Systemnahe Programmierung in C (SPiC)

Teil C Systemnahe Softwareentwicklung

Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2016

http://www4.cs.fau.de/Lehre/SS16/V_SPIC

Überblick: Teil C Systemnahe Softwareentwicklung

12 Programmstruktur und Module

13 Zeiger und Felder

14 μC-Systemarchitektur

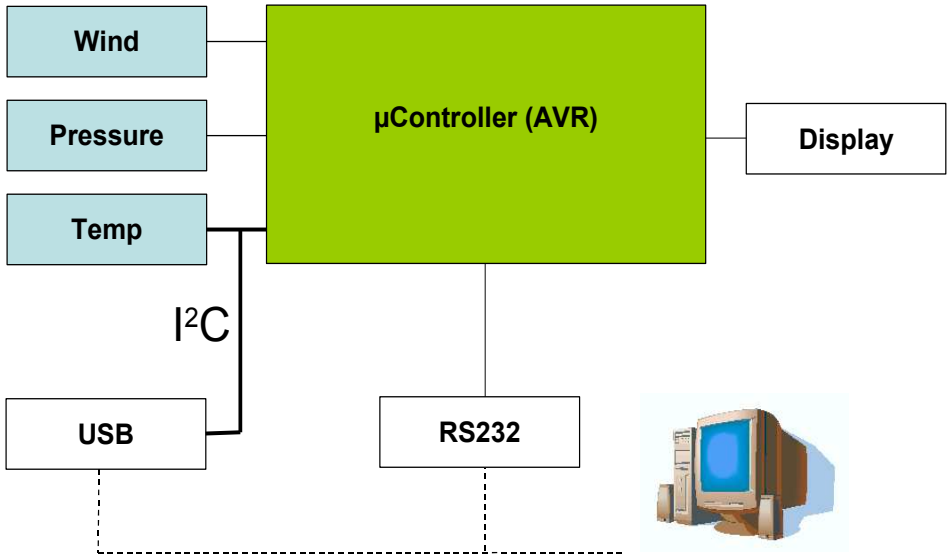
Softwareentwurf

- Softwareentwurf: Grundsätzliche Überlegungen über die Struktur eines Programms **vor** Beginn der Programmierung
  - Ziel: Zerlegung des Problems in beherrschbare Einheiten
- Es gibt eine Vielzahl von Softwareentwurfs-Methoden
  - Objektorientierter Entwurf [↪ GDI, 01-01]
    - Stand der Kunst
    - Dekomposition in Klassen und Objekte
    - An Programmiersprachen wie C++ oder Java ausgelegt
  - Top-Down-Entwurf / Funktionale Dekomposition
    - Bis Mitte der 80er Jahre fast ausschließlich verwendet
    - Dekomposition in Funktionen und Funktionsaufrufe
    - An Programmiersprachen wie Fortran, Cobol, Pascal oder C orientiert

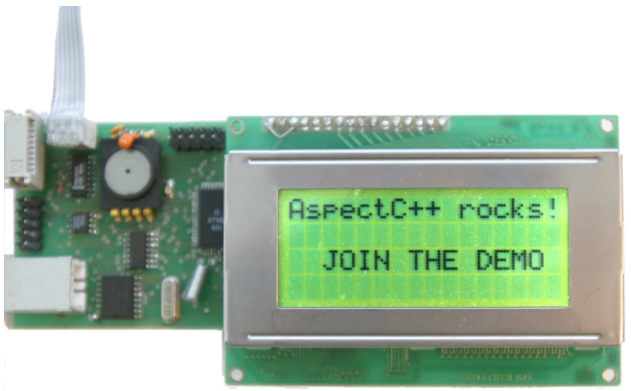
Systemnahe Software wird oft (noch) mit **Funktionaler Dekomposition** entworfen und entwickelt.

Beispiel-Projekt: Eine Wetterstation

- Typisches eingebettetes System
  - Mehrere Sensoren
    - Wind
    - Luftdruck
    - Temperatur
  - Mehrere Aktoren (hier: Ausgabegeräte)
    - LCD-Anzeige
    - PC über RS232
    - PC über USB
  - Sensoren und Aktoren an den μC angebunden über verschiedene Bussysteme
    - I^2C
    - RS232



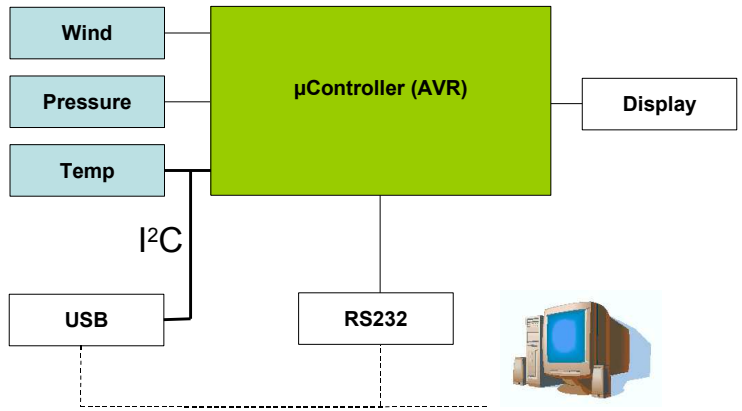
Wie sieht die **funktionale Dekomposition** der Software aus?



Funktionale Dekomposition: Beispiel

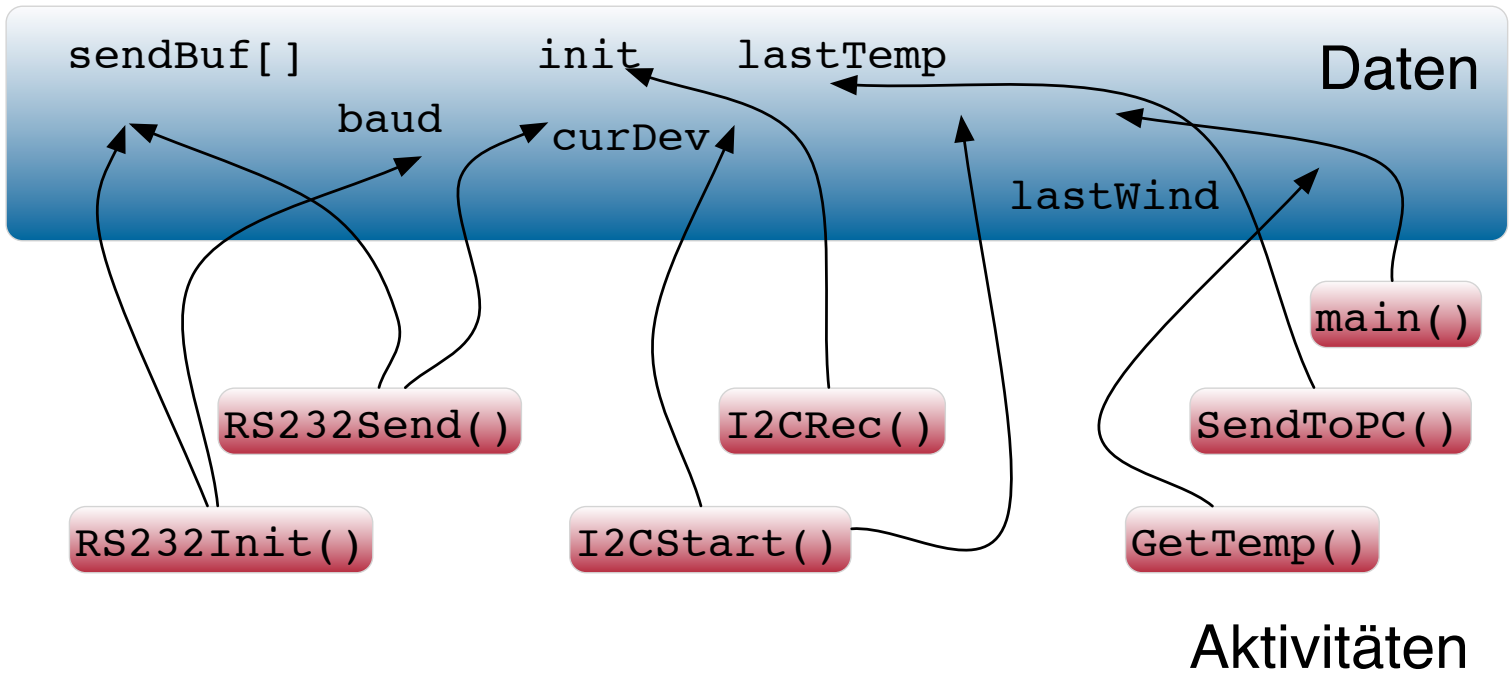
Funktionale Dekomposition der Wetterstation (Auszug):

1. Sensordaten lesen
   - 1.1 Temperatursensor lesen
     - 1.1.1 I²C-Datenübertragung initiieren
     - 1.1.2 Daten vom I²C-Bus lesen
   - 1.2 Drucksensor lesen
   - 1.3 Windsensor lesen
2. Daten aufbereiten (z. B. glätten)
3. Daten ausgeben
   - 3.1 Daten über RS232 versenden
     - 3.1.1 Baudrate und Parität festlegen (einmalig)
     - 3.1.2 Daten schreiben
   - 3.2 LCD-Display aktualisieren
4. Warten und ab Schritt 1 wiederholen



Funktionale Dekomposition: Probleme

- Erzielte Gliederung betrachtet nur die Struktur der Aktivitäten, nicht jedoch die die Struktur der Daten
- Gefahr: Funktionen arbeiten „wild“ auf einer Unmenge schlecht strukturierter Daten $\rightsquigarrow$ mangelhafte Trennung der Belange



Funktionale Dekomposition: Probleme

- Erzielte Gliederung betrachtet nur die Struktur der Aktivitäten, nicht jedoch die die Struktur der Daten
- Gefahr: Funktionen arbeiten „wild“ auf einer Unmenge schlecht strukturierter Daten $\leadsto$ mangelhafte Trennung der Belange

Prinzip der **Trennung der Belange**

Dinge, die **nichts miteinander** zu tun haben, sind auch **getrennt** unterzubringen!

Trennung der Belange (*Separation of Concerns*) ist ein **Fundamentalprinzip** der Informatik
(wie auch jeder anderen Ingenieursdisziplin).

Zugriff auf Daten (Variablen)

- Variablen haben ↪ 10–1
  - Sichtbarkeit (*Scope*) „Wer kann auf die Variable zugreifen?“
  - Lebensdauer „Wie lange steht der Speicher zur Verfügung?“
- Wird festgelegt durch Position (Pos) und Speicherklasse (SK)

Pos	SK ↦	Sichtbarkeit	Lebensdauer
Lokal	*keine*, `auto`	Definition → Blockende	Definition → Blockende
	`static`	Definition → Blockende	Programmstart → Programmende
Global	*keine*	unbeschränkt	Programmstart → Programmende
	`static`	modulweit	Programmstart → Programmende

```
int a = 0;                    // a: global
static int b = 47;            // b: local to module

void f(void) {
  auto int a = b;             // a: local to function (auto optional)
                              //    destroyed at end of block
  static int c = 11;          // c: local to function, not destroyed
}
```


Zugriff auf Daten (Variablen) (Forts.)

- Sichtbarkeit und Lebensdauer sollten restriktiv ausgelegt werden
  - Sichtbarkeit so beschränkt wie möglich!
    - Überraschende Zugriffe „von außen“ ausschließen (Fehlersuche)
    - Implementierungsdetails verbergen (Black-Box-Prinzip, *information hiding*)
  - Lebensdauer so kurz wie möglich
    - Speicherplatz sparen
    - Insbesondere wichtig auf μ-Controller-Plattformen

↪ 1–4

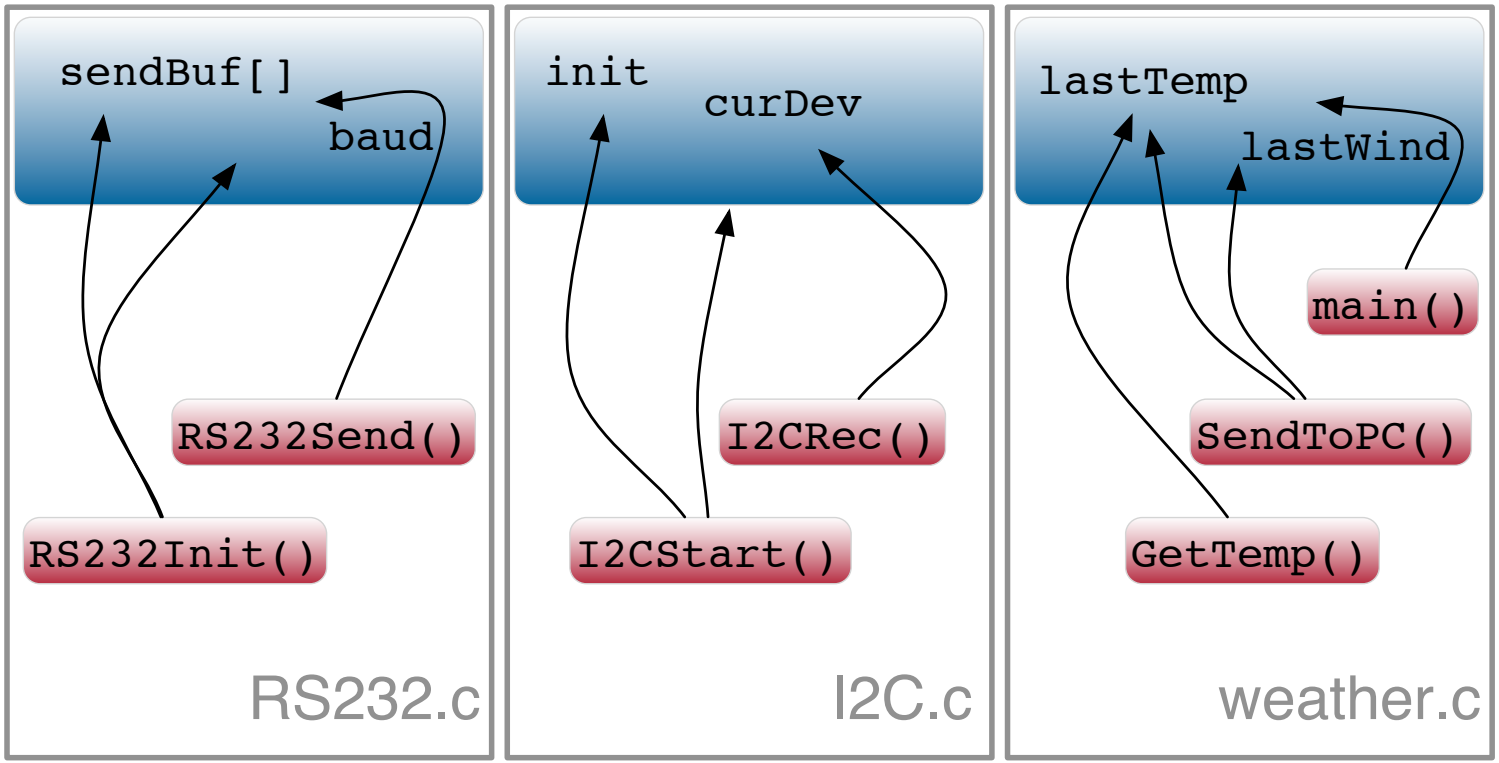
Konsequenz: Globale Variablen vermeiden!

- Globale Variablen sind überall sichtbar
- Globale Variablen belegen Speicher über die gesamte Programmlaufzeit

Regel: Variablen erhalten stets die **geringstmögliche Sichtbarkeit und Lebensdauer**

Lösung: Modularisierung

- Separation jeweils zusammengehöriger Daten und Funktionen in übergeordnete Einheiten $\rightsquigarrow$ **Module**



Was ist ein Modul?

- **Modul** := (<*Menge von Funktionen*>, (↦ „`class`“ in Java)
  <*Menge von Daten*>,
  <*Schnittstelle*>)
- Module sind größere Programmbausteine ↪ 9–1
  - Problemorientierte Zusammenfassung von Funktionen und Daten
    ⇝ Trennung der Belange
  - Ermöglichen die einfache Wiederverwendung von Komponenten
  - Ermöglichen den einfachen Austausch von Komponenten
  - Verbergen Implementierungsdetails (**Black-Box**-Prinzip)
    ⇝ Zugriff erfolgt ausschließlich über die Modulschnittstelle

Modul ↦ Abstraktion ↪ 4–1

- Die Schnittstelle eines Moduls abstrahiert
  - Von der tatsächlichen Implementierung der Funktionen
  - Von der internen Darstellung und Verwendung von Daten

Module in C

[≠Java]

- In C ist das Modulkonzept nicht Bestandteil der Sprache, sondern rein idiomatisch (über Konventionen) realisiert ↪ 3–12
  - Modulschnittstelle ↦ `.h`-Datei (enthält Deklarationen ↪ 9–7)
  - Modulimplementierung ↦ `.c`-Datei (enthält Definitionen ↪ 9–3)
  - Modulverwendung ↦ `#include <Modul.h>`

```
void RS232Init( uint16_t br );
void RS232Send( char ch );
...
```

RS232.h: **Schnittstelle / Vertrag (öffentl.)**
Deklaration der bereitgestellten Funktionen (und ggf. Daten)

```
#include <RS232.h>
static uint16_t   baud = 2400;
static char       sendBuf[16];
...
void RS232Init( uint16_t br) {
  ...
  baud = br;
}
void RS232Send( char ch ) {
  sendBuf[···] = ch;
  ...
}
```

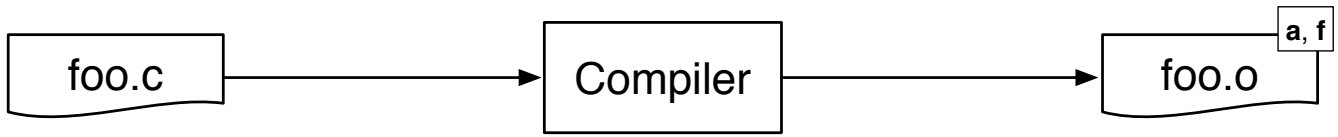
RS232.c: **Implementierung (nicht öffentl.)**
Definition der bereitgestellten Funktionen (und ggf. Daten)

Ggf. modulinterne Hilfsfunktionen und Daten (`static`)

Inklusion der eigenen Schnittstelle stellt sicher, dass der Vertrag eingehalten wird

Module in C – Export [≠Java]

- Ein C-Modul exportiert eine Menge von definierten Symbolen
  - Alle Funktionen und globalen Variablen (↦ „`public`“ in Java)
  - Export kann mit `static` unterbunden werden (↦ „`private`“ in Java)
    (↦ Einschränkung der Sichtbarkeit ↪ 12–5)
- Export erfolgt beim Übersetzungsvorgang (`.c`-Datei ⟶ `.o`-Datei)



Quelldatei (`foo.c`)

```
uint16_t a;
// public
static uint16_t b;
// private

void f(void)          // public
{ ... }
static void g(int)    // private
{ ... }
```

Objektdatei (`foo.o`)

Symbole **a** und **f** werden exportiert.

Symbole b und g sind `static` definiert und werden deshalb nicht exportiert.

Module in C – Import [≠Java]

- Ein C-Modul importiert eine Menge nicht-definierter Symbole
  - Funktionen und globale Variablen, die verwendet werden, im Modul selber jedoch nicht definiert sind
  - Werden beim Übersetzen als unaufgelöst markiert

Quelldatei (`bar.c`)

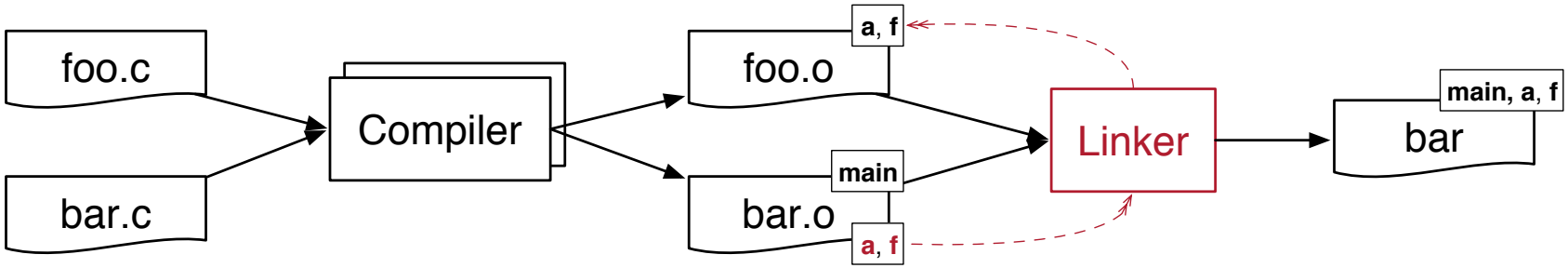
```
extern uint16_t a;
// declare
void f(void);       // declare

void main() {       // public
  a = 0x4711;       // use
  f();              // use
}
```

Objektdatei (`bar.o`)

Symbol **`main`** wird exportiert.
Symbole **`a`** und **`f`** sind unaufgelöst.

Module in C – Import (Forts.) [≠Java]

- Die eigentliche Auflösung erfolgt durch den Linker



Linken ist nicht typsicher!

- Typinformationen sind in Objektdateien nicht mehr vorhanden
- Auflösung durch den Linker erfolgt **ausschließlich** über die **Symbolnamen** (Bezeichner)

⇝ Typsicherheit muss beim **Übersetzen** sichergestellt werden

⇝ Einheitliche Deklarationen durch gemeinsame Header-Datei

Module in C – Header [≠Java]

- Elemente aus fremden Modulen müssen deklariert werden
  - Funktionen durch normale Deklaration ↪ 9–7

```
void f(void);
```

  - Globale Variablen durch `extern`

```
extern uint16_t a;
```

Das `extern` unterscheidet eine Variablendeklaration von einer Variablendefinition.

- Die Deklarationen erfolgen sinnvollerweise in einer Header-Datei, die von der Modulentwicklerin bereitgestellt wird
  - Schnittstelle des Moduls (↦ „`interface`“ in Java)
    - Exportierte Funktionen des Moduls
    - Exportierte globale Variablen des Moduls
    - Modulspezifische Konstanten, Typen, Makros
    - Verwendung durch Inklusion (↦ „`import`“ in Java)
  - Wird **auch vom Modul inkludiert**, um Übereinstimmung von Deklarationen und Definitionen sicher zu stellen (↦ „`implements`“ in Java)

Module in C – Header (Forts.)

[≠Java]

Modulschnittstelle: `foo.h`

```
// foo.h
#ifndef _FOO_H
#define _FOO_H

// declarations
extern uint16_t a;
void f(void);

#endif // _FOO_H
```

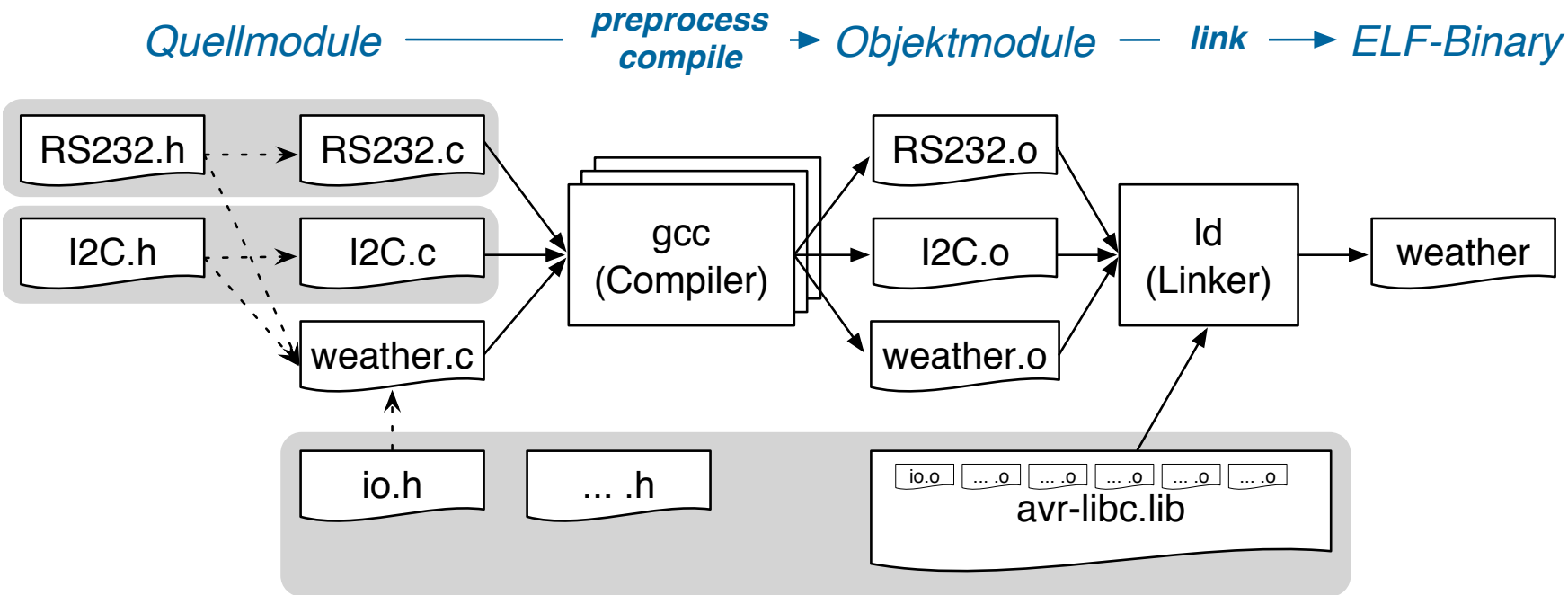
Modulimplementierung `foo.c`

```
// foo.c
#include <foo.h>


// definitions
uint16_t a;
void f(void){
  ...
}
```

Modulverwendung `bar.c`
(vergleiche ↪ 12–11)

```
// bar.c
extern uint16_t a;
void f(void);
#include <foo.h>

void main() {
  a = 0x4711;
  f();
}
```


Zurück zum Beispiel: Wetterstation



- Jedes Modul besteht aus Header- und Implementierungsdatei(en)
  - `.h`-Datei definiert die Schnittstelle
  - `.c`-Datei implementiert die Schnittstelle, inkludiert `.h`-Datei, um sicherzustellen, dass Deklaration und Definition übereinstimmen
- Modulverwendung durch Inkludieren der modulspezifischen `.h`-Datei
- Das Ganze funktioniert entsprechend bei Bibliotheken

Zusammenfassung

- Prinzip der Trennung der Belange $\leadsto$ Modularisierung
  - Wiederverwendung und Austausch wohldefinierter Komponenten
  - Verbergen von Implementierungsdetails

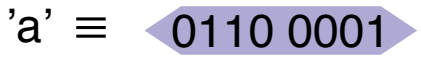
- In C ist das Modulkonzept nicht Bestandteil der Sprache, sondern idiomatisch durch Konventionen realisiert
  - Modulschnittstelle $\mapsto$ `.h`-Datei (enthält Deklarationen)
  - Modulimplementierung $\mapsto$ `.c`-Datei (enthält Definitionen)
  - Modulverwendung $\mapsto$ `#include <Modul.h>`
  - `private` Symbole $\mapsto$ als `static` definieren

- Die eigentliche Zusammenfügung erfolgt durch den Linker
  - Auflösung erfolgt ausschließlich über Symbolnamen
    $\leadsto$ **Linken ist nicht typsicher!**
  - Typsicherheit muss beim Übersetzen sichergestellt werden
    $\leadsto$ durch gemeinsame Header-Datei

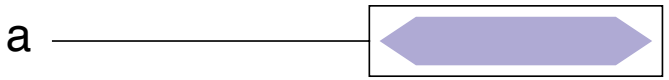
Überblick: Teil C Systemnahe Softwareentwicklung

Einordnung: Zeiger (*Pointer*)

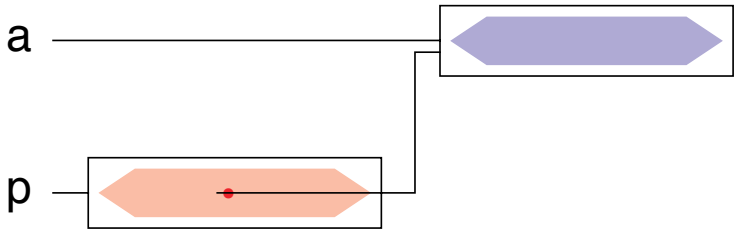
- **Literal:** `'a'`
  Darstellung eines Wertes



- **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*) ↪ 9–5
  - Speicher lässt sich direkt ansprechen
  - Effizientere Programme

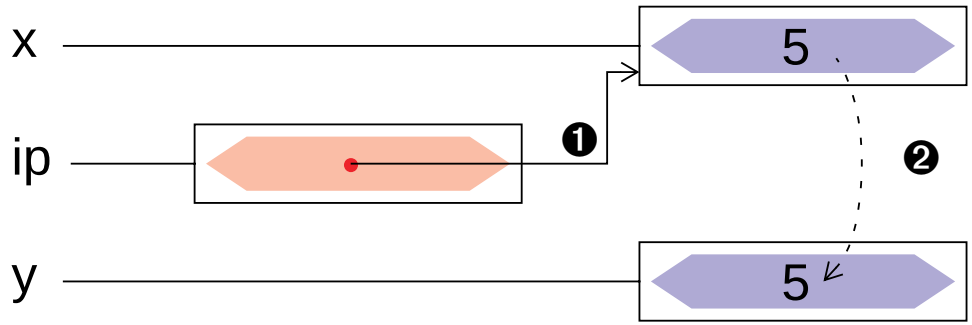
„Effizienz durch Maschinennähe“ ↪ 3–13

- Aber auch viele Probleme!
  - Programmstruktur wird unübersichtlicher (welche Funktion kann auf welche Variablen zugreifen?)
  - Zeiger sind die **häufigste Fehlerquelle** in C-Programmen!

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)) ≡ 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` `typedef char* CPTR`	in Definitionen und Deklarationen
3.	Verweis (unär):	`x = *p1`	in Ausdrücken

Insbesondere 2. und 3. führen zu Verwirrung
$\rightsquigarrow$ * 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

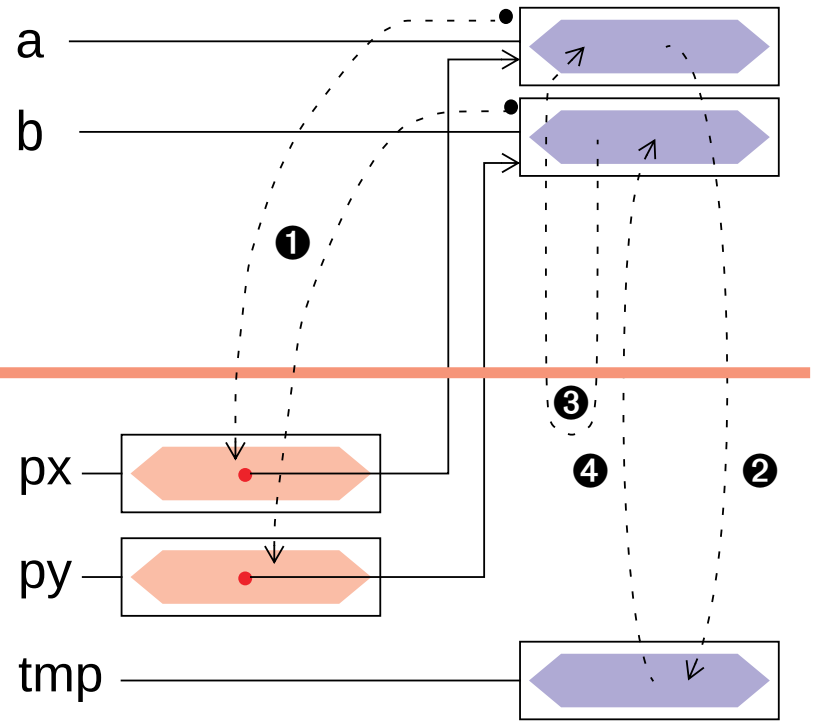
- Das gilt auch für Zeiger (Verweise) [↪ GDI, 14-01-01]
  - Aufgerufene Funktion erhält eine Kopie des Adressverweises
  - Mit Hilfe des *-Operators kann darüber jedoch auf die Zielvariable zugegriffen werden und diese verändert werden

  ⇝ **Call-by-reference**

Zeiger als Funktionsargumente (Forts.)

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

}
```

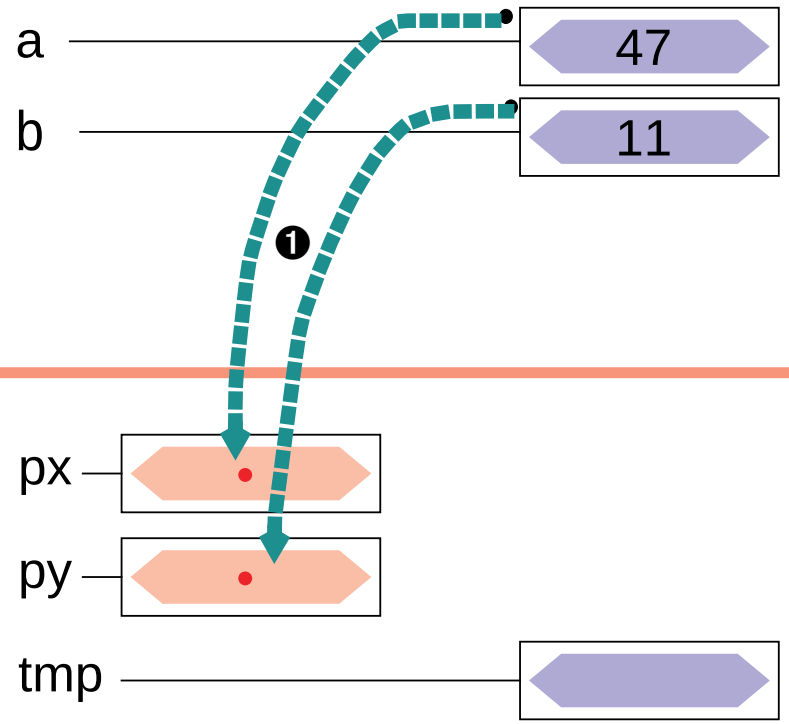


Zeiger als Funktionsargumente (Forts.)

- 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;
```

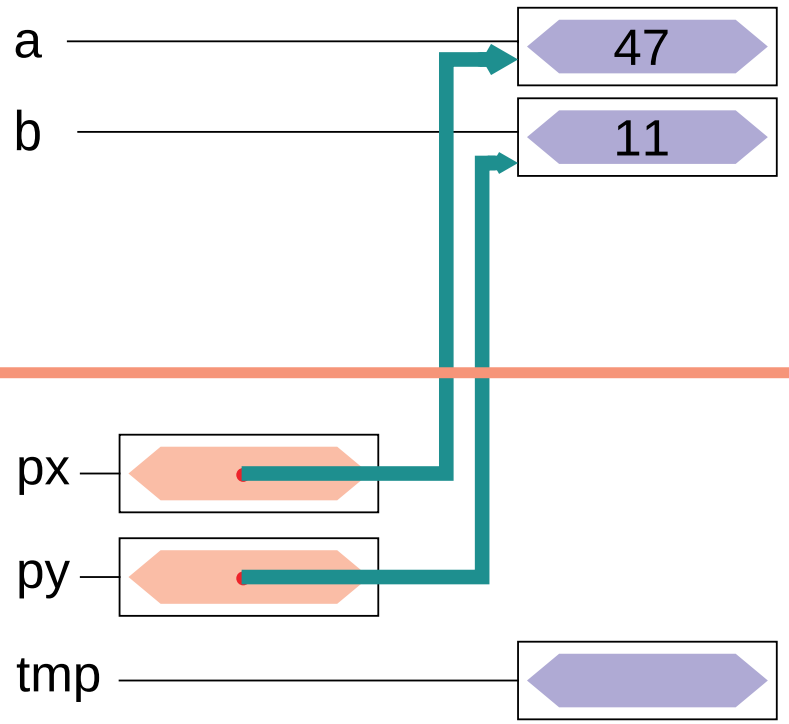


Zeiger als Funktionsargumente (Forts.)

- Beispiel (Einzelschritte)

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

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

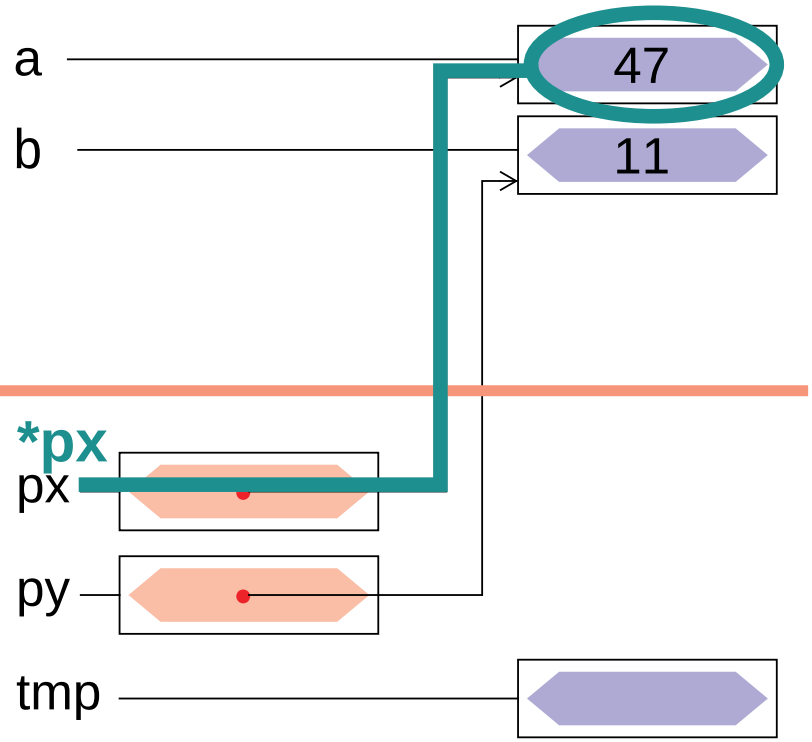


Zeiger als Funktionsargumente (Forts.)

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; ❷
```

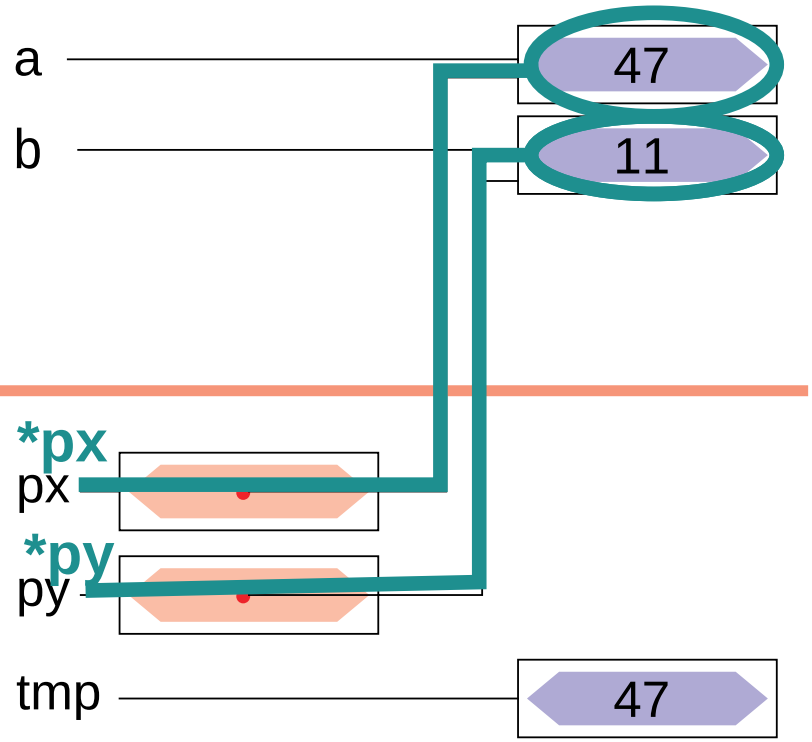


Zeiger als Funktionsargumente (Forts.)

- 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; ❷
    *px = *py; ❸
```

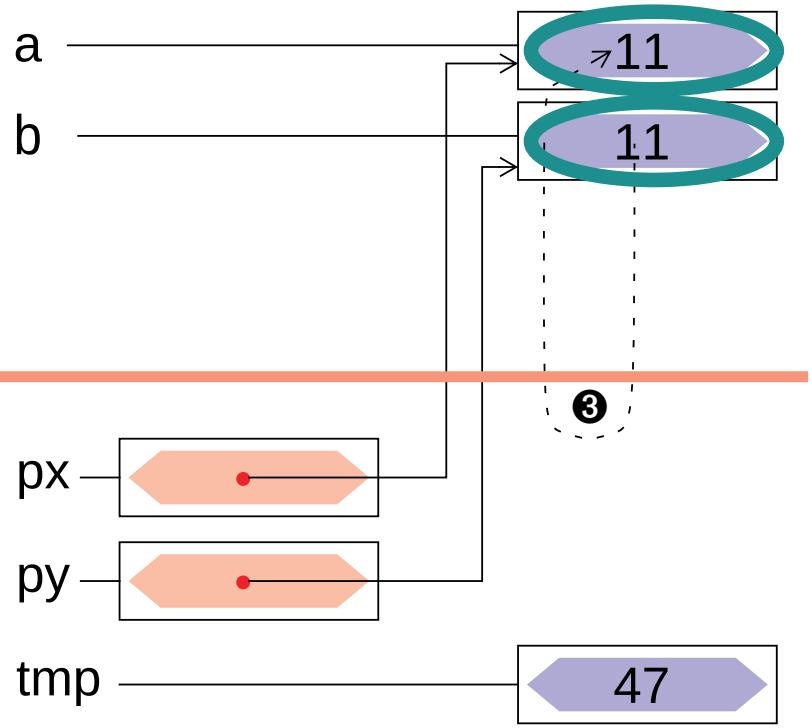


Zeiger als Funktionsargumente (Forts.)

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; ❷
    *px = *py; ❸
```

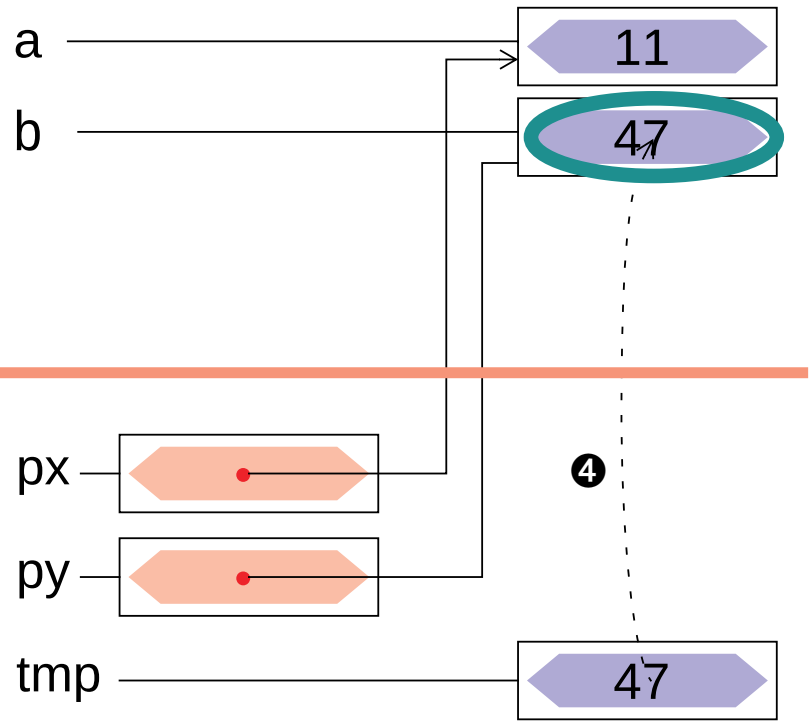


Zeiger als Funktionsargumente (Forts.)

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; ❷
    *px = *py; ❸
    *py = tmp; ❹

}
```



Einordnung: Felder (*Arrays*) [≈Java]

- **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

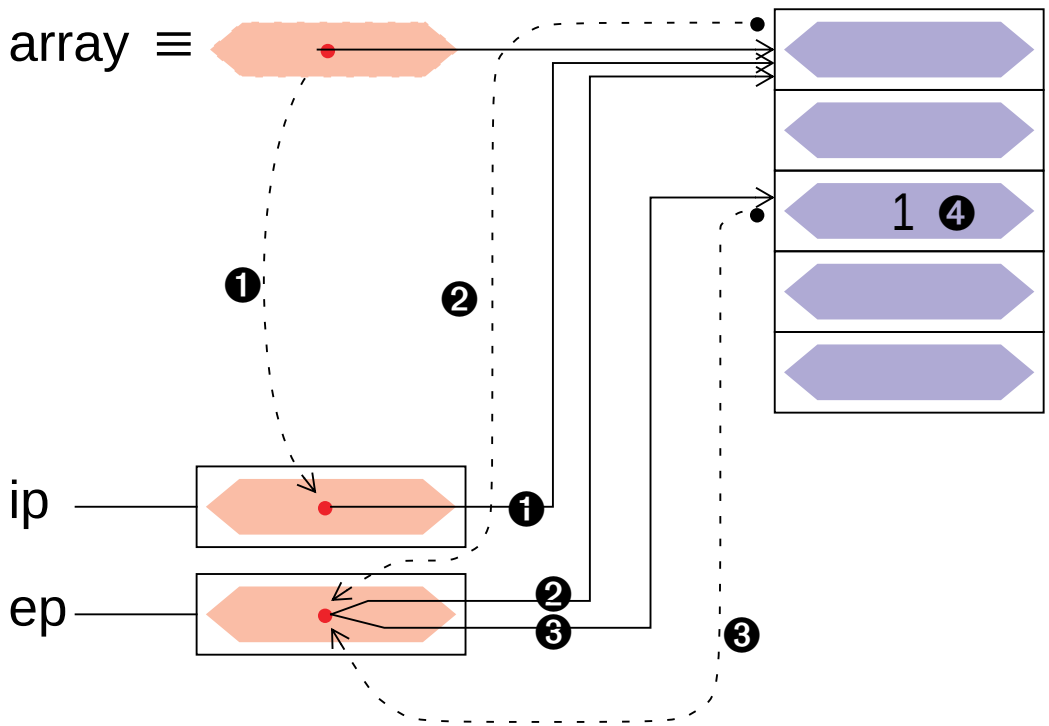
- Syntax: *Feld* **[** *IntAusdruck* **]** [=Java]
  - Wobei $0 \leq$ *IntAusdruck* $< n$ für $n =$ Feldgröße
  - **Achtung:** Feldindex wird nicht überprüft [≠Java]
    $\leadsto$ häufige Fehlerquelle in C-Programmen
- Beispiel

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

LEDs[ 3 ] = BLUE1;

for( unit8_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` ≡ `&array[0]`
  - Ein Alias – kein Behälter ⇝ Wert kann nicht verändert werden
  - Über einen so ermittelten Zeiger ist ein indirekter Feldzugriff möglich
- Beispiel (Gesamtüberblick)

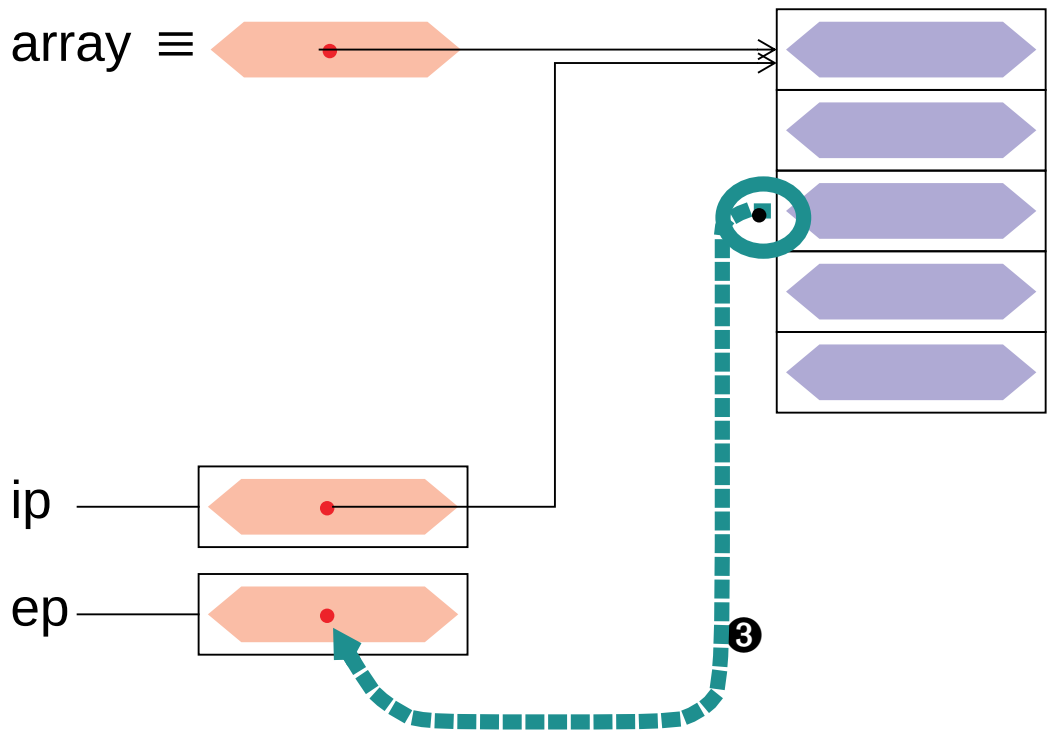
```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸

*ep = 1; ❹
```



Felder sind Zeiger

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

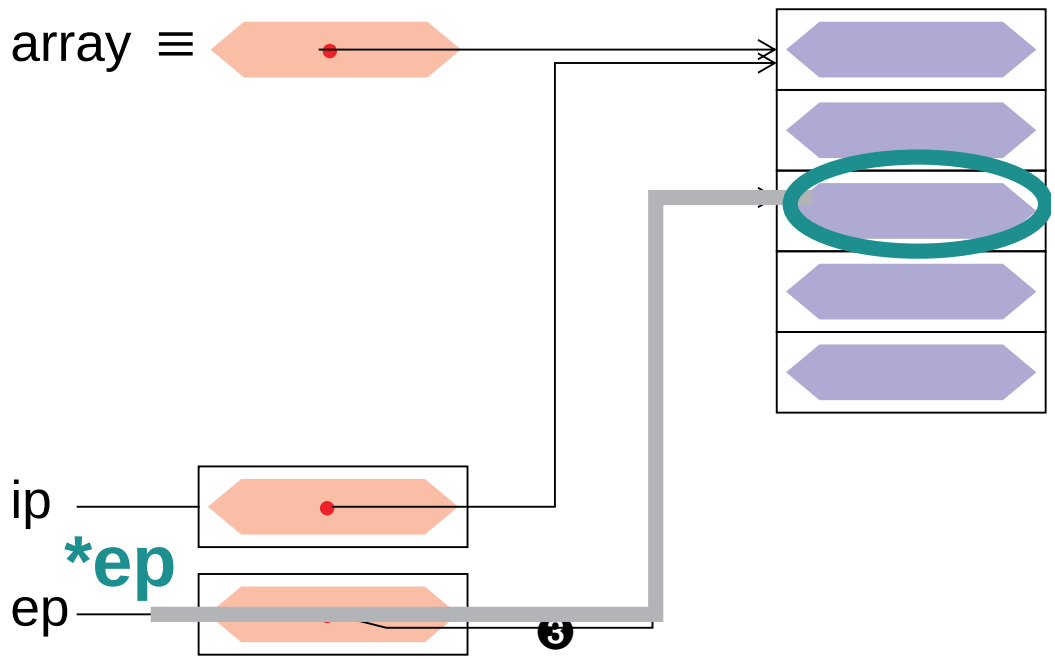
```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸
```



Felder sind Zeiger

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

```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸

*ep = 1; ❹
```



Zeiger sind Felder

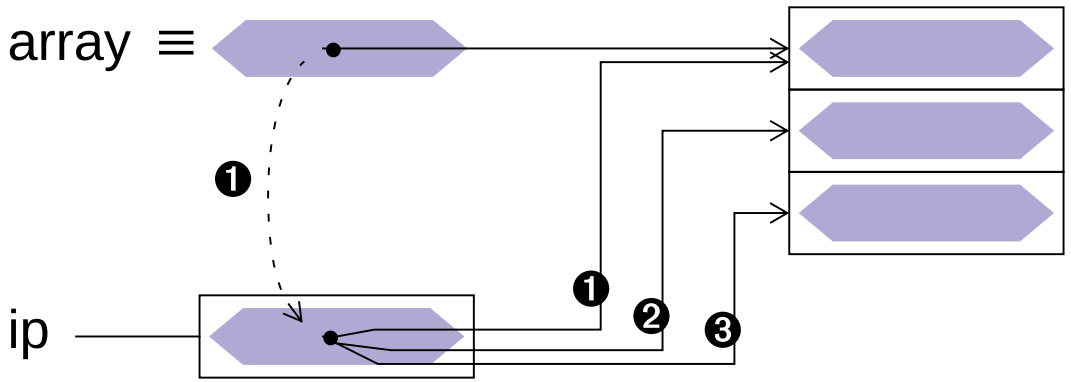
- Ein Feldbezeichner ist syntaktisch äquivalent zu einem konstanten Zeiger auf das erste Element des Feldes: `array` ≡ `&array[0]`
- Diese Beziehung gilt in beide Richtungen: `*array` ≡ `array[0]`
  - Ein Zeiger kann wie ein Feld verwendet werden
  - Insbesondere kann der **[ ]** - Operator angewandt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

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

LEDs[ 3 ]    = BLUE1;
uint8_t *p   = LEDs;
for( unit8_t i = 0; i < 4; ++i ) {
  sb_led_on( p[ i ] );
}
```

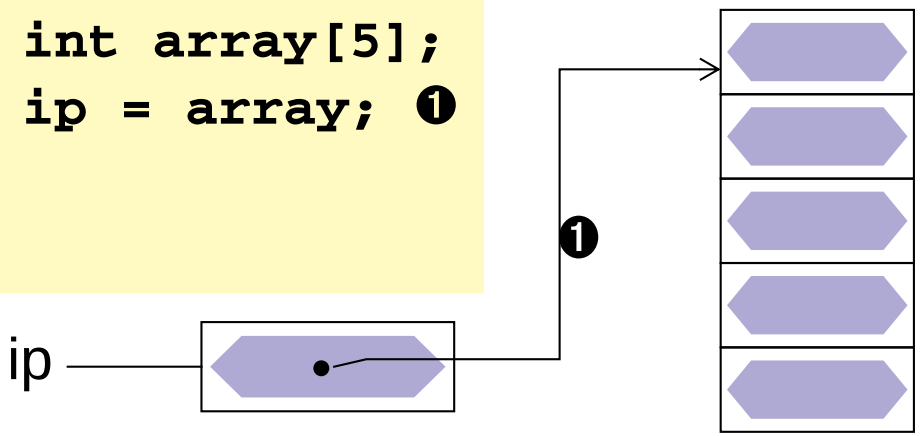
Rechnen mit Zeigern

- Im Unterschied zu einem Feldbezeichner ist eine Zeiger*variable* 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+3)` ≡ `&ip[3]`

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

Zeigerarithmetik – Operationen

- Arithmetische Operationen
  - ++ Prä-/Postinkrement
    - ⇝ Verschieben auf das nächste Objekt
  - −− Prä-/Postdekrement
    - ⇝ 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!

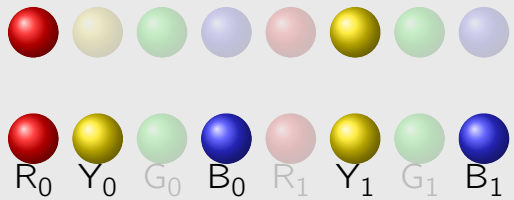
- Umgekehrt können Zeigeroperationen auch durch Feldoperationen dargestellt werden.
Der Feldbezeichner kann aber nicht verändert werden.

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 als Funktionsparameter (Forts.)

- 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 Funktionparameters
  - Bei Variablendefinitionen hat `array[]` eine völlig andere Bedeutung (Feldgröße aus Initialisierungsliste ermitteln, ↪ 13–8)

Felder als Funktionsparameter (Forts.)

- 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" ≡ 

↪ 6–13

- Implementierungsvarianten

Variante 1: Feld-Syntax

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

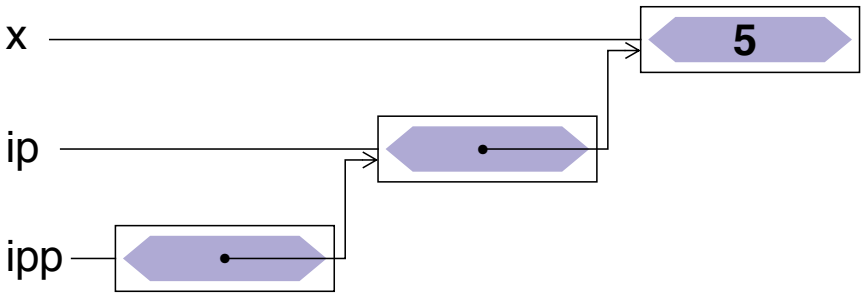
Variante 2: Zeiger-Syntax

```
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
    $\mapsto$ 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
}
```


Zeiger auf Funktionen (Forts.)

- Syntax (Definition): *Typ* **(** * *Bezeichner* **)(** $FormaleParam_{opt}$ **);**
  (sehr ähnlich zur Syntax von Funktionsdeklarationen) ↪ 9–3
  - *Typ* — Rückgabetyp der Funktionen, auf die dieser Zeiger verweisen kann
  - *Bezeichner* — Name des Funktionszeigers
  - $FormaleParam_{opt}$ — Formale Parameter der Funktionen, auf die dieser Zeiger verweisen kann: $Typ_1, \ldots, Typ_n$
- 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
  myfun = blink;          // blink is constant pointer to function
  myfun( RED0 );          // invoke blink() via function pointer
  blink( RED0 );          // invoke blink()
}
```

Zeiger auf Funktionen (Forts.)

- Funktionszeiger werden oft für Rückruffunktionen (*Callbacks*) zur Zustellung asynchroner Ereignisse verwendet ($\mapsto$ „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, BTNPRESSED,         //   for this button and events
    onButton                     //   invoke this function
  );
  sei();                         // enable interrupts (necessary!)
  while( 1 ) ;                   // wait forever
}
```

Zusammenfassung

- Ein Zeiger verweist auf eine Variable im Speicher
  - Möglichkeit des indirekten Zugriffs auf den Wert
  - Grundlage für die Implementierung von *call-by-reference* in C
  - Grundlage für die Implementierung von Feldern
  - Wichtiges Element der Maschinennähe von C
  - **Häufigste Fehlerursache in C-Programmen**
- Die syntaktischen Möglichkeiten sind vielfältig (und verwirrend)
  - Typmodifizierer `*`, Adressoperator `&`, Verweisoperator `*`
  - Zeigerarithmetik mit `+`, `-`, `++` und `--`
  - syntaktische Äquivalenz zu Feldern (`[]` Operator)
- Zeiger können auch auf Funktionen verweisen
  - Übergeben von Funktionen an Funktionen
  - Prinzip der Rückruffunktion