Inhalt

3 C-Ausdrücke

Entsprechend der üblichen Syntax, gegebenenfalls unter Verwendung der Klammer „(“ und „)“, lassen sich Variable und Konstanten mittels Operatoren zu **Ausdrücke** verknüpfen. Je nach dem Hauptverknüpfungsoperator unterscheidet man:

Arithmetische Ausdrücke
Wertzuweisungen
Inkrementieren und Dekrementieren
Logische Ausdrücke (Bedingungen)
Bedingte Ausdrücke
Bitausdrücke
Kommaausdrücke

3.1 Arithmetische Ausdrücke

Operatoren: **+ -** **(unär)**
*** / % + -** **(binär)**

Syntax: wie üblich, mit Klammern.

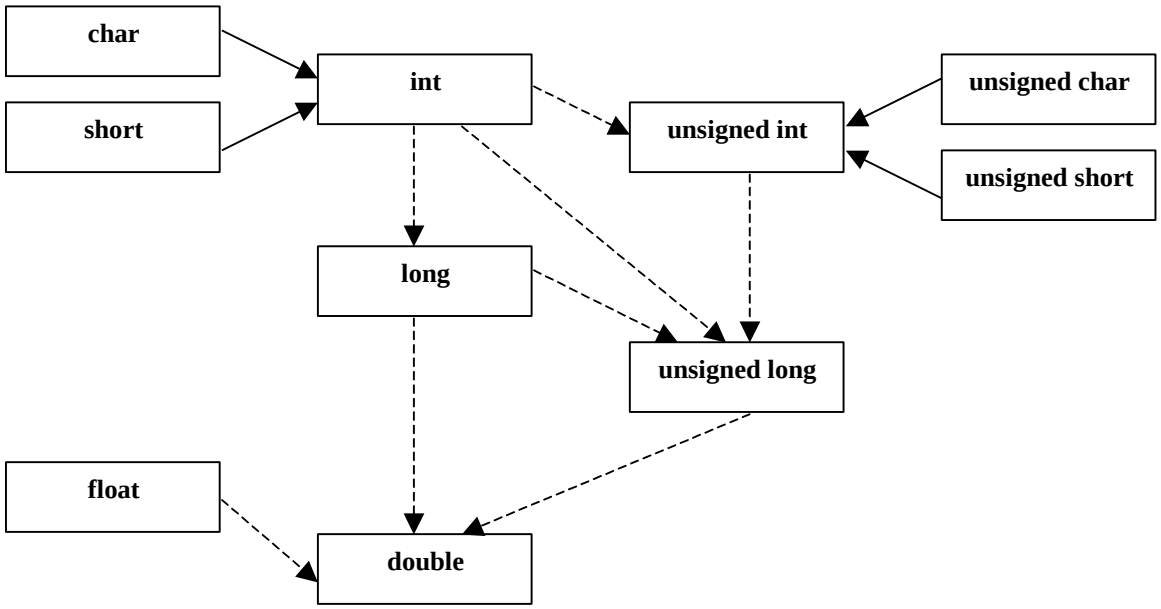
Semantik:

Die auszuführende Operation ist vom jeweiligen **Operator** und den **Operandentypen** abhängig. Es gibt feste Regel zur **Typenkonvertierung** eines oder beider Operanden. Sind beide Operanden vom selben Typ oder werden sie in diesen umgewandelt, so besitzt das Resultat ebenfalls diesen Typ. Damit ist für die ganzzahlige Division kein gesondertes Operationszeichen notwendig.

```
    7 / 3  =>         2
7.0 / 3.0  =>  2.333333
  7.0 / 3  =>  2.333333
1 / 2 * a  =>         0
```

Konvertierung zwischen den Grundtypen:

Bei der Berechnung arithmetischer Ausdrücke werden u. U. Typkonvertierungen ausgeführt. Diese sind in der folgenden Grafik dargestellt. Die Konvertierung erfolgt immer in Richtung des Pfeils. Durchgezogene Pfeile deuten an, dass diese Typumwandlung grundsätzlich vorgenommen wird. Die durch gestrichelte Pfeile dargestellten Typumwandlungen werden bei unterschiedlichen Operandentypen in arithmetischen Ausdrücken ausgeführt.



Explizite Typumwandlung bei Ausdrücken: **(** *Typ* **)** *Ausdruck*

Explizite Typumwandlung bei Konstanten: *Konstante* { *u* | *U* | *f* | *F* | *l* | *L*}

u | *U*: unsigned
f | *F*: float
l | *L*: long int bzw. long double, je nach Darstellung

```
( float) 1 / 2
         1 / 2f
         1. / 2  =>  0.500000
( float)( 1 / 2)  =>  0.000000
```

3.2 Wertzuweisungen

Operatoren: `= *= /= %= += -= <<= >>= &= ^= |=` **(binär)**

Zuweisungsoperatoren sind im Gegensatz zu den meistem anderen Programmiersprachen hier binäre Operatoren.

Syntax: *Variable* = *Ausdruck* (einfache Wertzuweisung)
Variable °= *Ausdruck* mit $\circ \in \{*, /, \%, +, -, <<, >>, \&, \wedge, |\}$
(zusammengesetzte Wertzuweisung)

```
int x, y, z;

x = y = 5;                          => y =  5, x =  5
x += y + ( z = 1 );          => z = 1, y = 5, x = 11
```

Semantik (***x = Aus***)**:**

1. Der Variablen ***x*** wird der Wert des Ausdrucks ***Aus*** zugewiesen.
2. Der komplexe Ausdruck ***x = Aus*** erhält den Wert des rechten Ausdrucks ***Aus***.

Semantik (*x* °= *Aus*):

x* °= *Aus	ist äquivalent mit	***x = x* ° Aus,**
Unterschied:	Anzahl der Speicherzugriffe	

Konvertierung:

Einer Wertzuweisung wird stets der Typ der links stehenden Variablen zugeordnet. D.h. bei der Zuweisung eines Ausdruck mit höherwertigen Typ in eine Variable mit niederwertigen Typ können Stellen verloren gehen.

```
int x; x = 3.1415 * 2;                                  => 6
```

Nebeneffekte (implementierungsabhängig) **:**

```
int x = 1;

x = x + ( x += 10);
                                          =>  1 + 11 => 12
                                          => 11 + 11 => 22
```

In Ausdrücken sollten die Variablen mit Wertzuweisung an keiner weiteren Stelle auftreten.

besser:

```
int x = y= 1; y += 10; x += y;          => 1 + 11 => 12
```

oder

```
int x = 1; x += 10; x *= 2;             => 11 * 2 => 22
```

3.3 Inkrementieren und Dekrementieren

Operatoren: ++ **Inkrementator** **(unär)**
-- **Dekrementator** **(unär)**

Syntax: ++ *int_Variable* (präfix)
-- *int_Variable*

int_Variable ++ (postfix)
int_Variable --

```
int x, y, z; x = y = z = 5;

x = --y + 5;                        => y =  4, x =  9
x *= y ++ + ++ z;          => z = 6, x = 90, y =  5
```

Semantik:

++ x x ++	ist äquivalent mit	**x = x + 1** bzw. **x += 1**
-- x x --	ist äquivalent mit	**x = x - 1** bzw. **x -= 1**
Unterschied:	Anzahl der Speicherzugriffe	

Postfix-Schreibweise (***x*** **++** ***x*** **--**)
Keine unmittelbare Wirkung auf den Wert des komplexen Ausdrucks, ***x*** hat während der Berechnung an dieser Stelle noch den alten Wert.

```
int y, x = 1; y = 3 * x ++;        => y = 3, x = 2
```

Präfix-Schreibweise (**++** ***x*** **--** ***x***)
Vor der Auswertung des komplexen Ausdrucks wird der Wert von ***x*** verändert.

```
int y, x = 1; y = 3 * ++ x;        => y = 6, x = 2
```

Nebeneffekte (implementierungsabhängig) **:**

```
int y, x = 1; y = x ++ + x;          => 1 + 1 => 2
                                     => 1 + 2 => 3
```

In Ausdrücken sollten inkrementierte bzw. dekrementierte Variablen an keiner weiteren Stelle auftreten.

3.4 Logische Ausdrücke (Bedingungen)

Operatoren:	`< <= > >= == !=`	**Vergleiche**	**(binär)**
	&& ||	**Und Oder**	**(binär)**
	!	**Negation**	**(unär)**

Syntax: wie üblich.

```
0                                   =>          falsch
1                                   =>            wahr
123                                 =>            wahr

int x = 2;

0 < x <= 1                          => 1:         wahr
0 < x && x <= 1                     => 0:       falsch

x == 4                              => 0:       falsch
x = 4                               => 4:         wahr
```

Semantik:

In C gibt es *keinen* speziellen logischen Wert (Wahrheitswerte), 0 wird als „**falsch**“ und jeder andere Wert als „**wahr**“ interpretiert. Der Wertebereich ist **{ 0, 1}**.

	&&		**||**		**!**
	0	**≠ 0**	**0**	**≠ 0**	
0	**0**	**0**	**0**	**1**	**1**
≠ 0	**0**	**1**	**1**	**1**	**0**

Optimierung:

Die logischen Operatoren **&&** und || werden grundsätzlich optimierend ausgewertet:

&&: Ist der erste Operand falsch, so wird der zweite Operand nicht ausgewertet.

||: Ist der erste Operand wahr, so wird der zweite Operand nicht ausgewertet.

```
y == 0 || x / y > toleranz
/* Division durch 0 wird vermieden. */

Anzahl < MAXZAHL && scanf( "%f", &Zahl) != EOF
/* Eingabe erfolgt, falls MAXZAHL nicht erreicht */
```

Die Funktion ***int scanf(const char *, . . .)*** ermöglicht eine Eingabe über Tastatur. Sie ist in ***<stdio.h>*** deklariert. Als Parameter muss ein **Formatbeschreiber**, hier "***%f***" für die erwartete Eingabe einer ***float*** - Konstanten, und eine Adresse, hier ***&Zahl*** als Speicheradresse für die einzugebene Konstante, angegeben werden.
EOF ist eine Konstante, ebenfalls in ***<stdio.h>*** definiert, welche als Dateiende fungiert. Über Tastatur kann ***EOF*** durch ***^D*** (Ctrl + d bzw. Strg + d) unter Unix bzw. durch ***^Z*** (Ctrl + z bzw. Strg + z) unter Windows erzeugt werden.

3.5 Bedingte Ausdrücke

Operatoren: **? :**

Syntax: *Bedingung* **?** *Ausdruck* **:** *Ausdruck*

```
float x, y;

y = x >= 0 ? sqrt( x) : -1;
if( y < 0)
  printf( "Nicht loesbar!\n");
else
  printf( "Die Wurzel von %f ist %f.\n", x, y);
```

double sqrt(double) ist eine mathematische Funktion für die **Quadratwurzel** nicht negativer Argumente, die in **<*math.h*>** deklariert ist. "***%f***" dient als Format für die Ausgabe von Gleitkommawerten mit der Funktion ***int printf(const char *, . . .)***.
if (...) ... else ... ist eine **Auswahlanweisung**. Ist der Klammerausdruck wahr, so wird die Anweisung nach dem Klammerausdruck ausgewertet, sonst die nach dem Schlüsselwort ***else***.

Semantik:

Wenn die *Bedingung* wahr ist, wird der erste *Ausdruck* ausgewertet, sonst wird, falls vorhanden, der zweite *Ausdruck* abgearbeitet. Verschachtelungen sind möglich.

Mit diesem Programmausschnitt wird der Quadrant eines Punktes im kartesischen Koordinatensystem bestimmt.

```
float x, y; int quadrant;

quadrant = x >= 0? (y >=0? 1: 4): (y >= 0? 2: 3);
printf( " Punkt ( %f, %f) liegt im Quadranten %d\n",
         x, y, quadrant);
```

3.6 Bitoperationen

Operatoren:	<< >>	**Verschieben**	**(binär)**
	& | ^	**Und Oder Exklusiv-Oder**	**(binär)**
	~	**Negation**	**(unär)**

Syntax: wie üblich, mit ganzzahligen Operanden.

Semantik:

	&		|		^		~
	0	**1**	**0**	**1**	**0**	**1**	
0	**0**	**0**	**0**	**1**	**0**	**1**	**1**
1	**0**	**1**	**1**	**1**	**1**	**0**	**0**

Beispiele (unsigned char):

- Beim **Verschieben** wird mit binären Nullen aufgefüllt

```
1 << 3  :  0000 0001 << 3    =>  0000 1000       =>     8
1 >> 3  :  0000 0001 >> 3    =>  0000 0000       =>     0
```

- **Negation**

```
~ 1     :     ~ 0000 0001    =>  1111 1110       =>   254
~ 254   :     ~ 1111 1110    =>  0000 0001       =>     1
~~ 1    :     ~ 254                              =>     1
```

- **Und**

```
1 & 3   : 0000 0001 & 0000 0011  => 0000 0001   =>     1
```

- **Oder**

```
1 | 3   : 0000 0001 | 0000 0011  => 0000 0011   =>     3
```

- **Exklusiv-Oder**

```
1 ^ 3   : 0000 0001 ^ 0000 0011  => 0000 0010   =>     2
```

3.7 Kommaoperator

Operator: **,** **(binär)**

Syntax:
In C gibt es zwei Anwendungen des Kommas:

1. als Trenner in Listen (Variablenlisten, Parameterlisten). int a**,** b; f(a**,** b);
2. als Operator: *Ausdruck* **,** *Ausdruck*

Semantik:
Die Ausdrücke werden von links nach rechts abgearbeitet. Der Wert und der Typ des komplexen Ausdrucks entsprechen dem Wert und Typ des rechten Ausdrucks.

komma.c

```
/*
 * Programm ruft Funktion zum Stringinvertieren auf.
 */
# include <stdio.h>
# include <string.h>

/*
 * Stringinvertieren
 * @param String
 */
void inv( char *s);

int main()
{ char str[ 10] = "beispiel";

  inv( str);
  printf( "%s\n", str);                  /* leipsieb */

  return 0;
}

void inv( char *s)
{ int c, i, j;                    /* Komma als Trenner */

                                 /* Komma als Operator */
  for ( i = 0, j = strlen( s) - 1; i < j; i++, j--)
  { c = s[ i], s[ i] = s[ j], s[ j] = c;
  }

  return;
}
```

In diesem etwas komplexeren Beispiel wird eine Zeichenkette, welche in einem **Feld** abgespeichert ist, umgekehrt, wobei in einer ***for*** - Schleife das erste und das letzte Zeichen, das zweite und das vorletzte Zeichen usw. vertauscht werden.

Beispiel
Euklidischer Algorithmus zur Berechnung des größten gemeinsamen Teilers:

Zunächst berechnet man den Rest der ganzzahligen Division. Anschließend werden der Dividend und der Divisor neu vereinbart.

```
unsigned int a = 53667, b = 25527, c;

c = a % b;  /* Rest der ganzzahligen Division */
a = b; b = c;        /* Vertauschen der Werte */
```

Die letzten beiden Anweisungen müssen so lange wiederholt werden, bis der Rest der ganzzahligen Division den Wert 0 liefert. Um Wiederholungen zu programmieren, benötigt man spezielle Anweisungen, welche im nächsten Kapitel behandelt werden sollen.

3.8 C-Operatoren (Übersicht)

(mit Rangfolge und Assoziativitätsrichtung, s. auch Anhang B):

15	**( )** **[ ]** **-> .**	Ausdrucksgruppierung (Tiefe max. 32) Auswahl der Feldkomponenten Auswahl der Strukturkomponenten	⇒
14	**! ~** **++ --** **+ -** **(** *Typ* **)** **& *** **sizeof(** *Typ* **)**	Negation (logisch, bitweise) Inkrementation, Dekrementation (Präfix oder Postfix) Vorzeichen Typumwandlung Adressbildung, Dereferenzierung Bestimmung des Speicherbedarfs	⇐
13	*** /** **%**	Multiplikation, Division Rest bei ganzzahliger Division	⇒
12	**+ -**	Summe, Differenz	⇒
11	**<< >>**	bitweise Verschiebung nach links, rechts	⇒
10	**< <=** **> >=**	Vergleich auf kleiner, kleiner oder gleich Vergleich auf größer, größer oder gleich	⇒
9	**== !=**	Vergleich auf gleich, ungleich	⇒
8	**&**	Und (bitweise)	⇒
7	^	exklusives Oder (bitweise)	⇒
6	|	inklusives Oder (bitweise)	⇒
5	**&&**	Und (logisch)	⇒
4	||	inklusives Oder (logisch)	⇒
3	**? :**	bedingte Auswertung (paarweise)	⇐
2	**=** **°=**	Wertzuweisung zusammengesetzte Wertzuweisung (*=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=)	⇐
1	**,**	sequentielle Auswertung	⇒