

## 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 Teilfunktionen
  - 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

## 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:

```
% cc hello.c           (C-Compiler)
oder
% gcc hello.c          (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 direkt benachbarter Worte erforderlich
- **Kommentar** wird durch `/*` und `*/` geklammert  
keine Schachtelung möglich
- **Identifizier** (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 *Identifizier* verwendet werden
  - **Identifizier** müssen vor ihrer ersten Verwendung **deklariert** werden
- Anweisungen werden generell durch `;` abgeschlossen

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

```
#define o define
#o __o write
#o ooo (unsigned)
#o o_o_ 1
#o _o_ char
#o _oo goto
#o _oo_ read
#o o_o for
#o o_ main
#o o_ if
#o oo_ 0
#o _o(_,_)(void)___o(_,_ooo(_))
#o __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_];_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.

## 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 */
    ...
}
```

## D.3 Datentypen

- |   |   |  |
|---|---|--|
| <ul style="list-style-type: none"> <li>■ Datentypen           <ul style="list-style-type: none"> <li>➤ Konstanten</li> <li>➤ Variablen</li> </ul> </li> </ul> |  | <ul style="list-style-type: none"> <li>◆ Ganze Zahlen</li> <li>◆ Fließkommazahlen</li> <li>◆ Zeichen</li> <li>◆ Zeichenketten</li> </ul> |
|---|---|--|

## 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

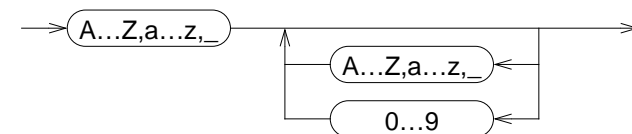
- Eine Reihe häufig benötigter Datentypen ist in C vordefiniert
- char** Zeichen (im ASCII-Code dargestellt, 8 Bit)
- int** ganze Zahl (16 oder 32 Bit)
- float** Gleitkommazahl (32 Bit)  
etwa auf 6 Stellen genau
- double** doppelt genaue Gleitkommazahl (64 Bit)  
etwa auf 12 Stellen genau
- void** ohne Wert

## 2 Standardtypen in C (2)

- Die Bedeutung der Basistypen kann durch vorangestellte **Typ-Modifizier** 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

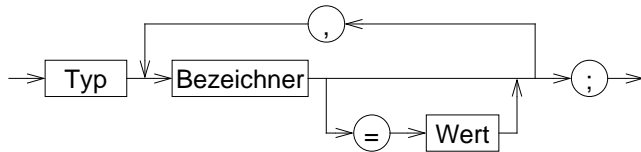
- Variablen haben:
  - ◆ **Namen** (Bezeichner)
  - ◆ **Typ**
  - ◆ zugeordneten Speicherbereich für einen Wert des Typs  
Inhalt des Speichers (= **aktueller Wert** der Variablen) ist veränderbar!
  - ◆ **Lebensdauer**  
wann wird der Speicherplatz angelegt und wann freigegeben
- Bezeichner



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

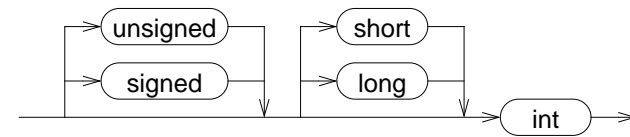
### 3 Variablen (2)

- 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



### 4 Ganze Zahlen

- Definition



- Speicherbedarf(short int) ≤ Speicherbedarf(int) ≤ Speicherbedarf(long int)
- Speicherbedarf(int): meist 32 Bit
- Konstanten (Beispiele):

42, -117  
035 (oktal = 29<sub>10</sub>)  
0x10 (hexadezimal = 16<sub>10</sub>)  
0x1d (hexadezimal = 29<sub>10</sub>)

### 3 Variablen (3)

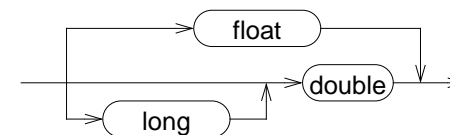
- Variablen-Definition: Beispiele

```
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

### 5 Fließkommazahlen

- Definition



- Speicherbedarf(float) ≤ Speicherbedarf(double) ≤ 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 · 10<sup>2</sup>, 0.003 = 3.0 · 10<sup>-3</sup>)
    - 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: '\''

## 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";**

## 6 Zeichen (2)

### American Standard Code for Information Interchange (ASCII)

NUL 00	SOH 01	STX 02	ETX 03	EOT 04	ENQ 05	ACK 06	BEL 07
BS 08	HT 09	NL 0A	VT 0B	NP 0C	CR 0D	SO 0E	SI 0F
DLE 10	DC1 11	DC2 12	DC3 13	DC4 14	NAK 15	SYN 16	ETB 17
CAN 18	EM 19	SUB 1A	ESC 1B	FS 1C	GS 1D	RS 1E	US 1F
SP 20	!	"	#	\$	%	&	'
(	)	*	+	,	-	.	/
0	1	2	3	4	5	6	7
8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G
H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W
X	Y	Z	[	\	]	^	_
`	a	b	c	d	e	f	g
h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w
x	y	z	{		}	~	DEL 7F

## 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

## 1 Zuweisungsoperator =

→ Zuweisung eines Werts an eine Variable

■ Beispiel:

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

## 3 spezielle Zuweisungsoperatoren

→ Verkürzte Schreibweise für Operationen auf einer Variablen

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

■ Beispiele:

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

## 2 Arithmetische Operatoren

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

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

■ Beispiel:

```
a = -5 + 7 * 20 - 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 ≠ 0: wahr
- Ergebnis eines Ausdrucks: falsch: 0  
wahr: 1

## 6 Bitweise logische Operatoren

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

"nicht"

~

"und"

&

"oder"

|

Antivalenz  
"exklusives oder"

^	f	w
f	f	w
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

## 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
```

## 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

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

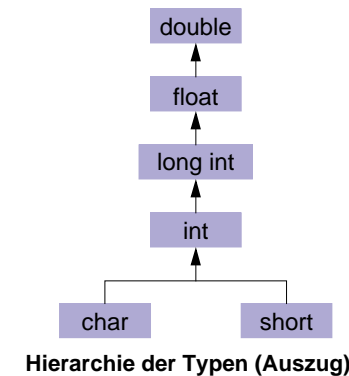
## 9 Komma-Operator

,

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





## 11 Vorrangregeln bei Operatoren

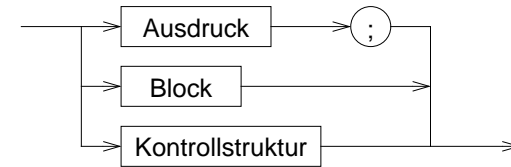
Operatorklasse	Operatoren	Assoziativität
unär	! ~ ++ -- + -	von rechts nach links
multiplikativ	* / %	von links nach rechts
additiv	+ -	von links nach rechts
shift	<< >>	von links nach rechts
relational	< <= > >=	von links nach rechts
Gleichheit	== !=	von links nach rechts
bitweise	&	von links nach rechts
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

## 1 Struktur eines C-Hauptprogramms

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

## 2 Anweisungen

Anweisung:



## D.6 Einfacher Programmaufbau

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

### 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
```

## 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);
    }
}
```

## 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**

## 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][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!)

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 einfache Verzweigung

Bedingung	
ja	nein
Anweisung_1	Anweisung_2

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

## 1 Bedingte Anweisung

Bedingung	
ja	nein
Anweisung	

```
if ( Bedingung )
    Anweisung
```

### ■ Beispiel:

Dampftemperatur > 450 Grad	
ja	nein
Ausgabe: 'Dampftemperatur gefährlich hoch!'	

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

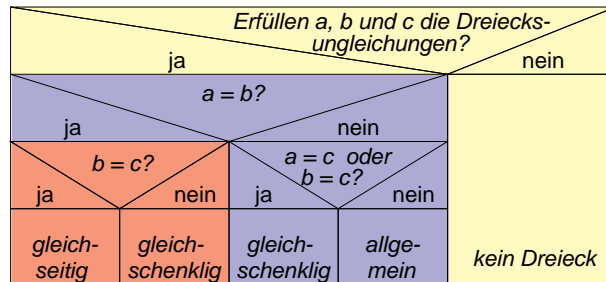
## 1 Bedingte Anweisung mehrfache Verzweigung

Bedingung_1		
ja	nein	
Anweisung_1	Bedingung_2	
	ja	nein
	Anweisung_2	Anweisung_3

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

## 1 Bedingte Anweisung mehrfache Verzweigung (2)

- Beispiel: Eigenschaften von Dreiecken — Struktogramm



## 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
}
```

## 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 — Beispiel

```
#include <stdio.h>

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

    ziffern = leer = sonstige = 0;

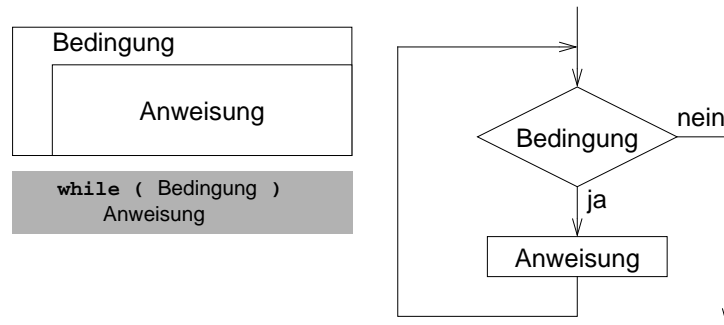
    while ((zeichen = getchar()) != EOF)
    {
        switch (zeichen) {
            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
          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

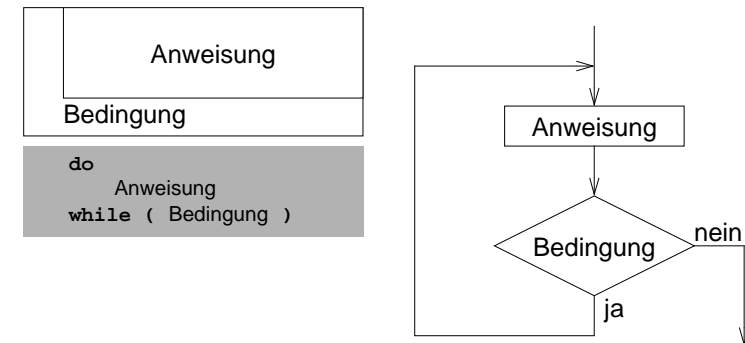
### 3 Schleifen

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

#### 4 abweisende Schleife

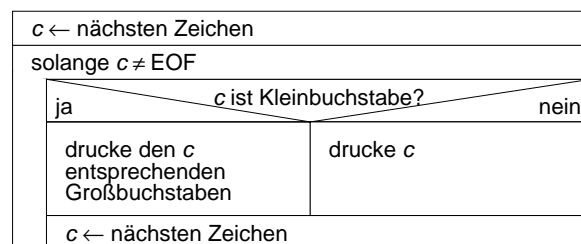


### 5 nicht-abweisende Schleife



#### 4 abweisende Schleife (2)

- Beispiel: Umwandlung von Klein- in Großbuchstaben

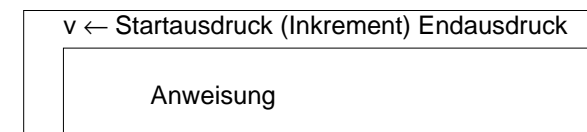


```
int c;
c = getchar();
while ( c != EOF ) {
    if ( c >= 'a' && c <= 'z' )
        putchar(c+'A'-'a');
    else
        putchar(c);
    c = getchar();
}
```

► abgekürzte Schreibweise

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

### 6 Laufanweisung



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

allgemein:

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

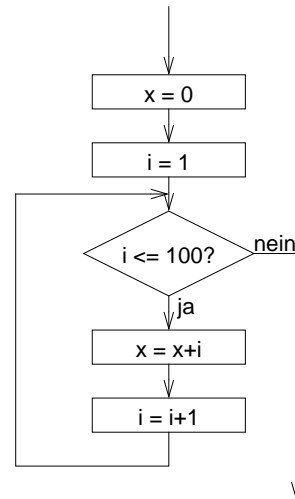
```
Ausdruck_1;
while ( Ausdruck_2 ) {
    Anweisung
    Ausdruck_3;
}
```

## 6 Laufanweisung (2)

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

```
x ← 0
i ← 1(1)100
x ← x + 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);
}

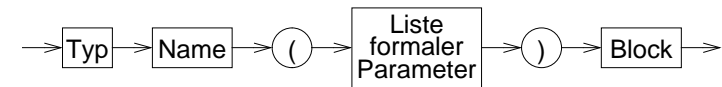
int main()
{
    double wert;

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

- beliebige Verwendung von `sinus` in Ausdrücken:

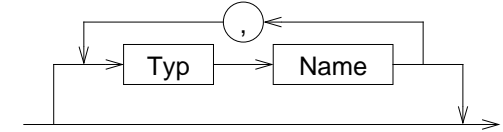
```
y = exp(tau*t) * sinus(f*t);
```

## 3 Funktionsdefinition (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

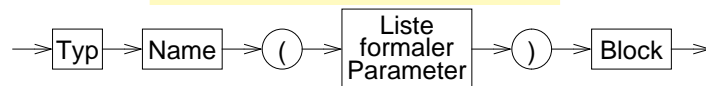


## 3 Funktionsdefinition

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

- ◆ Beispiel:

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



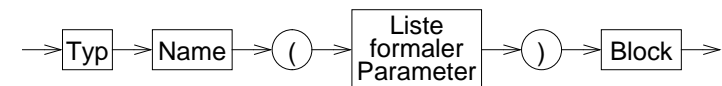
### Typ

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

### Name

- ◆ beliebiger Bezeichner, kein Schlüsselwort

## 3 Funktionsdefinition (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

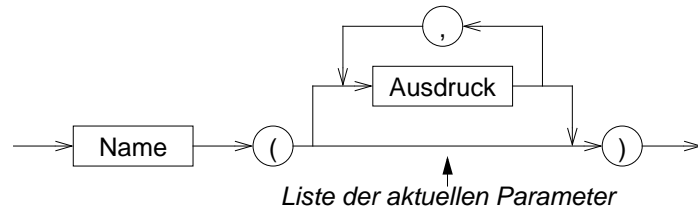


## 4 Funktionsaufruf

- 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)

## 5 Beispiel

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

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

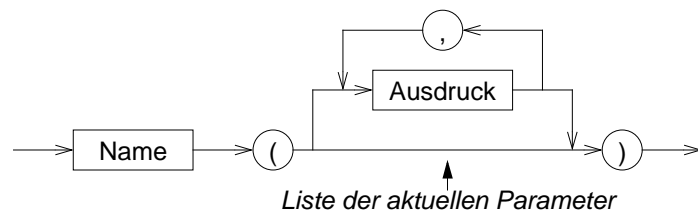
```
float x, y;

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;
```

## 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

## 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

## 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);
}
```

D.69

## 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);
```

D.70

## 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);
}
```

D.71

## 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

D.72

## 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

## 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
      - Lieferungsposten einlesen
      - Preis für Produkt ermitteln
      - Mehrwertsteuer ermitteln
      - Rechnungspositionen addieren
      - Positionen formatiert ausdrucken

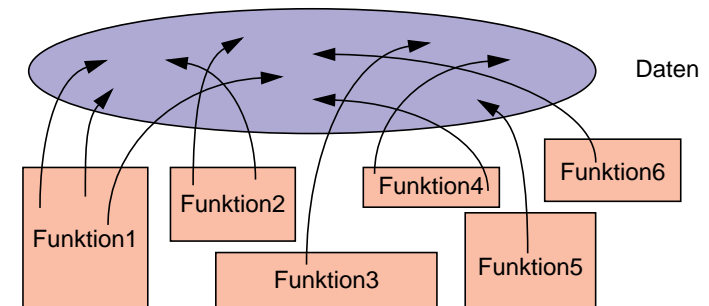
## D.9 Programmstruktur & Module

### 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 (2)

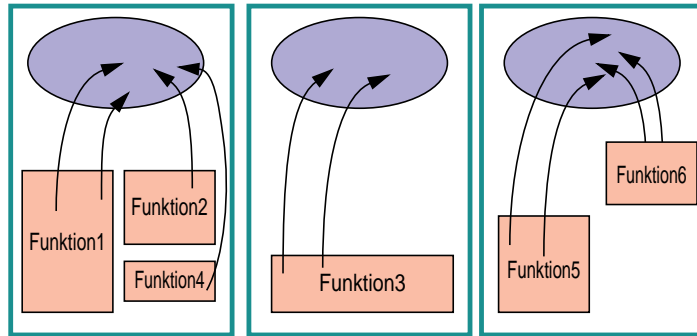
- Problem:  
Gliederung betrifft nur die Aktivitäten, nicht die Struktur der Daten
- Gefahr:  
Sehr viele Funktionen arbeiten "wild" auf einer Unmenge schlecht strukturierter Daten



## 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

```
% cc -c main.c           (erzeugt Datei main.o)
% cc -c f1.c              (erzeugt Datei f1.o)
% cc -c f2.c f3.c         (erzeugt f2.o und f3.o)
```

- 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

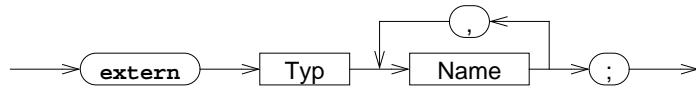
## 4 Gültigkeit von Namen

- 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 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

## 5 Globale Variablen (2)

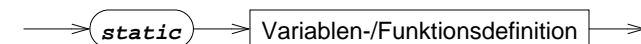
- Probleme mit globalen Variablen

- ◆ 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!!!**

## 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)

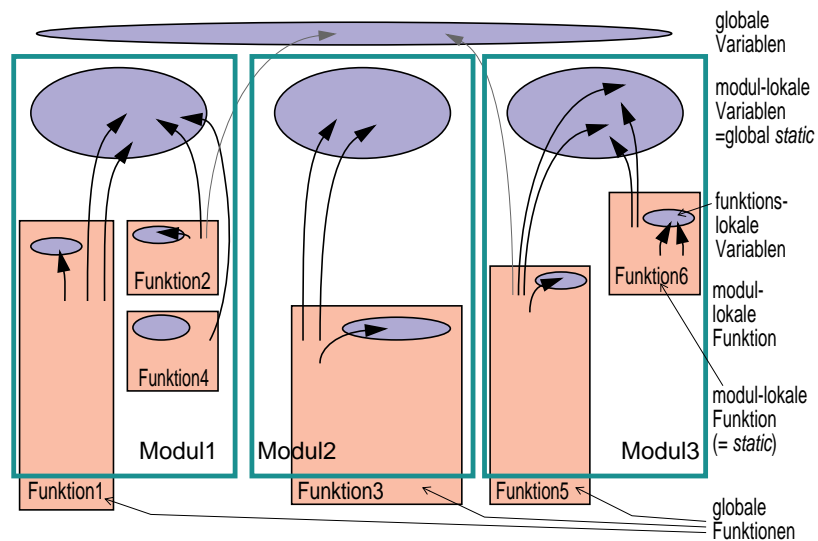
## 7 Lokale Variablen

- 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

## 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

## 8 Gültigkeitsbereiche — Übersicht



## 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 — 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 von Programmteilen — Beispiel

### ■ Hauptprogramm (Datei fplot.c)

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

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

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

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

## 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);
}
```

