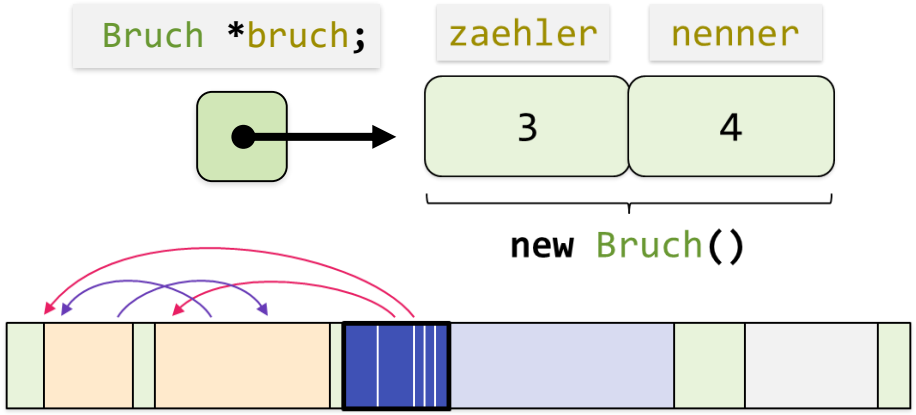
Einführung in die Softwareentwicklung

SOMMERSEMESTER 2020



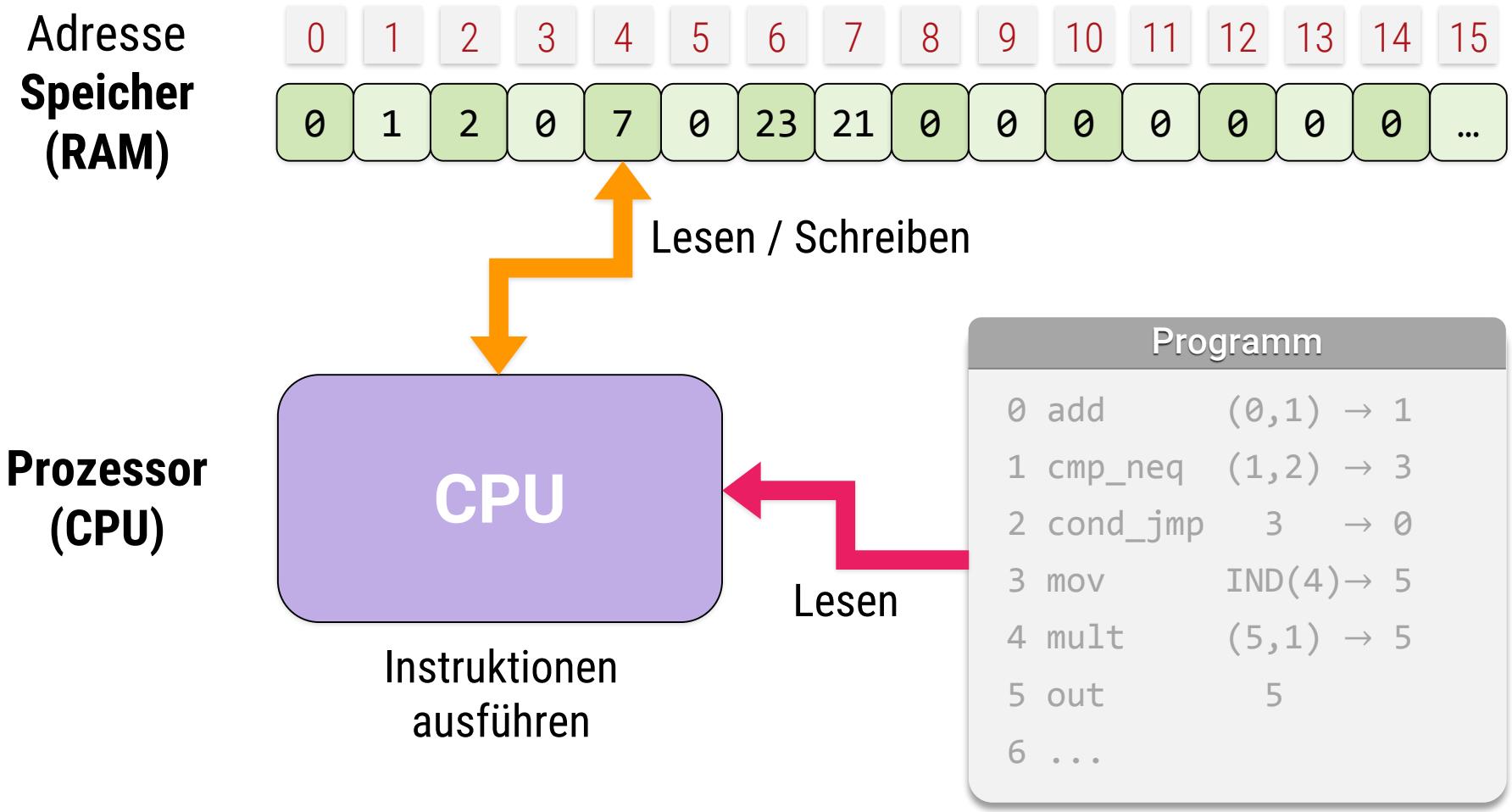
Foliensatz #04

Zeiger & Speicherverwaltung

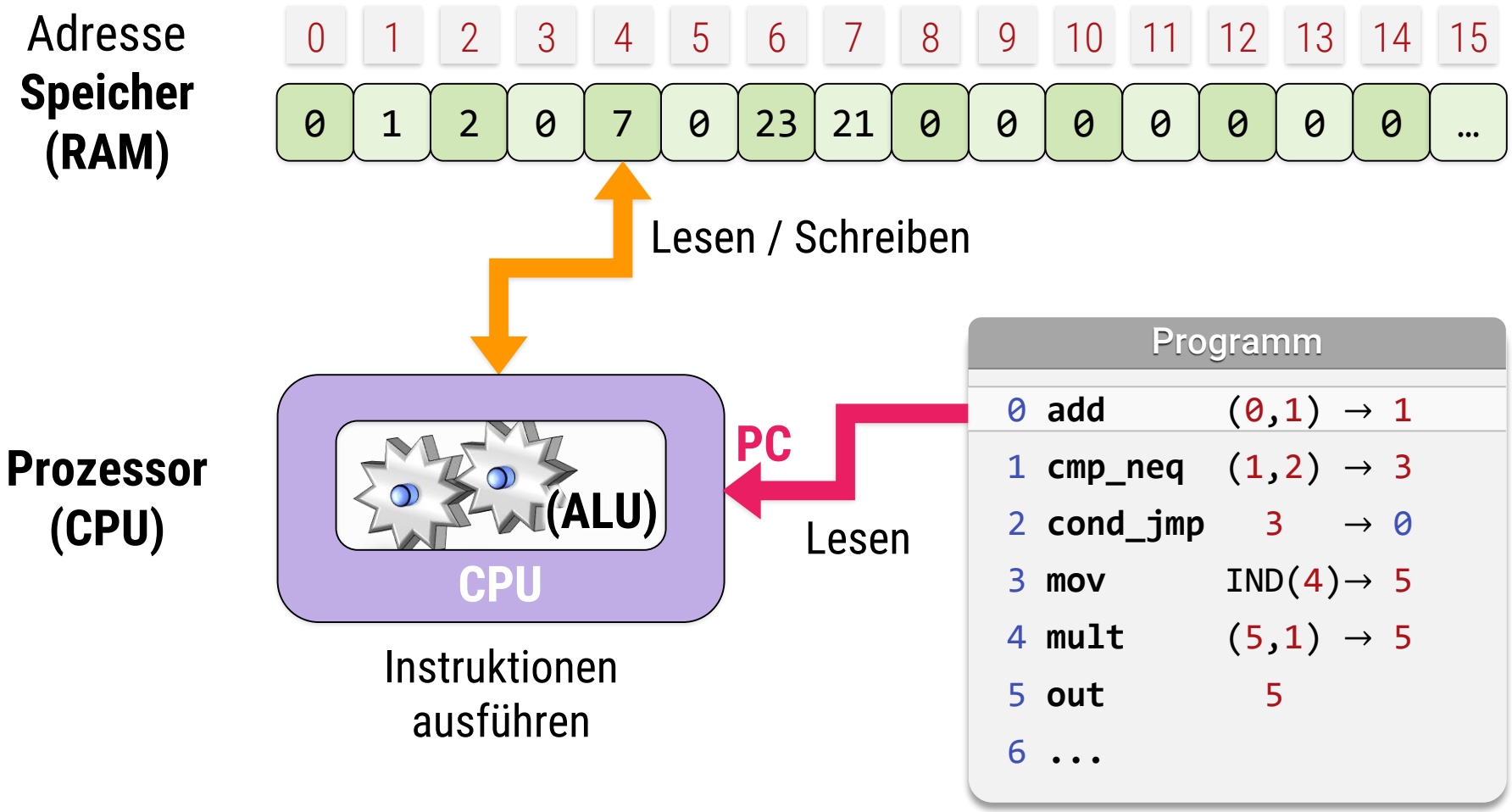
Michael Wand · Institut für Informatik · michael.wand@uni-mainz.de

Computer: Hardware

Ein abstrakter Computer



Ein abstrakter Computer



Das Prinzip

Computer

- Speicherzellen lesen und schreiben
- Instruktionen arbeiten mit deren Inhalt
  Typischerweise:
  - **Arithmetik** (Addition, Subtraktion, Multiplikation, Division,…)
  - **Vergleiche** (=, < , ≤, ≥, >, …)
  - **Sprünge** (Instruktionen wiederholen)
  - ***Bedingte* Sprünge**
    - abhängig von Speicherzellen
    - also: von früheren Rechenergebnissen / Vergleichen
  - **Indirekte Adressierung** (Werte als Adressen verwenden)
  - **Ein/Ausgabe** (damit man was sieht)

Maschinenbefehle

Beispiel – Befehle nicht für Klausur lernen!

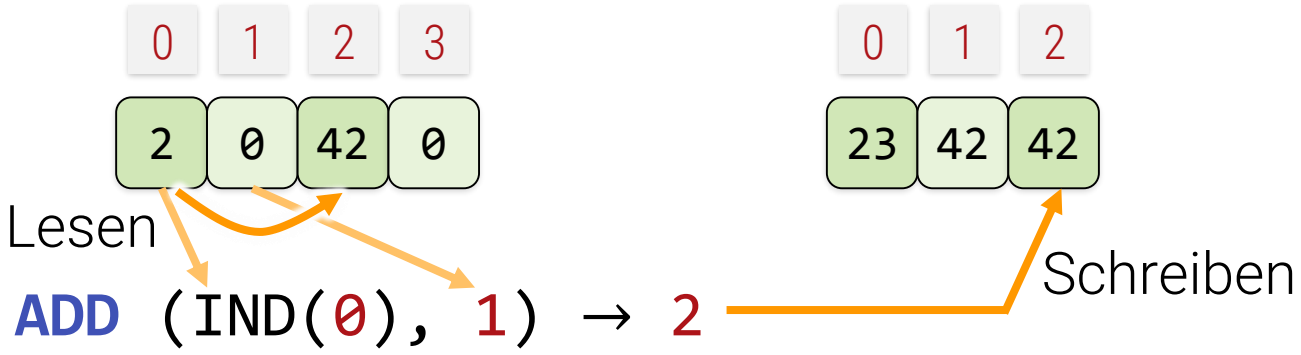
Typische Befehle

- ADD, SUB, MUL, DIV – Grundrechenarten
- FADD, FSUB, FMUL, FDIV – Fließkommazahlen
- MOV – Werte kopieren zwischen Speicherzellen
- JMP – Sprung zu anderem Befehl (PC setzen)
- CMP – Vergleichsoperationen (z.B. $<, \leq, =, \ldots$)
  - Hier: CMP: „=“; CMP_NEQ: „≠“
- COND_JMP – Sprung falls Speicherzelle Wert 1 hat

Praxis

- Viel mehr Details, stark CPU abhängig
- Immer Umweg über Prozessorregister (Effizienz)

Adressierung

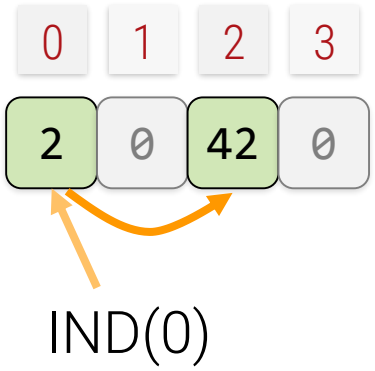


Indirekte Adressierung

- Speicherzelle enthält Adresse, nicht Wert
  - „Zeiger“ auf andere Speicherzelle
  - Indirektion
- Mein Beispielcode: `IND()` Attribut für alle Befehle, z.B.
  - `ADD (0,1) → 2`
  - `ADD (0,1) → IND(3)`
  - `ADD (IND(0), IND(1)) → 2`

(Beispiel)

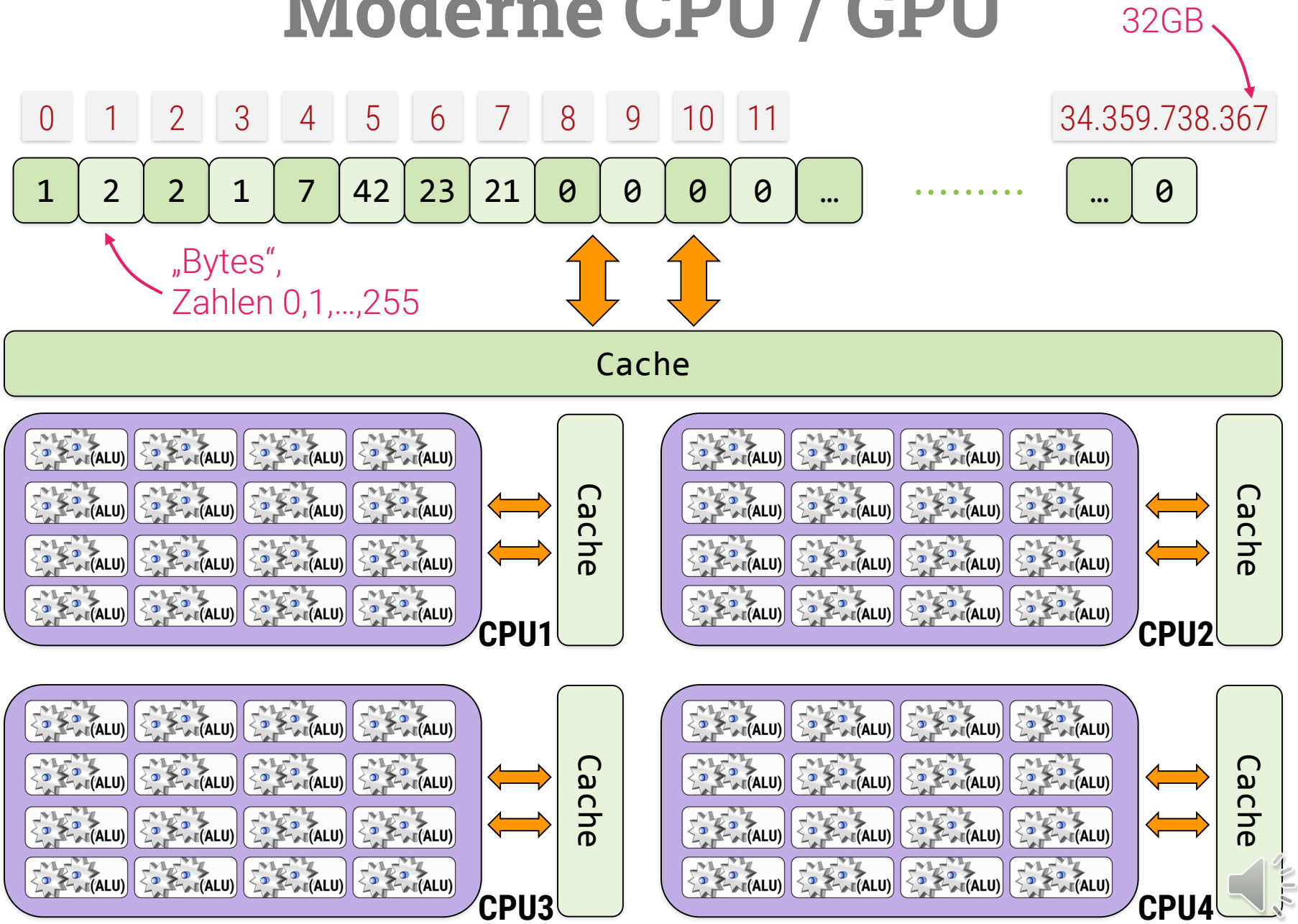
Indirekte Adressierung = Zeiger



Beispiel

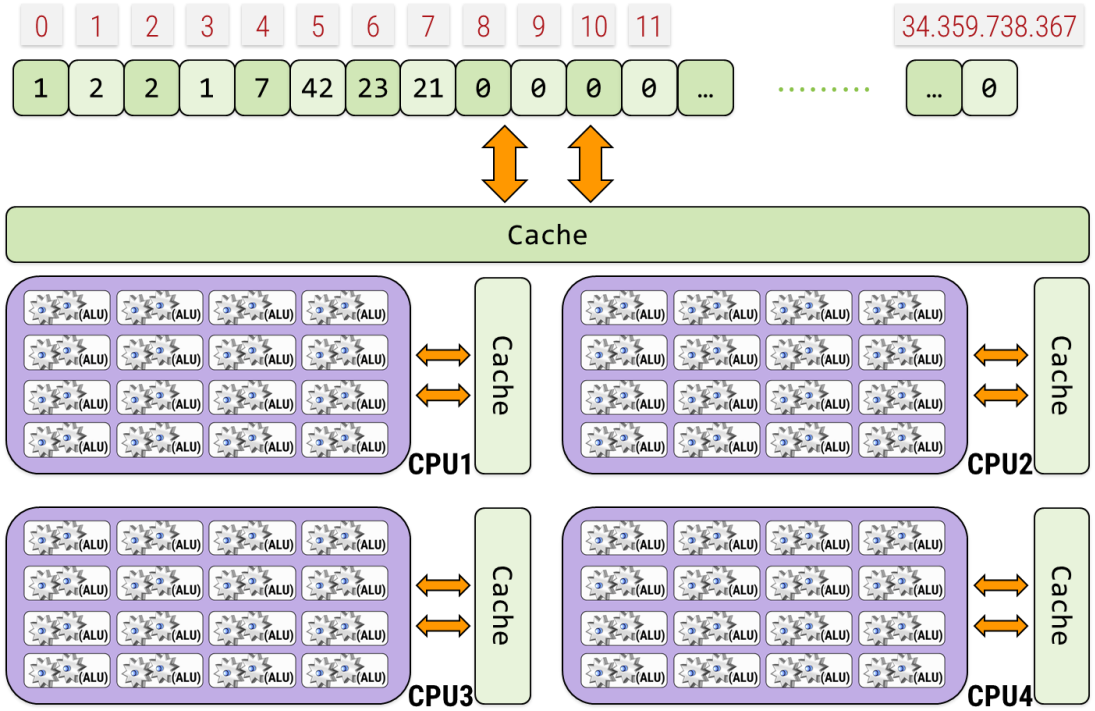
- Speicherzelle 0 enthält Zeiger
  - Zeigt auf Speicherzelle 2
- Programm kann Zeigern verändern
  - Einfach eine ganze Zahl
    - Auf x86 / 32-bit ARM / Motorola 68K u.ä.: 32 Bit Zahl
    - Auf AMD/Intel „x64“ / 64-bit ARM u.ä.: 64 Bit Zahl
  - Alle Rechenoperationen möglich

Moderne CPU / GPU

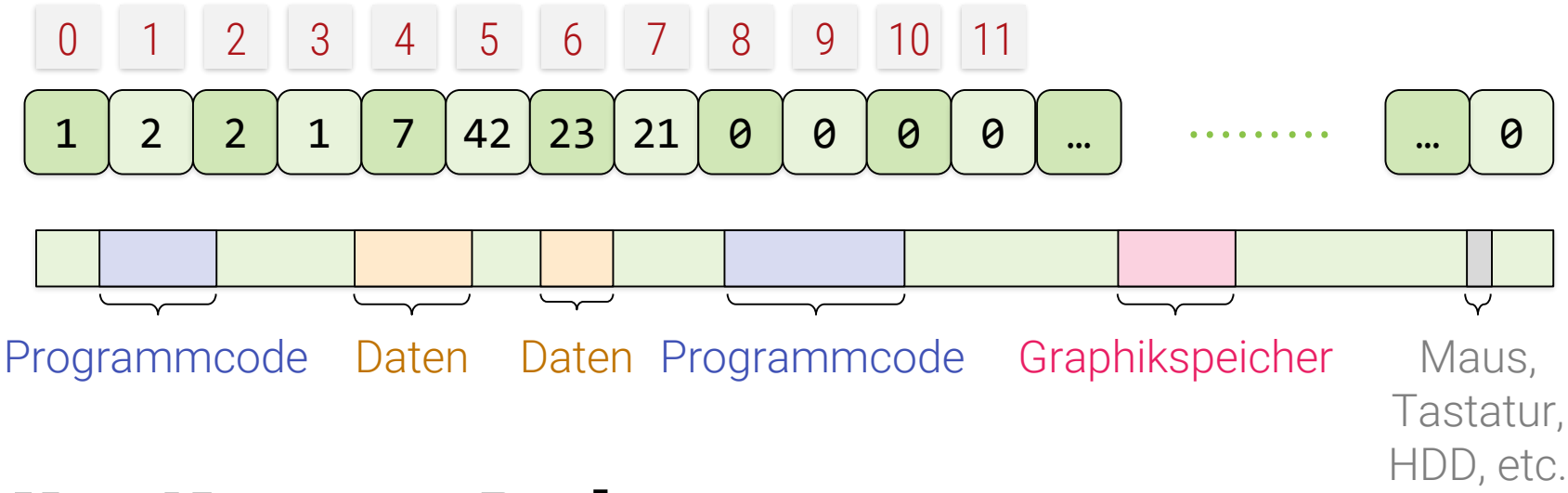


So ungefähr…

Immer noch stark vereinfacht!



Rechnerorganisation



Von-Neumann Rechner

- Programme im selben Speicher wie Daten

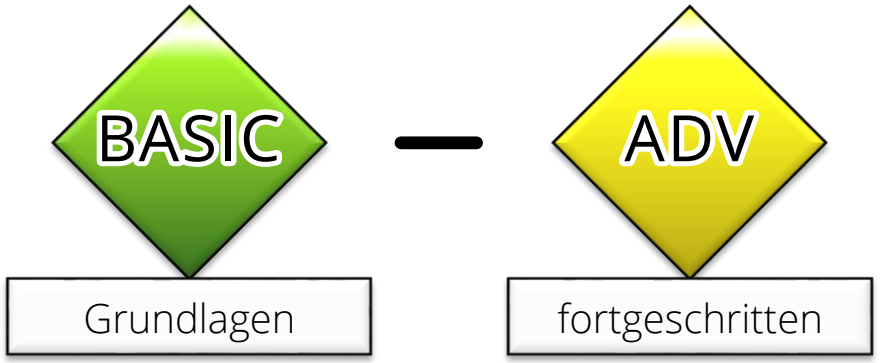
Memory-Mapped I/O

- Pseudo-Speicherzellen für Geräte und Bildschirm

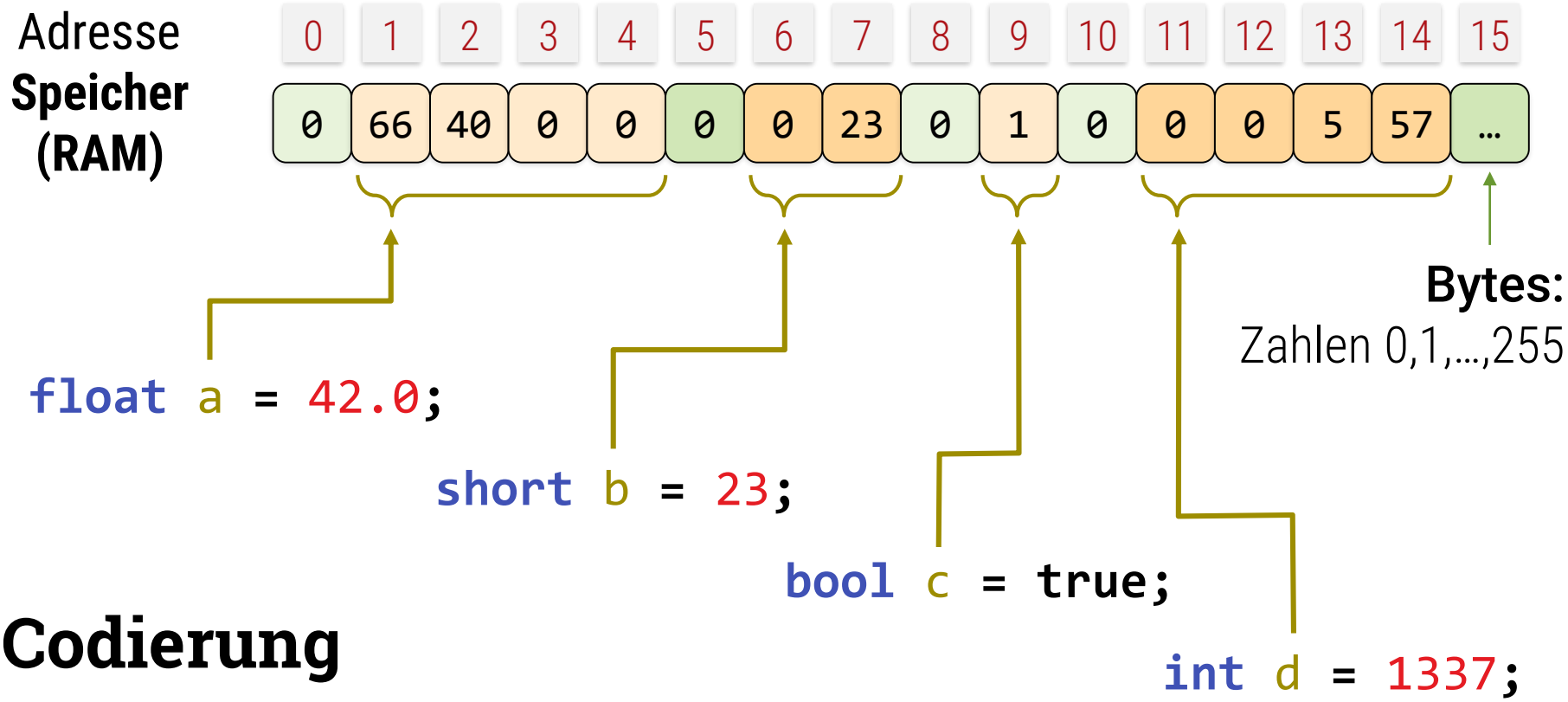
Interrupts:

- Code wird angesprungen bei Ereignissen (z.B.: Maus bewegt)

Zeiger



Maschinenrepräsentation



Codierung

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

Zeiger in C++

Zeiger:

- Integraler Bestandteil der Sprache
- Wichtig für effizientes und maschinennahes Programmieren
- Auch wichtig für OOP („Identity Objects“ wie in Python)

Drei Regeln

- **&**`<var>`: Addresse (Zeiger auf) `<var>`
- ***`<var>`**: Inhalt von Zeiger `<var>`
- `<typ>`***: Zeigertyp auf `<typ>`

nicht verwechseln mit **Referenzen <Typ> &**

Zeiger in C/C++

Beispiele für Zeiger

```
int a;
int *b; // Man schreibt den Stern an die Variable (Konvention weil: int *a,b nur ein Zeiger)

a = 42;
b = &a;

cout << a;  // Ergebnis 42
cout << *b; // Ergebnis ist auch 42

a = 23;

cout << a;  // Ergebnis 23
cout << *b; // Ergebnis ist auch 23

double *c = &a; // Typfehler! (Beim Übersetzen; zeigt nicht auf double)

uint8_t *c = (uint8_t*)&a; // Explizite Typkonvertierung (use at your own risk)

cout << *(c) <<" "<< *(c+1) <<" "<< *(c+2) <<" "<< *(c+3);
// ↑ „Pointerartithmetik“ – Erlaubt, aber evtl. riskant (z.B. crash falls int kleiner als 32 Bit) ↑
// Für 32Bit ints wird die Kodierung in einzelnen Bytes im Speicher ausgegeben (try it)
```

Zeiger in C++

Zeigertypen

- Es gibt verschiedene Zeigertypen!
  - **`int*`**, **`float*`**, **`short unsigned int*`** etc.
- Was ist gleich?
  - Alle Zeiger sind auf Maschinenebene gleich!
  - In der Regel 32-Bit oder 64-Bit unsigned int.
  - Hängt vom Betriebsystem + CPU ab (32/64 Bit)
- Wo ist der Unterschied?
  - „Bezugstyp“ – C++ weiß, worauf der Zeiger zeigt!
  - Konvertierung ist erlaubt (muss explizit erfolgen: cast)
  - **`void*`** zeigt auf „irgendetwas“ (zuweisungskompatibel zu allem)

Referenzen (nur C++) vs. Zeiger

Beispiele für Zeiger

```
int a = 42;
int *b = nullptr;  // Zeigervariable, Initialisierung optional

b = &a;   // Zeiger auf a an Zeigervariable b zuweisen
*b = 23;  // a indirekt verändern

int &c = a;  // Referenzvariable, Initialisierung vorgeschrieben!
c = 1337;    // a indirekt verändern
```

Referenzen

- Initialisierung wird erzwungen
  - „Non-nullable“ Pointer
  - Erzeugt „Alias“ – gleicher Speicherplatz
  - Auch nicht 100% sicher; Aliase auf dyn. allokierten Speicher erlaubt

Wege ins Nirvana

Zeiger ins Nichts

```
int *iWillCrashSoHard() {
   int localVariable = 42;
   return &localVariable; // WTF!
}

int &goingSouth() {
   int localVariable = 42;
   return localVariable; // subtil, aber auch tödlich
}

int *deadRef = iWillCrashSoHard();     // bis hierher noch ok
*deadRef = 23;                         // NOPE. Das geht schief.

goingSouth() = 23;                     // Syntaktisch richtig, aber tödlich
```

Manuelle Speicherverwaltung erfordert Sorgfalt…

Wege ins Nirvana

Zeiger ins Nichts

```
int *iWillCrashSoHard() {
    int localVariable = 42;
    return &localVariable; /
}

int &goingSouth() {
    int localVariable = 42;
    return localVariable; //
}

int *deadRef = iWillCrashSoH
*deadRef = 23;

goingSouth() = 23;
```

MSVC 2013
warning: C4172: returning address of local variable or temporary

MinGW (gcc) 4.9.2
warning: address of local variable 'localVariable' returned

MSVC 2013
warning: C4172: returning address of local variable or temporary

MinGW (gcc) 4.9.2
warning: reference to local variable 'localVariable' returned

Manuelle Speicherverwaltung erfordert Sorgfalt…

Regeln für Zeiger

Null-Zeiger

- Alle Variablen anfangs uninitialisiert
  - Auch Zeiger: Beliebige Adresse – sehr gefährlich!
  - Wenigstens null-Zeiger initialisierung
- Null Zeiger (entspr. in etwa „**None**“ in Python):
  - C: `int *a = 0;` // das passiert wirklich!
  - C++98: `int *a = NULL;` // NULL ist 0 (eine Konstante)
  - C++11: `int *a = nullptr;` // moderne Version (empfohlen)
  - Alles abwärtskompatibel!
- Lesen/Schreiben auf Null-Pointer stürzt auch ab
  - Aber weniger schlimm, die meisten OS erkennen dies und beenden das Programm

Regeln für Zeiger

Operationen auf Zeigern

- „*****“ Dereferenzierung (von links)
- Arithmetik: **+**, **-**, **++**, **--**, **+=**, **-=** erlaubt
- Achtung: „Bezugstyp“
- Zeiger vom typ **<Typ*>** wird um **sizeof(Typ)** Bytes verschoben:

```
uint32_t *a = …;
uint8_t  *b = …;
double   *c = …;

a += 1; // erhöht Adresse um 4 Bytes (32Bit-Zahl)
b += 1; // erhöht Adresse um 1 Byte
b += 7; // erhöht Adresse um 7*8=56 Bytes (64Bit-double)
```

Der Zeiger ins Nichts…

Für den Notfall: `void*`

- Zeiger auf Typ „**`void`**“ ist erlaubt
- „**`void*`**“ kann auf irgendetwas zeigen
- Zuweisungskompatibel zu allen Zeigertypen
  - `<Typ>*` zu **`void*`** aber nicht umgekehrt (cast erforderlich)
- Dereferenzierung nicht möglich
- Bezugstyp für **`void*`** Zeigerarithmetik (+/-) ist ein „Maschinenwort“ (typ. 4 Bytes; jeder Compiler sieht das anders)
  - Empfehlung: keine Zeigerarithmetik mit void*
  - Immer Konvertierung in **`uint8_t*`** (dann ist es klar)
  - Zeigertypkonvertierung erzeugt keinen Code, nur Hinweis an Compiler

Dynamische Allokation (C/alt)

```
#include <stdlib.h> // alte C-Bibliothek für malloc, free, …

int *a = malloc(sizeof(int)); // Speicher ein int

*a = 42;
cout << (*a);

free(a); // Speicher freigeben!
```

Speicher reservieren in C (alt)

- `void *pointer = malloc(<size in bytes>);`
  - Reserviert Anzahl Bytes im Speicher
  - Rückgabetyp „**`void*`**“ (Kompatibel zu allen Zeigertypen)
- `free(pointer);`
  - Gibt Speicher wieder frei (manueller Aufruf nötig!)

Dynamische Allokation (C/alt)

```
#include <stdlib.h> // alte C-Bibliothek für malloc, free, …

int *a = malloc(sizeof(int)*42); // Speicher für 42 ints

for (int count = 0; count < 42; count++) {
   *(a+count) = count;
}

for (int count = 0; count < 42; count++) {
   cout << *(a+count);
}

free(a); // Speicher freigeben!
```

Neue Schreibweise (C++)

```
#include <stdlib.h> // alte C-Bibliothek für malloc, free, …

int *a = new int(); // Speicher ein int

*a = 42;
cout << (*a);

delete a; // Speicher freigeben!
```

Speicher reservieren in C++ (neu)

- `<Typ> *pointer = new <Typ>;`
  - Reserviert `sizeof(<Typ>)` Bytes im Speicher
  - Rückgabetyp „**`<Typ>*`“ (sicherer!)
- **`delete`** `pointer;`
  - Gibt Speicher wieder frei (manueller Aufruf nötig!)

Neue Schreibweise (C++)

```
#include <stdlib.h> // alte C-Bibliothek für malloc, free, …

int *a = new int[42]; // Speicher für 42 ints

for (int count = 0; count < 42; count++) {
   *(a+count) = count;
}

for (int count = 0; count < 42; count++) {
   cout << *(a+count);
}

delete[] a; // Speicher freigeben! (Klammern „[]“ wichtig!!!)
```

Array Allokation

- **new** <Typ>**[**<Zahl>**]** reserviert <Zahl> Einheiten
- **delete[]** <Zeiger> gibt Array frei (**[ ]** wichtig!)

Zeiger auf Strukturen

Zugriff auf Strukturen via Zeiger

- Beispiel

```
struct A {
    int a; int b;
    A *next;
};
A* ptr = new A();
```

- Zugriff via `(*ptr).a` umständlich
  - Vor allem bei tieferer Schachtelung:
    `(*(*(*ptr).next).next).a`
- Alternativ: `(*ptr).a` ≡ `ptr->a`
- Pfeiloperator „`->`“ dereferenziert und wählt Member aus

Arrays in C/C++

Arrays = Pointer

```
int *a  = …; // Zeiger auf Speicher
int a[] = …; // Äquivalente Schreibweise

*(a+23) = 42; // Array-Zugriff per Hand
a[23] = 42;  // Äquivalente Schreibweise
```

Alternative Schreibweise

- `a[<Zahl>]` entspricht exakt `*(a+<Zahl>)`
- `<Typ> <var>[]` entspricht `<Typ>*`
  - z.B. `int a[]` entspr. `int *a`
- Achtung:
  - `int a[]` alleine reserviert keinen Speicher!
  - Keine Zugriffskontrollen („Buffer-Overrun“ möglich)

Dynamische Allokation (C-Stil)

```
#include <stdlib.h> // alte C-Bibliothek für malloc, free, …
int a[] = malloc(sizeof(int)*42); // Speicher für 42 ints
// int *a = malloc(sizeof(int)*42); // Geht auch!
for (int count = 0; count < 42; count++) {
   a[count] = count;
}
for (int count = 0; count < 42; count++) {
   cout << a[count];
}
free(a); // Speicher freigeben!
```

Arrays fester Größe

```
int *a   = …; // Zeiger auf Speicher
int a[8]; // Array fester Größe – Speicher wird reserviert!
int a[8] = {16,23,32,42,56,64,72,83}; // Initialisierung möglich
*(a+2) += 1; // Array sind weiterhin Zeiger!
a[2] += 1;    // Äquivalente Schreibweise
int *b   = a; // Zeiger wird kopiert, nicht Inhalt!
int b[8]; *b = *a; // Achtung: kopiert nur die erste Zahl! (unerwartet: Bezugstyp int)
```

Arrays fester Größe

- Syntax: `<Typ> <var>[<Größe>];`
- Speicher wird mit angelegt
- Falle: Bezugstype ist Elementtyp (keine ganzen Arrays)

Strings

Wo sind die Strings?

- Lange Geschichte…
- Mindestens drei Möglichkeiten

Stringtypen

- „C-Strings“
- `std::string`
- Bibliotheken (z.B. QT `QString`)

C-Strings

C-Strings

- Stringtyp: **`char*`**
- Zeiger auf Array von Zeichen (i.d.R. Bytes)
- Länge? Nullterminiert
  - Letztes Zeichen ist das 0-Byte
  - Reine Konvention
  - Länge immer ein Byte mehr
  - Geht gerne schief…

Encodings

Standard

- **`char`** ist ein Byte-Typ
- Traditionell:
  - ASCII Zeichentabelle (0..127) für englische Sprache
  - ANSI/Latin-1 Tabelle für europäische Sprachen (0..255)
- Heute: Unicode, UTF-8 ist Standard
  - UTF-8: Sonderzeichen durch Folgen von Bytes
  - Achtung: nicht jedes Byte ist genau ein Zeichen (nur ASCII)
- Alternative: USC-2 (veraltet), UTF-16, UTF-32
  - USC-2 Unicode waren 16Bit Strings (nicht alle Zeichen)
  - Typen: char16_t, char32_t, wchar (Linux: 32, Windows: 16)

C-Strings

```
#include <stdio.h> // alte C-Bibliothek für printf, etc…
#include <string.h> // alte C-Bibliothek für strlen, strcmp, etc …
char *text  = "Hello World"; // Zeiger-Konstante! (auf Bytefolge, 11+1 Bytes)
printf(text);


// Diverse Fallen bei C-Strings!
text = "Neuer Text"; // Zuweisung der Zeiger!
text[1] = 'c'; // Crash auf vielen Systemen (Data Segment write)
text[100] = 'c'; // Fehler auf allen Systemen (Buffer Overrun)
int length = strlen(text); // Länge berechnen durch suche nach ((char)0)-Zeichen
// Vergleich der Zeiger, nicht Vergleich der Texte!
if (text == "Hello World") printf("yes"); else printf("no"); // ??
// Vergleich der Texte
if (strncmp(text, t2, std::max(strlen(text), strlen(t2)) == 0)
   printf("yes"); else printf("no");
```

std::string

Standard-Strings

- Klasse `std::string`
  - Array von Bytes
  - Möglichst UTF-8 Kodierung verwenden (ASCII möglich)
- Überladene Operatoren
  - Verhält sich wie eine Wert (Zuweisung, Vergleich, etc.)
  - Wir bauen das gleich nach!
- Konvertierung zu anderen Libraries via Methode „`c_str()`“
  - Gibt einen Zeiger auf Zeichen (Typ **`char*`**) zurück (Array)
  - Achtung – Zeiger auf internen Puffer (mit String gelöscht!)

std::string

```
#include <string> // neue C++-Bibliothek!
using std::string;
string text  = "Hello World"; // Konvertierung von char* automatisch!
cout << text; // Einfacher!
printf(text.c_str()); // Back to old-school!

// Weniger Fallen
if (text == "Hello World") printf("yes"); else printf("no"); // yes!
text = "Neuer Text"; // Zuweisung der Inhalte :-)
int length = text.size(); // Länge bestimmen
text[1] = 'c';          // Funktioniert!
text[100] = 'c';        // Leider auch hier ein Buffer Overrun
```

std::string

Empfehlung

- C-Strings vermeiden wann immer möglich!
  - Beliebte Fehlerquelle
- Umständlich zu nutzen
- C-Strings nötig für
  - Konstanten ("..."), in Sprache fest verankert
  - Konvertierungen (auch andere Libraries)

std::string vs. QString (u.ä.)

Alternativen

- Bibliotheken wie QT haben eigene String-Klassen
  - In QT: `QString`
  - Konvertierung via C-strings
  - Aufpassen: Zeiger auf (temporäre?) Puffer
- Grund: `std::string` gab es am Anfang von C++ nicht
- Gewissen Vor- und Nachteile
  - Nachteil: Nicht Standard
  - Vorteil: Unicode Handling einfacher

Hardwarenahe Programmierung

Hardware ansprechen

Treiber / BS Programmierung

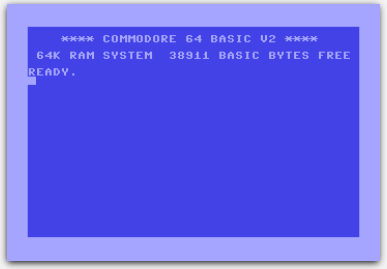
- Zugriff auf feste Adressen
- z.B. Kontrollregister (Memory-Mapped I/O)
- Möglich über konstante Pointer
  - Cast von (`long`) `int` auf Pointer-Typ

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
```

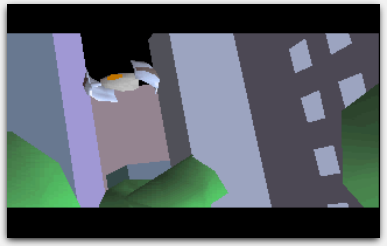


[C64 Boot Screen]

```
// 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]

Beispiel: Container

Klasse für Listen (Arrays)

Containerklasse: Listen

```
struct IntList {
    int *memory;
    int size;
    // Konstruktur
    IntList(unsigned initialSize = 0) {
        memory = new int[initialSize]; // Speicher reservieren (0 erlaubt)
        size = initialSize;
    }
    // Neu: Destruktor (bei Löschen des Objektes)
    ~IntList(unsigned initialSize = 0) {
        delete[] memory; // Speicher freigeben
    }
    // Zugriff auf Elemente
    int &operator[](unsigned index) {
        if (index >= size) {throw std::out_of_range("out of bounds");}
        return memory[index];   // ↑↑ Neu: Exceptions – ähnlich wie in Python
    }
};
```

Klasse für Listen (Arrays)

Containerklasse: Listen

```
struct IntList {
    …
    // copy-Konstruktur
    IntList(const IntList &other) {
        memory = new int[other.size]; // Speicher reservieren
        size = other.size;            // Daten kopieren
        for (int i=0; i<size; i++) {  // Daten kopieren
            memory[i] = other.memory[i];
        }
    }
    // Zuweisungsoperator
    void operator=(const IntList &other) {
        delete[] memory;
        memory = new int[other.size]; // Speicher reservieren
        size = other.size;            // Daten kopieren
        for (int i=0; i<size; i++) {  // Daten kopieren
            memory[i] = other.memory[i];
        }
    }
}
```

Benutzung

```
IntList *doSomething(int num) {
   IntList l1(3);
   for (int i=0; i<42; i++) {
      IntList l2(3); // Konstrukturaufruf l2! (42 mal!)
      l2[0] = 23;
      l2[1] = 42;
      l2[2] = 1337;
   } // Destrukturaufruf l2! (42 mal!)
   IntList *l3 = new IntList(3); // Konstrukturaufruf l3
   return l3; // l3 überlebt return, da auf Heap angelegt
} // Destrukturaufruf l1! (einmal pro Aufruf von doSomething)

void main() {
   IntList *my_l = doSomething();
   my_l[2] = 1337;
   delete my_l;
}
```

RAII – Halbautomatischer Speicher

```
{
   IntList l2(3); // Konstruktur
   l2[0] = 23;
   l2[1] = 42;
   l2[2] = 1337;
} // Destruktur (automatisch)
```

„RAII“-Stil

```
{
   IntList *l2 = new IntList(3);
   l2[0] = 23;
   l2[1] = 42;
   l2[2] = 1337;
}
…
delete l2;
```

Kein „RAII“-Stil
(üblich auch in JAVA, dort ohne delete)

RAII

- „Resource Acquisition Is Initialization“
- Lokale Variablen benutzen, um Ressource (z.B. Speicher) automatisch freizugeben
- **Tipp:** Nutzen, wann immer möglich (geht nicht immer)

Speicherverwaltungsstrategien

Wir brauchen ein Architekturmodell

- RAII
  - Einfach und sicher, aber Lebensdauer beschränkt auf Blöcke
  - Keine „Persistenz“ (z.B. Dokument einer Anwendung)
- „Ownership“-Modell
  - Baumstrukturierter Objektgraph
  - Eltern-Objekt ist „Owner“ der Kinder (verantwortlich für delete)
  - Querverweise (non-owned) möglich, kein delete
- Referenz-Counting
  - Für azyklisch-gerichtete Graphen (mehrere Owner)
- Allgemeine Graphen: Sonderlösungen oder GC

Generische Container

Container für Verschiedene Typen

Containerklasse: Listen

```
template <typename T>
struct List {
    T *memory;
    int size;
    // Konstruktur
    List(unsigned initialSize = 0) {
        memory = new T[initialSize]; // Speicher reservieren (0 erlaubt)
        size = initialSize;
    }
    // Destruktor (bei Löschen des Objektes)
    ~List(unsigned initialSize = 0) {…}
    // Zugriff auf Elemente
    T &operator[](unsigned index) {…}
    // Copy-Constructor
    List(const List<T> &other) {…}
    // Zuweisungsoperator
    void operator=(const List<T> &other) {
};
```

Templates

```
List<int> l1(3);
l1[0] = 23;
l1[1] = 42;
l1[2] = 1337;
```

```
List<double> l2(3);
l2[0] = 23.0;
l2[1] = 42.0;
l2[2] = 1337.0;
```

```
List<Bruch> l3(1);
l3[0].zaehler = 23.0;
l3[0].nenner = 42.0;
```

Statische Typisierung

- Erfordert genaue Typspezifikation
- Viele doppelte Tipparbeit

Lösung: „templates“ (generische Programmierung)

- Typparamter für Klassen/Structs oder Funktionen
- Benutzung mit Spitzen Klammern (s.o.)

Generische Programmierung

Templates in C++

„Generische Programmierung“

- Eine Form von Polymorphie
- Der selbe Code kann unterschiedliche Dinge tun
- Andere Datentypen führen zu anderem Verhalten

Werkzeug in C++: Templates

- Präambel „**`template <…>`**“ erzeugt generischen
  - Typ (struct/class)
  - Member-Funktion (von structs/classes)
  - Funktion (alleinstehend)
- Benutzung mit **`Typ<…>`** bzw. **`Funktion<…>`**

Beispiele

```
// Generischer Typ
template <typename T>
struct List {
   T *memory;
   …
};

// Generische Funktion
template <typename T>
T square(T val) {
   return val*val;
}

// Member-Template
struct IntList {
   int *memory;
   template <typename T>
   void setElementAs(unsigned i, const T val)
      {memory[i] = (int)val;}
};
```

```
// Benutzung generischer Klassen
List<int> a;
b = List<int>();
```

```
// Benutzung generischer Funkt.
float a = square(2.0f); // implizit
double b = square<double>(2); // explizit
```

```
// Benutzung generischer Memberf.
IntList a;
a.setElementAs<double>(2.0);
a.setElementAs(2.0f); // implizit
```

Explizite Spezialisierung

```
// Beispiel für Member-Templates; funktioniert mit allen Templates
struct IntList {
   int *memory;

   …

   template <typename T>
   void setElementAs(unsigned i, const T val) {
      static_assert(false, "Type not permitted."); // requires C++11 or later
   }

   template <>
   void setElementAs<int>(unsigned i, const int val) {
      memory[i] = val;
   }
   template <>
   void setElementAs<double>(unsigned i, const double val) {
      memory[i] = (int)val;
      cout << "Warning: had to round value.";
   }
};
```

Template Parameter

```
// Generisches Dictionary (langsam mit Liste)
template <typename KeyType, typename ValueType>
struct Dictionary {

   struct Entry { // lokale Definition erlaubt (Name außerhalb ist Dictionary::Entry)
      ValueType val;
      KeyType key;
   };

   List<Entry> allEntries; // Speichert alle Einträge

   const ValueType &operator[](const KeyType &key) const {
      for (int i=0; i<allEntries.size(); i++) {
         if (allEntries[i].key == key)
            return allEntries[i].value;
      }
      throw SomeException(…);
   }

};
```

```
// read-only (Instanz nicht beschreiben)
// kann überladen werden! (r/w extra)
```

```
// Benutzung
Dictionary<string, int> a;
…
cout << a["Hello"];
```

Integer-Parameter

Containerklasse: Listen

```
template <typename T, unsigned size>
struct FixedList { // Speicher reservieren
    T memory[size];

    // Konstruktur nicht nötig
    // Speicher ist bereits reserviert!
    FixedList() {}
    // Destruktor auch nicht nötig
    ~FixedList() {}

    // Zugriff auf Elemente
    T &operator[](unsigned index) {…}
    // Copy-Constructor
    FixedList(const FixedList<T,size> &other) {…}
    // Zuweisungsoperator
    void operator=(const FixedList<T,size> &other) {…}
}
```

```
// Benutzung
FixedList<double, 3> threeDVector;

// Man kann auch Namen vergeben
// (geht mit allen Typen!)
typedef FixedList<float,3> Vector3f;
typedef FixedList<double,3> Vector3d;

Vector3f v = Vector3f();
```

Python FTW

Gibt es templates in Python?

- Nicht notwendig!
- Jedes Unterprogramm ist generisch!

```
def square(a):
    return a*a
```

  - Dynamisch typisiert
  - Jeder Objektyp kann benutzt werden
  - Nachteil: Prüfung erst bei Ablauf
    - Fehler werden spät gefunden
    - Weniger Information über Funktion
    - Langsam

Duck-Typing

„Duck Typing“

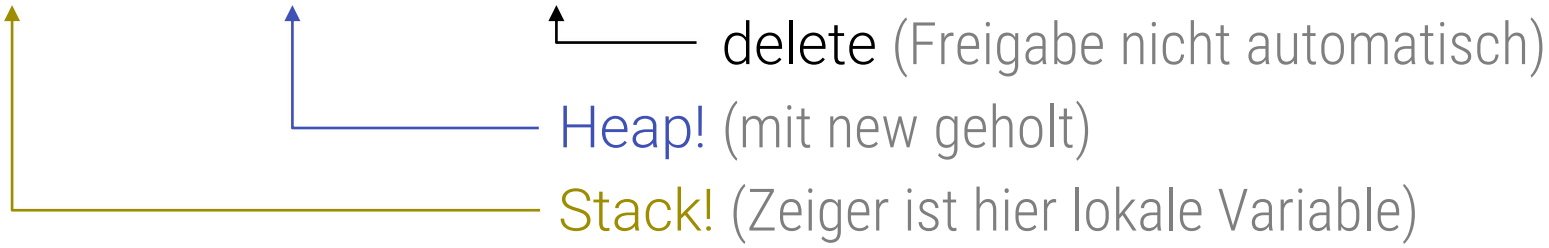
- Typen in C++ templates arbeiten wie Python Typen, nur zur Übersetzungszeit:
  - Bei Übersetzung der Instantiierung:
    - Konkrete Typen einsetzen
    - „Als wäre es so eingetippt worden“
    - Prüfen, ob der Code Sinn macht
      - Wenn ja: So übersetzen
      - Wenn nein: (Kryptischer) Fehler
- Python
  - Typen zur Laufzeit einsetzen
  - Code wird erfolgreich ausgeführt, wenn er Sinn macht

Speichermodell

Wie wird Speicher verwaltet?

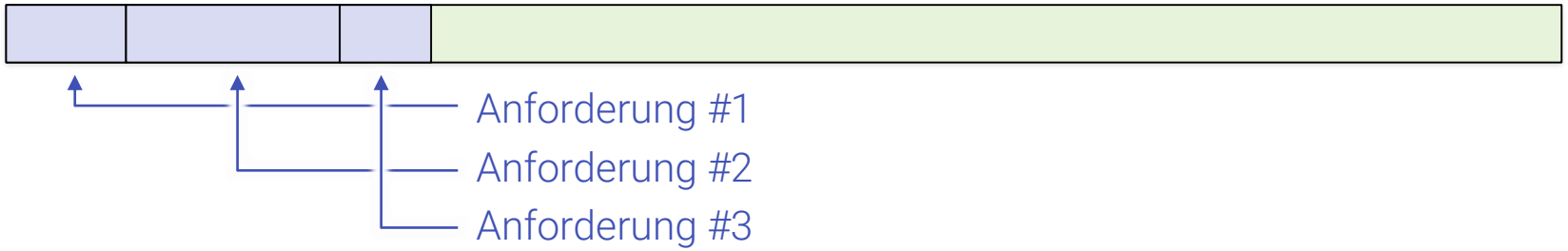
Zwei Sorten von Speicher

- **„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

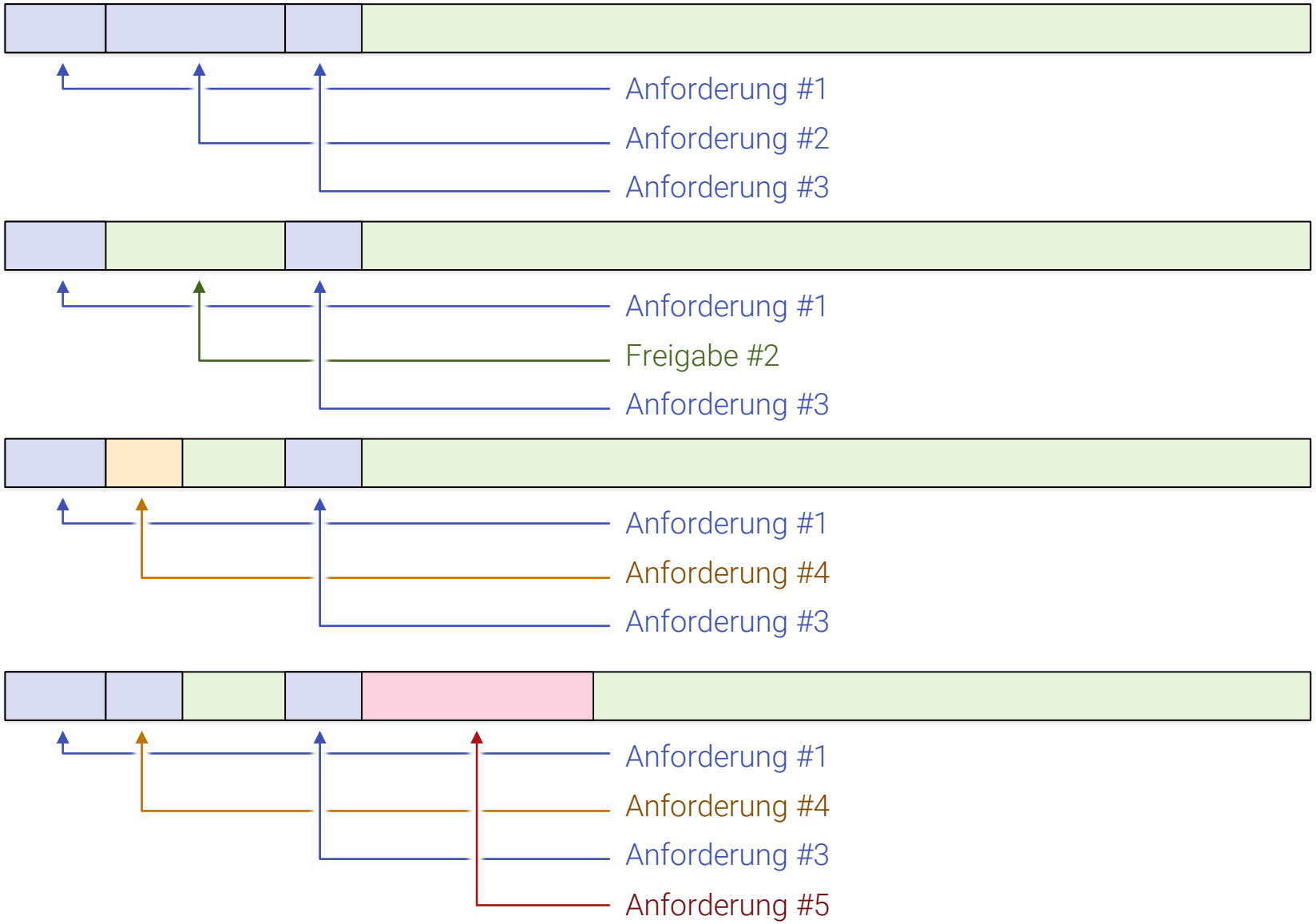
Heap Speicher



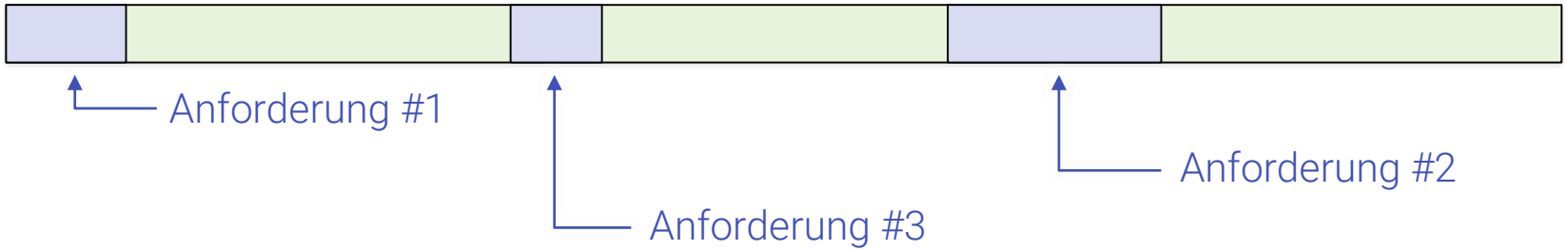
Einfaches Prinzip

- **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

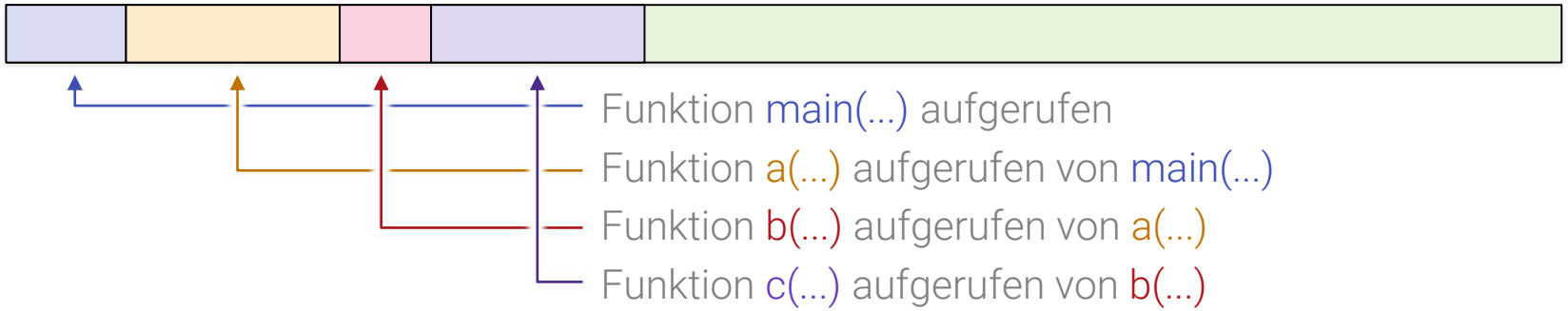


Freigabe

- 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

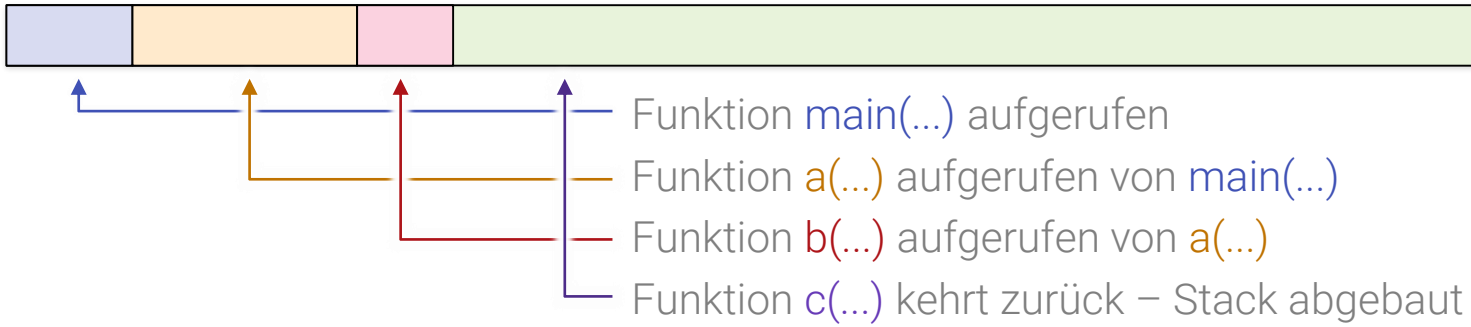
Stack Speicher



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)

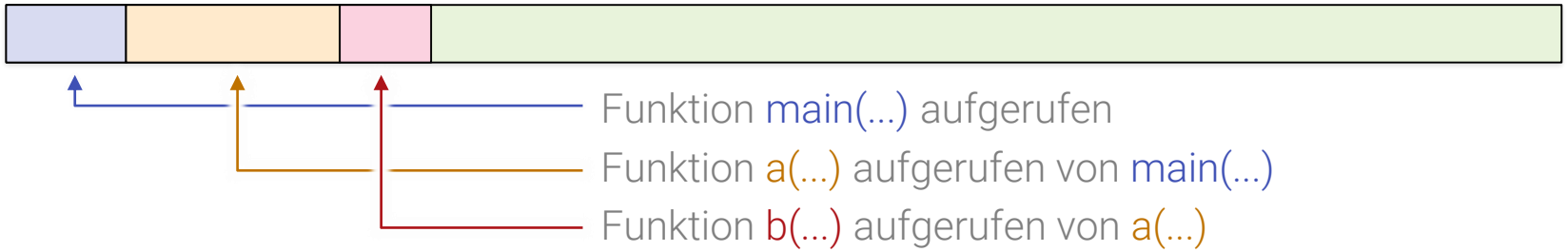
Stack Speicher



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)

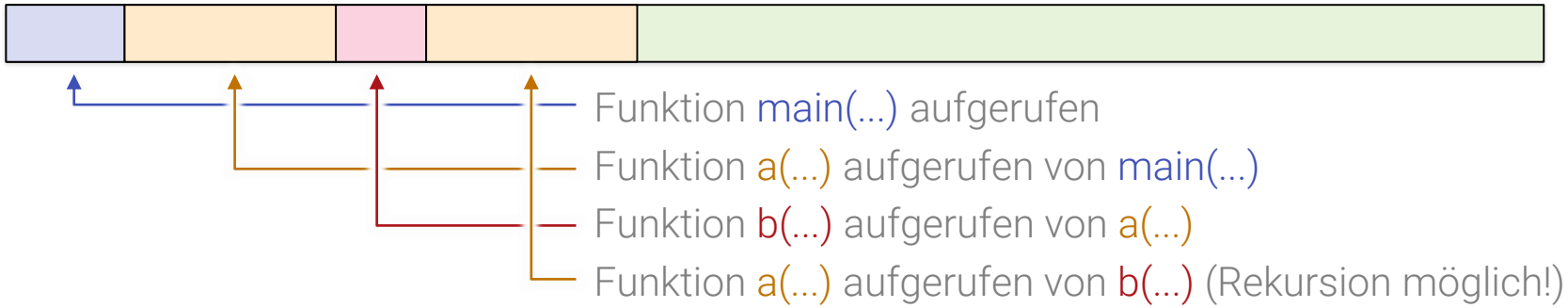
Stack Speicher



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)

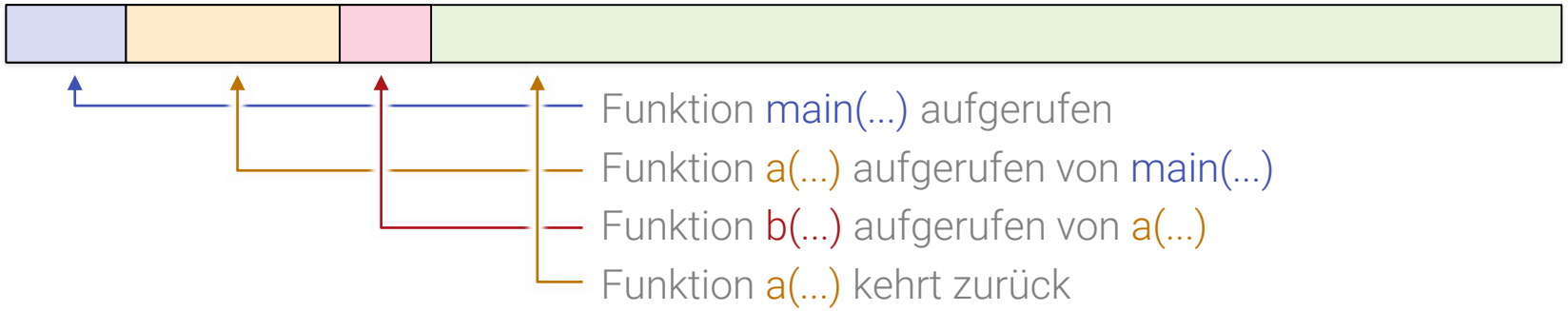
Stack Speicher



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)

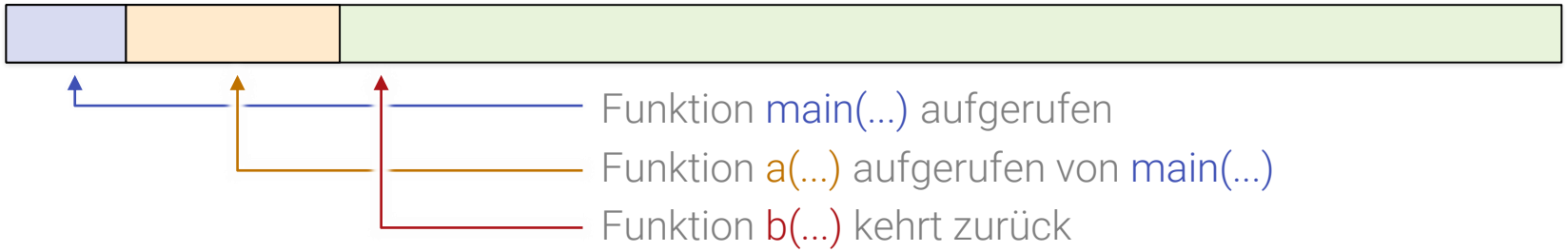
Stack Speicher



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)

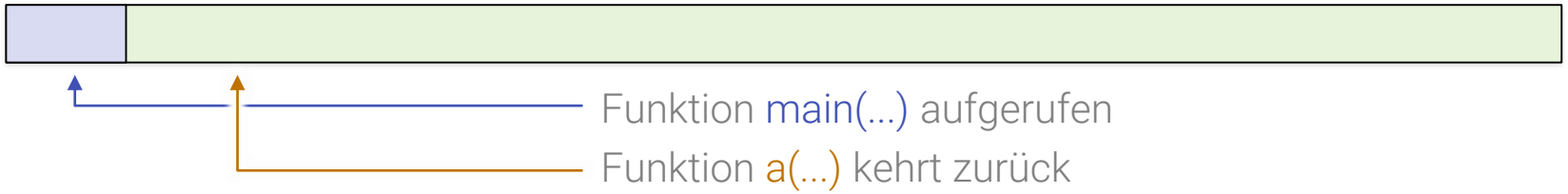
Stack Speicher



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);
}
```

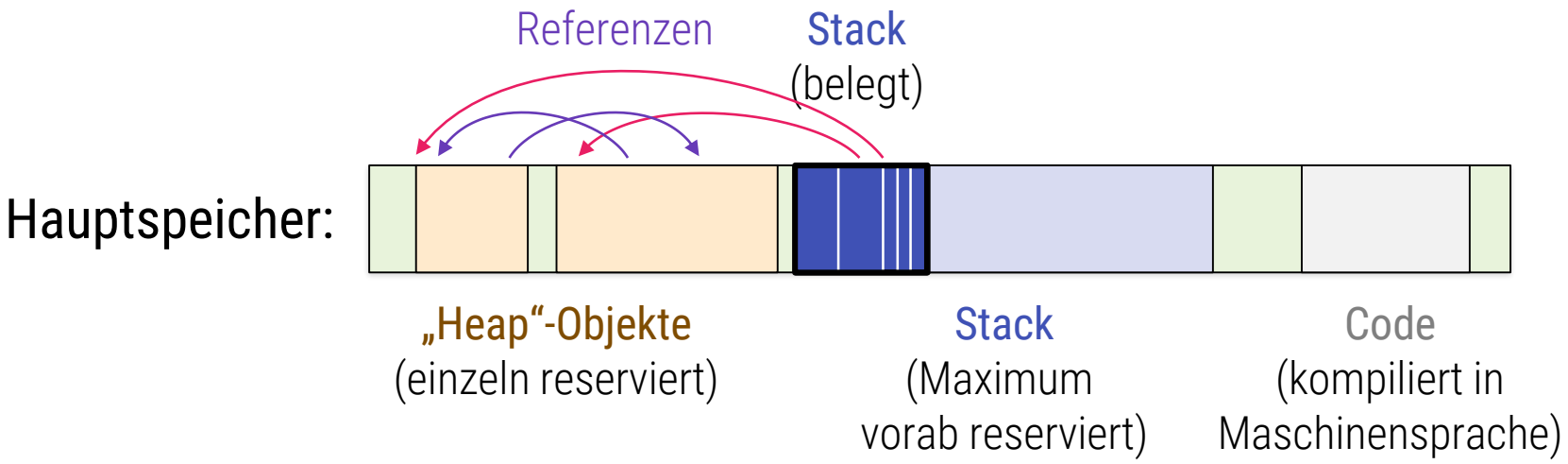
Return

*) 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



Hilfe durch das Typsystem?

Typsystem

Mögliche Garantien eines Typsystems

- Typsicher („type save“):
  - Datentypen werden nicht ungewollt vermischt bzw. falsch interpretiert, z.B. Zeichenkette als Zahl
  - Statisch (compile-time) oder dynamisch (run-time checks)
- Speicher-sicher („memory save“):
  - Kein Zugriff auf ungültigen Speicher
  - Z.B. Buffer-overruns, Stack-Overflow, Invalid Pointer Dereferentiation
- Abwesenheit von Laufzeittypfehlern
- Weitergehendes (Invarianten)
  - Oft unberechenbar, User-Support nötig (z.B. COQ, Agda, Isabelle)

Unterschiede

C, C++ und JAVA sind

- Statisch typisiert
- Stark*) typisiert
  - Nicht „sehr stark“, tatsächlich sind noch viele Fehler möglich

Python ist

- Dynamisch typisiert
- Stark*) typisiert
  - Stärker als C++, aber auch nicht „sehr stark“

Python, Java: Zusätzlich Garbage-Collection

- Kein „delete“; delete wird automatisch durchgeführt

Typsystem

Unsere Sprachen

- **Typsicher („type save“)**
  - Python: ja (dynamische Tests)
  - Java: ja (statische und dynamische Tests)
  - C++: i.d.R. ja (kann explizit umgangen werden!)
- **Speicher-sicher („memory save“)**
  - Python: ja (ggF. Laufzeitfehler)
  - Java: ja (Laufzeitfehler; Trick: garbage collector)
  - C++: nein (dangeling references, NULL-pointer deref.)
- **Abwesentheit von Laufzeitfehlern**
  - Alle drei Sprachen produzieren Laufzeitfehler
  - Nur C++ Programme können durch Typfehler abstürzen

Warum der Ärger?

C++ ist unsicher

- Ungewolltes Überschreiben von Speicher möglich
  - Andere Prozesse / OS:
    - „Segmentation Fault“ (Unix)
    - „General protection fault“ (Windows)
    - Systemcrash (schwache embedded Systeme, z.B. 68K)
  - Daten im eigenen Programm
    - Sehr schwer zu findende Fehlerquelle
    - Fehlerursache und Absturz liegen u.U. weit auseinander
- Warum tuen wir uns das an?
  - JAVA garantiert, dass das nie passieren kann
  - Python sowieso

It‘s not a Bug, it‘s a Feature!

Kosten für Speichersicherheit

- Lösung 1: Garbage Collection
  - Freigabe von ungenutzten Datenobjekten immer automatisch/implizit
  - Langsam (Overhead)
  - Schwer zu kontrollieren
  - Latenzen (GC startet plötzlich)
- Lösung 2: Komplexe Typsysteme
  - Z.B. Sprache „Rust“
  - Schwer zu handhaben (Programmierung)
  - Deckt nicht alle denkbaren Fälle ab (Architekturmuster)

C++ ist universell & schnell – dafür gefährlicher