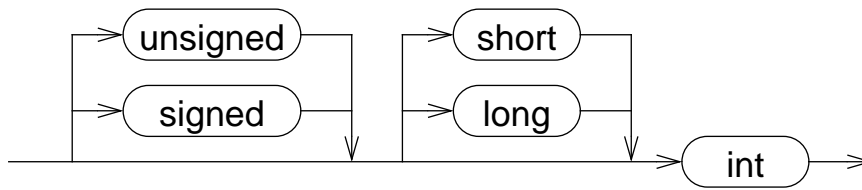


4 Ganze Zahlen

■ Definition



■ Speicherbedarf(`short int`) \leq Speicherbedarf(`int`) \leq Speicherbedarf(`long int`)

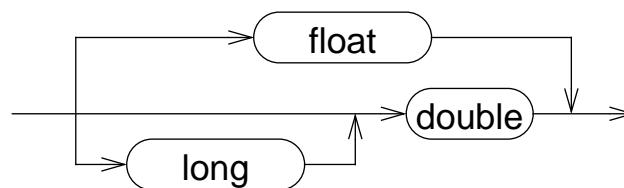
■ Speicherbedarf(`int`): meist 32 Bit

■ Konstanten (Beispiele):

42, -117
 035 (oktal = 29₁₀)
 0x10 (hexadezimal = 16₁₀)
 0x1d (hexadezimal = 29₁₀)

5 Fließkommazahlen

■ Definition



■ Speicherbedarf(`float`) \leq Speicherbedarf(`double`) \leq Speicherbedarf(`long double`)

■ Speicherbedarf(`float`): 32 Bit

■ Konstanten (Beispiele):

◆ normale Dezimalpunkt-Schreibweise

3.14, -2.718, 368.345, 0.003

1.0

aber nicht einfach 1 (wäre eine `int`-Konstante!)

◆ 10er-Potenz Schreibweise (368.345 = 3.68345 · 10², 0.003 = 3.0 · 10⁻³)

3.68345e2, 3.0e-3

6 Zeichen

- Bezeichnung: `char`
- Speicherbedarf: 1 Byte
- Repräsentation: ASCII-Code
zählt damit zu den ganzen Zahlen
- Konstanten: Zeichen durch `' '` geklammert
 - ◆ Beispiele: `'a'`, `'x'`
 - ◆ Sonderzeichen werden durch **Escape-Sequenzen** beschrieben
 - Tabulator: `'\t'` Backslash: `'\\'`
 - Zeilentrenner: `'\n'` Backspace: `'\b'`
 - Apostroph: `'\''`

6 Zeichen (2)

American Standard Code for Information Interchange (ASCII)

NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL
00	01	02	03	04	05	06	07
BS	HT	NL	VT	NP	CR	SO	SI
08	09	0A	0B	0C	0D	0E	0F
DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB
10	11	12	13	14	15	16	17
CAN	EM	SUB	ESC	FS	GS	RS	US
18	19	1A	1B	1C	1D	1E	1F
SP	!	"	#	\$	%	&	'
20	21	22	23	24	25	26	27
()	*	+	,	-	.	/
28	29	2A	2B	2C	2D	2E	2F
0	1	2	3	4	5	6	7
30	31	32	33	34	35	36	37
8	9	:	;	<	=	>	?
38	39	3A	3B	3C	3D	3E	3F
@	A	B	C	D	E	F	G
40	41	42	43	44	45	46	47
H	I	J	K	L	M	N	O
48	49	4A	3B	3C	3D	3E	3F
P	Q	R	S	T	U	V	W
50	51	52	53	54	55	56	57
X	Y	Z	[\]	^	_
58	59	5A	5B	5C	5D	5E	5F
`	a	b	c	d	e	f	g
60	61	62	63	64	65	66	67
h	i	j	k	l	m	n	o
68	69	6A	6B	6C	6D	6E	6F
p	q	r	s	t	u	v	w
70	71	72	73	74	75	76	77
x	y	z	{		}	~	DEL
78	79	7A	7B	7C	7D	7E	7F

7 Zeichenketten (Strings)

- Bezeichnung: `char *`
- Speicherbedarf: (Länge + 1) Bytes
- Repräsentation: Folge von Einzelzeichen,
letztes Zeichen: 0-Byte (ASCII-Wert 0)
- Werte: alle endlichen Folgen von `char`-Werten
- Konstanten: Zeichenkette durch " " geklammert
 - ◆ Beispiel: `"Dies ist eine Zeichenkette"`
 - ◆ Sonderzeichen wie bei `char`, " wird durch `\` " dargestellt
- Beispiel für eine Definition einer Zeichenkette:
`char *Mitteilung = "Dies ist eine Mitteilung\n";`

D.4 Ausdrücke

- Ausdruck = gültige Kombination von
Operatoren, Konstanten und Variablen
- Reihenfolge der Auswertung
 - ◆ Die Vorrangregeln für Operatoren legen die Reihenfolge fest,
in der Ausdrücke abgearbeitet werden
 - ◆ Geben die Vorrangregeln keine eindeutige Aussage,
ist die Reihenfolge undefiniert
 - ◆ Mit Klammern () können die Vorrangregeln überstimmt werden
 - ◆ Es bleibt dem Compiler freigestellt,
Teilausdrücke in möglichst effizienter Folge auszuwerten

D.5 Operatoren

1 Zuweisungsoperator =

→ Zuweisung eines Werts an eine Variable

■ Beispiel:

```
int a;
a = 20;
```

2 Arithmetische Operatoren

→ für alle `int` und `float` Werte erlaubt

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Rest bei Division, (modulo)
unäres -	negatives Vorzeichen (z. B. -3)
unäres +	positives Vorzeichen (z. B. +3)

■ Beispiel:

```
a = -5 + 7 * 20 - 8;
```

3 spezielle Zuweisungsoperatoren

→ Verkürzte Schreibweise für Operationen auf einer Variablen

$a \text{ op} = b \equiv a = a \text{ op } b$

mit $\text{op} \in \{ +, -, *, /, \%, \ll, \gg, \&, \wedge, | \}$

■ Beispiele:

```
a = -8;
a += 24;      /* -> a: 16 */
a /= 2;       /* -> a: 8  */
```

4 Vergleichsoperatoren

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

■ **Beachte!** Ergebnistyp `int`: wahr (true) = 1
 falsch (false) = 0

■ Beispiele:

```
a > 3
a <= 5
a == 0
if ( a >= 3 ) { ...
```


6 Bitweise logische Operatoren

→ Operation auf jedem Bit einzeln (Bit 1 = wahr, Bit 0 = falsch)

"nicht"	~	Antivalenz "exklusives oder"	∧	f	w
"und"	&		f	f	w
"oder"			w	w	f

■ Beispiele:

x	1	0	0	1	1	1	0	0
~x	0	1	1	0	0	0	1	1
7	0	0	0	0	0	1	1	1
x 7	1	0	0	1	1	1	1	1
x & 7	0	0	0	0	0	1	0	0
x ^ 7	1	0	0	1	1	0	1	1

7 Logische Shiftoperatoren

→ Bits werden im Wort verschoben

<<	Links-Shift
>>	Rechts-Shift

■ Beispiel:

x	1	0	0	1	1	1	0	0
x << 2	0	1	1	1	0	0	0	0

7 Inkrement / Dekrement Operatoren

<code>++</code>	inkrement
<code>--</code>	dekrement

- **linksseitiger Operator:** `++x` bzw. `--x`
 - es wird der Inhalt von `x` inkrementiert bzw. dekrementiert
 - das Resultat wird als Ergebnis geliefert
- **rechtsseitiger Operator:** `x++` bzw. `x--`
 - es wird der Inhalt von `x` als Ergebnis geliefert
 - anschließend wird `x` inkrementiert bzw. dekrementiert.

- **Beispiele:**

```
a = 10;
b = a++;      /* -> b: 10 und a: 11 */
c = ++a;     /* -> c: 12 und a: 12 */
```

8 Bedingte Bewertung

`A ? B : C`

- ➔ der Operator dient zur Formulierung von Bedingungen in Ausdrücken
- zuerst wird Ausdruck **A** bewertet
- ist **A ungleich 0**, so hat der gesamte Ausdruck als Wert den Wert des Ausdrucks **B**,
- sonst den Wert des Ausdrucks **C**

- **Beispiel:**

```
c = a>b ? a : b;      /* z = max(a,b) */
besser:
c = (a>b) ? a : b;
```

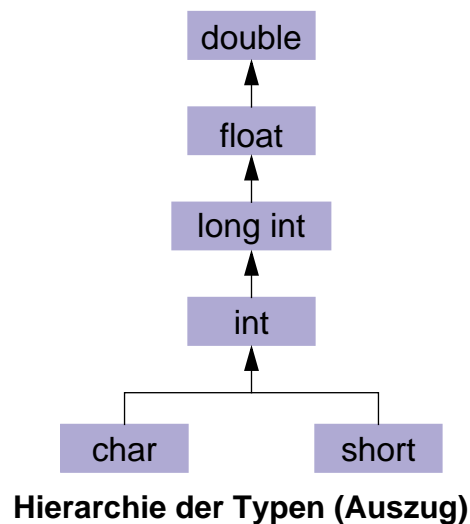

9 Komma-Operator

,

- ➔ der Komma-Operator erlaubt die Aneinanderreihung mehrerer Ausdrücke
- ein so gebildeter Ausdruck hat als Wert den Wert des letzten Teilausdrucks

10 Typumwandlung in Ausdrücken

- Enthält ein Ausdruck Operanden unterschiedlichen Typs, erfolgt eine automatische Umwandlung in den Typ des in der **Hierarchie der Typen** am höchsten stehenden Operanden. (*Arithmetische Umwandlungen*)



11 Vorrangregeln bei Operatoren

Operatorklasse	Operatoren	Assoziativität
unär	! ~ ++ -- + -	von rechts nach links
multiplikativ	* / %	von links nach rechts
additiv	+ -	von links nach rechts
shift	<< >>	von links nach rechts
relational	< <= > >=	von links nach rechts
Gleichheit	== !=	von links nach rechts
bitweise	&	von links nach rechts
bitweise	^	von links nach rechts
bitweise		von links nach rechts
logisch	&&	von links nach rechts
logisch		von links nach rechts
Bedingte Bewertung	?:	von rechts nach links
Zuweisung	= op=	von rechts nach links
Reihung	,	von links nach rechts

D.6 Einfacher Programmaufbau

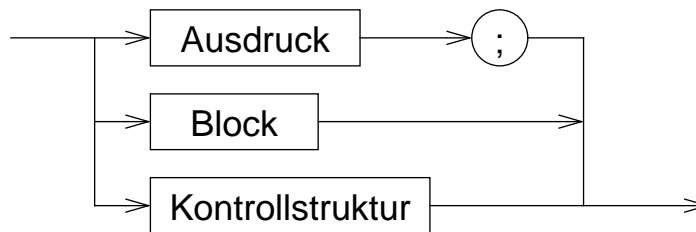
- Struktur eines C-Hauptprogramms
- Anweisungen und Blöcke
- Einfache Ein-/Ausgabe
- C-Präprozessor

1 Struktur eines C-Hauptprogramms

```
main()
{
    Variablendefinitionen
    Anweisungen
}
```

2 Anweisungen

Anweisung:



3 Blöcke

- Zusammenfassung mehrerer Anweisungen
- Lokale Variablendefinitionen → Hilfsvariablen
- Schaffung neuer Sichtbarkeitsbereiche (**Scopes**) für Variablen
 - ◆ bei Namensgleichheit ist immer die Variable des innersten Blocks sichtbar

```
main()
{
    int x, y, z;
    x = 1;
    {
        int a, b, c;
        a = x+1;
        {
            int a, x;
            x = 2;
            a = 3;
        }
        /* a: 2, x: 1 */
    }
}
```

4 Einfache Ein-/Ausgabe

- Jeder Prozess (jedes laufende Programm) bekommt unter Linux von der Shell als Voreinstellung drei Ein-/Ausgabekanäle:

stdin als Standardeingabe
stdout als Standardausgabe
stderr Fehlerausgabe

- Die Kanäle **stdin**, **stdout** und **stderr** sind in UNIX auf der Kommandozeile umlenkbar:

```
% prog < EingabeDatei > AusgabeDatei
```

4 Einfache Ein-/Ausgabe (2)

- Für die Sprache C existieren folgende primitive Ein-/Ausgabefunktionen für die Kanäle **stdin** und **stdout**:

getchar zeichenweise Eingabe
putchar zeichenweise Ausgabe
scanf formatierte Eingabe
printf formatierte Ausgabe

- folgende Funktionen ermöglichen Ein-/Ausgabe auf beliebige Kanäle (z. B. auch **stderr**)

getc, putc, fscanf, fprintf

5 Einzelzeichen E/A

- **`getchar()`, `getc()`** ein Zeichen lesen

◆ Beispiel:

```
int c;
c = getchar();
```

```
int c;
c = getc(stdin);
```

- **`putchar()`, `putc()`** ein Zeichen schreiben

◆ Beispiel:

```
char c = 'a';
putchar(c);
```

```
char c = 'a';
putc(c, stdout);
```

- Beispiel:

```
#include <stdio.h>

/*
 * kopiere Eingabe auf Ausgabe
 */
main()
{
    int c;
    while ( (c = getchar()) != EOF )
    {
        putchar(c);
    }
}
```

6 Formatierte Ausgabe

- Aufruf: `printf (format, arg)`
- **`printf`** konvertiert, formatiert und gibt die **Werte (`arg`)** unter der Kontrolle des Formatstrings **`format`** aus
 - ◆ die Anzahl der Werte (`arg`) ist abhängig vom Formatstring
- sowohl für **`format`**, wie für **`arg`** sind Ausdrücke zulässig
- **`format`** ist vom Typ **Zeichenkette (`string`)**
- **`arg`** muss dem durch das zugehörige **Formatelement** beschriebenen Typ entsprechen

6 Formatierte Ausgabe (2)

- die Zeichenkette **format** ist aufgebaut aus:
 - ↳ **einfachem Ausgabebetext**, der unverändert ausgegeben wird
 - ↳ **Formatelementen**, die Position und Konvertierung der zugeordneten **Werte** beschreiben

- Beispiele für **Formatelemente**:

Zeichenkette: %[-][min][.max]s
Zeichen: %[+][-][n]c
Ganze Zahl: %[+][-][n][l]d
Gleitkommazahl: %[+][-][n][.n]f

[] bedeutet *optional*

- Beispiel:

```
printf("a = %d, b = %d, a+b = %d", a, b, a+b);
```

7 C-Präprozessor — Kurzüberblick

- bevor eine C-Quelle dem C-Compiler übergeben wird, wird sie durch einen Makro-Präprozessor bearbeitet
- Anweisungen an den Präprozessor werden durch ein #-Zeichen am Anfang der Zeile gekennzeichnet
- die Syntax von Präprozessoranweisungen ist unabhängig vom Rest der Sprache
- Präprozessoranweisungen werden nicht durch ; abgeschlossen!
- wichtigste Funktionen:

#define	Definition von Makros
#include	Einfügen von anderen Dateien

8 C-Präprozessor — Makrodefinitionen

- Makros ermöglichen einfache textuelle Ersetzungen (parametrierbare Makros werden später behandelt)
- ein Makro wird durch die `#define`-Anweisung definiert
- Syntax:

```
#define Makroname Ersatztext
```

- eine Makrodefinition bewirkt, dass der Präprozessor im nachfolgenden Text der C-Quelle alle Vorkommen von ***Makroname*** durch ***Ersatztext*** ersetzt
- Beispiel:

```
#define EOF -1
```

9 C-Präprozessor — Einfügen von Dateien

- `#include` fügt den Inhalt einer anderen Datei in eine C-Quelldatei ein
- Syntax:

```
#include < Dateiname >
oder
#include "Dateiname "
```

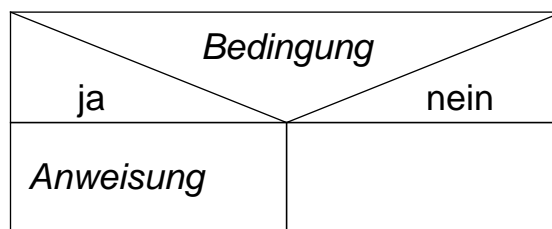
- mit `#include` werden *Header*-Dateien mit Daten, die für mehrere Quelldateien benötigt werden einkopiert
 - Deklaration von Funktionen, Strukturen, externen Variablen
 - Definition von Makros
- wird **Dateiname** durch `< >` geklammert, wird eine **Standard-Header-Datei** einkopiert
- wird **Dateiname** durch `" "` geklammert, wird eine Header-Datei des Benutzers einkopiert (vereinfacht dargestellt!)

D.7 Kontrollstrukturen

Kontrolle des Programmablaufs in Abhängigkeit von dem Ergebnis von Ausdrücken

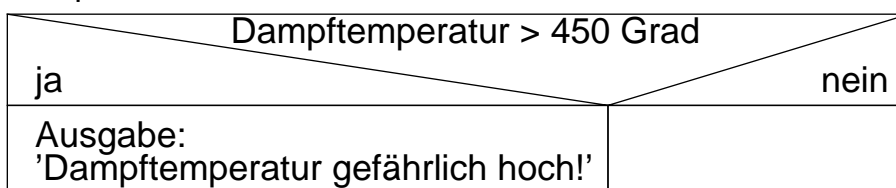
- Bedingte Anweisung
 - ◆ einfache Verzweigung
 - ◆ mehrfache Verzweigung
- Fallunterscheidung
- Schleifen
 - ◆ abweisende Schleife
 - ◆ nicht abweisende Schleife
 - ◆ Laufanweisung
 - ◆ Schleifensteuerung

1 Bedingte Anweisung



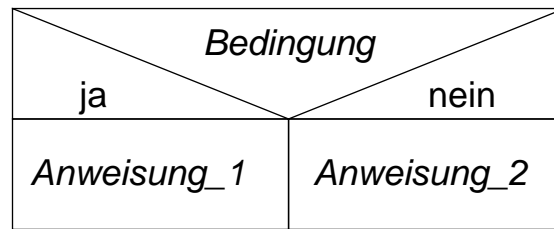
```
if ( Bedingung )
  Anweisung
```

- Beispiel:



```
if (temp >= 450.0)
  printf("Dampftemperatur gefaehrlich hoch!\n");
```


1 Bedingte Anweisung einfache Verzweigung

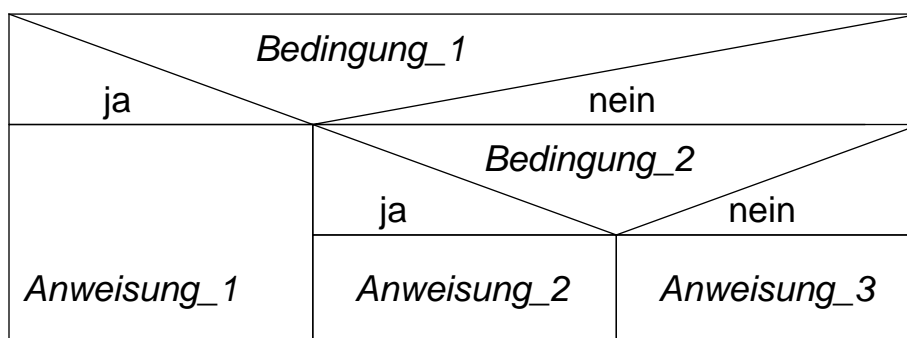


```

if ( Bedingung )
    Anweisung_1
else
    Anweisung_2

```

1 Bedingte Anweisung mehrfache Verzweigung



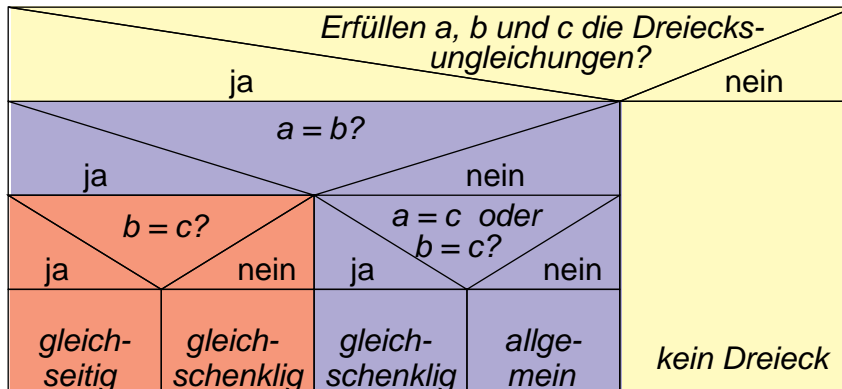
```

if ( Bedingung )
    Anweisung_1
else if ( Bedingung_2 )
    Anweisung_2
else
    Anweisung_3

```

1 Bedingte Anweisung mehrfache Verzweigung (2)

- Beispiel: Eigenschaften von Dreiecken — Struktogramm



1 Bedingte Anweisung mehrfache Verzweigung (3)

- Beispiel: Eigenschaften von Dreiecken — Programm

```
printf("Die Seitenlaengen %f, %f und %f bilden ", a, b, c);
```

```
if ( a < b+c && b < a+c && c < a+b )
    if ( a == b )
        if ( b == c )
            printf("ein gleichseitiges");
        else
            printf("ein gleichschenkliges");
    else
        if ( a==c || b == c )
            printf("ein gleichschenkliges");
        else
            printf("ein allgemeines");
else
    printf("kein");
printf(" Dreieck");
```

2 Fallunterscheidung

- Mehrfachverzweigung = Kaskade von if-Anweisungen
- verschiedene Fälle in Abhängigkeit von einem ganzzahligen Ausdruck

ganzzahliger Ausdruck = ?				
Wert1	Wert2			sonst
Anw. 1	Anw. 2		Anw. n	Anw. x

```

switch ( Ausdruck ) {
  case Wert_1:
    Anweisung_1
    break;
  case Wert_2:
    Anweisung_2
    break;
  .. .
  case Wert_n:
    Anweisung_n
    break;
  default:
    Anweisung_x
}

```

2 Fallunterscheidung — Beispiel

```

#include <stdio.h>

int main()
{
  int zeichen;
  int i;
  int ziffern, leer, sonstige;

  ziffern = leer = sonstige = 0;

  while ((zeichen = getchar()) != EOF)
    switch (zeichen) {
      case '0':
      case '1':
      case '2':
      case '3':
      case '4':
      case '5':
      case '6':
      case '7':
      case '8':
      case '9':
        ziffern++;
        break;

      case ' ':
      case '\n':
      case '\t':
        leer++;
        break;

      default:
        sonstige++;
    }

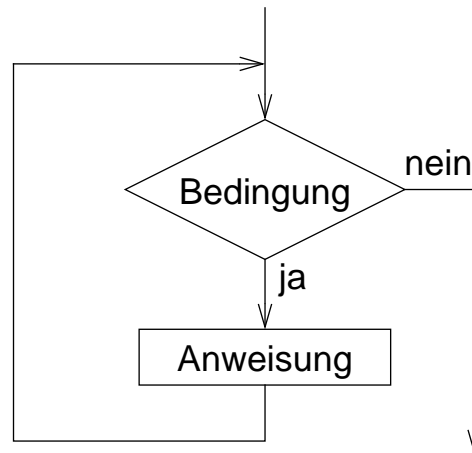
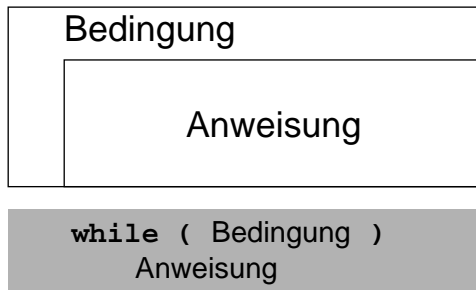
  printf("Zahl der Ziffern = %d\n", ziffern);
  printf("Zahl der Leerzeichen = %d\n", leer);
  printf("Zahl sonstiger Zeichen = %d\n", sonstige);
}

```

3 Schleifen

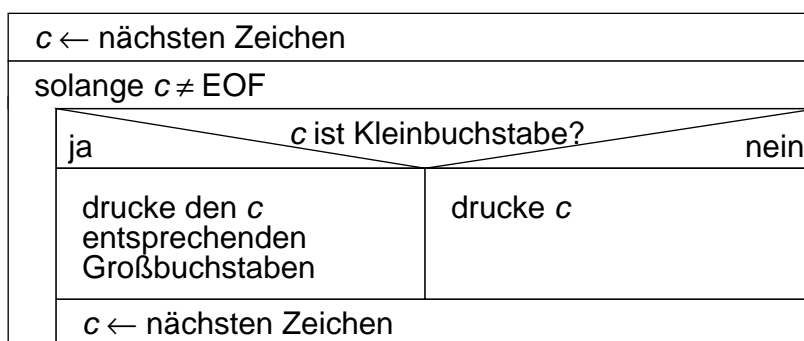
- Wiederholte Ausführung von Anweisungen in Abhängigkeit von dem Ergebnis eines Ausdrucks

4 abweisende Schleife



4 abweisende Schleife (2)

- Beispiel: Umwandlung von Klein- in Großbuchstaben

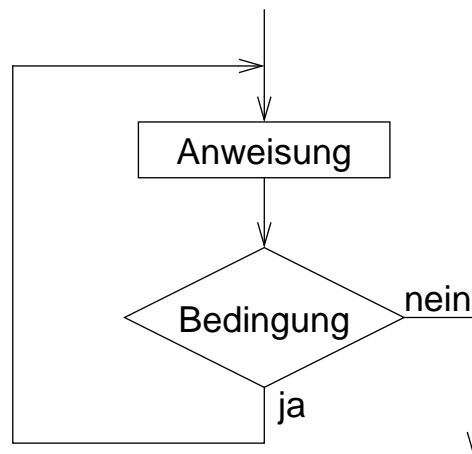
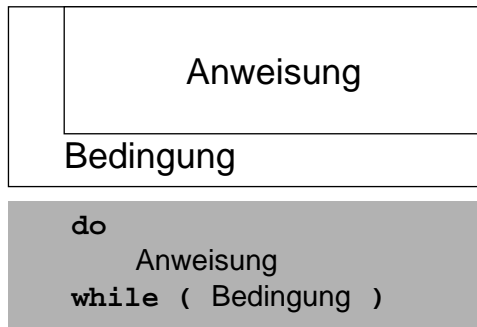


```
int c;
c = getchar();
while ( c != EOF ) {
    if ( c >= 'a' && c <= 'z' )
        putchar(c+'A'-'a');
    else
        putchar(c);
    c = getchar();
}
```

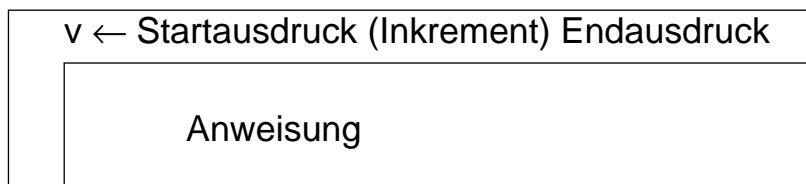
► abgekürzte Schreibweise

```
int c;
while ( (c = getchar()) != EOF )
    if ( c >= 'a' && c <= 'z' )
        putchar(c+'A'-'a');
    else
        putchar(c);
```

5 nicht-abweisende Schleife



6 Laufanweisung



```
for (v = Startausdruck; v <= Endausdruck; v += Inkrement)
  Anweisung
```

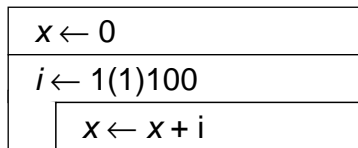
allgemein:

```
for (Ausdruck_1; Ausdruck_2; Ausdruck_3)
  Anweisung
```

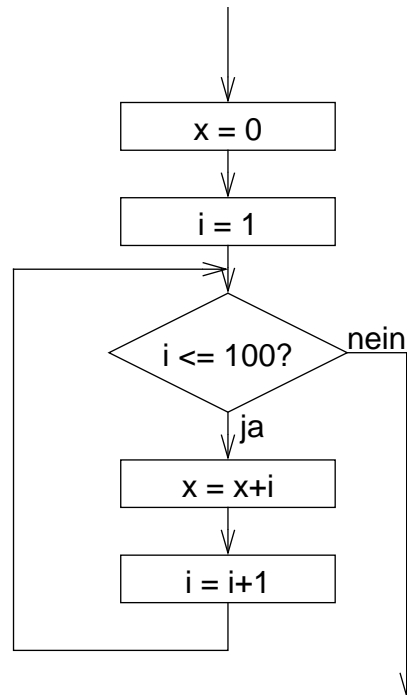
```
Ausdruck_1;
while (Ausdruck_2) {
  Anweisung
  Ausdruck_3;
}
```

6 Laufanweisung (2)

- Beispiel: Berechne $x = \sum_{i=1}^{100} i$



```
x = 0;
for ( i=1; i<=100; i++)
    x += i;
```



7 Schleifensteuerung

- break

- ◆ bricht die umgebende Schleife bzw. **switch**-Anweisung ab

```
char c;

do {
    if ( (c = getchar()) == EOF ) break;
    putchar(c);
}
while ( c != '\n' );
```

- continue

- ◆ bricht den aktuellen **Schleifendurchlauf** ab
 ◆ setzt das Programm mit der Ausführung des Schleifenkopfes fort

D.8 Funktionen

1 Überblick

- **Funktion =**
Programmstück (Block), das mit einem **Namen** versehen ist und dem zum Ablauf **Parameter** übergeben werden können
- Funktionen sind die elementaren Bausteine für Programme
 - ➔ gliedern umfangreiche, schwer überblickbare Aufgaben in kleine Komponenten
 - ➔ erlauben die Wiederverwendung von Programmkomponenten
 - ➔ verbergen Implementierungsdetails vor anderen Programmteilen (**Black-Box-Prinzip**)

1 überblick (2)

- ➔ Funktionen dienen der Abstraktion
- Name und Parameter abstrahieren
 - vom tatsächlichen Programmstück
 - von der Darstellung und Verwendung von Daten
- Verwendung
 - ◆ mehrmals benötigte Programmstücke können durch Angabe des Funktionsnamens aufgerufen werden
 - ◆ Schrittweise Abstraktion (**Top-Down-** und **Bottom-Up-Entwurf**)

2 Beispiel Sinusberechnung

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

double sinus (double x)
{
    double summe;
    double x_quadrat;
    double rest;
    int k;

    k = 0;
    summe = 0.0;
    rest = x;
    x_quadrat = x*x;

    while ( fabs(rest) > 1e-9 ) {
        summe += rest;
        k += 2;
        rest *= -x_quadrat/(k*(k+1));
    }
    return(summe);
}
```

```
int main()
{
    double wert;

    printf("Berechnung des Sinus von ");
    scanf("%lf", &wert);
    printf("sin(%lf) = %lf\n",
           wert, sinus(wert));
    return(0);
}
```

- beliebige Verwendung von `sinus` in Ausdrücken:

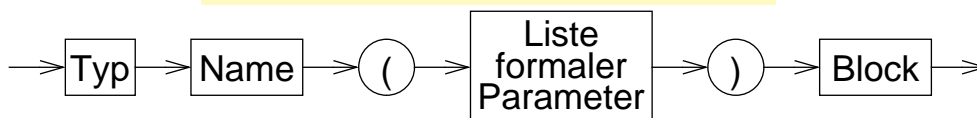
```
y = exp(tau*t) * sinus(f*t);
```

3 Funktionsdefinition

- Schnittstelle (Typ, Name, Parameter) und die Implementierung

- ◆ Beispiel:

```
int addition ( int a, int b ) {
    int ergebnis;
    ergebnis = a + b;
    return ergebnis;
}
```



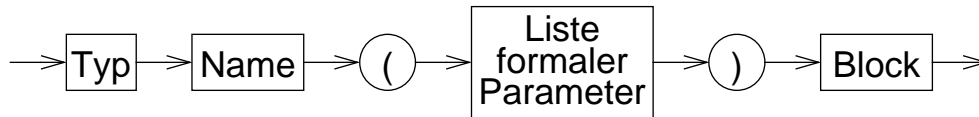
- Typ

- ◆ Typ des Werts, der am Ende der Funktion als Wert zurückgegeben wird
- ◆ beliebiger Typ
- ◆ `void` = kein Rückgabewert

- Name

- ◆ beliebiger Bezeichner, kein Schlüsselwort

3 Funktionsdefinition (2)



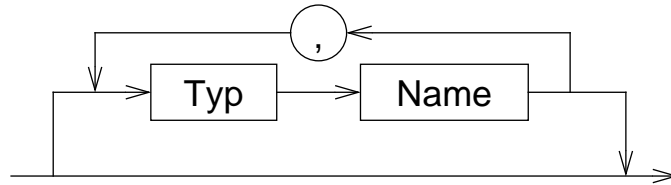
■ Liste formaler Parameter

◆ **Typ:** beliebiger Typ

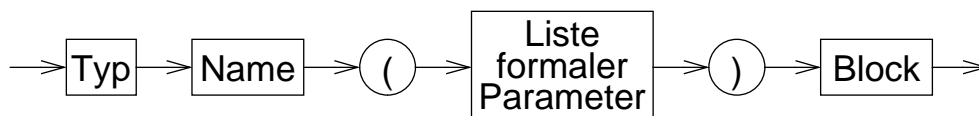
◆ **Name:**
beliebiger Bezeichner

◆ die formalen Parameter stehen innerhalb der Funktion für die Werte, die beim Aufruf an die Funktion übergeben wurden (= **aktuelle Parameter**)

◆ die formalen Parameter verhalten sich wie Variablen, die im **Funktionsrumpf** definiert sind und mit den aktuellen Parametern vorbelegt werden



3 Funktionsdefinition (3)



■ Block

◆ beliebiger Block

◆ zusätzliche Anweisung

```
return ( Ausdruck );
```

oder

```
return;
```

bei void-Funktionen

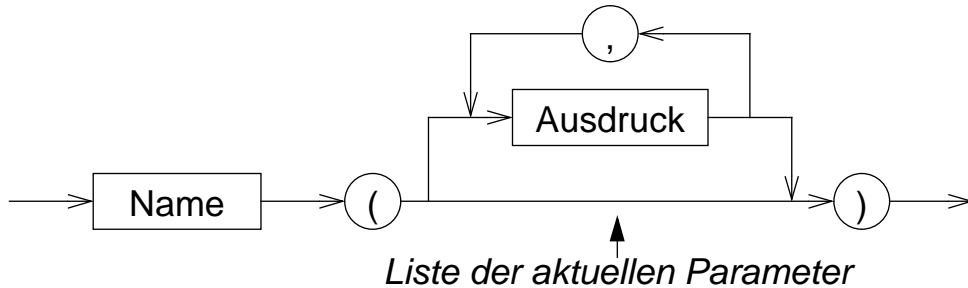
- Rückkehr aus der Funktion: das Programm wird nach dem Funktionsaufruf fortgesetzt
- der Typ des Ausdrucks muss mit dem Typ der Funktion übereinstimmen
- die Klammern können auch weggelassen werden

4 Funktionsaufruf

- Aufruf einer Funktion aus dem Ablauf einer anderen Funktion

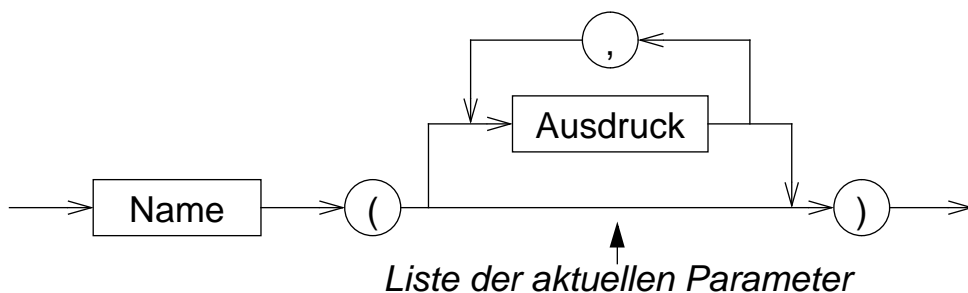
◆ Beispiel:

```
int main ( ) {
    int summe;
    summe = addition(3,4);
    ...
}
```



- Jeder Funktionsaufruf ist ein Ausdruck
- `void`-Funktionen können keine Teilausdrücke sein
 - ◆ wie Prozeduren in anderen Sprachen (z. B. Pascal)

4 Funktionsaufruf (2)



- Die Ausdrücke in der Parameterliste werden ausgewertet, **bevor** in die Funktion gesprungen wird
 - ➔ **aktuelle Parameter**
- Anzahl und Typen der Ausdrücke in der Liste der aktuellen Parameter müssen mit denen der formalen Parameter in der Funktionsdefinition übereinstimmen
- Die Auswertungsreihenfolge der Parameterausdrücke ist **nicht** festgelegt

5 Beispiel

```
float power (float b, int e)
{
    float prod = 1.0;
    int i;

    for (i=1; i <= e; i++)
        prod *= b;
    return(prod);
}
```

```
float x, y;

y = power(2+x,4)+3;
```

≡

```
float x, y, power;
{
    float b = 2+x;
    int e = 4;
    float prod = 1.0;
    int i;

    for (i=1; i <= e; i++)
        prod *= b;
    power = prod;
}
y=power+3;
```

6 Regeln

- Funktionen werden global definiert
 - ↳ keine lokalen Funktionen/Prozeduren wie z. B. in Pascal
- `main()` ist eine normale Funktion, die aber automatisch als erste beim Programmstart aufgerufen wird
 - Ergebnis vom Typ `int` - wird an die Shell zurückgeliefert (in Kommandoprozeduren z. B. abfragbar)
- rekursive Funktionsaufrufe sind zulässig
 - ↳ eine Funktion darf sich selbst aufrufen (z. B. zur Fakultätsberechnung)

```
fakultaet(int n)
{
    if ( n == 1 )
        return(1);
    else
        return( n * fakultaet(n-1) );
}
```

6 Regeln (2)

- Funktionen müssen **deklariert** sein, bevor sie aufgerufen werden
 - = Rückgabetyt und Parametertypen müssen dem Compiler bekannt sein
 - ◆ durch eine Funktionsdefinition ist die Funktion automatisch auch deklariert
- wurde eine verwendete Funktion vor ihrer Verwendung nicht deklariert, wird automatisch angenommen
 - Funktionswert vom Typ `int`
 - 1 Parameter vom Typ `int`
 - ➔ **schlechter Programmierstil → fehleranfällig**

6 Regeln (2)

- Funktionsdeklaration
 - ◆ soll eine Funktion vor ihrer Definition verwendet werden, kann sie durch eine **Deklaration** bekannt gemacht werden
 - ◆ Syntax:

Typ Name (Liste formaler Parameter);

 - Parameternamen können weggelassen werden, die Parametertypen müssen aber angegeben werden!
 - ◆ Beispiel:


```
double sinus(double);
```