

# D Einführung in die Programmiersprache C

## D.1 C vs. Java

### ■ Java: objektorientierte Sprache

- zentrale Frage: aus welchen Dingen besteht das Problem
- Gliederung der Problemlösung in Klassen und Objekte
- Hierarchiebildung: Vererbung auf Klassen, Teil-Ganze-Beziehungen
- Ablauf: Interaktion zwischen Objekten

### ■ C: imperative / prozedurale Sprache

- zentrale Frage: welche Aktivitäten sind zur Lösung des Problems auszuführen
- Gliederung der Problemlösung in Funktionen
- Hierarchiebildung: Untergliederung einer Funktion in Teilefunktionen
- Ablauf: Ausführung von Funktionen

## D.1 C vs. Java

---

### 1 C hat nicht

---

- Klassen und Vererbung
- Objekte
- umfangreiche Klassenbibliotheken

### 2 C hat

---

- Zeiger und Zeigerarithmetik
- Präprozessor
- Funktionsbibliotheken

## D.2 Sprachüberblick

### 1 Erstes Beispiel

- Die Datei `hello.c` enthält die folgenden Zeilen:

```
/* say "hello, world" */
main()
{
    printf("hello, world\n");
}
```

- Die Datei wird mit dem Kommando `cc` übersetzt:

<code>% cc hello.c</code>	(C-Compiler)
oder	
<code>% gcc hello.c</code>	(GNU-C-Compiler)

dadurch entsteht eine Datei `a.out`, die das ausführbare Programm enthält.

- ausführbares Programm liegt in Form von Maschinencode des Zielprozessors vor (kein Byte- oder Zwischencode)!

# 1 Erstes Beispiel (2)

- Mit der Option `-o` kann der Name der Ausgabedatei auch geändert werden – z. B.

```
% cc -o hello hello.c
```

- Das Programm wird durch Aufruf der Ausgabedatei ausgeführt:

```
% ./hello
hello, world
%
```

- Kommandos werden so in einem Fenster mit UNIX/Linux-Kommandointerpreter (Shell) eingegeben
  - es gibt auch integrierte Entwicklungsumgebungen (z. B. Eclipse)

## 2 Aufbau eines C-Programms

- frei formulierbar - **Zwischenräume** (Leerstellen, Tabulatoren, Newline und Kommentare) werden i. a. ignoriert - sind aber zur eindeutigen Trennung direkter benachbarter Worte erforderlich
- **Kommentar** wird durch /\* und \*/ geklammert  
keine Schachtelung möglich
- **Identifier** (Variablenamen, Marken, Funktionsnamen, ...) sind aus Buchstaben, gefolgt von Ziffern oder Buchstaben aufgebaut
  - "\_" gilt hierbei auch als Buchstabe
  - Schlüsselwörter wie **if**, **else**, **while**, usw. können nicht als **Identifier** verwendet werden
  - **Identifier** müssen vor ihrer ersten Verwendung **deklariert** werden
- Anweisungen werden generell durch ; abgeschlossen

### 3 Allgemeine Form eines C-Programms:

```
/* globale Variablen */
...
/* Hauptprogramm */
main(...)
{
    /* lokale Variablen */
    ...
    /* Anweisungen */
    ...
}

/* Unterprogramm 1 */
function1(...)
{
    /* lokale Variablen */
    ...
    /* Anweisungen */
    ...
}

/* Unterprogramm n */
functionN(...)
{
    /* lokale Variablen */
    ...
    /* Anweisungen */
    ...
}
```

## 4 wie ein C-Programm nicht aussehen sollte:

```

#define o define
#define _o write
#define _oo (unsigned)
#define _o_o_ 1
#define _o_ char
#define _oo goto
#define _oo_ read
#define o_o for
#define o_main
#define o_if
#define oo_ 0
#define _o(_,_,_)(void)_o(_,_,_oo(_))
#define _o(o_o_<<((o_o_<<(o_o_<<o_o_))+ (o_o_<<o_o_)))
+(o_o_<<(o_o_<<(o_o_<<o_o_)))
o_(){_o_=oo_,_,_,_o_[_o];_oo_____;_____:_=_o-o_
____:
_o(o_o_,____,_=_o_o_<____?_-
_o_o:_);_o_o(_;_o(o_o_,"\b",o_o_),____--);
_o(o_o_," ",o_o_);_o_(--____)_oo
____;_o(o_o_,"\n",o_o_);____:_o(_=_oo_(
oo_,____,_o))_oo____;}
```

sieht eher wie Morse-Code aus, ist aber ein **gültiges** C-Programm.

## D.3 Datentypen

### ■ Datentypen

- Konstanten
- Variablen



- ◆ Ganze Zahlen
- ◆ Fließkommazahlen
- ◆ Zeichen
- ◆ Zeichenketten

# 1 Was ist ein Datentyp?

## ■ Menge von Werten

+

Menge von Operationen auf den Werten

◆ **Konstanten** Darstellung für einen konkreten Wert (2, 3.14, 'a')

◆ **Variablen** Namen für Speicherplätze,  
die einen Wert aufnehmen können

→ Konstanten und Variablen besitzen einen **Typ**

## ■ Datentypen legen fest:

◆ Repräsentation der Werte im Rechner  
◆ Größe des Speicherplatzes für Variablen  
◆ erlaubte Operationen

## ■ Festlegung des Datentyps

◆ implizit durch Verwendung und Schreibweise (Zahlen, Zeichen)  
◆ explizit durch **Deklaration** (Variablen)

## 2 Standardtypen in C

- Eine Reihe häufig benötigter Datentypen ist in C vordefiniert

<b>char</b>	Zeichen (im ASCII-Code dargestellt, 8 Bit)
<b>int</b>	ganze Zahl (16 oder 32 Bit)
<b>float</b>	Gleitkommazahl (32 Bit) etwa auf 6 Stellen genau
<b>double</b>	doppelt genaue Gleitkommazahl (64 Bit) etwa auf 12 Stellen genau
<b>void</b>	ohne Wert

## 2 Standardtypen in C (2)

- Die Bedeutung der Basistypen kann durch vorangestellte **Typ-Modifier** verändert werden

### **short, long**

legt für den Datentyp **int** die Darstellungsbreite (i. a. 16 oder 32 Bit) fest.

Das Schlüsselwort **int** kann auch weggelassen werden

### **long double**

**double**-Wert mit erweiterter Genauigkeit (je nach Implementierung) – mindestens so genau wie **double**

### **signed, unsigned**

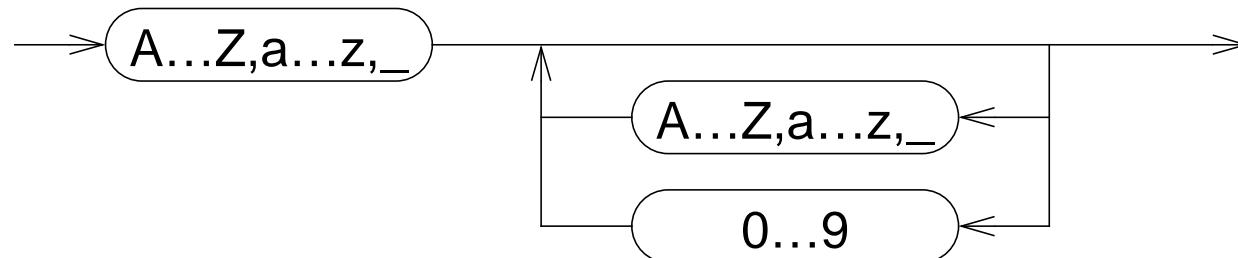
legt für die Datentypen **char, short, long** und **int** fest, ob das erste Bit als Vorzeichenbit interpretiert wird oder nicht

### 3 Variablen

#### ■ Variablen haben:

- ◆ **Namen** (Bezeichner)
- ◆ Typ
- ◆ zugeordneten Speicherbereich für einen Wert des Typs  
Inhalt des Speichers (= **aktueller Wert** der Variablen) ist veränderbar!
- ◆ **Lebensdauer**  
wann wird der Speicherplatz angelegt und wann freigegeben

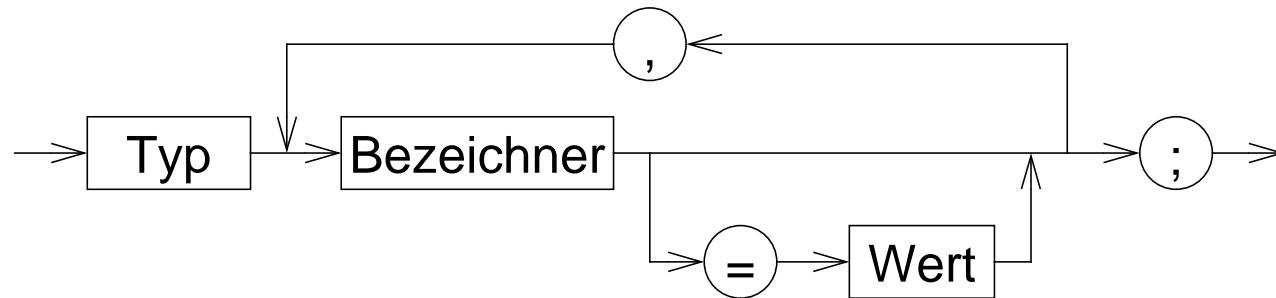
#### ■ Bezeichner



(Buchstabe oder \_,  
evtl. gefolgt von beliebig vielen Buchstaben, Ziffern oder \_)

### 3 Variablen (2)

- Typ und Bezeichner werden durch eine **Variablen-Deklaration** festgelegt (= dem Compiler bekannt gemacht)
  - ◆ reine Deklarationen werden erst in einem späteren Kapitel benötigt
  - ◆ vorerst beschränken wir uns auf Deklarationen in **Variablen-Definitionen**
- eine **Variablen-Definition** deklariert eine Variable und reserviert den benötigten Speicherbereich



## 3 Variablen (3)

### ■ Variablen-Definition: Beispiele

```
int a1;
float a, b, c, dis;
int anzahl_zeilen=5;
char Trennzeichen;
```

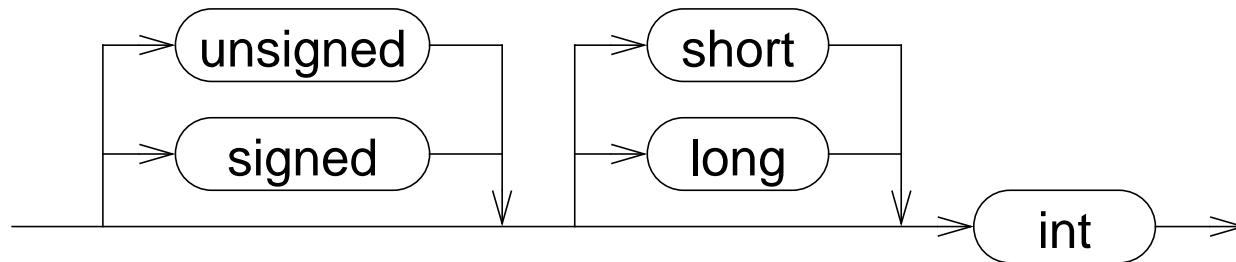
#### ◆ Position im Programm:

- nach jeder "{"
- außerhalb von Funktionen
- neuere C-Standards und der GNU-C-Compiler erlauben Definitionen an beliebiger Stelle im Programmcode: Variable ab der Stelle gültig

- Wert kann bei der Definition initialisiert werden
- Wert ist durch Wertzuweisung und spezielle Operatoren veränderbar
- Lebensdauer ergibt sich aus der Programmstruktur

## 4 Ganze Zahlen

### ■ Definition



- Speicherbedarf(`short int`)  $\leq$  Speicherbedarf(`int`)  $\leq$  Speicherbedarf(`long int`)
- Speicherbedarf(`int`): meist 32 Bit
- Konstanten (Beispiele):

`42, -117`

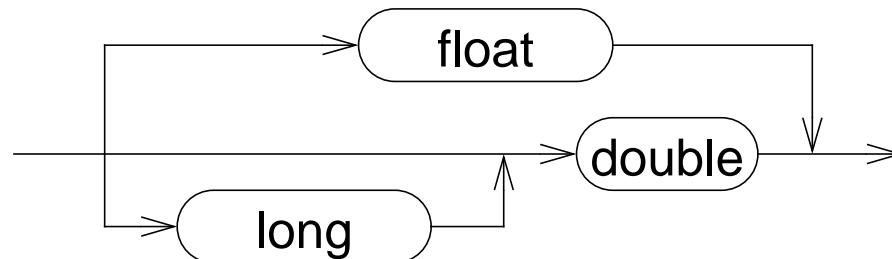
`035` (oktal =  $29_{10}$ )

`0x10` (hexadezimal =  $16_{10}$ )

`0x1d` (hexadezimal =  $29_{10}$ )

## 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 ( $368.345 = 3.68345 \cdot 10^2$ ,  $0.003 = 3.0 \cdot 10^{-3}$ )  
 $3.68345e2, 3.0e-3$

## 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)

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

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

### ■ Beispiel:

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

### 3 spezielle Zuweisungsoperatoren

- Verkürzte Schreibweise für Operationen auf einer Variablen

$a \ op= b \equiv a = a \ op b$   
 mit  $op \in \{ +, -, *, /, \%, <<, >>, \&, ^, | \}$

- 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 ) { ...
```

## 5 Logische Operatoren

- Verknüpfung von Wahrheitswerten (wahr / falsch)

"nicht"

!		
f	w	
w	f	

"und"

;&&	f	w
f	f	f
w	f	w

"oder"

	f	w
f	f	w
w	w	w

- ◆ Wahrheitswerte (Boole'sche Werte) werden in C generell durch int-Werte dargestellt:

► Operanden in einem Ausdruck:

Operand = 0: falsch

Operand  $\neq$  0: wahr

► Ergebnis eines Ausdrucks:

falsch: 0

wahr: 1

## 5 Logische Operatoren (2)

### ■ Beispiel:

```
a = 5; b = 3; c = 7;
a > b && a > c
  {1} und {0}
  0
```

- Die Bewertung solcher Ausdrücke wird abgebrochen, sobald das Ergebnis feststeht!

```
(a > c) && ((d=a) > b)
  {0}          {wird nicht ausgewertet}
  ↓
Gesamtergebnis=falsch
```

→ (d=a) wird nicht ausgeführt

## 6 Bitweise logische Operatoren

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

"nicht"

$\sim$

"und"

$\&$

"oder"

$|$

*Antivalenz*  
"exklusives oder"

$\wedge$	f	w
f	f	w
w	w	f

### Beispiele:

x

1	0	0	1	1	1	0	0
---	---	---	---	---	---	---	---

$\sim$ 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  $\wedge$  7

1	0	0	1	1	0	1	1
---	---	---	---	---	---	---	---

## 7 Logische Shiftoperatoren

- Bits werden im Wort verschoben

`<<` Links-Shift

`>>` Rechts-Shift

- Beispiel:

x	<table border="1"><tr><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr></table>	1	0	0	1	1	1	0	0
1	0	0	1	1	1	0	0		
$x << 2$	<table border="1"><tr><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	1	1	1	0	0	0	0
0	1	1	1	0	0	0	0		

## 7 Inkrement / Dekrement Operatoren

++	inkrement
--	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; /* N = max(a,b) */
```

**besser:**

c = (a>b) ? a : b;

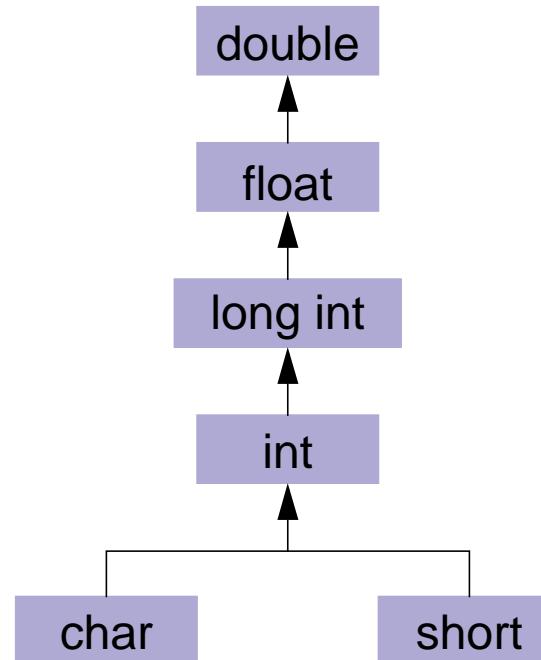
## 9 Komma-Operator

,

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

## 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*)



Hierarchie der Typen (Auszug)

# 11 Vorrangregeln bei Operatoren

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

## D.6 Einfacher Programmaufbau

---

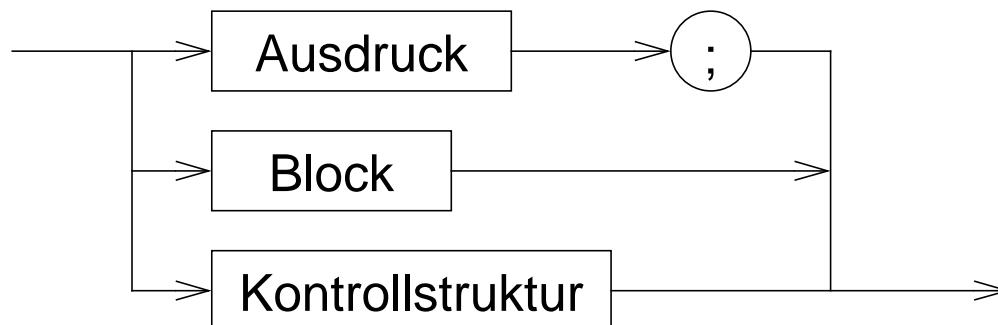
- Struktur eines C-Hauptprogramms
- Anweisungen und Blöcke
- Einfache Ein-/Ausgabe
- C-Präprozessor

# 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 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 Ausgabetext**, 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][1]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!)

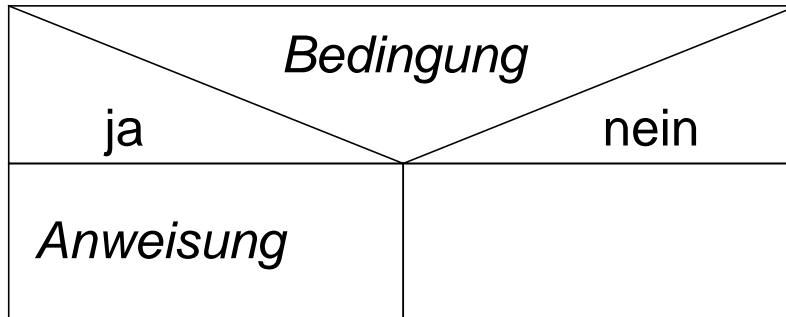
## D.7 Kontrollstrukturen

---

Kontrolle des Programmablaufs in Abhängigkeit von dem Ergebnis von Ausdrücken

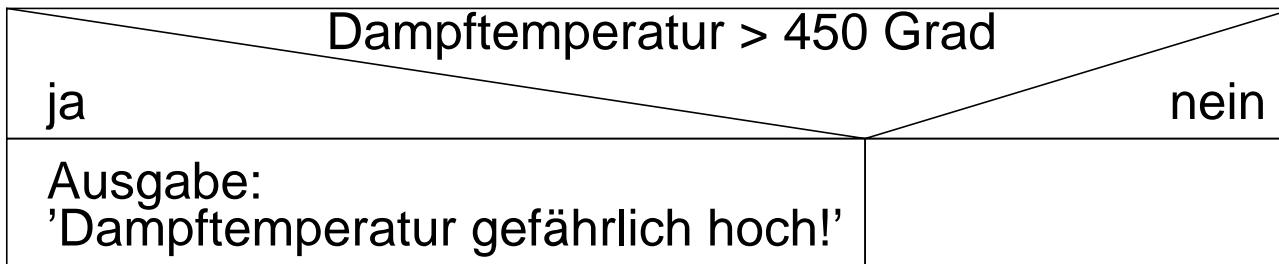
- Bedingte Anweisung
  - ◆ einfache Verzweigung
  - ◆ mehrfache Verzweigung
- Fallunterscheidung
- Schleifen
  - ◆ abweisende Schleife
  - ◆ nicht abweisende Schleife
  - ◆ Laufanweisung
  - ◆ Schleifensteuerung

# 1 Bedingte Anweisung



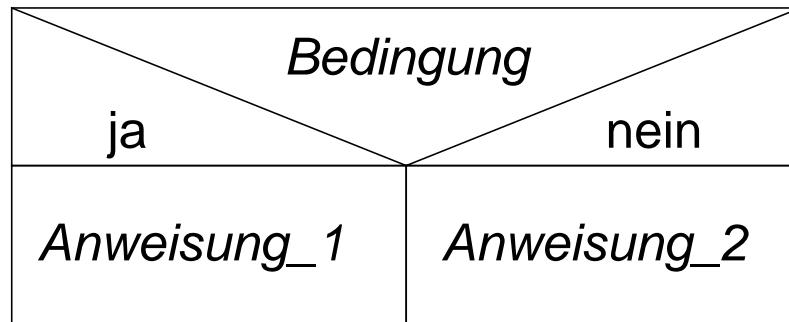
```
if ( Bedingung )
    Anweisung
```

## ■ Beispiel:



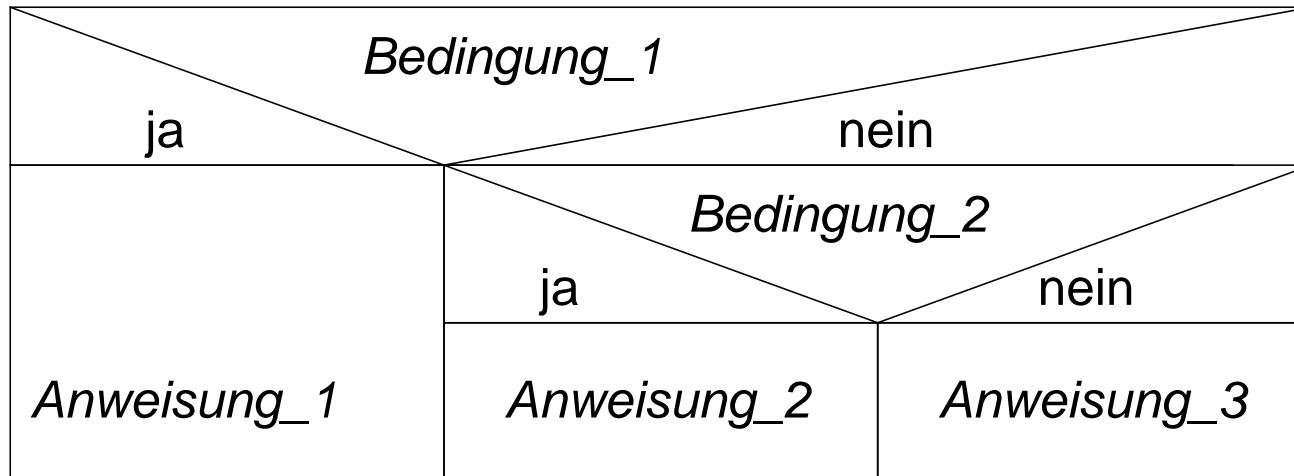
```
if (temp >= 450.0)
    printf("Dampftemperatur gefaehrlich hoch!\n");
```

# 1 Bedingte Anweisung einfache Verzweigung



```
if ( Bedingung )
    Anweisung_1
else
    Anweisung_2
```

# 1 Bedingte Anweisung mehrfache Verzweigung

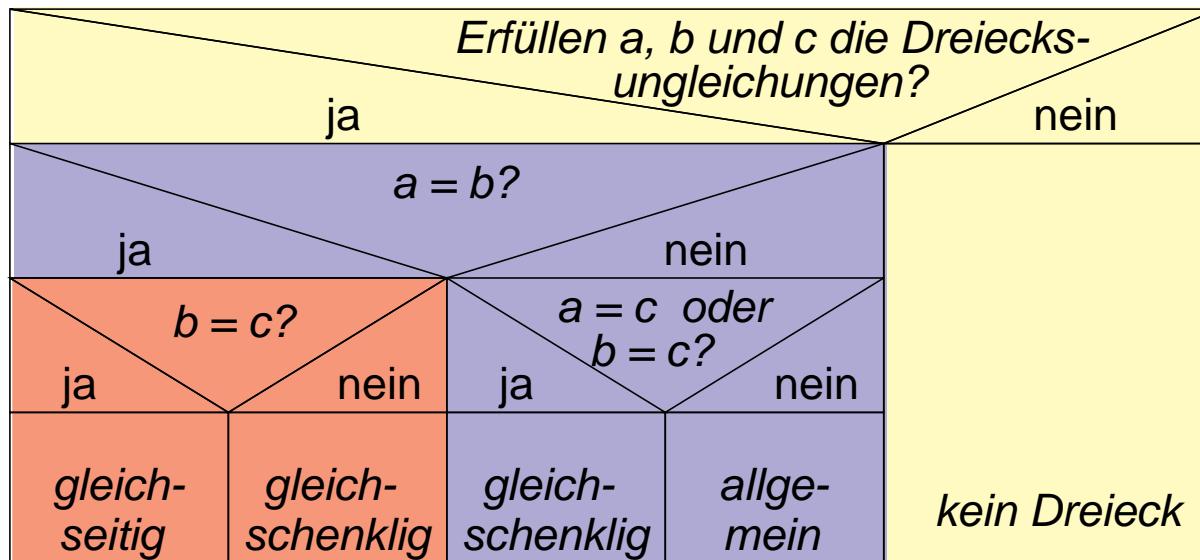


```

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)

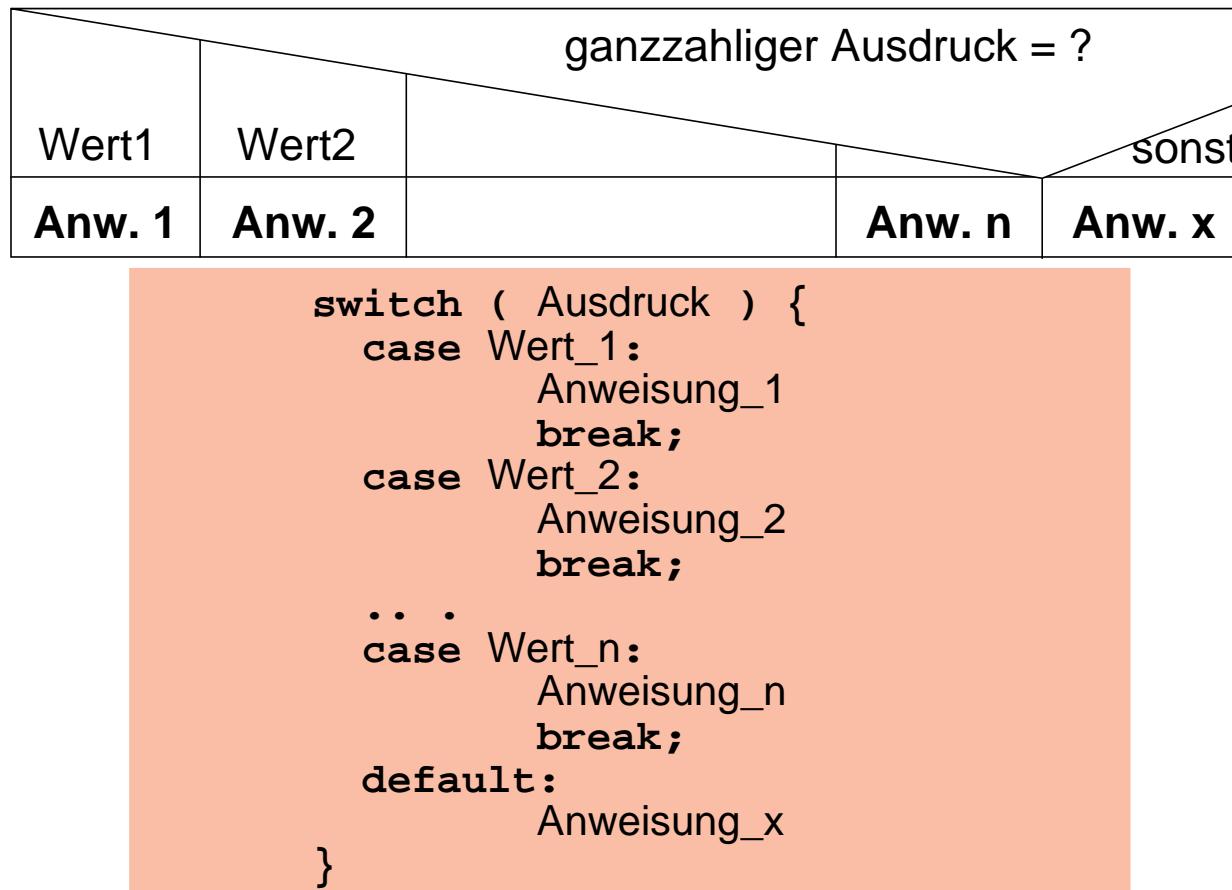
- Beispiel: Eigenschaften von Dreiecken — Programm

```
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



## 2 Fallunterscheidung — Beispiel

```
#include <stdio.h>

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

    ziffern = leer = sonstige = 0;

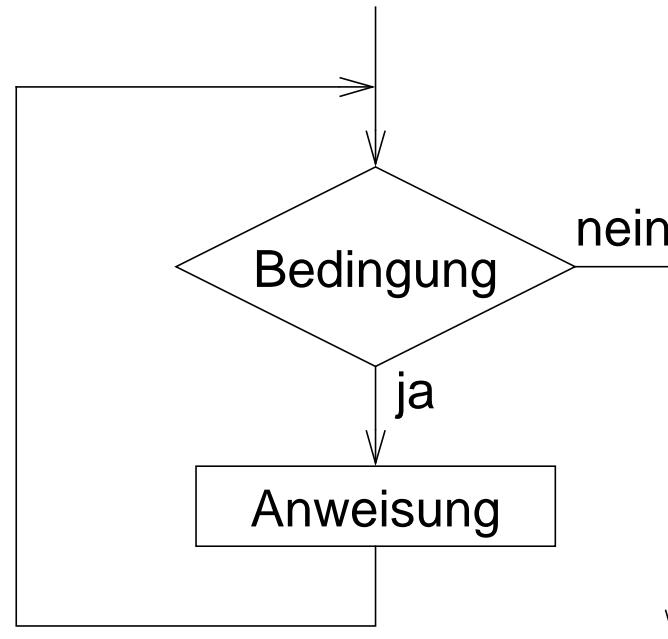
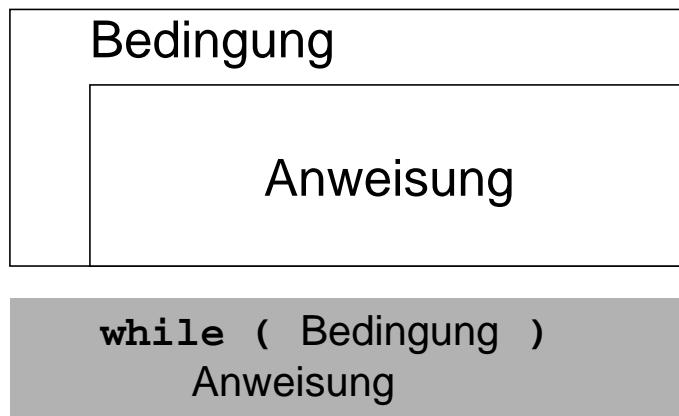
    while ((zeichen = getchar()) != EOF)
        switch (zeichen) {
            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
                ziffern++;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

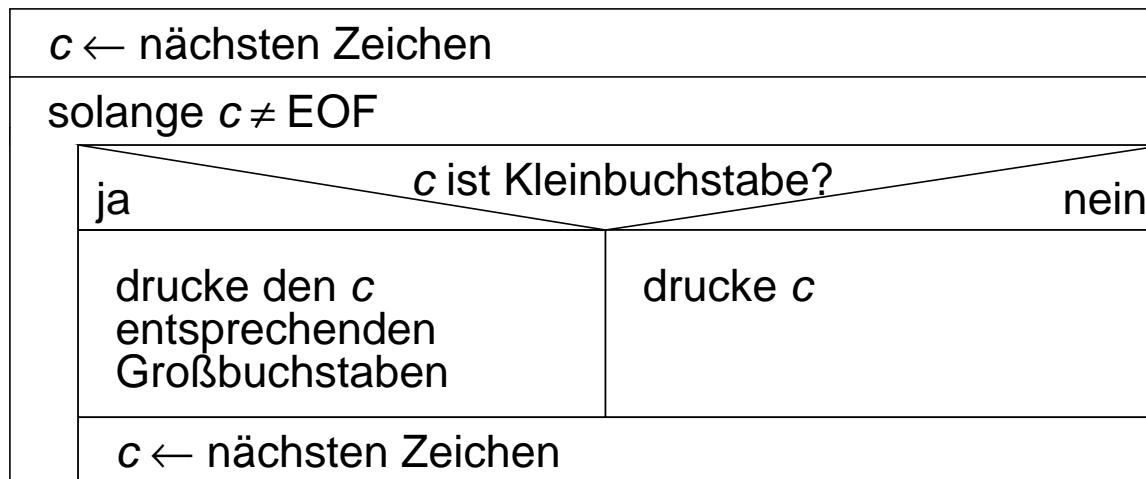
- Wiederholte Ausführung von Anweisungen in Abhängigkeit von dem Ergebnis eines Ausdrucks

## 4 abweisende Schleife



## 4 abweisende Schleife (2)

### Beispiel: Umwandlung von Klein- in Großbuchstaben



```

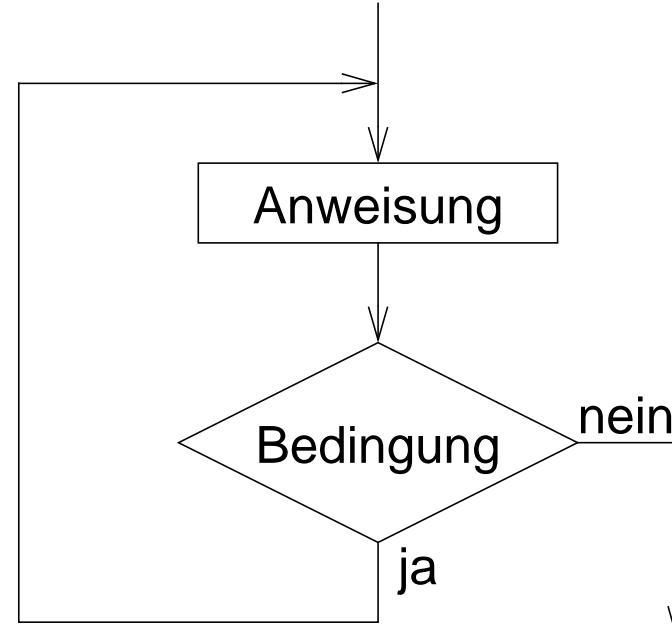
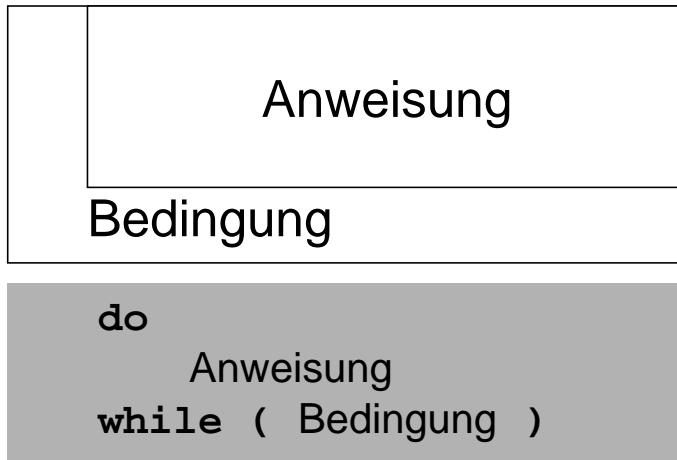
int c;
c = getchar();
while ( c != EOF ) {
    if ( c >= 'a' && c <= 'z' )
        putchar(c+'A'-'a');
    else
        putchar(c);
    c = getchar();
}
  
```

► 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

```
v ← Startausdruck (Inkrement) Endausdruck
```

```
    Anweisung
```

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

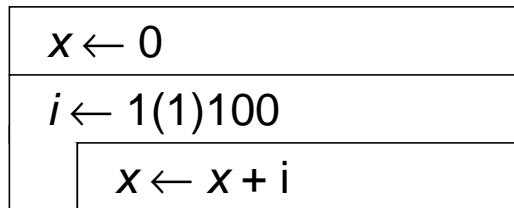
**allgemein:**

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

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

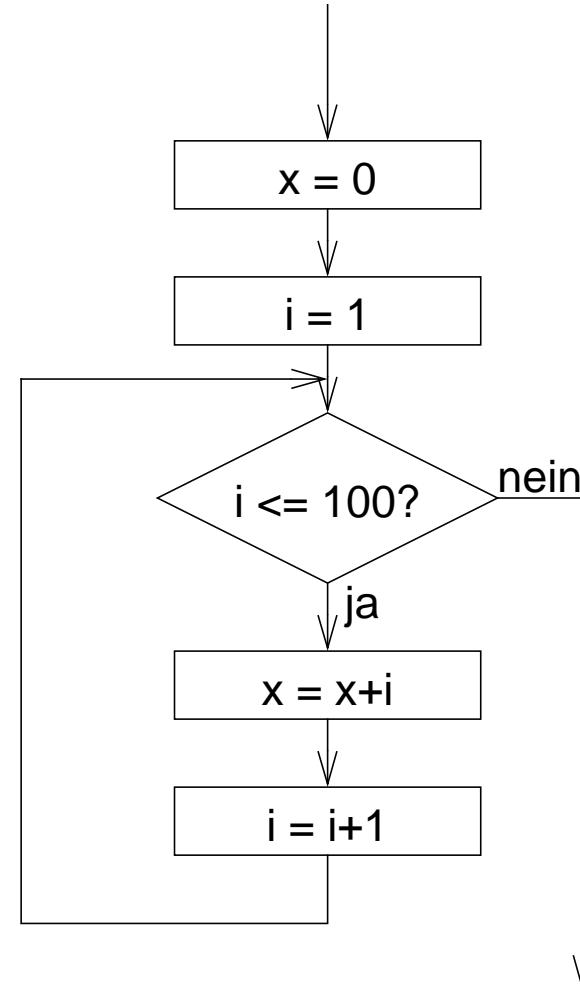
## 6 Laufanweisung (2)

■ 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:

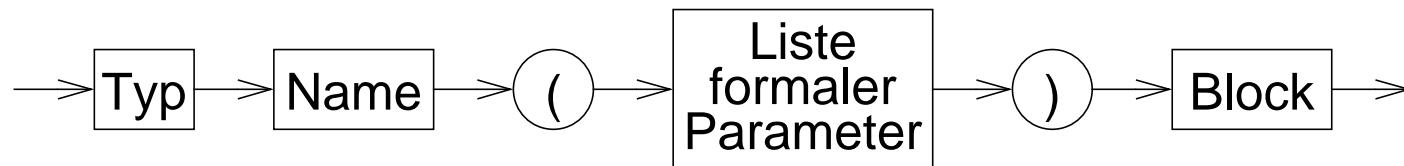
`y = exp(tau*t) * sinus(f*t);`

### 3 Funktionsdefinition

#### ■ Schnittstelle (Typ, Name, Parameter) und die Implementierung

- ◆ Beispiel:

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



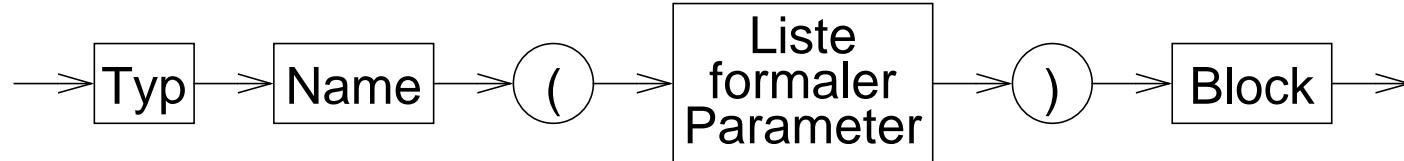
#### ■ Typ

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

#### ■ Name

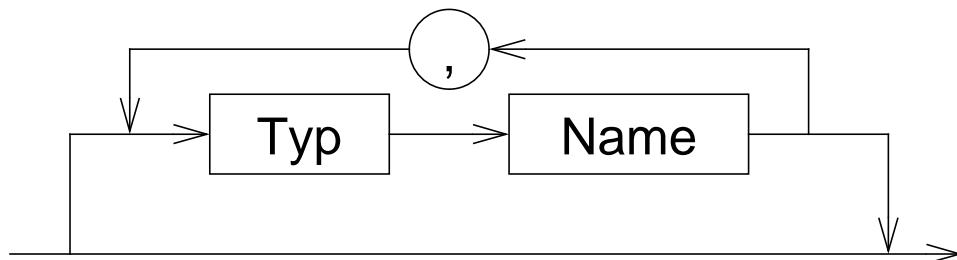
- ◆ beliebiger Bezeichner, kein Schlüsselwort

### 3 Funktionsdefinition (2)

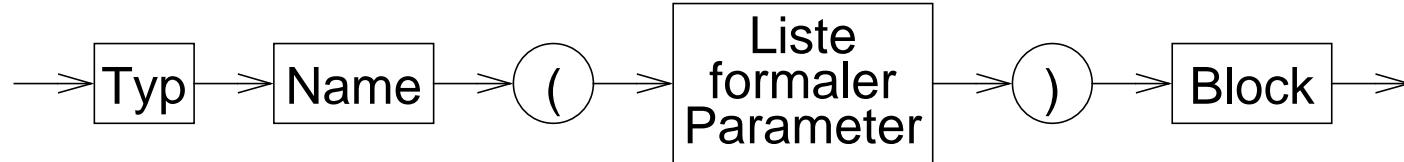


#### ■ Liste formaler Parameter

- ◆ **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

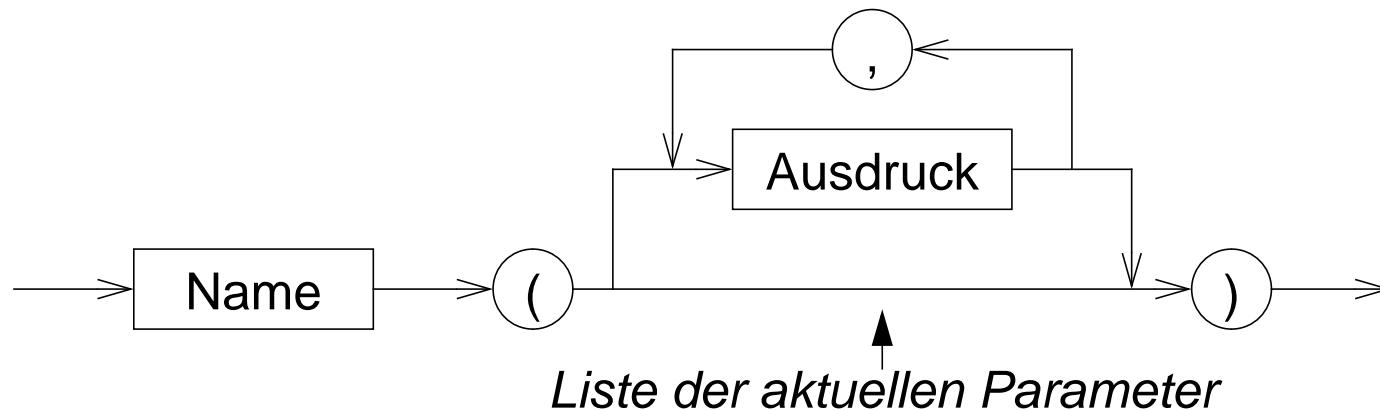
- Rückkehr aus der Funktion: das Programm wird nach dem Funktionsaufruf fortgesetzt
- der Typ des Ausdrucks muss mit dem Typ der Funktion übereinstimmen
- die Klammern können auch weggelassen werden

## 4 Funktionsaufruf

- Aufruf einer Funktion aus dem Ablauf einer anderen Funktion

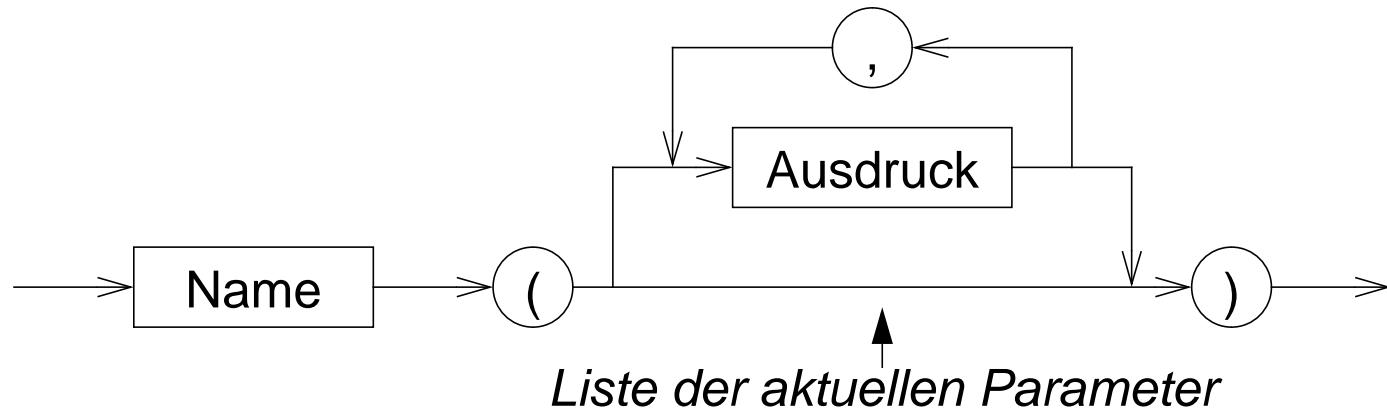
- ◆ Beispiel:

```
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
  - = Rückgabetyp 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);
```

# 7 Funktionsdeklarationen — Beispiel

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

double sinus(double);
/* oder: double sinus(double x); */

int main()
{
    double wert;

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

```
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);
}
```

## 8 Parameterübergabe an Funktionen

- allgemein in Programmiersprachen vor allem zwei Varianten:
  - call by value
  - call by reference

### call by value

- Normalfall in C
- Es wird eine Kopie des aktuellen Parameters an die Funktion übergeben
  - die Funktion kann den Übergabeparameter durch Zugriff auf den formalen Parameter lesen
  - die Funktion kann den Wert des formalen Parameters (also die Kopie!) ändern, ohne dass dies Auswirkungen auf den Wert des aktuellen Parameters beim Aufrufer hat
  - die Funktion kann über einen Parameter dem Aufrufer keine Ergebnisse mitteilen

## 8 Parameterübergabe an Funktionen (2)

### call by reference

- In C nur indirekt mit Hilfe von Zeigern realisierbar
- Der Übergabeparameter ist eine Variable und die aufgerufene Funktion erhält die Speicheradresse dieser Variablen
  - ➔ die Funktion kann den Übergabeparameter durch Zugriff auf den formalen Parameter lesen
  - ➔ wenn die Funktion den Wert des formalen Parameters verändert, ändert sie den Inhalt der Speicherzelle des aktuellen Parameters
  - ➔ auch der Wert der Variablen (aktueller Parameter) beim Aufrufer der Funktion ändert sich dadurch

# D.9 Programmstruktur & Module

## 1 Softwaredesign

- Grundsätzliche Überlegungen über die Struktur eines Programms **vor** Beginn der Programmierung
- Verschiedene Design-Methoden
  - ◆ Top-down Entwurf / Prozedurale Programmierung
    - traditionelle Methode
    - bis Mitte der 80er Jahre fast ausschließlich verwendet
    - an Programmiersprachen wie Fortran, Cobol, Pascal oder C orientiert
  - ◆ Objekt-orientierter Entwurf
    - moderne, sehr aktuelle Methode
    - Ziel: Bewältigung sehr komplexer Probleme
    - auf Programmiersprachen wie C++, Smalltalk oder Java ausgerichtet

## 2 Top-down Entwurf

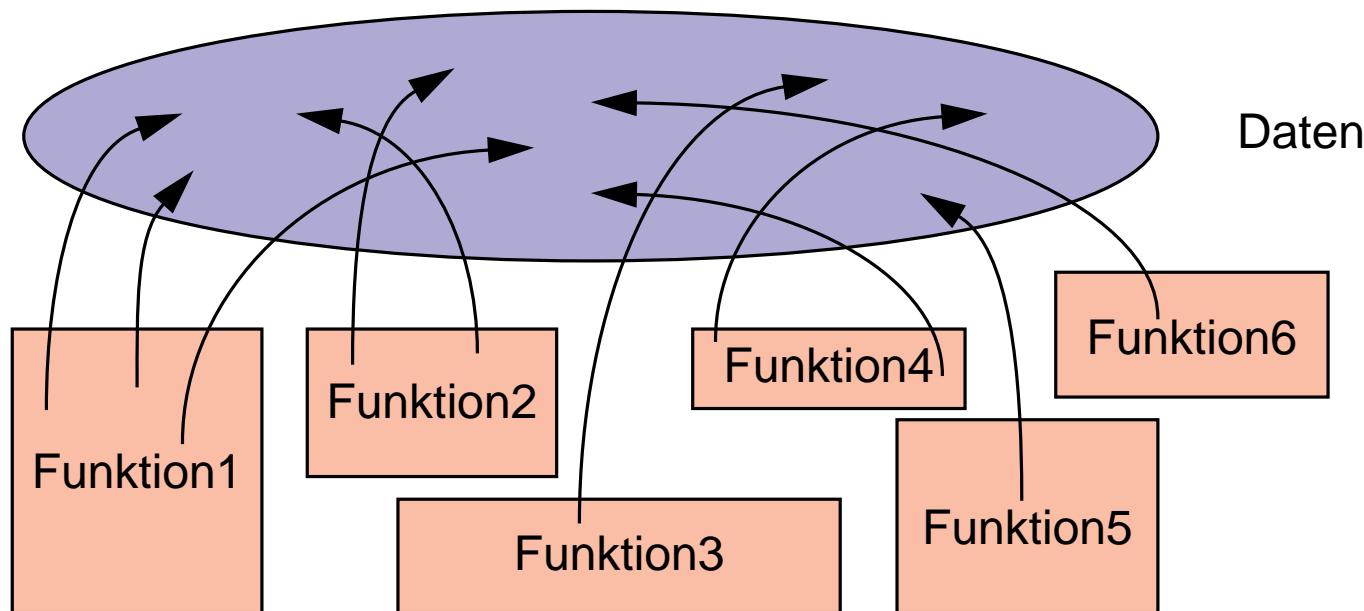
### ■ Zentrale Fragestellung

- ◆ was ist zu tun?
- ◆ in welche Teilaufgaben lässt sich die Aufgabe untergliedern?

- Beispiel:      Rechnung für Kunden ausgeben
  - Rechnungspositionen zusammenstellen
    - Lieferungspositionen einlesen
    - Preis für Produkt ermitteln
    - Mehrwertsteuer ermitteln
  - Rechnungspositionen addieren
  - Positionen formatiert ausdrucken

## 2 Top-down Entwurf (2)

- Problem:  
Gliederung betrifft nur die Aktivitäten, nicht die Struktur der Daten
- Gefahr:  
Sehr viele Funktionen arbeiten "wild" auf einer Unmenge schlecht strukturierter Daten



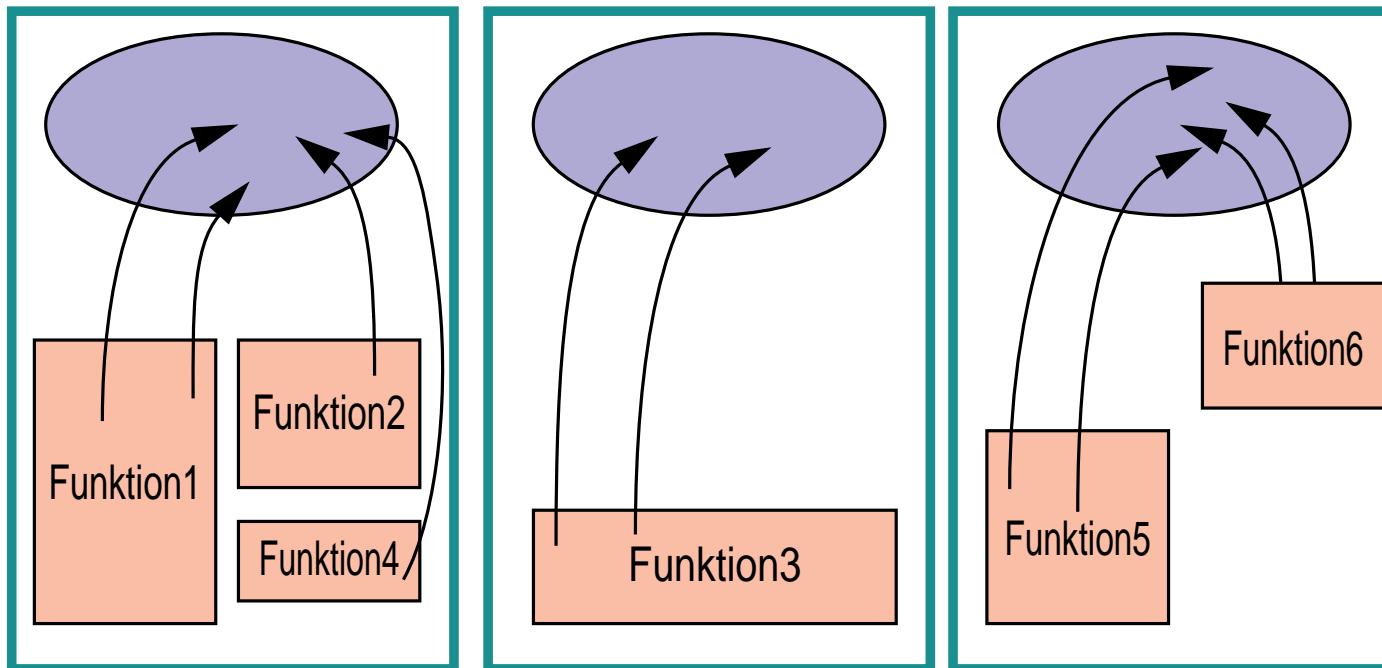
## 2 Top-down Entwurf (3)

### Modul-Bildung

#### ■ Lösung:

Gliederung von Datenbeständen zusammen mit Funktionen, die darauf operieren

→ **Modul**



### 3 Module in C

- Teile eines C-Programms können auf mehrere .c-Dateien (C-Quelldateien) verteilt werden
- Logisch zusammengehörende Daten und die darauf operierenden Funktionen sollten jeweils zusammengefasst werden
  - Modul
- Jede C-Quelldatei kann separat übersetzt werden (Option -c)
  - Zwischenergebnis der Übersetzung wird in einer .o-Datei abgelegt

<pre>% cc -c main.c</pre> <pre>% cc -c f1.c</pre> <pre>% cc -c f2.c f3.c</pre>	<p>(erzeugt Datei main.o )</p> <p>(erzeugt Datei f1.o )</p> <p>(erzeugt f2.o und f3.o )</p>
--	---

- Das Kommando cc kann mehrere .c-Dateien übersetzen und das Ergebnis — zusammen mit .o-Dateien — binden:

```
% cc -o prog main.o f1.o f2.o f3.o f4.c f5.c
```

### 3 Module in C

- !!! .c-Quelldateien auf keinen Fall mit Hilfe der `#include`-Anweisung in andere Quelldateien einkopieren**
- Bevor eine Funktion aus einem anderen Modul aufgerufen werden kann, muss sie **deklariert** werden
  - Parameter und Rückgabewerte müssen bekannt gemacht werden
- Makrodefinitionen und Deklarationen, die in mehreren Quelldateien eines Programms benötigt werden, werden zu **Header-Dateien** zusammengefasst
  - ◆ Header-Dateien werden mit der `#include`-Anweisung des Präprozessors in C-Quelldateien einkopiert
  - ◆ der Name einer Header-Datei endet immer auf `.h`

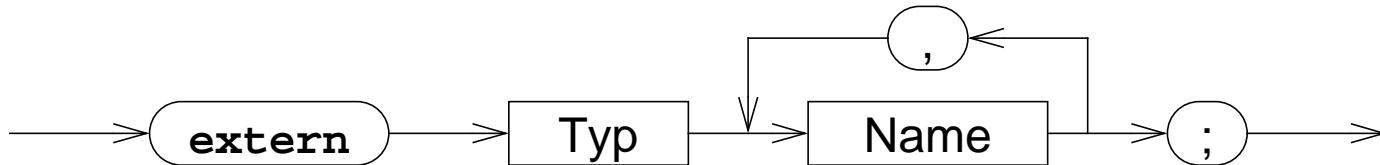
## 4 Gültigkeit von Namen

- Gültigkeitsregeln legen fest, welche Namen (Variablen und Funktionen) wo im Programm bekannt sind
- Mehrere Stufen
  1. Global im gesamten Programm  
(über Modul- und Funktionsgrenzen hinweg)
  2. Global in einem Modul  
(auch über Funktionsgrenzen hinweg)
  3. Lokal innerhalb einer Funktion
  4. Lokal innerhalb eines Blocks
- Überdeckung bei Namensgleichheit
  - eine lokale Variable innerhalb einer Funktion überdeckt gleichnamige globale Variablen
  - eine lokale Variable innerhalb eines Blocks überdeckt gleichnamige globale Variablen und gleichnamige lokale Variablen in umgebenden Blöcken

## 5 Globale Variablen

Gültig im gesamten Programm

- Globale Variablen werden außerhalb von Funktionen definiert
- Globale Variablen sind ab der Definition in der gesamten Datei zugreifbar
- Globale Variablen, die in anderen Modulen **definiert** wurden, müssen vor dem ersten Zugriff bekanntgemacht werden  
( **extern**–**Deklaration** = Typ und Name bekanntmachen)



- Beispiele:

```
extern int a, b;
extern char c;
```

## 5 Globale Variablen (2)

### ■ Probleme mit globalen Variablen

- ◆ Zusammenhang zwischen Daten und darauf operierendem Programmcode geht verloren
- ◆ Funktionen können Variablen ändern, ohne dass der Aufrufer dies erwartet (Seiteneffekte)
- ◆ Programme sind schwer zu pflegen, weil bei Änderungen der Variablen erst alle Programmteile, die sie nutzen gesucht werden müssen

→ **globale Variablen möglichst vermeiden!!!**

## 5 Globale Funktionen

- Funktionen sind generell global  
(es sei denn, die Erreichbarkeit wird explizit auf das Modul begrenzt)
- Funktionen aus anderen Modulen müssen ebenfalls vor dem ersten Aufruf **deklariert** werden  
(= Typ, Name und Parametertypen bekanntmachen)
- Das Schlüsselwort **extern** ist bei einer Funktionsdeklaration nicht notwendig
- Beispiele:

```
double sinus(double);  
float power(float, int);
```

- Globale Funktionen (und soweit vorhanden die globalen Daten) bilden die äußere Schnittstelle eines Moduls
  - "vertragliche" Zusicherung an den Benutzer des Moduls

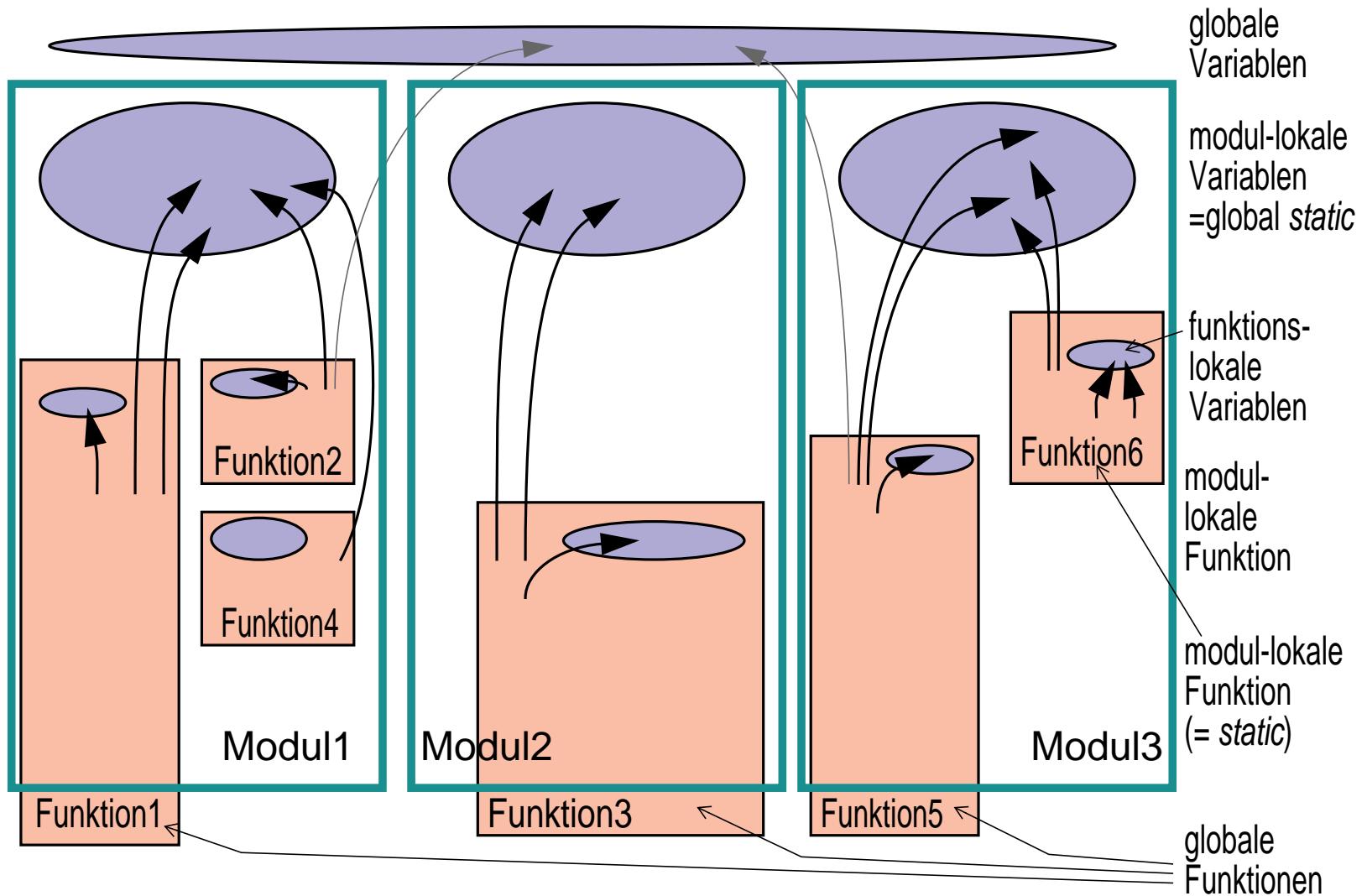
## 6 Einschränkung der Gültigkeit auf ein Modul

- Zugriff auf eine globale Variable oder Funktion kann auf das Modul (= die Datei) beschränkt werden, in der sie definiert wurde
  - Schlüsselwort **static** vor die Definition setzen
  - **extern**-Deklarationen in anderen Modulen sind nicht möglich
- Die **static**-Variablen bilden zusammen den Zustand eines Moduls, die Funktionen des Moduls operieren auf diesem Zustand
- Hilfsfunktionen innerhalb eines Moduls, die nur von den Modulfunktionen benötigt werden, sollten immer **static** definiert werden
  - sie werden dadurch nicht Bestandteil der Modulschnittstelle (= des "Vertrags" mit den Modulbenutzern)
- !!! das Schlüsselwort **static** gibt es auch bei lokalen Variablen (mit anderer Bedeutung! - zur Unterscheidung ist das hier beschriebene **static** immer kursiv geschrieben)

## 7 Lokale Variablen

- Variablen, die innerhalb einer Funktion oder eines Blocks definiert werden, sind lokale Variablen
- bei Namensgleichheit zu globalen Variablen oder lokalen Variablen eines umgebenden Blocks gilt die jeweils letzte Definition
- lokale Variablen sind außerhalb des Blocks, in dem sie definiert wurden, nicht zugreifbar und haben dort keinen Einfluss auf die Zugreifbarkeit von Variablen

# 8 Gültigkeitsbereiche — Übersicht



## 9 Lebensdauer von Variablen

- Die Lebensdauer einer Variablen bestimmt, wie lange der Speicherplatz für die Variable aufgehoben wird
- Zwei Arten
  - ◆ Speicherplatz bleibt für die gesamte Programmausführungszeit reserviert
    - statische (**static**) Variablen
  - ◆ Speicherplatz wird bei Betreten eines Blocks reserviert und danach wieder freigegeben
    - dynamische (**automatic**) Variablen

## 9 Lebensdauer von Variablen (2)

### auto-Variablen

- Alle lokalen Variablen sind automatic-Variablen
  - der Speicher wird bei Betreten des Blocks / der Funktion reserviert und bei Verlassen wieder freigegeben
    - ➔ der Wert einer lokalen Variablen ist beim nächsten Betreten des Blocks nicht mehr sicher verfügbar!
- Lokale auto-Variablen können durch beliebige Ausdrücke initialisiert werden
  - die Initialisierung wird bei jedem Eintritt in den Block wiederholt
  - !!! wird eine auto-Variable nicht initialisiert, ist ihr Wert vor der ersten Zuweisung undefiniert (= irgendwas)**

## 9 Lebensdauer von Variablen (2)

### static-Variablen

- Der Speicher für alle globalen Variablen ist generell von Programmstart bis Programmende reserviert
- Lokale Variablen erhalten bei Definition mit dem Schlüsselwort **static** eine **Lebensdauer über die gesamte Programmausführung** hinweg
  - ➔ der Inhalt bleibt bei Verlassen des Blocks erhalten und ist bei einem erneuten Eintreten in den Block noch verfügbar
- **!!!** Das Schlüsselwort **static** hat bei globalen Variablen eine völlig andere Bedeutung (Einschränkung des Zugriffs auf das Modul)
- Static-Variablen können durch beliebige konstante Ausdrücke initialisiert werden
  - die Initialisierung wird nur einmal beim Programmstart vorgenommen (auch bei lokalen Variablen!)
  - erfolgt keine explizite Initialisierung, wird automatisch mit 0 vorbelegt

# 10 Getrennte Übersetzung von Programmteilen

## — Beispiel

### ■ Hauptprogramm (Datei `fplot.c`)

```
#include "trig.h"
#define INTERVAL 0.01

/*
 * Funktionswerte ausgeben
 */
int main(void)
{
    char c;
    double i;

    printf("Funktion (Sin, Cos, Tan, cOt)? ");
    scanf("%x", &c);

    switch (c) {
    ...
    case 'T':
        for (i=-PI/2; i < PI/2; i+=INTERVAL)
            printf("%lf %lf\n", i, tan(i));
        break;
    ...
}
}
```

# 10 Getrennte Übersetzung — Beispiel (2)

## ■ Header-Datei (Datei `trig.h`)

```
#include <stdio.h>
#define PI 3.1415926535897932
double tan(double), cot(double);
double cos(double), sin(double);
```

## ■ Trigonometrische Funktionen (Datei `trigfunc.c`)

```
#include "trig.h"

double tan(double x) {
    return(sin(x)/cos(x));
}

double cot(double x) {
    return(cos(x)/sin(x));
}

double cos(double x) {
    return(sin(PI/2-x));
}
```

## 10 Getrennte Übersetzung — Beispiel (3)

- Trigonometrische Funktionen — Fortsetzung  
(Datei `trigfunc.c`)

...

```
double sin (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);
}
```