E Mikrocontroller-Programmierung

E.1 Überblick

- Mikrocontroller-Umgebung
  - ➤ Prozessor am Beispiel AVR-Mikrocontroller
  - ➤ Speicher
  - ➤ Peripherie
- Programmausführung
  - ➤ Programm laden
  - ➤ starten
  - ➤ Fehler zur Laufzeit
- Interrupts
  - ➤ Grundkonzepte
  - ➤ Nebenläufigkeit
  - ➤ Umgang mit Nebenläufigkeit

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2010
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

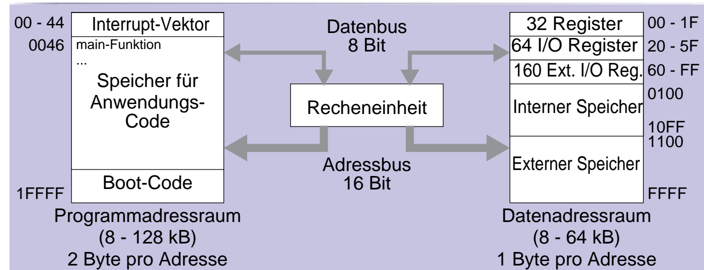
1 Mikrocontroller-Umgebung

- Programm läuft "nackt" auf der Hardware
  - ➥ Compiler und Binder müssen ein vollständiges Programm erzeugen
    - ➤ keine Betriebssystemunterstützung zur Laufzeit
  - ◆ Funktionalität muss entweder vom Anwender programmiert werden oder in Form von Funktionsbibliotheken zum Programm dazugebunden werden
  - ◆ Umgang mit "lästigen Programmierdetails" (z. B. bestimmte Bits setzen) wird durch Makros erleichtert
- Es wird genau ein Programm ausgeführt
  - ➤ Programm kann zur Laufzeit "niemanden stören"
  - ➤ Fehler betreffen nur das Programm selbst
  - ➤ keine Schutzmechanismen notwendig
    - ➥ ABER: Fehler ohne direkte Auswirkung werden leichter übersehen

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2010
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

E.2 Beispiel: AVR-Mikrocontroller (ATmega-Serie)

1 Architektur



- Getrennter Speicher für Programm (Flash-Speicher) und Daten (SRAM) (Harvard-Architektur — im Gegensatz zur von-Neumann-Architektur beim PC)
- Register des Prozessors und Register für Ein-/Ausgabe-Schnittstellen sind in Adressbereich des Datenspeichers eingebettet

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2010
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Architektur(2)

- Peripherie-Bausteine werden über I/O-Register angesprochen bzw. gesteuert (Befehle werden in den Bits eines I/O-Registers kodiert)
  - ➤ mehrere Timer
    (Zähler, deren Geschwindigkeit einstellbar ist und die bei einem bestimmten Wert einen Interrupt auslösen)
  - ➤ Ports
    (Gruppen von jeweils 8 Anschlüssen, die auf 0V oder V_{cc} gesetzt, bzw. deren Zustand abgefragt werden kann)
  - ➤ Output Compare Modulator (OCM)
    (zur Pulsweitenmodulation)
  - ➤ Serial Peripheral Interface (SPI)
  - ➤ Synchrone/Asynchrone serielle Schnittstelle (USART)
  - ➤ Analog-Comparator
  - ➤ A/D-Wandler
  - ➤ EEPROM
    (zur Speicherung von Konfigurationsdaten)

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2010
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 In Speicher eingebettete Register (memory mapped registers)

- Je nach Prozessor sind Zugriffe auf I/O-Register auf zwei Arten realisiert:
  - spezielle Befehle (z. B. in, out bei x86)
  - in den Adressraum eingebettet, Zugriff mittels Speicheroperationen
- Bei den meisten Mikrocontrollern sind die Register in den Speicheradressraum eingebettet
- Zugriffe auf die entsprechende Adresse werden auf das entsprechende Register umgeleitet

➥ sehr einfacher und komfortabler Zugriff

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 I/O-Ports

- Ein I/O-Port ist eine Gruppe von meist 8 Anschluss-Pins und dient zum Anschluss von digitalen Peripheriegeräten
- Die Pins können entweder als Eingang oder als Ausgang dienen
  - als Ausgang konfiguriert, kann man festlegen, ob sie eine logische "1" oder eine logische "0" darstellen sollen
    - ➤ Ausgang wird dann entsprechend auf V_{CC} oder GND gesetzt
  - als Eingang konfiguriert, kann man den Zustand abfragen
- Manche I/O Pins können dazu genutzt werden, einen Interrupt (IRQ = Interrupt Request) auszulösen (externe Interrupt-Quelle)
- Die meisten Pins können alternativ eine Spezialfunktion übernehmen, da sie einem integrierten Gerät als Ein- oder Ausgabe dienen können
  - z. B. dienen die Pins 0 und 1 von Port E (ATmega128) entweder als allgemeine I/O-Ports oder als RxD und TxD der seriellen Schnittstelle

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

4 Programme laden

- generell bei Mikrocontrollern mehrere Möglichkeiten
- ▲ Programm ist schon da (ROM)
- ▲ Bootloader-Programm ist da, liest Anwendung über serielle Schnittstelle ein und speichert sie im Programmspeicher ab
- ▲ spezielle Hardware-Schnittstelle
  - ➤ "jemand anderes" kann auf Speicher zugreifen
  - ➤ Beispiel: JTAG
    spezielle Hardware-Komponente im AVR-Chip, die Zugriff auf die Speicher hat und mit der man über spezielle PINs kommunizieren kann

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

5 Programm starten

- Reset bewirkt Ausführung des Befehls an Adresse 0x0000
  - ➤ dort steht ein Sprungbefehl auf die Speicheradresse einer start-Funktion, die nach einer Initialisierungsphase die main-Funktion aufruft
  - ➤ alternativ: Sprungbefehl auf Adresse des Bootloader-Programms, Bootloader lädt Anwendung, initialisiert die Umgebung und springt dann auf main-Adresse

6 Fehler zur Laufzeit

- Zugriff auf ungültige Adresse
  - es passiert nichts:
    - ➤ Schreiben geht in’s Leere
    - ➤ Lesen ergibt zufälliges Ergebnis
- ungültige Operation auf nur-lesbare / nur-schreibbare Register/Speicher
  - ➤ hat keine Auswirkung

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

E.3 Interrupts

1 Motivation

- An einer Peripherie-Schnittstelle tritt ein Ereignis auf
  - ➤ Spannung wird angelegt
  - ➤ Zähler ist abgelaufen
  - ➤ Gerät hat Aufgabe erledigt (z. B. serielle Schnittstelle hat Byte übertragen, A/D-Wandler hat neuen Wert vorliegen)
  - ➤ Gerät hat Daten für die Anwendung bereit stehen (z. B. serielle Schnittstelle hat Byte empfangen)

? wie bekommt das Programm das mit?
  - ➤ Zustand der Schnittstelle regelmäßig überprüfen (= ***Polling***)
  - ➤ Schnittstelle meldet sich von sich aus beim Prozessor und unterbricht den Programmablauf (= ***Interrupt***)

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Motivation (2)

- Polling vs. Interrupts: Vor und Nachteile
  - ◆ Polling
    - \+ Pollen erfolgt **synchron** zum Programmablauf, Programm ist in dem Moment auf das Ereignis vorbereitet
    - – Pollen erfolgt explizit im Programm und meistens umsonst — Rechenzeit wird verschwendet
    - – Polling-Funktionalität ist in den normalen Pragrammablauf eingestreut — und hat mit der "eigentlichen" Funktionalität dort meist nichts zu tun

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Motivation (3)

- Polling vs. Interrupts: Vor und Nachteile
  - ◆ Interrrupts
    - \+ Interrupts melden sich nur, wenn tatsächlich etwas zu erledigen ist
    - \+ Interrupt-Bearbeitung ist in einer Funktion kompakt zusammengefasst
    - – Interrupts unterbrechen den Programmablauf irgendwo (**asynchron**), sie könnten in dem Augenblick stören
      - ➥ durch die Interrupt-Bearbeitungentsteht **Nebenläufigkeit**

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Implementierung

- typischerweise mehrere Interrupt-Quellen
- Interrupt-Vektor
  - ◆ Speicherbereich (Tabelle), der für jeden Interrupt Informationen zur Bearbeitung enthält
    - ➤ Maschinenbefehl
      (typischerweise ein Sprungbefehl auf eine Adresse, an der eine Bearbeitungsfunktion (***Interrupt-Handler***) steht)
      oder
    - ➤ Adresse einer Bearbeitungsfunktion
  - ◆ feste Position im Speicher — ist im Prozessorhandbuch nachzulesen
- Maskieren von Interrupts
  - ➤ Bit im Prozessor-Statusregister schaltet den Empfang aller Interrupts ab
  - ➤ zwischenzeitlich eintreffende Interrupts werden gepuffert (nur einer!)
  - ➤ die Erzeugung einzelner Interrupts kann am jeweiligen Gerät unterbunden werden

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Ablauf auf Hardware-Ebene

① Gerät löst Interrupt aus, Ablauf des Anwendungsprogramms wird unmittelbar unterbrochen

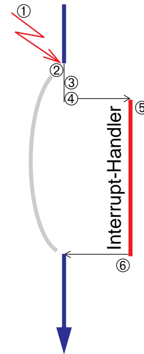
② weitere Interrupts werden deaktiviert

③ aktuelle Position im Programm wird gesichert

④ Eintrag im Interrupt-Vektor ermitteln

⑤ Befehl wird ausgeführt bzw. Funktion aufrufen (= Sprung in den Interrupt-Handler)

⑥ am Ende der Bearbeitungsfunktion bewirkt ein Befehl "Return from Interrupt" die Fortsetzung des Anwendungsprogramms und die Reaktivierung der Interrupts

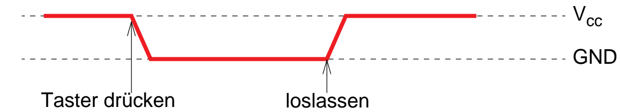


! Der Interrupt-Handler muss alle Register, die er ändert am Anfang sichern und vor dem Rücksprung wieder herstellen!

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

4 Pegel- und Flanken-gesteuerte Interrupts

- Beispiel: Signal eines (idealisierten) Tasters



- Flanken-gesteuert
  - ➤ Interupt wird durch die Flanke (= Wechsel des Pegels) ausgelöst
  - ➤ welche Flanke einen Interrupt auslöst kann bei manchen Prozessoren konfiguriert werden
- Pegel-gesteuert
  - ➤ solange ein bestimmter Pegel anliegt (hier Pegel = GND) wird immer wieder ein Interrupt ausgelöst

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F Zeiger, Felder und Strukturen in C

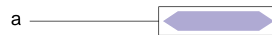
F.1 Zeiger(-Variablen)

1 Einordnung

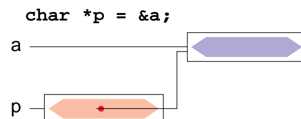
- **Literal:**
  Bezeichnung für einen Wert

’a’ ≡ 0110 0001

- **Variable:**
  Bezeichnung eines Datenobjekts



- **Zeiger-Variable (Pointer):**
  Bezeichnung einer Referenz auf ein Datenobjekt



© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2010
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Überblick

- Eine Zeigervariable ***(pointer)*** enthält als Wert einen Verweis auf den Inhalt einer anderen Variablen
  - ➥ *der Zeiger verweist auf die Variable*
- Über diese Adresse kann man **indirekt** auf die andere Variable zugreifen
- Daraus resultiert die große Bedeutung von Zeigern in C
  - ➥ Funktionen können ihre Argumente verändern ***(call-by-reference)***
  - ➥ dynamische Speicherverwaltung
  - ➥ effizientere Programme
- Aber auch Nachteile!
  - ➥ Programmstruktur wird unübersichtlicher (welche Funktion kann auf welche Variable zugreifen?)
  - ➥ häufigste Fehlerquelle bei C-Programmen

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2010
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

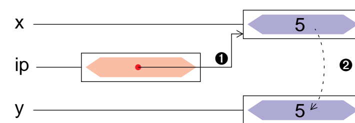
3 Definition von Zeigervariablen

- Syntax:

```
Typ *Name ;
```

4 Beispiele

```
int x = 5;

int *ip;

int y;

ip = &x; ❶

y = *ip; ❷
```



© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2010
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

5 Adressoperatoren

- Adressoperator `&`

  `&x` der unäre Adress-Operator liefert eine Referenz auf den Inhalt der Variablen (des Objekts) `x`

- Verweisoperator `*`

  `*x` der unäre Verweisoperator `*` ermöglicht den Zugriff auf den Inhalt der Variablen (des Objekts), auf die der Zeiger `x` verweist

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2010
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

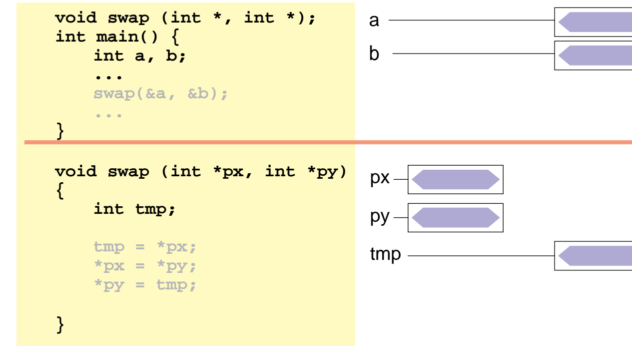
6 Zeiger als Funktionsargumente

- Parameter werden in C *by-value* übergeben
- die aufgerufene Funktion kann den aktuellen Parameter beim Aufrufer nicht verändern
- auch Zeiger werden *by-value* übergeben, d. h. die Funktion erhält lediglich eine Kopie des Adressverweises
- über diesen Verweis kann die Funktion jedoch mit Hilfe des `*`-Operators auf die zugehörige Variable zugreifen und sie verändern
  - ➥ *call-by-reference*

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

6 … Zeiger als Funktionsargumente (2)

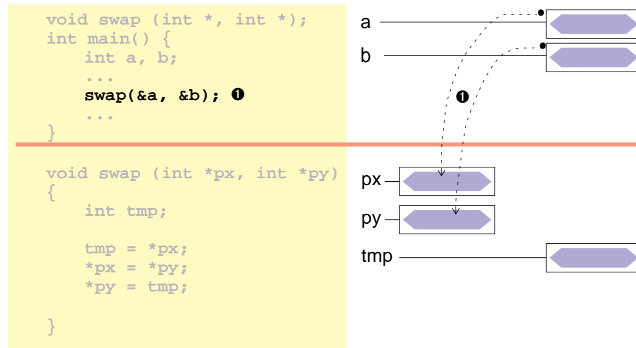
- Beispiel:



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

6 Zeiger als Funktionsargumente (2)

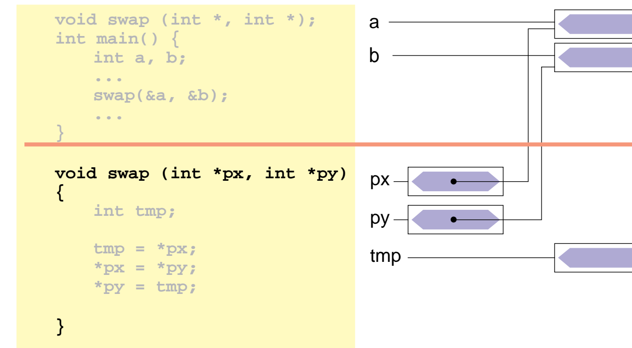
- Beispiel:



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

6 Zeiger als Funktionsargumente (2)

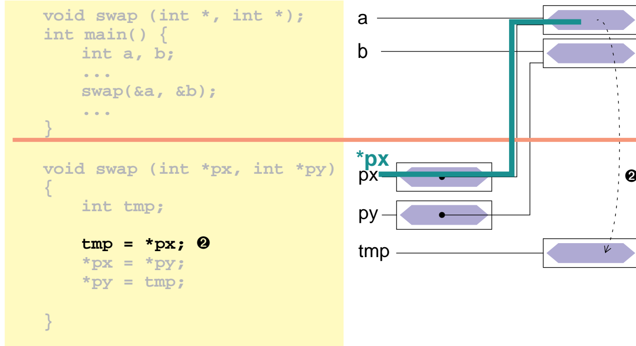
- Beispiel:



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

6 Zeiger als Funktionsargumente (2)

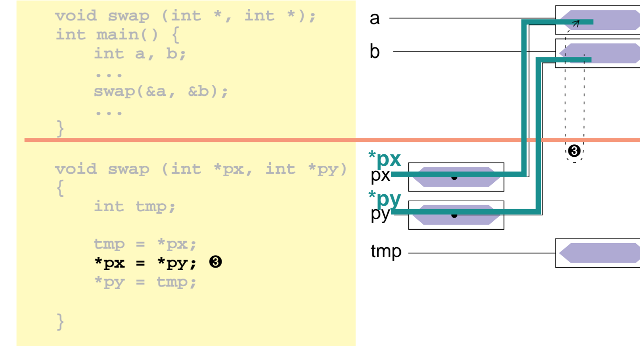
- Beispiel:



© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2010
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

6 Zeiger als Funktionsargumente (2)

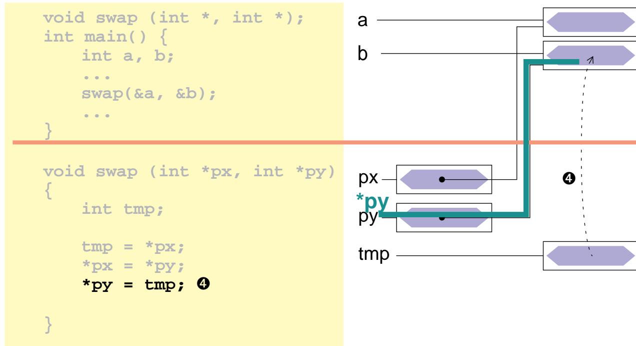
- Beispiel:



© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2010
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

6 Zeiger als Funktionsargumente (2)

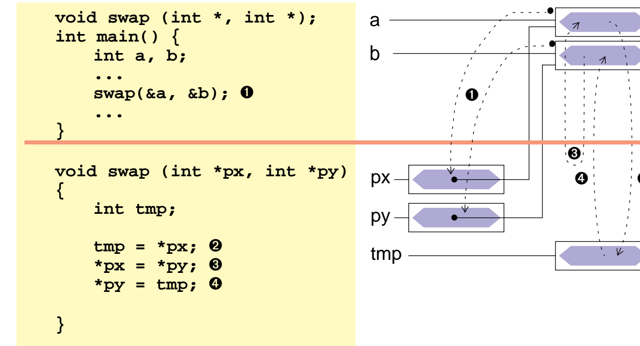
- Beispiel:



© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2010
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

6 Zeiger als Funktionsargumente (2)

- Beispiel:

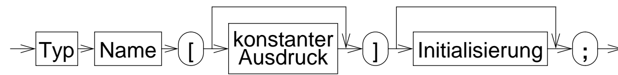


© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2010
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F.2 Felder

1 Eindimensionale Felder

- eine Reihe von Daten desselben Typs kann zu einem **Feld** zusammengefasst werden
- bei der Definition wird die Anzahl der Feldelemente angegeben, die Anzahl ist konstant!
- der Zugriff auf die Elemente erfolgt durch **Indizierung**, beginnend bei Null
- Definition eines Feldes

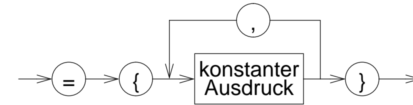


- Beispiele:

```
int x[5];
double f[20];
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Initialisierung eines Feldes



- Ein Feld kann durch eine Liste von konstanten Ausdrücken, die durch Komma getrennt sind, initialisiert werden

```
int prim[4] = {2, 3, 5, 7};
char name[5] = {’O’, ’t’, ’t’, ’o’, ’\0’};
```

- wird die explizite Felddimensionierung weggelassen, so bestimmt die Zahl der Initialisierungskonstanten die Feldgröße

```
int prim[] = {2, 3, 5, 7};
char name[] = {’O’, ’t’, ’t’, ’o’, ’\0’};
```

- werden zu wenig Initialisierungskonstanten angegeben, so werden die restlichen Elemente mit 0 initialisiert

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 … Initialisierung eines Feldes (2)

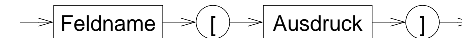
- Felder des Typs ***char*** können auch durch String-Literale initialisiert werden

```
char name1[5] = "Otto";
char name2[] = "Otto";
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Zugriffe auf Feldelemente

- Indizierung:



wobei: 0 <= Wert(Ausdruck) < Feldgröße

- Beispiele:

```
prim[0] == 2
prim[1] == 3
name[1] == ’t’
name[4] == ’\0’
```

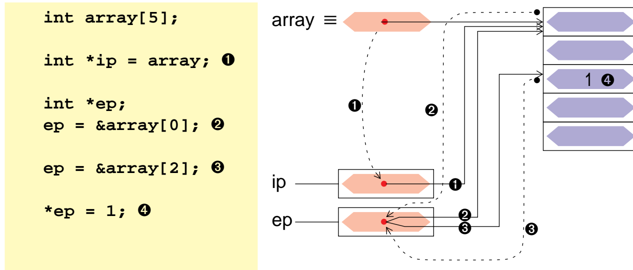
- Beispiel Vektoraddition:

```
float v1[4], v2[4], sum[4];
int i;
...
for ( i=0; i < 4; i++ )
    sum[i] = v1[i] + v2[i];
for ( i=0; i < 4; i++ )
    printf("sum[%d] = %f\n", i, sum[i]);
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F.3 Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

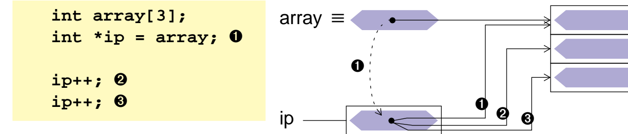


```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸

*ep = 1; ❹
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Arithmetik mit Adressen

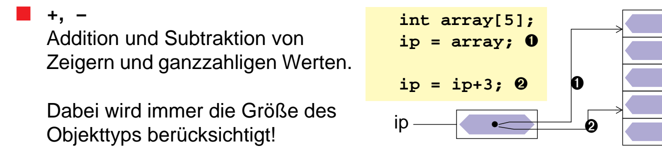
- ++ -Operator: Inkrement = nächstes Objekt



```
int array[3];
int *ip = array; ❶

ip++; ❷
ip++; ❸
```


- -- -Operator: Dekrement = vorheriges Objekt
- +, –
  Addition und Subtraktion von Zeigern und ganzzahligen Werten.

  Dabei wird immer die Größe des Objekttyps berücksichtigt!



```
int array[5];
ip = array; ❶

ip = ip+3; ❷
```

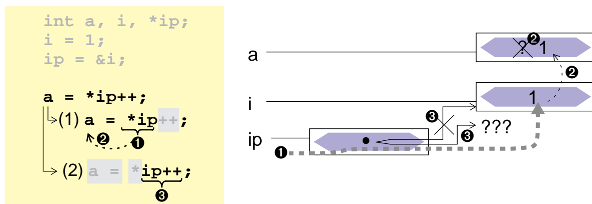

!!! **Achtung:** Assoziativität der Operatoren beachten !!

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Vorrangregeln bei Operatoren

Operatorklasse	Operatoren	Assoziativität
primär	**( ) Funktionsaufruf [ ]**	von links nach rechts
unär	**! ~ ++ -- + – * &**	von rechts nach links
multiplikativ	*** / %**	von links nach rechts
...		

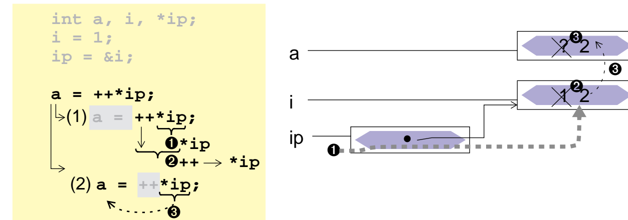
3 Beispiele



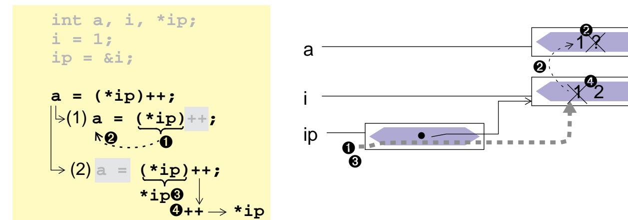
```
int a, i, *ip;
i = 1;
ip = &i;

a = *ip++;
 (1) a = *ip++;   ❷ ❶
 (2) a = *ip++;   ❸
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Beispiele (2)



```
int a, i, *ip;
i = 1;
ip = &i;

a = ++*ip;
 (1) a = ++*ip;
            ❶*ip
            ❷++ → *ip
 (2) a = ++*ip;
            ❸
```

```
int a, i, *ip;
i = 1;
ip = &i;

a = (*ip)++;
 (1) a = (*ip)++;
       ❷   ❶
 (2) a = (*ip)++;
          *ip❸
          ❹++ → *ip
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

4 Zeigerarithmetik und Felder

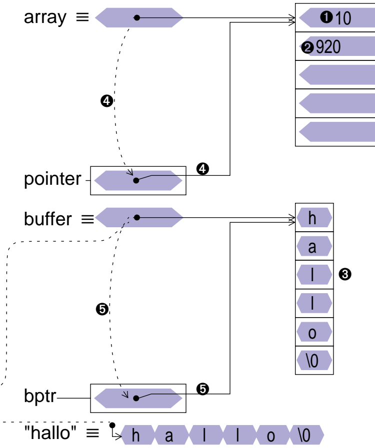
- Ein Feldname ist eine Konstante, für die Adresse des Feldanfangs
  - ➥ Feldname ist ein ganz normaler Zeiger
    - ➤ Operatoren für Zeiger anwendbar (`*`, `[ ]`)
  - ➥ aber keine Variable ➔ keine Modifikationen erlaubt
    - ➤ keine Zuweisung, kein `++`, `--`, `+=`, …
- es gilt:

```
int array[5]; /* → array ist Konstante für den Wert &array[0] */
int *ip = array;/* ≡ int *ip = &array[0] */
int *ep;

/* Folgende Zuweisungen sind äquivalent */
array[i] = 1;
ip[i] = 1;
*(ip+i) = 1;       /* Vorrang ! */
*(array+i) = 1;

ep = &array[i]; *ep = 1;
ep = array+i; *ep = 1;
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

4 Zeigerarithmetik und Felder

```
int array[5];
int *pointer;
char buffer[6];
char *bptr;

❶ array[0] = 10;
❷ array[1] = 920;
❸ strcpy(buffer,"hallo");
❹ pointer = array;
❺ bptr = buffer;
```



Fomale Parameter
der Funktion strcpy

ziel ❸

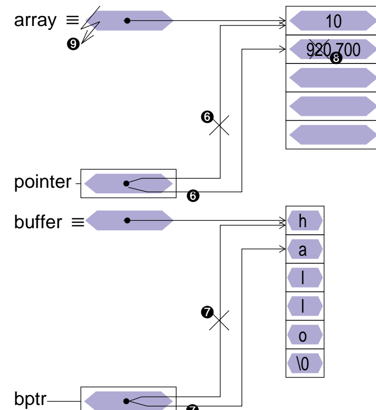
quelle ❸

"hallo" ≡ h a l l o \0

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

4 Zeigerarithmetik und Felder

```
int array[5];
int *pointer;
char buffer[6];
char *bptr;

❶ array[0] = 10;
❷ array[1] = 920;
❸ strcpy(buffer,"hallo");
❹ pointer = array;
❺ bptr = buffer;

❻ pointer++;
❼ bptr++;
❽ *pointer = 700;

❾ array++;
```



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

5 Vergleichsoperatoren und Adressen

- Neben den arithmetischen Operatoren lassen sich auch die Vergleichsoperatoren auf Zeiger (allgemein: Adressen) anwenden:

`<`	kleiner
`<=`	kleiner gleich
`>`	größer
`>=`	größer gleich
`==`	gleich
`!=`	ungleich

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F.4 Eindimensionale Felder als Funktionsparameter

- ganze Felder können in C **nicht *by-value*** übergeben werden
- wird einer Funktion ein Feldname als Parameter übergeben, wird damit der Zeiger auf das erste Element "by value" übergeben
  - ➥ die Funktion kann über den formalen Parameter (=Kopie des Zeigers) in gleicher Weise wie der Aufrufer auf die Feldelemente zugreifen (und diese verändern!)
- bei der Deklaration des formalen Parameters wird die Feldgröße weggelassen
  - ➤ die Feldgröße ist automatisch durch den aktuellen Parameter gegeben
  - ➤ die Funktion kennt die Feldgröße damit nicht
  - ➤ ggf. ist die Feldgröße über einen weiteren `int`-Parameter der Funktion explizit mitzuteilen
  - ➤ die Länge von Zeichenketten in `char`-Feldern kann normalerweise durch Suche nach dem `\0`-Zeichen bestimmt werden

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F.4 Eindimensionale Felder als Funktionsparameter (2)

- wird ein Feldparameter als `const` deklariert, können die Feldelemente innerhalb der Funktion nicht verändert werden
- Funktionsaufruf und Deklaration der formalen Parameter am Beispiel eines `int`-Feldes:

```
int a, b;
int feld[20];
func(a, feld, b);
…
int func(int p1, int p2[], int p3);
oder:
int func(int p1, int *p2, int p3);
```

- die Parameter-Deklarationen `int p2[]` und `int *p2` sind vollkommen äquivalent!
  - ➤ im Unterschied zu einer Variablendefinition!!!

```
int f[] = {1, 2, 3}; /* initialisiertes Feld mit 3 Elementen */
int f1[];            /* ohne Initialisierung und Dimension nicht erlaubt! */
int *p;              /* Zeiger auf einen int */
```

(Die Zeile `int f1[];` ist auf der Folie durchgestrichen.)

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F.5 Dynamische Speicherverwaltung

- Felder können (mit einer Ausnahme im C99-Standard) nur mit statischer Größe definiert werden
- Wird die Größe eines Feldes erst zur Laufzeit des Programm bekannt, kann der benötigte Speicherbereich dynamisch vom Betriebssystem angefordert werden: Funktion `malloc`
  - ➤ Ergebnis: Zeiger auf den Anfang des Speicherbereichs
  - ➤ Zeiger kann danach wie ein Feld verwendet werden (`[]`-Operator)
- `void *malloc(size_t size)`

```
int *feld;
int groesse;
...
feld = (int *) malloc(groesse * sizeof(int));
if (feld == NULL) {
        perror("malloc feld");
        exit(1);
}
for (i=0; i<groesse; i++) { feld[i] = 8; }
...
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F.5 Dynamische Speicherverwaltung (2)

- Dynamisch angeforderte Speicherbereiche können mit der `free`-Funktion wieder freigegeben werden
- `void free(void *ptr)`

```
double *dfeld;
int groesse;
...
dfeld = (double *) malloc(groesse * sizeof(double));
...
free(dfeld);
```

- die Schnittstellen der Funktionen sind in in der include-Datei stdlib.h definiert
  `#include <stdlib.h>`

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F.6 Explizite Typumwandlung — Cast-Operator

- C enthält Regeln für eine automatische Konvertierung unterschiedlicher Typen in einem Ausdruck (vgl. Abschnitt D.5.10)

Beispiel:

```
int i = 5;
float f = 0.2;
double d;
```

```
d = (i * f);
```

→float
→double

- In manchen Fällen wird eine explizite Typumwandlung benötigt (vor allem zur Umwandlung von Zeigern)
  - Syntax:

    (Typ) Variable

    Beispiele:

```
(int) a     (int *) a
(float) b   (char *) a
```

  - Beispiel:

```
feld = (int *) malloc(groesse * sizeof(int));
```

malloc liefert Ergebnis vom Typ (void *)

cast-Operator macht daraus den Typ (int *)

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F.7 sizeof-Operator

- In manchen Fällen ist es notwendig, die Größe (in Byte) einer Variablen oder Struktur zu ermitteln
  - ➤ z. B. zum Anfordern von Speicher für ein Feld (→ malloc)
- Syntax:

`sizeof x`	liefert die Größe des Objekts x in Bytes
`sizeof (`*Typ*`)`	liefert die Größe eines Objekts vom Typ *Typ* in Bytes

- Das Ergebnis ist vom Typ `size_t` (≡ `int`) (`#include <stddef.h>`!)
- Beispiel:

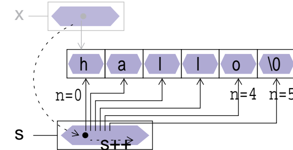
```
int a; size_t b;
b = sizeof a;          /* ⇒ b = 2 oder b = 4 */
b = sizeof(double)     /* ⇒ b = 8 */
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F.8 Zeiger, Felder und Zeichenketten

- Zeichenketten sind Felder von Einzelzeichen (`char`), die in der internen Darstellung durch ein `'\0'`–Zeichen abgeschlossen sind
- Beispiel: Länge eines Strings ermitteln — Aufruf `strlen(x);`

```
/* 1. Version */
int strlen(char *s)
{
    int n;
    for (n=0; *s != '\0'; s++)
        n++;
    return(n);
}
```



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F.9 Felder von Zeigern

- Auch von Zeigern können Felder gebildet werden
- Deklaration

```
int *pfeld[5];
int i = 1
int j;
```

- Zugriffe auf einen Zeiger des Feldes

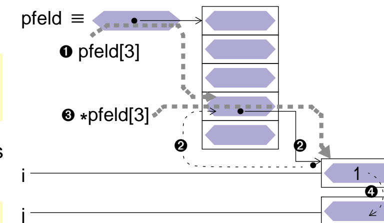
```
pfeld[3] = &i; ❷
```

❶

- Zugriffe auf das Objekt, auf das ein Zeiger des Feldes verweist

```
j = *pfeld[3]; ❹
```

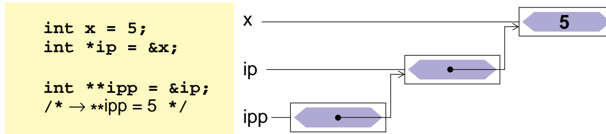
❶ ❸ ❹



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F.10 Zeiger auf Zeiger

- ein Zeiger kann auf eine Variable verweisen, die ihrerseits ein Zeiger ist



➤ wird vor allem bei der Parameterübergabe an Funktionen benötigt, wenn ein Zeiger "call bei reference" übergeben werden muss (z. B. swap-Funktion für Zeiger)

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F.11 Zeiger auf Funktionen

- Datentyp: Zeiger auf Funktion
  - ◆ Variablendef.: <*Rückgabetyp*> `(*`<*Variablenname*>`)(`<*Parameter*>`);`

```
int (*fptr)(int, char*);

int test1(int a, char *s) { printf("1: %d %s\n", a, s); }
int test2(int a, char *s) { printf("2: %d %s\n", a, s); }

fptr = test1;

fptr(42,"hallo");

fptr = test2;

fptr(42,"hallo");
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F.12 Strukturen

1 Motivation

- Felder fassen Daten eines einheitlichen Typs zusammen
  - ➤ ungeeignet für gemeinsame Handhabung von Daten unterschiedlichen Typs
- Beispiel: Daten eines Studenten

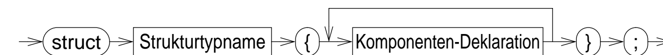
– Nachname	`char nachname[25];`
– Vorname	`char vorname[25];`
– Geburtsdatum	`char gebdatum[11];`
– Matrikelnummer	`int matrnr;`
– Übungsgruppennummer	`short gruppe;`
– Schein bestanden	`char best;`

- Möglichkeiten der Repräsentation in einem Programm
  - ◆ einzelne Variablen ➔ sehr umständlich, keine "Abstraktion"
  - ◆ Datenstrukturen

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Deklaration eines Strukturtyps

- Durch eine Strukturtyp-Deklaration wird dem Compiler der Aufbau einer Datenstruktur unter einem Namen bekanntgemacht
  - ➥ deklariert einen neuen Datentyp (wie `int` oder `float`)
- Syntax



- **Strukturtypname**
  - ◆ beliebiger Bezeichner, kein Schlüsselwort
  - ◆ kann in nachfolgenden Struktur-Definitionen verwendet werden
- **Komponenten-Deklaration**
  - ◆ entspricht normaler Variablen-Definition, aber keine Initialisierung!
  - ◆ in verschiedenen Strukturen dürfen die gleichen Komponentennamen verwendet werden (eigener Namensraum pro Strukturtyp)

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Deklaration eines Strukturtyps (2)

- Beispiele

```
struct student {
    char nachname[25];
    char vorname[25];
    char gebdatum[11];
    int matrnr;
    short gruppe;
    char best;
};
```

```
struct komplex {
    double re;
    double im;
};
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Definition einer Struktur

- Die Definition einer Struktur entspricht einer Variablen-Definition
  - ◆ Name der Struktur zusammen mit dem Datentyp bekanntmachen
  - ◆ Speicherplatz anlegen
- Eine Struktur ist eine Variable, die ihre Komponentenvariablen umfasst
- Syntax

- Beispiele

```
struct student stud1, stud2;
struct komplex c1, c2, c3;
```

- Strukturdeklaration und -definition können auch in einem Schritt vorgenommen werden

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

4 Zugriff auf Strukturkomponenten

- **.**-Operator
  - ➤ `x.y` ≡ Zugriff auf die Komponente `y` der Struktur `x`
  - ➤ `x.y` verhält sich wie eine normale Variable vom Typ der Strukturkomponenten `y` der Struktur `x`
- Beispiele

```
struct komplex c1, c2, c3;
…
c3.re = c1.re + c2.re;
c3.im = c1.im + c2.im;

struct student stud1;
…
if (stud1.matrnr < 1500000) {
    stud1.best = 'y';
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

5 Initialisieren von Strukturen

- Strukturen können — wie Variablen und Felder — bei der Definition initialisiert werden
- Beispiele

```
struct student stud1 = {
    "Meier", "Hans", "24.01.1970", 1533180, 5, 'n'
};

struct komplex c1 = {1.2, 0.8}, c2 = {0.5, 0.33};
```

!!! **Vorsicht**
bei Zugriffen auf eine Struktur werden die Komponenten durch die Komponentennamen identifiziert,
bei der Initialisierung jedoch nur durch die Postion

➥ potentielle Fehlerquelle bei Änderungen der Strukturtyp-Deklaration

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

6 Strukturen als Funktionsparameter

- Strukturen können wie normale Variablen an Funktionen übergeben werden
  - ◆ Übergabesemantik: **call by value**
    - ➤ Funktion erhält eine Kopie der Struktur
    - ➤ auch wenn die Struktur ein Feld enthält, wird dieses komplett kopiert!
    - !!! Unterschied zur direkten Übergabe eines Feldes
- Strukturen können auch Ergebnis einer Funktion sein
  - ➤ Möglichkeit mehrere Werte im Rückgabeparameter zu transportieren
- Beispiel

```
struct komplex komp_add(struct komplex x, struct komplex y) {
    struct komplex ergebnis;
    ergebnis.re = x.re + y.re;
    ergebnis.im = x.im + y.im;
    return(ergebnis);
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

7 Zeiger auf Strukturen

- Konzept analog zu "Zeiger auf Variablen"
  - ➤ Adresse einer Struktur mit &-Operator zu bestimmen
  - ➤ Name eines Feldes von Strukturen = Zeiger auf erste Struktur im Feld
  - ➤ Zeigerarithmetik berücksichtigt Strukturgröße
- Beispiele

```
struct student stud1;
struct student gruppe8[35];
struct student *pstud;
pstud = &stud1;          /* ⇒ pstud → stud1 */
pstud = gruppe8;         /* ⇒ pstud → gruppe8[0] */
pstud++;                 /* ⇒ pstud → gruppe8[1] */
pstud += 12;             /* ⇒ pstud → gruppe8[13] */
```

- Besondere Bedeutung zum Aufbau
  - ➥ rekursiver Strukturen

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

7 Zeiger auf Strukturen (2)

- Zugriff auf Strukturkomponenten über einen Zeiger
- Bekannte Vorgehensweise
  - ➤ `*`-Operator liefert die Struktur
  - ➤ `.`-Operator zum Zugriff auf Komponente
  - ➤ Operatorenvorrang beachten
  - ➥ `(*pstud).best = 'n';` unleserlich!
- Syntaktische Verschönerung
  - ➥ `->`-Operator

```
pstud->best = 'n';
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

8 Felder von Strukturen

- Von Strukturen können — wie von normale Datentypen — Felder gebildet werden
- Beispiel

```
struct student gruppe8[35];
int i;
for (i=0; i<35; i++) {
    printf("Nachname %d. Stud.: ", i);
    scanf("%s", gruppe8[i].nachname);
    …
    gruppe8[i].gruppe = 8;

    if (gruppe8[i].matrnr < 1500000) {
        gruppe8[i].best = 'y';
    } else {
        gruppe8[i].best = 'n';
    }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

9 Strukturen in Strukturen

- Die Komponenten einer Struktur können wieder Strukturen sein
- Beispiel

```
struct name {
    char nachname[25];
    char vorname[25];
};

struct student {
    struct name name;
    char gebdatum[11];
    int matrnr;
    short gruppe;
    char best;
}

struct prof {
    struct name pname;
    char gebdatum[11];
    int lehrstuhlnr;
}

struct student stud1;
strcpy(stud1.name.nachname, "Meier");
if (stud1.name.nachname[0] == 'M') {
    …
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

10 Rekursive Strukturen

- Strukturen in Strukturen sind erlaubt — aber
  - ◆ die Größe einer Struktur muss vom Compiler ausgerechnet werden können
    - ➤ Problem: eine Struktur enthält sich selbst

```
struct liste {
    struct student stud;
    struct liste rest;
};
```

falsch!

  - ◆ die Größe eines Zeigers ist bekannt (meist 4 Byte)
    - ➤ eine Struktur kann einen Zeiger auf eine gleichartige Struktur enthalten

```
struct liste {
    struct student stud;
    struct liste *rest;
};
```

➥ Programmieren rekursiver Datenstrukturen

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F.13 Verbundstrukturen — Unions

- In einer Struktur liegen die einzelnen Komponenten hintereinander, in einem Verbund liegen sie übereinander
  - ➤ die gleichen Speicherzellen können unterschiedlich angesprochen werden
  - ➤ Beispiel: ein int-Wert (4 Byte) und die einzelnen Bytes des int-Werts

```
union intbytes {
    int ivalue;
    char bvalue[4];
} u1;
...
u1.ivalue = 259000;
printf("Wert=%d, Byte0=%d, Byte1=%d, Byte2=%d, Byte3=%d\n",
    u1.ivalue, u1.bvalue[0], u1.bvalue[1], u1.bvalue[2], u1.bvalue[3]);
```

- Einsatz nur in sehr speziellen Fällen sinnvoll
  konkretes Wissen über die Speicherorganisation unbedingt erforderlich!

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F.14 Bitfelder

- Bitfelder sind Strukturkomponenten bei denen die Zahl der für die Speicherung verwendeten Bits festgelegt werden kann.
- Anwendung z. B. um auf einzelnen Bits eines Registers zuzugreifen

```
struct cregister {
    unsigned int protection : 1;
    unsigned int interrupt_mask : 3;
    unsigned int enable_read : 1;
    unsigned int enable_write : 1;
    unsigned int pedding : 2;
    unsigned int address : 8;
};
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F.14 Bitfelder (2)

- Struktur mit Bitfeld kann ihrerseits Teil einer Union sein
  - ➤ Zugriff auf Register als Ganzes und auf die einzelnen Bits

```
union cregister {
    unsigned short all;
    struct bits {
        unsigned int protection : 1;
        unsigned int interrupt_mask : 3;
        unsigned int enable_read : 1;
        unsigned int enable_write : 1;
        unsigned int pedding : 2;
        unsigned int address : 8;
    };
};
```

- Adresse und Aufbau eines Registers steht üblicherweise in der Hardwarebeschreibung.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F.14 Bitfelder (3)

- Beispiel:
  - ➤ Adresse auf Register anlegen,
    Registerinhalt sichern
    Bits verändern
    Registerinhalt wieder herstellen

```
union cregister *creg;
unsigned short oldvale;
creg = (union cregister *)0x2400; /* Addr. aus Manual */
oldvalue = creg->all;            /* Wert sichern */
creg->bits.protection = 0;
creg->bits.enable_read = 1;
creg->bits.address = 0x40;
...
creg->all = oldvalue;            /* Wert restaurieren */
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F.15 Typedef

- Typedef erlaubt die Definition neuer Typen
  - ➤ neuer Typ kann danach wie die Standardtypen (int, char, ...) genutzt werden
- Beispiele:

```
typdef int Laenge;
Laenge l = 5;
Laenge *pl; Laenge fl[20];
```

```
typdef struct student Student;
Student s; Student *ps1;
s.matrnr = 1234567; ps1 = &s; ps1->best = 'n';
```

```
typdef struct student *Studptr;
Studptr ps2;
ps2 = &s; ps2->best = 'n';
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F.16 Enumerations

- Enumerations sind Datentypen, für die explizit angegeben wird, welche Werte (symbolische Namen) sie annehmen können
  - ➤ interne Darstellung als int (Werte beginnend ab 0 oder explizit angebbar)
- Beispiele:

```
enum led {
  RED0, YELLOW0, GREEN0, BLUE0, RED1, YELLOW1, GREEN1, BLUE1
};
enum led signal;
signal = GREEN0;
```

```
typedef enum {
  BUTTON0 = 4, BUTTON1 = 8
} BUTTON;
BUTTON b = BUTTON0;
```

- Effekt auch mit `#define` - Makros erreichbar
  - ➤ Vorteil von enum: Compiler kennt den Typ und ***könnte*** Ausdrücke prüfen (Nachteil: die meisten Compiler tun es nicht ⇒ `enum` ≡ `int`)

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.