Teil 4: Datentypen in C

Gliederung

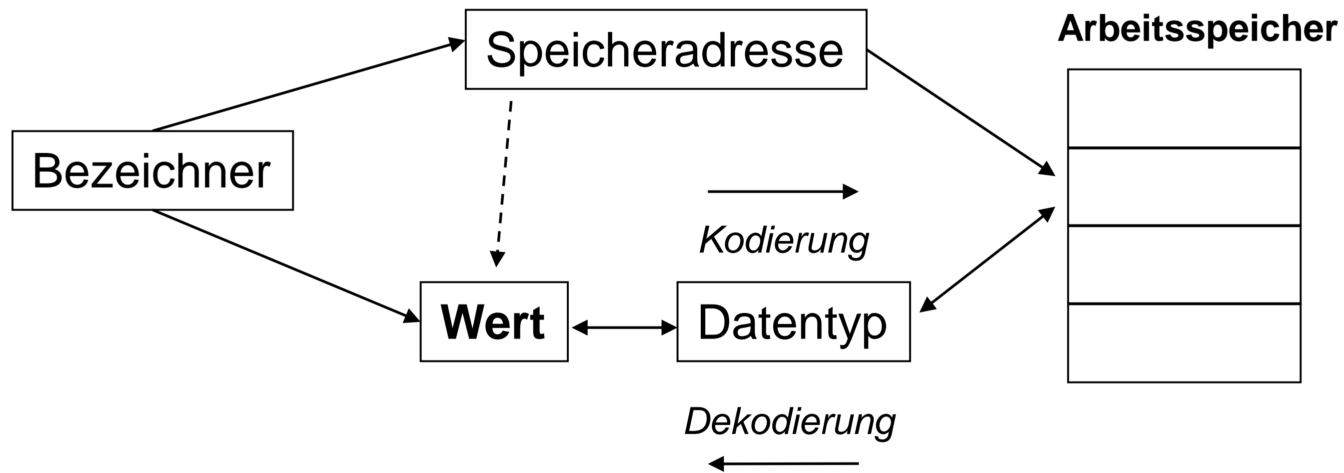
Numerische Typen

Ausdrücke und Operatoren

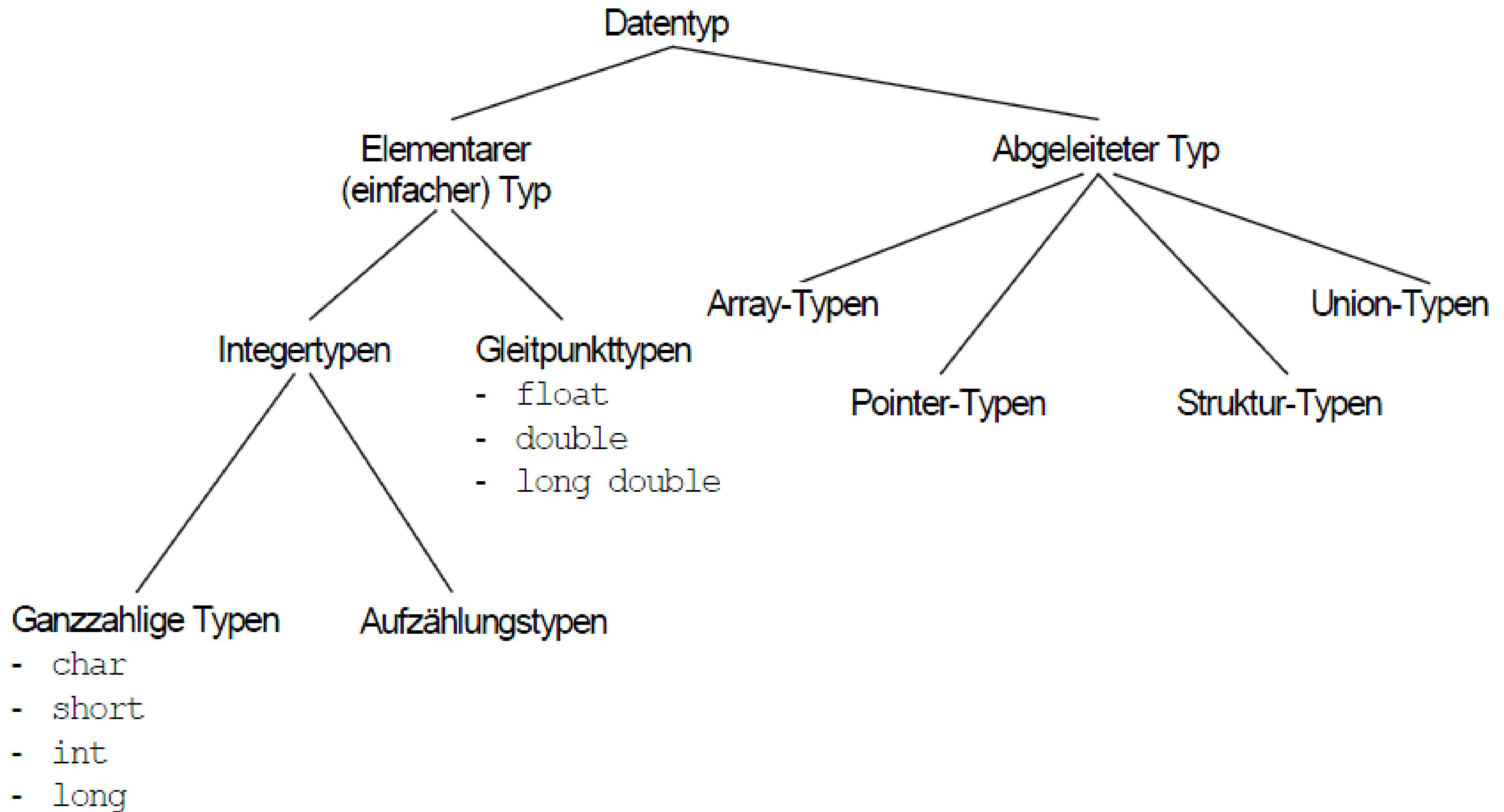
Typumwandlung

Numerische Typen

Variablenmodell



Typ-Klassifikation



ANSI C89: Elementare Datentypen

Typ	Wertebereich (Genauigkeit)	Größe	E / A
int	- 32 768 ... 32 767 bei 16 Bit Maschinen - 2 147 483 648 ... 2 147 483 647 32 Bit	2 Byte 4 Byte	%d
unsigned int	0 ... 65 535 bei 16 Bit Maschinen 0 ... 4 294 967 295 bei 32 Bit Maschinen	2 Byte 4 Byte	%u
short int	- 32 768 ... 32 767	2 Byte	%d
unsigned short int	0 ... 65 535	2 Byte	%u
long int	- 2 147 483 648 ... 2 147 483 647	4 Byte	%ld
unsigned long int	0... 4 294 967 295	4 Byte	%lu
char	alle Zeichen im ASCII Code	1 Byte	%c
char	-128 ... 127	1 Byte	%d
unsigned char	0 ... 255	1 Byte	%u
float	1,2 E-38 ... 3,4 E+38 (6 Stellen)	4 Byte	%f
double	2,3 E-308 ... 1,7 E+ 308 (15 Stellen)	8 Byte	%lf
long double	3.4 E-4932 ... 1.1 E+4932 (19 Stellen)	10 Byte	%lf
void	leerer Typ	0 Byte	

ANSI C99: Erweiterungen

Typ	Wertebereich	Größe
_Bool	0 und 1	1 Byte
long long	–9223372036854775808 bis 9223372036854775807	8 Byte
unsigned long long	0 bis 18446744073709551615	8 Byte

Regeln für die Compiler-Implementierung:

short int	≤	int
long int	≥	int
int	≥	2 Byte
long long int	=	8 Byte

Bestimmung der Typgröße zur Laufzeit

- Ermittlung zur Laufzeit, wieviel Speicher Datentyp/Variable/Konstante belegt:

```
int sizeof(Variable);
int sizeof(Konstante);
int sizeof(Datentyp);
```

- Beispiel

```
#include <stdio.h>

main()
{
   long int x;

   printf("size of long int: %d\n", sizeof(long int));
   printf("size of x:        %d\n", sizeof(x));
}
```

Ganzzahltypen

unsigned				
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

signed				
-8	1	0	0	0
-7	1	0	0	1
-6	1	0	1	0
-5	1	0	1	1
-4	1	1	0	0
-3	1	1	0	1
-2	1	1	1	0
-1	1	1	1	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1

Ganzzahltypen: Zweierkomplement

- Invertierung aller Bits und Addition von 1

	$_1$0000	$_1$0000	$_1$0000		$_1$0000
B	– 0001	– 0010	– 0011		– 0111
A	1111	1110	1101	....	1001

- zu jedem Bit-Tupel gibt es ein Bit-Tupel so dass gilt:

$$B = \{\mathbf{0}\ B_{n-1}\ \dots\ B_1\}$$
$$A = \{\mathbf{1}\ A_{n-1}\ \dots\ A_1\},$$
$$A + B = \mathbf{1}\{0\ \dots\ 0\}$$

- `A` heißt "Zweier-Komplement" zu `B`
- wenn man Überlauf ignoriert, ist `A = -B`

Ganzzahltypen: Beispiel

```
#include <stdio.h>

int main()
{
    short int a = 50;
    short int b = 1000;
    short int c;

    c = a * b;
    printf("%i * %i = %i", a, b, c);

    return 0;
}
```

Ganzzahltypen: Beispiel

```
#include <stdio.h>

int main()
{
    short int a = 50;
    short int b = 1000;
    short int c;

    c = a * b;
    printf("%i * %i = %i", a, b, c);

    return 0;
}
```

Mögliche Fehler bei Rechenoperationen: **Überlauf**

Reelle Zahlen

- gebrochene Zahlen - wie Speichern?
- Bestandteile: **Vorzeichen**

  Ziffernfolge

  Komma (bzw. Position)
- in Exponentialschreibweise: $\mathbf{y} \cdot 10^{\mathbf{z}}$

  y: Stellenzahl bestimmt Genauigkeit
  z: Stellenzahl bestimmt Größe
- Mantisse **y** = Festkommazahl durch Normierung auf $1 \leq \mathbf{y} < 10$
- Exponent **z** = Position des Kommas

→ Gleitkommazahl = **Vorzeichenbit + Mantisse + Exponent**

Reelle Zahlen: Datentypen

- IEEE 754-1985

Typ	Bits	Zahlenbereich	Genauigkeit
float	32	$+/-3.4*10^{-38}$ ... $+/-3.4*10^{38}$	7
double	64	$+/-1.7*10^{-308}$ ... $+/-1.7*10^{308}$	15
long double	80	$+/-3.4*10^{-4932}$ ... $+/-3.4*10^{4932}$	19

- beliebige Genauigkeit **nicht für alle Zahlen** möglich
- mit 32 Bits existieren **nur** 2^{32} = 4.294.967.296 Möglichkeiten

Reelle Zahlen: Beispiel

Programmbeispiel:

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

Reelle Zahlen: Codierung der Mantisse

dez 2^x	binär	dezimal
2^5	100000	32
2^4	10000	16
2^3	1000	8
2^2	100	4
2^1	10	2
2^0	1	1
2^{-1}	0.1	0.5
2^{-2}	0.01	0.25
2^{-3}	0.001	0.125
2^{-4}	0.0001	0.0625
2^{-5}	0.00001	0.03125

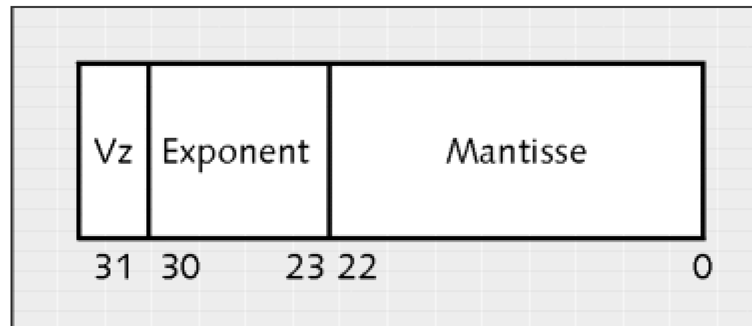
Reelle Zahlen: Codierung der Mantisse

dez 2^x	binär	dezimal
2^5	100000	32
2^4	10000	16
2^3	1000	8
2^2	100	4
2^1	10	2
2^0	1	1
2^{-1}	0.1	0.5
2^{-2}	0.01	0.25
2^{-3}	0.001	0.125
2^{-4}	0.0001	0.0625
2^{-5}	0.00001	0.03125

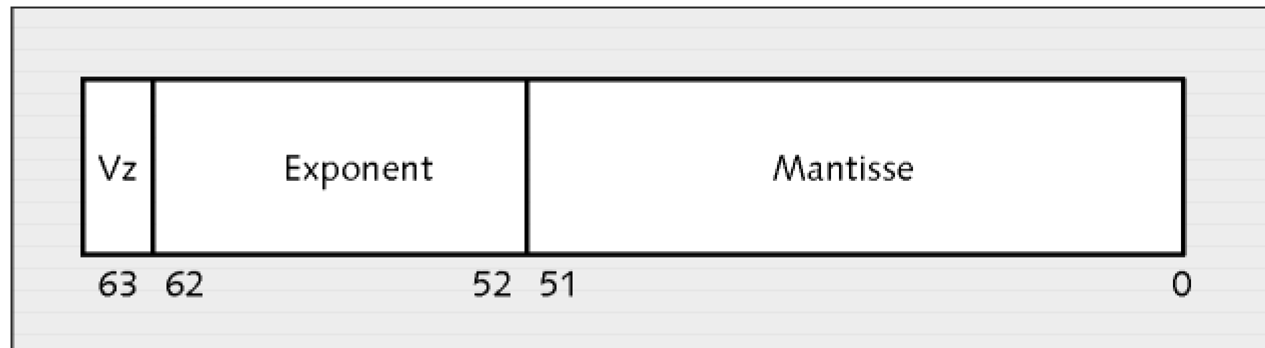
← 0.1 ?

Codierung der Typen float, double

float



double



float (nach IEEE 754)

```
SEEEEEEE EMMMMMMM MMMMMMMM MMMMMMMM
|                                 |
MSB                             LSB
```

S	Sign (Vorzeichen): 0 = Plus, 1 = Minus
EEEEEEEE	Exponent: 1 Byte, Bereich von 0...255
MM...MMM	Normalisierte Mantisse: Vorkommabit ist immer 1 **1 ≤ Mantisse < 2** Wertigkeit der Bits: ½, ¼, usw.

$$\textbf{Wert}_{\textbf{dez}} = (-1)^{S} * 1{,}MM...MMM * 2^{EEEEEEEE-127}$$

Reelle Zahlen: Rechenfehler 1

```
#include <stdio.h>

int main()
{
   float a = 1.0, b = -1.0;
   float c = 0.000000001234567;
   float s1, s2;

   s1 = a + b;
   s1 = s1 + c;

   s2 = a + c;
   s2 = s2 + b;

   printf("(a + b) + c: %e\n", s1);
   printf("a + (b + c): %e\n", s2);

   return 0;
}
```

Reelle Zahlen: Rechenfehler 2

```
float i;

for (i = 0.0; i <= 1.0; i = i + 0.1)
  printf("%f\n", i);
```

Reelle Zahlen: Rechenfehler 2

```
float i;
for (i = 0.0; i <= 1.0; i = i + 0.1)
  printf("%f\n", i);
```

besser:

```
float i;
for (i = 0.0; i <= 1.01; i = i + 0.1)
  printf("%f\n", i);
```

Reelle Zahlen: Rechenfehler 2

```
float i;
for (i = 0.0; i <= 1.0; i = i + 0.1)
  printf("%f\n", i);
```

besser:

```
float i;
for (i = 0.0; i <= 1.01; i = i + 0.1)
  printf("%f\n", i);
```

richtig:

```
int i;
for (i = 0; i <= 10; i = i + 1)
  printf("%f\n", i / 10.);
```

Reelle Zahlen: Rechenfehler

Problem 1: **Numerische Auslöschung**
Subtraktion 2er fast gleich großer Werte
→ signifikante Stellen heben sich auf

Problem 2: **Überlauf** (Overflow): Division durch betragsmäßig zu kleinen Wert

Problem 3: **Reihenfolge der Operationen relevant**

Problem 4: **Rundungsfehler**

Standardbibliothek <math.h>

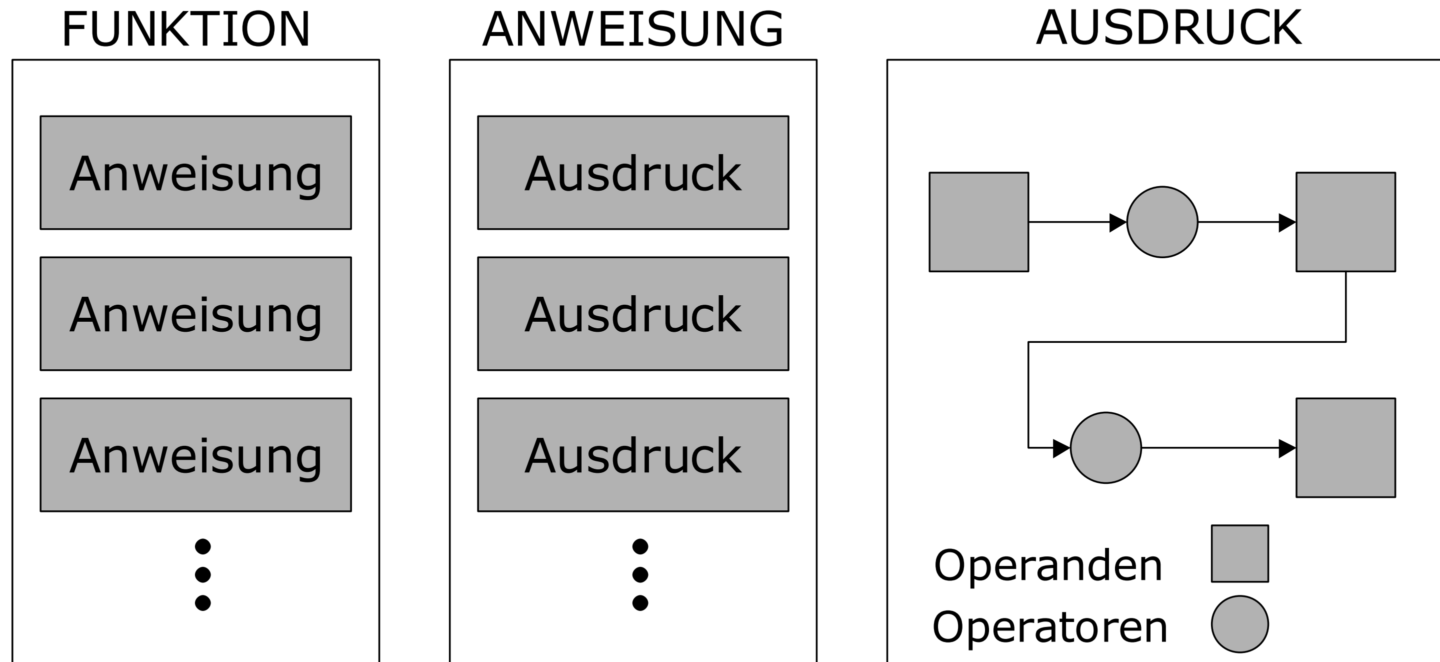
```
#include <math.h>
```

Trigonometrische, Hyperbel-, Logarithmische, Exponential-, Potenz-, Rundungsfunktionen:

```
double sin(double x);
double log(double x);
double pow(double base, double exponent);
double round(double x);
...
```

Ausdrücke und Operatoren

Anweisung und Ausdruck



Anweisungen

Block { ... }

durch geschweifte Klammern eingeschlossen,
jede Anweisung, die kein Block ist, wird durch Semikolon terminiert

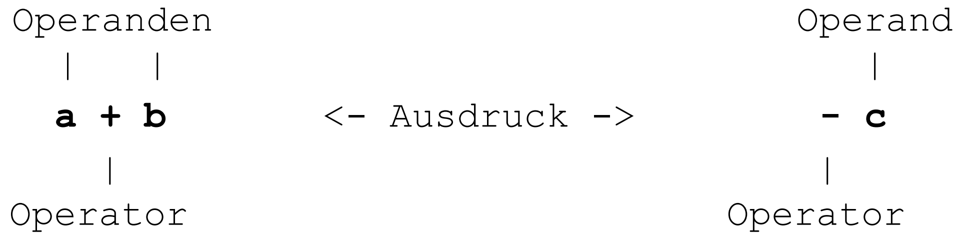
Ablaufsteuerung

Auswahlanweisung
Wiederholungsanweisung
Sprunganweisung

Ausdrucksanweisung

Zuweisung:	`m = n + 2;`
Funktionsaufruf	`printf("Hallo");`
Leere Anweisung	`;`

Ausdrucksanweisung




```
5 * 5;
i++;            // Anweisungen, bestehend aus Ausdrücken
a = b + 3;
```

- **Ausdruck**

  Folge von Operanden, Operatoren und möglichen Klammern

- **Ausdrucksanweisung**

  hat **immer** einen Rückgabewert
  (vgl. Ablaufsteuerung: **muss keinen** Rückgabewert haben)

Arithmetikoperatoren

Klasse	Operatoren	Beispiele
Unär	**-**	`-a`

Binär	**+ , - , * , /**
	% (nur int)
		`a + b`
		`a % d`
		`a * b / c`

Relationale Operatoren

Klasse	Operatoren	Beispiele
relational	**< , > , <= , >=**	`a < b` `x >= d`
Gleichheit	**==** **!=**	`a == b` `c != d`

relationale Ausdrücke liefern den Typ **int** zurück: **0** (false) oder **1** (true)

Zuweisungsoperator

- Variablen / Ausdrücke besitzten L-value und R-value

  L-value = Adressewert
  R-value = Variablenwert oder Ausdruck

- L = Links und R = Rechts

```
a = b;
wert = 3 + 2 * (8 - 4);
```

- Ergebnis der Zuweisung = zugewiesener Wert

```
a = b = c = d = 0;
x = 2 * (y = (z = 4) + 1);
```

Beispiele

Beispiel 1: Wird das gewünschte Ergebnis erreicht?

```
int ok, wert;

printf("Geben Sie eine Zahl zwischen 10 und 20 ein: ");
scanf("%d", &wert);
ok = (10 <= wert <= 20);
```

Beispiel 2: Fließkommatypen?

```
if (d == 7.123)
{
    ...
}
```

Beispiele

Beispiel 3: Wann werden beide Funktionen f1() und f2() aufgerufen?

```
if ( f1() || f2() )
...
```

Beispiel 4: Was bewirkt der folgende Ausdruck?

```
(x > 0) || (x = 0)
```

Beispiel 5:

```
!(x1 && x2 && x3 ... && xn)
```

ist logisch äquivalent zu

```
!x1 || !x2 || !x3 ... || !xn
```

→ KV-Diagramme zur Vereinfachung komplexer Ausdrücke

Inkement und Dekrement

Der Inkement Operator ++ bedeutet "das nächste":

```
int x;
x++;   // entspricht   x = x + 1;
```

Dekrement Operator -- bedeutet "das vorherige":

```
x--;   // entspricht   x = x - 1;
```

Beispiel:

```
int x;
x = 3;
z = x++ - 2;
```

POSTFIX

```
int x;
x = 3;
z = ++x - 2;
```

PREFIX

Bitweise Operatoren

Typische Anwendung: Stati, Fehlerzustände, Signale

Symbol	Bedeutung
~	Negation
<< , >>	left shift (Bitweises Linksschieben), right shift (Bitweises Rechtsschieben)
& , | , ^	AND , OR , XOR

x	0 0 1 1
y	1 0 1 0
~x	
x & y	
x | y	
x ^ y	
x << 2	
y >> 2	

Sonstige Operatoren der Sprache C

Operator	Bezeichnung	Beispiel	Erklärung
(<Datentyp>)	explizite Typumwandlung	float x; ... x = (float) a;	Wert von a wird in angegebenen Datentyp umgewandelt
sizeof	Größe in Bytes	unsigned int y; ... y = sizeof (double);	Anzahl der Bytes, die der Ausdruck oder der Datentyp belegt
?:	bedingte Bewertung	int x, y; ... x = (y > 1) ? 3 : -3;	=> y > 1, dann ist x = 3 => y <= 1, dann ist x = -3

expr1,
expr2,
...
	Sequenz	int x, c, d = 3;
		...
		x = (c = d, c <= 5);
			liefert x=1 und c=3,
			Das Ergebnis ist der Wert des letzten Operanden.

Übung 1

Ermitteln Sie die Rückgabewerte der folgenden logischen Ausdrücke:

```
a) 4 < 8 && 21 + 3 != 10
b) 10 > 4 || 12 > 9
c) 4 > 5 && 3 * 4 == 12
d) 2 * 8 != 41 || 4 == 5
e) !3 || 4 - 2 != 2
f) 16 + 5 > 15 && 3 * 5 == 16
```

Hinweis: Fügen Sie zunächst Klammern gemäß der Prioritäten ein, um die Übersichtlichkeit zu erhöhen.

Übung 2

Entwickeln Sie einen Ausdruck zur Berechnung eines Schaltjahres. Die Variable `jahr` enthält die Jahreszahl. Der Ausdruck soll 1 liefern, wenn das Jahr ein Schaltjahr ist.

Ein Jahr ist dann ein Schaltjahr, wenn die Jahreszahl durch 4 teilbar, jedoch nicht, wenn sie durch 100 teilbar ist. Ist die Jahreszahl durch 400 teilbar, so ist das Jahr trotzdem ein Schaltjahr.

Tipp: Machen Sie vom Modulo-Operator Gebrauch.

Übung 3

Finden Sie (ohne Computer) heraus, welche der folgenden Ausdrücke korrekt und welche falsch sind. Welchen Wert haben die korrekten Ausdrücke?

```
1)    12 * 9+3
2)    -+1
3)    24 / 8+3 / 2
4)    (((1)))
5)    7/-3+8*-2- -1/-2
6)    5+-+5+-5
7)    5*/*5*/5
8)    (((1)))(((1)))
9)    23/7%4+1
10)   23.0/7%4+1
```

Typumwandlung

Typumwandlung (cast)

- Konvertierung in anderen Datentyp (u.U. Datenverlust!)
- Compiler nimmt Typumwandlung in Ausdrücken vor → **impliziter cast**
- Typumwandlung wird vom Programmierer erzwungen → **expliziter cast**

Beispiel: Welche Typumwandlung liegt vor? (implizit / explizit / keine)

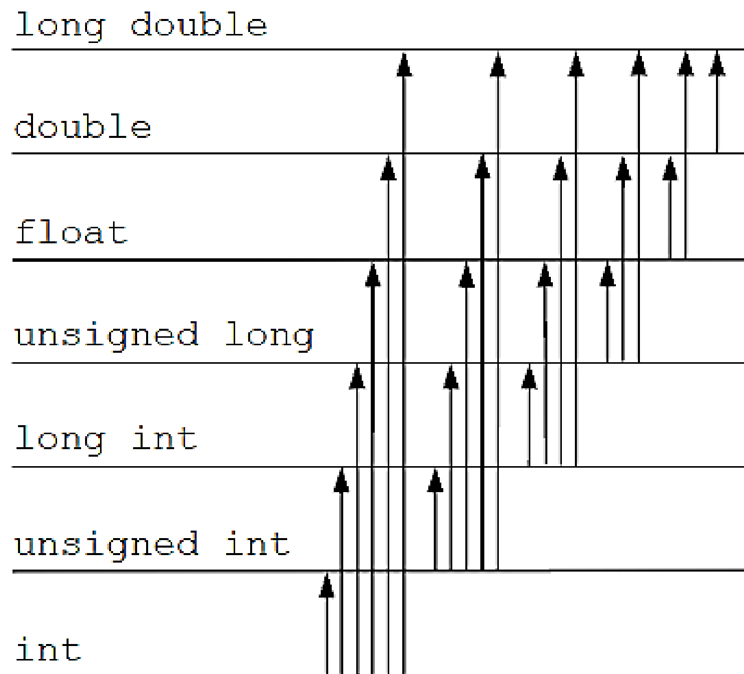
```
int a, b = 77;
double e, d = 1.23;
char f = 'A';

a = f;
e = b;
f = b;
a = d;
a = (int)d;
```

Typumwandlung (cast)

- Operanden eines binären Operators haben unterschiedlichen Datentyp
  → gemeinsamer Datentyp wird **implizit** gebildet:

```
char < short < int < long < long long
```

- Hierarchie typübergreifend:



Typumwandlung Beispiel 1

Programmbeispiel: Wie oft wird die folgende Schleife durchlaufen?

```
int i;
unsigned int grenze = 10;

for ( i = -1; i < grenze; i++)
{
   printf("%d\n", i);
}
```

Typumwandlung Beispiel 2

- Variablen unterschiedlichen Typs in einem Ausdruck
- Compiler rechnet automatisch mit "größten" Typ
- Ausnahmefälle erfordern expliziten cast

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

ergebnis = zaehler / nenner * faktor;
```

Typumwandlung Beispiel 2

- Variablen unterschiedlichen Typs in einem Ausdruck
- Compiler rechnet automatisch mit "größten" Typ
- Ausnahmefälle erfordern expliziten cast

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

~~`ergebnis = zaehler / nenner * faktor;`~~

```
ergebnis = (double) zaehler / (double) nenner * faktor;
```

Typumwandlung Beispiel 2

- Variablen unterschiedlichen Typs in einem Ausdruck
- Compiler rechnet automatisch mit "größten" Typ
- Ausnahmefälle erfordern expliziten cast

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

~~ergebnis = zaehler / nenner * faktor;~~

```
ergebnis = (double) zaehler / (double) nenner * faktor;

ergebnis = (double) zaehler / nenner * faktor;
```

Typumwandlung Beispiel 2

- Variablen unterschiedlichen Typs in einem Ausdruck
- Compiler rechnet automatisch mit "größten" Typ
- Ausnahmefälle erfordern expliziten cast

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

~~ergebnis = zaehler / nenner * faktor;~~

```
ergebnis = (double) zaehler / (double) nenner * faktor;

ergebnis = (double) zaehler / nenner * faktor;

ergebnis = (double) (zaehler / nenner) * faktor;
```

Typumwandlung Beispiel 2

- Variablen unterschiedlichen Typs in einem Ausdruck
- Compiler rechnet automatisch mit "größten" Typ
- Ausnahmefälle erfordern expliziten cast

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

~~`ergebnis = zaehler / nenner * faktor;`~~

`ergebnis = (double) zaehler / (double) nenner * faktor;`

`ergebnis = (double) zaehler / nenner * faktor;`

~~`ergebnis = (double) (zaehler / nenner) * faktor;`~~

`ergebnis = faktor * zaehler / nenner;`