Programmieren in C
für Elektrotechniker

Kapitel 7: Ausdrücke und Operatoren (Vertiefung)

- **Ausdrücke**
- Operatoren
- Bitmanipulationen
- Typumwandlung
- Pointer und Arrays (Teil 2)
- Unions und Bitfelder

Ausdrücke

- **Ausdrücke sind**
  - Operanden (Konstanten, Zeichenketten, Bezeichner von Variablen und Funktionen)
  - über Operatoren verknüpfte Teilausdrücke
  - geklammerte Teilausdrücke.
- Ausdrücke besitzen einen Rückgabewert.
- Klammern beeinflussen Auswertereihenfolge.

- **Verwendung von Ausdrücken**
  - Berechnung von Werten.
  - Erzeugung von Seiteneffekten (Nebeneffekten)
    → *gefährlich.*
  - Bereitstellung von Speicherobjekten (Variablen oder Funktionen).

- **Ausdrücke und Anweisungen**
  - **Ausdrücke**
    - Haben immer einen Rückgabewert *(mit Typ)*.
    - Rückgabewert hängt vom Typ der Operanden ab.
      → Compiler führt Typwandlung von Operanden durch *(s.u.)*
      Beispiel: `3 + 4.5` → Ergebnis: `7.5` (`double`)
    - Können Teil eines komplexeren Ausdrucks sein.

- **Ausdrücke und Anweisungen**
  - **Anweisungen**
    - Haben keinen Rückgabewert.
    - Werden mit ‚`;`' abgeschlossen.
    - Können nicht Teil eines komplexen Ausdrucks sein.
    - In C: - Selektionsanweisung
      - Iterationsanweisung
      - Sprunganweisung
      - Ausdrucksanweisung

      } *vgl. Kapitel 4*

      → in C kann jeder Ausdruck eine Anweisung werden, z.B.

```
int i = 0;
5 + 5;      → möglich, aber sinnlos
i++;        → i = i + 1; - Ausnutzen des Seiteneffekts
```

- **Seiteneffekte (Nebeneffekte)**
  - Während der Auswertung eines Ausdrucks können Variablenwerte nebenbei geändert werden.
  - Beispiel:
```
int i = 1;
int j;
j = i++;
```
→ j = _____, i = _____
  - Seiteneffekte werden erst nach Auswertung des Ausdrucks wirksam.

- **Auswertereihenfolge**
  - Prinzipiell wie in der Mathematik: „Punkt vor Strich“
    `5 + 2 * 3` → `5 + (2 * 3)`
  - Genauer: Ausdrücke werden in folgender Reihenfolge ausgewertet
    1. Teilausdrücke in Klammern.
    2. Einstellige Operatoren (von rechts nach links): gleiche Priorität
       → zuerst postix, dann präfix.

       Beispiel: `*p++` → `*(p++)`

       → Rückgabewert?

       1. **p++**: - gibt **p** zurück
          - erhöht dann **p** um 1 Speicherstelle
       2. *: - gibt Speicherobjekt von **p** zurück
          (aber: **p** zeigt eine Speicherstelle weiter)

       Meist jedoch gewünscht:
       (*p)++: Speicherobjekt von **p** wird inkrementiert

- **Auswertereihenfolge**
  - Prinzipiell wie in der Mathematik: „Punkt vor Strich“
    `5 + 2 * 3` → `5 + (2 * 3)`
    - Genauer: Ausdrücke werden in folgender Reihenfolge ausgewertet
      1. Teilausdrücke in Klammern.
      2. Einstellige Operatoren (von rechts nach links): gleiche Priorität
         → zuerst postfix, dann präfix.
      3. Mehrstellige Operatoren nach Prioritätstabelle *(s.u.)*.
         - Bei gleicher Priorität werden
           - linksassoziative Operatoren von links nach rechts
           - rechtsassoziative Operatoren von rechts nach links

           ausgewertet.

           Beispiele: **+/-** sind linksassoziativ
           `a + b – c` → `(a + b) – c`
           = ist rechtsassoziativ
           `a = b = c` → `a = (b = c)`

- **Auswertereihenfolge**
  - Achtung: Reihenfolge der Operatoren ist unabhängig von der Reihenfolge der Auswertung der Operanden.
    - Z.B. ist bei **`A + B + C`** nicht festgelegt, in welcher Reihenfolge **A**, **B**, **C** berechnet werden.
    - Beispiel:

      A **B**
      `x = n++ - n;`

      - **A** vor **B**:
        ```
        A = n; n = n + 1;
        → B = n + 1;
        → x = n – (n + 1) = -1
        ```
      - **B** vor **A**:
        ```
        B = n;
        → A = n; n = n + 1;
        → x = n – n = 0;
        ```

Programmieren in C
für Elektrotechniker

Kapitel 7: Ausdrücke und Operatoren (Vertiefung)

- Ausdrücke
- **Operatoren**
- Bitmanipulationen
- Typumwandlung
- Pointer und Arrays (Teil 2)
- Unions und Bitfelder

Operatoren (Wdh.)

- **Operatoren in C**

```
() [] -> .
! ~ ++ -- + - * & (Typname) sizeof
/ % << >> < <= > >= == != ^ | &&
|| ?: = += -= *= /= %= &= ^= |= <<= >>= ,
```

- **Operatorklassen**
  - einstellig (unär)
    - präfix (z.B. `++i`)
    - postfix (z.B. `i++`)
  - zweistellig (binär)
    (z.B. `i + 1`, `x = y`)
  - dreistellig (nur: `?:`)

Operatoren in C

- **Einstellige arithmetische Operatoren (Wdh.)**
  - Vorzeichenoperatoren +A, -A
    - keine Nebeneffekte
  - Präfix-Inkrement-/-Dekrement-Operatoren ++A, --A
    - nur auf modifizierbare L-Werte anwendbar
    - Ergebnis: A + 1, A - 1
    - Seiteneffekt: Wert des Operanden wird um 1 inkrementiert
    - Inkrement bei Pointern: Erhöhung um Objektgröße
  - Postfix-Inkrement-/-Dekrement-Operatoren A++, A--
    - Wie Präfix-Inkrement-/-Dekrement, jedoch Ergebnis: A
- **Zweistellige arithmetische Operatoren (Wdh.)**
  - Addition (A+B), Subtraktion (A-B), Multiplikation (A*B), Division (A/B), Rest (A%B)
    - keine Nebeneffekte
    - Ergebnistyp ist abhängig von den Operanden (5/3 → int, 5/3.0 → double) → automatisches Casting *(s.u.)*

- **Zuweisungsoperatoren**
  - Zuweisung „=“ und Kombination mit anderen Operatoren ohne Zwischenraum
    - Additions-Zuweisungsoperator: `A += B`
    - Subtraktions-Zuweisungsoperator: `A -= B`
    - Multiplikations-Zuweisungsoperator: `A *= B`
    - Divisions-Zuweisungsoperator: `A /= B`
    - Restwert-Zuweisungsoperator: `A %= B`
    - Bitweises-UND-Zuweisungsoperator: `A &= B`
    - Bitweises-ODER-Zuweisungsoperator: `A |= B`
    - Bitweises-XOR-Zuweisungsoperator: `A ^= B`
    - Linksschiebe-Zuweisungsoperator: `A <<= B`
    - Rechtsschiebe-Zuweisungsoperator: `A >>= B`

- **Zuweisungsoperatoren**
  - Zuweisung „=“
    - In C: Ausdruck *(in anderen Sprachen meist Anweisung)*
      - Rückgabewert: Wert der rechten Seite
      - Seiteneffekt: Variable der linken Seite erhält diesen Wert
    - rechtsassoziativer Operator
      Beispiel: **`a = b = c`** entspr. **`a = (b = c)`**
      → Rückgabewert: Wert von **c**
      → Seiteneffekt: zuerst erhält **b**, dann **a** den Wert von **c**
    - geringe Priorität *(s.u.)*
    - ggf. werden Ausdrücke der rechten Seite in den Typ der linken Seite gewandelt.
  - Zuweisung „+=“
    - **`a += b;`** entspr. **`a = a + b;`**
      - Rückgabewert: Wert von **`a + b`**
      - **`a += 1;`** entspr. **`++a`** *(nicht a++)*
    - Früher hatte **`a += b;`** schnelleren Code erzeugt als **`a = a + b;`** da der Compiler **a** nur einmal auswerten muss.
      Heute machen dies optimierende Compiler automatisch.

- **Relationale Operatoren (Wdh.)**
  - Vergleichsoperatoren
    - Gleichheitsoperator: **==**
    - Ungleichheitsoperator: **!=**
    - Größeroperator: **>**
    - Kleineroperator: **<**
    - Größergleichoperator: **>=**
    - Kleinergleichoperator: **<=**
  - Rückgabewert: int *(0 oder 1)*
  - ggf. implizite Typwandlung bei ungleichen Operanden
  - unterschiedliche Prioritäten *(s.u.)*

Logische Operatoren (Wdh.)

- Dürfen nicht mit den logischen Bitoperationen verwechselt werden.
  - Operator für das logische UND: **&&**
  - Operator für das logische ODER: **||**
  - Logischer Negationsoperator: **!**
  - Rückgabewert: int *(0 oder 1)*
  - Kein fester Datentyp „boolean“ in C
    → Verwendung von Integer *(0 = false, sonst true)*
  - Semantik der Funktionen in Kapitel 5 über Wertetabellen beschrieben.
  - Operanden werden von links nach rechts ausgewertet
    → Nebeneffekte der Auswertung der linken Operanden vor Auswertung der rechten Operanden.

Bit-Operatoren (→ später mehr)

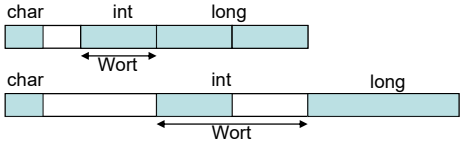
- C kennt Bitmanipulationen *(vgl. Assembler)*.
  - UND/ODER-Operator für Bits: **&** **|**
  - XOR-Operator für Bits: **^**
  - Negationsoperator für Bits: **~**
  - Links-/Rechtsshift-Operator: **<<** **>>**
  - Nur für ganze Zahlen
    → sinnvoll nur für natürliche Zahlen, da negative Zahlen systemabhängig.
  - Bitweise Verknüpfung des gesamten Datenworts.
  - Beispiele:

```
0 & 1      = _____        0 | 1   = _____
14 & 1     = _____        14 | 1  = _____
var & var  = _____        var | 0 = _____
unsigned char a, b;
a = 8;  → 0000 1000
b = a << 3 → ________     b = a >> 3; → __________
char a;
a = -8; → 1111 1000       a = a >> 3; → __________
```

sizeof-Operator

- Ermittelt die Größe von Datenobjekten im Speicher
  → **`sizeof Ausdruck`** oder
  → **`sizeof (Typname)`**
- Rückgabewert: ganze Zahl
  → Größe des Objekts in Bytes
  (Typdefinition von size_t in <stddef.h> - meist unsigned oder unsigned long)
- Meist zur Größenbestimmung von Strukturen *(s.u.)*, da die Elemente vom Compiler beliebig ausgerichtet werden können.

```
struct beispiel {
   char c1;
   int i;
   long int l;
}
```



16-Bit-Rechner
sizeof(beispiel) = 8

32-Bit-Rechner
sizeof(beispiel) = 12

sizeof-Operator

- Übergebene Ausdrücke werden nicht ausgewertet
  → es wird lediglich der Typ und dann seine Größe bestimmt.
- Beispiele:
  - **`sizeof zahl++`**
    → ändert **`zahl`** nicht *(kein Seiteneffekt)*,
    da Ausdruck „**`zahl++`**“ nicht ausgewertet wird
  - Berechnung der Anzahl an Array-Elementen
    (falls zur Übersetzungszeit bekannt):

```
int a [] = {4, 7, 11, 0, 8, 15};
size_t length = sizeof (a) / sizeof (a[0]);
```

→ **`length`** ==_____

- **Komma-Operator A, B**
  - Nur zur Berechnung von Seiteneffekten *(→ selten eingesetzt)*.
  - Von links nach rechts ausgewertet.
  - Rückgabewert: Wert und Typ von B.
  - Beispiel:

```
int main (void) {
   char text1 [4] = "EIN";
   char text2 [4];
   int index1, index2;

   for (index2 = 0, index1 = 2;
        index1 >= 0;
        index2++, index1--) {
      text2 [index2] = text1 [index1];
   }
   text2 [index2] = '\0';
   printf ("\n%s\n%s\n", text1, text2);
   return 0;
}
```


- **Bedingungsoperator A ? B : C**
  - Vergleichbar mit if-else-Anweisung.
  - Einziger Operator mit 3 Operanden.
  - Zunächst Auswertung von A (inkl. Seiteneffekte).
    Ist A wahr, wird Ausdruck B ausgewertet, ansonsten C.
  - Rechtsassoziativ.
  - Rückgabewert: Ergebnis von B bzw. C
    → bei unterschiedlichen Typen wird immer der „höhere Typ“ von B und C zurückgegeben.
  - Beispiele:

```
1 == 1 ? 0 : 1    → _____
0 ? 0 : 1         → _____
(3>4) ? 5.0 : 6   → _____
if (A) return B;
else return C;    entspr. return (A) ? B : C;
A ? B : C ? D : E ? F : G
                  entspr. A ? B : (C ? D : (E ? F : G))
```

Operator-Prioritäten und -Assoziativitäten

Priorität	Operatoren	Beschreibung	Assoziativität
1	( ) [ ] -> .	Funktionsaufruf Array-Index Komponentenzugriff	links links links
2	! ~ ++ -- sizeof + - (Typname) * &	Negation (logisch, bitweise) Inkrement, Dekrement Vorzeichen Cast *(s.u.)* Dereferenzierung, Adresse	rechts rechts rechts rechts rechts rechts
3	* / %	Multiplikation, Division, Rest	links
4	+ -	Addition, Subtraktion	links
5	<< >>	bitweises Schieben	links
6	< <= > >=	Vergleich	links
7	== !=	(Un-) Gleichheit	links
8	&	bitweises UND	links
9	^	bitweises XOR	links
10	|	bitweises ODER	links
11	&&	logisches UND	links
12	||	logisches ODER	links
13	?:	bedingte Auswertung	rechts
14	= += -= *= /= %= &= ^= |= <<= >>=	Wertzuweisung kombinierte Wertzuweisung	rechts rechts
15	,	Kommaoperator	links

- **Implementierungsabhängige Reihenfolge**
  - Prioritäten und Assoziativitäten regeln die Reihenfolge der Verknüpfungen, aber nicht die Reihenfolge der Operandenauswertungen.
    → Ausnahme: logisches UND/ODER, Kommaoperator, Bedingungsoperator.
    - Beispiel: `a = f1(....) * f2(....) * f3(....);`

      Funktionsergebnisse werden von links nach rechts multipliziert, C schreibt aber nicht vor, in welcher Reihenfolge die Funktionen f1, f2, f3 ausgewertet werden.
      → Problem, wenn die Funktionen auf die selben globalen Variablen zugreifen.
  - Reihenfolge der Bewertung aktueller Parameter bei einem Funktionsaufruf ist auch undefiniert.
    - Beispiel:

```
int potenz (int a, int b) {
    erg = a;
    for (i=0; i<b; i++)
        erg *= a;
    return erg;
}
```

Aufruf:

```
int x = 3;
y = potenz (x++, x++);
```

→ y = 4^3 oder y = 3^4 oder y = 4^5???

Programmieren in C
für Elektrotechniker

Kapitel 7: Ausdrücke und Operatoren (Vertiefung)

- Ausdrücke
- Operatoren
- **Bitmanipulationen**
- Typumwandlung
- Pointer und Arrays (Teil 2)
- Unions und Bitfelder

- **Bit-Operatoren**
  - Systemprogrammierung: oft notwendig, einzelne Bits zu lesen oder zu ändern
    Beispiele: - Ausgänge eines Mikrocontrollers
    - Zugriffsrechte einer Datei
    - Statuswort eines E/A-Kanals
  - C kennt folgende Bitoperationen *(s.o.)*.

```
unsigned char a = 0x25;
unsigned char b = 0x0c;
```

a	0	0	1	0	0	1	0	1
b	0	0	0	0	1	1	0	0
a & b	0	0	0	0	0	1	0	0
a | b	0	0	1	0	1	1	0	1
a ^ b	0	0	1	0	1	0	0	1
~a	1	1	0	1	1	0	1	0
a << 3	0	0	1	0	1	0	0	0
a >> 3	0	0	0	0	0	1	0	0

- **Bit-Masken**

  Bit-Masken erlauben es, einzelne Bits eines Datenworts zu manipulieren (setzen, zurücksetzen, invertieren)

  - **Bits löschen**

    UND-Operator ‚&' zusammen mit 0-Bits einer Maske

    Beispiel:
    ```
    short a = 0xfa06;      1111101000000110
    a = a & 0xf4a;         0000111101001010
                           0000101000000010
    ```

  - **Bits setzen**

    OR-Operator ‚|' zusammen mit 1-Bits einer Maske

    Beispiel:
    ```
    short a = 0xfa06;      1111101000000110
    a = a | 0xf4a;         0000111101001010
                           1111111101001110
    ```

  - **Bits invertieren**

    XOR-Operator ‚^' zusammen mit 1-Bits einer Maske

    Beispiel:
    ```
    short a = 0xfa06;      1111101000000110
    a = a & 0xf4a;         0000111101001010
                           1111010101001100
    ```


- **Bit-Masken**

  Bit-Masken erlauben es, einzelne Bits eines Datenworts zu manipulieren (setzen, zurücksetzen, invertieren)

  - **Bits abfragen**

    UND-Operator ‚&' zur Abfrage eines einzelnen Bits

    Beispiel: Abfrage, ob 5. Bit von rechts gesetzt ist
    ```
    if (a & 0x10) ...
    ```

Bit-Masken

Bit-Masken erlauben es, einzelne Bits eines Datenworts zu manipulieren (setzen, zurücksetzen, invertieren)

- **Beispiel: Zugriff auf Hardwareadressen**

  Bisher: keine direkte Manipulation von Pointer-Werten.
  Systemprogrammierung: Teilweise direkte Verwendung von Adressen.
  Beispiel: Statusregister der 1. Parallelports des Intel-PC an Adresse 0x379.
  Druckertreiber: Manipulation des Werts an dieser Adresse wichtig.

```
unsigned char * printerStateRegister;
unsigned char;

printerStateRegister = 0x379;

printerState = * printerStateRegister;

if (printerState & 0%10000000) {
    // Drucker ist frei
} else  {
    // Drucker ist BUSY
}
```


Shift-Operationen

- zur Bitmanipulation
- zur Arithmetik (Multiplikation und Division mit Zweierpotenzen)
- **Links-Shift**

  MSB entfallen, LSB werden Nullen nachgeschoben

  Beispiel:
  ```
  short a = 0xfa06;      1111101000000110
  a = a << 4;            1010000001100000
  ```

- **Rechts-Shift**

  LSB entfallen, MSB werden Nullen (→ logischer Shift) oder Vorzeichen (→ arithmetischer Shift) nachgeschoben
  ➔ Compiler-abhängig (→ möglichst nur unsigned Integer verwenden)

  Beispiel:
  ```
  short a = 0xfa06;      1111101000000110
  a = a >> 4;            0000111110100000  log. Shift
                         1111111110100000  arithm. Shift
  ```

- **Multiplikation und Division mit Zweierpotenzen**

  → Schneller als Multiplikation/Division
  Beispiel:
  ```
  int a = 5;
  a = a << 2;            a = 5 * 4 = 20
  ```

Programmieren in C
für Elektrotechniker

Kapitel 7: Ausdrücke und Operatoren (Vertiefung)

- Ausdrücke
- Operatoren
- Bitmanipulationen
- **Typumwandlung**
- Pointer und Arrays (Teil 2)
- Unions und Bitfelder

Typwandlung

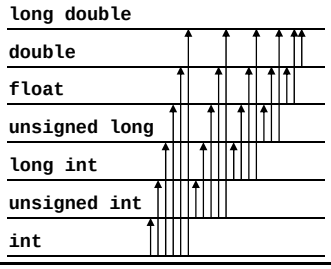
- **Explizit: Cast-Operator (Typname) A**
  - Explizite Typwandlung zwischen skalaren Variablen untereinander und zu **`void`**.
  - Gleiche Regeln wie bei impliziter Typwandlung *(s.u.)*.
    Mögliche Wandlungen:

von \ nach	char, short, int, long	float, double	Pointer	void
char, short, int, long	✓	✓	✓	✓
float, double	✓	✓		✓
Pointer	✓		✓	✓
void			✓	

  - Beispiele:
    - `int a = 1;`
      `float b = (float) a;` → 1.0
    - `(void) printf ("%d", x);` → explizit kein Rückgabewert.

- **Implizite Typwandlung**
  - Compiler führt implizite Typwandlung durch, falls die Operanden eines Ausdrucks ungleichen Typs sind.
  - Nur zwischen verträglichen Typen möglich *(sonst Compiler-Fehlermeldung)* bei
    - Pointer zu **`void`**,
    - Arithmetischen Operanden *(immer „kleinerer" Typ in „größeren")*,
    - Zuweisungen *(auch „größeren" Typ in „kleineren")*,
    - Rückgabewert und Parametern von Funktionen
      → wie bei Zuweisungen in Typ des formalen Parameters bzw. Funktionstyps.

- **Implizite Typwandlung**
  - **Arithmetische Konvertierung**
    - Typwandlung bei allen 2-stelligen Operatoren, außer **`&&`**, **`||`** *(auch bei `?:`)*
    - Integer-Erweiterung
      - In C wird mindestens mit **`int`** gerechnet *(→ nur `(unsigned) int | long`)*
        `(unsigned) char` und **`short`** → **`int`**
        `unsigned short` → **`unsigned int`**, falls **`short`** ≡ **`int`**
        → **`int`**, falls **`short`** ≠ **`int`**
    - Hierarchie von Datentypen



Beispiel:

```
4 * 2 * 3L + 1.1
int
    long
        double
```

- **Implizite Typwandlung**
  - **Zuweisungen (auch Rückgabe und Parameter von Funktionen)**
    - Rechter Operand wird in Typ des linken Operanden gewandelt, falls beide Operanden verträglich *(sonst Fehlermeldung).*
    - Arithmetische Operanden sind immer verträglich.
      → Gefährlich, falls „größerer" Typ in „kleineren" gewandelt wird *(s.u.)*.
    - **Vorschriften**
      - kurzer Integer → langer Integer
        z.B. - **`unsigned int`** → **`unsigned long`**
        - **`char`** → **`short`**
        - **`unsigned char`** → **`int`**
        → signed: Replizieren des Vorzeichens
        unsigned: 0…0 voranstellen.
      - Integer → unsigned Integer gleicher Länge
        → unverändert, aber neue Interpretation
      - langer Integer → kurzer Integer
        → unverändert, falls kurzer Integer ausreichend, sonst undefiniert.
      - langer Integer → kurzer unsigned Integer
        → Abschneiden der vorderen Bits und neue Interpretation.

- **Implizite Typwandlung**
  - **Zuweisungen (auch Rückgabe und Parameter von Funktionen)**
    - Rechter Operand wird in Typ des linken Operanden gewandelt, falls beide Operanden verträglich *(sonst Fehlermeldung).*
    - Arithmetische Operanden sind immer verträglich.
      → Gefährlich, falls „größerer" Typ in „kleineren" gewandelt wird *(s.u.)*.
    - **Vorschriften**
      - Integer → Gleitkommazahl
        → Nachkommastellen zu Null, ggf. Runden,
        → undefiniert, falls außerhalb des Wertebereichs *(→ selten).*
      - Gleitkommazahl → Integer
        → Nachkommastellen werden abgeschnitten,
        → undefiniert, falls außerhalb des Wertebereichs.
      - negative Gleitkommazahl → unsigned Integer
        → nicht definiert
      - kleine Gleitkommazahl → große Gleitkommazahl
        → unproblematisch
      - große Gleitkommazahl → kleine Gleitkommazahl
        → undefiniert, falls außerhalb des Wertebereichs

- **Implizite Typwandlung**
  - **Zuweisungen (auch Rückgabe und Parameter von Funktionen)**
    - Rechter Operand wird in Typ des linken Operanden gewandelt, falls beide Operanden verträglich *(sonst Fehlermeldung).*
    - Arithmetische Operanden sind immer verträglich.
      → Gefährlich, falls „größerer" Typ in „kleineren" gewandelt wird *(s.u.)*.
    - **Vorschriften**
      - Beispiele:

```
double x = 4.2;
int zahl = x;          → zahl = __________

int i = -1;              /* 1111 1111 ... 1111 1111 */
unsigned int u = 2;      /* 0000 0000 ... 0000 0010 */
printf ("\n%u", u * i);      → Ergebnis 1111 ... 1111 1110
                               wird unsigned interpretiert als
                               4.294.967.294

i = u * i;
printf ("\n%d\n", i);        → 4.294.967.294 wird bei
                               Zuweisung als signed interpretiert:
                               -2.
```


Programmieren in C
für Elektrotechniker

Kapitel 7: Ausdrücke und Operatoren (Vertiefung)

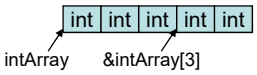
- Ausdrücke
- Operatoren
- Bitmanipulationen
- Typumwandlung
- **Pointer und Arrays (Teil 2)**
- Unions und Bitfelder

- **Beispiel: Pointer auf Array-Elemente**

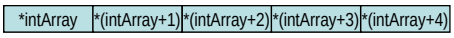
```
int intArray [5];
int * ptr;
…
ptr = &intArray[i-1];
```

*→ **ptr** zeigt auf das i-te Array-Element (**i=0**: erstes Array-Element)*

- **Der Name eines Arrays kann als konstanter Zeiger auf das erste Array-Element verwendet werden.**
  - **intArray[0]** ≡ ***intArray**
  - **intArray** ≡ **&intArray[0]**



  - **intArray[i]** ≡ ***(intArray + i)**
    (Erhöhung des Pointers um 1 → Pointer zeigt ein Element weiter)



- **Pointer-Notation und Array-Notation sind gleichwertig (in beide Richtungen)**
  - Bemerkung:
    Compiler arbeitet intern nur mit Pointern, nicht mit Array-Indizes.
  - Ein Array *(Array-Name)* wird in einen konstanten Pointer auf seinen Komponententyp gewandelt,
    außer es ist ein Operand von **sizeof()** und **&**.
    *→ nur **sizeof()** arbeitet auf Arrays als Ganzes.*
  - Array-Name und Adressoperator auf Array-Name liefern beide den selben konstanten Pointer auf das 1. Element.

```
int intArray [5];
int * ptr;
…
ptr = intArray;
ptr = &intArray;
```

identisch

  - **intArray++** nicht möglich, da konst. Pointer *(nicht modifizierbarer L-Wert)*
    → Pointer (hier: **ptr**) kann Wert zugewiesen werden,
    Array-Namen (hier: **intArray**) nicht.

- **Vergleich von Arrays**

```
int array1 [5];
int array2 [5];
…
if (array1 == array2) …
```

  - Was wird hier verglichen?
    → Vergleicht lediglich, ob **`array1`** an der gleichen Adresse im Speicher liegt wie **`array2`** *(→ gleiche Pointer)*
  - **Vergleich der Array-Inhalte**
    - Eigene Funktion, die über die Array-Elemente iteriert *(s.u.)*.
    - Bibliotheksfunktion `memcmp (array1, array2, sizeof(array1))`
      → byteweiser Vergleich von Speicherinhalten
      → Problem, falls **`array2`** länger als **`array1`** ist, aber in ersten Elementen übereinstimmt

Pointer-Arithmetik

- **NULL-Pointer**
  - **`NULL`** ≡ **`(void *) 0`**
    → typfreier Pointer mit Wert **`0`**
  - In **`<stddef.h>`**.

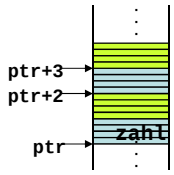
- **Mögliche Operationen auf Pointer**
  - **Zuweisung**
    - Nur Pointer gleichen Typs oder vom Typ **`void *`** (links und rechts der Zuweisung).
      → bei **`void *`** hebt der Compiler die Typprüfung auf.

```
int * p1;
double * p2;
void * p;
p1 = p2;      → möglich?

p = p2;
p1 = p;       → möglich?
```


- **Mögliche Operationen auf Pointer**
  - **Addition und Subtraktion**
    - Zu einem Pointer kann eine ganze Zahl n addiert/subtrahiert werden.
      → Wert des Pointers verändert sich um n-mal die Elementgröße.

```
int zahl;
int * ptr;
ptr = &zahl;
```



    - Verweist ein Pointer auf eine Variable eines anderen Typs
      (→ kann z.B. durch Kopieren über void-Pointer geschehen),
      so wird der Pointer gemäß seiner Definition interpretiert und nicht gemäß des Typs der Variablen, auf die der Pointer zeigt.

```
int * p1;
double * p2;
void * p;
p1 = (int *) p2;
p1++;         → wie verändert sich p1?
```

- **Mögliche Operationen auf Pointer**
  - **Addition und Subtraktion**
    - Das Ergebnis einer Pointer-Addition (z.B. `p + 5;`) kann über das Array hinaus zeigen.
      Man darf/sollte dann nicht auf den Speicher zugreifen,
      da das Resultat undefiniert ist.

```
int main (void)
{
   int intArray [5] = {1, 2, 3, 4, 5};
   int * ptr;
   int i = 0;
   ptr = intArray;
   while (i < 5)
   {
      printf ("\n%d", *ptr);
      ptr++;
      i++;
   }
   return 0;
}
```

ptr zeigt nach Schleifenende hinter das Array ***intArray****.*

- **Mögliche Operationen auf Pointer**
  - **Addition und Subtraktion**

Sei `p1 == (arr + i)`
`p2 == (arr + j)`
und `i > j`

- Was ist das Ergebnis von `p1 - p2`?
  → Anzahl von Elementen, d.h. `i - j`
  (nicht Differenz in Byte)

- **Mögliche Operationen auf Pointer**
  - **Vergleiche**

    ==, != bei Pointer gleichen Typs oder void-Pointer möglich
    → *gleiche Adresse*

    <, > teilweise möglich,
    z.B. bei Verweis auf Elemente des selben Arrays oder Struktur
    → *sonst macht der Vergleich wenig Sinn*

- **Beispiel: Ausgabe des LSB und MSB einer 16-Bit-Integer-Zahl**

```
int main (void)
{
   short int zahl, * ptr1;
   void * dummy;
   unsigned char * ptr2;

   zahl = 0x7FED;
   printf ("Zahl ist: %x \n", zahl);
   ptr1 = &zahl;
   printf ("\nptr1 zeigt auf Adresse: %p\n", ptr1);
   printf ("Inhalt an Adresse ptr1: %x \n", *ptr1);
   dummy = ptr1; /* Pointertyp-Konvertierung, s.o. */
   ptr2 = dummy;
   /* Ausgabe des LSB */
   printf ("\nptr2 zeigt auf Adresse: %p \n", ptr2);
   printf ("Inhalt an Adresse ptr2: %x \n", *ptr2);
   ptr2 = ptr2 + 1;
   printf ("\n\nptr2 wurde um 1 erhoeht\n");
   /* Ausgabe des MSB */
   printf ("\nptr2 zeigt jetzt auf Adresse: %p \n", ptr2);
   printf ("Inhalt an neuer Adresse ptr2: %x \n\n", *ptr2);
   return 0;
}
```

Arrays (Vertiefung)

- **Initialisierung**
  - **Automatische Initialisierung**
    - Globale Arrays: Alle Elemente werden mit 0 initialisiert.
    - Lokale Arrays: Werden nicht automatisch initialisiert.
  - **Manuelle Initialisierung** *(→ nur globale Arrays)*
    - Initialisierungsliste: Liste mit Werten nach dem Zuweisungsoperator.
      - `int arr [3] = {1, 2*3, 4};`
        → Konstanten oder konstante Ausdrücke.
      - `int arr [200] = {1, 2*3, 4};`
        → Restliche Array-Elemente (3 .. 199) werden mit 0 initialisiert.
        → Arrays müssen von Anfang an mit Werten belegt werden.
      - `int arr [2] = {1, 2, 3, 4};`
        → Compiler-Fehler
    - Implizite Längenbestimmung durch den Compiler
      `int arr [] = {1, 2, 3, 4};`
      → Array erhält implizit seine Länge (hier: 4).
      → Meist bei Zeichenketten.

- **Mehrdimensionale Arrays**
  - Anhängen zusätzlicher Klammern.
    `int arr [3][4];`
  - Zuweisung
    `arr[1][2] = 6;`
  - Initialisierung

```
int arr [3][4] =
          {
             {1, 2, 3, 4},
             {5, 6, 7, 8},
             {9, 10, 11, 12},
          };
```

→ identisch zu

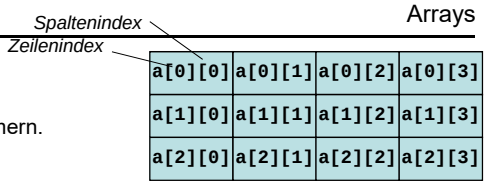
```
int arr [3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

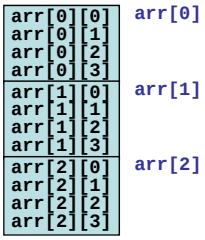
→ Bei der „flachen“ Initialisierung wird von vorne nach hinten aufgefüllt.

  - Unvollständige Initialisierung

```
int arr [3][4] =
          {
             {1},
             {2,3},
          };
```

1	0	0	0
2	3	0	0
0	0	0	0





Zeichenketten (Strings)

- Zeichenketten: Arrays mit Zeichen; mit ‚\0' abgeschlossen *(s.o.)*
  → Platz für Terminierung nicht vergessen!
- Rückgabewert einer Zeichenkette ist ein Pointer auf das erste Zeichen.
  Der Typ ist `char *`.
- Konstante Zeichenketten
  `"hallo"`
- Variable Zeichenketten

```
char str[20];
char str[20] = "hallo";
char str[20] = {'h', 'a', 'l', 'l', 'o', '\0'};
char str[20] = {"hallo"};
```
→ *erlaubt, aber unüblich*

- Einem (globalen) char-Array kann nur bei der Initialisierung eine Zeichenkette zugewiesen werden.

```
...
str = "oha";
...
```
→ *nicht möglich*

→ *Zuweisung nur über spezielle Bibliotheksfunktionen*

- **Parameterübergabe**
  - **Arrays**
    - Aktueller Parameter: Array-Name
      → Pointer auf das erste Element
    - Formaler Parameter: - Offenes Array oder
      - Pointer auf Komponententyp des Arrays
    - Beispiel

```
#define GROESSE 3

void init (int * alpha, int dim) {
    for (int i = 0; i < dim; i++)
        scanf ("%d", alpha++);
}

void ausgabe (int alpha[], int dim) {
for (int i = 0; i < dim; i++)
    printf ("\narr[%d] hat den Wert: %d", i, alpha[i]);
}

int main (void) {
    int arr [GROESSE];
    init (arr, GROESSE);
    ausgabe (arr, GROESSE);
    return 0;
}
```

- **Parameterübergabe**
  - **Zeichenketten**
    → analog
  - Da Zeichenketten mit `'\0'` abgeschlossen werden, kann bei den meisten Funktionen auf den Längenparameter verzichtet werden.
    `int bsp (char *);` oder `int bsp (char[]);`
    → falls doch, berechnet `strlen();` *die Länge einer Zeichenkette.*
  - Ausgabe von Zeichenketten
    - `printf ("hallo");`
      → Übergabe eines Pointers auf eine konstante Zeichenkette.
    - `char str [] = "hallo";` entspr. `char * ptr = "hallo";`
      `printf ("%s\n", str);` `printf ("%s\n", ptr);`

- **char-Arrays und Pointer auf Zeichenketten**
  - `char str [] = "hallo";`
    - Definition eines char-Arrays mit Länge 6 und Initialisierung mit `"hallo"`
      → Array-Komponenten sind L-Werte.
      - `str[1] = 'e'` möglich, aber `str` kann nicht auf andere Adressen zeigen *(→ konstanter Pointer)*
  - `char * ptr = "hallo";`
    - Definition einer Konstanten (→ R-Wert), `ptr` ist Zeiger auf diese Zeichenkette
      (genauer: Zeiger auf das erste Zeichen)
      - `prt` kann verändert werden, z.B. `ptr = "haha";` (genauer: `strcpy(ptr, „haha“);`)
      - `*ptr` nicht änderbar *(→ konstante Zeichenkette)*
        Genauer: Änderung von `*ptr` ist in C nicht vorgesehen, aber der Compiler kann es erlauben
        → mit `const` definitiv unveränderbar

- **Schlüsselwort „const“**
  - `char * str = "hallo";`
    → I.d.R. Konstante im Programmbereich *(s.o.)*.
  - `const char str [] = "hallo";`
    → **`str`** besitzt Name, Typ, Adresse im adressierbaren Speicherbereich
    → Wert aber unveränderbar.
  - `#define STR hallo`
    - → **`STR`** hat keinen expliziten Typ
      → **`#define`** funktioniert nicht bei zusammengesetzten Datentypen
  - `const int feld [] = {1, 2, 3};`
    - → **`feld[0]`**, …, **`feld[2]`** sind Konstanten
    - Gilt auch bei Pointer-Schreibweise:
      `const int * ptr = feld;` → Array-Elemente auch nicht änderbar.

- **Beispiele**
  - Sei `char str1 [] = "Ich liebe Rad fahren.";`
    - `char * str2 = str1;`
      - `str2[4] = 'L';` ☐ möglich ☐ nicht möglich?
      - `str2 = "Du ...";` ☐ möglich ☐ nicht möglich?
    - `const char * str2 = str1;`
      - `str2[4] = 'L';` ☐ möglich ☐ nicht möglich?
      - `str2 = "Du ...";` ☐ möglich ☐ nicht möglich?
    - `char * const str2 = str1;`
      - `str2[4] = 'L';` ☐ möglich ☐ nicht möglich?
      - `str2 = "Du ...";` ☐ möglich ☐ nicht möglich?
    - `const char * const str2 = str1;`
      - `str2[4] = 'L';` ☐ möglich ☐ nicht möglich?
      - `str2 = "Du ...";` ☐ möglich ☐ nicht möglich?

- **Schlüsselwort „const“**
  - Der Schutz durch **`const`** gilt auch bei Übergabeparameter.
    → Die Parameter sind dann in der Funktion unveränderbar.
    - Beispiel:

```
void func (const int * p) {
   ...
   p[3] = 5;    Fehler!
   ...
}
```

  - Häufig Schutz von Zeichenketten (Texten), die nicht verändert werden sollen.
    - Beispiel:

```
int printf (const char * formatstring, ...);
...
printf ("Test \n");
```

→ **`printf`** kann nur lesend auf den Formatierungstext zugreifen.

  - Auch Rückgabeparameter können durch **`const`** geschützt werden.

```
const char * func (. . .) . . .
```


- **Manuelles Kopieren von Zeichenketten**
  - **char-Arrays**

```
char alpha [30] = "zu kopierender String";
char beta [30] = "";

int i = 0;
while (alpha[i] != '\0')
{
    beta[i] = alpha[i];
    i++;
}
beta[i] = '\0';
```

Könnte man verkürzen zu:

```
int i = 0;
while ((beta[i] = alpha[i]) != '\0')
{
    i++;
}
```

Noch kürzer:

```
int i = 0;
while (beta[i] = alpha[i])
{
    i++;
}
```

- **Manuelles Kopieren von Zeichenketten**
  - **Verwendung von Pointern**

```
while (*ptralpha != '\0')
{
    *ptrbeta = *ptralpha; /* ein Zeichen kopieren */
    ptralpha++;
    ptrbeta++;
}
*ptrbeta = '\0';
```

→keine Laufvariable notwendig
→i.d.R. weniger intuitiv/verständlich

Könnte man verkürzen zu:

```
while (*ptralpha != '\0')
{
    *ptrbeta++ = *ptralpha++;
}
*ptrbeta = '\0';
```

Noch kürzer:

```
while (*ptrbeta++ = *ptralpha++);
```


- **Standardfunktionen zur Verarbeitung von Zeichenketten**

```
#include <string.h>
```

- **strcpy()**

```
char * strcpy (char * dest, const char * src);
```

  - Kopiert Zeichenkette inkl. Ende `'\0'` von **`src`** nach **`dest`**.
  - Prüft nicht, ob **`dest`** genügend Platz zur Verfügung stellt.
  - Verhalten undefiniert, falls sich Bereiche **`src`** und **`dest`** überlappen.

- **strcat()**

```
char * strcat (char * dest, const char * src);
```

  - Hängt Zeichenkette **`src`** an **`dest`** an.
  - Prüft nicht, ob **`dest`** genügend Platz zur Verfügung stellt.
  - Verhalten undefiniert, falls sich Bereiche **`src`** und **`dest`** überlappen.

- **strcmp()**

```
int strcmp (const char * s1, const char * s2);
```

  - Zeichenweiser Vergleich der Zeichenketten **`s1`** und **`s2`**.
  - Stoppt bei ersten verschiedenen Zeichen oder `'\0'`.

- **strlen()**

```
size_t strlen (const char * s);
```

  - Bestimmt die Anzahl von Zeichen in Zeichenkette **`s`** (ohne `'\0'`).

- **Standardfunktionen zur Speicherbearbeitung**

```
#include <string.h>
```

- **memcpy()**

```
void * memcpy (void * dest, const void * src, size_t n);
```

  - Kopiert n Bytes von **`src`** nach **`dest`**.
  - Prüft nicht, ob **`dest`** genügend Platz zur Verfügung stellt.
  - Verhalten undefiniert, falls sich Bereiche **`src`** und **`dest`** überlappen.

- **memmove()**

```
void * memmove (void * dest, const void * src, size_t n);
```

  - Analog zu **`memcpy()`**, aber definiert, falls **`src`** und **`dest`** überlappen.

- **memcmp()**

```
int memcmp (const void * s1, const void * s2, size_t n);
```

  - Byteweiser Vergleich der ersten n Byte von **`s1`** und **`s2`**.
  - Stoppt bei ersten verschiedenen Werten.

- **memchr()**

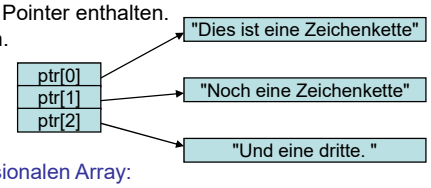
```
void * memchr (const void * s, int c, size_t n);
```

  - Prüft, ob in den ersten n Byte des Puffer **`s`** das Zeichen **`c`** enthalten ist.
  - Ergebnis: Pointer auf das erste Vorkommen von **`c`** oder NULL-Pointer.

Pointer-Vektoren

- Arrays können als Elemente auch Pointer enthalten.
  → i.d.R. Pointer auf Zeichenketten.

  Beispiel: `char * ptr [3];`



- Vorteil gegenüber einem 2-dimensionalen Array:
  - Speicherbereiche, auf die die Pointer zeigen *(hier: 3 Zeichenketten)* können unterschiedlich lang sein.
  - Beispiel:

```
char err_desc1 [][27] = {
   "Fehlercode existiert nicht",
   "Fehlertext 1",
   "Fehlertext 2" };
```

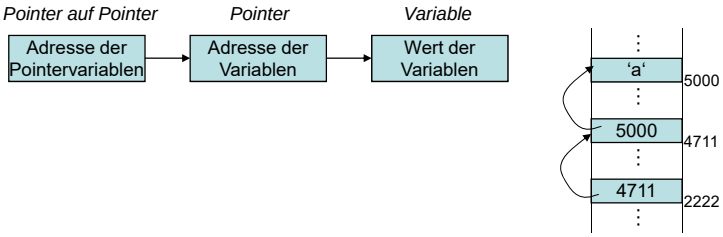
→ *Gesamtgröße:*
*3 * 27 Byte = 81 Byte*

```
char * err_desc2 [] = {
   "Fehlercode existiert nicht",
   "Fehlertext 1",
   "Fehlertext 2" };
```

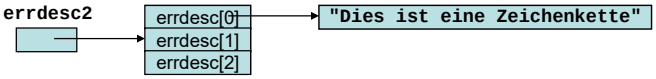
→ *Gesamtgröße:*
27 + 13 + 13 = 53 Byte

Pointer auf Pointer

- Pointer-Vektoren können wiederum auf Pointer verweisen.

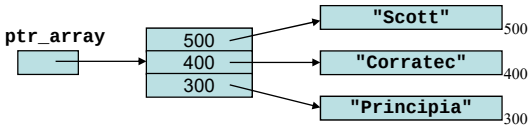


- oben: **`errdesc2`** ist ein Array
  → Pointer auf 1. Array-Element *(nach Übersetzung)*

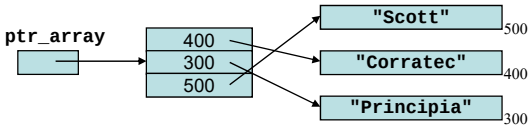


Anwendungsbeispiel 1: Sortieren von Zeichenketten

- z.B. Namenslisten.



→ nach dem Sortieren



- **Anwendungsbeispiel 2: Ausgabe von Texten**

In Parameterlisten immer ein offenes Array
→ Übergabe eines Pointers auf 1. Element
(auch: `void textausgabe (char ** txtptr, int n)`)

```
void textausgabe (char * textPointer[], int anzZeilen)
{
   int i;
   for (i = 0; i < anzZeilen; i++)
      printf ("%s\n", textPointer[i]);
}
```

Zeichenkette und Pointer auf Zeichenkette sind identisch.

- **Anwendungsbeispiel 3: Initialisierung von Pointer-Vektoren**

```
char * fehlertext (int n);

int main (void)
{
   int fehlernummer;
   printf ("Geben Sie eine Fehlernummer von 1 bis 6 ein: ");
   scanf ("%d", &fehlernummer);
   printf ("\nText zu Fehler %d: %s\n",
                  fehlernummer, fehlertext (fehlernummer));
   return 0;
}

char * fehlertext (int n)
{
   /* Initialisierung der Fehlertexte */
   static char * err_desc[] = {
      "Fehlercode existiert nicht",
      "Fehlertext 1", "Fehlertext 2",
      "Fehlertext 3", "Fehlertext 4",
      "Fehlertext 5", "Fehlertext 6" };
   return (n < 1 || n > 6) ? err_desc[0] : err_desc[n];
}
```

Warum „static“?

- **`static`** notwendig, da sonst **`err_desc`** und damit auch *die Rückgabepointer nach Rücksprung undefiniert.*
- ***`err_desc`*** *ist global/permanent; da lokal* definiert aber auch privat, d.h. nur in **`fehlertext`** sichtbar.

Pointer auf Funktionen

- Pointer können auch auf Funktionen verweisen.
  - Funktionen sind Speicherobjekte.
  - Pointer ist die Adresse der Funktion *(Speicherobjekt).*

- Beim Dereferenzieren eines Pointers auf eine Funktion wird diese gestartet.
  → Pointer zeigt auf den ersten Befehl der Funktion.
  - Damit kann zur Laufzeit bestimmt werden, welche Funktion ausgeführt werden soll.

- Vereinbarung eines Pointers auf eine Funktion
  Beispiel: `int (*ptr) (char, int);`
- Zuweisung einer Funktion an einen Pointer
  Beispiel: `ptr = funktionsname;`
- Aufruf einer Funktion über einen Pointer
  Beispiel:
  ```
  int a;
  a = (*ptr)('A', 5);
  ```
  → entspricht: `a = funktionsname('A', 5);`

Funktionsname entspr. Pointer auf Funktion (vgl. Array)
→ auch: `a = ptr('A', 5);`

- **Beispiel: Interrupt-Programmierung**

```
int interrupt_routine_0 (int addr);
int interrupt_routine_1 (int addr);
...

int interrupt_handler (int int_code, int addr)
{
   static int (*interrupt_table[])(int) = {
             interrupt_routine_0,
             interrupt_routine_1,
             ...
             }
   ...
   return (*interrupt_table[int_code])(addr);
}
```

→ statt

```
switch (int_code) {
   case 0: interrupt_routine_0 (addr);
           break;
   case 1: interrupt_routine_1(addr);
           break;
   ...
}
```

- **Funktionspointer als Parameter**
  - Mittels Pointer können Funktionen auch als Parameter an andere Funktionen übergeben werden.
  - Beispiel 1: Laufzeitbestimmung von Funktionen

```
#include <time.h>

void f1 (void) {                    void f2 (void) {
{                                   {
   …                                   …
}                                   }
```

äquivalent:

```
double eval_time (void ptr(void)) {
   time_t begin, end;
   begin = time (NULL);
   (*ptr)();
   end = time (NULL);
   return difftime (end, begin);
}

int main (void) {
   printf ("Zeit fuer f1: %3.0f sec\n\n", eval_time (f1));
   printf ("Zeit fuer f2: %3.0f sec\n\n", eval_time (f2));
   return 0;
}
```


- **Funktionspointer als Parameter**
  - Mittels Pointer können Funktionen auch als Parameter an andere Funktionen übergeben werden.
  - Beispiel 2: Nullstellenbestimmung beliebiger Funktionen mittels Newton-Iteration.

    x(i+1) = x(i) - f(x(i)) / f '(x(i))

  - Beispielsfunktion: f(x) = x - cos (x)

    → f '(x) = 1 + sin(x)

```
double func (double x)
{
    return (x - cos(x));
}

double ableitung (double x)
{
    return (1 + sin(x));
}

double nullstelle (double x, double genauigkeit,
                   double f (double),
                   double fstrich (double))
{
    double z = x; /* Naeherungswert fuer Nullstelle */
    do {
        x = z;
        z = x - f(x) / fstrich(x);
    } while (fabs (z - x) > genauigkeit);
    return z;
}

int main (void) {
    double startwert = 1.;   /* Startwert fuer die Newton-Iteration */
    double epsilon = 0.001; /* Genauigkeit der Nullstellenbestimmung */
    printf ("Die Nullstelle ist bei: %10.2f\n",
                        nullstelle (startwert, epsilon, func, ableitung));
    return 0;
}
```

äquivalent

Programmieren in C
für Elektrotechniker

Kapitel 7: Ausdrücke und Operatoren (Vertiefung)

- Ausdrücke
- Operatoren
- Bitmanipulationen
- Typumwandlung
- Pointer und Arrays (Teil 2)
- **Unions und Bitfelder**

- **Unions (Datentyp union)**
  - Unions bestehen wie Strukturen aus mehreren Komponenten.
    Beispiel:
```
union vario {
    int intval;
    long longval;
    float floatval;
    } variant;

int x = 123;
variant.intval = x;
```
    bzw.
```
union vario * ptr;
ptr = &variant;
ptr->intval = x;
```
  - Komponenten werden als Alternativen ab der selben Speicheradresse gespeichert.
  - Zu einem Zeitpunkt ist aber nur eine Alternative gespeichert.
    → Programmierer ist für konsistentes Speichern und Lesen verantwortlich.
  - Speicherbedarf der Datenstruktur **`union`** ist gleich dem Speicherbedarf der größten Komponente.
  - Unions können in Strukturen und Arrays auftreten – und umgekehrt.

Beispiel

```
int main (void) {
   int i;
   enum punktArt {KARTESISCH, RADIAL};
   struct kartesischerPunkt {
      double x, y; /* Koordinaten */
   };
   struct radialPunkt {
      double radius; /* Radius ab Ursprung */
      double phi; /* Winkel in Grad */
   };
   struct punkt {
      enum punktArt art;
      union {
         struct kartesischerPunkt kPunkt;
         struct radialPunkt rPunkt;
      } u;
   };
   struct punkt pts[2];
   pts[0].art = KARTESISCH;
   pts[0].u.kPunkt.x = 1.; pts[0].u.kPunkt.y = 1.;
   pts[1].art = RADIAL;
   pts[1].u.rPunkt.radius = sqrt (2.); pts[1].u.rPunkt.phi = 45.;
   for (i=0; i < sizeof (pts) / sizeof (pts[0]); i++) {
      switch (pts[i].art) {
         case KARTESISCH: …;
         case RADIAL: …;
      }
   }
}
```

Unlogisch, aber möglich:
`pts[0].u.rPunkt.phi = 45.;`

Bitfelder

- Zur Hardware-nahen Programmierung.
- Bisher: bitweise Manipulation von Variablen *(Byte, Wort, Langwort)* durch Bit-Operationen **&**, **|**, **^**, **~**
- Alternative: Bitfelder: Gruppierung von Bits *(mit oder ohne Vorzeichen)*
  → Länge eines Bitfelds durch Doppelpunkt vom Namen getrennt
  → Komponente einer Struktur oder union
- Beispiel:

```
struct bitfeld_t  {
    unsigned a:4;
    signed b:4;
}
```

a [0][0][0][0] Wertebereich: 0 .. 15
a = 3 [0][0][1][1]
a = 19 [0][0][1][1] → Überlauf

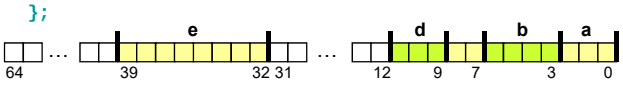
b [0][0][0][0] Wertebereich: -8 .. 7
b = 3 [0][0][1][1]
b = 9 [1][0][0][1] → Überlauf

- Speicherbelegung ist compiler-abhängig
  (→ vom C-Standard explizit gewollt – eingeschränkte Portabilität)
  → meist nahtlos hintereinander *(s.u.)*

Bitfelder

- Nicht benannte Bitfelder für
  - reservierte Bits
  - Alignment auf Wortlängen: Bitfelder der Länge 0.
- Beispiel:

```
struct Bitfeld_Struktur_3 {
    unsigned a:3;
    unsigned b:4;
    unsigned :2;
    unsigned d:3;
    unsigned :0;
    unsigned e:8;
};
```



- **Bitfelder**
  - Nicht benannte Bitfelder für
    - reservierte Bits
    - Alignment auf Wortlängen: Bitfelder der Länge 0.
  - Beispiel: MIPS-Statusregister *(vgl. Vorlesung Architektur digitaler Systeme)*

```
struct status {
   unsigned current:2;
   unsigned previous:2;
   unsigned old:2;
   unsigned :2;
   unsigned int_mask:8;
};
```