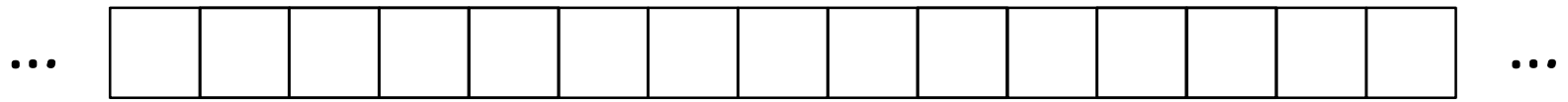
C-Kurs
Arrays und Adressen

Speicher

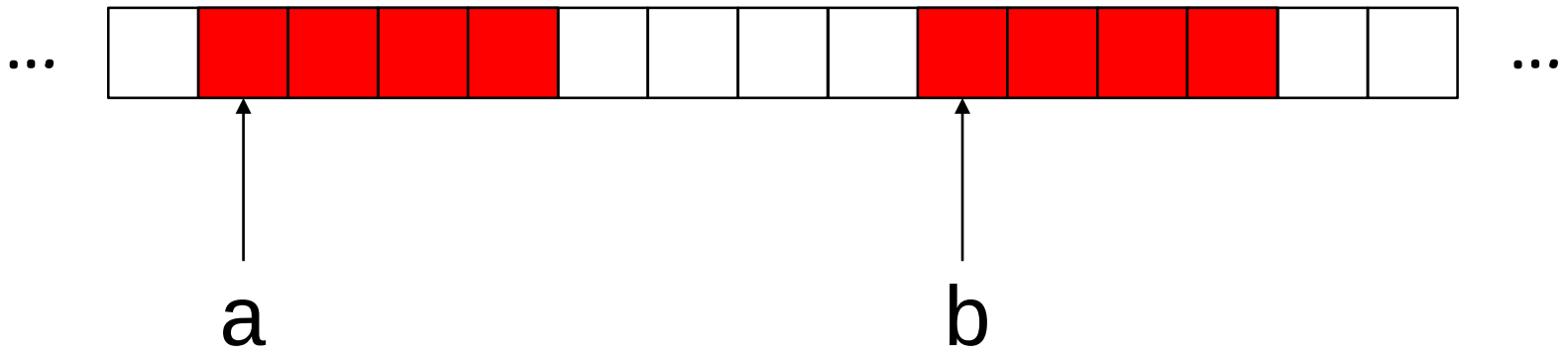
Speicher

- Speicher besteht aus einer Reihe von Bytes



Variablen im Speicher

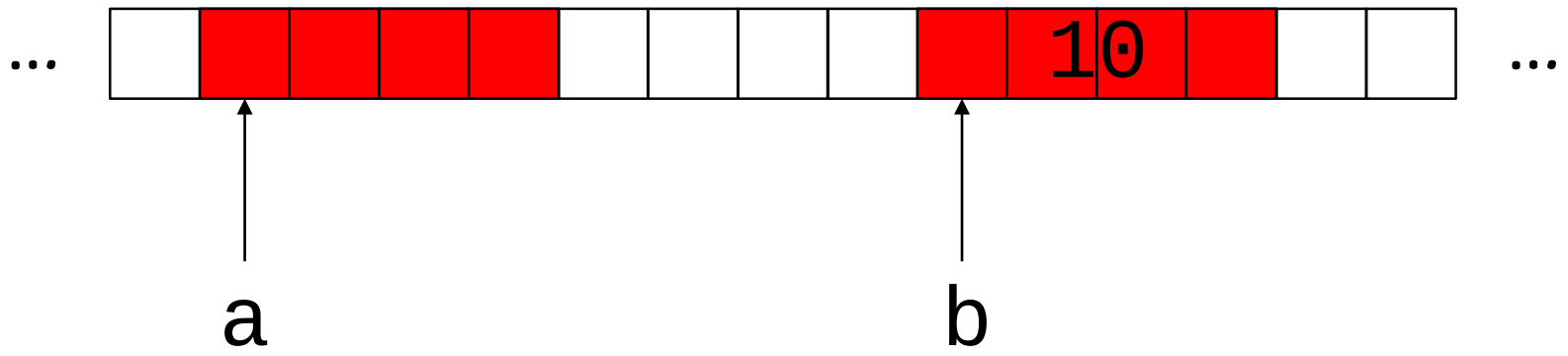
- Speicher besteht aus einer Reihe von Bytes
- Integers werden in 4 aufeinanderfolgenden Bytes gespeichert




```
int a;
int b;
```

Variablen im Speicher

- Deklaration einer Variablen entspricht Speicherreservierung
- Zuweisung entspricht Belegung des Speichers mit dem Wert




```
b = 10;
```

Unsere erste Datenstruktur

Felder / Arrays

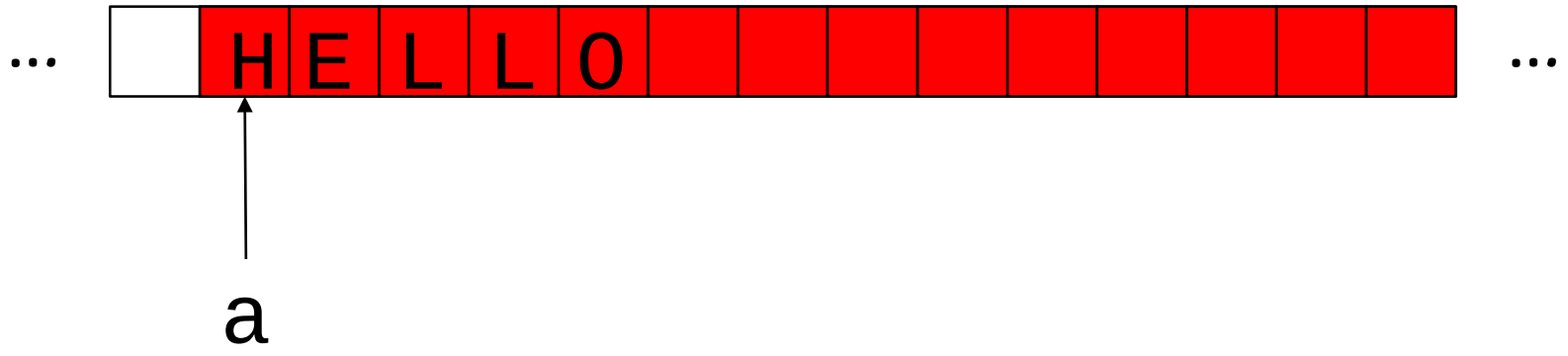
Motivation für Datenstrukturen

- 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

Erste Datenstruktur: Arrays

- Ein Array (Feld):
  - Ist eine Liste von Daten gleichen Typs
  - Hat eine feste Länge
  - Zugriff auf Arrayelemente mit Index in `[]`
- Beispiele:

```
char a[128];   // char belegt 1 Byte
a[0] = 'H';
a[1] = 'E';
a[2] = 'L';
a[3] = 'L';
a[4] = 'O';
```




```
char a[128];
a[0] = ‘H‘;
a[1] = ‘E‘;
a[2] = ‘L‘;
a[3] = ‘L‘;
a[4] = ‘O‘;
```

Arrays

- Ein Array (Feld):
  - Ist eine Liste von Daten gleichen Typs
  - Hat eine feste Länge
  - Zugriff auf Arrayelemente mit Index in `[]`
- Beispiele:

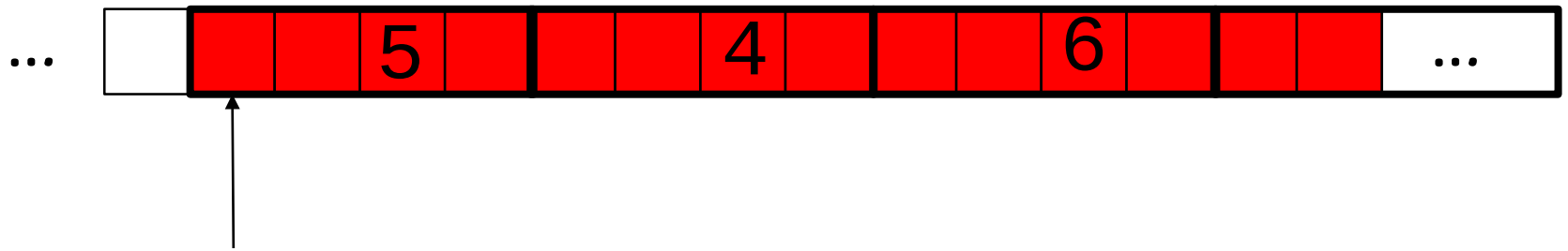
```
char a[128];
a[0] = 'H';    // Arrayindexanfang 0!
a[1] = 'E';
a[2] = 'L';
a[3] = 'L';
a[4] = 'O';
```

- Kommentar: Speicher in Maßen reservieren
  Nicht genug Speicher: Problem!!!

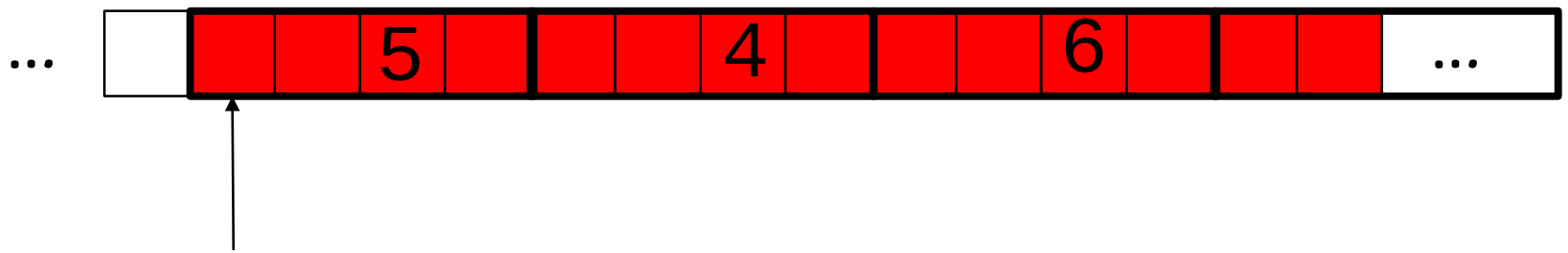
Arrays (2)

- Ein Array (Feld):
  - Ist eine Liste von Daten gleichen Typs
  - Hat eine feste Länge
  - Zugriff auf Arrayelemente mit Index in `[]`
- Beispiele:

```
int data[10];    // Integer belegt 4 Bytes
data[0] = 5;
data[1] = 4;
data[2] = 6;
...
```




```
int data[10];
data[0] = 5;
data[1] = 4;
data[2] = 6;
```




```
int data[10];
data[2] = 6;    // Zugriff in
data[0] = 5;    // beliebiger
data[1] = 4;    // Reihenfolge
```

Arrays (3)

- Arrays können, aber müssen nicht initialisiert werden

- Beispiele:

```
char a[128];     // Deklaration

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

- Hinweis:
  Wie bei anderen Variablen wird bei der Arraydefinition der notwendige Speicherplatz automatisch reserviert!

Arrays (3)

- Zugriff auf Arrayelemente mit Index in `[]`:

```
char c;
c = a[32];
a[0] = 'A';
```

- Die Arrayindizierung beginnt immer mit 0!

```
int s, mat[] = {2, 3, 5};
s = mat[0];      // s == 2
```

- Arrays werden elementweise hintereinander abgespeichert

- Hinweis:
  Es gibt beim Zugriff keinerlei Überprüfungen auf Bereichsgrenzen der Arrays!

Arrays (4)

- Zugriff auf Arrayelemente mit Index in []:

```
int mat[] = {2, 3, 5, 7, 10, 14};
int mat_len = 6;
```

- Arrays werden elementweise hintereinander abgespeichert
- Es ist möglich alle Arrayelemente mittels einer Schleife zu durchlaufen

```
for (int i = 0; i < mat_len; i++) {
    printf(„elem mat[%d] = %d\n“,i,mat[i]);
};
```

- Hinweis:
  Es gibt beim Zugriff keinerlei Überprüfungen auf Bereichsgrenzen der Arrays!

Arrays (5)

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

- Beispiele:

```
int mat[2][2] = { 11, 12, 21, 22};
 s = mat[0][0];     // s == 11
```

- Arrays werden elementweise und Zeile für Zeile hintereinander abgespeichert

Zusammenfassung: 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
  - Hinreichend, aber nicht übermäßig Speicherplatz reservieren
- Zugriff auf Arrayelemente via `[index]`
  - Es wird nicht überprüft, ob `index` innerhalb des reservierten Platz des Arrays liegt
  - Somit kann unbeabsichtigt anderer Speicher ausgelesen oder überschrieben werden

Adressen von Variablen

Was ist das & vor Variablennamen?

Variablen (Wiederholung)

- Variablen
  - Sind Platzhalter für Daten
  - Geben somit Daten einen „Namen“
  - Haben einen festgelegten Speicherort, wo der aktuelle Wert gespeichert wird
  - Der aktuelle Wert ist veränderbar
- Beispiel:

```
1. int x;
2. x = 5;
3. int y;
4. x = 6;
```

Zustand	x			
Zustand	x	5		
Zustand	x	5	y	
Zustand	x	6	y	

Adressen von Variablen

- Adressen von Variablen
  - Der Ort an dem Daten gespeichert sind
  - Kann sich somit nicht ändern!
  - Zugriff auf die Adresse mittels: &
  - Adressen sind vom Typ unsigned int
- Beispiel:

```
1. int x = 5, y = 3;
2. printf("The value of x is %d\n", x);
3. printf("Addresses of x and y are %u %u\n",
   &x, &y);
4. x = 6;
5. printf …
```

Zustand	x	5		
Zustand	x	5	y	3
Zustand	x	6	y	3

Funktionsaufrufe und Parameterübergabe

Call by Value: Korrektes Beispiel

- Beispiel:

```
// add computes the sum of a, b and returns the sum
int add1 (int a, int b){ // a,b are passed by value
   return a + b;
}

int main () {
   int x = 5, y = 3, sum = 0;
   sum = add1(x, y);
   printf("The value of sum is %d\n", sum);
}
```

Call by Value: Falsches Beispiel

- Beispiel – falsch:

```
// add computes the sum of a, b in s and the value
// of s should be available to calling functions
// 1st attempt: s, a, b are passed by value
void add2 (int s, int a, int b) {
   s = a + b;
}
int main () {
   int x = 5, y = 3, sum = 0;
   add2(sum, x, y);
   printf("The value of sum is %d\n", sum);
}
```

Call by Reference: Korrektes Beispiel

- Beispiel – korrekt:

```
// add computes the sum of a, b in s and the value
// of s should be available to calling functions
// 2nd attempt: a, b are passed by value
//              s is passed by reference as address
void add3 (int *s, int a, int b) {
   *s = a + b;
}
int main () {
   int x = 5, y = 3, sum = 0;
   add3(&sum, x, y);// pass address of sum to add
   printf("The value of sum is %d\n", sum);
}
```

Parameterübergabe an Funktionen

- Call by Value
  - Parameterübergabe als Wert
  - Werte der Variablen werden übergeben
  - Damit stehen die Werte der Variablen als lokale Kopie zur Verfügung
  - Konsequenz: Änderungen nur sichtbar innerhalb der Funktion
- Call by Reference
  - Parameterübergabe als Adresse
  - Adressen der Variablen werden übergeben
  - Damit steht die Adresse lokal zur Verfügung und Zugriff auf den Speicherort der übergebenen Variablen möglich
  - Konsequenz: Änderungen sichtbar über die Funktion hinaus

Adressvariablen (Pointer)

Was macht das * vor sum?

Adressen / Pointer

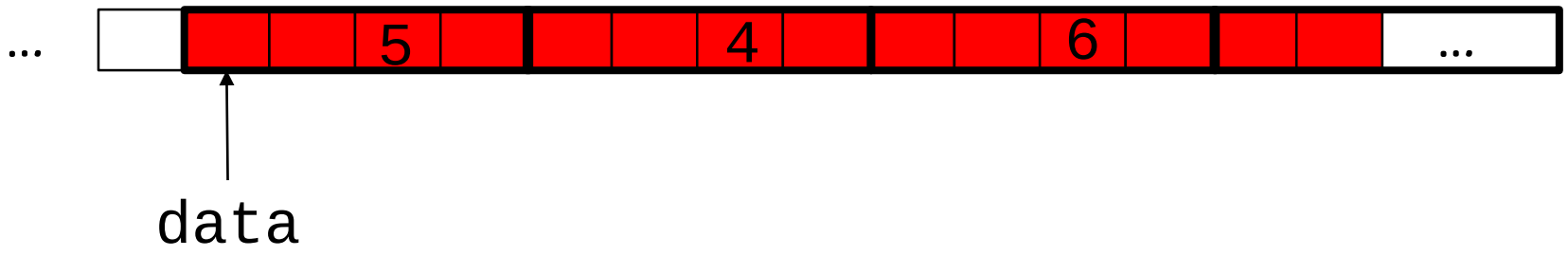
- Ein weiterer Datentyp: Adressen
- Variablen, die Adressen von Variablen speichern:
  - Syntax: *
  - Warum die Syntax *
    - Brauchen Adressen von Integer, Floats, Chars, etc.
    - Entsprechend gibt es int *, float *, ...

- Beispiel:

```
void add3(int *s, int a, int b) {
  *s = a + b;
}
```

Adressen / Pointer

- Warum Pointervariablen
  - Zum Speichern von Adressen, z.B: innerhalb von Funktionen, die mit Variablen via Call by Reference aufgerufen werden
  - Für Arrays
    - Arrayvariablen sind Pointer
    - Der Inhalt der Arrayvariable ist die Adresse des ersten Elements des Arrays (d.h. `&data[0]`)
    - Arrayindizes sind Offsets, sie geben den Abstand zum Arrayanfang an



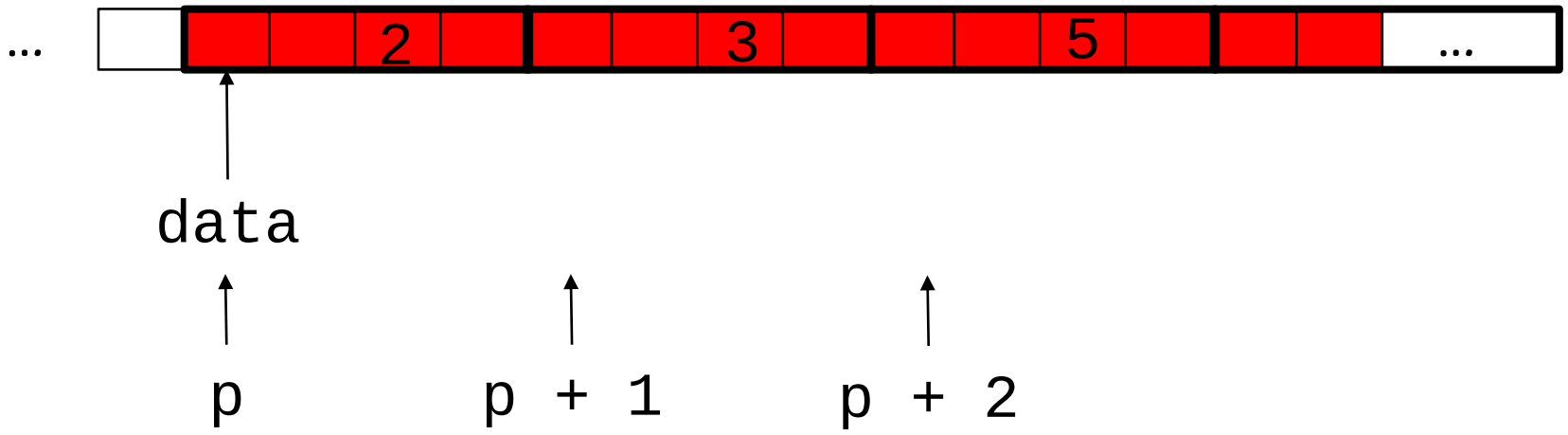
Arrays und Pointer

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

```
int data[] = {2, 3, 5, 7, 9};
int *p;
p = data;
printf(„Content of p %u == address of data %u\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));
```

Adressen / Pointer

- Arithmetik auf Pointern verschiebt die Adresse




```
printf(„data[2] %d == *(p + 2) %d\n“, data[2], *(p+2));
int *q;     // q is pointer variable of type int *
q = p+2;    // q is assigned the address as p+2
printf(„data[2] %d == *q %d\n“, data[2], *q );
```

Arrays und Funktionen

- Arrays werden mit Adresse an Funktionen übergeben, d.h. mittels Call-by-Reference
  (Call-by-Value gibt es für Arrays nicht!)

```
void sum(int a[] , int n, int *s) {
    *s = 0;
    for (int i = 0; i < n; i++) {
          // s is the pointer to the address where
          // we store the sum of the array elements
      *s += a[i]; // *s += *(a+i);
}}
int main() {
   int s = 0, a[] = {2, 3, 5, 7, 11};
   sum(a, 5, &s);
}
```