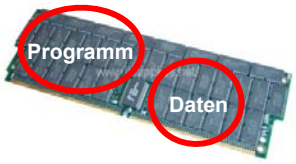
Programmieren in C
für Elektrotechniker

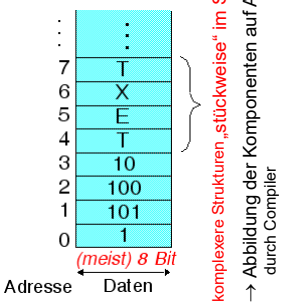
Kapitel 3: Variablen, Datentypen und Operatoren (Teil 1)

- **Variablen**
- Einfache Datentypen
- Operatoren
- Abgeleitete Datentypen

Variablen

- C gehört zu den imperativen Programmiersprachen, d.h.
  Manipulation von Variablen durch Sequenz von Befehlen
  z.B. „`falls x größer als y, subtrahiere b von a`“
- Variable: benannte Speicherzelle
  → veränderliche Größe eines bestimmten Wertebereichs (Datentyps)
  ↔ Konstante
- Wert einer Variablen muss explizit zugeordnet werden
  → da Bitmuster im Speicher immer irgend einen Wert repräsentiert, sollten Variablen initialisiert werden *(Zuweisung eines Standardwerts)*





- Variablen liegen im Arbeitsspeicher
  - Speicherzelle: 1 Byte (8 Bit)
  - Variablen: 1 … n Speicherzellen
- 4 Kennzeichen für Variablen:
  - Variablen-Name
  - Datentyp (Wertebereich)
  - Aktueller Wert
  - Adresse

- **In C wird auf Variablen über deren Namen oder Adresse zugegriffen.**
  → *Neuere Sprachen wie Java: kein Zugriff über Adressen durch Programmierer.*
- Achtung: Häufiger Programmierfehler: falsche Adressen.

(Daten-) Typen von Variablen

- Datentyp einer Variablen legt fest:
  - Wertebereich der Variablen
  - Darstellung der Variablen im Speicher (auch Größe in Bits)
  - mögliche Operationen auf den Variablen

- Standard-Datentypen einer Programmiersprache (z.B. int, float)
  → direkt verwendbar
  → meist „atomar", d.h. nicht aus anderen Datentypen zusammengesetzt

- Selbst definierte Datentypen
  → Aufbau und Struktur aus Standard-Datentypen vom Programmierer festgelegt

Initialisierung von Variablen

- Variablen sollten vor ihrer ersten Verwendung initialisiert werden.
  → Ausnahmen (später mehr): - automatisch initialisierte Variablen
  - erster Zugriff: Zuweisung.
- Beispiel:

```
int main (void)
{
   int a, b;
   ...
   /* Initialisierung (lokaler) Variablen */
   a = 0;
   b = 100;
   ....
   while (a < b) ...
}
```

Für „Experten“:
Was passiert bei folgendem Programm?

```
   a == 100;
   b = 100;
   while (a = b)
        a = a + 1;
```


Initialisierung von Variablen

- Einfache Variablen können direkt bei ihrer Definition initialisiert werden.
  → Konstanten des passenden Typs zuweisen.
- Beispiel:

```
int main (void)
{
   /* Initialisierung lokaler Variablen */
   int a = 9;
   int b = 1, c = 2;
   int d, e = 3;
   char c1 = 'c', c2 = 'd';
   ....
}
```

Programmieren in C
für Elektrotechniker

Kapitel 3: Variablen, Datentypen und Operatoren (Teil 1)

- Variablen
- **Einfache Datentypen**
- Operatoren
- Abgeleitete Datentypen

Datentypen

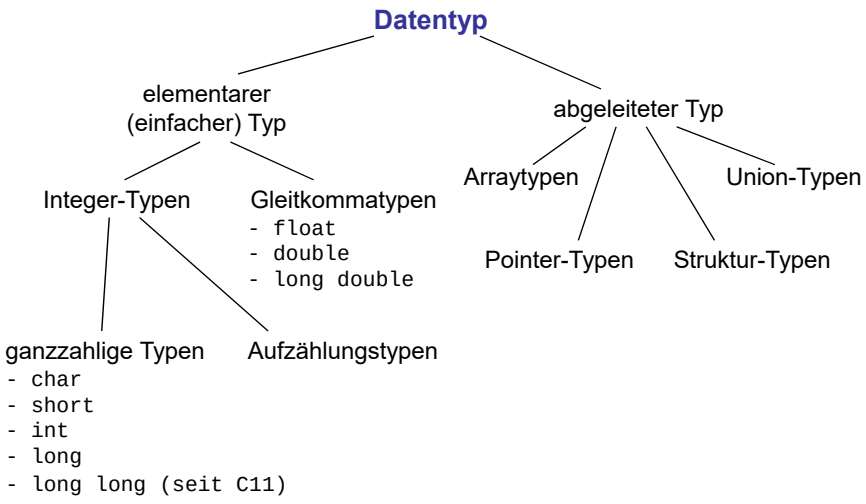
- Typkonzept in C
  - C verlangt, dass alle Variablen einen vom Programmierer festgelegten Datentyp haben *(↔ Scriptsprachen wie z.B. php, Javascript).*
  - Programmiersprachen mit strengem Typkonzept verbieten die Zuweisung von Variablen unterschiedlichen Typs.
  - C besitzt ein eingeschränktes Typkonzept, d.h. einige Typkonvertierungen erfolgen „automatisch“.

```
...
float x = 3.9;
int y;
y = x;                /* o.k.? */
printf ("%d", y);     /* Ausgabe = _____*/
...
```

→ Zuweisung eines **`float`**-Werts an eine **`int`**-Variable: Fehlermeldung bei strengem Typkonzept.

  - C: **`float`**-Wert wird nach **`int`** gewandelt, indem Dezimalstellen abgeschnitten werden
    → *mehr im folgenden Kapitel*

- Typkonzept in C
  - Datentyp legt fest:
    - Speicherbedarf
    - Kodierung im Speicher
    - Wertebereich
    - Genauigkeit (bei Gleitkommazahlen)
  - Alle elementaren, d.h. einfachen Datentypen sind arithmetische Typen:
    - Ganzzahltypen (Integer-Typen)
    - Gleitkommatypen



Programmieren in C

für Elektrotechniker

Kapitel 3: Variablen, Datentypen und Operatoren (Teil 1)

- Variablen
- **Einfache Datentypen**
  - **Ganze Zahlen (Integer)**
  - Gleitkommazahlen
  - Zeichen
  - Aufzählung
  - Konstanten und lexikalische Konventionen
- Operatoren
- Abgeleitete Datentypen

Zahlen im Rechner

a) Positive ganze Zahlen

Wir rechnen im Dezimalsystem: jede Ziffer *d* an der Stelle *i* hat den Wert $d \cdot 10^i$
bzw. $d \cdot Basis^i$, mit Basis = 10
Beispiel: $365 = 3 \cdot 10^2 + 6 \cdot 10^1 + 5 \cdot 10^0$

Aber: Für Computer ist die Basis 2 besser geeignet.

allgemein: **positive ganze Zahl zur Basis b**:
Folge von *n* Ziffern $X_{n-1}, ..., X_0$ mit Werten zwischen *0* und *b-1*

Für den Wert der Zahl gilt: $(X_{n-1}, ..., X_0)_b = \sum_{i=0}^{n-1} b^i \cdot X_i$.

Dualzahlen

Jede natürliche Zahl $k \in \mathbb{N}$ mit $0 \leq k \leq 2^n-1$
lässt sich als Dualzahl $(X_{n-1}, ..., X_0)$, $X_i \in \mathbb{B} = \{0, 1\}$ darstellen.

$$X_{(n)} = (X_{n-1}, ..., X_0)_2 = \sum_{i=0}^{n-1} 2^i \cdot X_i \quad , X_i \in \mathbb{B}$$

Schreibweise für Wert einer Dualzahl mit n Bit

- X_0: LSB (least significant bit)
- X_{n-1}: MSB (most significant bit)

Dezimalzahl → Dualzahl:

```
i ← 0; x0 ← 0;
WHILE k ≠ 0 DO
    xi ← k MODULO 2
    k ← k DIV 2
    i ← i + 1
```

$$X_{(n)} = \sum_{i=0}^{n-1} 2^i \cdot X_i \quad , X_i \in \mathbb{B}$$
$$= (\ldots(x_{n-1} \cdot 2 + x_{n-2}) \cdot 2 + \ldots + x_1) \cdot 2 + x_0$$

Dezimalzahl → Dualzahl:

$i \leftarrow 0; x_0 \leftarrow 0;$
WHILE $k \neq 0$ DO
  $x_i \leftarrow k$ MODULO 2
  $k \leftarrow k$ DIV 2
  $i \leftarrow i + 1$

Beispiel:

Dezimalzahl → Dualzahl:

$k = 317_{10} = ?_2$

k	k MODULO 2	k DIV 2	i
317	$x_0 = 1$	158	0
158	$x_1 = 0$	79	1
79	$x_2 = 1$	39	2
39	$x_3 = 1$	19	3
19	$x_4 = 1$	9	4
9	$x_5 = 1$	4	5
4	$x_6 = 0$	2	6
2	$x_7 = 0$	1	7
1	$x_8 = 1$	0	8

$\Rightarrow 317_{10} = 100111101$

Oktal- und Hexadezimalzahlen

Zusammenfassung von Bitgruppen zur besseren Übersicht

Oktaden: 3 Bit, Ziffern: 0, 1, ..., 6, 7

Tetraden: 4 Bit, Ziffern: 0, 1, ..., 8, 9, A, B, C, D, E, F

⇒ 1 Byte = 8 Bit
= 2 Hexadezimalziffern

Wert	binär	oktal	hexadezimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	-	8
9	1001	-	9
10	1010	-	A
11	1011	-	B
12	1100	-	C
13	1101	-	D
14	1110	-	E
15	1111	-	F

Beispiel:

$k = 53 = 110|101_2 = 65_8$

b) Ganze Zahlen

Bezeichnungen:

X:		Ziffer
x:		Wert einer Zahl
$X_{(n)}$, $\overrightarrow{X_{(n)}}$:		Bitkette den Länge n: $(X_{n-1}, ..., X_0)$, $X_i \in IB = \{0, 1\}$
$X_{(n)}$:		auch Wert einer Dualzahl mit n Bit, d.h. Wert von $(X_{n-1}, ..., X_0)$

allg.: $(X_{n-1}, ..., X_0)_b \Rightarrow b^n$ verschiedene Zahlen

Natürliche Zahlen *(s.o.)*

Wert einer Zahl: $(X_{n-1}, ..., X_0)_b = \sum_{i=0}^{n-1} b^i \cdot X_i$.

Dezimalsystem (n = 3)
10^3 = 1000 versch. Zahlen

$\overrightarrow{X_{(n)}}$	Wert
000	0
001	1
002	2
003	3
: :	: :
998	998
999	999

Binärsystem (n = 8) - Dualzahlen -
2^8 = 256 versch. Zahlen

$\overrightarrow{X_{(n)}}$	Wert
00000000	0
00000001	1
00000010	2
00000011	3
: :	: :
11111110	254
11111111	255

Wie könnte man in $\overrightarrow{X_{(n)}} = (X_{n-1}, X_{n-2}, ..., X_1, X_0)$ positive und negative Zahlen kodieren?

Positive und negative ganze Zahlen

⇒ Zahlenbereich halbieren

$b^n/2$ positive und $b^n/2$ negative Zahlen

Vorzeichenzahlen

Kodierung: $X_{(n)} = X_{n-1}(X_{n-2}, ..., X_0)_b = X_V(X_{n-2}, ..., X_0)_b$

Dezimalsystem (n = 3)
$\pm(X_1, X_0)_{10}$ mit 0 entspr. '+', 1 entspr. '-'

$\overrightarrow{X_{(n)}}$	Wert
000	0
001	1
002	2
:	:
098	98
099	99
(100	-0)
101	-1
102	-2
:	:
198	-98
199	-99

Beispiel:
7 + (-5) = ?

7: **0**07
-5: **1**05

112
entspr. -12

Binärystem (n = 8)
$\pm(X_6, X_5, ..., X_0)_2$ mit 0 entspr. '+', 1 entspr. '-'

$\overrightarrow{X_{(n)}}$		Wert
00000000	= 0	0
00000001	= 1	1
00000010	=2	2
:		:
01111110	= 126	126
01111111	= 127	127
(10000000	= 128	-0)
10000001	= 129	-1
10000010	= 130	-2
:		:
11111110	= 254	-126
11111111	= 255	-127

Beispiel:
7 + (-5) = ?

7: **0**0000111
-5: **1**0000101

10001100
entspr. -12

geringerer Zahlenbereich, da X_2 nur 0 und 1

Ist diese Zahlenkodierung für Arithmetik geeignet?
(Annahme: Recheneinheit, die 2 Dualzahlen korrekt addiert/subtrahiert/multipliziert/dividiert)

Addition: 3 + -5 (000011 + 100101)

→ geht nicht mit Addierer
→ *abhängig vom Vorzeichen: Addierer oder Subtrahierer verwenden*

weiteres Problem: Operanden müssen ggf. vertauscht werden
(-5) + 3 → 3 - 5

wünschenswert: ganze Zahlen so kodieren, dass auch bei negativen Zahlen nur Addierer benötigt wird
ist möglich: 2K-Zahlen (s.u.)

Subtraktion: → analog

2K-Zahlen (2-Komplement)

negative Zahlen als Komplement gegen 2^n gebildet → *am häufigsten verwendete Kodierung*

Beispiel: 10K- und 2K-Zahlen *negative Zahlen: Komplement gegen kleinste nicht mehr darstellbare Zahl*

Dezimalsystem (n = 3)
$0 \le x \le 10^3/2 - 1$: $X_{(3)} = |x|$
$-10^3/2 \le x < 0$: $X_{(3)} = 10^3 - |x|$

$\overrightarrow{X_{(n)}}$	Wert
000	0
001	1
002	2
:	:
498	498
499	499
500	-500
501	-499
:	:
998	-2
999	-1

Beispiel:
7 + (-5) = ?

7: 007
-5: 995

1|002

Binärsystem (n = 8)
$0 \le x \le 2^7 - 1$: $X_{(8)} = |x|$, $X_v = 0$
$-2^7 \le x < 0$: $X_{(8)} = 2^8 - |x|$, $X_v = 1$

$\overrightarrow{X_{(n)}}$		Wert
00000000	= 0	0
00000001	= 1	1
00000010	=2	2
:		:
01111110	= 126	126
01111111	= 127	127
10000000	= 128	-128
10000001	= 129	-127
:		:
11111110	= 254	-2
11111111	= 255	-1

Beispiel:
7 + (-5) = ?

7: **0**0000111
-5: **1**1111011

1|**0**0000010

Umrechnung Dezimalzahlen ⇔ 2K-Zahlen

a) Dezimalzahl → 2K-Zahl (n Bit)

- Positive Zahl x: wie bei Dualzahlen (nur, wenn $x < 2^{n-1}$)
- Negative Zahl x: Umformung von $2^n - x$ in eine Dualzahl

b) 2K-Zahl (n Bit) → Dezimalzahl

- Analog zu Dualzahlen: stellenrichtiges Aufsummieren mit MSB = -2^{n-1}

  $x = -2^{n-1} + x_{n-2}*2^{n-2} + \ldots + x_1*2^1 + x_0*2^0$

- Beispiel:

1	0	1	0	0	1	1	1

$x = -1*2^7 + 0*2^6 + 1*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 1*2^0$

$= -128 + 32 + 4 + 2 + 1 = = -89$

- Über folgende Komplementbildung (alternatives Verfahren)

Berechnung des Komplements $X'_{(n)}$:

(Bezeichnung: $\overline{X_{(n)}}$: Wert der Bitkette, bei der alle Stellen invertiert ($1 \rightarrow 0$; $0 \rightarrow 1$))

Es gilt: $X'_{(n)} = 2^n - X_{(n)}$

Weiter gilt: $X_{(n)} + \overline{X_{(n)}} = \sum_{i=0}^{n-1} X_i \cdot 2^i + \sum_{i=0}^{n-1} \overline{X_i} \cdot 2^i$

$= \sum_{i=0}^{n-1} (X_i + \overline{X_i}) \cdot 2^i = \sum_{i=0}^{n-1} 1 \cdot 2^i = \sum_{i=0}^{n-1} 2^i = 2^n - 1$

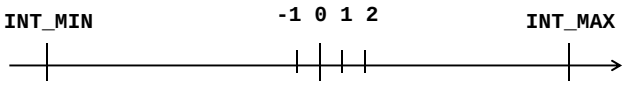
$\Rightarrow \overline{X_{(n)}} + 1 = 2^n - X_{(n)} = X'_{(n)}$

⇒ Komplementbildung: alle Bits negieren und 1 aufaddieren

Bemerkungen:
- Komplement von 100...0 = -2^{n-1} nicht möglich
- es gilt immer: a - b = a + (-b) ohne Übertragsbetrachtung
- positive Zahlen haben immer führende Nullen, negative Zahlen führende Einsen
- Replikation des Vorzeichens (nach links) verändert Zahlenwert nicht (⇒ leicht von 32 Bit auf 64 Bit änderbar)

Datentyp `int`

- ganze Zahlen (Integer) in C
- endlicher Wertebereich



int: ganze Zahlen im Bereich INT_MIN bis INT_MAX

- **`INT_MIN`**, **`INT_MAX`**: rechnerabhängig *(→ in **`limits.h`** festgelegt)*
  - 32 Bit: -2^{31} … $2^{31}-1$ *(meistens)*
  - 16 Bit: -2^{15} … $2^{15}-1$
- **Zahlenüberlauf**:
  Ergebnis einer Operation kleiner als **`INT_MIN`** oder größer als **`INT_MAX`**
  → wird in C nicht geprüft

Integer-Typen in C

→ 2K-Zahlen (unsigned: Dualzahlen)

- **(signed | unsigned) char**
  - immer 8 Bit *(durch C festgelegt)*
  - Kodierung von ASCII-Zeichen (s.u.) bzw. 8 Bit ganze Zahl
  - signed: -128 … +127
  - unsigned: 0 … 255

Integer-Typen in C

→ 2K-Zahlen (unsigned: Dualzahlen)

- **(signed) int | unsigned int**
  - mindestens 16 Bit, typisch 32 Bit
  - 16-Bit-Rechner → 16 Bit
    32-Bit-Rechner → 32 Bit
    64-Bit-Rechner → meist 32 Bit
  - signed: -2.147.483.648 … +2.147.438.647 *(bei 32 Bit)*
  - unsigned: 0 … 4.294.967.295 *(bei 32 Bit)*

Integer-Typen in C

→ 2K-Zahlen (unsigned: Dualzahlen)

- **(signed) short (int) | unsigned short (int)**
  - mindestens 16 Bit, typisch 16 Bit
  - 16-Bit-Rechner → 16 Bit
    32-Bit-Rechner → meist 16 Bit
    64-Bit-Rechner → meist 16 Bit
  - signed: -32.768 … +32.767 *(bei 16 Bit)*
  - unsigned: 0 … 65.535 *(bei 16 Bit)*

Integer-Typen in C

→ 2K-Zahlen (unsigned: Dualzahlen)

- **(signed) long (int) | unsigned long (int)**
  - mindestens 16 Bit, typisch 32 Bit
  - 32-Bit-Rechner → meist 32 Bit
    64-Bit-Rechner → meist 32 Bit
  - signed: -2.147.483.648 … +2.147.438.647 *(bei 32 Bit)*
  - unsigned: 0 … 4.294.967.295 *(bei 32 Bit)*

Integer-Typen in C

→ 2K-Zahlen (unsigned: Dualzahlen)

- **(signed) long long (int) | unsigned long long (int)**
  - mindestens 16 Bit, typisch 64 Bit
  - 32-Bit-Rechner → meist 32 Bit
    64-Bit-Rechner → meist 64 Bit
  - signed: -9.223.372.036.854.755.808 ... + 9.223.372.036.854.755.807 *(bei 64 Bit)*
  - unsigned: 0 ... 18.446.744.073.709.551.615 *(bei 64 Bit)*
- **garantierte 64 Bit**
  - int64_t, uint46_t
    in <inttypes.h>
- **Ausgabe in printf**
  - dezimal: %lld
  - hexadezimal: %llx (z.B. **`a43c55f`**)
    %0#18llx (z.B. **`0x000000000a43c55f`**)

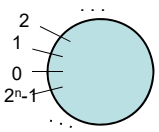
Integer-Typen in C

→ 2K-Zahlen (unsigned: Dualzahlen)

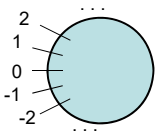
- **Aufzählungstypen** (s.u.)
  - Größe ist maschinenabhängig
- Vorgabe für Wertebereiche: char < short ≤ int ≤ long ≤ long long
  → Größen in limits.h festgelegt.

Überläufe bei Integer-Typen

- **unsigned (char | short | int | long | long long)**
  - kein Überlauf
  - alle (arithm.) Operationen: modulo 2^n



- **signed (char | short | int | long | long long)**
  - Überlauf durch die ALU (Rechenwerk)
  - C-Laufzeitsystem muss auf die Überläufe nicht reagieren
  - C-Programmierer muss die Überläufe selbst abfangen
  - Keine modulo-Arithmetik wegen 2K-Kodierung



Programmieren in C

für Elektrotechniker

Kapitel 3: Variablen, Datentypen und Operatoren (Teil 1)

- Variablen
- **Einfache Datentypen**
  - Ganze Zahlen (Integer)
  - **Gleitkommazahlen**
  - Zeichen
  - Aufzählung
  - Konstanten und lexikalische Konventionen
- Operatoren
- Abgeleitete Datentypen

Durch $\overrightarrow{X_{(n)}}$ lassen sich 2^n verschiedene Zahlen kodieren.
(bisher: ganze Zahlen, d.h. 2^n Zahlen im Abstand von 1)

→ (ganze) Zahlen mit einem fiktiven Komma, das ganz rechts steht

Verschiebung des Kommas um m Stellen nach links bedeutet Division durch 2^m
→ Abstände der Zahlen jetzt 2^{-m} *(→ näherungsweise rationale Zahlen)*

Festkommazahlen: Dualzahlen, bei denen wir uns das Komma an einer festen Stelle vorstellen
(fix point number) → Stelle des Kommas bestimmt Abstand der Zahlen

Gleitkommazahlen: Verwendung zusätzlicher Bits, die die Stelle des Kommas angeben
(floating point number) → Abstand der Zahlen variabel

c) Gleitkommazahlen

Prinzip

$x = mx \cdot 2^{ex}$ mx: Mantisse, ex: Exponent } hintereinander binär kodieren

Mantisse: meist normalisierte Vorzeichenzahl *(⇒ Multiplikation und Division einfach - s.o.)*
normalisiert ⇒ Komma direkt vor bzw. nach erster '1'

vor '1': $\frac{1}{2} \le mx_{norm} \le 1 - \frac{1}{2^r} < 1$, r: Mantissenlänge

nach '1': $1 \le mx_{norm} \le 2 - \frac{1}{2^{r-1}} < 2$

Problem: Null nicht darstellbar
→ meist speziell kodiert (z.B. durch mx = ex = 0)

Exponent: ganzzahlig zwischen ex_{min} und ex_{max}

⇒ Zahlenbereich (bei Komma vor erster '1'): $\frac{1}{2} \cdot 2^{ex_{min}} = 2^{ex_{min}-1} \le x \le \left(1 - \frac{1}{2^r}\right) \cdot 2^{ex_{max}}$

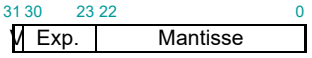
Abstand zweier Zahlen bei festem Exponent ex: $\Delta x = \frac{1}{2^r} \cdot 2^{ex}$ (Rundungsfehler)

IEEE-Gleitkommaformat ⇒ seit 1980 in so gut wie jedem Rechner

→ IEEE 754-Standard

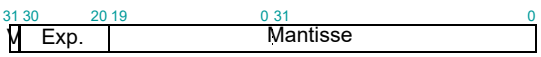
FP-Zahlen: Vielfaches der Wortgrenzen (32 Bit)

float



Zahlenbereich: ca. $\pm 2{,}0 \cdot 10^{-38}$ ... $\pm 2{,}0 \cdot 10^{38}$

double



Zahlenbereich: ca. $\pm 2{,}0 \cdot 10^{-308}$ ... $\pm 2{,}0 \cdot 10^{308}$
aber wichtiger als großer Zahlenbereich: größere Genauigkeit

Addition und Subtraktion von Gleitkommazahlen

Verfahren: Addition/Subtraktion der Mantissen bei gleichem Exponenten
hier: Addition (→ Subtraktion analog)

Beispiel: $9{,}999 \cdot 10^{1} + 1{,}610 \cdot 10^{-1}$
$= 9{,}999 \cdot 10^{1} + 0{,}016 \cdot 10^{1}$ bzw. $999{,}9 \cdot 10^{-1} + 1{,}610 \cdot 10^{-1}$
$= 10{,}015 \cdot 10^{1}$ bzw. $1001{,}5 \cdot 10^{-1}$
$= 1{,}0015 \cdot 10^{2}$

Probleme:
- durch Denormalisieren/Normalisieren können Ziffern verloren gehen
- Überlauf bei Mantissenaddition
- Rundung

Programmieren in C Rationale Zahlen

Addition und Subtraktion von Gleitkommazahlen

Verfahren: Addition/Subtraktion der Mantissen bei gleichem Exponenten
hier: Addition (→ Subtraktion analog)

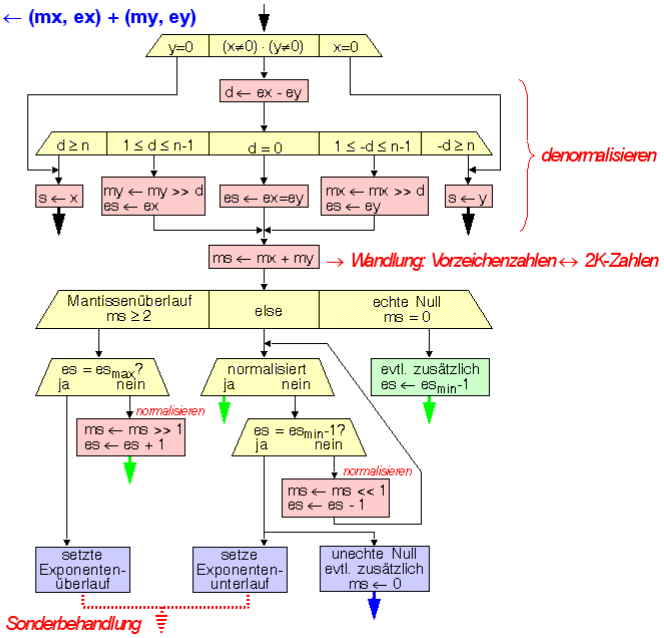
Addition setzt sich aus 4 Schritten zusammen:

$$s = x + y = mx \cdot 2^{ex} + my \cdot 2^{ey}$$

- Denormalisieren → Exponenten anpassen
- Addition der Mantissen
- Summe normalisieren
- Summe runden

Gleitkommaaddition: (ms, es) ← (mx, ex) + (my, ey)
(ohne Runden)

n: Mantissenlänge



Rundung von Gleitkommazahlen

Rundung in IEEE-Norm:

Modus	Bedeutung
round up	Rundung in Richtung ∞
round down	Rundung in Richtung -∞
truncate	Rundung in Richtung 0
to nearest even	Rundung zur nächsten ganzen Zahl bei 0,5 wird zur nächsten geraden Zahl gerundet

Beispiel: Mantisse soll durch Runden um zwei Bit gekürzt werden

Mantisse	truncate	down	up	nearest even	nearest
.....010000	..0100	..0100	..0100	..0100	..0100
.....010001	..0100	..0100	..0101	..0100	..0100
.....010010	..0100	..0100	..0101	..0100	..0101
.....010011	..0100	..0100	..0101	..0101	..0101

Mantisse	truncate	down	up	nearest even	nearest
.....010100	..0101	..0101	..0101	..0101	..0101
.....010101	..0101	..0101	..0110	..0101	..0101
.....010110	..0101	..0101	..0110	..0110	..0110
.....010111	..0101	..0101	..0110	..0110	..0110

Mantisse	truncate	down	up	nearest even	nearest
neg.....010110	..0101	..0110	..0101	..0110	..0110

▪ Gleitkomma-Typen

- Teilmenge der reellen bzw. rationalen Zahlen
- endlicher Wertebereich
  → Brüche und reelle Zahlen i.d.R. nicht exakt darstellbar
  → Genauigkeit der Darstellung begrenzt

→ 1 bis 3 unterschiedliche Formate und Genauigkeiten

- **float**: typ. 32 Bit, IEEE 754-Format
  ISO-Standard: mind. 10^{-37} … 10^{37}, Genauigkeit mind. 10^{-5}
- **double**: typ. 64 Bit, IEEE 754-Format
- **long double**: typ. 80 Bit (bis 128 Bit)

- Größe und Genauigkeit in **`float.h`**.

Programmieren in C
für Elektrotechniker

Kapitel 3: Variablen, Datentypen und Operatoren (Teil 1)

- Variablen
- **Einfache Datentypen**
  - Ganze Zahlen (Integer)
  - Gleitkommazahlen
  - **Zeichen**
  - Aufzählung
  - Konstanten und lexikalische Konventionen
- Operatoren
- Abgeleitete Datentypen

Variablen, Datentypen, Unterprogramme

Zeichen

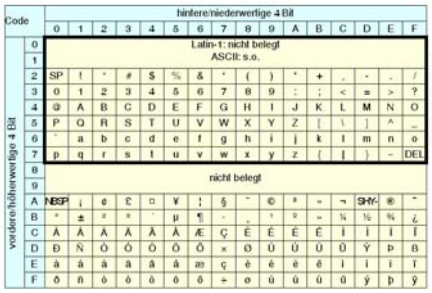
- Grundelement eines Programms *(über Editor eingegeben)*
  → Buchstaben, Ziffern, Sonderzeichen, auch Steuerzeichen *(z.B. ctrl-C)*
- **Alphabet**: geordnete Zeichenmenge
  0,1: Binäralphabet
  0, 1, …, 9: Dezimalalphabet
  0, …, 9, A, .., F: Hexadezimalalphabet
  a, b, …, z: Kleinbuchstaben *(ohne Umlaute und „ß“)*
- **Kodierung**: über Abbildungstabelle
  → nicht eindeutig
  z.B. ASCII, EBCDIC, Unicode

ASCII-Code (American Standard Code for Information Interchange)

Darstellung von Buchstaben, Ziffern, Sonder- und Steuerzeichen
je Zeichen: 7 Bit (→ 128 Zeichen) + 1 Prüfbit

Latin-1-Code

Darstellung von Buchstaben und Ziffern
je Zeichen: 8 Bit (→ 256 Zeichen - erweiterter ASCII-Code)



Name	Char	Dec	Hex	Description	Name	Char	Dec	Hex	Description
NUL	^@	00	00	null	DLE	^P	16	10	data link escape
SOH	^A	01	01	start of heading	DC1	^Q	17	11	device control 1
STX	^B	02	02	start of text	DC2	^R	18	12	device control 2
ETX	^C	03	03	end of text	DC3	^S	19	13	device control 3
EOT	^D	04	04	end of transmission	DC4	^T	20	14	device control 4
ENQ	^E	05	05	enquiry	NAK	^U	21	15	negative acknowledge
ACK	^F	06	06	acknowledge	SYN	^V	22	16	synchronous idle
BEL	^G	07	07	bell	ETB	^W	23	17	end of trans. block
BS	^H	08	08	backspace	CAN	^X	24	18	cancel
HT	^I	09	09	horizontal tab	EM	^Y	25	19	end of medium
LF,NL	^J	10	0A	line feed, new line	SUB	^Z	26	1A	substitute
VT	^K	11	0B	vertical tab	ESC	^[	27	1B	escape
FF,N P	^L	12	0C	form feed, new page	FS	^\	28	1C	file separator
CR	^M	13	0D	carriage return	GS	^]	29	1D	group separator
SO	^N	14	0E	shift out	RS	^^	30	1E	record separator
SI	^O	15	0F	shift in	US	^_	31	1F	unit separator

erste 32 ASCII-Zeichen (Steuerzeichen)

EBCDIC (Extended Binary Code for Digital Interchange)

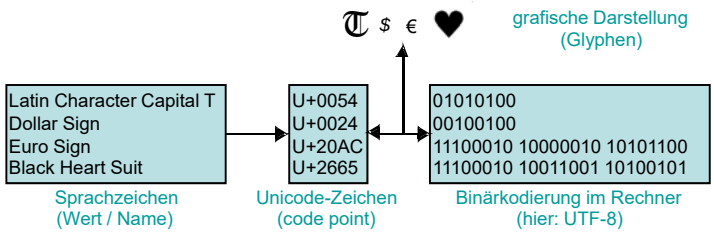
→ lediglich andere Abbildungstabelle als ASCII-Zeichensatz
→ auf IBM-kompatiblen und Siemens-Großrechnern

Unicode-Standard (ISO 10646)

→ Benennung und Kodierung aller Zeichen aller Sprachen

Werte, nicht Darstellungen/Glyphen

→ zweistufige Abbildung



Anforderungen:
- Unicode-Zeichen (code points) für alle Zeichen aller Sprachen
- Sonderzeichen wie math. Symbole, Pfeile, Dingbats, ...
- Erweiterungsmöglichkeit für anwendungsspezifische Symbole
- “Modifier”-Zeichen wie Tilde, Umlaut-Punkte, Akzente etc.
- gleiche Zeichen in mehreren Sprachen nur einmal kodiert
  (→ Kritik asiatischer Staaten)

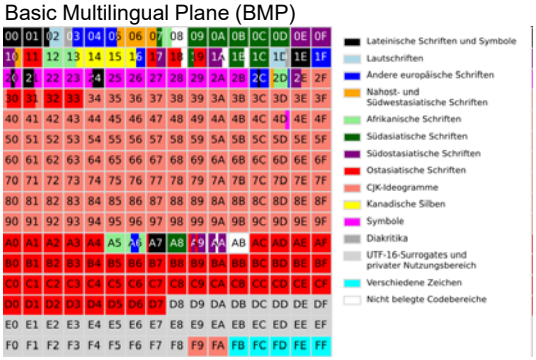
Zeichensequenzen: Sprachzeichen können durch Sequenzen mehrerer Unicode-Zeichen (Basiszeichen + Modifier) definiert werden
Beispiele:
- Basiszeichen “a” gefolgt von Modifier “^”: â
- “ü”: entweder einzelnes Unicode-Zeichen U+00CF “ü”
  oder Basiszeichen U+0075 “u” gefolgt von
  Modifier U+0308 “¨”

Problem: Zeichen sind „wild verstreut“
→ spezieller Sortieralgorithmus für ein Schriftsystem

Zeichenkodierung in Unicode

U+0000 ... U+10FFFF: 17 Codebereiche (Ebenen/Planes) zu je 16 Bit (65.536 Zeichen), hexadezimal kodiert mit führendem “U+”

Codebereiche: 0: Basic Multilingual Plane (BMP)

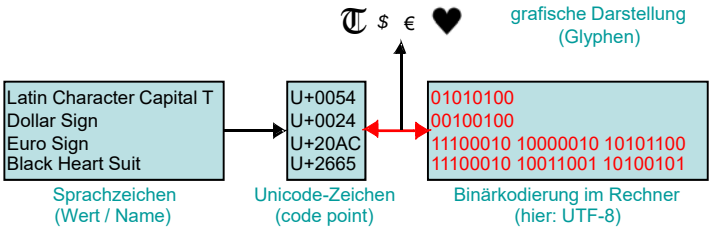


Unicode 5.1:
- 1.114.112 mögliche Zeichen
- ca. 100.000 Zeichen definiert

1: SMP: historische Schriftzeichen, Mahjongg-Steine, ...
2: SIP: seltene Zeichen wie alte vietnamesische Zeichen
3 .. 13: noch nicht belegt bzw. vergeben
14: einige Kontrollzeichen zur Sprachmarkierung
15 .. 16: für private Nutzung

Binärkodierung der Unicode-Zeichen im Rechner

→ zweistufige Abbildung



Binärkodierung der Unicode-Zeichen im Rechner

UTF-8: 8-Bit-Symbole - UNIX, E-Mail, WWW
Kodierung durch variabel lange Byteketten (1 .. 4 Bytes)

Unicode (hexadezimal)	UTF-8-Kodierung (binär)	Kommentar
0000 - 007F	0xxxxxxx	ASCII
0080 - 07FF	110xxxxx 10xxxxxx[a]	BMP
0800 - FFFF	1110xxxx 10xxxxxx 10xxxxxx	
01.0000 - 10.FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	Codebereiche 1 .. 16

UTF-16: 16-Bit-Symbole – Windows, Java, .NET
direkte Kodierung der BMP-Zeichen in einem 16-Bit-Halbwort
→ auf BMP optimiert, andere Codebereiche oft nicht umgesetzt
→ Bereiche 1 .. 16: zweimal 16 Bit nach aufwendigem Algorithmus

UTF-32: jedes Unicode-Zeichen in einem 32-Bit-Wort
→ einfach, ausreichend
→ hoher Speicherbedarf

- **Zeichen in C**

  → Integer (2K-Zahlen bzw. Dualzahlen)

  - **(signed | unsigned) char**
    - immer 8 Bit *(durch C festgelegt)*
    - signed: 7 Bit ASCII-Zeichen (bzw. 8 Bit ganze Zahl)
    - unsigned: erweiterter 8-Bit-Zeichensatz (bzw. 8 Bit natürliche Zahl)
    - ohne Modifier: Compiler-abhängig, ob signed oder unsigned

    - Ob Zeichen oder kleine Zahl, ist interpretationsabhängig

      Beispiel:
      ```
      char c = 100;
      printf ("%d\n%c ", c, c)
      ```
      Ausgabe:
      ```
      100
      d
      ```


Programmieren in C
für Elektrotechniker

Kapitel 3: Variablen, Datentypen und Operatoren (Teil 1)

- Variablen
- **Einfache Datentypen**
  - Ganze Zahlen (Integer)
  - Gleitkommazahlen
  - Zeichen
  - **Aufzählung**
  - Konstanten und lexikalische Konventionen
- Operatoren
- Abgeleitete Datentypen

Selbst definierte Datentypen: Aufzählung

- C erlaubt Programmierer, eigene, komplexere Datentypen zu definieren
  → *Strukturen, Unions, Bitfelder: später*
- Beispiele
  - enum: Aufzählungskonstanten
    → werden auf ganze Zahlen abgebildet *(1. = 0, 2. = 1, …)*

    Beispiele: - `enum boolean {FALSE, TRUE}` bzw.
    - `enum boolean {TRUE=1, FALSE=0}`

Programmieren in C
für Elektrotechniker

Kapitel 3: Variablen, Datentypen und Operatoren (Teil 1)

- Variablen
- **Einfache Datentypen**
  - Ganze Zahlen (Integer)
  - Gleitkommazahlen
  - Zeichen
  - Aufzählung
  - **Konstanten und lexikalische Konventionen**
- Operatoren
- Abgeleitete Datentypen

Lexikalische Konventionen

- **Zeichen**
  - Buchstaben in C: Klein- und Großbuchstaben plus '_' ohne Umlaute, 'ß'
    → in C jeweils 1 Byte
    (erweiterter Zeichensatz: **`wchar_t`** – u.a. für asiatische Zeichen)
    → C unterscheidet Groß- und Kleinbuchstaben (case sensitive)
  - **Trenner** von lexikalischen Einheiten
    - Zwischenräume (white spaces): ' ', Tabulator, Zeilenumbruch, Kommentar
    - Operatoren (Sonderzeichen): +, -, &&, …
    - Satzzeichen (Sonderzeichen, die keine Operatoren sind): [,], {, }, ;, …
  - **Namen** (für Variablen, Funktionen, Aufzählungskonstanten, …)
    → Zeichenfolgen aus Buchstaben und Ziffern – beginnen mit Buchstaben
    - korrekt: **`summe`**, **`SuMMe`**, **`SUM4x`**, **`x_quadrat`**
    - unzulässig: **`1x`**, **`x-quadrat`**, **`Schürmann, float`**
    - nicht empfohlen: **`_tmp`**, **`__tmp`** (→ typisch für Bibliotheken)

Lexikalische Konventionen

- **Styleguide**
  - Namen in Kleinbuchstaben:
    ```
    summe_von_x
    summeVonX
    ```
  - Konstanten in Großbuchstaben (s.u.):
    ```
    const float PI = 3.14;
    #define PI 3.14
    ```
  - Aufzählungskonstanten in Großbuchst.:
    ```
    enum bool {TRUE, FALSE}
    ```

Konstanten

Literale Konstanten (→ nur durch Wert bestimmt)
symbolische Konstanten (→ benannt)

Integer-Konstanten

- Je nach Größe: int → long int → unsigned long int
- Schreibweisen: - dezimal: **`1234`**
  - hexadezimal: **`0x12AB`**
  - Suffixe l, L, u, U: z.B. **`1234ul`**: unsigned long int

Gleitkomma-Konstanten

- **`300.0  3e2  3.E2  .3E3`**
- Default: Typ double
- Suffixe f, F, l, L: **`10.3f`** → float
  `10.3l` → long double

Konstanten

Literale Konstanten (→ nur durch Wert bestimmt)
symbolische Konstanten (→ benannt)

Zeichen-Konstanten

- ASCII-Zeichensatz, 8-Bit-Integer
  `'0'` == 48 → arithmetische Operationen möglich
  → typ. Vergleichsoperationen
- Escaping: Ersatzdarstellung von (Steuer-) Zeichen
  - **`'\n'`**: newline
  - **`'\t'`**: Tabulator
  - **`'\\'`**: backslash selbst
  - **`'\0'`**: 0 (Nullzeichen)
  - **`'\033'`**: ASCII-Zeichen 27 (Oktal 33)
- Zeichenketten:
  - „Bernd“:

'B'	'e'	'r'	'n'	'd'	'\0'

▪ Konstanten

Literale Konstanten (→ nur durch Wert bestimmt)
symbolische Konstanten (→ benannt)

▪ Aufzählungskonstanten

- Integer-Werte, die standardmäßig bei 0 beginnen
  - `enum test {A, B, C}; /* A = 0, B = 1, C = 2 */`
  - `enum test {A = 4, B = 5, C = 7};`
  - `enum boolean {FALSE, TRUE}; = _____ */`
  -
```
enum boolean b;
b = TRUE;
```
finition notwendig
  - `enum {A, B, C}`
    → Definition der Konstanten A =0, B =1, C = 2
- keine Typprüfung in C

▪ Beispiel

```
/* Datei: bool.c */
#include <stdio.h>
enum boolean {FALSE, TRUE};
int main (void)
{
    enum boolean b;
    b = TRUE;
    printf ("%d\n", b);
    b = FALSE;
    printf ("%d\n", b);
    b = 5;
    printf ("%d\n", b);
    return 0;
}
```

→ Was wird hier ausgegeben bzw. passiert?

Ausgabe von 1
0
5

- b = 5: schwere Typverletzung, wird in C aber nicht geprüft

- **Typ-Attribute const und volatile**
  - → Idee der Typattribute von C++ übernommen.
  - **Const**
    - Variable kann nur einmal initialisiert werden – anschließend vor Schreibzugriffen (Änderungen) geschützt.
    - Alternativen:

```
const double PI = 3.14;   /* double-Konstante */
#define PI 3.14           /* Textersetzung – s.u.*/
```

      → aber const-Variablen auch bei komplexen/zusammengesetzten Typen möglich *(s.u.)*
  - **Volatile**
    - Implementierungsabhängige Variablen, bei denen der Compiler keine Optimierung ausführen soll.
    - Beispiel: memory mapped I/O
      → Register einer E/A-Karte werden über Adressen angesprochen *(statt Ports)*.
      → Variablen müssen an diesen Adressen stehen und dürfen nicht an andere Stellen verschoben/optimiert werden.

Programmieren in C
für Elektrotechniker

Kapitel 3: Variablen, Datentypen und Operatoren (Teil 1)

- Variablen
- Einfache Datentypen
- **Operatoren**
- Abgeleitete Datentypen

Operatoren

- **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: **`?:`**)

- **Einstellige arithmetische Operatoren**
  - Vorzeichenoperatoren +A, -A
    - keine Nebeneffekte
  - Präfix-Inkrement-/-Dekrement-Operatoren ++A, --A
    - Ergebnis: A + 1, A - 1
    - Seiteneffekt: Wert des Operanden wird um 1 inkrementiert
    - Inkrement bei Pointern: Erhöhung um Objektgröße *(s.u.)*
  - Postfix-Inkrement-/-Dekrement-Operatoren A++, A--
    - Wie Präfix-Inkrement-/-Dekrement, jedoch
      Ergebnis: A

- **Beispiele: i++ und ++i**
  - **i++**

```
int i = 3;
printf ("%d", i++);      Ausgabe: _____
printf ("%d", i);        Ausgabe: _____
```

  - **++i**

```
int i = 3;
printf ("%d", ++i);      Ausgabe: _____
printf ("%d", i);        Ausgabe: _____
```


- **Zweistellige arithmetische Operatoren**
  - 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.)*

Relationale Operatoren

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

Logische Operatoren

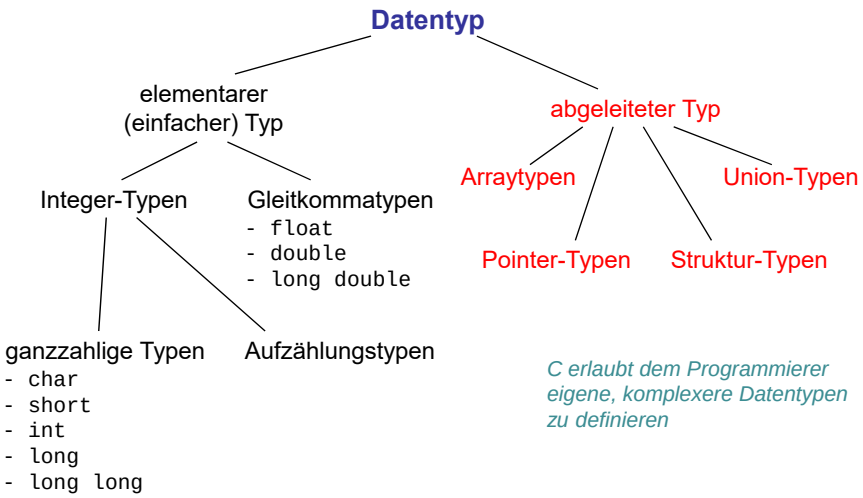
- Dürfen nicht mit den (logischen) Bitoperationen *(s.u.)* verwechselt werden.
  - Operator für das logische UND: **&&**
  - Operator für das logische ODER: **||**
  - Logischer Negationsoperator: **!**
  - Rückgabewert: int *(0 oder 1)*
  - Kein fester Datentyp „boolean“ in C
    → Verwendung von Integer *(0 = false, sonst true)*
  - Semantik der Funktionen in Kapitel 4 über Wertetabellen beschrieben.
  - Operanden werden von links nach rechts ausgewertet
    → Nebeneffekte der Auswertung der linken Operanden vor Auswertung der rechten Operanden.
  - Auswertung nur soweit notwendig.

Programmieren in C

für Elektrotechniker

Kapitel 3: Variablen, Datentypen und Operatoren (Teil 1)

- Variablen
- Einfache Datentypen
- Operatoren
- **Abgeleitete Datentypen**



C erlaubt dem Programmierer eigene, komplexere Datentypen zu definieren

Programmieren in C
für Elektrotechniker

Kapitel 3: Variablen, Datentypen und Operatoren (Teil 1)

- Variablen
- Einfache Datentypen
- Operatoren
- **Abgeleitete Datentypen**
  - **Arrays**
  - Strukturen
  - Pointer

Eindimensionale Arrays (dt. Felder)

- Array: Zusammenfassung mehrerer Variablen gleichen Typs
  (→ direkt hintereinander im Speicher).
- Deklaration: **<Typname> <Arrayname> [Anzahl]**
  Beispiel:
```
int a [5];            /* 5  ganze Zahlen */
char b [7];           /* 7 Zeichen */
float a[10], b, c[4]; /* möglich, aber nicht */
                      /* empfehlenswert */
```
- Kein Schlüsselwort „array“ in C
  (→ wird an ‚[‘, ‚]‘ erkannt)
- **`Anzahl`** muss eine positive ganze Zahl sein.
  - Konstante oder konstanter Ausdruck, keine Variable.
  - Größe kann nicht dynamisch während der Laufzeit festgelegt werden (hierfür: **`malloc ()`** – später)
- Lokale Variablen werden auf dem Stack abgelegt
  → große Arrays können zu Stacküberläufen führen
  (v.a. bei rekursiven Funktionen)

Eindimensionale Arrays (dt. Felder)

- Array: Zusammenfassung mehrerer Variablen gleichen Typs
  (→ direkt hintereinander im Speicher).
- Zugriff auf Array über Index (**`0 … Anzahl – 1`**)

  Beispiel:
```
int alpha [5]; /* Arraydefinition */
alpha[0] = 1;
alpha[1] = 2;
alpha[2] = 3;
alpha[3] = 4;
alpha[4] = 5;
```
- Achtung: In C keine Überprüfung der Grenzen:
  → **`alpha[5]`** würde nachfolgende Speicherzelle überschreiben.

(Auf UNIX-Rechnern wird mit dem C-Compiler das Werkzeug lint ausgeliefert, das erweiterte Prüfungen des Quellcodes durchführt).

Eindimensionale Arrays (dt. Felder)

- Array: Zusammenfassung mehrerer Variablen gleichen Typs
  (→ direkt hintereinander im Speicher).
- Vorteil von Arrays gegenüber n Einzelvariablen:
  Arrays ermöglichen Schleifen über Elemente (mit Hilfe von Indexvariablen).
- Beispiel:
```
const int ALPHA_L = 5;
int index = 0;
float summe = 0.0;
int alpha[ALPHA_L];
...
while (index < ALPHA_L) {
   summe = summe + alpha [index];
   index++;
}
printf ("\nDurchschnitt: %f", summe / index);
```
- Tipp:
  Array-Größen immer über symbolische Konstanten definieren
  (→ Programm besser änderbar).

- **Beispiel**

```
int MAX = 40;
int main (void) {
   int fahrenheit [MAX+1]; /* Tabelle fuer Fahrenheitwerte */
   int i = 0;
   while (i <= MAX) {
      fahrenheit[i] = ((9 * i) / 5) + 32;
      i++;
   }
   while (1) { /* Endlosschleife – Schleifen später genauer */
      printf ("Geben Sie bitte eine Temperatur zwischen");
      printf (" 0 und %d Grad Celsius ein ", MAX);
      printf ("(Abbruch durch Eingabe von -1):\n");
      scanf ("%d", &i);
      if (i >= 0 && i <= MAX)
         printf ("\n %d Grad Fahrenheit\n", fahrenheit [i]);
   }
   return 0;
}
```


- **Time-Space-Tradeoff**
  - Beim Fahrenheit-Beispiel hätte man statt der Verwendung eines Array mit Temperaturwerten auch jedes Mal die Ergebnistemperatur berechnen können.
  - In der Praxis werden häufig wiederkehrende Werte vorab berechnet, um das Programm zu beschleunigen (auf Kosten des Speicherverbrauchs).
  - Häufig lassen sich Speicherverbrauch und Laufzeit gegeneinander tauschen (→ „Time-Space-Tradeoff“).

Zeichenketten (strings)

- Zeichenketten werden als Array von Zeichen realisiert, deren Enden durch ein Nullzeichen **`\0`** markiert werden.
- Beispiel: **`char name [30]; /* max. 29 Zeichen plus \0 */`**
- Häufige Fehler:
  - Das Array wir vollständig mit Zeichen gefüllt, sodass kein Platz mehr für **`\0`**.
  - Beim zeichenweise Kopieren von Zeichenketten wird **`\0`** nicht mit kopiert, da Abbruchbedingung.

Beispiel

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

int main (void) {
   int MAX = 40;
   int i = 0;
   char eingabe [MAX+1];
   printf ("Bitte String eingeben (max. %d Zeichen): ", MAX);
   gets (eingabe); /* Einlesen eines Strings in string.h */
   while (eingabe[i] != '\0' && eingabe[i] != 'a')
      i++;
   if (eingabe[i] == '\0')
      printf ("\nKein 'a' enthalten.\n");
   else
      printf ("\na befand sich an der %d. Stelle\n", i + 1);
   return 0;
}
```

→ **Was macht das Programm?**

Programmieren in C
für Elektrotechniker

Kapitel 3: Variablen, Datentypen und Operatoren (Teil 1)

- Variablen
- Einfache Datentypen
- Operatoren
- **Abgeleitete Datentypen**
  - Arrays
  - **Strukturen**
  - Pointer

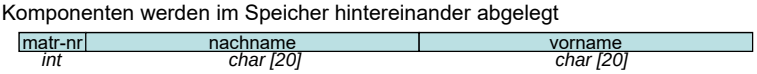
- **Strukturen (Datentyp struct)**
  - Bereits seit COBOL.
  - Aus verschiedenen Datentypen zusammengesetzter komplexer Datentyp
    (→ Kapselung logisch zusammengehöriger Daten).
  - Zusammenfassung einer festen Anzahl von benannten Komponenten

```
struct <Typ-Name>
{
    Komponententyp_1 Komponentenname_1;
    Komponententyp_2 Komponentenname_2;
    …
    Komponententyp_n Komponentenname_n1;
}
```

  - Beispiel:

```
struct student_t {
    int matr_nr;
    char nachname [20];
    char vorname [20];
}
```

  - Komponenten werden im Speicher hintereinander abgelegt



Weiteres Beispiel

- C erlaubt Programmierer, eigene, komplexere Datentypen zu definieren

Beispiel: Zeit in `time.h`

```
struct tm
{
  int tm_sec;   /* [0,59] */
  int tm_min;   /* [0,59] */
  int tm_hour;  /* [0,23] */
  int tm_mday;  /* Tag im Monat [1,31] */
  int tm_mon;   /* [0,11] */
  int tm_year;  /* Jahr seit 1900 */
  int tm_wday;  /* Wochentag [0,6] */
  int tm_yday;  /* Tag in Jahr [0,365] */
  int tm_isdst; /* Sommerzeit? */
}
```


Zugriff auf Struktur-Komponenten

- Ausgehend von Strukturvariablen mittels Punkt-Operator.

```
struct student_t stud;
stud.matr_nr = 123456;
```

- Beispiel:

```
int main (void) {
   struct kartesische_koordinaten {
      float x;
      float y;
   } punkt;
   punkt.x = 3;
   punkt.y = 4;
   printf ("\n%f %f", punkt.x, punkt.y);
}
```

- Vergleich von Strukturen muss komponentenweise erfolgen

```
struct student_t stud1, stud2;
stud1 == stud2
```
geht nicht

- Strukturen als Parameter und Rückgabewerte von Funktionen möglich.

Schachtelung

- Strukturen und Arrays können geschachtelt sein.
  Beispiel:

```
struct adresse
{
    char strasse [20];
    int hausnummer;
    int postleitzahl;
    char stadt [20];
};

struct student
{
    int matrikelnummer;
    char name [20];
    char vorname [20];
    struct adresse wohnort;
};
```

- Beispiele für die Definition von Strukturvariablen:

```
struct student meyer, mueller;
struct student jahrgang_2013 [200]; /* Array mit 200 Stud. */
```


Initialisierung von Strukturkomponenten

- Dynamische Initialisierung während des Programmlaufs muss komponentenweise erfolgen.
- Zeichenketten als Komponenten müssen mittels **`strcpy()`** initialisiert werden.

```
struct student_t stud;
stud.matr nr = 123456;
strcpy(stud.nachname, "Maier");
```

- Globale Strukturen können über eine Initialisierungsliste initialisiert werden.

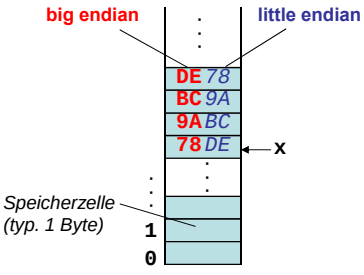
```
struct student maier =  {
                            66202,
                            "Maier",
                            "Herbert",
                            {
                                "Schillerplatz",
                                20,
                                73730,
                                "Esslingen"
                            }
                        };
```

Programmieren in C
für Elektrotechniker

Kapitel 3: Variablen, Datentypen und Operatoren (Teil 1)

- Variablen
- Einfache Datentypen
- Operatoren
- **Abgeleitete Datentypen**
  - Arrays
  - Strukturen
  - **Pointer**

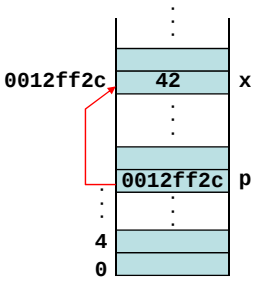
- Zugriff auf Daten
  - Alle Daten eines Programms (und Programm selbst) liegen im Arbeitsspeicher (und Register)
    → *vereinfachte Sicht.*



(Speicher-) Adresse = Nummer der Speicherzelle
→ jedes Byte eines Programms ist über seine Adresse ansprechbar

Bei komplexeren Daten wird das erste Byte des Datums adressiert.
little endian: niederwertigste Byte wird adressiert
big endian: höchstwertige Byte wird adressiert
Beispiel: `int x = 0x789abcde`

- **Pointer** (Verweis auf Objekte im Speicher)
  - Ein Pointer (dt. Zeiger) ist eine Variable, die die **Adresse** einer im Speicher befindlichen Variable oder Funktion aufnehmen kann.
  - Eine Pointer-Variable (Zeigervariable) verweist mit ihrem Wert auf ein Objekt im Speicher.




```
int x = 42;
int * p = &x;
```

p Pointer-Variable
Typ: Pointer auf **int**
Wert: Adresse von **x**.

Pointer können auch auf Funktionen oder Pointer verweisen.

- **Pointer** (Verweis auf Objekte im Speicher)

Speicherobjekte:	Pointer verweisen auf:
• Datenobjekte • Variablen, die keine Pointer sind • Pointervariablen • Funktionen	• Datenobjekte • Pointervariablen • Funktionen

▪ **Pointer** (Verweis auf Objekte im Speicher)

- Datenobjekte und Pointer haben jeweils einen Typ.
- Zeigt ein Pointer auf ein Datenobjekt,
  so muss der Typ des Pointer dem Typ des Datenobjekts entsprechen.
  →Pointer und Speicherobjekt sind über den Typ gekoppelt:
  - **`<Pointer auf Typ>`** verweist auf Speicherobjekt vom Typ **`<Typ>`**
  - Pointer auf unterschiedliche Typen können nicht gegenseitig zugewiesen werden.

Syntax: Den Pointer-Namen wird ein Stern vorangestellt.

```
Typname * Pointername
```

Datentyp des Pointer *Name des Pointer (Variable)*

→ von rechts nach links gelesen: „Pointername ist Pointer auf Typname“

Beispiel:
```
int * pointer1;
short * p;
```

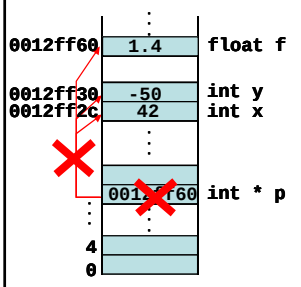
Achtung:

`int * p1, p2;`	entspr.	`int *p1;` `int p2;`
`int *P1, *p2;`	entspr.	`int * p1;` `int * p2;`

Tipp: In der Regel eine Zeile pro Variable.
→ auch vorteilhaft für Dokumentation

▪ **Wertebereich einer Pointer-Variablen**

- **p** vom Typ „Pointer auf **`Typname`**“
  = Menge aller Pointer,
  die auf Speicherobjekte vom Typ **`Typname`** zeigen können
  und Null-Pointer



- **`int * p = NULL;`**
  → **p** zeigt auf Adresse 0 *(→ kein reguläres Speicherobjekt!)*
  `<stddef.h>: #define NULL 0`
- Generell darf einem Pointer keine ganze Zahl zugewiesen werden (Ausnahme: NULL-Pointer = 0).
- NULL-Pointer
- Funktionen, die einen Pointer als Funktionsergebnis zurückgeben, können mit dem NULL-Pointer einen Fehlerfall kodieren.
- Ein Pointer sollte immer (mit **`NULL`**) initialisiert werden
  (zufälliger Wert lässt sich nicht von gültigen Adressen unterscheiden).

- **`<typ> * p;`**
  - reserviert noch keinen Speicher für ein Objekt vom Typ **`<typ>`**
  - reserviert Speicher für eine Adresse (Größe ist abhängig vom Adressraum)

▪ **Pointer** (Verweis auf Objekte im Speicher)

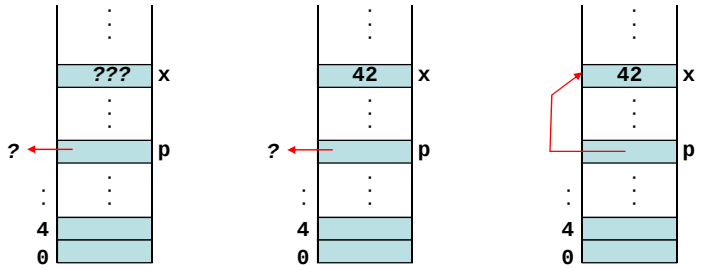
• Adressoperator &

Sei x eine Variable vom Typ Typname.
→ &x: Pointer bzw. Adresse auf Objekt x (Typ: Pointer auf Typname)

```
int x;
int * p;
```

```
x = 42;
```

```
p = &x;
```



Adressoperator liefert nur Pointer auf vorhandene Speicherobjekte
(nicht auf Registervariablen – s.u.).

▪ **Pointer** (Verweis auf Objekte im Speicher)

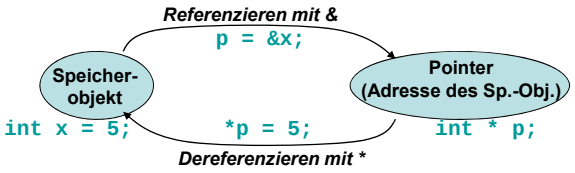
• Dereferenzierung *

Sei Pointer **p**: Zugriff auf ein Speicherobjekt.
Dereferenzierungsoperator (bzw. Inhaltsoperator) ***** auf **p** greift auf das Speicherobjekt zu, auf das **p** verweist.

Beispiel:

```
int x = 1;
int * p = &x;  /* p: Adresse von x */
*p = 2;        /* Dereferenzierung */
```

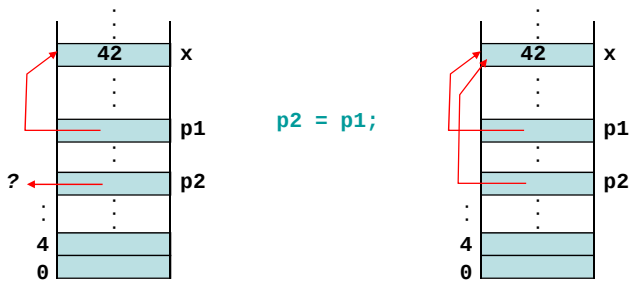
äquivalent zu `*&x = 2;`



- **Pointer** (Verweis auf Objekte im Speicher)
  - **Zuweisung**

```
p2 = p1;
```

  - nur, wenn **p1**, **p2** gleicher Typ *(oder Typ **void** – s.u.)*
  - nach Zuweisung verweisen **p1** und **p2** auf das selbe Objekt



- **Pointer** (Verweis auf Objekte im Speicher)
  - **Beispiel 1**

```
#include <stdio.h>

int main (void)
{
   float zahl = 3.5f;
   printf("Adresse von zahl: %p\n", &zahl);
   printf("Wert von zahl: %f\n", *&zahl);
   printf("*&*&*&*&*&zahl = %f\n", *&*&*&*&*&zahl);
   return 0;
}
```

→ **Ausgabe?**

```
Adresse von zahl: 0012ff60
Wert von zahl: 3.500000
*&*&*&*&*&zahl = 3.500000
```

- **Pointer** (Verweis auf Objekte im Speicher)
  - **Beispiel 2**

```
int main (void)
{
   int alpha = 2;
   int * p1;
   int * p2;
   p1 = &alpha;             /* Initialisierung */
   *p1 = 5;
   printf ("\n%d", alpha);
   *p1 = *p1 + 1;
   p2 = p1;                 /* Initialisierung */
   printf ("\n%d", *p2);
   return 0;
}
```

→ **Ausgabe?** 5
6

- **Pointer** (Verweis auf Objekte im Speicher)
  - **Beispiel 2**

```
int main (void)
{
   int alpha = 2;
   int * p1;
   int * p2 = p1;           /* Initialisierung */
   p1 = &alpha;             /* Initialisierung */
   *p1 = 5;
   printf ("\n%d", alpha);
   *p1 = *p1 + 1;
   printf ("\n%d", *p2);
   return 0;
}
```

→ **Ausgabe?** 5
→ **p2 zeigt auf eine unbekannte Adresse**
(→ vorauss. Adressfehler)

▪ **Pointer** (Verweis auf Objekte im Speicher)

• Beispiel 3

```
int * p;
*p = 6;
printf ("\n%d", *p);
```

→ **Ausgabe?**

Häufiger Fehler.
p ist noch nicht initialisiert (**p = NULL** oder beliebige Adresse).
→ Fehlermeldung, falls **p == NULL** oder
außerhalb des Adressraums des Programms.
→ **p** korrekte Adresse: beliebige Speicherzelle wird überschrieben.

▪ **Pointer** (Verweis auf Objekte im Speicher)

• Pointer auf `void`

- Annahme:
  Bei Definition eines Pointer **p** steht der Typ der Variablen, auf die **p** zeigen soll, noch nicht fest.
  - Definition eines Pointer **p** auf Typ **`void`** (→ untypisierter, generischer Pointer).
  - **p** kann später in Pointer eines anderen Typs gewandelt werden.
  - Pointer auf **`void`** ist mit allen Pointer-Typen kompatibel.
  - Pointer auf **`void`** darf nicht dereferenziert werden.

- Bei Zuweisungen in C darf links typfreier Pointer und rechts typisierter Pointer stehen und umgekehrt.

```
int x = 42;
void * p1;
int * p2 = &x;
p1 = p2;  /* korrekt */
p2 = p1;  /* korrekt */
```

- Pointer auf **`void`** umgeht bei Zuweisungen die Typprüfung
  (→ später mehr, Beispiel).

Zugriff auf Struktur-Komponenten *(s.o.)*

- Ausgehend von Strukturvariablen mittels Punkt-Operator.

```
struct student_t stud;
stud.matr_nr = 123456;
```

- Ausgehend von einem Pointer auf eine Struktur mittels Pfeil-Operator.

```
struct student_t * p_stud;
p_stud->matr_nr = 123456;
```
(entspr. `(* p_stud).matr_nr = 123456;`)

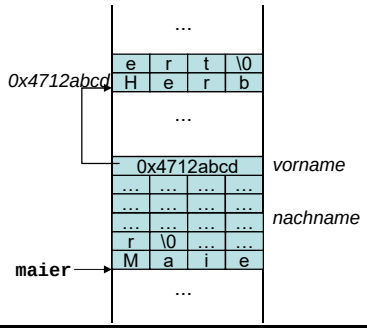
- Beispiel:

```
int main (void) {
   struct kartesische_koordinaten {
      float x;
      float y;
   };
   struct kartesische_koordinaten punkt;
   struct kartesische_koordinaten * punkt_ptr;
   punkt_ptr = &punkt;
   punkt_ptr->x = 3;
   punkt.y = 4;
   printf ("\n%f %f", punkt.x, punkt_ptr->y);
}
```


Zeichenketten in Strukturen

- Zeichenketten in Strukturen können sein
  - char-Arrays *(s.o.)*
  - Pointer auf char
- Beispiel:

```
struct name {
   char nachname [20];
   char * vorname;
}
struct name maier = {"Maier", "Herbert"};
```



- **Beispiel: Systemzeit in C/Unix**

```
struct tm {
    int tm_sec;    /* Sekunden - [0,59] */
    int tm_min;    /* Minuten - [0,59] */
    int tm_hour;   /* Stunden - [0,23] */
    int tm_mday;   /* Tag - [1,31] */
    int tm_mon;    /* Monat - [0,11] */
    int tm_year;   /* Jahre seit 1900 */
    int tm_wday;   /* Tage seit Sonntag - [0,6] */
    int tm_yday;   /* Tage seit 1. Januar -[0,365] */
    int tm_isdst;  /* Daylight Saving Time */
};
```

tm wird von der Funktion **localtime()** genutzt, um die Sekundenanzahl, die von der Funktion **time()** berechnet wird, strukturiert zurückzugeben.

```
#include <time.h>

int main (void) {
    time_t sekunden;
    struct tm * ortszeit;
    char * tag [7] = {"So", "Mo", "Di", "Mi", "Do", "Fr", "Sa"};

    time (&sekunden); /* Sekundenanzahl beim Programmstart */
    ortszeit = localtime (&sekunden);
    printf ("Stunden:Minuten:Sekunden \n");
    printf ("%02d:%02d:%02d\n\n", ortszeit->tm_hour,
                    ortszeit->tm_min, ortszeit->tm_sec);
    printf ("Wochentag: \t %s \n", tag[ortszeit->tm_wday]);
}
```