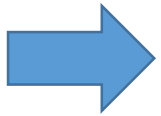
```
ret
```

- Vereinfacht das Aufrufen/Verlassen von Funktionen
- Kümmert sich um Rücksprungsadresse

Rückgabewert und Optimierungen

- Rückgabewert: In **`rax`**
- Bei größeren Rückgabewerten: Mit „out pointer“

```
fn foo() -> BigStruct {
    build_big()
}
```



```
fn foo(out: &mut BigStruct) {
    *out = build_big();
}
```

- Optimierung in der x86_64 calling convention:
  - Ersten sechs Argumente werden in Registern übergeben
  - Redzone: Bei Blattfunktionen **`rsp`** anpassen manchmal nicht nötig

1.	**`rdi`**
2.	**`rsi`**
3.	**`rdx`**
4.	**`rcx`**
5.	**`r8`**
6.	**`r9`**

Limitationen Stack

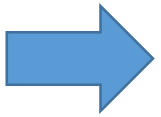
- Größe vom Stackframe ist pro Funktion konstant!
  - Jede Variable hat eine feste Position relativ to **`rbp`**
- Keine *unsized types* auf dem Stack!
  - Beispiel: Wachsende Arrays
  - Verboten als lokale Variable, Funktionargument oder -rückgabewert
- Alle *unsized types* auf Heap gespeichert
  - „werden geboxt“

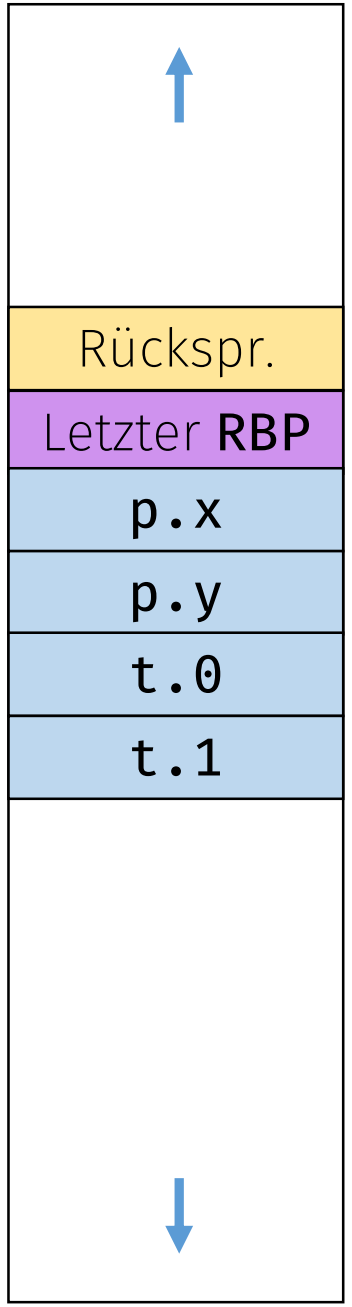
Speicherlayout in Rust

- Struct, Tuple, …: Alle Felder (fast) hintereinander

```rust
struct Point2 {
    x: u64,
    y: u64,
}
```

```rust
fn main() {
    let p = Point2 { … };
    let t = (3u64, 4u64);
}
```





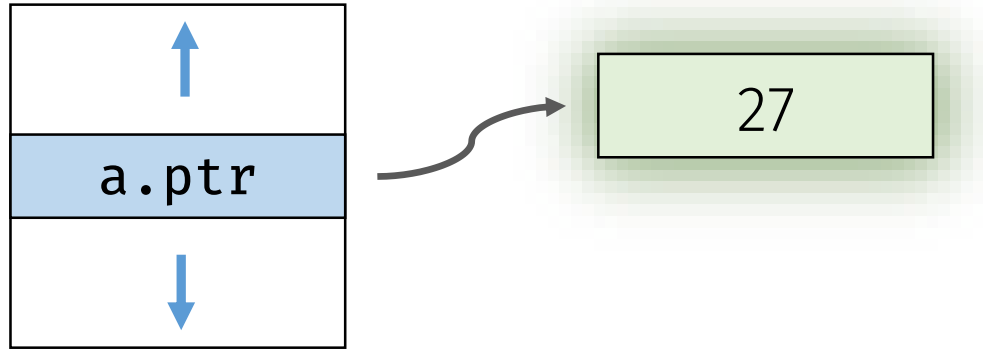
- Padding möglich: Auffüllen mit ungenutzen Bytes
  - Bestimmte Werte möchten *aligned* sein
  - **u64** möchten nur an Addressen liegen die mod 8 == 0
- Enum: Etwas komplizierter, aber auch alles auf Stack

Heap muss explizit genutzt werden!

Heap nutzen

- `malloc()`: Blöcke fester Größe vom Betriebssystem anfragen
- `free()`: Block wieder freigeben
- `malloc()` und `free()` sind C-Library Funktionen, keine Syscalls
- Typische Speicherfehler:
  - *Memory Leak:* Kein `free()` nach `malloc()`
  - *Double free*: Zwei mal `free()` des selben Blockes
  - *Use after free*
- RAII Pattern um diese Fehler zu verhindern

Heap in Rust (Box)



- Schon benutzt?
  - `Vec<T>`, `String`, `HashMap`, …
- Einen Wert auf dem Heap: `Box<T>`

```
fn main() {
    let a = Box::new(27u64);
}
```

```
// Implements Deref and
// DerefMut
let v = *a;  // : u64
```

- Nutzt *RAII* (später mehr):
  - Gibt Speicher am Ende des Scopes wieder frei
  - „Owned Pointer“ (entspricht C++‘s `unique_ptr`)

Speicherlayout

- Direkte, rekursive Typen in Rust unmöglich
  - Mit **`Box`** möglich
- In *Java*: Nicht-Primitive Typen (z.B. Klassen) automatisch auf dem Heap
  - Auch automatisch null-able

Java

```
class Tree {
    Tree left;
    Tree right;
}
```

```
struct Tree {
    left: Option<Tree>,
    right: Option<Tree>,
}
```

```
error[E0072]: recursive type `Tree` has infinite size
  --> <anon>:7:1
   |
7  | struct Tree {
   | ^ recursive type has infinite size
   |
   = help: insert indirection (e.g., a `Box`, `Rc`, or
`&`) at some point to make `Tree` representable
```

```
struct Tree {
    left: Option<Box<Tree>>,
    right: Option<Box<Tree>>,
}
```

Warum Heap nutzen?

- Strukturen, die zur Laufzeit ihre Größe ändern
  - `Vec<T>`, `String`, `HashMap`, …
- Indirektion: z.B. rekursive Datenstrukturen
- *Owned unsized types* (insb. *Trait Objects*, später mehr)
- Scope/Lebenszeit unabhängig von Stackframes
- Selten: Extrem große Typen (Struct mit Tausenden Feldern)

Standardmäßig Stack nutzen,
wenn benötigt Heap.

Zusammenfassung

- CPU führt primitive Befehle (Instruktionen) nacheinander aus
- Register: Schnelle kleine Speicher direkt in der CPU
  - Rechnungen passieren immer in Registern (erst aus Speicher laden)
- Grob: Zwei Arten von Speicher für unsere Variablen
  - *Stack*: LIFO Lebenszeit, sehr schnell, für lokale Variablen und Argumente
  - *Heap*: Freie Lebenszeiten, langsamer, feste Blöcke vom Betriebssystem erfragen
- Compiler muss Größe jedes Stackframes kennen
- Heap-Nutzung in Rust immer explizit