

# Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

**13 Zeiger und Felder**

**14  $\mu$ C-Systemarchitektur**



# Einordnung: Zeiger (*Pointer*)

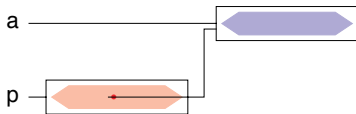
- **Literal:** 'a'  
Darstellung eines Wertes

'a'  $\equiv$  

- **Variable:** `char a;`  
Behälter für einen Wert



- **Zeiger-Variable:** `char *p = &a;`  
Behälter für eine Referenz  
auf eine Variable



# Zeiger (*Pointer*)

- Eine Zeigervariable (*Pointer*) enthält als Wert die **Adresse** einer anderen Variablen
  - Ein Zeiger verweist auf eine Variable (im Speicher)
  - Über die Adresse kann man **indirekt** auf die Zielvariable (ihren Speicher) zugreifen
- Daraus resultiert die große Bedeutung von Zeigern in C
  - Funktionen können Variablen des Aufrufers verändern (*call-by-reference*)
  - Speicher lässt sich direkt ansprechen
  - Effizientere Programme
- Aber auch viele Probleme!
  - Programmstruktur wird unübersichtlicher (welche Funktion kann auf welche Variablen zugreifen?)
  - Zeiger sind die **häufigste Fehlerquelle** in C-Programmen!

↪ 9-5

„Effizienz durch  
Maschinennähe“

↪ 3-14



# Definition von Zeigervariablen

- **Zeigervariable** := Behälter für Verweise ( $\mapsto$  Adresse)
- Syntax (Definition):  $Typ * Bezeichner ;$
- Beispiel

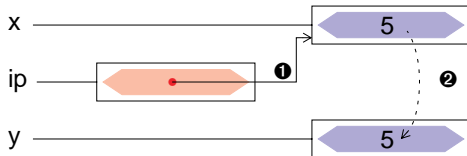
```
int x = 5;
```

```
int *ip;
```

```
int y;
```

```
ip = &x; ❶
```

```
y = *ip; ❷
```



# Adress- und Verweisoperatoren

- Adressoperator:  $\&x$  Der unäre  $\&$ -Operator liefert die **Referenz** ( $\mapsto$  Adresse im Speicher) der Variablen  $x$ .
- Verweisoperator:  $*y$  Der unäre  $*$ -Operator liefert die **Zielvariable** ( $\mapsto$  Speicherzelle / Behälter), auf die der Zeiger  $y$  verweist (Dereferenzierung).
- Es gilt:  $(*(&x)) \equiv x$  Der Verweisoperator ist die Umkehroperation des Adressoperators.

**Achtung:** Verwirrungsgefahr (\*\**Ich seh überall Sterne*\*\*)

Das  $*$ -Symbol hat in C verschiedene Bedeutungen, **je nach Kontext**

1. Multiplikation (binär):  $x * y$  in Ausdrücken
2. Typmodifizierer: `uint8_t *p1, *p2` in Definitionen und  
`typedef char* CPTR` Deklarationen
3. Verweis (unär):  $x = *p1$  in Ausdrücken

Insbesondere 2. und 3. führen zu Verwirrung

$\leadsto$   $*$  wird fälschlicherweise für ein Bestandteil des Bezeichners gehalten.

# Zeiger als Funktionsargumente

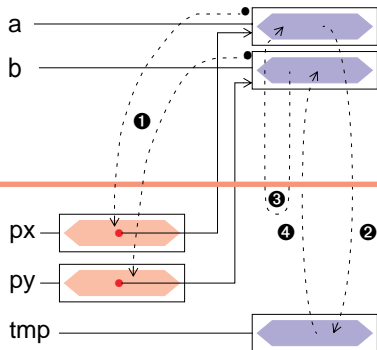
- Parameter werden in C immer *by-value* übergeben ↔ 9-5
  - Parameterwerte werden in lokale Variablen der aufgerufenen Funktion kopiert
  - Aufgerufene Funktion kann tatsächliche Parameter des Aufrufers nicht ändern
  
- Das gilt auch für Zeiger (Verweise) [↔ GDI, II-89]
  - Aufgerufene Funktion erhält eine Kopie des Adressverweises
  - Mit Hilfe des \*-Operators kann darüber jedoch auf die Zielvariable zugegriffen werden und diese verändert werden  
↳ **Call-by-reference**



## ■ Beispiel (Gesamtüberblick)

```
void swap (int *, int *);  
int main() {  
    int a=47, b=11;  
    ...  
    swap(&a, &b); ❶  
    ...  
}
```

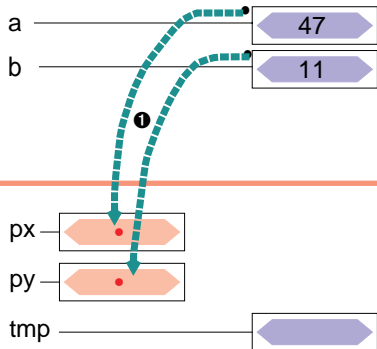
```
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px; ❷  
    *px = *py; ❸  
    *py = tmp; ❹  
}
```



## ■ Beispiel (Einzelschritte)

```
void swap (int *, int *);  
int main() {  
    int a=47, b=11;  
    ...  
    swap(&a, &b); ❶
```

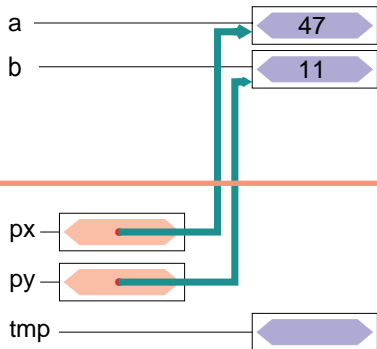
```
void swap (int *px, int *py)  
{  
    int tmp;
```



## ■ Beispiel (Einzelschritte)

```
void swap (int *, int *);  
int main() {  
    int a=47, b=11;  
    ...  
    swap(&a, &b);  
}
```

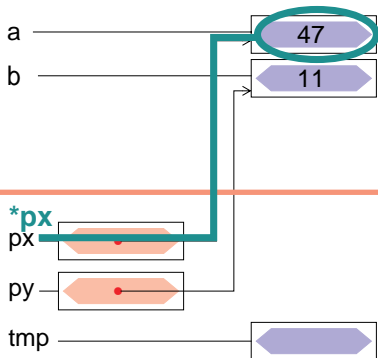
```
void swap (int *px, int *py)  
{  
    int tmp;
```



## ■ Beispiel (Einzelschritte)

```
void swap (int *, int *);  
int main() {  
    int a=47, b=11;  
    ...  
    swap(&a, &b);  
}
```

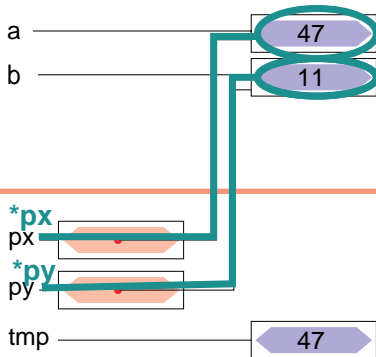
```
void swap (int *px, int *py)  
{  
    int tmp;  
    tmp = *px; ②  
}
```



## ■ Beispiel (Einzelschritte)

```
void swap (int *, int *);  
int main() {  
    int a=47, b=11;  
    ...  
    swap(&a, &b);  
}
```

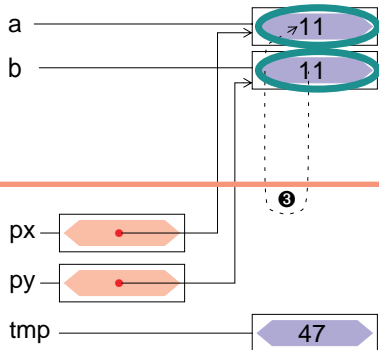
```
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px; ②  
    *px = *py; ③  
}
```



## ■ Beispiel (Einzelschritte)

```
void swap (int *, int *);  
int main() {  
    int a=47, b=11;  
    ...  
    swap(&a, &b);  
}
```

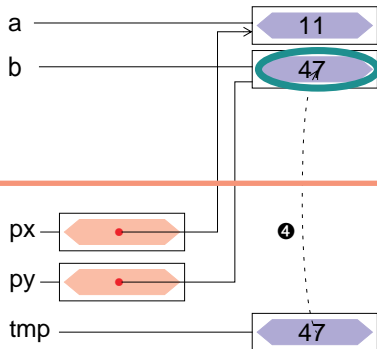
```
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px; ②  
    *px = *py; ③  
}
```



## ■ Beispiel (Einzelschritte)

```
void swap (int *, int *);  
int main() {  
    int a=47, b=11;  
    ...  
    swap(&a, &b);  
}
```

```
void swap (int *px, int *py)  
{  
    int tmp;  
  
    tmp = *px; ②  
    *px = *py; ③  
    *py = tmp; ④  
}
```



- **Feldvariable** := Behälter für eine Reihe von Werten desselben Typs
- Syntax (Definition): *Typ Bezeichner [ IntAusdruck ] ;*
  - *Typ*                      Typ der Werte                      [=Java]
  - *Bezeichner*              Name der Feldvariablen                      [=Java]
  - *IntAusdruck*            **Konstanter** Ganzzahl-Ausdruck, definiert die Feldgröße (↪ Anzahl der Elemente).                      [≠Java]  
Ab **C99** darf *IntAusdruck* bei **auto**-Feldern auch **variabel** (d. h. beliebig, aber fest) sein.
- Beispiele:

```
static uint8_t LEDs[ 8*2 ];            // constant, fixed array size

void f( int n ) {
    auto char a[ NUM_LEDS * 2];        // constant, fixed array size
    auto char b[ n ];                   // C99: variable, fixed array size
}
```



# Feldinitialisierung

- Wie andere Variablen auch, kann ein Feld bei Definition eine **initiale Wertzuweisung** erhalten

```
uint8_t LEDs[4] = { RED0, YELLOW0, GREEN0, BLUE0 };  
int prim[5]     = { 1, 2, 3, 5, 7 };
```

- Werden zu wenig Initialisierungselemente angegeben, so werden die restlichen Elemente **mit 0 initialisiert**

```
uint8_t LEDs[4] = { RED0 }; // => { RED0, 0, 0, 0 }  
int prim[5]     = { 1, 2, 3 }; // => { 1, 2, 3, 0, 0 }
```

- Wird die explizite Dimensionierung ausgelassen, so bestimmt die **Anzahl** der Initialisierungselemente die Feldgröße

```
uint8_t LEDs[] = { RED0, YELLOW0, GREEN0, BLUE0 };  
int prim[]     = { 1, 2, 3, 5, 7 };
```



# Feldzugriff

- Syntax: `Feld [ IntAusdruck ]` [=Java]
  - Wobei  $0 \leq \text{IntAusdruck} < n$  für  $n = \text{Feldgröße}$
  - **Achtung:** Feldindex wird nicht überprüft [≠Java]
    - ↪ häufige Fehlerquelle in C-Programmen
- Beispiel

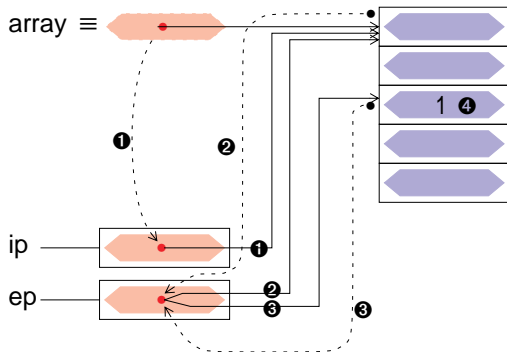
```
uint8_t LEDs[] = { RED0, YELLOW0, GREEN0, BLUE0 };  
LEDs[ 3 ] = BLUE1;  
for( uint8_t i = 0; i < 4; ++i ) {  
    sb_led_on( LEDs[ i ] );  
}  
LEDs[ 4 ] = GREEN1;    // UNDEFINED!!!
```



# Felder sind Zeiger

- Ein Feldbezeichner ist **syntaktisch äquivalent** zu einem konstanten Zeiger auf das erste Element des Feldes: `array`  $\equiv$  `&array[0]`
  - Ein Alias – kein Behälter  $\rightsquigarrow$  Wert kann nicht verändert werden
  - Über einen so ermittelten Zeiger ist ein indirekter Feldzugriff möglich
- Beispiel (Gesamtüberblick)

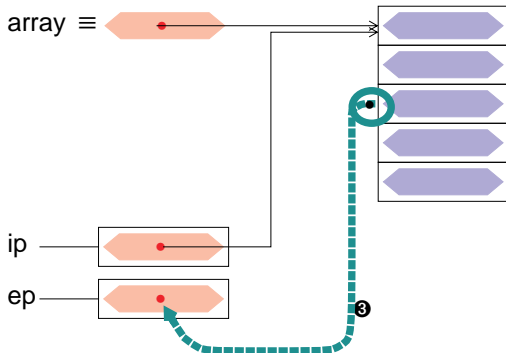
```
int array[5];  
  
int *ip = array; ①  
  
int *ep;  
ep = &array[0]; ②  
  
ep = &array[2]; ③  
  
*ep = 1; ④
```



# Felder sind Zeiger

- Ein Feldbezeichner ist **syntaktisch äquivalent** zu einem konstanten Zeiger auf das erste Element des Feldes: `array`  $\equiv$  `&array[0]`
  - Ein Alias – kein Behälter  $\rightsquigarrow$  Wert kann nicht verändert werden
  - Über einen so ermittelten Zeiger ist ein indirekter Feldzugriff möglich
- Beispiel (Einzelschritte)

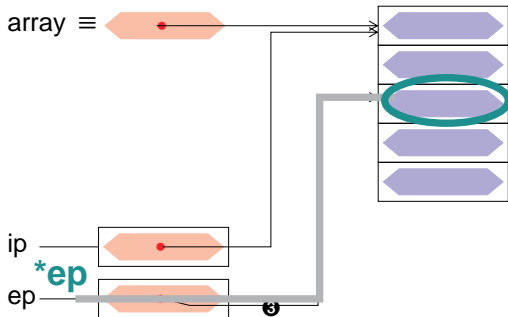
```
int array[5];  
  
int *ip = array; ①  
  
int *ep;  
ep = &array[0]; ②  
  
ep = &array[2]; ③
```



# Felder sind Zeiger

- Ein Feldbezeichner ist **syntaktisch äquivalent** zu einem konstanten Zeiger auf das erste Element des Feldes: `array`  $\equiv$  `&array[0]`
  - Ein Alias – kein Behälter  $\rightsquigarrow$  Wert kann nicht verändert werden
  - Über einen so ermittelten Zeiger ist ein indirekter Feldzugriff möglich
- Beispiel (Einzelschritte)

```
int array[5];  
  
int *ip = array; ①  
  
int *ep;  
ep = &array[0]; ②  
  
ep = &array[2]; ③  
  
*ep = 1; ④
```



# Zeiger sind Felder

- Ein Feldbezeichner ist **syntaktisch äquivalent** zu einem konstanten Zeiger auf das erste Element des Feldes: `array`  $\equiv$  `&array[0]`
- Diese Beziehung gilt in beide Richtungen: `*array`  $\equiv$  `array[0]`
  - Ein Zeiger kann wie ein Feld verwendet werden
  - Insbesondere kann der `[ ]`-Operator angewandt werden ↪ 13-9
- Beispiel (vgl. ↪ 13-9)

```
uint8_t LEDs[] = { RED0, YELLOW0, GREEN0, BLUE0 };
```

```
LEDs[ 3 ] = BLUE1;
```

```
uint8_t *p = LEDs;
```

```
for( uint8_t i = 0; i < 4; ++i ) {  
    sb_led_on( p[ i ] );  
}
```

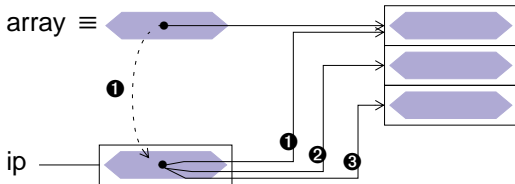


# Rechnen mit Zeigern

- Im Unterschied zu einem Feldbezeichner ist eine *Zeigervariable* ein Behälter  $\rightsquigarrow$  Ihr Wert ist veränderbar
- Neben einfachen Zuweisungen ist dabei auch **Arithmetik** möglich

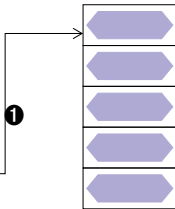
```
int array[3];  
int *ip = array; ❶
```

```
ip++; ❷  
ip++; ❸
```



```
int array[5];  
ip = array; ❶
```

ip



$(ip+3) \equiv \&ip[3]$

Bei der Zeigerarithmetik wird immer die Größe des Objekttyps mit berücksichtigt.



## ■ Arithmetische Operationen

- ++ Prä-/Postinkrement  
↷ Verschieben auf das nächste Objekt
- Prä-/Postdecrement  
↷ Verschieben auf das vorangegangene Objekt
- +, - Addition / Subtraktion eines `int`-Wertes  
↷ Ergebniszeiger ist verschoben um  $n$  Objekte
  - Subtraktion zweier Zeiger  
↷ Anzahl der Objekte  $n$  zwischen beiden Zeigern (Distanz)

## ■ Vergleichsoperationen: `<`, `<=`, `==`, `>=`, `>`, `!=`

↷ 7-3

- ↷ Zeiger lassen sich wie Ganzzahlen vergleichen und ordnen



# Felder sind Zeiger sind Felder – Zusammenfassung

- In Kombination mit Zeigerarithmetik lässt sich in C **jede** Feldoperation auf eine äquivalente Zeigeroperation abbilden.
- Für `int i, array[N], *ip = array;` mit  $0 \leq i < N$  gilt:

```
array    ≡ &array[0]  ≡ ip      ≡ &ip[0]
*array   ≡ array[0]   ≡ *ip     ≡ ip[0]
*(array + i) ≡ array[i] ≡ *(ip + i) ≡ ip[i]
          array++ ≠ ip++
          Fehler: array ist konstant!
```

- Umgekehrt können Zeigeroperationen auch durch Feldoperationen dargestellt werden.  
Der Feldbezeichner kann aber **nicht verändert** werden.



# Felder als Funktionsparameter

- Felder werden in C **immer** als Zeiger übergeben

[=Java]

↪ *Call-by-reference*

```
static uint8_t LEDs[] = {RED0, YELLOW1};

void enlight( uint8_t *array, unsigned n ) {
    for( unsigned i = 0; i < n; ++i )
        sb_led_on( array[i] );
}

void main() {
    enlight( LEDs, 2 );
    uint8_t moreLEDs[] = {YELLOW0, BLUE0, BLUE1};
    enlight( moreLEDs, 3 );
}
```



- Informationen über die Feldgröße gehen dabei verloren!
  - Die Feldgröße muss explizit als Parameter mit übergeben werden
  - In manchen Fällen kann sie auch in der Funktion berechnet werden (z. B. bei Strings durch Suche nach dem abschließenden **NUL**-Zeichen)



- Felder werden in C **immer** als Zeiger übergeben [=Java]  
↳ *Call-by-reference*
- Wird der Parameter als **const** deklariert, so kann die Funktion die Feldelemente **nicht verändern** → Guter Stil! [≠Java]

```
void enlight( const uint8_t *array, unsigned n ) {  
    ...  
}
```

- Um anzuzeigen, dass ein Feld (und kein „Zeiger auf Variable“) erwartet wird, ist auch folgende **äquivalente Syntax** möglich:

```
void enlight( const uint8_t array[], unsigned n ) {  
    ...  
}
```

- **Achtung:** Das gilt so nur bei Deklaration eines Funktionsparameters
- Bei Variablendefinitionen hat **array[]** eine **völlig andere** Bedeutung (Feldgröße aus Initialisierungsliste ermitteln, ↔ 13-8)



- Die Funktion `int strlen(const char *)` aus der Standardbibliothek liefert die Anzahl der Zeichen im übergebenen String

```
void main() {  
    ...  
    const char *string = "hallo"; // string is array of char  
    sb_7seg_showNumber( strlen(string) );  
    ...  
}
```



Dabei gilt: "hallo"  $\equiv$    $\leftrightarrow$  6-13

- Implementierungsvarianten

## Variante 1: Feld-Syntax

```
int strlen( const char s[] ) {  
    int n=0;  
    while( s[n] != 0 )  
        n++;  
    return n;  
}
```

## Variante 2: Zeiger-Syntax

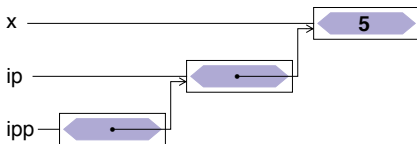
```
int strlen( const char *s ) {  
    const char *end = s;  
    while( *end )  
        end++;  
    return end - s;  
}
```



# Zeiger auf Zeiger

- Ein Zeiger kann auch auf eine Zeigervariable verweisen

```
int x = 5;  
int *ip = &x;  
  
int **ipp = &ip;  
/* → **ipp = 5 */
```



- Wird vor allem bei der Parameterübergabe an Funktionen benötigt
  - Zeigerparameter *call-by-reference* übergeben (z. B. `swap()`-Funktion für Zeiger)
  - Ein Feld von Zeigern übergeben



# Zeiger auf Funktionen

- Ein Zeiger kann auch auf eine Funktion verweisen
  - Damit lassen sich Funktionen an Funktionen übergeben
    - ↳ Funktionen höherer Ordnung
- Beispiel

```
// invokes job() every second
void doPeriodically( void (*job)(void) ) {
    while( 1 ) {
        job(); // invoke job
        for( volatile uint16_t i = 0; i < 0xffff; ++i )
            ; // wait a second
    }
}

void blink( void ) {
    sb_led_toggle( RED0 );
}

void main() {
    doPeriodically( blink ); // pass blink() as parameter
}
```



- Syntax (Definition): `Typ ( * Bezeichner ) ( FormaleParamopt );`  
(sehr ähnlich zur Syntax von Funktionsdeklarationen) ↔ 9-3
  - *Typ* Rückgabetyt der **Funktionen**, auf die dieser Zeiger verweisen kann
  - *Bezeichner* Name des **Funktionszeigers**
  - *FormaleParam<sub>opt</sub>* Formale Parameter der **Funktionen**, auf die dieser Zeiger verweisen kann:  $Typ_1, \dots, Typ_n$
- Ein Funktionszeiger wird genau wie eine Funktion verwendet
  - Aufruf mit `Bezeichner ( TatParam )` ↔ 9-4
  - Adress- (&) und Verweisoperator (\*) werden nicht benötigt ↔ 13-4
  - Ein Funktionsbezeichner ist ein konstanter Funktionszeiger

```
void blink( uint8_t which ) { sb_led_toggle( which ); }

void main() {
    void (*myfun)(uint8_t); // myfun is pointer to function
    myfunc = blink;        // blink is constant pointer to function
    myfun( RED0 );         // invoke blink() via function pointer
    blink( RED0 );         // invoke blink()
}
```



- Funktionszeiger werden oft für **Rückruffunktionen** (*Callbacks*) zur Zustellung asynchroner Ereignisse verwendet (→ „Listener“ in Java)

```
// Example: asynchronous button events with libspicboard
#include <avr/interrupt.h>           // for sei()
#include <7seg.h>                    // for sb_7seg_showNumber()
#include <button.h>                  // for button stuff

// callback handler for button events (invoked on interrupt level)
void onButton( BUTTON b, BUTTONEVENT e ) {
    static int8_t count = 1;
    sb_7seg_showNumber( count++ ); // show no of button presses
    if( count > 99 ) count = 1;    // reset at 100
}

void main() {
    sb_button_registerListener(      // register callback
        BUTTON0, BTNPPRESSED,      // for this button and events
        onButton                    // invoke this function
    );
    sei();                          // enable interrupts (necessary!)
    while( 1 ) ;                    // wait forever
}
```

