Fachhochschule Jena
University of Applied Sciences Jena

Informatik I

Kapitel 2: Die Programmiersprache C

Burkart Voss

Was benötige ich zum Programmieren mit C?

- Texteditor
  - Mit ihm wird der erstellt
  - Empfehlung: Editor, der die von C farbig hervorhebt.

- Compiler
  - Macht aus einer Quelldatei eine (Maschinencodedatei)
  - Linker (zum Einfügen von schon kompiliertem Code) meist mit integriert

Oder:

- Entwicklungsumgebung
  - Alles, was zum Programmieren benötigt wird, in einem Fenster:
    - Editor
    - Compiler
    - Linker
    - Projektverwaltung
    - Debugger
    - Profiler
    - Versionskontrolle

Hallo Welt


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

- Kein direkter Bestandteil von C – Befehl für _____
  - Teil des Compliers, der nicht bleibende Änderungen im Programmtext vornimmt
  - Markiert durch #-Zeichen
- In `stdio.h` ist die im folgenden benutzte Funktion `printf()` deklariert.
  - Include-Dateien heißen auch Header-Dateien
  - Irgendwo im Compiler-Verzeichnis unter \include
- stdio steht für Standard-Input/Output

```
int main(void)
```

- Beginn des _____ immer in Funktion `main`
- `void`: leerer Datentyp
- `int`: Ganze Zahl, Integer
  - Funktion hat einen Rückgabewert vom Typ Integer.
  - In Beispiel: Rückgabewert `0` durch

```
return 0;
```

→ Programm wurde ordnungsgemäß beendet.

Hallo Welt


```
{
   ...
}
```

- Zwischen geschweiften Klammern {} steht ein _____
  - Hier befinden sich alle Anweisungen, die die Funtion `int main(void)` auszuführen hat.

Merke: Geschweifte Klammern fassen Anweisungen zu einem Block zusammen.

```
printf(„Hallo Welt\n“);
```

- Funktion, die in `stdio.h` deklariert ist.
  - Formatierte Ausgabe einer beliebigen Stringkonstante
  - Stringkonstante
    - steht immer zwischen Hochkommata (`„Stringkonstante“`)
    - Darf nicht über das Zeilenende fortgesetzt werden:
      ```
      printf(„Dies ist in C
      nicht erlaubt“);
      ```
    - Für lange Stringkonstanten:
      ```
      printf(„Hier ist die Ausnahme der Regel \
      dies hier ist erlaubt, dank Backslash“);
      ```
- `\n` ist ein Steuerzeichen und bedeutet newline (Zeilenvorschub)
- Semikolon (`;`) zeigt das Ende einer Anweisung an.

Merke: Anweisungen werden mit einem _____ abgeschlossen.

- **Datentypen: Arten von Variablen, in denen Daten gespeichert werden können.**
- **Variable besteht aus 4 Teilen:**
  - **, der den Speicherbedarf bestimmt**
  - **, mit dem Variable im Programm angesprochen werden kann**
  - **, unter der die Variable abgelegt wird**
  - **der Variablen, der sich ändern kann.**

- ---
- **Mindestens Größe von 2 Byte (16 bit)**
  - → $2^0+2^1+2^2+2^3+2^4+2^5+2^6+2^7+2^8+2^9+2^{10}+2^{11}+2^{12}+2^{13}+2^{14}+2^{15}$
  - → 1 +2 +4 +8 +16+32+64+128+256+512+1024+2048+4096+8192+16384+32768=65535
  - → **Positive und negative Zahlen**
    - → **Zahlenraum von –32.768 bis +32.767**
- **Auf 32-bit Systemen (Linux, Window95 und höher) 32bit für Integer**
  - → **Zahlenraum von –2.147.483.648 bis +2.147.483.647**
- **Wertebereich im System auf das compiliert wird steht in der Headerdatei `limits.h` in `INT_MIN` und `INT_MAX`**

Variablen deklarieren

- Vor Verwendung müssen Variablen deklariert (bekanntgemacht) werden:

```
typ name;
```

- typ: wie viel Speicherplatz muss für Variable auf System reserviert werden
- name: Bezeichner,
  - der innerhalb eines Anweisungsblocks eindeutig sein muss
  - selbsterklärend sein sollte.

```
int main(void) {
   int a;   ←
   int b;
   int c;
   a=5;     ←
   b=100;
   c=12345;
   ...
   return 0;
}
```

- **Erst, wenn bei einer Variablen-Deklaration eine Initialisierung vorgenommen wird, handelt es sich um eine Definition**

```
int wert = 5;

int wert1 = 10, wert2 = 20;


int wert1, wert2=33; ←


int wert1;

int wert2=wert1=10;
```

- **Wenn eine Variable nicht mit einem Wert initialisiert wurde, ist der Wert undefiniert → nicht vorhersehbar, welcher Wert benutzt werden wird.**

Erlaubte Bezeichner

- Ein Bezeichner darf aus einer Folge von Buchstaben, Dezimalziffern und Unterstrichen bestehen:
  - `var8, _var, _666, var_fuer_100tmp, VAR, Var`
- C unterscheidet zwischen Groß- und Kleinbuchstaben:
  - `Var, VAr, VAR, vAR, vaR, var`
    Hierbei handelt es sich jeweils um _____ Bezeichner.
- Das erste Zeichen darf keine Dezimalzahl sein.
- Die Länge des Bezeichners ist beliebig lang. Nach ANSI C-Standard sind aber nur die ersten 31 Zeichen bedeutend.

Datentypen long und short

- **Long _____ 4 byte**
- **Short _____ 2 byte**

Name	Größe	Wertebereich	Formatzeichen
Short	2 byte	-32768 +32767	%hd oder %hi
Integer	Mindestens 2 byte	-32768 +32767	%d oder %i
Long	Mindestens 4 byte	-2147483648 + 2147483647	%ld oder %li

Kommentare


```
/* kommentare.c */
#include <stdio.h>

int main (void) {       //Beginn des Hauptprogramms
   int i = 10;          //Variable int mit dem Namen i und Wert
   10
   printf("%d",i);      //gibt die Zahl 10 aus
   printf("\n");        //springt eine Zeile weiter
   printf("10");        //gibt den String "10" aus
   return 0;

/*  Hier sehen Sie noch eine 2. Möglichkeit, Kommentare
    einzufügen. Dieser Kommentar wird mit einem Slash-
    Sternchen eröffnet und mit einem Sternchen-Slash
    wieder beendet. Alles dazwischen wird vom Compiler
    ignoriert  */
}
```

Formatierte Ausgabe → printf()

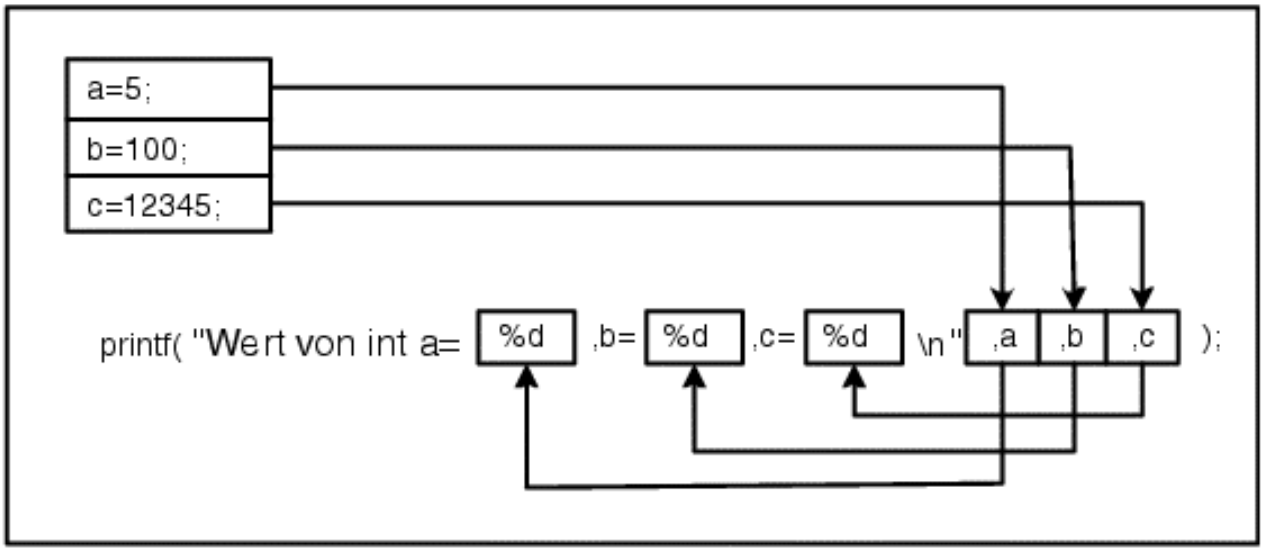
Funktionsaufruf	Formatstring	Formatanweisung	Variablen Liste
printf(	“Der Wert lautet	%d“,	wert);

Formatanweisungen beginnen mit ____ gefolgt von einem Buchstaben, der Datentyp des Werts angibt, der in der Variablen steht – muss nicht der Datentyp der Variablen sein!

- **Z.B. d steht für dezimale Ganzzahl.**

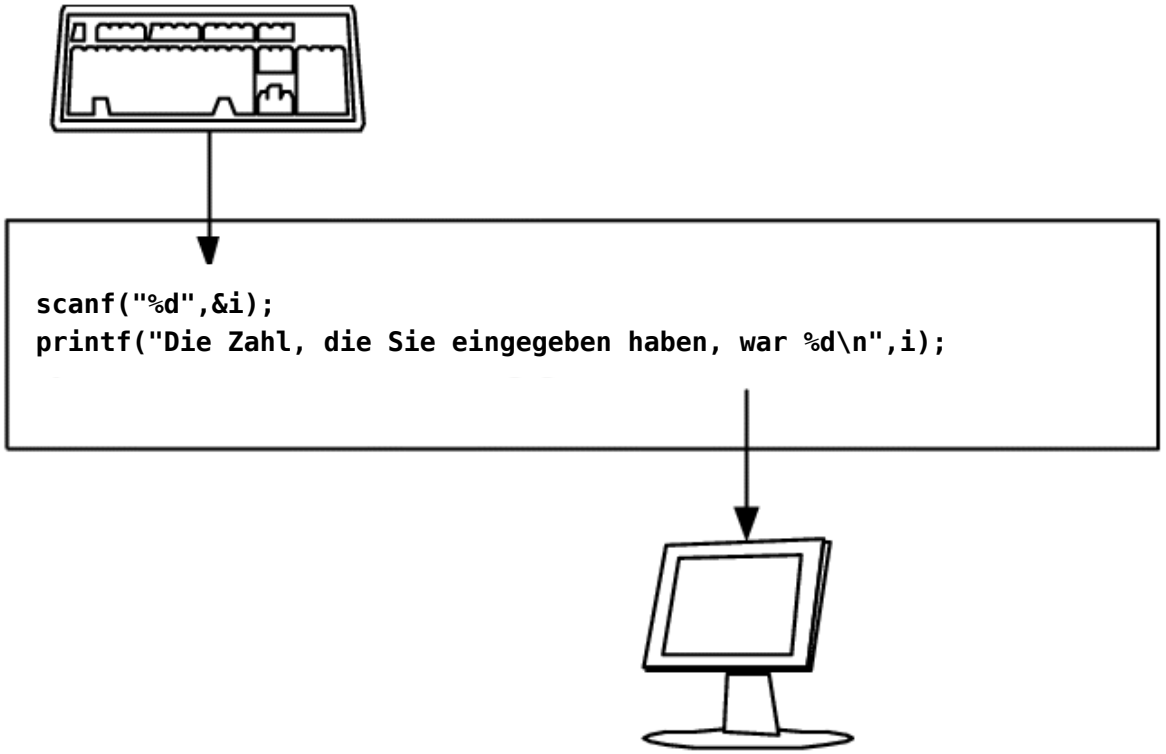
printf - Beispiel


```
#include <stdio.h>  //hier ist printf()
                    // deklariert
int main(void) {
   int a, b, c;     // Deklaration
   a=5;             // Initialisieren
   b=100;
   c=12345;
   printf("Wert von int a=%d ,b=%d ,c=%d\n",a,b,c);
   return 0;
}
```

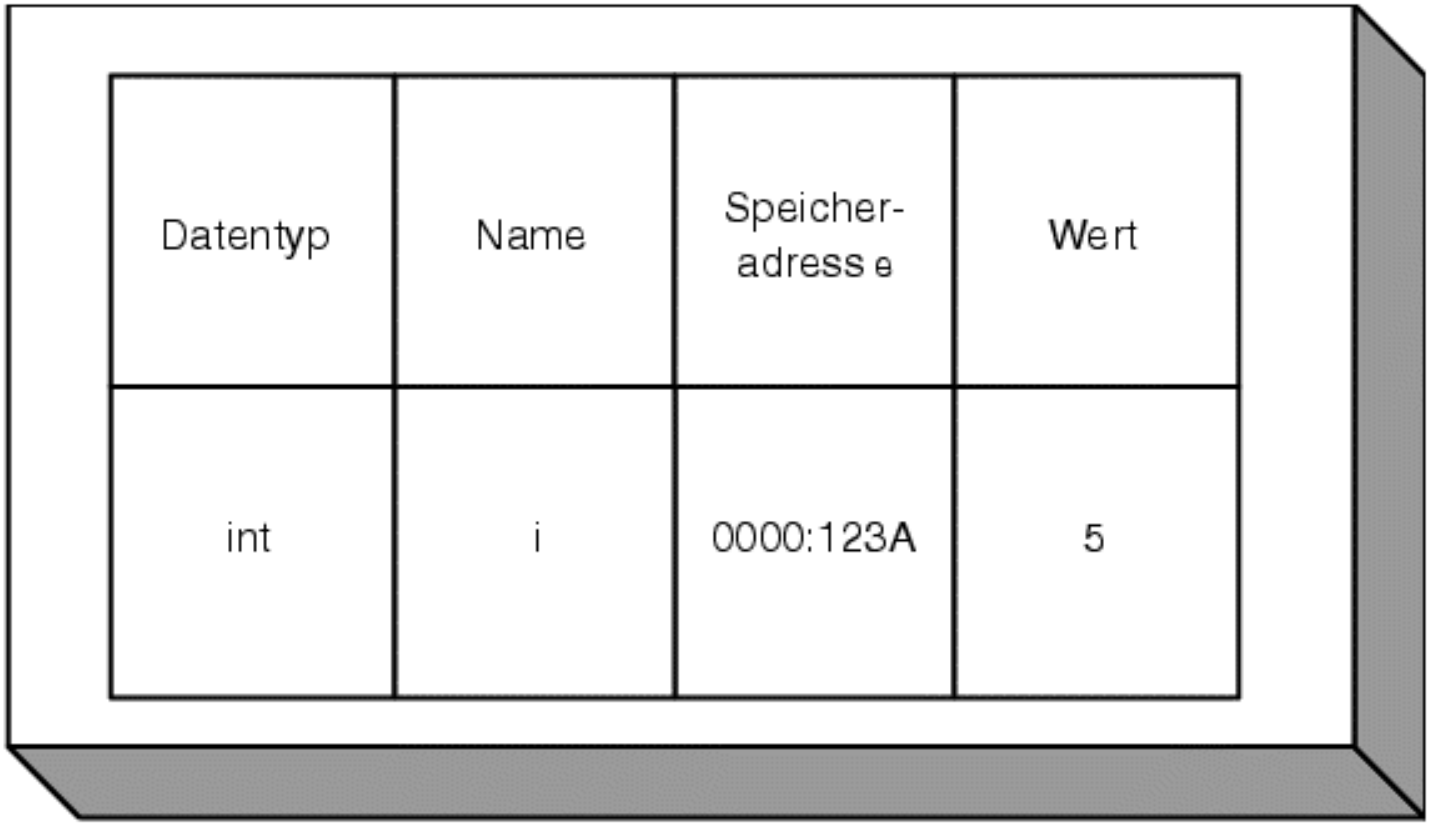


Formatierte Eingabe → scanf()


```
#include <stdio.h>
int main ()
{
   int i;                  /* Ein ganzzahliger Datentyp */
   printf("Bitte geben Sie eine Zahl ein : ");
   scanf("%d",&i);         /* Wartet auf die Eingabe */
   printf("Die Zahl, die Sie eingegeben haben, war %d\n",i);
   return 0;
}
```



Adressoperator &



Datentyp	Name	Speicher-adresse	Wert
int	i	0000:123A	5

Der Variablen i vom Typ int mit der Speicheradresse 000:123A wird der Wert 5 zugewiesen.

Scanf() braucht die Adresse, um dort die Daten abzulegen!

Eingabe überprüfen


```
/* scanf.c */
#include <stdio.h>
int main() {
   int a, b,check;
   printf("Bitte Eingabe machen (2 Zahlen): ");
   check=scanf("%d %d",&a,&b);
   printf("Check = %d \n",check);
   return 0;
}
```

- `scanf()` gibt die Anzahl der erfolgreich gelesenen Werte zurück.

- **Unterscheidung hinsichtlich der Zahl der Operanden**
  - _____ – der Operator hat einen Operanden
  - _____ – der Operator hat zwei Operanden
  - Ternär – der Operator hat drei Operanden
- **Unterscheidung hinsichtlich Position**
  - _____ – der Operator steht zwischen den Operanden
  - Präfix – der Operator steht vor den Operanden
  - Postfix – der Operator steht hinter den Operanden
- **Unterscheidung hinsichtlich Auswertungsreihenfolge (Assoziativität)**
  - _____ – Auswertung von links nach rechts
  - Rechtsassoziativität – Auswertung von rechts nach links

Arithmetische Operatoren

Operator	Bedeutung
+	Addiert zwei Werte
-	Subtrahiert zwei Werte
*	Multipliziert zwei Werte
/	Dividiert zwei Werte
%	Modulo (Rest einer Division)

Regeln:

- **Punkt-vor-Strich-Regel: * und / binden stärker als + und –**

  ```
  5 + 5 * 5 = 5 + (5 * 5)
  ```

- **Arithmetische Operatoren sind _____ und werden in _____-Schreibweise verwendet, also:**

  ```
  <Operand><Operator><Operand>
  ```

Beispiel


```
/* arithmetik.c */
#include <stdio.h>

int main(void) {
   int zahl1,zahl2,zahl3;
   int ergeb;

   zahl1=10;
   zahl2=20;
   zahl3=30;

   printf("Zahl 1= %d\n",zahl1);
   printf("Zahl 2= %d\n",zahl2);
   printf("Zahl 3= %d\n",zahl3);

   /* Möglichkeit 1: zuerst
    Berechnung, dann Ausgabe*/
   ergeb=zahl1+zahl2+zahl3;
   printf("Summe aller \
    Zahlen:%d\n",ergeb);
   /* Möglichkeit 2: wie oben, nur
    mit Ausgabe in einem Schritt */
   ergeb=zahl3-zahl2;
   printf("%d - %d = \
    %d\n",zahl3,zahl2,ergeb);
   /* Möglichkeit 3: mit Anzeige
    * und Berechnung am Ende der
    * 'printf'-Anweisung         */
   printf("%d * %d = \
   %d\n",zahl1,zahl1,zahl1*zahl1);

   // Möglichkeit 4: weitere
   //'printf'-Berechnung
   printf("Zahl3 / Zahl1=%d\n",zahl3/zahl1);

   // Möglichkeit 5: wieder eine mit
   // 'printf'
   printf("Zahl 1 + x-Beliebige Zahl \
   =%d\n",zahl1+11);

   // Ein Klammerbeispiel
   ergeb=(zahl1+zahl2)/zahl3;
   printf("(%d + %d)/%d = \
   %d\n",zahl1,zahl2,zahl3,ergeb);
   return 0;
}
```

Dividieren von Ganzzahlen


```
/* zeit.c */
#include <stdio.h>

int main(void) {
   int sekunden,minuten;

   printf("Bitte geben Sie die Zeit in Sekunden ein :");
   scanf("%d",&sekunden);
   minuten=sekunden/60;
   sekunden=sekunden%60;
   printf("genauer = %d min. %d sek.\n", minuten, sekunden);
   return 0;
}
```

Erweiterte Darstellung	Bedeutung
+=	a+=b ist gleichwertig zu a=a+b
-=	a-=b ist gleichwertig zu a=a-b
*=	a*=b ist gleichwertig zu a=a*b
/=	a/=b ist gleichwertig zu a=a/b
%=	a%=b ist gleichwertig zu a=a%b

Bsp: die folgenden Schreibweisen sind _____:

```
printf("Die Fläche beträgt : %d\n",x*=x);
printf("Die Fläche beträgt : %d\n",x=x*x);
```

Inkrement- und Dekrement-Operatoren

- **++ : Increment (Variable um eins erhöhen)**
- **-- : Decrement (Variable um eins verringern)**

Postfix: var++
→ Wert wird ausgegeben, dann wird var inkrementiert.
Präfix: ++var
→ Wert wird inkrementiert und danach ausgegeben.

```
#include <stdio.h>
int main(void) {
   int i=1;
   printf("i=%d\n",i);          // i=1
   i++;
   printf("i=%d\n",i);          // i=2
   printf("i=%d\n",i++);        // i=2
   printf("i=%d\n",i);          // i=3
   printf("i=%d\n",++i);        // i=4
   return 0;
}
```

Datentyp char Character

- **Zur Darstellung von einzelnen Zeichen (wie `'a'`, `'A'`, `'b'`, `'B'`, `'5'`, `'7'`, `'§'` usw.) für Tastatureingabe und Bildschirmausgabe**
- **Für kleine Ganzzahlen**
- **_____ ist der kleinste darstellbare elementare Datentyp in C.**

Name	Größe	Wertebereich	Formatzeichen
char	1 byte	-128 ... +127 oder 0 ... 255	%c
short	2 byte	-32.768 +32.767	%hd oder %hi
integer	Mindestens 2 byte	-32.768 +32.767	%d oder %i
long	Mindestens 4 byte	-2.147.483.648 + 2.147.483.647	%ld oder %li

Datentyp char

1 Byte							
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
128	64	32	16	8	4	2	1
0	1	0	0	0	0	1	0

ASCII – Tabelle: 66 = ‘B‘

```
char a = 'A';
/*falsch, in doppelte Hochkommata
== string*/
char a = "A";
/*falsch, Variablenzuweisung*/
char a = A;
/* Schlechter Stil, da nicht
gleich durchschaubar, ob der
Programmierer hier den ASCII-
Buchstaben oder den dezimalen Wert
verwenden will */
char b = 65;
```

ASCII - Zeichentabelle

0	NUL	32		64	@	96	`	128	Ç	160	á	192	└	224	α
1	☺	33	!	65	A	97	a	129	ü	161	í	193	┴	225	β
2	☻	34	“	66	B	98	b	130	é	162	ñ	194	┬	226	Γ
3	♥	35	#	67	C	99	c	131	â	163	ú	195	├	227	π
4	♦	36	$	68	D	100	d	132	ä	164	ñ	196	–	228	Σ
5	♣	37	%	69	E	101	e	133	à	165	Ñ	197	┼	229	σ
6	♠	38	&	70	F	102	f	134	å	166	ª	198	╞	230	µ
7	•	39	‘	71	G	103	g	135	ç	167	º	199	╟	231	τ
8	◘	40	(	72	H	104	h	136	ê	168	¿	200	╚	232	Φ
9	○	41	)	73	I	105	i	137	ë	169	⌐	201	╔	233	Θ
10	◙	42	*	74	J	106	j	138	è	170	¬	202	╩	234	Ω
11	♂	43	+	75	K	107	k	139	ï	171	½	203	╦	235	δ
12	♀	44	,	76	L	108	l	140	î	172	¼	204	╠	236	∞
13	♪	45	-	77	M	109	m	141	ì	173	¡	205	═	237	ø
14	♫	46	.	78	O	110	n	142	Ä	174	«	206	╬	238	∈
15	☼	47	/	79	O	111	o	143	Å	175	»	207	╧	239	∩
16	►	48	0	80	P	112	p	144	É	176	░	208	╨	240	≡
17	◄	49	1	81	Q	113	q	145	æ	177	▒	209	╤	241	±
18	↕	50	2	82	R	114	r	146	Æ	178	▓	210	╥	242	≥
19	‼	51	3	83	S	115	s	147	ô	179	│	211	╙	243	≤
20	¶	52	4	84	T	116	t	148	ö	180	┤	212	╘	244	⌠
21	§	53	5	85	U	117	u	149	ò	181	╡	213	╒	245	⌡
22	▬	54	6	86	V	118	v	150	û	182	╢	214	╓	246	÷
23	↨	55	7	87	W	119	w	151	ù	183	╖	215	╫	247	≈
24	↑	56	8	88	X	120	x	152	ÿ	184	╕	216	╪	248	°
25	↓	57	9	89	Y	121	y	153	Ö	185	╣	217	┘	249	·
26	→	58	:	90	Z	122	z	154	Ü	186	║	218	┌	250	·
27	←	59	;	91	[	123	{	155	ø	187	╗	219	█	251	√
28	∟	60	<	92	\	124	|	156	£	188	╝	220	▄	252	“
29	↔	61	0	93	]	125	}	157	Ø	189	╜	221	▌	253	²
30	▲	62	>	94	^	126	~	158	×	190	╛	222	▐	254	■
31	▼	63	?	95	_	127	|	159	ƒ	191	┐	223	▀	255	

char vs. integer

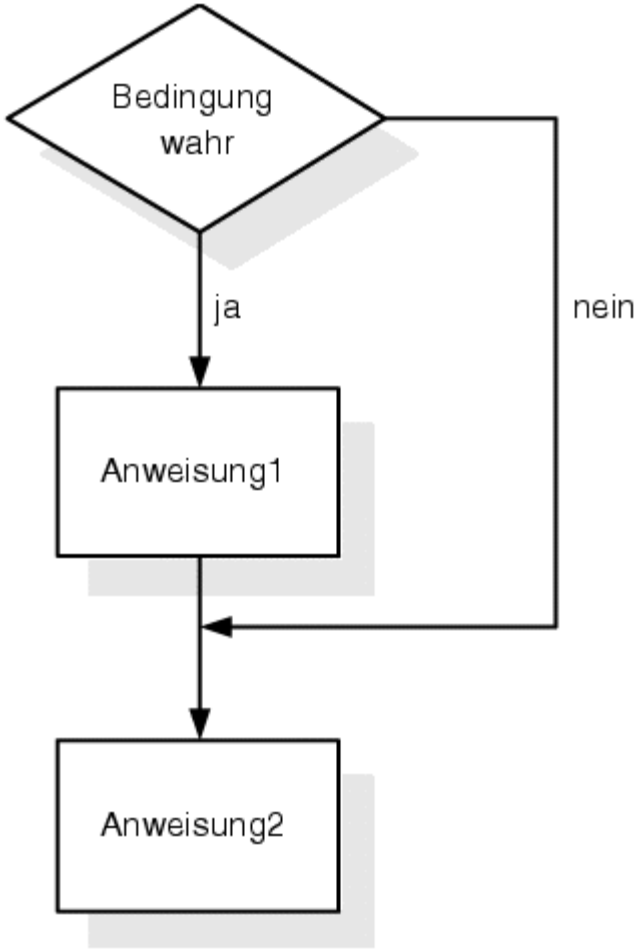

```
/* playing_char.c */
#include <stdio.h>

int main(void) {
   char a = 'A';
   char b = 65;
   int c  = 65;
   int d;

   printf("a = %c\n",a);
   printf("b = %c\n",b);     // ‚A‘
   printf("c = %c\n",c);     // ‚A‘

   d = a + b + c;            // Rechenbeispiel
   printf("d = %d\n",d);     //3*65=195

   d = 'a' + 'A';
   printf("d = %d\n",d);     //65+97=162

   printf("char a = %c und %d\n",a,a);     // char a = A und 65
   printf("char b = %c und %d\n",b,b);     // char b = A und 65
   printf("int c = %c und %d\n",c,c);      // int c = A und 65
   return 0;
}
```

- **Verzweigungen:**
  - **Im Programm wird eine _____ definiert, die entscheidet, an welcher Stelle das Programm fortgesetzt werden soll.**
- **Schleifen (Iteration):**
  - **Ein Anweisungsblock wird sooft _____, bis eine bestimmte Abbruchbedingung erfüllt wird.**
- **Sprünge:**
  - **Die Programmausführung wird mit Hilfe von Sprungmarken an einer anderen Position fortgesetzt.**
    Obwohl nach wie vor möglich, werden Sprünge in einem Programm mittlerweile als schlechter Stil angesehen und sind auch nicht notwendig. Es wird dabei von direkten Sprüngen gesprochen.
    Mit Schlüsselworten wie return, break, continue, exit und der Funktion abort() können jedoch kontrollierte Sprünge ausgeführt werden.

Verzweigungen mit der if-Bedingung

Syntax:

```
if(Bedingung == wahr) {
   Anweisung1;
   ...
}
Anweisung2;
```



Vergleichsoperatoren

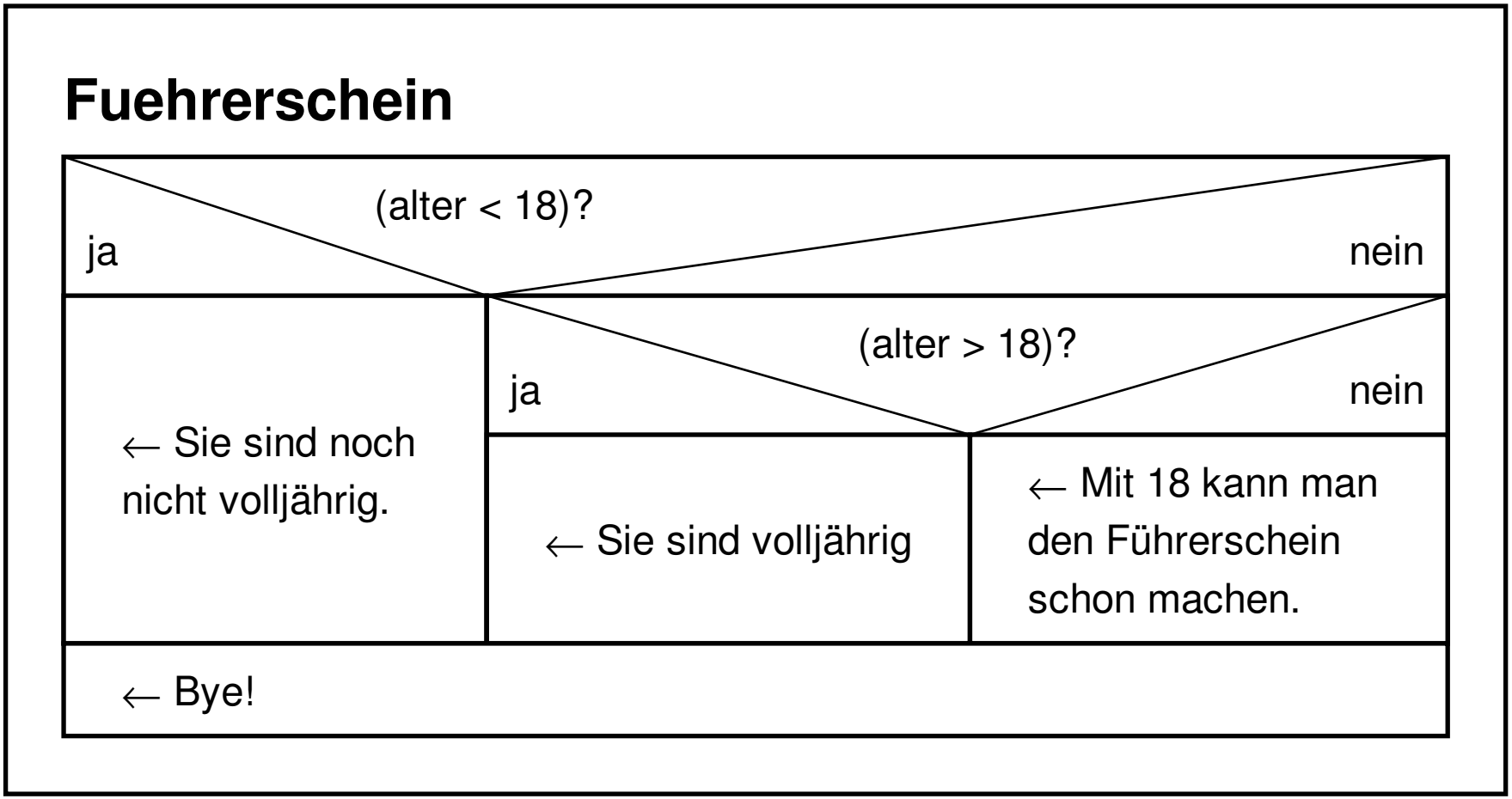
Vergleichsoperator	Bedeutung
a < b	Wahr, wenn a kleiner als b
a <= b	Wahr, wenn a kleiner oder gleich b
a > b	Wahr, wenn a größer als b
a >= b	Wahr, wenn a größer oder gleich b
a == b	Wahr, wenn a gleich b
a != b	Wahr, wenn a ungleich b

- **Vergleichsoperatoren und logische Operatoren liefern als Ergebnis ...**
  einfach einen Zahlenwert, welcher entsprechend interpretiert wird
  - **wahr // falsch**
  - **true // false**
  - **≠0 // 0 (interne Realisierung)**

- **Besonderheit:**
  - **Jeder Ausdruck kann als Wahrheitswert _____ werden**
  - **Daher: `if (3*x/12)` zulässig; In Abhängigkeit von x wird verzweigt**
  - **Insbesondere:`if ( x = 12 )` ebenfalls zulässig; Konsequenzen???**

if - Beispiel


```
/* if2.c */
#include <stdio.h>

int main(void) {
   unsigned int alter;

   printf("Wie alt sind Sie: ");
   scanf("%d", &alter);

   if(alter < 18)  {
      printf("Sie sind noch nicht volljährig\n");
   }
   if(alter > 18)  {
      printf("Sie sind volljährig\n");
   }
   if(alter == 18) {
      printf("Mit 18 kann man den Führerschein schon machen.\n");
   }
   printf("Bye\n");
   return 0;
}
```


Fuehrerschein
(alter < 18)?
ja
nein
(alter > 18)?
ja
nein
← Sie sind noch nicht volljährig.
← Sie sind volljährig
← Mit 18 kann man den Führerschein schon machen.
← Bye!

Verschachtelte Anweisungsblöcke


```
/* if5.c */
#include <stdio.h>

int main(void) {
   unsigned int alter;

   printf("Wie alt sind Sie: ");
   scanf("%d", &alter);

   if(alter <= 18) {
      if(alter == 18) {
         printf(" Mit 18 kann man den Führerschein schon machen. \n");
      }
      else {
         printf("Sie sind noch nicht volljährig\n");
      }
   }
   else {
      printf("Sie sind volljährig\n");
   }
   printf("Bye\n");
   return 0;
}
```

Häufiger Fehler:

```
if(alter=18)
```

Variablen `alter` wird Wert 18 zugewiesen → Abfrage immer wahr (> 1 entspricht wahr).

Compiler findet diesen Fehler nicht!!!

Abhilfe:

```
if(18==alter)
```

(`if(18=alter)` erzeugt Syntaxfehler!)

Semikolon hinter Bedingungsanweisung:

```
else if(alter > 18); //Fehler (das
                     //Semikolon)
{
   printf("Sie sind volljährig\n");
}
```

Der !-Operator (logischer Operator)

Unärer Operator, der Wert bzw. Bedingung _____:

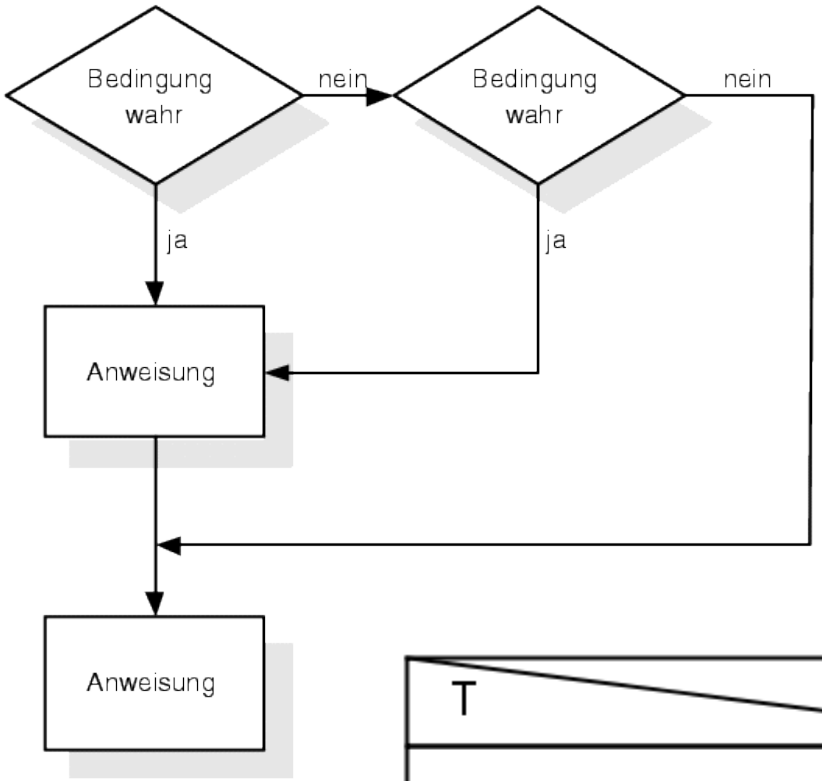
```
/* logic_not1.c */
#include <stdio.h>

int main(void) {
   int checknummer;

   printf("Bitte geben Sie Ihren Code-Schlüssel ein: ");
   scanf("%d", &checknummer);

   if( ! (checknummer == 4711) ) {
      printf("Error - Falscher Code-Schlüssel \n");
   }
   else {
      printf("Success -  Login erfolgreich \n");
    }
   return 0;
}
```

Anweisung	==	Anweisung
`if(a == 1)`	gleich	`if(a)`
`if(a == 0)`	gleich	`if(!a)`
`if(a > b)`	gleich	`if(! (a <= b) )`
`if( (a-b) == 0)`	gleich	`if(! (a-b) )`

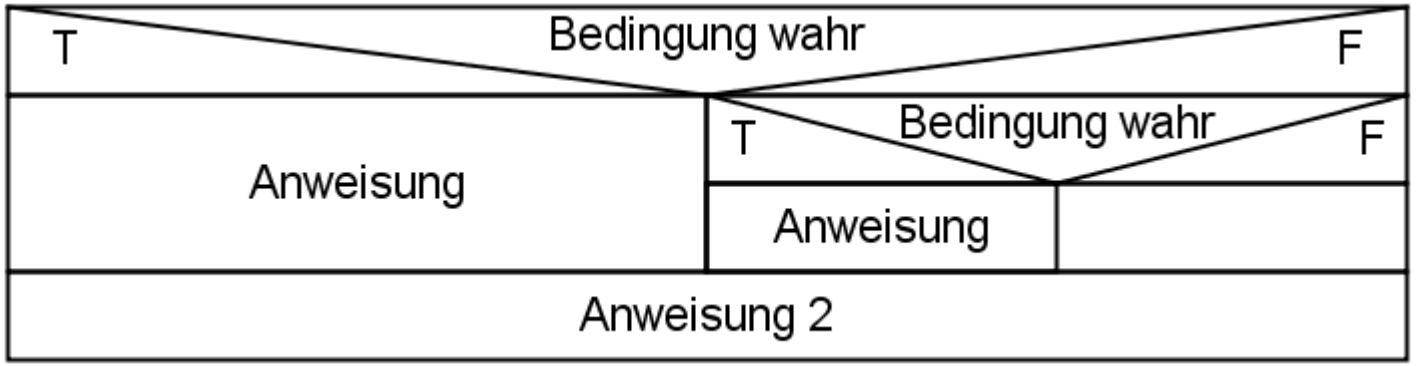
```
if(!zahl1)
        printf("Error: Der Wert ist gleich 0!! \n");
```

→So kann z.B. eine Division durch 0 vermieden werden!

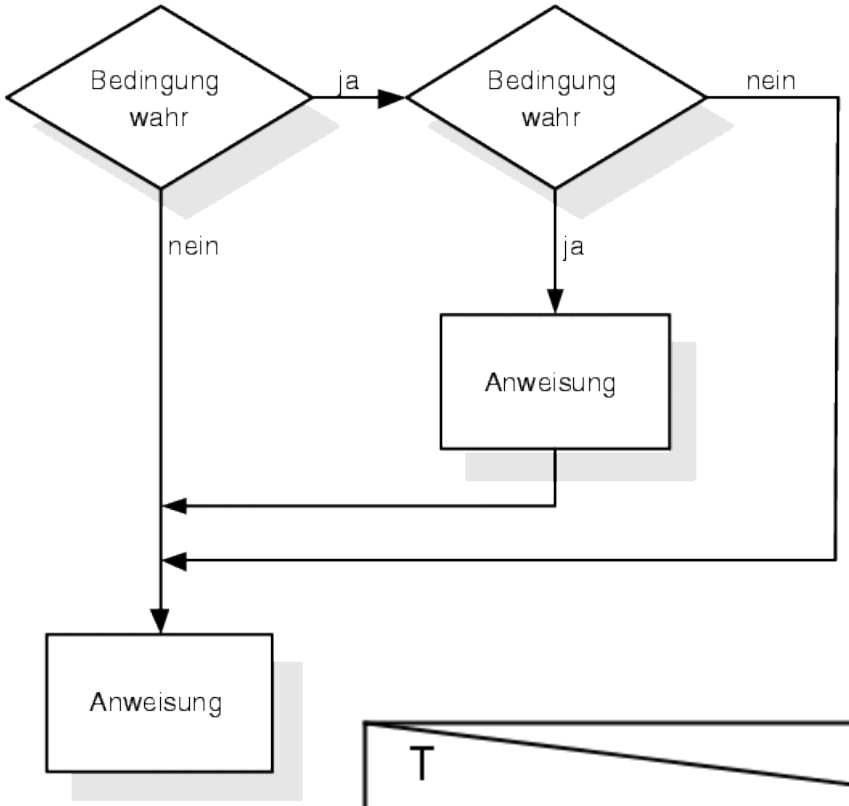
Logisches ODER (||)



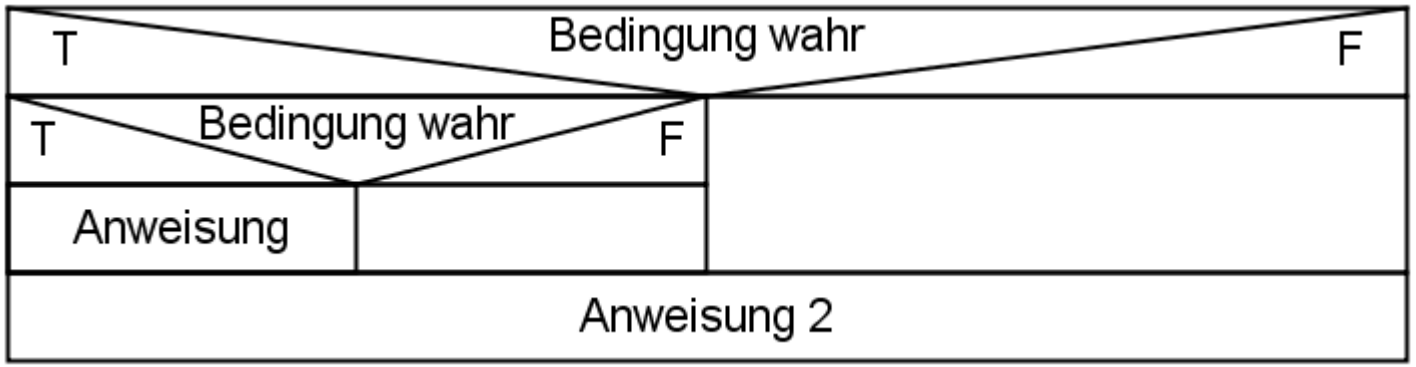

```
if( (Bedingung1) || (Bedingung2) )
   /* mindestens eine der
   Bedingungen ist wahr */
   Anweisung;
else
   /* keine Bedingung ist wahr */
```



Logisches UND (&&)




```
if( (Bedingung1) && (Bedingung2) )
   /* beide Bedingungen sind wahr */
   Anweisung;
else
   /* mindestens eine Bedingung ist
   falsch */
```



- Vergleiche haben geringere Auswertepriorität als arithmetische Operatoren
  - `if (i < lim-1) --> if (i < (lim-1)) /* wie erwartet! */`

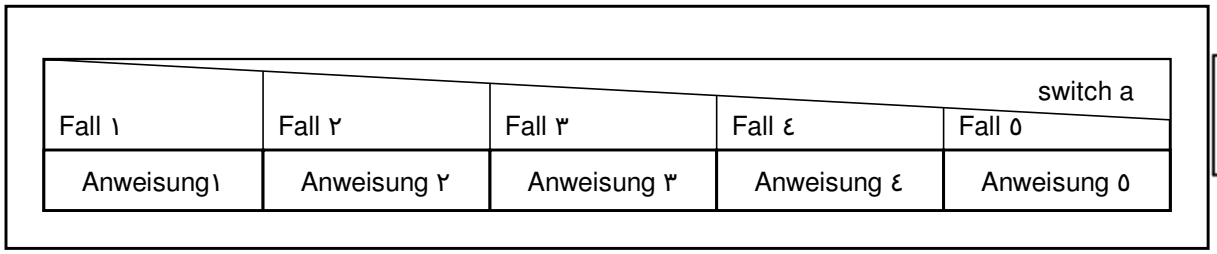
- Wertzuweisung hat niedrigere Priorität als Vergleichsoperatoren!
  - `if ( (c=getchar()) != 'n') /* Klammerung nötig! */`

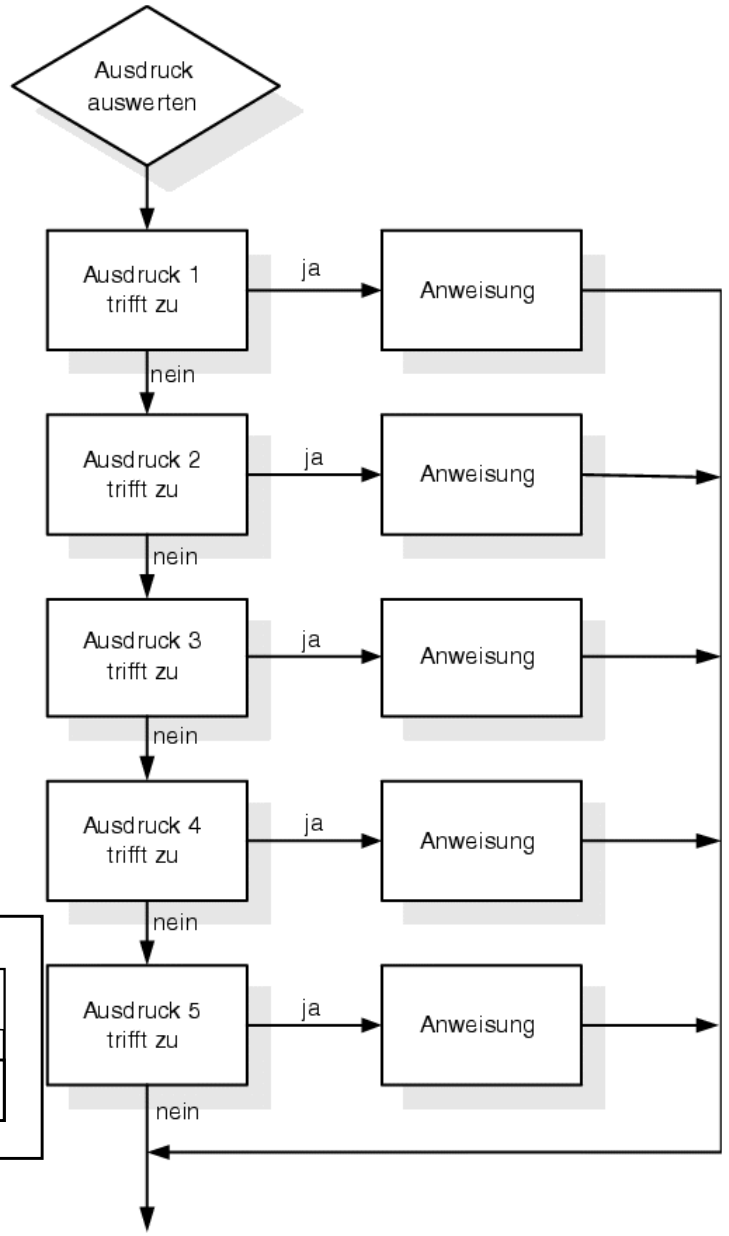
- ! hat höhere Priorität als die Vergleichsoperatoren
- die Priorität von && ist größer als die von ||
- beide haben geringere Priorität als die Vergleichsoperatoren!

Fallunterscheidung - switch

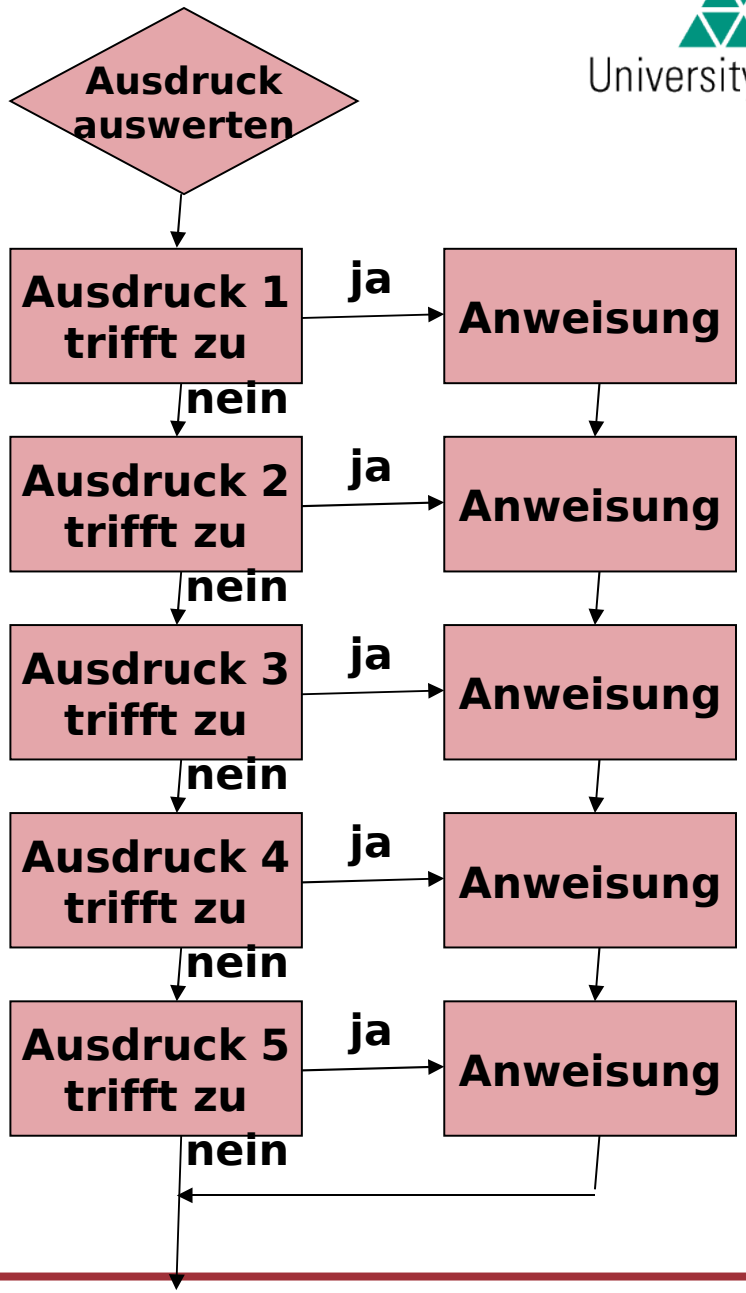
Bsp: Abfragen der Zahlen 1-5:

```
#include <stdio.h>
int main()
{
   int a;
   printf("Bitte eine Zahl von 1-5 eingeben: ");
   scanf("%d", &a);

   switch(a)
      {
         case 1: printf("Das war eins \n");
                 break;
         case 2: printf("Das war zwei \n");
                 break;
         case 3: printf("Das war drei \n");
                 break;
         case 4: printf("Das war vier \n");
                 break;
         case 5: printf("Das war fuenf \n");
                 break;
      }       /*Ende switch*/
   return 0;
}
```





Effekt von break;



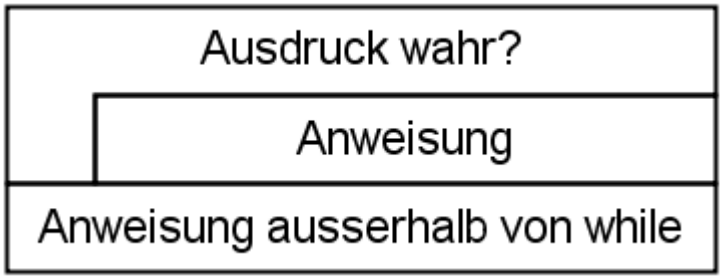
Default

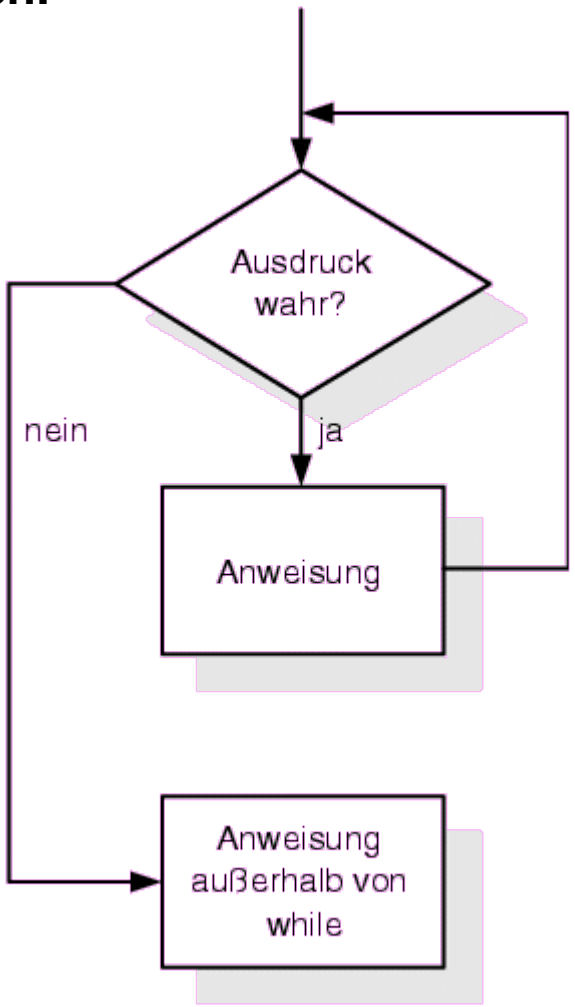

```
#include <stdio.h>


int main(void) {
   int a,b;
   char opera;
   printf("Grundrechnen \n");
   printf(" (zahl)(Operator)(zahl)\
            ohne Leerzeichen \n");


   printf("Rechnung bitte eingeben : ");
   scanf("%d%c%d", &a, &opera, &b);
 /* Bsp.: 10+12 */
```

```
switch(opera) {
     case '+': printf("%d + %d = %d \n",a ,b
,a+b);
               break;
     case '-': printf("%d - %d = %d \n", a, b, a-b);
               break;
     case '*': printf("%d * %d = %d \n", a, b, a*b);
               break;
     case '/': printf("%d / %d = %d \n", a, b, a/b);
               break;
     default: printf("%c kein Rechenoperator \n",
              opera);
     }      /* Ende switch */
 return 0;
}
```

Mehrfache Ausführung von Anweisungsblöcken.

```
while(Bedingung==wahr)
    {
     /*Abarbeiten von
    Befehlen bis Bedingung
    ungleich wahr*/
    }
```





Anwendung von while

3 Schritte:

- **Initialisierung – die Schleifenvariable bekommt einen Wert.**
- **Bedingung – die Schleifenvariable wird daraufhin überprüft, ob eine bestimmte Bedingung bereits erfüllt ist.**
- **Reinitialisieren – die Schleifenvariable erhält einen anderen Wert.**

```
int var=0;              /* Initialisieren */
while(var < 10)         /* Solange var kleiner als 10 –
                  Bedingung */
   {
      /* weitere Anweisungen */
      var++;            /* Reinitialisieren */
   }
```

Endlosschleife mit while


```
/* while2.c */
#include <stdio.h>

int main(void) {
   int zahl, summe=0;
   printf("Summenberechnung\nBeenden der Eingabe mit 0 \n");

   while(1) {    /* Endlosschleife, weil: 1 ist immer wahr */
      printf("Bitte Wert eingeben > ");
      scanf("%d", &zahl);
      if(zahl == 0)     /*  Haben wir 0 eingegeben ...?   */
         break;         /* ... dann raus aus der Schleife */
      else
         summe+=zahl;
   }
   printf("Die Summe aller Werte beträgt: %d\n", summe);
   return 0;
}
```

```
int x=0;
while(x < 10); /* Fehler durch Semikolon am Ende */
   {
      printf("Der Wert von x beträgt %d\n", x);
      x++;
   }
```

→ **Endlosschleife!!!**

Merke: niemals Schleifen mit _____ schließen!

```
int x=2;
while(x != 10)
    {
       printf("Der Wert von x beträgt %d \n", x);
       x*=x;
    }
```

→ **Endlosschleife!!!**

Besser:

```
while(x <= 10)
```

Merke: möglichst Schleifenabbruch mit _____ definieren!

Fehler mit while III


```
#include <stdio.h>

int main()
{
   int zahl1=0, zahl2=0;
   while((zahl1++ < 5) || (zahl2++ < 5) )
      printf("Wert von zahl1: %d zahl2: %d \n ", zahl1,
  zahl2);
   return 0;
}
```

→ Wie oft wird Schleife durchlaufen?

→ Wie ist der Endwert von zahl1 und zahl2?

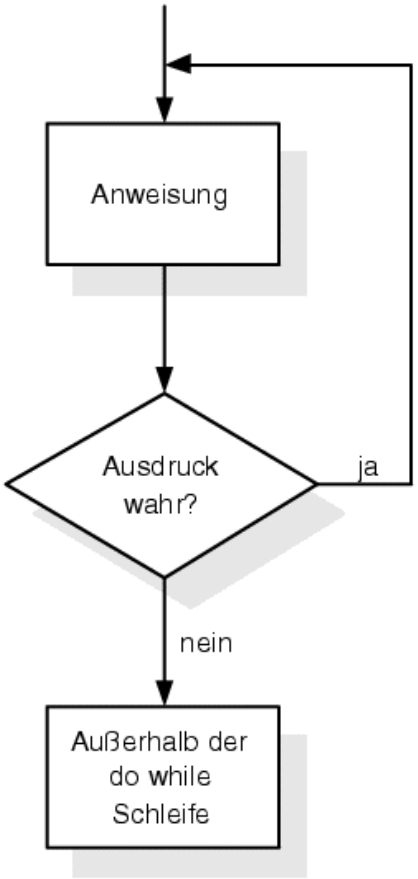
→ && oder || - Ausdrücke werden strikt von links nach rechts bewertet ... und zwar nur solange (!), bis das Resultat der logischen Verknüpfung feststeht!

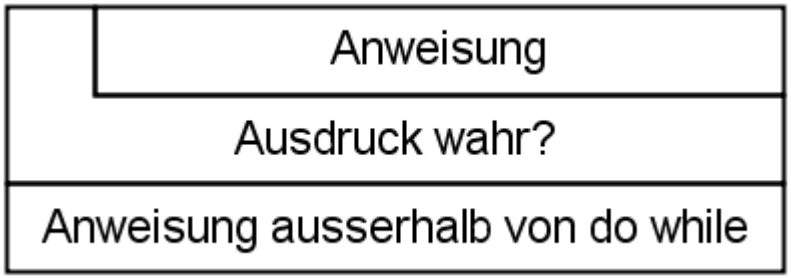
Bsp: `if (test() && tue_manches() )`

→ Achtung: `tue_manches()` wird nur aufgerufen, wenn `test()` true lieferte!!!

Wie while-Schleife, nur daß die Bedingung am Ende des Anweisungsblocks überprüft wird (nicht abweisende Wiederholungsanweisung):

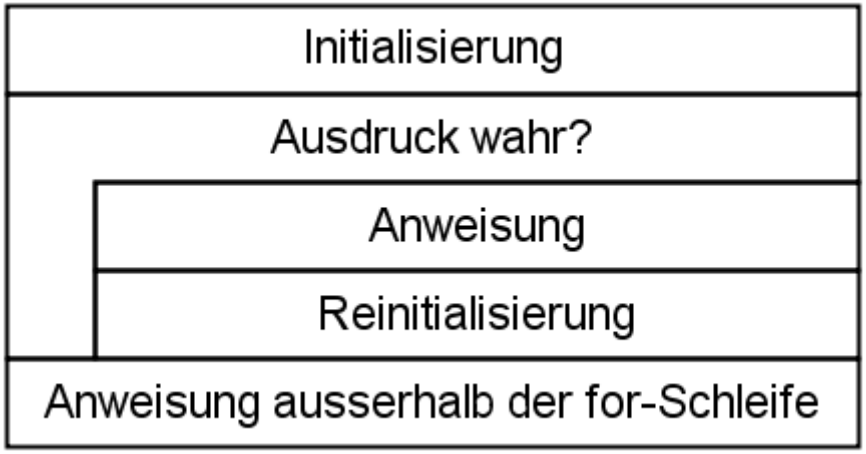
```
do {
      /* Anweisungen */
 }while(BEDINGUNG==wahr);
```

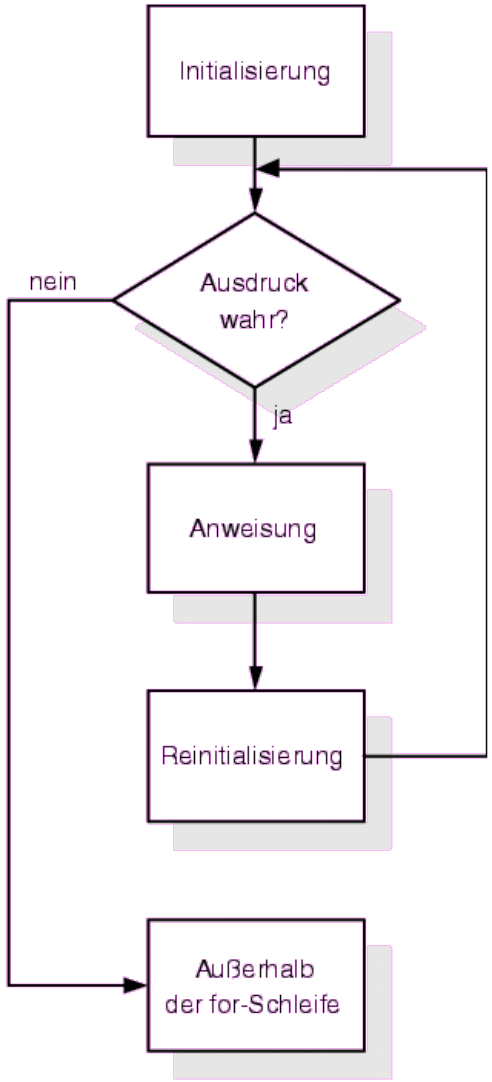




Mehrfache Ausführung von Anweisungsblöcken bei Initialisierung, Bedingungsprüfung und Reinitialisierung der Schleifenvariable in der for-Anweisung:

```
for(Initialisierung; Bedingung;
   Reinitialisierung)
    {
       /* Anweisungen */
    }
```





Forschleife mit While


```
for ((<Initialisierungsausdruck> ; /* initialisiert Zählvar.*/
<Durchlassausdruck> ;              /* Bedingung für Durchlauf*/
<Schleifennachlaufausdruck> ){     /* Operation auf Zählvariable*/
< Anweisungen … >}                 /* Schleifenrumpf */
```

zu

```
<Initialisierungsausdruck> ;       /* initialisiert Zählvar.*/

while (<Durchlassausdruck>){       /* Bedingung für Durchlauf*/
      <Anweisungen....>            /* Schleifenrumpf */
      <Schleifennachlaufausdruck>/* Operation auf Zählvariable*/
}
```

Besonderheiten

- Durchlassausdruck wird vor dem (ersten) Betreten des Schleifenrumpfs überprüft und kann somit zum "Überspringen" des Schleifenrumpfs führen
- Wert einer globalen Zählvariable ist nach der for-Schleife definiert
- Die Berechnung innerhalb des Schleifenrumpfs muß der Abbruchbedingung zustreben (Durchlassausdruck = false)
- Komma-Operator ermöglicht Initialisierung mehrerer Variablen bzw. Fortschaltung mehrerer Variablen im Schleifennachlaufausdruck
  `for (j=0, i=1; j<max; i++, j++)`
- Jeder der drei for-Schleifenabschnitte (Init, Durchlassen,Fortschalten) kann leer sein. Ein fehlender Durchlassausdruck wird als 'wahr' angenommen!!!
- `for ( ; ; )`
  Formulierung einer endlos-Schleife
  → Die Abbruchbedingung muß somit im Schleifenrumpf stehen (break, return, oä.)
- Der Anweisungsteil kann leer sein (weil schon alles im for-Kopf passiert). Empfehlung: Dies optisch durch eine Zeile nur mit Semikolon und Kommentar verdeutlichen!!!

```
for (i=0; z[i]=q[i]; i++) /* z und q Zeichenketten */
;/* leere Anweisung*/
```

Typische Fehler

- Ein häufiger Fehler ist das Vergessen des _____ zwischen der Initialisierung, Bedingung und der Reinitialisierung der Schleifenvariable.

```
for(i=1 i<=5 i++) /*korrekt->for(i=1; i<=5; i++)*/
```

- Natürlich sollten Sie auch nicht den Fehler machen, am Ende der Schleife ein Semikolon zu setzen; außer es ist gewollt.

```
for(i=1; i<=5; i++); /*korrekt->for(i=1; i<=5; i++){ }
                     */
```

- Und das Wichtigste ist, für die richtige _____ zu sorgen, da sonst das Programm die for-Schleife nicht mehr verlässt. Man spricht dann von einer Endlosschleife.

```
for(i=2; i==5; i+=2) /* korrekt-> for(i=2; i<=5; i+=2)
                      */
```

Kontrollierte Sprünge – continue, break, return, exit

- **continue – damit beenden Sie bei Schleifen nur den aktuellen Schleifendurchlauf.**
- **break – beendet die Schleife oder eine Fallunterscheidung. Befindet sich break in mehreren geschachtelten Schleifen, wird nur die innerste verlassen.**
- **exit – beendet das komplette Programm.**
- **return – beendet die Iteration und die Funktion, in der return aufgerufen wird.**
  Im Fall der main()-Funktion würde dies das Ende des Programms bedeuten.

Name	Größe	Wertebereich	Formatzeichen
char	1 byte	-128 ... +127 oder 0 ... 255	%c
short	2 byte	-32.768 +32.767	%hd oder %hi
integer	Mindestens 2 byte	-32.768 +32.767	%d oder %i
long	Mindestens 4 byte	-2.147.483.648 + 2.147.483.647	%ld oder %li

Darstellung reeller Zahlen

- **Reelle Zahlen werden durch Gleitkommazahlen (Komma steht nicht an einer festen Stelle):**
  Beispiel Lichtgeschwindigkeit im Vakuum:
  c = 299.792.458 m/s = 299.792,458·10^3 m/s = 0,299.792.458·10^9 m/s = 2,997.924.58·10^8 m/s
- **Basis b:**
  - **2,997.924.58·10^8**
- **Mantisse m:**
  - 2,997.924.58·**10^8**
  - **Drückt aus, wie exakt Zahl approximiert wird**
- **Exponent e:**
  - **2,997.924.58·10^8**
  - **Gibt Stelle des Kommas an**

$$\text{Wert} = (-1)^{\text{sign}} \times \text{Mantisse} \times \text{Basis}^{\text{Exponent}}$$

- **Darstellung einer Gleitkommazahl _____:**
  - Zahl 2 als $2{,}0 \cdot 10^0$ oder als $0{,}2 \cdot 10^1$ geschrieben werden.

  → normalisierte Gleitkommazahl, bei der $1 \leq m < b$.

  → $2{,}0 \cdot 10^0$ wäre normierte Zahl.

  → Zahl 0 kann nicht normalisiert dargestellt werden → für den kleinsten Exponenten auch nichtnormalisierte Darstellung zulässig.
- **Hardware für Rechnen mit normalisierten Zahlen sehr viel einfacher.**

- **Exponent hat Vorzeichen**
  - **Implementierung einer zusätzlichen ganzzahligen Arithmetik oder**
  - **Addition eines festen Bias-Wertes B zu dem Exponenten e: E=e+B**
    - **Summe E ist vorzeichenfrei**
- **Vorteil der Bias-Darstellung: Größer/kleiner-Vergleich zwischen 2 positiven Gleitkommazahlen**

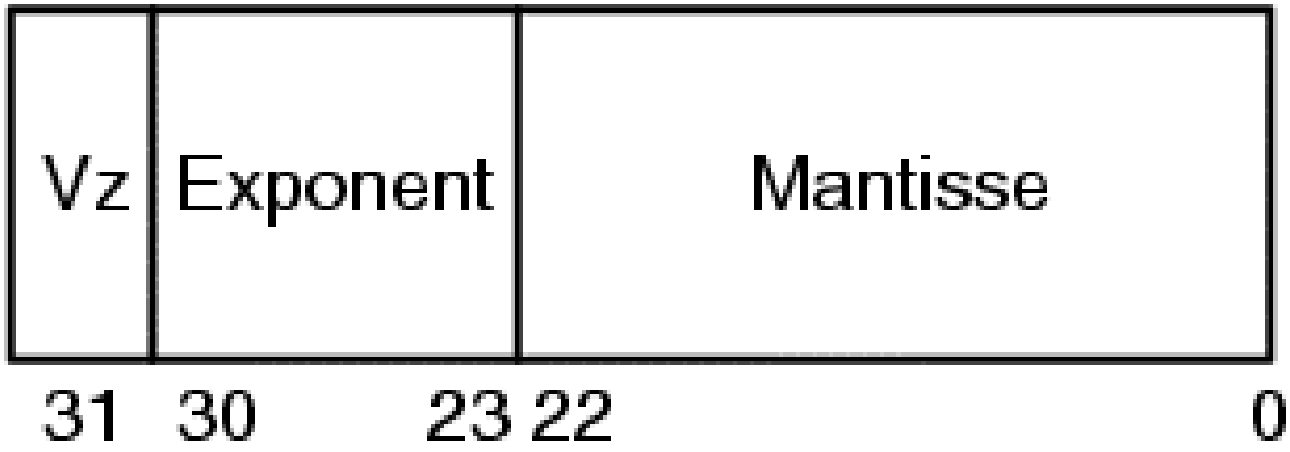
  _____:

  - **em (Exponent e gefolgt von Mantisse m) lexikografisch miteinander vergleichen!**
- **Nachteil der Bias-Darstellung: nach einer Addition zweier Biased-Exponenten muß Bias subtrahiert werden.**

- **Darstellung von Zahlen mit _____:**
  - `float a=1,5;  /* FALSCH */`
  - `float b=1.5;  /* RICHTIG */`
- **4 Byte groß**
- **Formatzeichen**
  - **%f für [–] dddd.dddd**
  - **%e für [–]d.dddd e [sign]ddd**
- **Wertebereich 1.2E-38 bis 3.4E+38**
  - **Darstellung halb logarithmisch → Zerteilung in Vorzeichen, Mantisse und Exponent zur Basis 2**
  - **Genauigkeit wird durch Anzahl bits für Mantisse bestimmt**
  - **Wertebereich wird durch Anzahl bits für Exponenten bestimmt**

- **Vorzeichen (Vz)-Bit (___ Bit): In Bit 31 wird das Vorzeichen der Zahl gespeichert. Ist dieses 0, dann ist die Zahl positiv, bei 1 ist sie negativ.**
- **Exponent (____ Bits): In Bit 23 bis 30 wird der Exponent mit einer Verschiebung (Bias) der Zahl gespeichert (Bias bei float 127).**

- **Mantisse (_____ Bits): In Bit 0 bis 22 wird der Bruchteil der Mantisse gespeichert. Das erste Bit der Mantisse ist immer 1 und wird nicht gespeichert.**



Exponent	Mantisse	dargestelltes Objekt
00...00	00...00	0
1–254	beliebig	± normalisierte Gleitkommazahl
00...00	beliebig, von Null verschieden	± denormalisierte Zahl (ohne Exponent)
255	00...00	± ∞ Unendlich (infinity)
255	beliebig, von Null verschieden	NaN (not a number)

Beispiel

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

Gleitkomma-Addition

- **Prinzip illustriert für Addition zweier Dezimal-Gleitkommazahlen mit vierstelliger Mantisse (für Binärzahlen analog)**

  $9.999 \times 10^{1} + 1.611 \times 10^{-1} = ?$

- ***1. Schritt:*** **_____ der Dezimalkomma-Position durch Rechts-Shift der Zahl mit dem kleineren Exponenten:**

  $9.999 \times 10^{1} + 0.016 \times 10^{1}$

- ***2. Schritt:*** **_____ der Addition:**

  9.999×10^{1}
  $+ 0.016 \times 10^{1}$
  10.015×10^{1}

- ***3. Schritt:*** **_____ des Ergebnisses durch Verschieben:**

  1.0015×10^{2}

- ***4. Schritt:*** **Anpassen der Mantisse an verfügbare Stellenzahl (Runden!)**

  1.002×10^{2}

Gleitpuntdatentypen

Name	Größe	Wertebereich	Genauigkeit	Formatzeichen
float	4 Byte	1.2E-38 3.4E+38	6-stellig	%f
double	8 Byte	2.3E-308 1.7E+308	15-stellig	%lf
long double	10 Byte	3.4E-4932 1.1E+4932	19-stellig	%Lf

- Zahlen im Gleitpunktformat nie auf _____ prüfen!

```
/* never_ending.c */
#include <stdio.h>

int main(void) {
    float i=0.0;
    for (i=0.0; i != 1.0; i += 0.1)
       printf("%f",i);
    return 0;
}
```

Fachhochschule Jena
University of Applied Sciences Jena

Typumwandlung mit

```
(typ) ausdruck
```

Erst wird _____ berechnet und dann der Typ umgewandelt!

```
#include <stdio.h>
int main()
{
   int x=5,y=2;
   float z;

   z = x / y;
   printf("%f\n",z);              /*=2.000000*/

   z = (float)x / (float)y;    /*=2.500000*/
   printf("%f\n",z);
   return 0;
}
```

```
int zaehler = 8 ;
int nenner = 16;
double ergebnis;
double faktor = 1.5 ;
```

- `ergebnis = zaehler / nenner * faktor;`
  → ergebnis = 0.0
- `ergebnis = (double) zaehler;`
  → ergebnis = 8.0
- `ergebnis = (double)zaehler/(double)nenner*faktor;`
  → ergebnis = 0.75
- `ergebnis = (double)zaehler/nenner*faktor;`
  → ergebnis = 0.75
- `ergebnis = (double)(zaehler/nenner)*faktor;`
  → ergebnis = 0.0
- `ergebnis = faktor*zaehler/nenner;`
  → ergebnis = 0.75

Signed/Unsigned

Name	Größe	Wertebereich
char, signed char	1 Byte = 8 Bit	-128...+127
unsigned char	1 Byte = 8 Bit	0...255
short, signed short	2 Byte = 16 Bit	-32768...+32767
unsigned short	2 Byte = 16 Bit	0...65535
int, signed int	4 Byte = 32 Bit	-2147483648... +2147483648
unsigned int	4 Byte = 32 Bit	0...4294967295
long, signed long	4 Byte = 32 Bit	-2147483648... +2147483648
unsigned long	4 Byte = 32 Bit	0...4294967295
float	4 Byte = 32 Bit	$3.4*10^{-38}$ ... $3.4*10^{38}$
double	8 Byte = 64 Bit	$1.7*10^{-308}$ ... $1.7*10^{308}$
long double	10 Byte = 80 Bit	$3.4*10^{-4932}$ ... $3.4*10^{4932}$

Direkte Konstante → unmittelbar definitierter Typ

- **Numerische Konstanten sind in C/C++ typbehaftet!**
  - **12345 → int**
  - **1.2345 → double (! nicht float !)**
  - **1.2345f → float**
  - **0x1A , 0X1A → int, hexadezimal (Wert: 26)**
  - **012345 → int, oktal (Wert: 5349)**
  - **7926U → unsigned**
  - **7926L → long**
  - **'A' → char, Wert abhängig vom Zeichensatz**
- **Typinterpretation durch U(unsigned) / L(long) / f(float)**
- **Achtung: Versehentliches Vergessen des Typspezifizierers bei float kann zu undefinierten Belegungen der Variablen führen!!!**

- % FLAG F W G L U
  - Flag
  - F = [Formatierungszeichen]
  - W = [Weite]
  - G = [Genauigkeit]
  - L = [Längenangabe]

Flag	Bedeutung
-	Linksbündig justieren
+	Ausgabe des Vorzeichens '+' oder '-'
Leer-zeichen	Ist ein Argument kein Vorzeichen, wird ein Leerzeichen mit ausgegeben.
0	Bei numerischer Ausgabe wird mit Nullen bis zur angegebenen Weite aufgefüllt.
#	Bei o bzw. x oder X wird mit vorangestellter 0 bzw. 0x ausgegeben. Bei e, E oder f wird der Wert mit einem Dezimalpunkt ausgegeben, auch wenn keine Nachkommastelle existiert.

Formatierungszeichen

Formatierungs-zeichen	Es wird ausgegeben (eine) ...
%d, %i	... vorzeichenbehaftete ganze Dezimalzahl
%o	... vorzeichenlose ganze Oktalzahl
%u	... vorzeichenlose ganze Dezimalzahl
%x, %X	... vorzeichenlose ganze Hexzahl (a,b,c,d,e,f) bei x; (A,B,C,D,E,F) bei X
%f	... Gleitpunktzahl in Form von ddd.dddddd
%e, %E	... Gleitpunktzahl in Form von d.ddde+-dd bzw. d.dddE+-dd. Der Exponent enthält mindestens 2 Ziffern.
%g, %G	... float ohne Ausgabe der nachfolgenden Nullen
%c	... Form von einem Zeichen (unsigned char)
%s	... Form einer Zeichenkette
%p	... der Zeigerwert
%%	... das Zeichen %

Angaben	Bedeutung
n	Es werden mindestens n Stellen ausgegeben, auch wenn der Wert weniger als n Stellen besitzt.
*	Wert des nächsten Arguments (ganzzahlig) legt die Weite fest. Bei negativem Wert wird linksbündig justiert.

- **Der Schreibweise geht ein Punkt voran:**
  - **printf(„%.2f\n“,3.143234);**
- **Bezieht sich nur auf Ausgabe, Wert wird nicht verändert.**

Modifikation	Auswirkung
h	Die Umwandlungszeichen d, i, o, u, x, X werden als short-Wert behandelt.
l	Die Umwandlungszeichen d, i, o, u, x, X werden als long-Wert behandelt. e, E, f, g, G werden als double behandelt
L	Die Umwandlungszeichen e, E, f, g, G werden als long double-Wert behandelt.

Mit Hilfe von Bit-Operatoren kann direkt auf die _____ Darstellung der Zahlen zurückgegriffen werden.

Bit-Operator	Bedeutung
&, &=	Bitweise AND-Verknüpfung
|, |=	Bitweise OR-Verknüpfung
^, ^=	Bitweise XOR
~	Bitweises Komplement
>>, >>=	Rechtsverschiebung
<<, <<=	Linksverschiebung

Bitweises Und

X=55 & 7

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	128	64	32	16	8	4	2	1
55	0	0	1	1	0	1	1	1
&7	0	0	0	0	0	1	1	1
7	0	0	0	0	0	1	1	1

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Bsp: Test auf Gerade oder Ungerade Zahl


```
/* gerade.c */
#include <stdio.h>

int main(void) {
   int x;

   printf("Bitte geben Sie eine Zahl ein: ");
   scanf("%d",&x);
   if(x&1)  // Ist das erste Bit gesetzt?
      printf("Eine ungerade Zahl\n");
   else     // Nein, es ist nicht gesetzt
      printf("Eine gerade Zahl\n");
   return 0;
}
```

Bitweises Oder

X=1 | 126

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	128	64	32	16	8	4	2	1
1	0	0	0	0	0	0	0	1
|126	0	1	1	1	1	1	1	0
127	0	1	1	1	1	1	1	1

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

Bitweises XOR

X=20 ^55

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	128	64	32	16	8	4	2	1
20	0	0	0	1	0	1	0	0
^55	0	0	1	1	0	1	1	1
35	0	0	1	0	0	0	1	1

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Bsp: Prüfen auf Ungleichheit

Bitweises Komplement

A	~A
0	1
1	0

Verschiebung

X= 8<<1

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	128	64	32	16	8	4	2	1
8	0	0	0	0	1	0	0	0
<<1	0	0	0	1	0	0	0	0

- **Linksverschiebung → Multiplikation mit 2**
- **Rechtsverschiebung → Division durch 2**

- Präprozessor nimmt vor der _____ Änderungen am Quelltext vor.
- Beginnt mit dem Zeichen _____ am Anfang der Zeile
- Aufgaben:
  - Quelltextersetzung
  - String-Literale werden zusammengefasst (konkateniert).
  - Zeilenumbrüche mit einem Backslash am Anfang werden entfernt.
  - Kommentare werden entfernt und durch Leerzeichen ersetzt.
  - Whitespace-Zeichen zwischen Tokens werden gelöscht.
  - Header- und Quelldateien in den Quelltext kopieren (#include)
  - Symbolische Konstanten einbinden (#define)
  - Bedingte Kompilierung (#ifdef, #elseif ...)

- **#include kopiert andere Dateien in das Programm ein:**
  #include <header.h>
  #include „header.h“

Standard Header-Dateien

Headerdate	Bedeutung
assert.h	Fehlersuche und Debugging
ctype.h	Zeichentest und Konvertierung
errno.h	Fehlercodes
float.h	Limits/Eigenschaften für Gleitpunkttypen
limits.h	Implementierungskonstanten
locale.h	Länderspezifische Eigenschaften
math.h	Mathematische Funktionen
setjmp.h	Unbedingte Sprünge
signal.h	Signale
stdarg.h	Variable Parameterübergabe
stddef.h	Standard-Datentyp
stdio.h	Standard-I/O
stdlib.h	Nützliche Funktionen
string.h	Zeichenkettenoperationen
time.h	Datum und Uhrzeit

```
# define Bezeichner Ersatzbezeichner
Bsp:

#include <stdio.h>
#define EINS 1

int main()
{
   printf(„EINS ist %d\n",EINS);
   return 0;
}
```