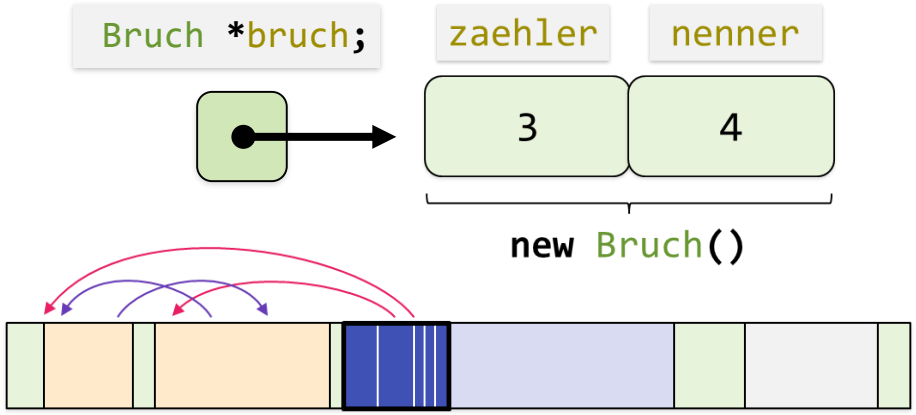
Einführung in die Softwareentwicklung

SOMMERSEMESTER 2020

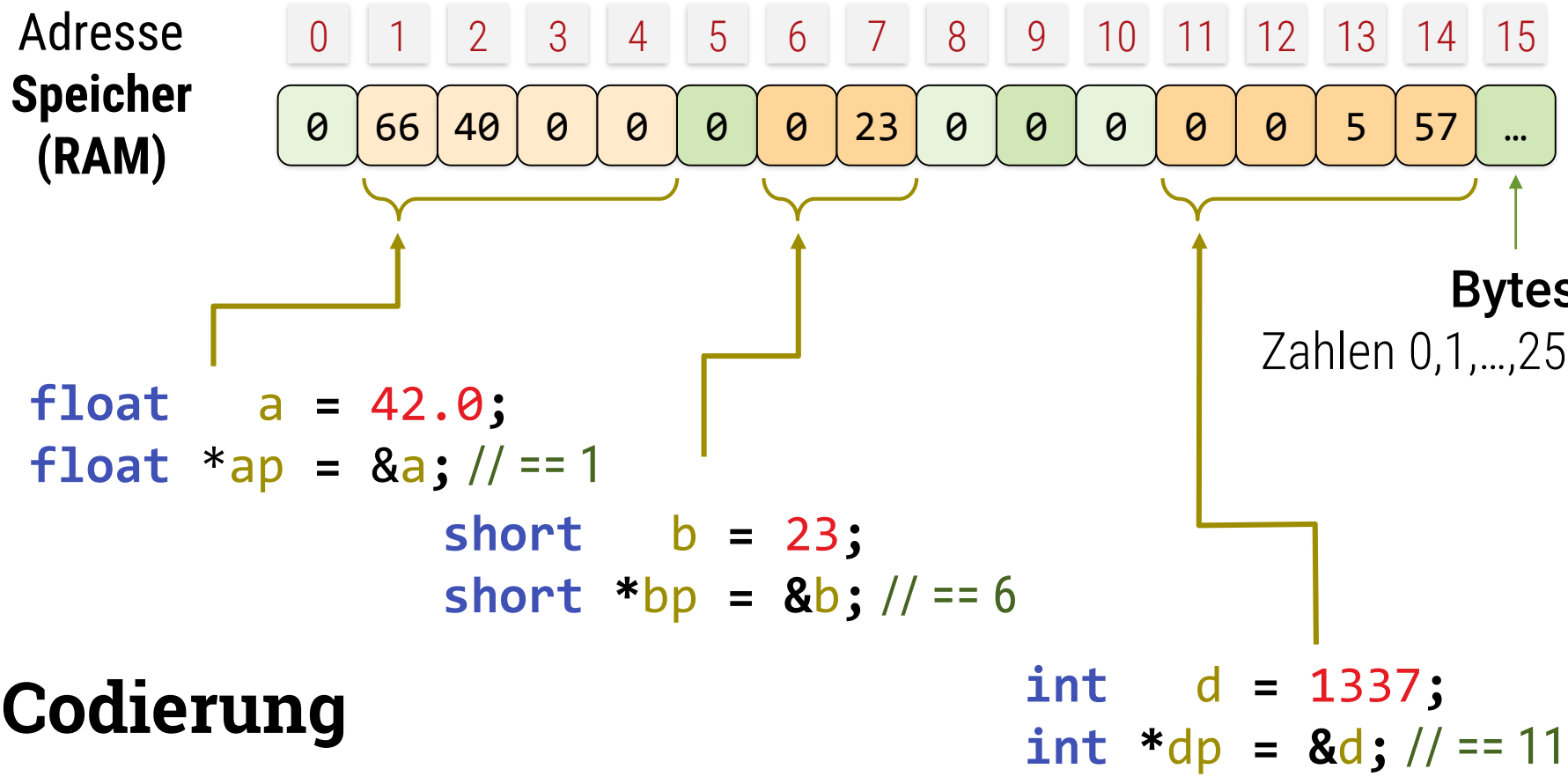


Foliensatz #05

Hardwarenahe Programmierung

Michael Wand · Institut für Informatik · michael.wand@uni-mainz.de

Maschinenrepräsentation



Codierung

- Aufeinanderfolgende Bytes
- Zeiger = Zeiger auf erstes Byte (Länge implizit)

Hardwarenahe Programmierung

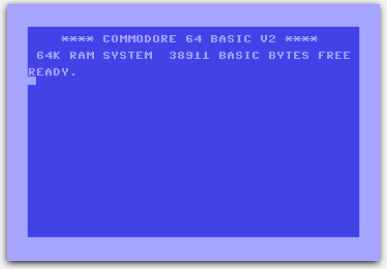
Hardware ansprechen

Treiber / BS Programmierung

- Zugriff auf feste Adressen
- z.B. Kontrollregister (Memory-Mapped I/O)
- Möglich über konstante Pointer
  - Cast von (`long`) `int` auf Pointer-Typ

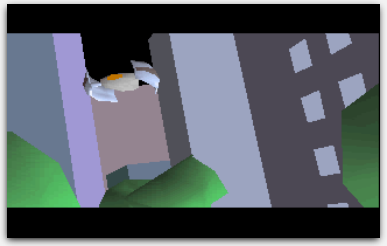
Hardware ansprechen

```
// Beispiel: Hintergrundfarbe auf C64
// (Memory-mapped „MOS/VIC 6569“ Videocontroller)
// Programm läuft nur auf einem Comodore C64

uint8_t *border     = (uint8_t*)53280; // Rahmenfarbe
uint8_t *background = (uint8_t*)53281; // Hintergrundfarbe

*border = (uint8_t*)14;     // Blau
*background = (uint8_t*)6; // Hellblau
```



[C64 Boot Screen]

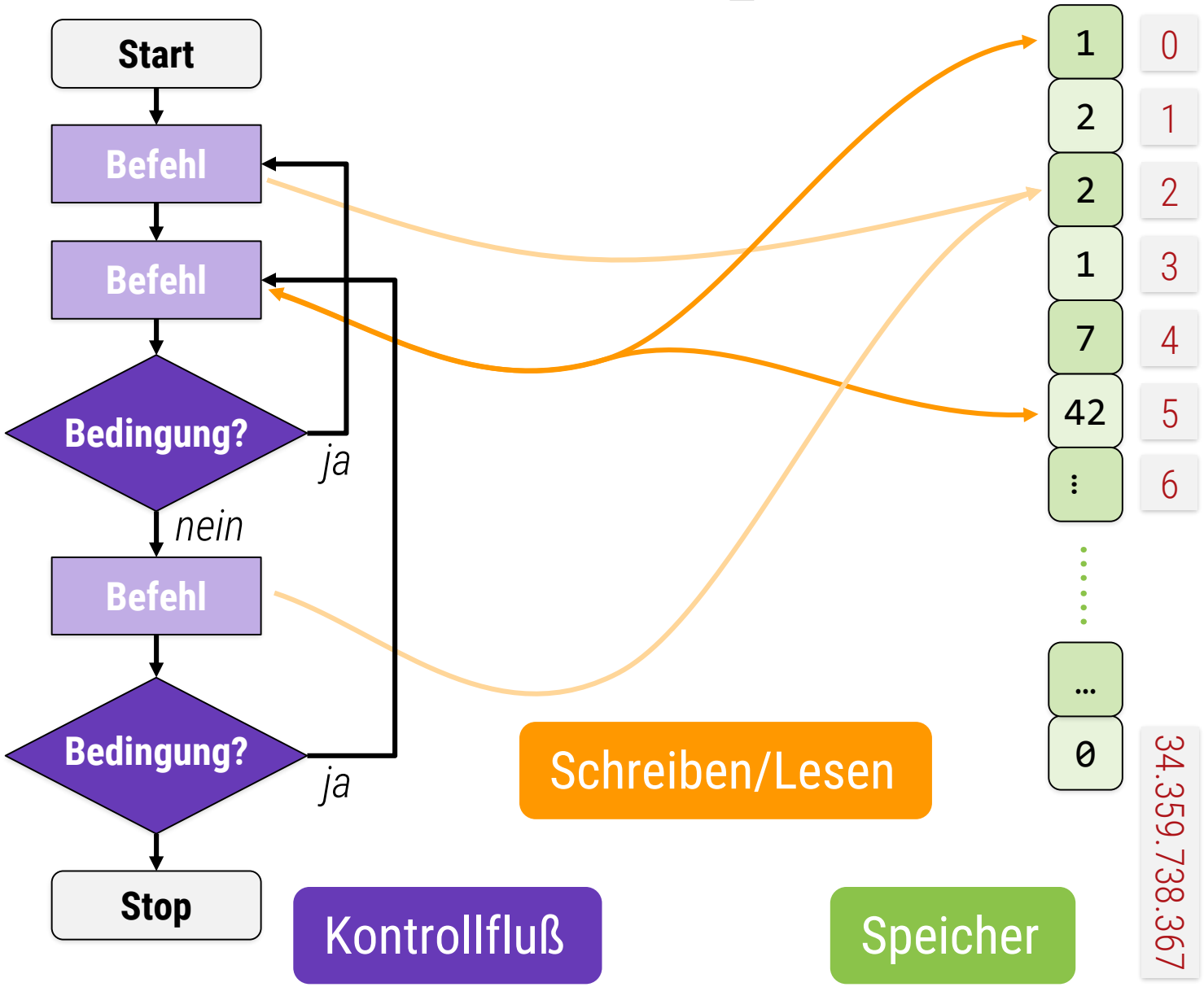
```
// Beispiel: VGA Mode 0x13 320x200
// Erfordert IBM-PC kompatiblen 32bit Rechner mit VGA-Graphik

uint8_t *videoMem = (uint8_t*)0xA0000; // Hexadezimal ≙ 655.360 dezimal

for (int y=0; y<200; y++) {
   for (int x=0; x<320; x++) {
      videoMem[x+320*y] = 0; // Bildschirm löschen
   }
}
```



[FutureCrew/2nd Reality, 1993]

Übersetzung in Maschinensprache

Maschinensprache

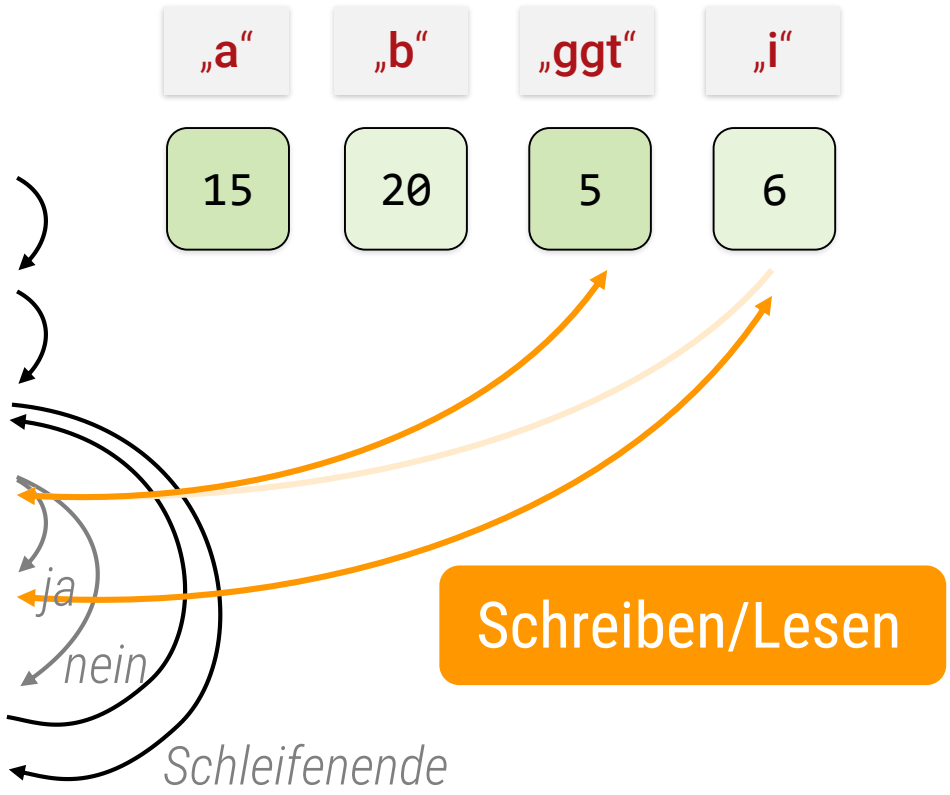


„Strukturierte Programmierung“

Kontrollfluß

Speicher

```
Eingabe: a,b ∈ ℕ
Ausgabe: ggt ∈ ℕ
Setze ggt = 1
Setze i = 2
Solange i ≤ a, b Wiederhole
    Falls i teilt a und b Dann
        Setze ggt = i
    Setze i = i + 1
```



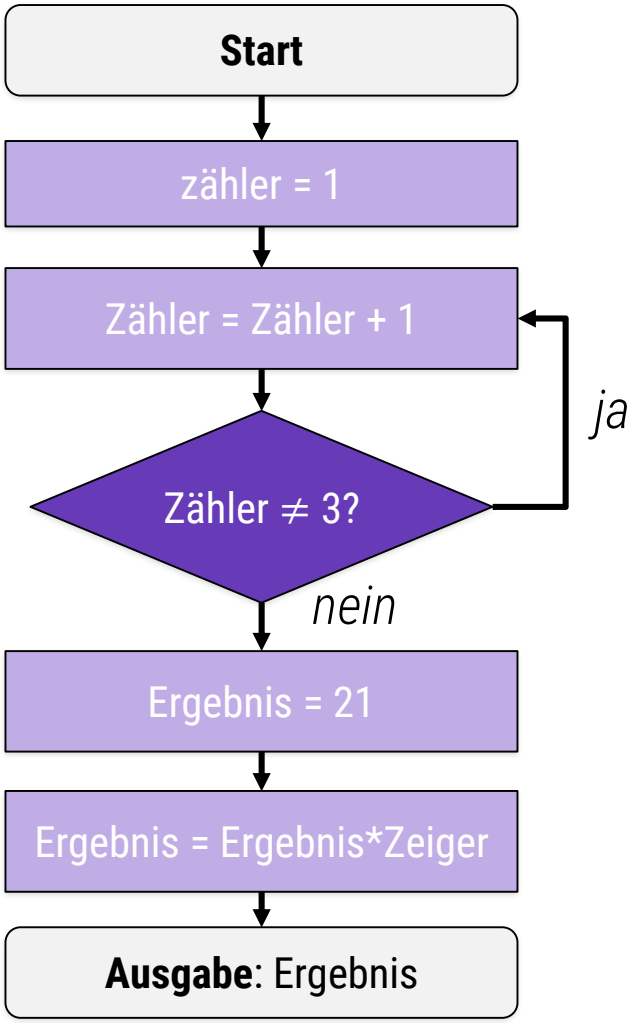
Beispiel

C++ Code

Java – Vereinfacht

```
// Einfacher
int zaehler = 0;
do {
    zaehler = zaehler + 1
while (zaehler != 7);
int ergebnis = 6;
ergebnis = ergbnis * zähler;
cout << ergebnis;
```

Flußdiagramm



Beispiel

Code

Symbolisches Machinenprogramm

```
ANFANG:
   add       (eins, zähler) → zähler
   cmp       (zähler, maximum) → fortsetzen
   cond_jmp   fortsetzen    → ANFANG
   mult      (ergebnis, zähler) → ergebnis
   out        ergebnis
```

„Echtes“ Machinenprogramm (nach Linken)

```
0 add       (0,1) → 1
1 cmp       (1,2) → 3
2 cond_jmp    3   → 0
3 mult      (4,1) → 4
4 out         4
```

Variablen (initialisiert)

Das Betriebsystem lädt Code & Daten später an andere Addressen

(Adressen im Code werden dabei entsprechend geändert)

Prinzip der automatischen Übersetzung

Hochsprachen

Algorithmus

```
Eingabe: Zahl a ∈ ℕ
Ausgabe: Zahl b ∈ ℕ

Setze Zähler i auf 1
Setze Summe s auf 0
Solange a ≥ 0 wiederhole
   Setze s auf s + i
Setze Ausgabe b auf s
```

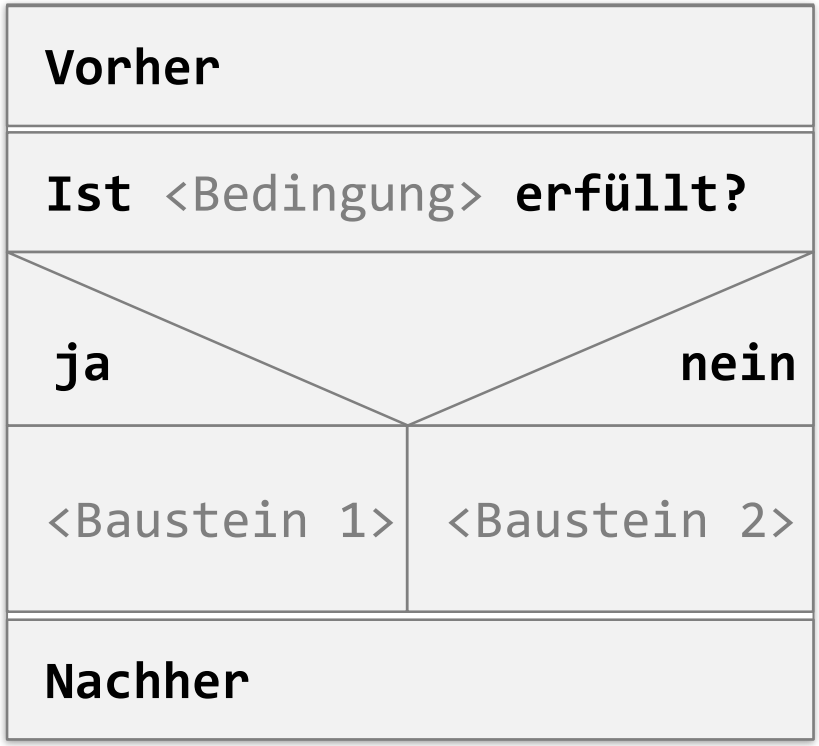
Maschinenprogramm

```
0 add       (0,1) → 1
1 cmp       (1,2) → 3
2 cond_jmp    3   → 0
4 ind_load    4   → 5
5 mult      (5,1) → 5
6 sub       (0,1) → 1
7 ...
```

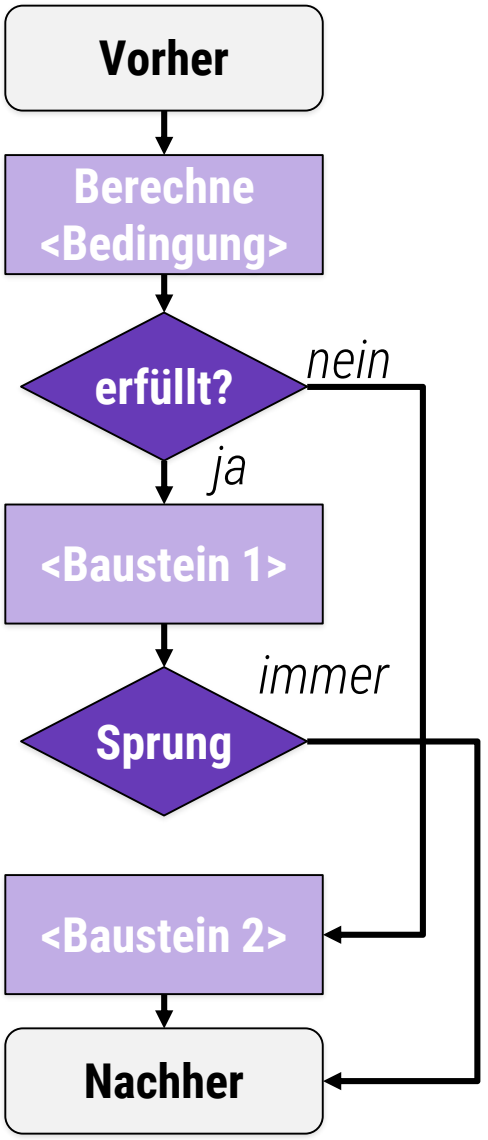
Automatische Übersetzung!

- Welche Befehle? → JAVA

Kontrollfluß

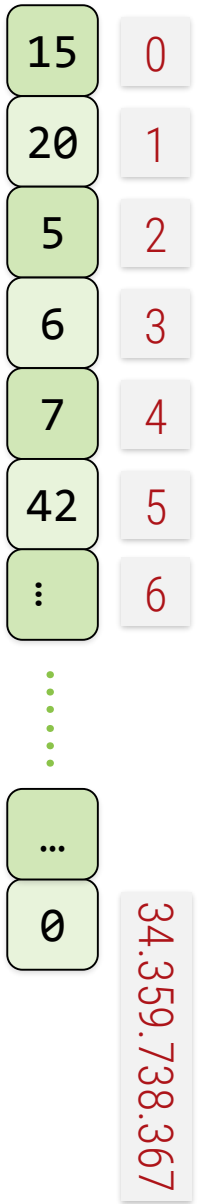


Übersetzung

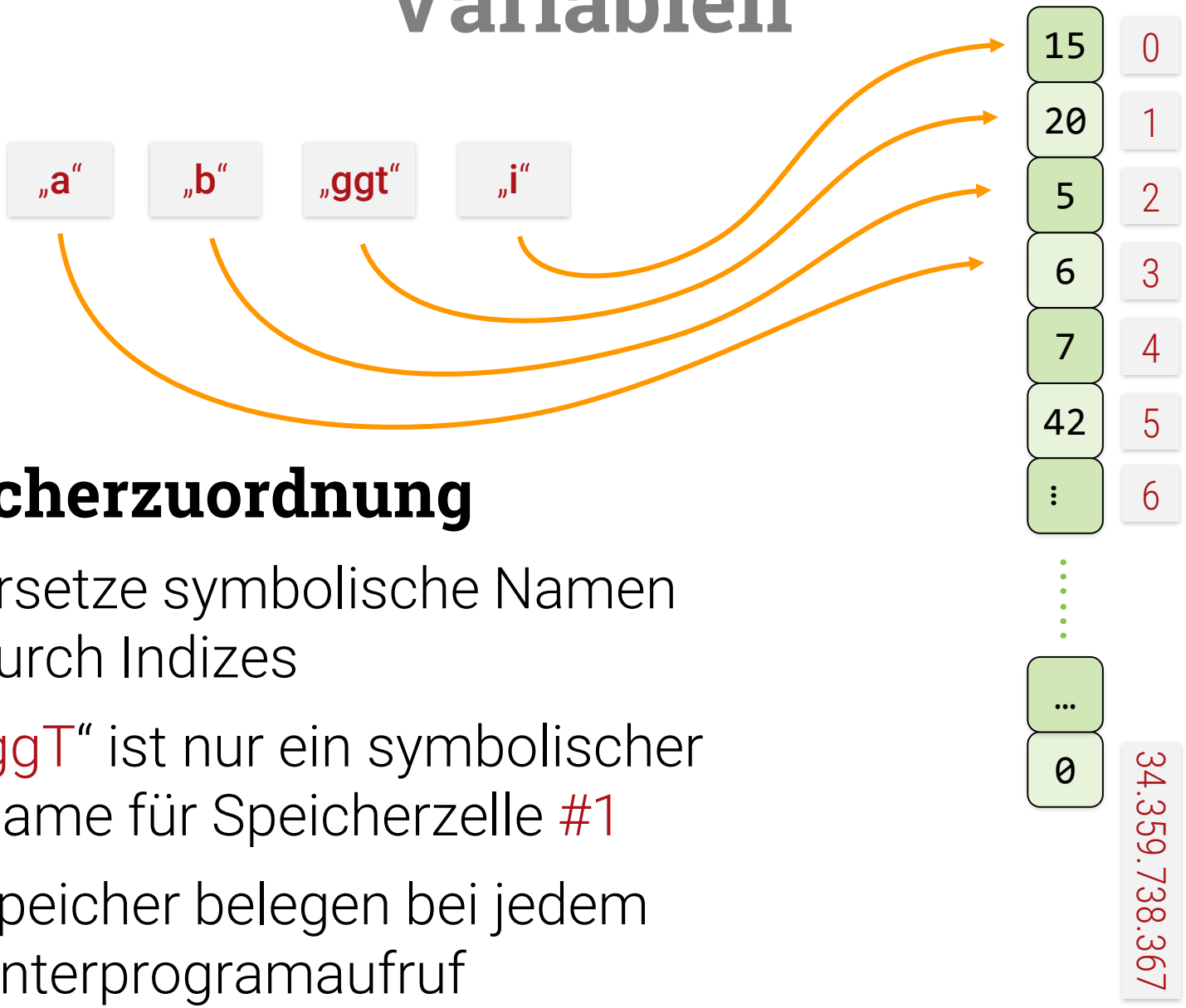


Variablen

„a“	„b“	„ggt“	„i“
15	20	5	6



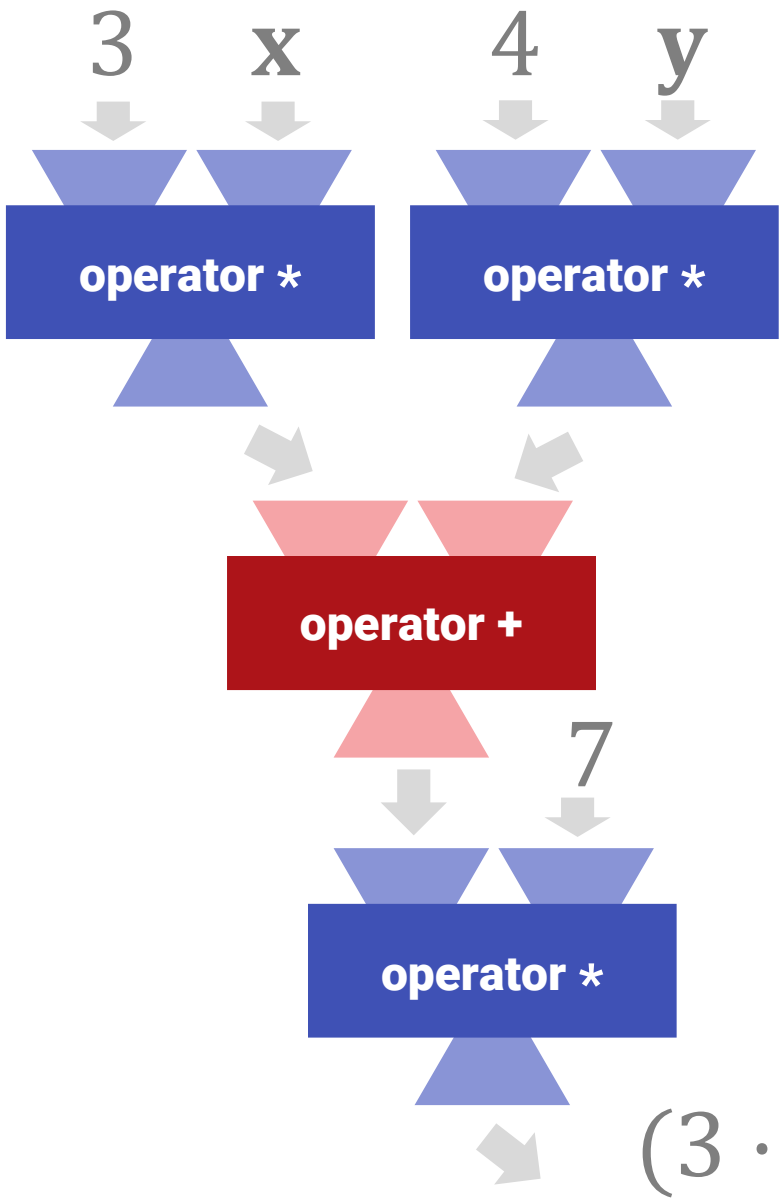
Variablen



Speicherzuordnung

- Ersetze symbolische Namen durch Indizes
- „ggT“ ist nur ein symbolischer Name für Speicherzelle #1
- Speicher belegen bei jedem Unterprogramaufruf

Komplexe Ausdrücke

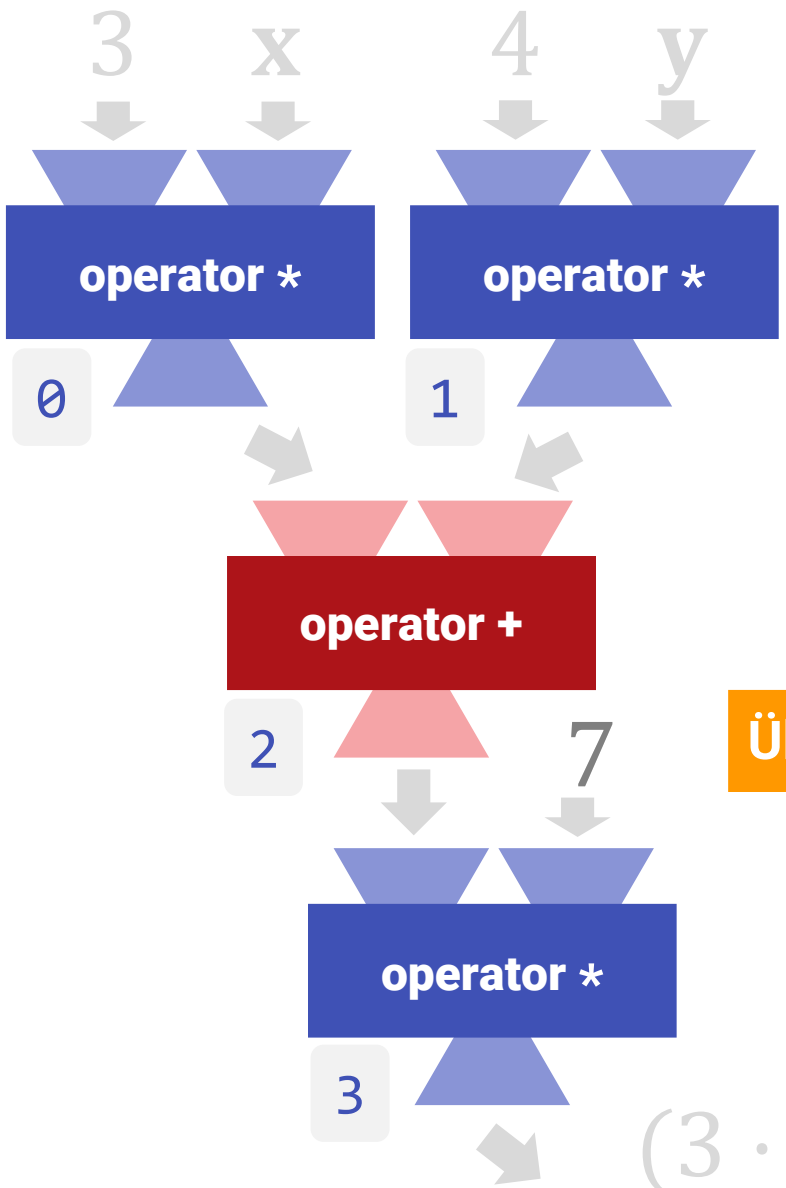


“$(3 \cdot \mathbf{x} + 4 \cdot \mathbf{y}) \cdot 7$”

Bedeutung

$$(3 \cdot \mathbf{x} + 4 \cdot \mathbf{y}) \cdot 7$$

Komplexe Ausdrücke



“(3 · x + 4 · y) · 7”

0	1	2	3	4	5	6	7
3	„x“	…	4	„y“	…	7	…

Übersetzung

Programm

```
0 mult    (0,1) → 2
1 mult    (3,4) → 5
2 add     (2,5) → 5
3 mult    (5,6) → 7
```

Vorteile von C/C++/Pascal etc.

Direkte Übersetzung

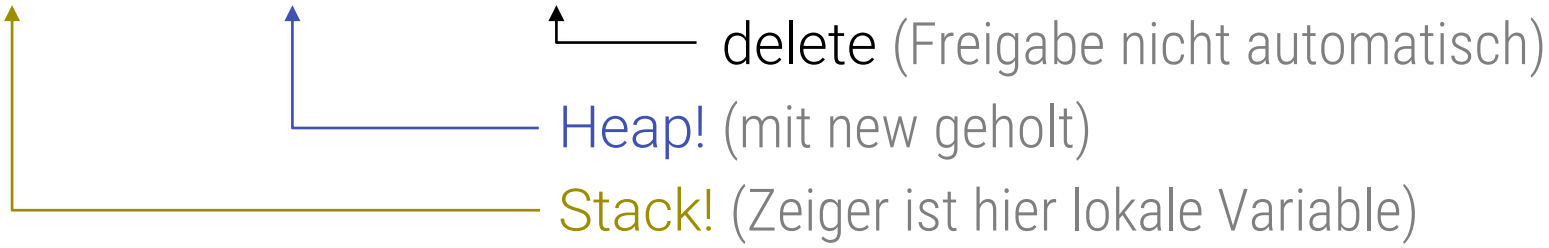
- Arithmetische Operationen in Maschinenbefehle
- Typen statisch bekannt – feste Operationen
- Design der Sprache nah an Hardwaremöglichkeiten
- Kein Overhead wenn nicht nötig
- Höhere Konstrukte mit wenig oder gar keinem Overhead
  - Memberfunktionen → inline-Code (wenn lohnend)
  - Lokale und globale Optimierung nach Übersetzung
  - Templates ohne Laufzeitoverhead
  - Minimaler Overhead für OOP/Vererbung (“virtual methods”)

Speichermodell

Wie wird Speicher verwaltet?

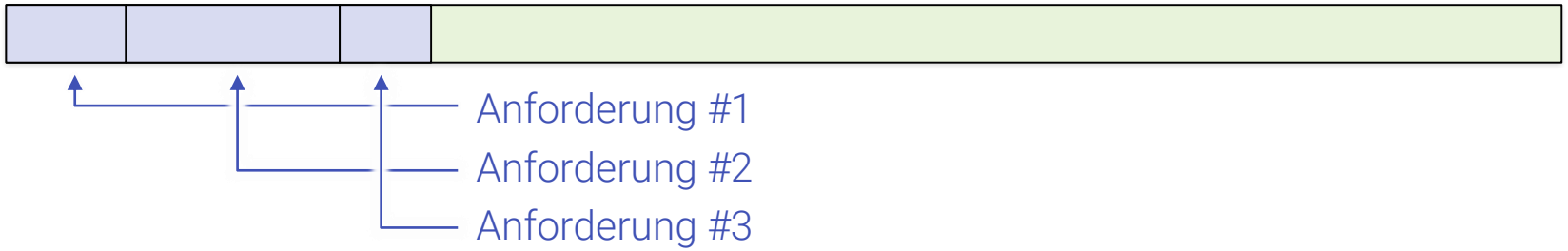
Zwei Sorten von Speicher

- **„Stack“ Speicher** – für lokale Variablen
  - Variablen (inkl. Parameter von Funktionen)
    - `int a; float b;`
  - Zeiger:
    - `int *c; int[] d;`
    - Nicht der Wert auf den gezeigt wird!
- **„Heap“ Speicher** – für dynamische alloziierte Objekte
  - Genau das, was mit „new“ angelegt wird
  - `c = new int; … delete c;`



Heap

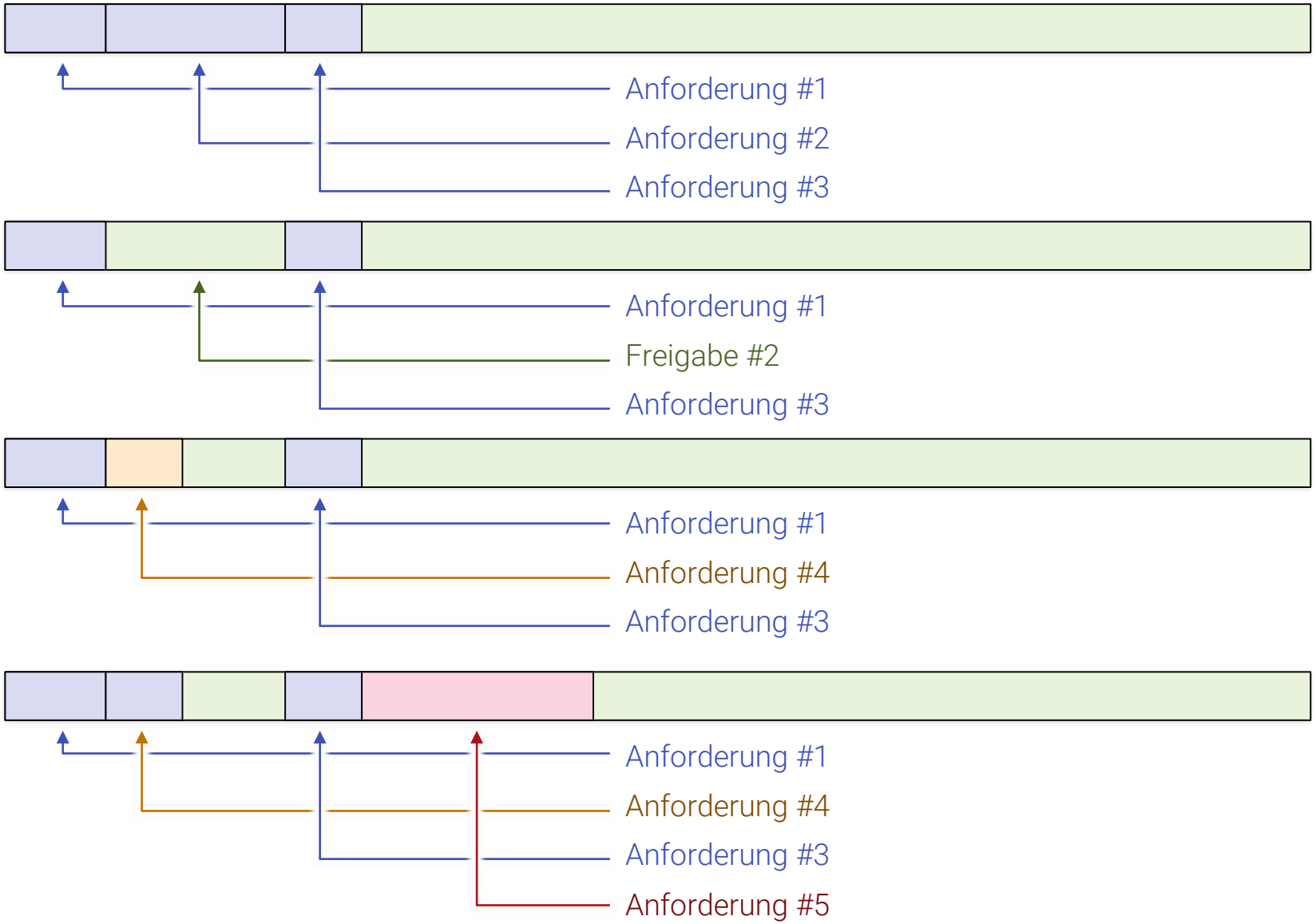
Heap Speicher



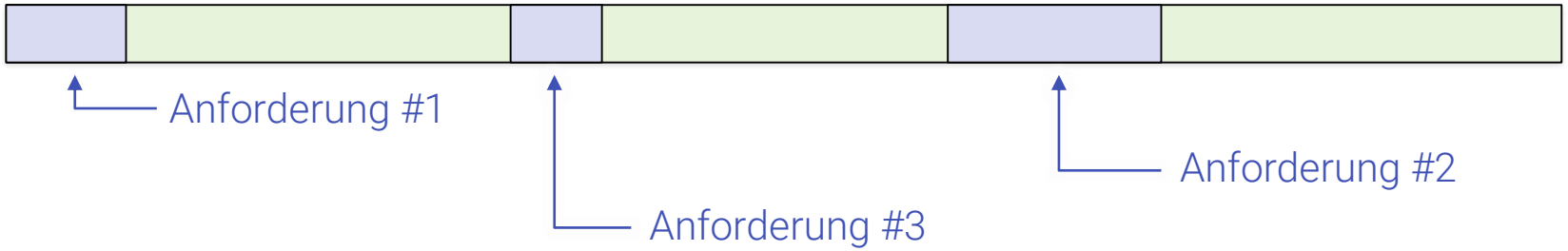
Einfaches Prinzip

- **Anforderung:** `new` fordert Speicher an
  - Freier Block an Hauptspeicher wird gesucht
  - Achtung: Kosten! (Suchkosten)
- **Freigabe:** `delete` gibt Speicher wieder frei
  - Speicher kann danach wiederverwendet werden
- Heap = Freier Hauptspeicher

Heap Speicher



Heap Speicher

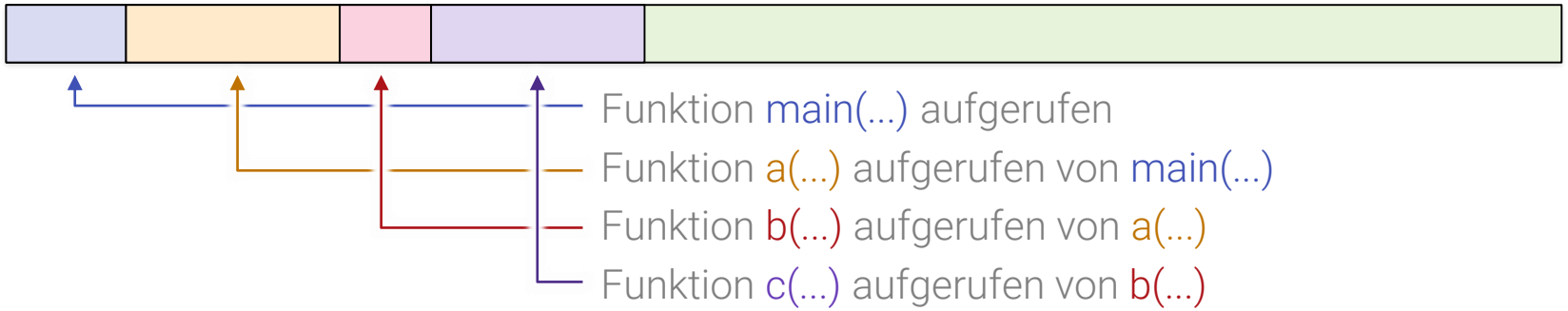


Freigabe

- Via „**`delete`**“
  - Aufpassen! Typischer Fehler wenn man JAVA gewohnt ist.
- Fehlerquelle!
  - Vergisst man delete: „Speicherloch“
  - Ruft man es doppelt auf: Absturz
- In Python + Java: vollautomatisch
  - „Garbage Collection“
  - Löschen, falls Block unerreichbar ist

Stack

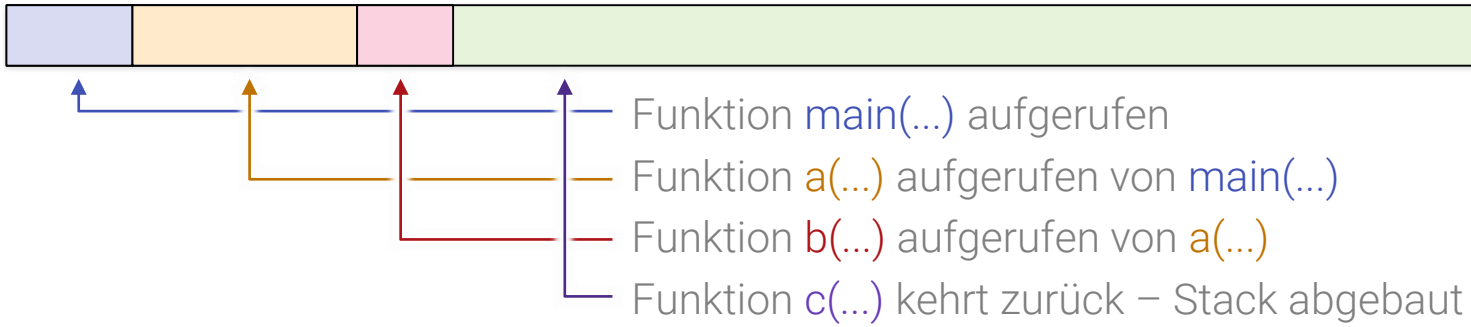
Stack Speicher



Stackspeicher

- Bei jedem Aufruf einer Funktion (Unterprogramm)
  - Für alle Lokalen Variablen wird Speicher angelegt
  - Immer für alle gleichzeitig → Geschwindigkeit!
- Unterprogrammaufruf:
  - Wieder neuen Speicher anlegen
  - Stapeln auf altem (→ *englisch* Stapel = „Stack“)
- Teil des Hauptspeichers (z.B. spezieller Heap-Block)

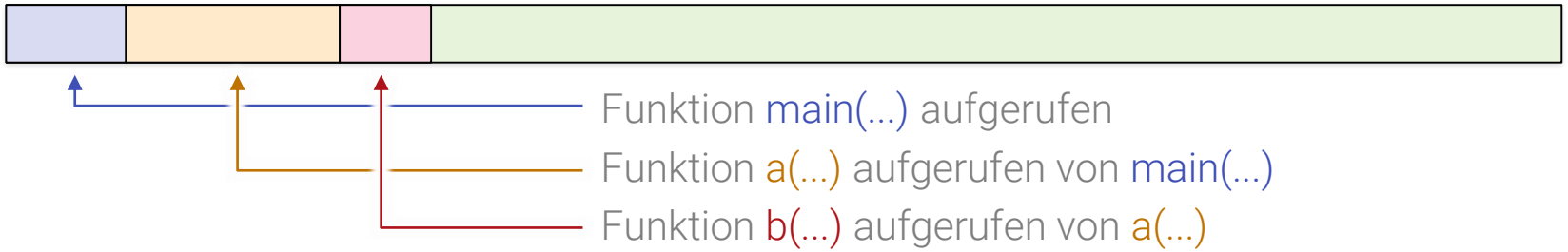
Stack Speicher



Stackspeicher

- Bei jedem Aufruf einer Funktion (Unterprogramm)
  - Für alle Lokalen Variablen wird Speicher angelegt
  - Immer für alle gleichzeitig → Geschwindigkeit!
- Unterprogrammaufruf:
  - Wieder neuen Speicher anlegen
  - Stapeln auf altem (→ *englisch* Stapel = „Stack“)
- Teil des Hauptspeichers (z.B. spezieller Heap-Block)

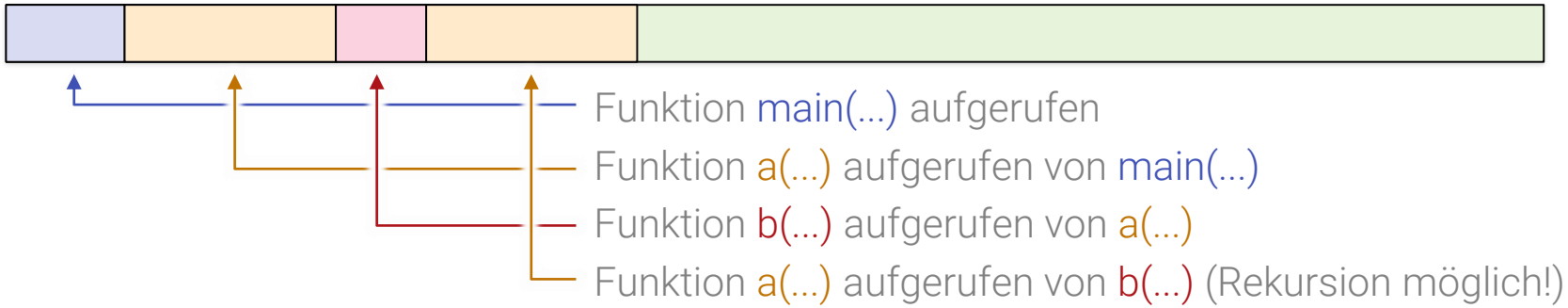
Stack Speicher



Stackspeicher

- Bei jedem Aufruf einer Funktion (Unterprogramm)
  - Für alle Lokalen Variablen wird Speicher angelegt
  - Immer für alle gleichzeitig → Geschwindigkeit!
- Unterprogrammaufruf:
  - Wieder neuen Speicher anlegen
  - Stapeln auf altem (→ *englisch* Stapel = „Stack“)
- Teil des Hauptspeichers (z.B. spezieller Heap-Block)

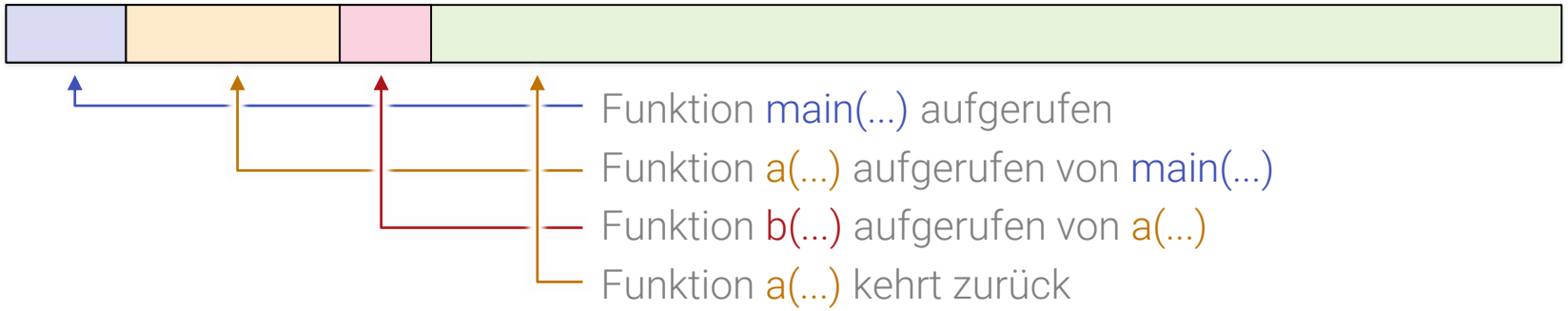
Stack Speicher



Stackspeicher

- Bei jedem Aufruf einer Funktion (Unterprogramm)
  - Für alle Lokalen Variablen wird Speicher angelegt
  - Immer für alle gleichzeitig → Geschwindigkeit!
- Unterprogrammaufruf:
  - Wieder neuen Speicher anlegen
  - Stapeln auf altem (→ *englisch* Stapel = „Stack“)
- Teil des Hauptspeichers (z.B. spezieller Heap-Block)

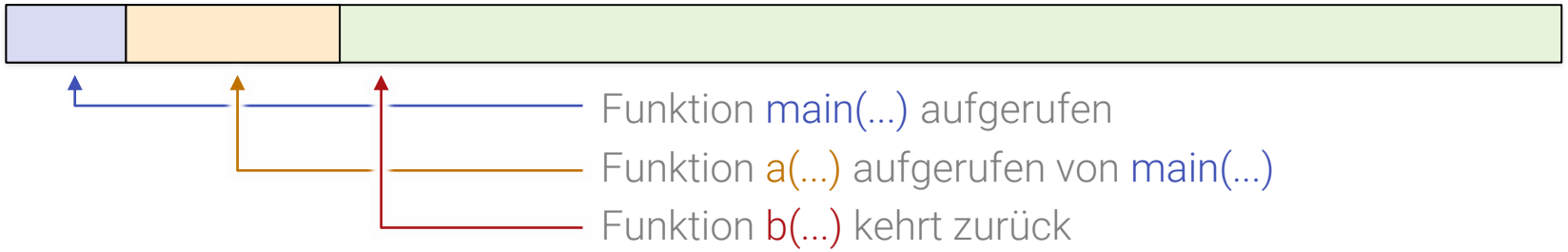
Stack Speicher



Stackspeicher

- Bei jedem Aufruf einer Funktion (Unterprogramm)
  - Für alle Lokalen Variablen wird Speicher angelegt
  - Immer für alle gleichzeitig → Geschwindigkeit!
- Unterprogrammaufruf:
  - Wieder neuen Speicher anlegen
  - Stapeln auf altem (→ *englisch* Stapel = „Stack“)
- Teil des Hauptspeichers (z.B. spezieller Heap-Block)

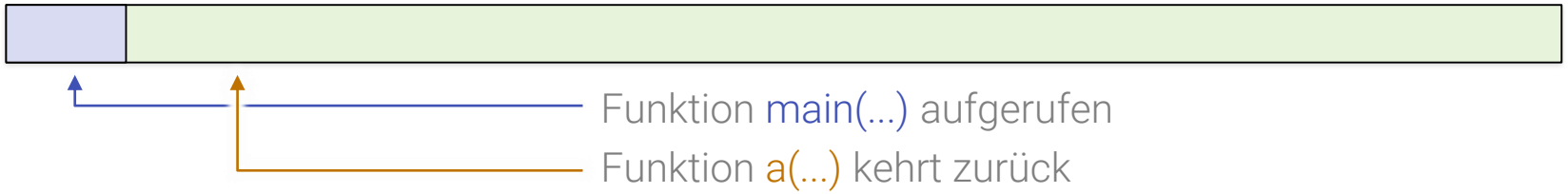
Stack Speicher



Stackspeicher

- Bei jedem Aufruf einer Funktion (Unterprogramm)
  - Für alle Lokalen Variablen wird Speicher angelegt
  - Immer für alle gleichzeitig → Geschwindigkeit!
- Unterprogrammaufruf:
  - Wieder neuen Speicher anlegen
  - Stapeln auf altem (→ *englisch* Stapel = „Stack“)
- Teil des Hauptspeichers (z.B. spezieller Heap-Block)

Stack Speicher



Stackspeicher

- Bei jedem Aufruf einer Funktion (Unterprogramm)
  - Für alle Lokalen Variablen wird Speicher angelegt
  - Immer für alle gleichzeitig → Geschwindigkeit!
- Unterprogrammaufruf:
  - Wieder neuen Speicher anlegen
  - Stapeln auf altem (→ *englisch* Stapel = „Stack“)
- Teil des Hauptspeichers (z.B. spezieller Heap-Block)

Stack Speicher

Stackspeicher

- Bei jedem Aufruf einer Funktion (Unterprogramm)
  - Für alle Lokalen Variablen wird Speicher angelegt
  - Immer für alle gleichzeitig → Geschwindigkeit!
- Unterprogrammaufruf:
  - Wieder neuen Speicher anlegen
  - Stapeln auf altem (→ *englisch* Stapel = „Stack“)
- Teil des Hauptspeichers (z.B. spezieller Heap-Block)

Beispiel

Lokale Variablen

```
int addNumbers(int a, int b) {
    int c = a + b;
    return c;
}
void mainProg(int anum) {
    int c = 123;
    // Aufruf des Unterprogramms
    int d = addNumbers(1,2);
}
```

Return

*) Die Rücksprungaddresse zur Aufrufstelle wird auch auf dem Stack gespeichert

Lokale Variablen

```
int addNumbers(int a, int b) {
    int c = a + b;
    return c;
}
void mainProg(int args) {
    int c = 123;
    // Aufruf des Unterprogramms
    int d = addNumbers(1,2);
}
```

Datenobjekte