Programmierkurs
Speicher und Arrays

Rückblick

VL 0 „Organisation und Inhalt“: Ablauf der Vorlesung, Termine
VL 1 „Hello World“: „Lebenswichtiges“, Programablauf, Programmierablauf, Kompilierung und Ausführung von Programmen
VL 2 „Die ersten Schritte“: Erstes C-Programm, Elementare C-Strukturen, Datentypen, Operatoren, Schleifen
VL 3 „Kontrollstrukturen & Funktionen“: Syntax, Semantik, bedingte Anweisungen, Blöcke, Sichtbarkeit
VL 4 „Rekursive Funktionen & Bibliotheken“: rekursive Funktionsaufrufe, Modularisierung
VL 5 „Typen“: Einfache und strukturierte Datentypen, Wertebereiche, Typendefinition
VL 6 „Speicher und Adressen“: Speicher, Pointer, Funktionsaufrufe „call by value“ vs. „call by reference“
VL 7 „Speicher und Arrays“: Speicher, Arrays, mehrdimensionale Arrays, Arrays und Pointer
VL 8 „Dynamische Speicherverwaltung“: Speicherallokation, Fehlerbehandlung, Rückgabewerte, Arrays/Pointer/Adressen
VL 9 „Strings, Kanäle, Git“: Strings und Arrays, Zeichensätze, Stringlänge, Ein- und Ausgabe, Arbeiten mit git

Recap: Speicher

Recap: Speicher

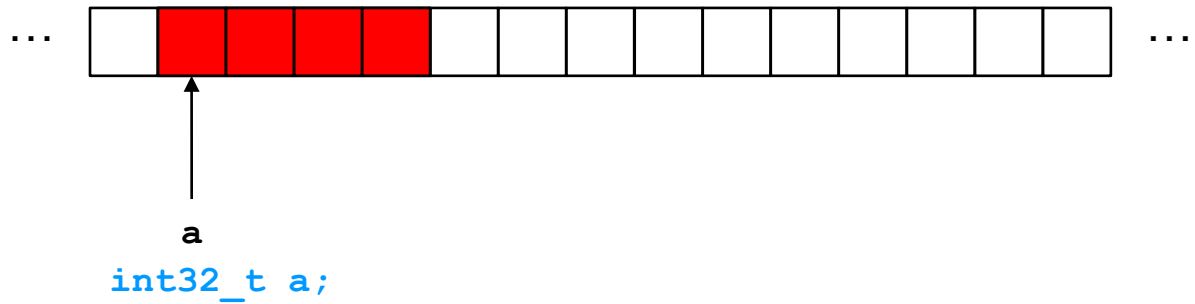
- Speicher besteht aus einer Folge von Bytes

Recap: Variablen im Speicher

- Speicher besteht aus einer Folge von Bytes
- Beispiel `int32_t`: wird in 4 aufeinanderfolgenden Bytes gespeichert

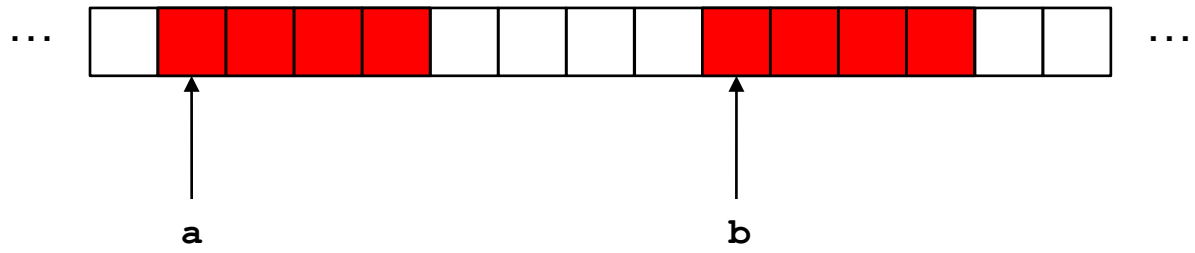
Recap: Variablen im Speicher

- Speicher besteht aus einer Folge von Bytes
- Beispiel `int32_t`: wird in 4 aufeinanderfolgenden Bytes gespeichert



Recap: Variablen im Speicher

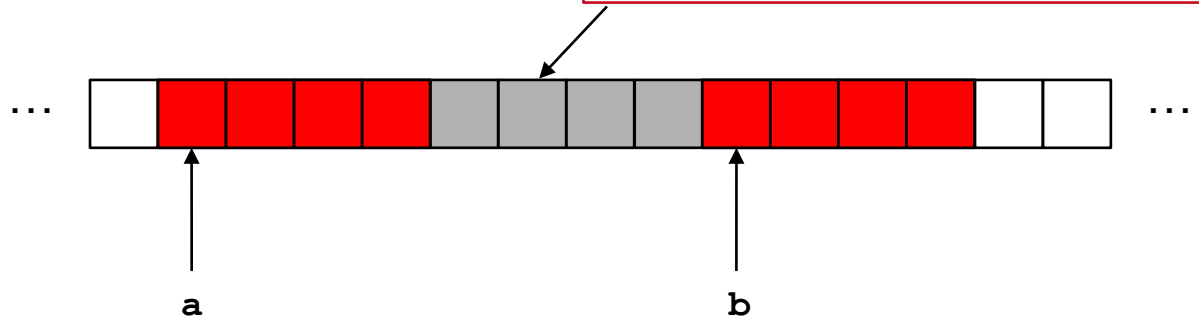
- Speicher besteht aus einer Folge von Bytes
- Beispiel `int32_t`: wird in 4 aufeinanderfolgenden Bytes gespeichert



Recap: Variablen im Speicher

- Speicher besteht aus einer Folge von Bytes
- Beispiel `int32_t`: wird in 4 aufeinanderfolgenden Bytes gespeichert

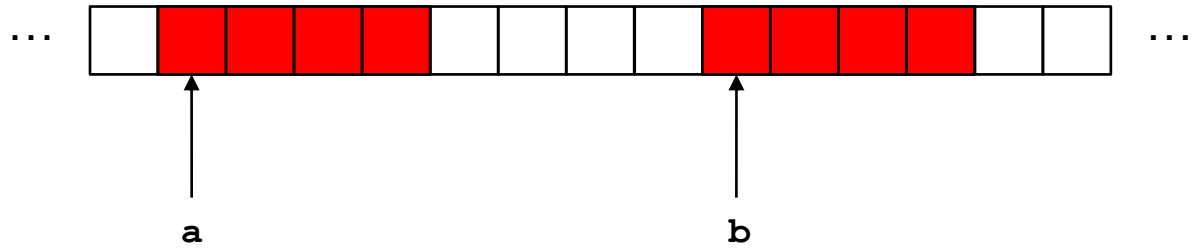
Achtung: Die Variablen werden nicht unbedingt hintereinander im Speicher abgelegt!




```
int32_t a;
int32_t b;
```

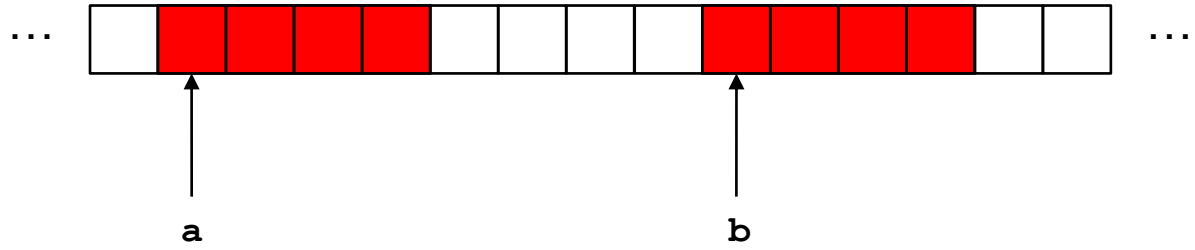
Recap: Variablen im Speicher

- Deklaration einer Variablen reserviert Speicher für die Variable



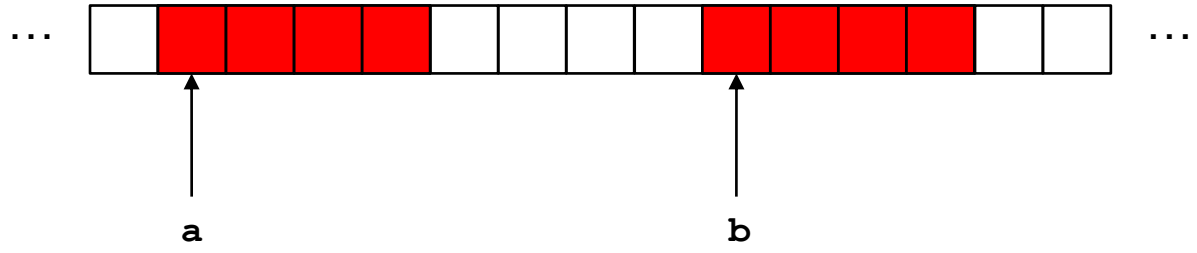
Recap: Variablen im Speicher

- Deklaration einer Variablen reserviert Speicher für die Variable
- Zuweisung entspricht Belegung des Speichers mit einem Wert



Recap: Variablen im Speicher

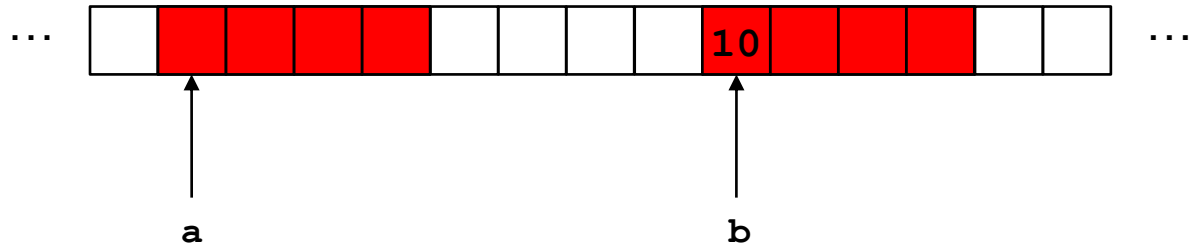
- Deklaration einer Variablen reserviert Speicher für die Variable
- Zuweisung entspricht Belegung des Speichers mit einem Wert




```
b = 10;
```

Recap: Variablen im Speicher

- Deklaration einer Variablen reserviert Speicher für die Variable
- Zuweisung entspricht Belegung des Speichers mit einem Wert




```
b = 10;
```

Datenstruktur:
Arrays (Felder)

Motivation für Datenstrukturen

- Variablen sind perfekt für einzelne Elemente
- Problem:
  - Oft gibt es mehrere Elemente des gleichen Typs
  - Möglichkeit der Darstellung: a1, a2, a3, a4, a5, …, a10
  - Einschränkungen:
    - Viel zu viel Schreibarbeit
    - Keine Möglichkeit der Iteration
    - Keine Strukturierung
- Beispiele:
  - Zahlenreihen: 1, 1, 2, 3, 5, 8, 13, …
    2, 3, 5, 7, 11, 13, 17, 19, …
  - Tabellen

Arrays (Felder)

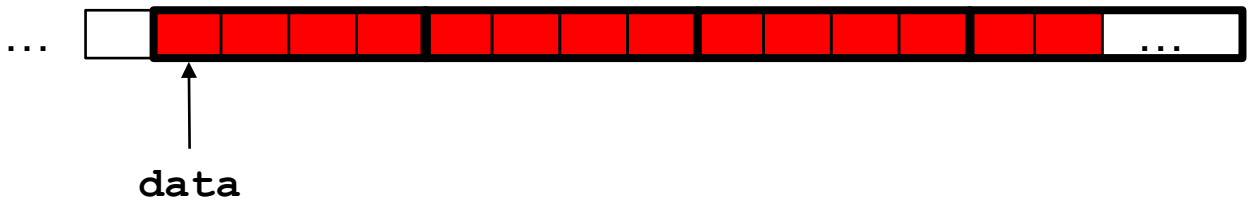
- Ein Array (Feld):
  - Ist eine Liste von Datenelementen gleichen Typs
  - Hat eine feste Länge
  - Wird im Speicher hintereinander abgelegt
  - Keine „Löcher“

Arrays (Felder)

- Ein Array (Feld):
  - Ist eine Liste von Datenelementen gleichen Typs
  - Hat eine feste Länge
  - Wird im Speicher hintereinander abgelegt
  - Keine „Löcher“
- Beispiel:

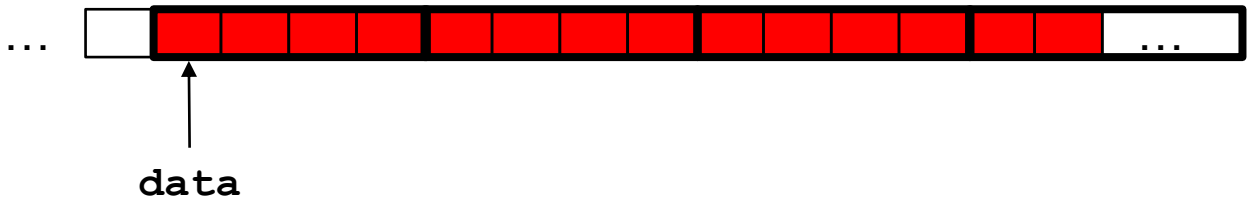
Anzahl der Elemente

```
int32_t data[10]; // ein int32_t belegt 4 Bytes => 40 Bytes
```



Arrays und Pointer

- Warum Pointervariablen?
  - Zum Speichern von Adressen, z.B: innerhalb von Funktionen, die mit Variablen via „Call by Reference“ aufgerufen werden.
- Arrayvariablen sind Pointer
  - Der Inhalt der Arrayvariable ist die Adresse des ersten Elements des Arrays



Zugriff auf Array-Elemente

- Arithmetik auf Pointern verschiebt die Adresse abhängig vom Datentyp, auf den der Pointer zeigt

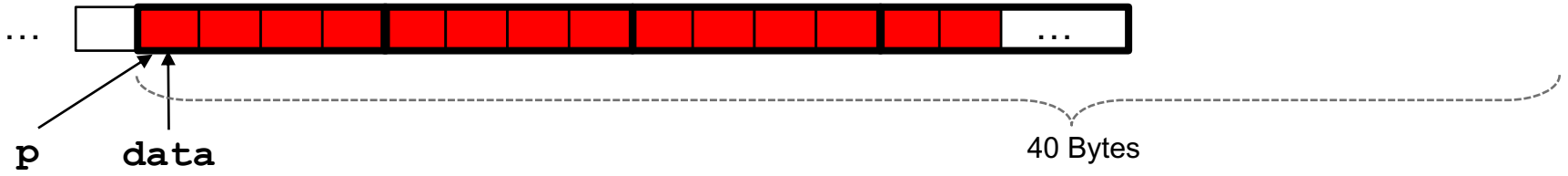
Zugriff auf Array-Elemente

- Arithmetik auf Pointern verschiebt die Adresse abhängig vom Datentyp, auf den der Pointer zeigt


```
int32_t  data[10];  // int32_t = 4 Bytes => 40 Bytes
```

Zugriff auf Array-Elemente

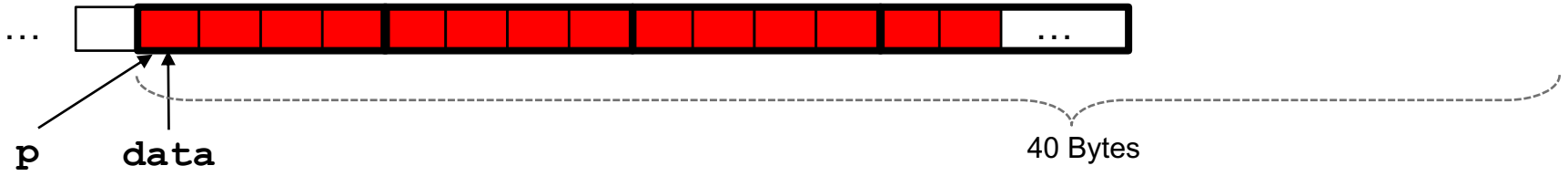
- Arithmetik auf Pointern verschiebt die Adresse abhängig vom Datentyp, auf den der Pointer zeigt




```
int32_t  data[10];  // int32_t = 4 Bytes => 40 Bytes
int32_t* p = data;  // p points to array
```

Zugriff auf Array-Elemente

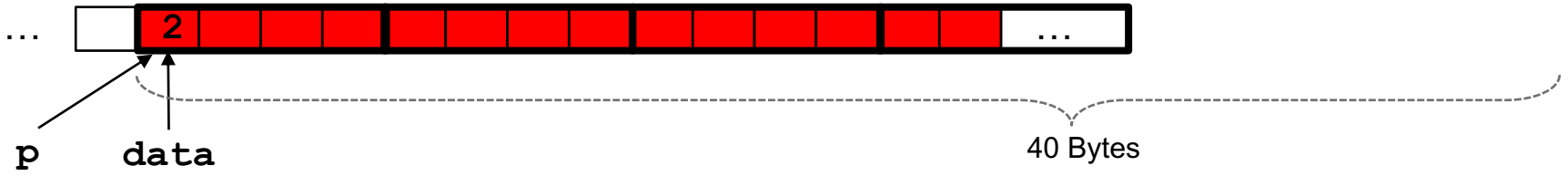
- Arithmetik auf Pointern verschiebt die Adresse abhängig vom Datentyp, auf den der Pointer zeigt




```
int32_t  data[10];  // int32_t = 4 Bytes => 40 Bytes
int32_t* p = data;  // p points to array
int32_t* q;
```

Zugriff auf Array-Elemente

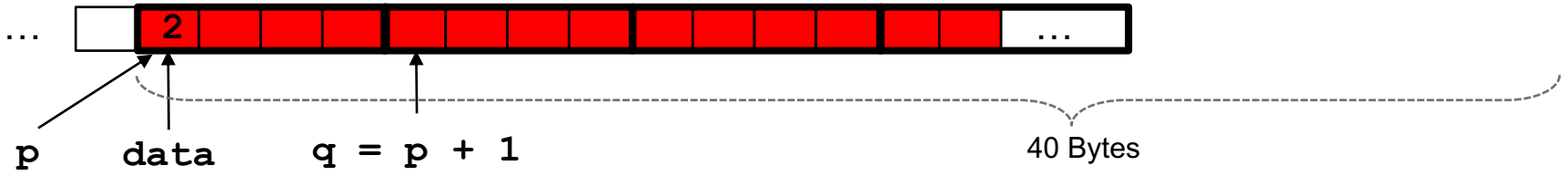
- Arithmetik auf Pointern verschiebt die Adresse abhängig vom Datentyp, auf den der Pointer zeigt




```
int32_t  data[10];  // int32_t = 4 Bytes => 40 Bytes
int32_t* p = data;  // p points to array
int32_t* q;
*p = 2;
```

Zugriff auf Array-Elemente

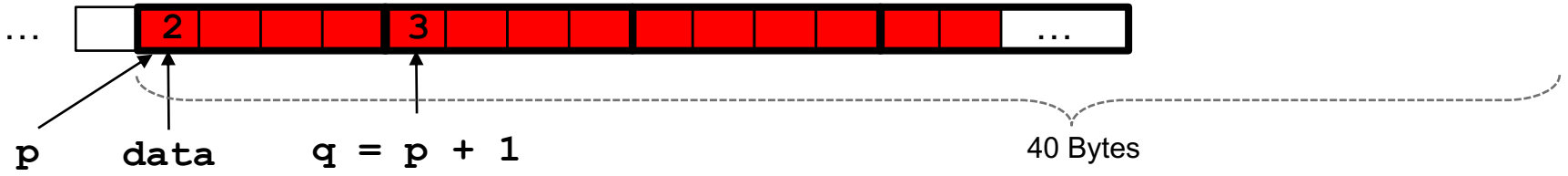
- Arithmetik auf Pointern verschiebt die Adresse abhängig vom Datentyp, auf den der Pointer zeigt




```
int32_t  data[10];  // int32_t = 4 Bytes => 40 Bytes
int32_t* p = data;  // p points to array
int32_t* q;
*p = 2;
q = p + 1;     // "+ 1" means:
               // "+ (1 * element size)" => + 4 Bytes
```

Zugriff auf Array-Elemente

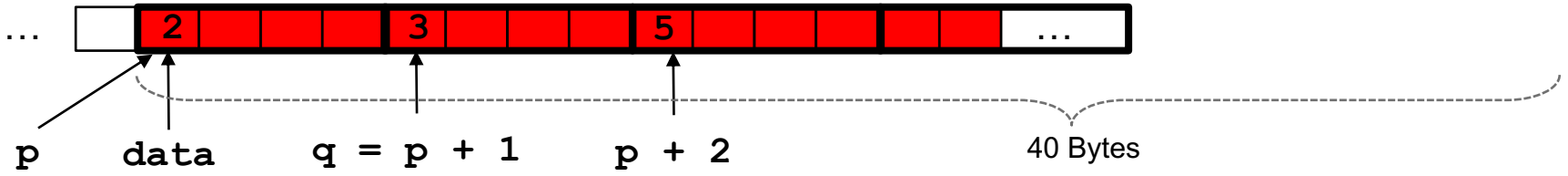
- Arithmetik auf Pointern verschiebt die Adresse abhängig vom Datentyp, auf den der Pointer zeigt




```
int32_t  data[10];  // int32_t = 4 Bytes => 40 Bytes
int32_t* p = data;  // p points to array
int32_t* q;
*p = 2;
q = p + 1;     // "+ 1" means:
               // "+ (1 * element size)" => + 4 Bytes
*q = 3;
```

Zugriff auf Array-Elemente

- Arithmetik auf Pointern verschiebt die Adresse abhängig vom Datentyp, auf den der Pointer zeigt




```
int32_t  data[10];  // int32_t = 4 Bytes => 40 Bytes
int32_t* p = data;  // p points to array
int32_t* q;
*p = 2;
q = p + 1;     // "+ 1" means:
               // "+ (1 * element size)" => + 4 Bytes
*q = 3;
*(p + 2) = 5; // + 2 elements !!!
```

Einfacher: Index

- Zugriff auf Arrayelemente mit Index in [ ]
- Erstes Element hat den Index „0“, d.h. [0] !

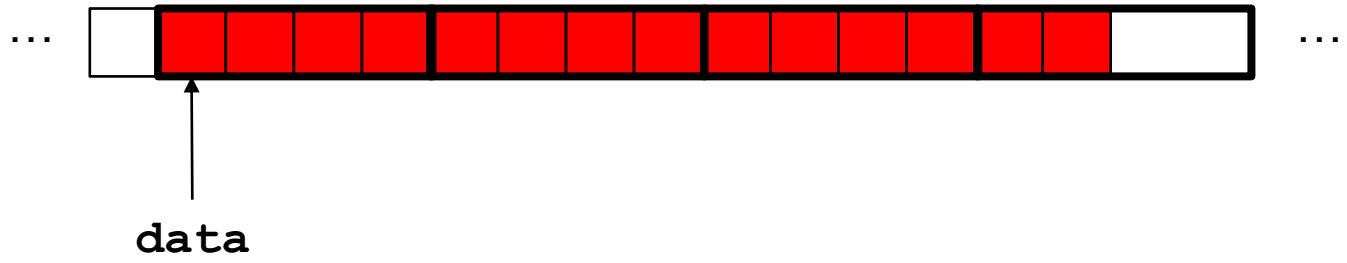
- Beispiel:

```
int32_t data[10];  // Integer belegt 4 Bytes => 40 Bytes

data[0]  = 5;   // first element of array
data[1]  = 4;   // second element of array
data[2]  = 6;   // third element of array
data[9]  = 7;   // 10th, i.e, last element of array
data[10] = 8;   // error: no 11th element exists
...
```

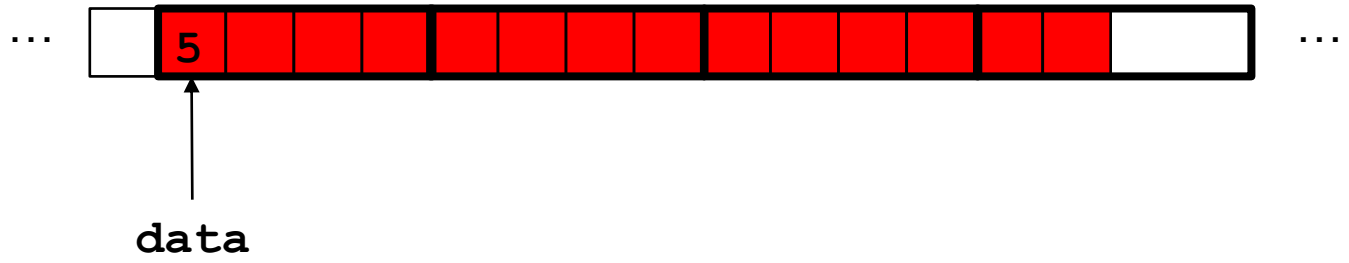
Index: Beispiel

Index: Beispiel




```
int32_t data[10];
```

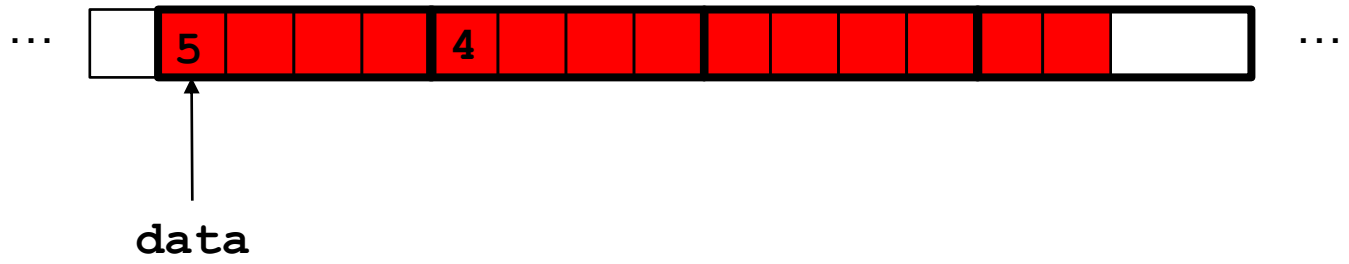
Index: Beispiel




```
int32_t data[10];

data[0] = 5;
```

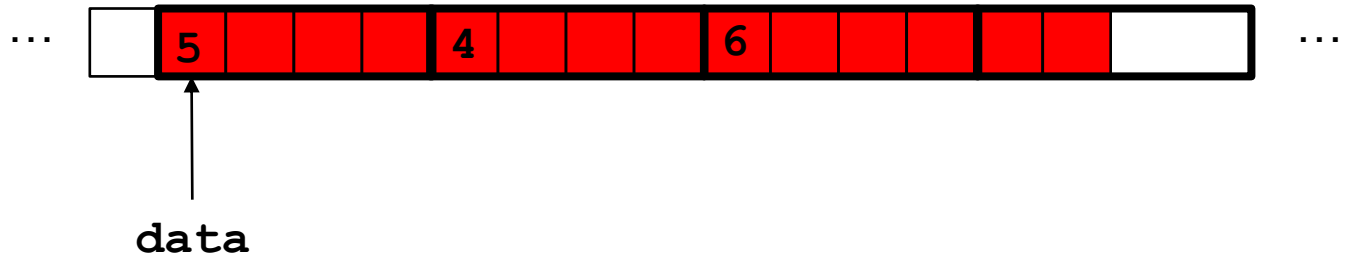
Index: Beispiel




```
int32_t data[10];

data[0] = 5;
data[1] = 4;
```

Index: Beispiel

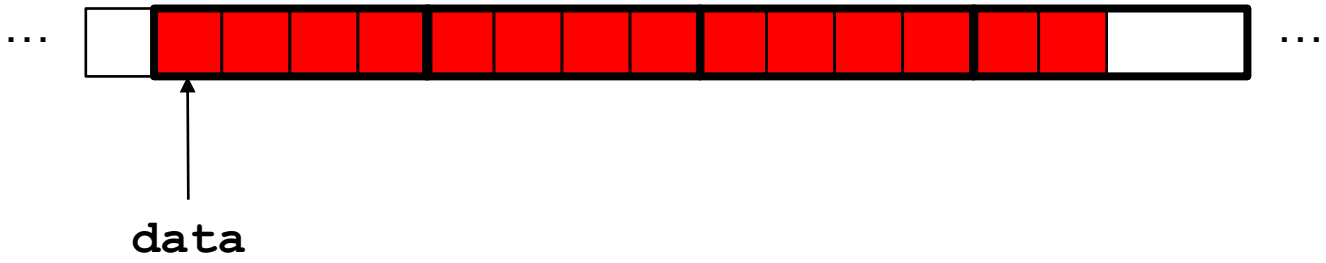



```
int32_t data[10];

data[0] = 5;
data[1] = 4;
data[2] = 6;
```

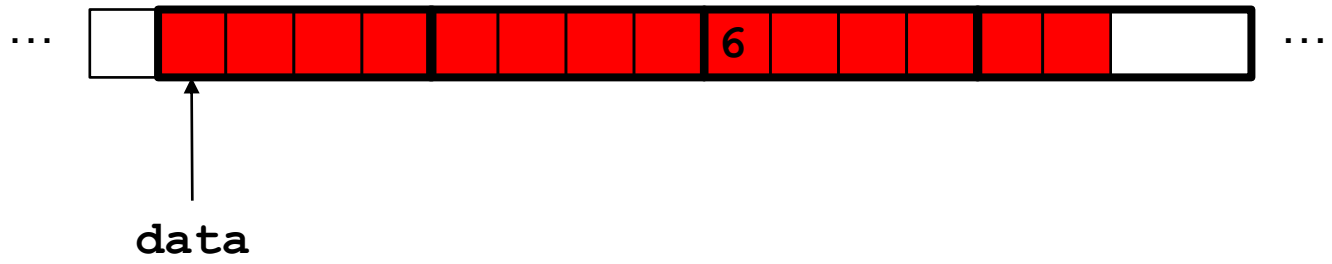
Zugriff in beliebiger Reihenfolge

Zugriff in beliebiger Reihenfolge




```
int32_t data[10];
```

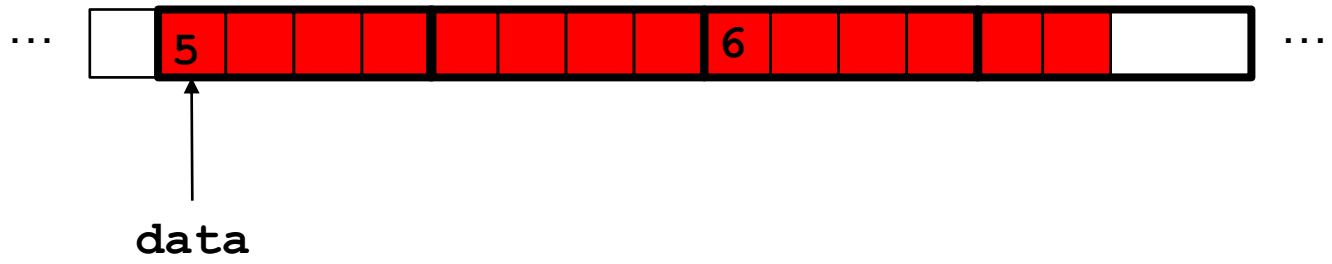
Zugriff in beliebiger Reihenfolge




```
int32_t data[10];

data[2] = 6;   // Zugriff in
```

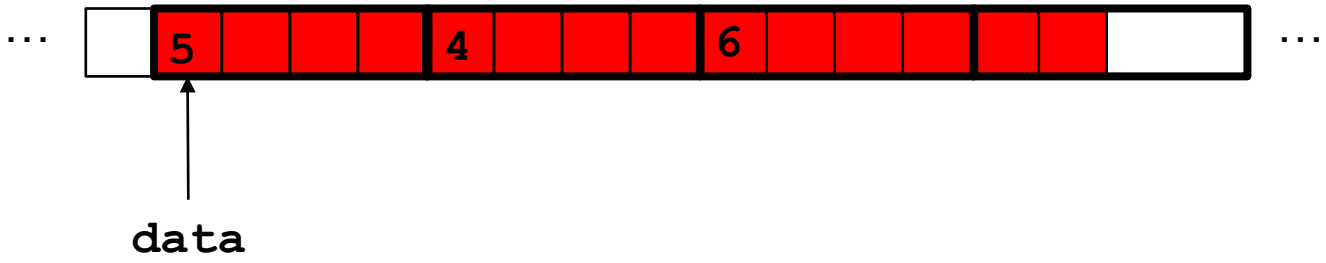
Zugriff in beliebiger Reihenfolge




```
int32_t data[10];

data[2] = 6;    // Zugriff in
data[0] = 5;    // beliebiger
```

Zugriff in beliebiger Reihenfolge



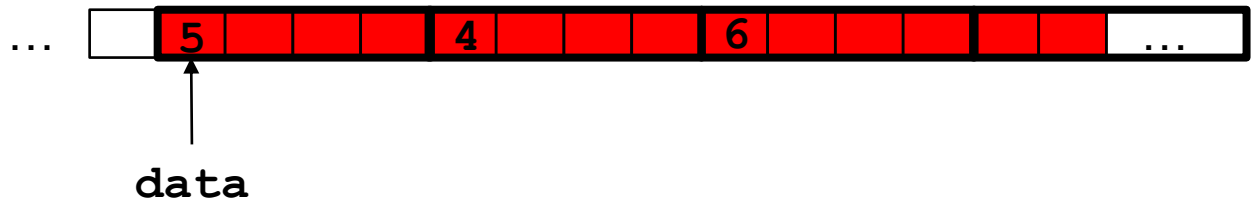

```
int32_t data[10];

data[2] = 6;    // Zugriff in
data[0] = 5;    // beliebiger
data[1] = 4;    // Reihenfolge
```

Arrays vs. Pointer

- Arrayvariablen sind Pointer
- Der Inhalt der Arrayvariable ist die Adresse des ersten Elements des Arrays, d.h. `&data[0]`

  `p = data` ist gleichbedeutend mit `p = &data[0]`
- Arrayindizes sind Offsets und geben den Abstand zum Arrayanfang an.



Beispiel: Arrays vs. Pointer

```
int data[] = {2, 3, 5, 7, 9};  // easy initialization!
        // data[0] = 2, data[1] = 3, data[2] = 5, data[3] = 7, data[4] = 9
int *p;
p = data;
printf("Content of p: %p == address of data: %p\n", p, data);
printf("data[0]: %d == *p: %d\n", data[0], *p);
printf("data[2]: %d == *(p + 2): %d\n", data[2], *(p+2));
```

Ausgabe (Pointer sind willkürlich gewählte Adressen):

```
Content of p: 0x7edf04ec == address of data: 0x7edf04ec
data[0]: 2 == *p: 2
data[2]: 5 = *(p + 2): 5
```

Initialisierung von Arrays

- Arrays können, aber müssen nicht initialisiert werden

```
// Deklaration Array mit Speicherplatz für 5 Integer
int32_t a[5];

/* Deklaration mit Initialisierung
   Array mit Speicher für 5 Integer */
int32_t arr[] = { 1, 8, 7, -1, 2 };
```

- Bei der Arraydeklaration wird der notwendige Speicherplatz automatisch reserviert!

Durchlaufen von Arrays

- Es ist möglich alle Arrayelemente mittels einer Schleife zu durchlaufen:

```
for (int i = 0; i < array_length; i++) {
    printf("element mat[%d] = %d\n", i, mat[i]);
}
```

- Es gibt beim Zugriff keinerlei Überprüfungen auf Grenzen des Arrays!

Mehrdimensionale Arrays

- Arrays können mehrere Dimensionen haben (Vektoren, Matrizen, …).

- Arrays werden elementweise und Zeile für Zeile hintereinander abgespeichert.

Beispiele:

```
int8_t mat[3][2] = { {00, 01}, {10, 11}, {20, 21} };
int8_t s;
// alternativ: int8_t mat[3][2] = {00, 01, 10, 11, 20, 21};
s = mat[0][1];  // s == 01
s = mat[1][1];  // s == 11
s = mat[2][0];  // s == 20
```

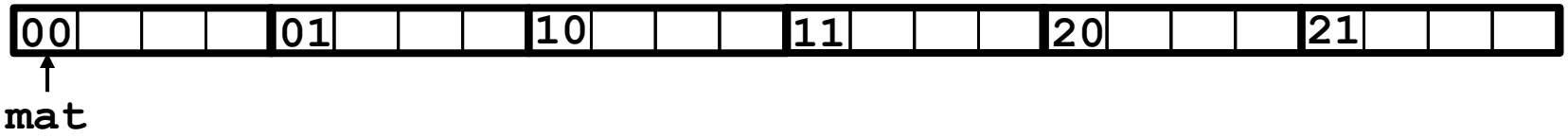
2-dimensionales Array visualisiert

```
int8_t mat[3][2] = { {00, 01}, {10, 11}, {20, 21} };
```

Konzeptuell:

00	01
10	11
20	21

Im Speicher tatsächlich:



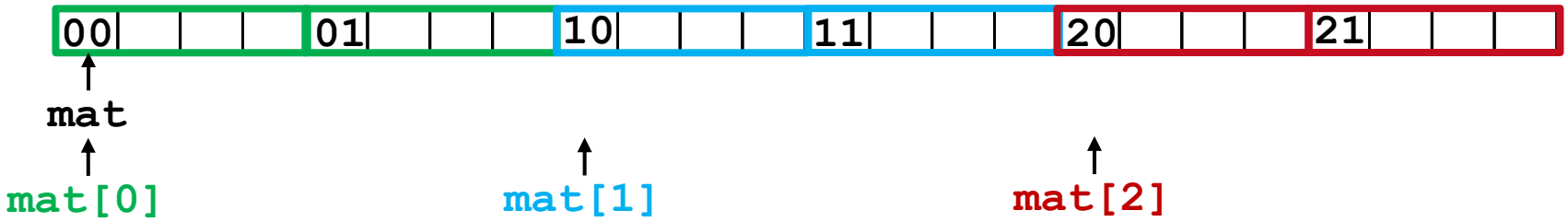
2-dimensionales Array visualisiert

```
int8_t mat[3][2] = { {00, 01}, {10, 11}, {20, 21} };
```

Konzeptuell:

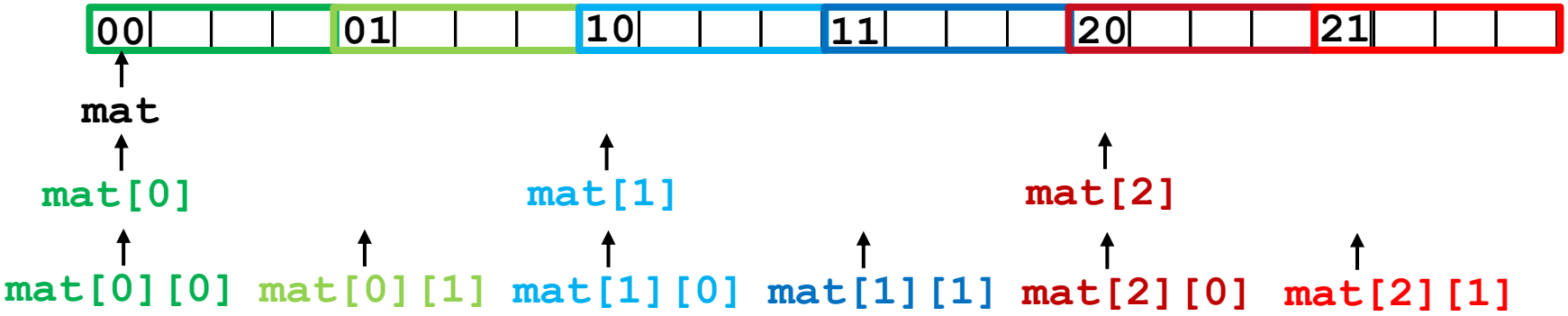
00	01
10	11
20	21

Im Speicher tatsächlich:



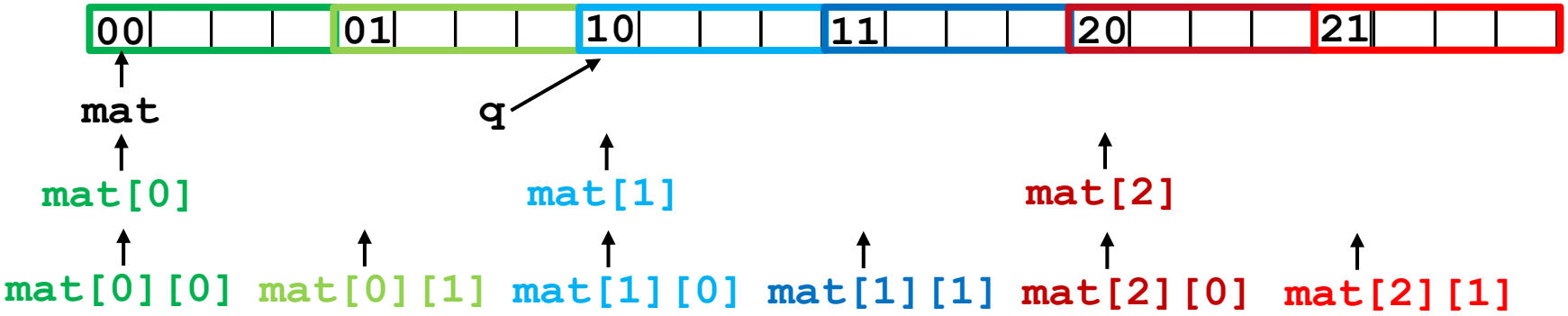
2-dimensionales Array visualisiert

```
int8_t mat[3][2] = { {00, 01}, {10, 11}, {20, 21} };
```



2-dimensionales Array visualisiert

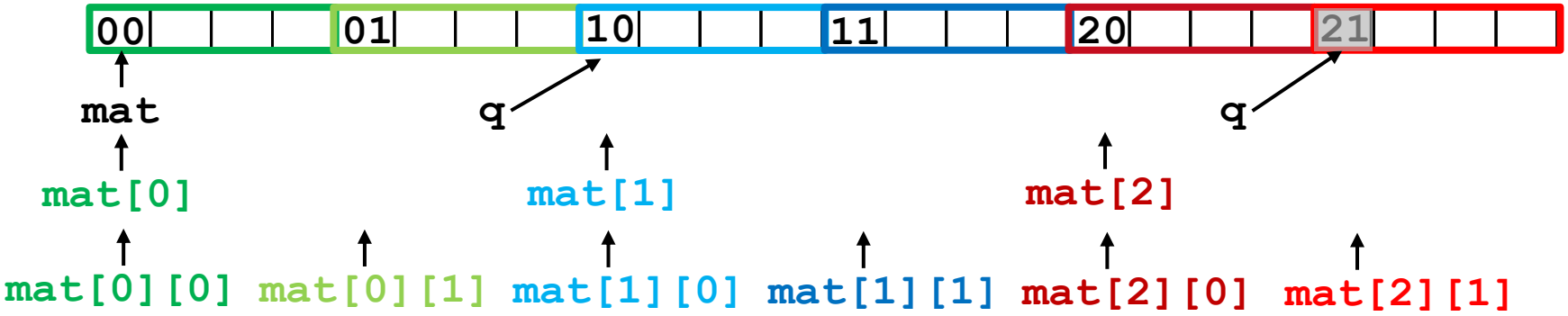
```
int8_t mat[3][2] = { {00, 01}, {10, 11}, {20, 21} };
```




```
int8_t *q  = mat[1];     // 1-dim int8_t array !!
```

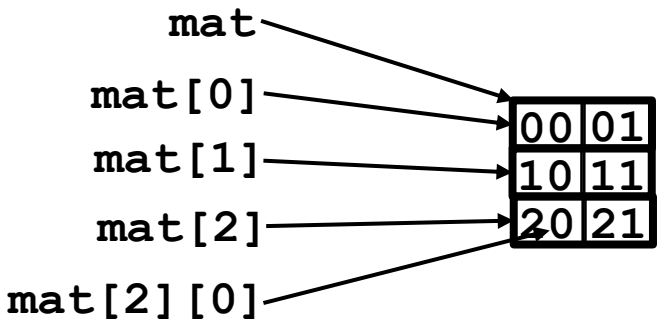
2-dimensionales Array visualisiert

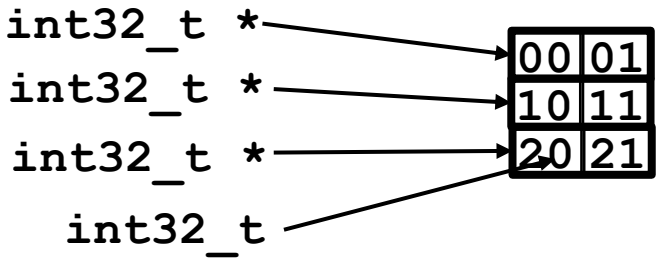
```
int8_t mat[3][2] = { {00, 01}, {10, 11}, {20, 21} };
```




```
int8_t *q  = mat[1];    // 1-dim int8_t array !!
int8_t w   = mat[2][1]; // w == 21
```

2-dimensionales Array visualisiert





Was ist bei Arrays zu beachten?

- Ein Array (Feld):
  - Ist eine Liste von Daten gleichen Typs.
  - Hat eine feste Länge.

- Array Definition reserviert den vorgegebenen Speicherplatz automatisch.

- Zugriff auf Arrayelemente via `[index]`
  - Es wird nicht überprüft, ob `index` innerhalb des reservierten Speicherplatzes des Arrays liegt. Somit kann unbeabsichtigt anderer Speicher ausgelesen oder überschrieben werden!
  - Manchmal wird allerdings geprüft (lernt man später) ⇒ Programmabsturz

Arrays und Funktionen

- Arrays werden mit Adresse an Funktionen übergeben, d.h. mittels Call-by-Reference

  Call-by-Value gibt es für Arrays nicht!

```
// summarize 'n' array elements and return sum in parameter 'sum'
int32_t sum(int32_t a[], int32_t n, int32_t *sum) {
  *sum = 0;
  for (int32_t i = 0; i < n; i++) {
     /* sum is the pointer to the address where
        we store the sum of the array elements */
     *sum += a[i];   // *sum = *sum + a[i] or *sum += *(a+i);
  }
}

int main() {
  int s = 0, a[] = {2, 3, 5, 7, 11};
  sum(a, 5, &s);
  printf("Die Summe der Arrayaelemente ist: %d\n", s);
}
```

Ausblick

VL 0 „Organisation und Inhalt“: Ablauf der Vorlesung, Termine
VL 1 „Hello World“: „Lebenswichtiges“, Programablauf, Programmierablauf, Kompilierung und Ausführung von Programmen
VL 2 „Die ersten Schritte“: Erstes C-Programm, Elementare C-Strukturen, Datentypen, Operatoren, Schleifen
VL 3 „Kontrollstrukturen & Funktionen“: Syntax, Semantik, bedingte Anweisungen, Blöcke, Sichtbarkeit
VL 4 „Rekursive Funktionen & Bibliotheken“: rekursive Funktionsaufrufe, Modularisierung
VL 5 „Typen“: Einfache und strukturierte Datentypen, Wertebereiche, Typendefinition
VL 6 „Speicher und Adressen“: Speicher, Pointer, Funktionsaufrufe „call by value“ vs. „call by reference“
VL 7 „Speicher und Arrays“: Speicher, Arrays, mehrdimensionale Arrays, Arrays und Pointer
VL 8 „Dynamische Speicherverwaltung“: Speicherallokation, Fehlerbehandlung, Rückgabewerte, Arrays/Pointer/Adressen
VL 9 „Strings, Kanäle, Git“: Strings und Arrays, Zeichensätze, Stringlänge, Ein- und Ausgabe, Arbeiten mit git