

## 5. Operatoren

Prinzipiell unterscheidet man bei Operatoren unäre- und binäre Operatoren. Welcher Typ vorliegt kommt darauf an auf wie viele Operanden ein Operator wirkt.

### 5.5. Arithmetische Operatoren

Anders als bei vielen anderen Programmiersprachen stellt die Wertzuweisung auch einen Operator dar. Sein Wert ist das Ergebnis der Wertzuweisung. Daher kann eine Wertzuweisung überall dort stehen, wo ein Operator stehen darf. Das führt dann zu Konstruktionen wie:

```
if (z = x + y) { .... }
```

`z` wird der Wert der Addition von `x` und `y` zugewiesen. Gleichzeitig ergibt der Zuweisungsoperator einen Wert, der von der `if` Anweisung ausgewertet werden kann. Ist `x + y != 0`, dann ist das Ergebnis wahr und der Anweisungsblock wird ausgeführt. Ob die obige Konstruktion unbedingt eleganter und lesbarer als

```
z = x + y;  
if (z) { ... }
```

ist, mag dahingestellt sein.

Da die Zuweisung ein Operator ist, sind auch Mehrfachzuweisungen wie z. B. `a = b = c = 12` möglich. Die Zuweisung bindet schwächer als andere Operatoren wie z.B. arithmetische oder Vergleichs-Operatoren. Daher sollte man Zuweisungen, die in Ausdrücken stehen, zur Sicherheit klammern. Erst durch den abschließenden Strichpunkt wird der Zuweisungsoperator zu einer Anweisung.

Es gibt sechs arithmetische Operatoren. Sie stehen für die vier Grundrechenarten (+ - \* /), die Modulofunktion (Rest nach Division; %) und die arithmetische Negation.

- Vorzeichen (Unär)
- \* Multiplikation
- / Division
- % Modulofunktion
- + Addition
- Subtraktion

Die Modulofunktion ist nicht auf `float` oder `double` Operanden anwendbar. Die Rangfolge ist wie gewohnt: \*, / und % gleich und höher als + und -. Das Minuszeichen als unärer Operator (Vorzeichen) hat einen höheren Vorrang als \*, / und %. Es gibt kein positives Vorzeichen.

Beispiele für die Anwendung arithmetischer Operatoren:

```
int h, i=3, j=5, k=10;
float x, y=5.0, z=3.0;
h = i+j+k;           /* 18 */
h = k-j-i;           /* 2 */
h = i-k;             /* -7 */
h = k/i;             /* 3 */
h = k/i+j;           /* 8 */
h = k/(i+j);         /* 1 */
h = k*i+j;           /* 35 */
h = k*(i+j);         /* 80 */
h = k%i;             /* 1 */
x = y+z;             /* 8. */
x = y/z;             /* 1.666... */
x = y*z;             /* 15. */
x = k/z;             /* 3.333... */
x = k/z + y/z;       /* 5. */
x = k/i + y/i;       /* 4.666... */
x = k%z;             /* verboten! */
```

Höhere Rechenoperationen wie Wurzelziehen, Potenzieren, Exponentiation usw. fehlen in C als Operator ganz. Diese sind über Funktionsbibliotheken realisiert. Bei ihrer Verwendung muss in jedem Fall die Includedatei „math.h“ mit eingebunden werden.

## 5.6. Inkrement- und Dekrementoperatoren

C kennt spezielle Operatoren zum Inkrementieren und Dekrementieren. Es sind dies:

**++** Inkrement

**--** Dekrement

Sie sind wie der Vorzeichenoperator - rechts assoziativ, werden also von rechts her zusammengefasst. Sie können vor oder nach ihrem Operanden stehen. Im ersten Fall wird der Operand zunächst in- oder dekrementiert und dann weiterverwendet, im zweiten Fall wird der Operand erst verwendet und dann in- oder dekrementiert.

```
int i;
int k;
i=1;
if (++i > 1)           /* wird durchlaufen */
    k = 5;
if (i++ > 2)           /* wird nicht durchlaufen */
    k = 10;
```

## 5.7. Logische Verknüpfungen

Unter die logischen Verknüpfungen für Wahrheitswerte fallen die Operatoren:

! logische Negation

&& logisches UND

|| logisches ODER

Sie sind hier in ihrer Rangfolge geordnet aufgeführt, logisches UND hat also einen höheren Vorrang als das logische ODER. In Ausdrücken erfolgt die Abarbeitung von links nach rechts aber nur solange, bis das Ergebnis eindeutig feststeht. Das führt zu einer kürzeren Ausführungszeit, wenn man die häufigste verwendete Bedingung an den Anfang einer Bedingungsabfrage stellt. Bei Operationen, die Nebeneffekte haben können z.B. Inkrementation, muss man allerdings vorsichtig sein.

```
if (a<b      &&  (d=(a+b)) != c)    /* diese 2 Zeilen */
if ((d=(a+b)) != c &&  a<b)          /* wirken nicht gleich */
Beispiel:
#include <stdio.h>

int main (void)
{
    enum logik { false = 0, true = 1} a, b;

    printf ("/ a b | a&&b | a||b |\n");
    printf ("/-----|-----|-----|\n");
    for (a = false; a <= true; a++)
        for (b = false; b <= true; b++)
            printf ("/ %d %d |    %d |    %d  |\n",a, b, (a && b), (a
|| b));
    return 0;
}
```

Ergebnis:

a	b	a&&b	a  b
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

## 5.8. Bitmanipulationen

C stellt auch Operatoren für Bitmanipulationen zur Verfügung. Die Verknüpfung der beiden Operanden erfolgt bitweise. Sie können nicht auf Variablen oder Konstanten vom Typ float oder double angewendet werden. Die Operatoren sind:

~ Komplement

<< Linksshift

>> Rechtsshift

& bitweises UND

^     bitweises EXKLUSIVES ODER

|     bitweises ODER

```
int i=7;
int j=9;
int k;
k = i & j;           /* k=1 */
k = i | j;           /* k=15 */
k = i ^ j;           /* k=14 */
k = ~0;              /* k=max(unsigned int) */
k = ~i;              /* k=max(unsigned int) -7 */
k = i << 3;          /* k=56 */
k = i >> 3;          /* k=0 */
k = i & 0x03;        /* k=3; alle Bits bis auf die beiden */
                     /* unteren ausmaskiert */
```

Es werden teilweise die gleichen Operatorzeichen für verschiedene Operatoren verwendet.

Operator- zeichen	1. Bedeutung	2. Bedeutung
()	Klammerung	typedef
*	Pointerdeklaration	Multiplikation
-	Vorzeichen	Subtraktion
&	Adresse von	bitweises UND

Inkrement- und Dekrementoperatoren haben einen sehr hohen Vorrang  
die Arithmetikoperatoren haben einen Vorrang wie erwartet und gewohnt  
Shifts haben einen höheren Vorrang als Vergleiche, im Gegensatz zu den anderen  
Bitmanipulationen, die nach den Vergleichen stehen.

bitweise logische Operationen stehen vor den logischen Operationen, die sich auf  
ganze Zahlen beziehen.

die bedingte Bewertung hat einen sehr niedrigen Vorrang, trotzdem sollte man sich  
bei ihrer Anwendung eine Klammerung angewöhnen, damit die Anweisung leichter  
lesbar ist.

außer durch den Vorrang ist die Reihenfolge der Bewertung undefiniert:  $z = (a*2) + ++a$   
kann als  $(a*2) + ++a$  aber auch als  $++a + (a*2)$  bewertet werden. Seiteneffekt!  
ebenso ist die Reihenfolge der Bewertung von Funktionsparametern nicht  
gewährleistet:

```
i=1;
printf("%d %d", ++i, i*3) /* kann 2,6 oder 4,3 ausgeben */
```

ebenfalls nicht eindeutig definiert ist folgende Konstruktion:  $a[i]=i++$ ;

Dagegen bedeutet  $x---y$ :  $x-- - y$  und nicht  $x - --y$ . Der Compiler führt seine lexikalische  
Analyse immer von links nach rechts aus und nimmt dabei so viele Zeichen, wie er  
nur kann, um ein Token zu bilden.

## 5.9. Logische Datentypen und Operatoren, Vergleiche

Für Vergleiche stehen folgende Operatoren zur Verfügung:

< kleiner  
<= kleiner gleich  
> größer  
>= größer gleich  
== gleich (identisch)  
!= ungleich

Die ersten vier haben untereinander gleichen Rang, stehen aber eine Rangstufe höher als die beiden letzten. Es gibt in C grundsätzlich keinen Datentyp BOOLEAN; WAHR oder FALSCH werden einfach über den numerischen Integer-Wert entschieden. Dabei gilt:

ungleich 0 WAHR (erhält den Wert eins)

gleich 0 FALSCH (Wert 0)

Arbeitet man in einem Programm viel mit diesen Werten, kann man folgende Konstantendefinitionen dazu benutzen:

```
#define FALSE 0
#define TRUE !FALSE
```

Beispiele für die Verwendung dieser Operatoren:

```
int a=5;
int b=12;
int c=7;
a < b           /* WAHR */
b < c           /* FALSCH */
a+c <= b        /* WAHR */
b-a >= c         /* WAHR */
a == b          /* FALSCH */
a+c == b        /* WAHR */
a != b          /* WAHR */
a = b<c;         /* möglich: a=0 */
a = c<b;         /* -"- a=1 */
```

Wie schon erwähnt, bindet die Zuweisung schwächer als andere Operatoren (arithmetische oder Vergleichs-Operatoren). Daher sollte man Zuweisungen, die in Ausdrücken stehen, zur Sicherheit klammern. Dazu einige Beispiele:

```
if (x = y)
    tuwas();
```

Die Anweisung besagt nicht: "Wenn x gleich y ist, dann tuwas();", sondern "x erhalte den Wert y; Wenn das Ergebnis der Operation (x = y) ungleich Null ist, dann tuwas();". Es gibt also zwei Möglichkeiten:  
Es soll wirklich x mit y verglichen werden. Dann darf nicht "=" stehen, sondern der Vergleichsoperator "==". Solche Flüchtigkeitsfehler kommen oft vor.

```
if (x == y) /* Vergleich x == y (doppeltes Gleichheitszeichen)
*/
    tuwas();
```

Es soll tatsächlich die Zuweisung x = y erfolgen und abhängig vom Wert der Variablen x bzw. y verzweigt werden. Dann ist zwar if (x = y) richtig, aber der Deutlichkeit halber sollte man schreiben:

```
if ((x = y) != 0)
    tuwas();
```

Die Klammerung von (x = y) ist notwendig, da der Vergleichsoperator stärker bindet als die Zuweisung. Fehlt die Klammer, wäre das schon wieder ein schwer zu findender Fehler. x = y != 0 bedeutet: x = (y != 0). x hätte nachher nicht den Wert von y, sondern 0 oder 1.

## 5.10. Zusammengesetzte Operatoren, implizite Typumwandlung

### Zusammengesetzte Operatoren

Eine Spezialität von C ist die abgekürzte Schreibweise für bestimmte Zuweisungen. Diese abgekürzte Schreibweise ist vorteilhaft, wenn die linke Seite eine komplizierte Struktur hat (weniger Tippfehler, bessere Lesbarkeit). Sie ist aber für den Anfänger schwerer zu lesen. Bei der abgekürzten Schreibweise gilt folgendes:

Ausdruck1 op= Ausdruck2

ist äquivalent zu (beachten Sie die Klammern!):

Ausdruck1 = (Ausdruck1) op (Ausdruck2)

op kann einer der Operatoren: + - \* / % << >> & ^ oder | sein. Die Klammern sind sehr wichtig, damit keine unerwünschten Nebeneffekte auftreten:

```
i *= k+1;      /* wird wie */
i = i * (k+1); /* behandelt und nicht wie */
i = i * k + 1;
```

Der Vorteil der abgekürzten Schreibweise wird an folgendem Beispiel deutlich:

```
feld[i*3+j-7] = feld[i*3+j-7] + 10;      /* normal */
```

```
feld[i*3+j-7] += 10;
```

```
/* abgekürzt */
```

Zusammengesetzte Zuweisungsoperatoren:

`+=`      `x += ausdr`    `<-- x = x + ausdr`

`-=`      `x -= ausdr`    `<-- x = x - ausdr`

`*=`      `x *= ausdr`    `<-- x = x * ausdr`

`/=`      `x /= ausdr`    `<-- x = x / ausdr`

`%=`      `x %= ausdr`    `<-- x = x % ausdr`

`&=`      `x &= ausdr`    `<-- x = x & ausdr`

`|=`      `x |= ausdr`    `<-- x = x | ausdr`

`^=`      `x ^= ausdr`    `<-- x = x ^ ausdr`

`<<=`     `x <<= ausdr`   `<-- x = x << ausdr`

`<B>>=`   `x >>= ausdr`   `<-- x = x >> ausdr`

### Implizite Typumwandlung

Datentypwandlungen sind immer dann notwendig, wenn zwei Operanden in einem Ausdruck verschiedene Typen haben. Die Datentypwandlungen werden soweit notwendig implizit durchgeführt. Es gelten folgende Regeln:

`char` wird bei Bewertungen immer in `int` gewandelt, dadurch sind `int` und `char` beliebig mischbar. Allerdings kann die `char-int` Wandlung rechnerabhängig sein, `char` kann auf die Zahlenwerte -128 bis +127 oder 0 bis +255 abgebildet sein, die "normalen" Zeichen liegen aber immer im positiven Bereich. Die `char-int` Wandlung ist vor allem bei der Abfrage eines eingelesenen Zeichens auf EOF wichtig (Eingelesen werden immer `int` Werte, die dann aber einfach `char` Werten zugewiesen werden können). die Wandlung von `int` nach `long` erfolgt durch Vorzeichenerweiterung (bei signed) oder Voranstellen von Nullen (unsigned), bei der umgekehrten Wandlung werden die höchstwertigen Bits einfach abgeschnitten.

`float` wird bei Bewertungen immer in `double` gewandelt, alle Rechnungen erfolgen daher mit derselben großen Genauigkeit. Die Wandlung von `int` nach `float` ist wohldefiniert, die umgekehrte Richtung kann rechnerabhängig sein, insbesondere bei negativen Gleitkommazahlen.

Zusammenfassung:

Variable	=	Variable	durchgeführte Typumwandlung
<code>int</code> <code>int</code>	<code>&lt;-</code> <code>-</code> <code>&lt;-</code> <code>-</code>	<code>float</code> <code>double</code>	Weglassen des gebrochenen Anteils
<code>int</code> <code>char</code> <code>char</code>	<code>&lt;-</code> <code>-</code> <code>&lt;-</code> <code>-</code>	<code>long</code> <code>int</code> <code>short</code>	Weglassen der höherwertigen Bits





```
int j = 12, k = 18;  
float f;  
...  
f = k/j;    /* in f steht 1.0 */
```

Beide Operanden der Division  $k/j$  sind int-Variablen. Also ist auch das Ergebnis der Division vom Typ int ( $18/12 = 1$  Rest 6). Die Zuweisung erfolgt nach der Division. Mit der impliziten Typkonversion erfolgt dann die Expansion von 1 zu 1.0. Sollte das "echte" Ergebnis gewünscht werden, so wird dies durch eine explizite Typumwandlung möglich.

Die explizite Typumwandlung (type cast) erfolgt mittels des Cast-Operators, der unmittelbar vor den Ausdruck zu setzen ist, dessen Typ konvertiert werden soll. Der Cast-Operator ist ganz einfach eine in Klammern gesetzte Typangabe. Obiges Beispiel wird geändert zu:

```
int j = 12, k = 18;  
float f;  
...  
f = (float)k/j;
```

Die Variable  $k$  wird nun zu einer float-Variablen typkonvertiert (Neudeutsch: "gecastet"). Bei der Division von float durch int wird  $j$  implizit auf float typkonvertiert und so ergibt sich das korrekte Ergebnis 1.5.