

9 Ausdrücke, Anweisungen und Operatoren



In diesem Kapitel wird auf das Thema der Anweisungen und Ausdrücke sowie noch einmal detailliert auf die in C vorhandenen Operatoren eingegangen. Diese drei Themengebiete sind eng miteinander verknüpft.

In den ersten Unterkapiteln werden die in C gültigen Regeln für das Auswerten von Ausdrücken besprochen, sowie Möglichkeiten der Klassifizierung wie beispielsweise Rangordnung und Assoziativität. Außerdem wird auf Eigenheiten der Programmiersprache eingegangen wie Nebeneffekte, L-Werte und R-Werte.

Daraufhin werden die Operatoren einzeln mit Beispielen aufgelistet. Sie werden dabei nach ihrer Wirkungsweise klassifiziert, wie etwa arithmetische Operatoren, logische Operatoren, Zuweisungs-Operatoren oder Vergleichs-Operatoren.

9.1 Ausdrücke und Anweisungen

Im Quellcode einer C-Programmdatei werden grundsätzlich **Anweisungen** geschrieben. Sie werden vom Compiler in Prozessor-Befehle umgewandelt, die der Reihe nach vom Prozessor abgearbeitet werden.

Man unterscheidet zwischen zwei Arten von Anweisungen: Kontrollstrukturen und Ausdrucksanweisungen. Kontrollstrukturen werden in Kapitel → 10 behandelt.

Eine **Ausdrucksanweisung** besteht – wie der Name schon sagt – aus einem Ausdruck.

Ein **Ausdruck** ist eine Aneinanderreihung von Operanden und Operatoren. Jeder Ausdruck hat einen Rückgabewert und kann Teil eines größeren Ausdrucks sein.



Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann https://doi.org/10.1007/978-3-658-45209-4_9.

Folgendes sind Beispiele von Ausdrücken:

```
x
7
5 * 5
(a + b) * (c - d)
sin(PI / 2.)
i++
i = 0
printf("Hallo")
```

Ein Ausdruck ist in C im einfachsten Fall der Bezeichner (Name) einer Variablen oder einer Funktion, eine Konstante oder ein String. Meist interessiert der Wert eines Ausdrucks.



Werden alle Operanden eines Ausdrucks zu einer Konstanten aus, so handelt es sich um einen sogenannten „**konstanten Ausdruck**“ (auf Englisch „**const expression**“). Ein solcher Ausdruck kann vom Compiler bereits zur Compilezeit aufgelöst werden und benötigt somit keinerlei Laufzeit. Im Standard C23 wird zur Markierung eines konstanten Ausdrucks das Schlüsselwort `constexpr` eingeführt.



So hat eine Konstante einen Wert, eine Variable kann einen Wert liefern, ebenso wie der Aufruf einer Funktion.

Der Wert eines Ausdrucks wird auch als sein **Rückgabewert** bezeichnet. Jeder Rückgabewert hat einen Typ. Ausdrücke können Werte an Speicherstellen verändern, was als „Nebeneffekt“ bezeichnet wird.



Durch Verknüpfungen von Operanden durch Operatoren und gegebenenfalls auch runde Klammern entstehen komplexe Ausdrücke. Runde Klammern beeinflussen dabei die Auswertungsreihenfolge. Jeder Operand kann selbst auch ein Ausdruck sein.



Die Ziele der Verknüpfungen von Operatoren und Operanden zu Ausdrücken sind:

- Die Berechnung neuer Werte. Alles, was als Verknüpfung von Operatoren geschrieben werden kann, hat einen Wert und stellt einen Ausdruck dar.
- Das Erzeugen von gewollten Nebeneffekten.
- Ein Speicherobjekt, also eine Variable oder eine Funktion zu erhalten.

In C kann jeder Ausdruck eine Anweisung werden, indem ihr ein Semikolon angehängt wird.



Anweisungen haben keinen Rückgabewert. Sie stellen ein abgeschlossenes Element dar. Das bedeutet, dass der Rückgabewert des Ausdrucks der Anweisung nicht weiter behandelt wird. Folgendes sind Beispiele von Ausdrucksanweisungen:

```
5 * 5;  
i = 0;  
printf("Hallo");  
ang = sin(2 * PI) / 4.;
```

Dabei ist zu beachten, dass eine Ausdrucksanweisung wie `5 * 5` zwar erlaubt ist, jedoch keine Wirkung hat. Ausdrucksanweisungen sind lediglich dann sinnvoll, wenn der Ausdruck einen sogenannten „Nebeneffekt“ hat.

9.1.1 Nebeneffekte

Nebeneffekte werden auch als Seiteneffekte oder als Nebenwirkungen bezeichnet und bedeuten nichts anderes als: Speicherinhalte werden verändert.



Ein Ausdruck wie $5 * 5$ gibt zwar einen Wert zurück, verändert jedoch keine Speicherinhalte. Einen solchen Ausdruck in eine Anweisung umzuwandeln ist nutzlos, da er nichts bewirkt.

Diejenigen Ausdrücke, welche offensichtlich Nebeneffekte aufweisen, sind die Zuweisungen, welche mit dem Zuweisungs-Operator `=` geschrieben werden:

```
i = 1;
```

Hier ist eindeutig klar, dass der Inhalt (der Speicherinhalt) der Variablen `i` mit dem Wert 1 überschrieben wird. Dies ist der Nebeneffekt dieser Anweisung.

In der Programmiersprache C gibt es jedoch Operatoren, die nicht offensichtlich Nebeneffekte hervorrufen. Sie wurden definiert, um eine schnelle und kurze Programmierschreibweise zu erlauben. Es ist nämlich möglich, während der Auswertung eines Ausdrucks Programmvariablen nebenbei zu verändern. Ein Beispiel hierzu ist:

```
i++;
```

Der Operator `++` ist der sogenannte Inkrement-Operator. Er gibt hier grundsätzlich den Wert der Variablen `i` zurück, erhöht als Nebeneffekt jedoch zusätzlich die Variable `i` um 1.

Dann gibt es auch noch den Dekrement-Operator `--`, welcher die Variable um 1 verringert. Die beiden Operatoren `++` und `--` haben eine lange Tradition in den C-Sprachen und werden in diesem Kapitel noch an einigen Stellen erwähnt werden. Sie gelten jedoch aufgrund dessen, dass sie diese Nebeneffekte aufweisen, als nicht unproblematisch.

9.1.2 Sequenzpunkte bei Nebeneffekten



Nebeneffekte sind grundsätzlich notwendig, um überhaupt irgendetwas mit einem Programm zu bewirken, da ansonsten keinerlei Speicherinhalte verändert werden können. Wenn jedoch in einem Ausdruck Speicherinhalte gleichzeitig gelesen wie auch geschrieben werden, so kann dies unter gewissen Umständen zu Problemen führen, wie das folgende Beispiel zeigt:

```
n++ - n
```

Hier könnte der Compiler erst den linken Operanden auswerten und dessen Nebeneffekt durchführen. Dann ist der Wert des gesamten Ausdrucks -1 . Wird jedoch erst der rechte Operand ausgewertet, so ist der Wert des gesamten Ausdrucks gleich 0 .

Auch im folgenden Beispiel ist das Ergebnis nicht definiert, da beide Argumente einen Nebeneffekt haben.

```
a = f1(par++, par++);
```

Zwar wird gemäß Standard verlangt, dass alle Nebeneffekte vor dem eigentlichen Aufruf der Funktion abgearbeitet sein müssen, aber die Reihenfolge, in welcher die Argumente ausgewertet werden, ist Sache des Compilers. Es ist somit nicht klar, welche Werte tatsächlich an die Funktion übergeben werden.

Problematisch wird es auch, wenn zwei Funktionen $f1()$ und $f2()$ auf dieselben globalen Variablen zugreifen und diese verändern, also einen Nebeneffekt haben. Das Ergebnis des folgenden Ausdrucks ist dann ebenfalls nicht definiert, da der Standard nicht festlegt, welche der beiden Funktionen $f1()$ oder $f2()$ zuerst ausgewertet wird.

```
a = f1() * f2();
```

Um Nebeneffekte beim Programmieren unter Kontrolle zu halten, wurden im ISO-Standard sogenannte „**Sequenzpunkte**“ (auf Englisch „**sequence points**“) definiert.

An einem Sequenzpunkt müssen alle Nebeneffekte einer vorangegangenen Berechnungen durchgeführt worden sein und es dürfen noch keine Nebeneffekte von Ausdrücken stattgefunden haben, die im Programm nach dem Sequenzpunkt stehen.



Sequenzpunkte liegen an folgenden Stellen vor:

- Nach der Berechnung der Argumente und des Funktions-Bezeichners, bevor eine Funktion aufgerufen wird.
- Bei den folgenden Operatoren jeweils nach der Auswertung des ersten Operanden und vor der Auswertung des zweiten Operanden (gemäß der Abarbeitungsrichtung des Operators):
 - Logisches UND &&
 - Logisches ODER ||
 - Bedingungs-Operator, also der Bedingung $a \text{ in } a ? b : c$
 - Komma-Operator ,
- Am Ende der Auswertung der folgenden Ausdrücke:
 - Initialisierungsausdruck einer manuellen Initialisierung
 - Ausdruck in einer Ausdrucksanweisung
 - Bedingung in einer if-Anweisung
 - Selektionsausdruck in einer switch-Anweisung
 - Bedingung einer while- oder do while-Schleife
 - Alle drei Ausdrücke einer for-Anweisung
 - Ausdruck einer return-Anweisung

Aufgrund dieser Regeln ist undefiniertes Verhalten beim täglichen Programmieren äußerst selten anzutreffen. Dennoch gilt:

Wird ein Objekt zwischen zwei aufeinanderfolgenden Sequenzpunkten mehrmals gelesen, während es gleichzeitig verändert wird, so ist das Ergebnis nicht definiert.



Weitaus am verbreitetsten sind heutzutage versteckte Nebeneffekte bei der Parameterübergabe bei Funktionsaufrufen. Auf diese wird in Kapitel [→ 11.5.1](#) eingegangen.

9.1.3 L-Werte und R-Werte

Ausdrücke haben eine unterschiedliche Bedeutung, je nachdem, ob sie links oder rechts vom Zuweisungs-Operator stehen.

```
a = b
```

Hier beispielsweise steht der Ausdruck `b` auf der rechten Seite des Zuweisungs-Operators für einen Wert, während der Ausdruck `a` auf der linken Seite die Stelle angibt, an der dieser Wert zu speichern ist. Wenn wir dieses Beispiel noch etwas modifizieren, wird der Unterschied noch deutlicher:

```
a = a + 5
```

Der Variablenname `a`, der ja auch einen einfachen Ausdruck darstellt, wird hier in unterschiedlicher Bedeutung verwendet. Rechts vom Zuweisungs-Operator ist der Wert gemeint, der in der Speicherzelle `a` gespeichert ist, und links ist die Speicherzelle `a` gemeint, in der der Wert des Gesamtausdrucks der rechten Seite gespeichert werden soll.

Aus dieser Stellung links oder rechts des Zuweisungs-Operators wurden auch die Begriffe L-Wert und R-Wert abgeleitet.

Ein Ausdruck stellt einen **L-Wert** (**lvalue** oder **left value**) dar, wenn er sich auf ein Speicherobjekt bezieht. Steht ein Ausdruck links des Zuweisungs-Operators, so muss er ein L-Wert sein.



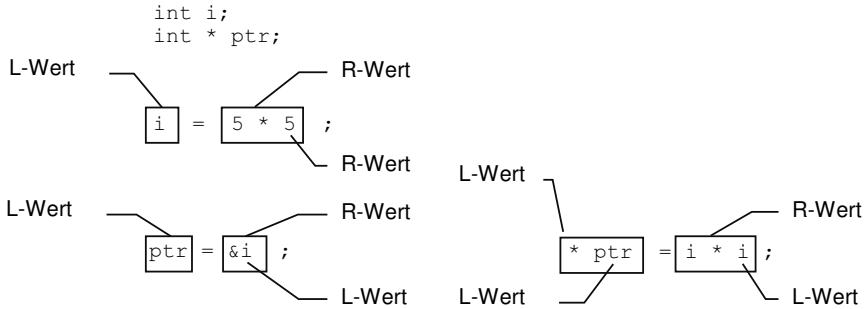
Ein Ausdruck, der keinen L-Wert darstellt, stellt einen **R-Wert** (**rvalue** oder **right value**) dar. Er bezieht sich nicht auf ein Speicherobjekt. Er darf nur rechts des Zuweisungs-Operators stehen.



Dabei ist zu beachten, dass ein Ausdruck, der einen L-Wert darstellt, auch rechts vom Zuweisungs-Operator stehen darf, er hat dann aber, wie oben erwähnt, eine andere Bedeutung. Steht ein L-Wert rechts vom Zuweisungs-Operator, so wird dessen Name beziehungsweise Adresse benötigt, um an der entsprechenden Speicherstelle den Wert der Variablen abzuholen. Links des Zuweisungs-Operators muss immer ein L-Wert stehen, da man den Namen beziehungsweise die Adresse einer Variablen braucht, um an der entsprechenden Speicherstelle den zuzuweisenden Wert abzulegen.

Damit einem L-Wert etwas zugewiesen werden kann, muss er jedoch zusätzlich noch modifizierbar sein. Ein nicht modifizierbarer L-Wert ist beispielsweise der Name eines Arrays. Ein Arrayname ist konstant und kann nicht modifiziert werden. Des Weiteren sind Variablen nicht modifizierbar, wenn sie einen unvollständigen Typen besitzen oder einen Typen mit dem Qualifikator `const`. Dies gilt insbesondere auch für Struktur- oder Union-Typen (siehe dazu Kapitel [→ 13.1](#) und [→ 13.3](#)), bei welchen eine seiner Komponenten – einschließlich aller rekursiv in einer Komponente enthaltenen Strukturen und Unionen – einen mit dem Qualifikator `const` versehenen Typ hat.

Auf der linken Seite einer Zuweisung darf also nur ein modifizierbarer L-Wert stehen, jedoch nicht ein R-Wert oder ein nicht modifizierbarer L-Wert. Das folgende Bild zeigt Beispiele für L- und R-Werte:



Bestimmte Operatoren können nur auf L-Werte angewandt werden. So kann man den Inkrement-Operator ++ oder den Adress-Operator & nur auf L-Werte anwenden.



5++ ist falsch, i++ korrekt, wobei i eine Variable darstellt.

Es ist zu beachten, dass ein Pointer zwar auf einen L-Wert zeigt, der Pointer selbst jedoch nicht zwingendermaßen als L-Wert verfügbar sein muss. Ein Pointer kann auch aus einem beliebigen Ausdruck entstehen, welcher zu einem R-Wert evaluiert.



```

int* pointer;
int alpha;
pointer = &alpha;
  
```

In diesem Beispiel ist &alpha ein R-Wert. (&alpha)++ ist nicht möglich, pointer++ hingegen schon. Demgegenüber ist jedoch sowohl *pointer = 2 als auch *&alpha = 2 zugelassen.

Folgende Operatoren erwarten stets einen L-Wert:

- Operand des Adress-Operators &
- Operand des Inkrement-Operators ++
- Operand des Dekrement-Operators --
- Der linke Operand des Punkt-Operators . bei Strukturen
- Der linke Operand des Zuweisungs-Operators =
- Der linke Operand des Funktionsaufruf-Operators ()

Folgende Operatoren geben einen L-Wert zurück:

- Rückgabewert des Punkt-Operators . bei Strukturen
- Rückgabewert des Pfeil-Operators -> bei Pointern
- Rückgabewert des Array-Element-Operators []
- Rückgabewert des Dereferenzierungs-Operators *

So wird im Ausdruck `&*pointer` beispielsweise der pointer zuerst dereferenziert, was einen L-Wert ergibt. Danach wird von diesem L-Wert gleich wieder die Adresse genommen.

9.2 Operatoren und Operanden

Die Sprache C kennt mehr als 30 **Operatoren**, von welchen die meisten in diesem Kapitel besprochen werden. Sie alle haben unterscheidbare Eigenschaften wie Priorität, die Anzahl an Operanden und Abarbeitungsrichtung (Assoziativität). Auf diese Eigenschaften wird in den folgenden Unterkapiteln eingegangen.

Folgende Tabelle gibt einen Überblick über die in C verfügbaren Operatoren:

Prio.	Operatoren		Operanden	Assoz.
1	() [] -> .	Funktionsaufruf Array-Index Komponentenzugriff	2	links
2	++ --	Postfix-Inkrement Postfix-Dekrement	1	links
3	! ~ ++ -- (Typname) sizeof + - * &	Negation logisch Negation bitweise Präfix-Inkrement Präfix-Dekrement Cast Byte-Größe Vorzeichen Dereferenzierung, Adresse	1	rechts
4	* / %	Multiplikation, Division Modulo	2	links
5	+ -	Summe, Differenz	2	links
6	<< >>	bitweises Schieben	2	links
7	< <= > >=	Kleiner, kleiner gleich Größer, größer gleich	2	links
8	== !=	Gleichheit, Ungleichheit	2	links
9	&	bitweises UND	2	links
10	^	bitweises Exklusiv-ODER	2	links
11		bitweises ODER	2	links
12	&&	logisches UND	2	links
13		logisches ODER	2	links
14	?:	bedingte Auswertung	3	rechts
15	= += -= *= /= %= &= ^= = <<= >>=	Wertzuweisungen	2	rechts
16	,	Komma-Operator	2	links

9.2.1 Anzahl Operanden

In C gibt es einstellige (unäre, monadische) Operatoren, zweistellige (binäre, dyadische) Operatoren und einen einzigen dreistelligen (ternären, triadischen) Operator.

Ein einstelliger (unärer) Operator hat einen einzigen Operanden.

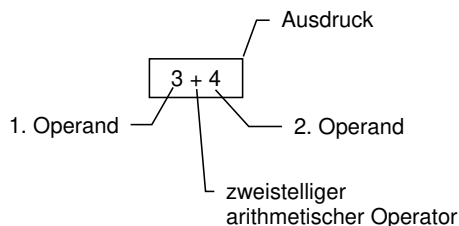


Ein Beispiel hierfür ist der Minus-Operator als Vorzeichen-Operator, der auf einen einzigen Operanden wirkt und das Vorzeichen dessen Wertes ändert. So ist in `-3` das `-` ein Vorzeichen-Operator, der auf die positive Konstante 3 angewandt wird.

Benötigt ein Operator zwei Operanden für die Verknüpfung, so spricht man von einem zweistelligen (binären) Operator.



Ein vertrautes Beispiel für einen binären Operator ist der Additions-Operator, der hier zur Addition der beiden Zahlen 3 und 4 verwendet werden soll:



Der einzige ternäre Operator ist der Bedingungs-Operator `?:` mit drei Operanden. Er wird in Kapitel [9.9.4](#) behandelt. Der erste Operand stellt eine Bedingung dar, gemäß welcher entweder der zweite oder dritte Operand ausgewertet wird:

```
(powerLevel > 9000) ? watchMeme() : doWork()
```

9.2.2 Postfix- und Präfix-Operatoren

Postfix-Operatoren sind unäre Operatoren, die hinter (post) ihrem Operanden stehen. **Präfix-Operatoren** sind unäre Operatoren, die vor (prä) ihrem Operanden stehen.



Der Ausdruck `i++` stellt die Anwendung des Postfix-Operators `++` auf seinen Operanden `i` dar.

Im Ausdruck `i++` bewirkt der Postfix-Operator `++`, dass die Variable `i` um 1 inkrementiert wird. Der Operator gibt jedoch den alten Wert von `i` zurück.



Im folgenden Beispiel wird durch den Postfix-Operator `++` als Nebeneffekt die Variable `i` um 1 erhöht, doch der Rückgabewert des Operators entspricht dem alten Wert von `i`. Erst ein erneutes Ansprechen der Variablen liefert den durch den Nebeneffekt errechneten und gespeicherten Wert:

```
int i = 3;
printf("%d", i++); // Ausgabe: 3
printf("%d", i);   // Ausgabe: 4
```

Genau andersrum funktioniert der Präfix-Operator `++`. Auch hier wird als Nebeneffekt die Variable `i` um 1 erhöht, doch wird nun der neue, sprich soeben berechnete Wert zurückgegeben:

```
int i = 3;
printf("%d", ++i); // Ausgabe: 4
printf("%d", i);   // Ausgabe: 4
```

Diese Unterscheidung ist jedoch nur bei den beiden Operatoren `++` und `--` von Wichtigkeit.

9.2.3 Auswertungsreihenfolge komplexer Ausdrücke

Hat man einen Ausdruck aus mehreren Operatoren und Operanden, so stellt sich die Frage, in welcher **Reihenfolge** die einzelnen Operatoren bei der Auswertung „dran kommen“.

Wie in der Mathematik spielt es bei C eine wichtige Rolle, in welcher Reihenfolge ein Ausdruck berechnet wird. So gilt beispielsweise auch in C die Regel „Punkt vor Strich“: Der Ausdruck $5 + 2 * 3$ ergibt somit 11 und nicht 21, da der Multiplikations-Operator vor dem Additions-Operator ausgeführt wird.

In C gibt es jedoch weitaus mehr Operatoren als nur die vier Grundoperationen. Daher muss genau festgelegt werden, welcher Operator im Zweifelsfall Priorität hat.

Es gelten die folgenden Regeln:

1. Wie in der Mathematik werden als erstes Teilausdrücke in **runden Klammern** ausgewertet.
2. Dann werden die Operatoren der Ausdrücke entsprechend festgelegter **Prioritäten** (auf Englisch „**operator precedence**“) abgearbeitet.
3. Wenn zwei Operatoren die gleiche Priorität besitzen, so wird mit der sogenannten „**Assoziativität**“ eine **Abarbeitungsrichtung** festgelegt, sprich, ob die Operatoren von rechts nach links oder von links nach rechts abgearbeitet werden.

Mithilfe der Tabelle zu Beginn dieses Kapitels ([→ 9.2](#)) kann man nun für jeden beliebigen Ausdruck bestimmen, in welcher Reihenfolge der Compiler die Operanden auswerten wird. In der Tabelle ist Priorität 1 die höchste Priorität. Beispielsweise ist in der Tabelle auch zu sehen, dass der Multiplikations-Operator wie in der Mathematik eine höhere Priorität als der Additions-Operator hat.

Als Beispiel zur Verdeutlichung der Vorgehensweise wird der folgende Ausdruck betrachtet.

```
*p++
```

Da dieser Ausdruck keine Klammern hat, wird bei der Auswertung dieses Ausdrucks nach Regel 2 verfahren: Der Inkrement-Operator ++ hat eine höhere Priorität als der Dereferenz-Operator *, also wird erst der Ausdruck p++ ausgewertet und danach der Operator * auf den Rückgabewert des Ausdrucks p++ angewandt.

Will man hingegen den Wert des Objektes *p durch den Postfix-Operator ++ erhöhen, so kann man mit Regel 1 diese Abarbeitungsreihenfolge erzwingen, indem man einfach Klammern um den gewünschten Ausdruck setzt: (*p)++.

*p++ ist gleichbedeutend mit *(p++) und nicht mit (*p)++.



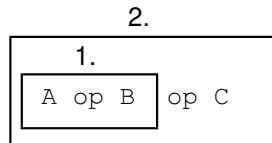
Haben zwei Operatoren dieselbe Priorität, so kommt die Regel 3 zum Zuge: Die Einhaltung der Assoziativität (Abarbeitungsrichtung).

Unter Assoziativität versteht man die Reihenfolge, wie Operatoren und Operanden verknüpft werden, wenn Operanden mit Operatoren gleicher Priorität (Vorrangstufe) miteinander verkettet sind.



Wiederum kann die Assoziativität von Operatoren in der Tabelle in Kapitel → 9.2 nachgelesen werden.

Ist ein Operator in C rechtsassoziativ, so wird eine Verkettung von Operatoren dieser Art von rechts nach links abgearbeitet, bei Linksassoziativität dementsprechend von links nach rechts. Das folgende Bild symbolisiert die Verknüpfungsreihenfolge bei einem linksassoziativen Operator op:



Es wird also zuerst der linke Operator op auf die Operanden A und B angewandt, und erst als zweites wird dann die Verknüpfung op mit C durchgeführt.

Als Beispiel dienen hier der Additions- und Subtraktions-Operator. Beide haben dieselbe Priorität:

$a - b + c$

Da beide Operatoren linksassoziativ sind, wird dieser Ausdruck wie $(a - b) + c$ verknüpft und nicht wie $a - (b + c)$.

Beachten Sie aber bitte, dass die Reihenfolge der Verknüpfung durch Operatoren nicht mit der Reihenfolge der Auswertung der Operanden übereinstimmen muss. Siehe dazu Kapitel [→ 9.1.2](#) .



Generell empfiehlt sich der Einsatz von Klammern, wenn man unsicher über die Priorität oder Assoziativität von Operatoren ist.

9.3 Einstellige arithmetische Operatoren

Im Folgenden werden die einstelligen (unären) Operatoren mit ihren Operanden anhand von Beispielen vorgestellt.

positiver Vorzeichen-Operator	<code>+x</code>
negativer Vorzeichen-Operator	<code>-x</code>
Postfix-Inkrement-Operator	<code>x++</code>
Präfix-Inkrement-Operator	<code>++x</code>
Postfix-Dekrement-Operator	<code>x--</code>
Präfix-Dekrement-Operator	<code>--x</code>

9.3.1 Positiver Vorzeichen-Operator: `+x`

Der positive Vorzeichen-Operator wird selten verwendet, da er lediglich den Wert seines Operanden wiedergibt. Es gibt keine Nebeneffekte.

`+a`

`+a` hat denselben Rückgabewert wie `a`.

9.3.2 Negativer Vorzeichen-Operator: `-x`

Will man den Wert eines Operanden mit umgekehrtem Vorzeichen erhalten, so ist der negative Vorzeichen-Operator von Bedeutung. Es gibt keine Nebeneffekte.

`-a`

`-a` hat denselben Betrag wie `a`, aber das umgekehrte Vorzeichen.

9.3.3 Postfix-Inkrement-Operator: x++

Der Rückgabewert ist der unveränderte Wert des Operanden. Als Nebeneffekt wird der Wert des Operanden um 1 inkrementiert. Bei Pointern wird um eine Objektgröße vom Typ, auf den der Pointer zeigt, inkrementiert. Der Inkrement-Operator kann nur auf modifizierbare L-Werte angewandt werden.

```
a = 1;  
b = a++;          // Ergebnis: b = 1, Nebeneffekt: a = 2  
ptr++;
```

9.3.4 Präfix-Inkrement-Operator: ++x

Der Rückgabewert ist der um 1 inkrementierte Wert des Operanden. Als Nebeneffekt wird der Wert des Operanden um 1 inkrementiert. Bei Pointern wird um eine Objektgröße vom Typ, auf den der Pointer zeigt, inkrementiert. Der Inkrement-Operator kann nur auf modifizierbare L-Werte angewandt werden.

```
a = 1;  
b = ++a;          // Ergebnis: b = 2, Nebeneffekt: a = 2  
++ptr;
```

9.3.5 Postfix-Dekrement-Operator: x--

Der Rückgabewert ist der unveränderte Wert des Operanden. Als Nebeneffekt wird der Wert des Operanden um 1 dekrementiert. Bei Pointern wird um eine Objektgröße des Typs, auf den der Pointer zeigt, dekrementiert. Der Dekrement-Operator kann nur auf modifizierbare L-Werte angewandt werden.

```
a = 1;  
b = a--;          // Ergebnis: b = 1, Nebeneffekt: a = 0  
ptr--;
```

9.3.6 Präfix-Dekrement-Operator: --x

Der Rückgabewert ist der um 1 dekrementierte Wert des Operanden. Als Nebeneffekt wird der Wert des Operanden um 1 dekrementiert. Bei Pointern wird um eine Objektgröße vom Typ, auf den der Pointer zeigt, dekrementiert. Der Dekrement-Operator kann nur auf modifizierbare L-Werte angewandt werden.

```
a = 1;  
b = --a;           // Ergebnis: b = 0, Nebeneffekt: a = 0  
--ptr;
```

9.4 Zweistellige arithmetische Operatoren

Im Folgenden werden die zweistelligen (binären) Operatoren mit ihren Operanden anhand von Beispielen vorgestellt. Zweistellige arithmetische Operatoren haben keine Nebeneffekte.

Es gelten die „üblichen“ Rechenregeln, also Klammerung vor Punkt und Punkt vor Strich. Der Multiplikations-, Divisions- und Modulo-Operator haben eine höhere Priorität wie der Additions- und Subtraktions-Operator.

Additions-Operator	$x + y$
Subtraktions-Operator	$x - y$
Multiplikations-Operator	$x * y$
Divisions-Operator	x / y
Restwert-Operator	$x \% y$

9.4.1 Additions-Operator: $x + y$

Wendet man den zweistelligen Additions-Operator auf seine Operanden an, so ist der Rückgabewert die Summe der Werte der beiden Operanden.

```
6 + (4 + 3)
a + 1.2e6
PI + 1
```

9.4.2 Subtraktions-Operator: $x - y$

Wendet man den zweistelligen Subtraktions-Operator auf die Operanden x und y an, so ist der Rückgabewert die Differenz der Werte der beiden Operanden.

```
6 - 4
7 - 2.3e4
PI - KONSTANTE_A
```

9.4.3 Multiplikations-Operator: $x * y$

Es wird die Multiplikation des Wertes von x mit dem Wert von y durchgeführt.

```
3 * 5 + 3      // Ergebnis: 18
3 * (5 + 3)    // Ergebnis: 24
```

9.4.4 Divisions-Operator: x / y

Der Divisions-Operator dividiert zwei Zahlen. Für Gleitpunkt-Zahlen entspricht dies der normalen mathematischen Division. Das Resultat einer solchen Operation ist wiederum eine Gleitpunkt-Zahl.

Bei der Verwendung des Divisions-Operator mit Integer-Operanden ist das Ergebnis wieder ein Integer. Der Nachkommateil des Ergebnisses wird abgeschnitten.

Eine Division durch 0 ist bei `int`-Zahlen nicht erlaubt.

Ist mindestens ein Operand eine double- oder float-Zahl, so ist das Ergebnis eine Gleitpunkt-Zahl.

```
5 / 5          // Ergebnis: 1
5 / 3          // Ergebnis: 1
5 / 0          // dieser Ausdruck ist nicht zulaessig
52.0 / 5.8     // Ergebnis: 8.9655...
11.0 / 5       // Ergebnis: 2.2
```

Beim Ausdruck `5 / 0` kann schon der Compiler den Fehler erkennen und das Programm zurückweisen. Lautet der Ausdruck jedoch etwa `5 / x`, wobei `x` während des Programmlaufs erst berechnet wird, ist für den Compiler kein Fehler ersichtlich. Hat jedoch dann `x` den Wert `0`, wird beim Versuch, diesen Ausdruck zu berechnen, das Programm mit einer Fehlermeldung abgebrochen.

9.4.5 Restwert-Operator: `x % y`

Der Restwert-Operator (auch als Modulo-Operator bezeichnet) gibt den Rest bei der Integer-Division des Operanden `x` durch den Operanden `y` an. Eine Division durch `0` ist nicht erlaubt. Gleitpunkt-Zahlen sind ebenfalls nicht erlaubt.

```
5 % 3          // Ergebnis: 2
10 % 5         // Ergebnis: 0
3 % 7          // Ergebnis: 3
2 % 0          // Dieser Ausdruck ist nicht zulaessig
3. % 2.        // Dieser Ausdruck ist nicht zulaessig
```

9.5 Zuweisungs-Operatoren

Zu den Zuweisungs-Operatoren gehören der einfach Zuweisungs-Operator (auf Englisch „simple assignment operator“) sowie die kombinierten Zuweisungs-Operatoren (auf Englisch „compound assignment operator“).

Zuweisungs-Operator	x = y
Additions-Zuweisungs-Operator	x += y
Subtraktions-Zuweisungs-Operator	x -= y
Multiplikations-Zuweisungs-Operator	x *= y
Divisions-Zuweisungs-Operator	x /= y
Restwert-Zuweisungs-Operator	x %= y
Bitweises-UND-Zuweisungs-Operator	x &= y
Bitweises-ODER-Zuweisungs-Operator	x = y
Bitweises-Exklusiv-ODER-Zuweisungs-Operator	x ^= y
Linksschiebe-Zuweisungs-Operator	x <<= y
Rechtsschiebe-Zuweisungs-Operator	x >>= y

Dabei darf zwischen den Zeichen eines kombinierten Zuweisungs-Operators kein Leerzeichen stehen.

9.5.1 Einfacher Zuweisungs-Operator $x = y$

Der Zuweisungs-Operator wird in C als binärer Operator betrachtet. Er speichert den Rückgabewert des rechten Operanden in dem linken Operanden. Diese Zuweisung wird als „Nebeneffekt“ bezeichnet, sprich es wird eine Speicherstelle verändert.

```
b = 1 + 3;
```

Es handelt sich bei einer Zuweisung um einen Ausdruck.



In C können Zuweisungen wiederum in größeren Ausdrücken weiterverwendet werden. Der Ausdruck liefert als Rückgabewert den Wert des rechten Operanden. Folgendes Beispiel ist also möglich:

```
a = b = c;
```

Dabei wird der Operator von rechts nach links abgearbeitet, sprich zuerst wird $b = c$ ausgeführt und ausgewertet und danach wird dieses Resultat a zugewiesen.

Zuweisungs-Operatoren haben eine geringe Priorität, so dass man beispielsweise bei einer Zuweisung $b = x + 3$ den Ausdruck $x + 3$ nicht in Klammern setzen muss. Erst erfolgt die Auswertung des arithmetischen Ausdrucks, dann erfolgt die Zuweisung.

9.5.2 Kombinierte Zuweisungs-Operatoren

Die kombinierten Zuweisungs-Operatoren sind – wie der Name schon verrät – aus mehreren Symbolen zusammengesetzte Operatoren. Sinngemäß führen sie auch mehrere Operationen auf einmal aus.

Beispielsweise wird beim Ausdruck $x += y$ zum einen die Addition $x + (y)$ durchgeführt. Der Rückgabewert dieser Addition ist $x + (y)$. Zum anderen erhält die Variable x als Nebeneffekt den Wert dieser Addition zugewiesen. Damit entspricht der Ausdruck $x += y$ semantisch genau dem Ausdruck $x = x + (y)$.

Die Klammer in dieser Erklärung sind nötig, da y selbst ein Ausdruck wie beispielsweise $b * 3$ sein kann. Es wird also zuerst der Ausdruck y ausgewertet, bevor $x + (y)$ berechnet wird.

Für die anderen kombinierten Zuweisungs-Operatoren gilt das Gleiche wie bei dem Additions-Zuweisungs-Operator. Hier sind alle kombinierten Zuweisungs-Operatoren aufgeführt mit der jeweiligen ausführlichen Schreibweise:

```
a -= 1           // a = a - 1
b *= 2           // b = b * 2
c /= 5           // c = c / 5
d %= 5           // d = d % 5
e &= 8           // e = e & 8
f |= 4           // f = f | 4
g ^= 3           // g = g ^ 3
h <<= 1          // h = h << 1
i >>= 1          // i = i >> 1
```

9.6 Relationale Operatoren

Im Folgenden werden die zweistelligen relationalen Operatoren vorgestellt:

Gleichheits-Operator	<code>x == y</code>
Ungleichheits-Operator	<code>x != y</code>
Größer-Operator	<code>x > y</code>
Kleiner-Operator	<code>x < y</code>
Größergleich-Operator	<code>x >= y</code>
Kleinergleich-Operator	<code>x <= y</code>

Relationale Operatoren werden auch als Vergleichs-Operatoren bezeichnet. Die Priorität der Operatoren `==` und `!=` (manchmal auch Äquivalenz-Operatoren genannt) ist kleiner als die der Operatoren `>`, `>=`, `<` und `<=`. Besitzen die Operanden unterschiedliche Datentypen, werden implizite Typumwandlungen durchgeführt, siehe Kapitel [→ 9.10](#). Nebeneffekte treten bei Vergleichsoperationen nicht auf.

9.6.1 Boolesche Werte

C kannte bis zum Standard C23 in seinem Sprachkern keine eigenen Datentypen für **boolesche Werte (Wahrheitswerte)**. Statt der booleschen Werte „wahr“ und „falsch“ oder `true` und `false` wurden stets einfach Zahlen verwendet.

Die `0` gilt als „falsch“, jede Zahl ungleich `0` wie beispielsweise `1`, `3`, `-17` oder `0.1` gilt als „wahr“.



Ab dem Standard C23 sind der Datentyp `bool`, sowie die beiden Werte `true` und `false` im Sprachkern eingebaut. Die alten Werte `0` und nicht-`0` sind jedoch nach wie vor genauso gültig.

Der Rückgabewert von Vergleichsoperationen ist immer vom Datentyp `int`. Wenn ein Vergleich falsch ist, ist der Rückgabewert `0`, wenn er wahr ist, ist der Rückgabewert `1`. Vergleichsoperationen für Pointer liefern dieselben Rückgabewerte. In Kapitel [→ 12.2.4](#) wird behandelt, welche Vergleiche für Pointer definiert sind.

9.6.2 Gleichheits-Operator: $x == y$

Mit dem Gleichheits-Operator wird überprüft, ob der Wert des linken Operanden mit dem Wert des rechten Operanden übereinstimmt. Ist das der Fall, hat der Rückgabewert den Wert 1. Andernfalls hat der Rückgabewert den Wert 0:

```
1 + 2 == 3          // Ergebnis: 1 (wahr)
2 - 2 == 1          // Ergebnis: 0 (falsch)
```

Ein folgenschwerer Fehler ist in C, statt des Gleichheits-Operators `==` versehentlich den Zuweisungs-Operator `=` zu schreiben. Ein solches Programm ist oft compiler- und lauffähig, erzeugt aber andere Ergebnisse als erwartet.



9.6.3 Ungleichheits-Operator: $x != y$

Mit dem Ungleichheits-Operator wird überprüft, ob der Wert des linken Operanden verschieden vom Wert des rechten Operanden ist. Bei Ungleichheit hat der Rückgabewert den Wert 1. Andernfalls hat der Rückgabewert den Wert 0.

```
5 != 5              // Ergebnis: 0 (falsch)
3 != 5              // Ergebnis: 1 (wahr)
```

9.6.4 Größer-Operator: $x > y$

Mit dem Größer-Operator wird überprüft, ob der Wert des linken Operanden größer als der Wert des rechten Operanden ist. Ist der Vergleich wahr, hat der Rückgabewert den Wert 1. Andernfalls hat der Rückgabewert den Wert 0.

```
5 > 3               // Ergebnis: 1 (wahr)
4 > 1 + 3           // Ergebnis: 0 (falsch)
```

Beim zweiten Beispiel ist zu beachten, dass der Größer-Operator weniger prioritär behandelt wird als der Additions-Operator, es wird also zuerst 1 und 3 zusammengezählt.

9.6.5 Kleiner-Operator: $x < y$

Mit dem Kleiner-Operator wird überprüft, ob der Wert des linken Operanden kleiner als der Wert des rechten Operanden ist. Ist der Vergleich wahr, hat der Rückgabewert den Wert 1. Andernfalls hat der Rückgabewert den Wert 0.

```
3 < 4                // Ergebnis: 1 (wahr)
1 == 1 < 3           // Ergebnis: 1 (wahr)
```

Beim zweiten Beispiel ist zu beachten, dass der Kleiner-Operator eine höhere Priorität hat als der Gleichheits-Operator, es wird also zuerst geprüft, ob 1 kleiner ist als 3.

9.6.6 Größergleich-Operator: $x >= y$

Der Größergleich-Operator liefert genau dann den Rückgabewert 1 (wahr), wenn entweder der Wert des linken Operanden größer als der Wert des rechten Operanden ist oder der Wert des linken Operanden dem Wert des rechten Operanden entspricht. Ansonsten ist der Rückgabewert 0 (falsch).

```
2 >= 1               // Ergebnis: 1 (wahr)
1 >= 1               // Ergebnis: 1 (wahr)
```

9.6.7 Kleingleich-Operator: $x <= y$

Der Kleingleich-Operator liefert genau dann den Rückgabewert 1 (wahr), wenn entweder der Wert des linken Operanden kleiner als der Wert des rechten Operanden ist oder der Wert des linken Operanden dem Wert des rechten Operanden entspricht. Ansonsten ist der Rückgabewert 0 (falsch).

```
10 <= 11             // Ergebnis: 1 (wahr)
11 <= 11             // Ergebnis: 1 (wahr)
```

9.7 Logische Operatoren

Im Folgenden werden die drei logischen Operatoren mit ihren Operanden anhand von Beispielen vorgestellt.

Operator für das logische UND	x && y
Operator für das logische ODER	x y
Logischer Negations-Operator	!x

Achten Sie bitte auf die Schreibweise für die logischen Operatoren && und ||. C kennt nämlich auch die Bit-Operatoren & und |. Daher meldet der Compiler keinen Fehler bei einem Ausdruck wie etwa `a & b`. Die Wirkung der Bit-Operatoren ist jedoch eine ganz andere als die der logischen Operatoren.



Die Operatoren für das logische UND/ODER sind zweistellig, wohingegen der logische Negations-Operator einstellig ist. Mit diesen Operatoren lassen sich logische Verknüpfungen von Ausdrücken durchführen. Von den logischen Operatoren hat der Negations-Operator die höchste Priorität, der ODER-Operator die geringste.

9.7.1 Operator für das logische UND: x && y

Der Operator für das logische UND liefert genau dann den Rückgabewert 1 (wahr), wenn der linke und der rechte Operand einen von 0 verschiedenen Wert – also beide den Wahrheitswert „wahr“ – haben. Ansonsten ist der Rückgabewert 0 (falsch). Die Operanden können verschiedene Typen besitzen, aber es muss ein skalarer Typ sein. Mit anderen Worten, die Operanden müssen einen arithmetischen Typ oder einen Pointer-Typ haben. Das folgende Bild zeigt die Wahrheitstabelle für das logische UND:

x	y	x && y
falsch	falsch	falsch
falsch	wahr	falsch
wahr	falsch	falsch
wahr	wahr	wahr

Die Wahrheitstabelle wird folgendermaßen interpretiert: Der logische Ausdruck `x && y` ist nur dann wahr, wenn der Ausdruck `x` und der Ausdruck `y` wahr sind.

Beispiele:

```
0 && 1           // Ergebnis: 0 (falsch)
5 && 6           // Ergebnis: 1 (wahr)
3 < 5 && 5 > 3   // Ergebnis: 1 (wahr)
```

9.7.2 Operator für das logische ODER: `x || y`

Der Operator für das logische ODER liefert genau dann den Rückgabewert 1 (wahr), wenn der linke oder der rechte Operand oder beide Operanden einen von 0 verschiedenen Wert, also den Wahrheitswert „wahr“, haben. Ansonsten ist der Rückgabewert 0 (falsch). Die Operanden können verschiedene Typen besitzen, aber es muss ein skalarer Typ sein. Mit anderen Worten, die Operanden müssen einen arithmetischen Typ oder einen Pointer-Typ haben. Das folgende Bild zeigt die Wahrheitstabelle für das logische ODER:

x	y	x y
falsch	falsch	falsch
falsch	wahr	wahr
wahr	falsch	wahr
wahr	wahr	wahr

Die Wahrheitstabelle wird folgendermaßen interpretiert: Der logische Ausdruck `x || y` ist dann wahr, wenn der Ausdruck `x` oder der Ausdruck `y` oder beide Ausdrücke wahr sind.

Beispiele:

```
0 || 1           // Ergebnis: 1 (wahr)
0 || (1 && 0)     // Ergebnis: 0 (falsch)
'b' == 'a' + 1 || 0 // Ergebnis: 1 (wahr)
```

Bei der Auswertung des letzten Beispiels ist die Priorität der Operatoren zu beachten. Es gilt die Reihenfolge: Additions-Operator, Vergleichs-Operator, dann der ODER-Operator.

9.7.3 Logischer Negations-Operator: !x

Mit dem einstelligen Negations-Operator werden Wahrheitswerte negiert, sprich aus „wahr“ wird „falsch“ (Rückgabewert 0) und aus „falsch“ wird „wahr“ (Rückgabewert 1). Wird der Negations-Operator zweimal auf seinen Operanden angewandt, bleibt der Wahrheitswert unverändert. Das folgende Bild zeigt die Wahrheitstabelle für die Negation:

x	!x
falsch	wahr
wahr	falsch

Die Wahrheitstabelle wird folgendermaßen interpretiert: Der logische Ausdruck !x ist nur dann wahr, wenn der Ausdruck x falsch ist.

Beispiele:

```
!0           // Ergebnis: 1 (wahr)
!!5          // Ergebnis: 1 (wahr)
!0 == 1      // Ergebnis: 1 (wahr)
```

9.7.4 Spezielles bei der Abarbeitung von logischen Operatoren

Die Operatoren für das logische UND/ODER haben eine sehr geringe Priorität. Die Vergleichs-Operatoren haben eine höhere Priorität als die logischen Operatoren. Deshalb sind Klammern für die Auswertung von Ausdrücken bei Kontrollstrukturen (siehe Kapitel [→ 10](#)) oft nicht notwendig. So sind die beiden folgenden Ausdrücke äquivalent. Die Klammern erhöhen lediglich die Übersichtlichkeit der Programme.

```
(a < b) && (c == d)
a < b  &&  c == d
```

Gemäß den Regeln der binären Logik hat der Operator `&&` eine höhere Priorität als der `||`-Operator. Dennoch werden zur besseren Lesbarkeit logische Ausdrücke oftmals zusätzlich geklammert. Manche Compiler geben sogar Warnungen aus, wenn ein logischer Ausdruck die beiden Operatoren mischt. Die folgenden beiden Ausdrücke sind äquivalent:

```
(a && b) || (c && d)
a && b || c && d
```

Logische Operatoren definieren Sequenzpunkte (siehe Kapitel [→ 9.1.2](#)). Damit ist eindeutig festgelegt, dass bei einem Ausdruck wie `x && y` zuerst der Ausdruck `x` ausgewertet werden muss.

Nebeneffekte des linken Operanden werden ausgeführt, bevor die weiteren Operanden ausgewertet werden. Die Auswertung wird abgebrochen, wenn das Ergebnis des Ausdrucks schon feststeht.



Damit ist gemeint:

- Ist bei einer logischen ODER-Verknüpfung bereits der erste Operand „wahr“, so ist der gesamte Ausdruck automatisch „wahr“ und der zweite Operand wird nicht ausgewertet.
- Ist bei einer logischen UND-Verknüpfung bereits der erste Operand „falsch“, so ist der gesamte Ausdruck automatisch „falsch“ und der zweite Operand wird nicht ausgewertet.

Das kann dazu führen, dass Nebeneffekte der weiter rechts stehenden Ausdrücke nicht mehr ausgeführt werden.



Hierfür ein Beispiel:

```
1 < 0 && 2 < a++    // a++ wird nie ausgeführt.
```

9.8 Bit-Operatoren

Bit-Operatoren erlauben es der Sprache C, Bit-Manipulationen auszuführen. Mit solchen Operatoren wird eine hardwarenahe Programmierung unterstützt.

Bits können bekanntermaßen zwei Zustände annehmen: 0 oder 1. Jeder Wert ist in einem Computer aus mehreren Bits zusammengesetzt.

Bit-Operationen finden auf allen Bits der Operanden statt. Bei den Bit-Operationen werden jeweils die Bits der entsprechenden Position miteinander verknüpft.



Die logischen Bit-Operatoren dürfen nur für Integer-Datentypen benutzt werden. Vorsicht ist geboten bei der Verwendung von vorzeichenbehafteten Datentypen, da hier implementierungsabhängige Aspekte auftreten. Bei vorzeichenlosen Datentypen ist die Verwendung der Bit-Operatoren problemlos.

Nebeneffekte treten bei den logischen Bit-Operatoren nicht auf. Im Folgenden werden die vier logischen Bit-Operatoren mit ihren Operanden sowie die beiden Shift-Operatoren für Bits vorgestellt:

UND-Operator für Bits	<code>x & y</code>
ODER-Operator für Bits	<code>x y</code>
Exklusiv-ODER-Operator für Bits	<code>x ^ y</code>
Negations-Operator für Bits	<code>~x</code>
Rechtsshift-Operator	<code>x >> y</code>
Linksshift-Operator	<code>x << y</code>

Achten Sie bitte auf die Schreibweise für die Bit-Operatoren `&` und `|`. C kennt nämlich auch die logischen Operatoren `&&` und `||`. Daher meldet der Compiler keinen Fehler bei einem Ausdruck wie etwa `a && b`. Die Wirkung der logischen Operatoren ist jedoch eine ganz andere als die der Bit-Operatoren.



9.8.1 UND-Operator für Bits: x & y

Die Operation bitweises UND findet auf allen Bits der Operanden statt. Dabei werden jeweils die Bits der entsprechenden Position miteinander verknüpft. Im Folgenden die Wahrheitstabelle für das bitweise UND:

Bit n von x	Bit n von y	Bit n von x & y
0	0	0
0	1	0
1	0	0
1	1	1

Die Wahrheitstabelle wird folgendermaßen interpretiert: Bei der Verknüpfung mit UND ist die 0 dominant, das heißt ist mindestens eines der Bits (Bit n von x oder Bit n von y) eine 0, so ist das Ergebnis 0 (falsch).

Mit dem UND-Operator für Bits kann man Bits in Bitmustern ausblenden. Damit ist gemeint, dass diejenigen Bits, welche mit 0 verknüpft werden, garantiert im Resultat auch 0 ergeben.



```
0 & 1           //      0 &    1 =    0
14 & 1          //  1110 & 0001 = 0000
14 & 7          //  1110 & 0111 = 0110
```

Der logische UND-Operator für Bits hat eine höhere Priorität als der logische ODER-Operator für Bits.

9.8.2 ODER-Operator für Bits: $x \mid y$

Die Operation bitweises ODER findet auf allen Bits der Operanden statt. Dabei werden jeweils die Bits der entsprechenden Position miteinander verknüpft. Hier die Wahrheitstabelle für das bitweise ODER:

Bit n von x	Bit n von y	Bit n von $x \mid y$
0	0	0
0	1	1
1	0	1
1	1	1

Die Wahrheitstabelle wird folgendermaßen interpretiert: Bei der Verknüpfung mit ODER ist die 1 dominant, das heißt ist mindestens eines der Bits (Bit n von x oder Bit n von y) eine 1, so ist das Ergebnis 1 (wahr).

Mit dem ODER-Operator für Bits kann man Bits in Bitmustern einblenden. Damit ist gemeint, dass diejenigen Bits, welche mit 1 verknüpft werden, garantiert im Resultat auch 1 ergeben.



```

0 | 1          // 0 | 1 = 1
8 | 1          // 1000 | 0001 = 1001
14 | 1         // 1110 | 0001 = 1111

```

9.8.3 Exklusiv-ODER-Operator für Bits: $x \wedge y$

Die Operation bitweises Exklusiv-ODER findet auf allen Bits der Operanden statt. Dabei werden jeweils die Bits der entsprechenden Position miteinander verknüpft. Es folgt die Wahrheitstabelle für das bitweise Exklusives-ODER:

Bit n von x	Bit n von y	Bit n von $x \wedge y$
0	0	0
0	1	1
1	0	1
1	1	0

Die Wahrheitstabelle wird folgendermaßen interpretiert: Bei der Verknüpfung mit Exklusiv-ODER ist das Ergebnis 1 (wahr), wenn entweder Bit n von Operand x oder Bit n von Operand y eine 1 ist, aber nicht beide gleichzeitig.

Mit dem Exklusiv-ODER-Operator für Bits kann man Bits invertieren.



```
0 ^ 1          // 0 ^ 1 = 1
14 ^ 1         // 1110 ^ 0001 = 1111
14 ^ 3         // 1110 ^ 0011 = 1101
```

9.8.4 Negations-Operator für Bits: ~x

Die Operation bitweise Negation findet auf allen Bits des Operanden statt. Hier die Wahrheitstabelle für die bitweise Negation:

Bit n von x	Bit n von ~x
0	1
1	0

Die Wahrheitstabelle wird folgendermaßen interpretiert: Bei der Negation für Bits wird einfach jedes Bit invertiert (Einer-Komplement). Aus der 0 wird durch Negation eine 1 und aus der 1 eine 0.

Der Negations-Operator für Bits invertiert jedes Bit.



```
unsigned char a, b;
a = 9;          // a = 00001001
b = ~a;         // b = 11110110
```

9.8.5 Rechtsshift-Operator: $x \gg y$

Mit dem Rechtsshift-Operator \gg werden die Bits von x um y Bitstellen nach rechts geschoben. Es darf nur um ganzzahlige positive Stellen von Bits verschoben werden.

Dabei gehen die y niederwertigen Bits von x verloren. Wenn der Operand x von einem unsigned-Typ ist, werden die höherwertigen Bits mit Nullen aufgefüllt, was einer Integer-Division durch 2^y entspricht. Bei einem Operanden x eines signed-Typs mit einem negativen Wert ist das Ergebnis vom Compiler abhängig. Bei vielen Compilern bleibt das Vorzeichenbit erhalten.

```
unsigned char a;
a = 8;          /* 0000 1000 Bitmuster von 8 */
a = a >> 3;     /* 0000 0001 Bitmuster von 1 */
                aufgefüllt      verloren
```

Die Verschiebung um 3 Bits nach rechts entspricht einer Division durch 2^3 .

9.8.6 Linksshift-Operator: $x \ll y$

Bei dem Linksshift-Operator \ll werden die Bits von x um y Bitstellen nach links geschoben. Es darf nur um ganzzahlige positive Stellen von Bits verschoben werden.

Dabei gehen die y höherwertigen Bits von x verloren. Die nachrückenden niederwertigen Bits werden mit Nullen aufgefüllt. Falls kein Überlauf eintritt, entspricht dies bei einem unsigned-Operanden einer Multiplikation mit 2^y . Im Falle eines Überlaufs ist das Resultat undefiniert.

```
unsigned char a = 128; /* 1000 0000 Bitmuster von 128 */
a = a << 1;           /* Overflow. Ergebnis: */
/* 0000 0000 */
a = 8;               /* 0000 1000 Bitmuster von 8 */
a = a << 3;          /* 0100 0000 Bitmuster von 64 */
                    verloren      aufgefüllt
```

Die Verschiebung um 3 Bits nach links entspricht einer Multiplikation mit 2^3 .

9.9 Sonstige Operatoren

Im diesem Kapitel werden die folgenden Operatoren vorgestellt:

sizeof-Operator	sizeof
_Alignof-Operator	_Alignof
Komma-Operator	,
Bedingungs-Operator	?:
Typumwandlungs-Operator	(Type)

9.9.1 Der sizeof-Operator

Der Operator **sizeof** dient zur Ermittlung der Größe in Bytes von Typen sowie von Datenobjekten im Hauptspeicher.



Der Operator `sizeof` darf dabei sowohl auf einen Ausdruck, also einen L-Wert wie einen Variablennamen oder einen R-Wert, als auch auf einen Typ-Bezeichner angewandt werden.

Die Syntax ist:

```
sizeof Ausdruck  
sizeof(Typname)
```

Der Rückgabewert ist dabei jeweils ein Integer, nämlich die Größe des angegebenen Operanden gemessen in Bytes. Der Typ des Rückgabewertes ist `size_t`.

Die Anzahl der Bytes wird angegeben in dem Typ `size_t`, der extra für den `sizeof`-Operator geschaffen wurde. Der Wertebereich von `size_t` ist ausreichend, um eine beliebige Speichergröße aufzunehmen. In der Regel entspricht `size_t` einem `unsigned int` mit 32 oder 64 Bit.

Im folgenden Beispielprogramm, welches auf einem spezifischen Computer läuft, werden einige Anwendungsfälle, auch für Arrays vorgestellt:

sizeof.c

```
#include <stdio.h>

int main(void) {
    int zahl1 = 1;
    int array[20] = {0};
    double zahl2 = 1.;

    printf("size of integer:   %2d Bytes\n", (int)sizeof zahl1);
    printf("size of float:     %2d Bytes\n", (int)sizeof(float));
    printf("size of double:    %2d Bytes\n", (int)sizeof zahl2);
    printf("size of array:      %2d Bytes\n", (int)sizeof array);
    printf("size of array[10]: %2d Bytes\n", (int)sizeof array[10]);
    return 0;
}
```

Hier das Protokoll des Programmlaufs:

```
size of integer:   4 Bytes
size of float:     4 Bytes
size of double:    8 Bytes
size of array:     80 Bytes
size of array[10]:  4 Bytes
```

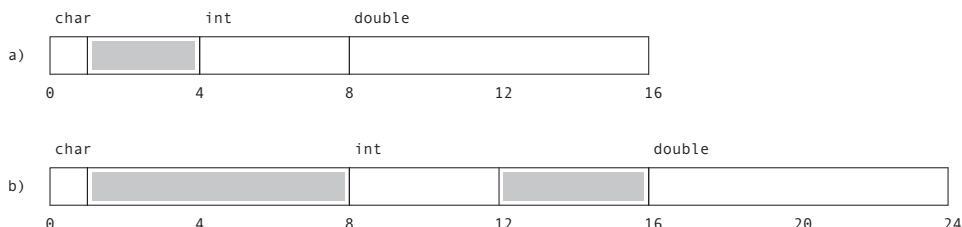
Wenn der sizeof-Operator auf einen Typ angewandt wird, müssen Klammern geschrieben werden. Bei Ausdrücken können die Klammern weggelassen werden. Wenn man sich nicht merken kann (oder will), ob man bei dem Operator sizeof Klammern braucht oder nicht, sollte man immer Klammern setzen. Denn ein geklammerter Ausdruck bleibt syntaktisch ein Ausdruck und wird somit beim sizeof-Operator akzeptiert.

Besonders oft wird der sizeof-Operator zur Berechnung der Größe von Strukturen verwendet. Strukturen sind zusammengesetzte Datentypen, deren Komponenten verschiedene Datentypen haben können. Da es dem Compiler freisteht, Komponenten der Struktur auf bestimmten Wortgrenzen beginnen zu lassen, kann die Struktur ungenutzten, namenlosen Platz enthalten. Damit lässt sich die Größe einer Struktur nicht durch Addition der Größen der Komponenten ermitteln.

Ein Compiler legt Objekte eines bestimmten Typs bei vielen Architekturen auf Speicheradressen, die ein gegebenes Vielfaches einer Byte-Adresse sind. Eine solche Anordnung von Objekten im Speicher wird als **Alignment** bezeichnet.



Das folgende Bild zeigt ein Speicherabbild einer Struktur für zwei verschiedene Compiler. Unbenutzte Bereiche sind grau hinterlegt:



Der Compiler legt im Beispiel a `int`-Objekte auf 4-Bytegrenzen und `double`-Objekte auf 8-Bytegrenzen. Der Compiler im Beispiel b legt sowohl `int`- als auch `double`-Variablen auf 8-Bytegrenzen. `char`-Variablen werden in beiden Fällen auf Byte-Adressen gelegt.

Der `sizeof`-Operator wird vor allem angewandt, um Programme portabler zu machen.



So ist es beispielsweise oft nötig, unter Verwendung der Bibliotheksfunktion `memcpy()` Objekte direkt im Speicher zu kopieren oder zu verschieben (genauer dazu kann in Kapitel [→ 12.8.1](#) nachgelesen werden). Dazu benötigt diese Funktion unter anderem die Größe des zu bearbeitenden Objektes. Statt diesen Wert nun als Konstante im Programm einzuführen, ist es bezüglich der Portierbarkeit auf andere Maschinen besser, dem Compiler die Ermittlung der Größe des Objektes zu überlassen. Sollte das Objekt auf einer anderen Maschine aufgrund der internen Darstellung eine andere Größe haben, so wird dieses Problem über den `sizeof`-Operator vom Compiler erledigt. Ein manueller Eingriff in das Programm ist also nicht erforderlich.

Auf einem Computer mit 32-Bit Integer-Zahlen ergeben sich folgende Beispiele:

```
int zahl = 1;
sizeof 534;           // Ergebnis: 4 (Bytes)
sizeof(int);          // Ergebnis: 4 (Bytes)
sizeof zahl++;        // Ergebnis: 4 (Bytes)
```

Der sizeof-Operator wertet einen Ausdruck, der ihm übergeben wurde, nicht aus. Er bestimmt lediglich den Typ des Ausdrucks und dann die Größe dieses Typs.



Der Wert von `zahl` wird also durch `sizeof zahl++` also nicht verändert, da der Ausdruck `zahl++` gar nicht ausgewertet wird, sondern nur sein Typ bestimmt wird.

Die Anzahl der Elemente eines Arrays lässt sich mit `sizeof` wie folgt portabel bestimmen:

```
int a[] = {4, 7, 11, 0, 8, 15};
size_t len = sizeof(a) / sizeof(a[0]);
```

Dieses Vorgehen funktioniert aber nur bei Arrays, deren Größe bei der Compilation des Programms bereits bekannt ist. Für ein Array `a`, das beispielsweise als Pointer einer Funktion übergeben wurde (siehe [→ 12.4](#)), ist das Vorgehen nicht erfolgreich.

9.9.2 Der `_Alignas`-, `_Alignof`- und `offsetof`-Operator



Der Compiler ordnet Daten im Speicher so an, dass sie vom Prozessor möglichst effizient gelesen und geschrieben werden können. Je nach Compiler und Prozessor wird die **Byte-Ausrichtung** (auf Englisch „**Alignment**“) der verschiedenen Datentypen im Speicher unterschiedlich ausfallen. Ein Objekt eines Typs, welcher 4 Bytes benötigt, wird beispielsweise von vielen Compilern an eine Adresse gesetzt, welche durch 4 teilbar ist. Dies wird als „Alignment an eine 4-Byte-Grenze“ bezeichnet.

Mittels des Operators `_Alignas` kann die Anordnung von Variablen explizit gesteuert werden. Ob und wie ein Compiler diese Angaben umsetzen kann, ist je nach Implementation unterschiedlich.



Mittels `_Alignof` kann die tatsächlich verwendete Anordnung eines Typs abgefragt werden.



```
int a;
_Alignas(16) int b;
_Alignas(double) int c;
printf("%d\n", (int)_Alignof(a)); // Ergebnis: 4
printf("%d\n", (int)_Alignof(b)); // Ergebnis: 16
printf("%d\n", (int)_Alignof(c)); // Ergebnis: 8
```

Der `_Alignas`-Operator erwartet eine Bytegröße oder einen bekannten Typ als Eingabe. Er kann gemäß Standard im Gegensatz zum `sizeof`-Operator nur auf Typen angewendet werden, nicht aber auf Ausdrücke. Manche Compiler erlauben dies dennoch.

Der `_Alignof`-Operator gibt genauso wie der `sizeof`-Operator einen (zur Compilierzeit bekannten) konstanten Wert des Typs `size_t` zurück. Dieser Wert entspricht dem Alignment des Typs, sprich der Bytegrenze, an welcher ein Typ ausgerichtet ist. Beispielsweise:

```
_Alignof(float); // Ergebnis 4 (Bytes)
_Alignof(double); // Ergebnis 8 (Bytes)
```


Im Folgenden ein Beispiel für Arrays:

```
sizeof (int[10]);      // Ergebnis 40 (Bytes)
_Alignof (int[10]);    // Ergebnis 4 (Bytes)
```

In diesem Beispiel werden auf einem System 4 Bytes für den Datentyp `int` benötigt. Dementsprechend werden 40 Bytes Speicher für ein Array mit 10 Komponenten vom Typ `int` benötigt. Die einzelnen Elemente und somit das gesamte Array jedoch werden vom Compiler auf eine (beliebige) 4-Byte-Grenze ausgerichtet.

Interessant wird der Rückgabewert bei Strukturen. Der Compiler hat die Aufgabe, alle Elemente einer Struktur so anzuordnen, dass jedes einzelne Element an seiner vom Compiler bestimmten Ausrichtungsgrenze zu liegen kommt. Mit Hilfe des `_Alignof`-Operators kann man herausfinden, wo der gesamte Speicherblock zu liegen kommt:

```
struct structtype{float x; double y;};
printf("%d\n", (int)sizeof(struct structtype)); // Ergebnis: 16
printf("%d\n", (int)_Alignof(struct structtype)); // Ergebnis: 8
```

Und mithilfe des `offsetof`-Operators erhält man die Startadresse innerhalb der Struktur für ein spezifisches Element:

```
printf("%d\n", (int)offsetof(struct structtype, x)); // Ergebnis: 0
printf("%d\n", (int)offsetof(struct structtype, y)); // Ergebnis: 8
```

Der `offsetof`-Operator existiert seit Beginn der Standardisierung von C in der `<stddef.h>`-Bibliothek. Die Operatoren `_Alignof()` und `_Alignas()` existieren jedoch erst seit dem C11-Standard. Der C11-Standard beschreibt zudem eine neue Bibliothek `<stdalign.h>`, welche die Makros `alignof` und `alignas` definiert, die zu `_Alignof` und `_Alignas` ausgewertet werden. Ab dem Standard C23 werden die neuen Schlüsselwörter `alignof` und `alignas` eingeführt.

Ab dem Standard C23 wird zudem die Funktion `memalignment()` eingeführt. Mit dieser Funktion kann die Byte-Ausrichtung eines beliebigen Pointers bestimmt werden, sprich an welcher Byte-Grenze Speicher reserviert werden soll. In diesem Buch wird auf diese Funktionalität nicht eingegangen.

9.9.3 Der Komma-Operator: x, y

Der Komma-Operator wird in der Praxis selten eingesetzt.

Ein Ausdruck x, y wird von links nach rechts abgearbeitet. Erst wird der Ausdruck x ausgewertet, dann der Ausdruck y . Nebeneffekte des linken Ausdrucks sind nach dessen Auswertung eingetreten.



Der Rückgabewert und -typ ist der Wert und der Typ von y , das heißt des rechten Operanden. Ein Komma-Operator liefert keinen L-Wert.

Die allermeisten Vorkommen des Komma-Zeichens im Quellcode sind jedoch nicht dem Komma-Operator zuzuschreiben, sondern beispielsweise Parameterlisten, Initialisierern oder enum-Deklarationen.



Der Komma-Operator wird nur sehr selten verwendet, da er grundsätzlich nichts anderes macht, als zwei Ausdrücke sequenziell hintereinander auszuwerten. Im Normalfall wird man dies mittels zwei aufeinanderfolgenden Anweisungen bewerkstelligen.

Im Vergleich zu zwei aufeinanderfolgenden Anweisungen hat der Komma-Operator jedoch zwei Vorteile: Erstens gibt der gesamte Ausdruck einen Wert zurück und zweitens werden die verknüpften Ausdrücke als ein einziger Ausdruck angesehen.



Wozu kann der Komma-Operator gut sein? Beispielsweise kann man damit in einer for-Anweisung zwei Indizes gleichzeitig bearbeiten, um einen String mit einem herabzuzählenden Index und einem hochzählenden Index umzudrehen:

revers.c

```
#include <stdio.h>

int main(void) {
    char text1[] = "Hallo Welt";
    char text2[sizeof text1 / sizeof(char)];

    int i1, i2;
    for (i2 = 0, i1 = sizeof text1 - 2; i1 >= 0; ++i2, --i1) {
        text2[i2] = text1[i1];
    }
    text2[i2] = '\0';
    printf("%s\n%s\n", text1, text2);
    return 0;
}
```

Das Programm gibt aus:

```
Hallo Welt
tleW ollaH
```

9.9.4 Der Bedingungs-Operator: $x ? y : z$

Eine echte „Rarität“ in der Programmiersprache C ist der Bedingungs-Operator. Er ist nämlich der einzige Operator, der drei Operanden verarbeitet.

In einem bedingten Ausdruck $x ? y : z$ wird zuerst der Ausdruck x ausgewertet. Ist der Rückgabewert von Ausdruck x ungleich 0, also wahr, dann ist der Rückgabewert des gesamten bedingten Ausdrucks das Ergebnis von y . Ist jedoch der Ausdruck x gleich 0, also falsch, so wird der Ausdruck z ausgewertet und dessen Ergebnis ist der Rückgabewert des bedingten Ausdrucks.



```
5 < 10 ? 123 : 52          // Rueckgabewert: 123
0 ? 0 : 1                  // Rueckgabewert: 1
```

Ist die Bedingung x wahr, dann wird der Ausdruck z nicht ausgewertet und ist die Bedingung x falsch, dann wird der Ausdruck y nicht ausgewertet!



Der Typ des bedingten Ausdrucks $x ? y : z$ ist – unabhängig davon, ob der Rückgabewert dieses Ausdrucks y oder z ist – stets der mächtigere Typ (siehe Kapitel [→ 9.10](#)) der beiden Ausdrücke y und z . So ist beispielsweise der Rückgabotyp des folgenden Ausdrucks vom Typ `double` und der Rückgabewert ist `6.0`:

```
(3 > 4) ? 5.0 : 6
```

Zu beachten ist, dass beim Bedingungs-Operator zuerst die Bedingung ausgewertet wird. Nebeneffekte des linken Operanden werden damit ausgeführt, bevor die weiteren Operanden ausgewertet werden.



Der Bedingungs-Operator ist grundsätzlich eine vereinfachte Schreibweise einer Bedingung, welche gerne für Zuweisungen oder Funktionsrückgaben genutzt wird. Eine Funktion kann mit der `return`-Anweisung einen Wert an den Aufrufer zurückliefern, siehe dazu Kapitel [→ 11.4](#):

```
if (x)
    return y;
else
    return z;
```

Mit dem Bedingungs-Operator kann dies viel einfacher geschrieben werden:

```
return x ? y : z;
```

9.9.5 Der Typumwandlungs-Operator (Typname)

Eine **explizite Typumwandlung** eines beliebigen Ausdrucks kann man mit dem **cast-Operator** (Typumwandlungs-Operator) durchführen.



Das englische Wort „cast“ bedeutet hier „in eine Form gießen“.

Mit diesem Operator können impliziten Typumwandlungen (siehe dazu Kapitel [→ 9.10](#)), die der Compiler automatisch durchführen würde, vermieden werden. Da implizite Typumwandlungen komplex und fehlerträchtig sind, sollte man die notwendigen Typumwandlungen möglichst immer selbst durch Einsatz des cast-Operators festlegen.

Die Syntax des Typumwandlungs-Operators ist die folgende:

```
(Typname)Ausdruck
```

Damit wird der Wert des Ausdrucks in den Typ gewandelt, der in der Klammer angegeben ist. So erwartet beispielsweise die Bibliotheksfunktion `cos()`, die in der Bibliothek `<math.h>` deklariert ist, einen Ausdruck vom Typ `double`. Ist `n` ein Integer, dann kann mit `cos((double)n)` der Wert von `n` in `double` umgewandelt werden, bevor er als Parameter an `cos()` übergeben wird.

Es kann nicht jeder Typ in einen beliebigen anderen Typ gewandelt werden. Möglich sind insbesondere folgende Wandlungen:

- Wandlungen zwischen Integer-Typen
- Wandlungen zwischen Gleitpunkt-Typen
- Wandlungen zwischen Integer- und Gleitpunkt-Typen
- Wandlungen zwischen Pointern auf Variablen
- Wandlungen zwischen Pointern und Integer-Typen
- Wandlungen zwischen Pointern und dem Typ `void*`
- Wandlungen in den Typ `void`

Wandlungen zwischen arithmetischen Typen sind grundsätzlich erlaubt, verursachen jedoch möglicherweise eine Verfälschung des gespeicherten Wertes, wenn der Wertumfang des Ziel-Typs nicht mächtig genug ist, den ursprünglichen Wert abzubilden. Siehe hierzu die Konvertierungsregeln in Kapitel [9.11](#).

Zwischen Pointern und Integer-Typen zu wandeln gilt heutzutage als verpönt, da dadurch gefährliche Adress-Berechnungen durchgeführt werden können, welche nicht als sicher gelten.

Eine Umwandlung in den Typ `void` kann beispielsweise verwendet werden, um explizit darzustellen, dass der Rückgabewert eines Ausdrucks nicht verwendet wird, wie beispielsweise:

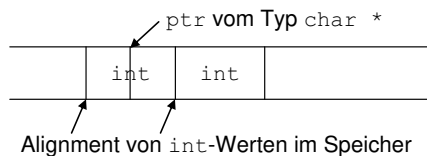
```
(void) printf("%d", x);
```

Bei der Umwandlung von Pointern sollte darauf geachtet werden, dass die zugrundeliegenden Typen zueinander kompatibel sind. Beispielsweise macht es wenig Sinn, einen `double*` in einen `char*` zu casten, verboten ist es jedoch nicht. Problematisch wird es jedoch spätestens, wenn das Alignment des zugrundeliegenden Typs nicht übereinstimmt.

Ein Pointer auf ein Objekt kann in einen Pointer auf ein anderes Objekt gewandelt werden. Der resultierende Pointer kann ungültig sein, wenn das Alignment für den Typ, auf den er zeigt, nicht stimmt.



Dies symbolisiert das folgende Beispiel:



Wird der Pointer `ptr` in diesem Beispiel in einen Pointer auf `int` gewandelt, so stimmt das Alignment nicht und der gewandelte Pointer zeigt auf einen unbrauchbaren Wert. Je nach Prozessor kann dies zu einem sofortigen Abbruch des Programmes führen.

9.10 Implizite Typumwandlung

In C ist es nicht notwendig, dass die Operanden eines Ausdrucks vom selben Typ sind. Genauso wenig muss bei einer Zuweisung der Typ der Operanden übereinstimmen. Auch bei der Übergabe von Werten an Funktionen und bei Rückgabewerten von Funktionen (siehe Kapitel [→ 11.3](#)) können übergebene Ausdrücke beziehungsweise der rückzugebende Ausdruck von den formalen Parametern beziehungsweise dem Rückgabebetyp verschieden sein. In solchen Fällen kann der Compiler selbsttätig **implizite Typumwandlungen** (also automatische Typumwandlungen) durchführen, die nach einem von der Sprache vorgeschriebenen Regelwerk ablaufen. Diese Regeln sollen in diesem Kapitel vorgestellt werden.

Implizite Typumwandlungen können gefährlich sein, da man sie oft nicht richtig einschätzt. Sorgen Sie am besten selbst dafür, dass die Typen jeweils übereinstimmen!



Man kann implizite Umwandlungen vermindern, indem bereits beim Schreiben der literalen Konstanten im Quellcode die Zahlen mit dem korrekten Suffix typisiert werden (siehe dazu Kapitel [→ 6.5](#)), oder aber man den expliziten Casting-Operator anwendet (siehe Kapitel [→ 9.9.5](#)). Wenn man selbst dafür sorgt, dass solche Typverschiedenheiten nicht vorkommen, braucht man sich um die implizite Typumwandlung nicht zu kümmern.

Implizite Typumwandlungen erfolgen in C prinzipiell nur zwischen verträglichen Datentypen. Zwischen unverträglichen Datentypen gibt es keine impliziten Umwandlungen. Hier wird der Compiler einen Fehler melden.

Implizite Typumwandlungen gibt es in folgenden Fällen:

- Bei einem Pointer auf void (siehe Kapitel [→ 8.2](#))
- Bei Operanden von arithmetischem Typ
- Bei Zuweisungen, Rückgabewerten und formalen Parametern von Funktionen (siehe Kapitel [→ 11.3](#))

Im Folgenden wird auf die Fälle der arithmetischen Typen eingegangen.


9.10.1 Arithmetische Typumwandlung

Wenn arithmetische Operanden als Teil einer Zuweisung auftreten, so wird stets der Operand auf der rechten Seite in den Typ auf der linken Seite umgewandelt.



Bei allen anderen binären Operatoren versucht der Compiler, eine möglichst optimale Umwandlung entweder des linken, des rechten oder von beiden Operanden vorzunehmen. Solche „gewöhnlichen“ **arithmetische Typumwandlungen** werden bei binären Operatoren mit Ausnahme der logischen Operatoren && und || durchgeführt. Sie werden auch beim ternären Bedingungs-Operator ?: durchgeführt. Das Ziel ist es, einen gemeinsamen Typ der Operanden des binären Operators zu erhalten, der dann auch der Typ des Ergebnisses ist.



Die allgemeinen Regeln, welche Typen in welche umgewandelt werden können, sind ganz genau spezifiziert im gültigen C-Standard. Dabei gibt es Unterschiede zwischen C90 und dem neueren Standard C11. Insbesondere werden in C11 Umwandlungen zwischen den Typmodifikatoren short, long und long long (siehe Anhang  E) für die jeweils vorzeichenbehaftete und vorzeichenlose Variante festgelegt.

Die genauen Regeln aufzulisten sprengt jedoch den Rahmen dieses Buches. Hier wird nur folgende, allgemeine Regel aufgeführt, welche für die alltägliche Programmierung mehr als ausreichend ist:

Die Sprache C weist jedem arithmetischen Typ eine sogenannte „Mächtigkeit“ beziehungsweise eine „Rangordnung“ zu. Ein Typ ist grundsätzlich mächtiger als ein anderer, wenn sein Wertebereich im positiven Bereich größer ist. Bei arithmetischen Operanden gilt generell, dass der „kleinere“ Datentyp in den „größeren“ Datentyp umgewandelt wird. Nur bei Zuweisungen kommen auch Wandlungen vom „größeren“ in den „kleineren“ Datentyp vor.



Wir betrachten als Beispiel den folgenden Code-Ausschnitt:

```
int fahr = 54;  
float celsius = (5.0 / 9) * (fahr - 32);
```

Die Temperatur in Fahrenheit ist hier in der Variablen `fahr` gespeichert, welche vom Typ `int` ist. Der Ausdruck `fahr - 32` in sich selbst benötigt keine implizite Konversion, da beide Operanden vom Typ `int` sind.

Der Ausdruck `(5.0 / 9)` beinhaltet die literale Konstante `5.0`, welche vom Typ `double` ist. Dadurch wird die `9` implizit ebenfalls in einen `double` umgewandelt.

Aufgrund dessen muss nun jedoch das Ergebnis des Ausdrucks `(fahr - 32)` als Ganzes in einen `double` umgewandelt werden, damit der Multiplikations-Operator ausgeführt werden kann. Das Resultat der gesamten rechten Seite der Anweisung ist somit vom Typ `double`.

Schlussendlich wird das Resultat der rechten Seite in die Variable auf der linken Seite gespeichert. Damit muss jedoch der `double`-Typ in einen `float`-Typ umgewandelt werden gemäß den Regeln der Zuweisung.

An dieser Stelle sei angemerkt, dass man sich diese letzte Umwandlung sparen kann, indem die Gleitpunkt-Zahl mit `5.0f` geschrieben wird. Dadurch wird die Zahl automatisch als `float` aufgefasst und sämtliche impliziten Konvertierungen werden danach ebenfalls vom Typ `float` sein.

In dem vorangegangenen Beispiel wurde der Divisions-Operator verwendet. Es muss darauf hingewiesen werden, dass der Divisions-Operator eine häufige Fehlerquelle darstellt, wenn es um implizite Typumwandlung geht: Der Ausdruck `(5.0 / 9)` dividiert eine Gleitpunkt-Zahl durch einen Integer. Würde anstelle der `5.0` nur der Integer `5` dastehen, so würde keine implizite Umwandlung stattfinden und es würde (fälschlicherweise) die Integer-Division `(5 / 9)` mit dem Ergebnis `0` ausgeführt werden.

Bei Divisionen wird der Compiler die Integer-Division durchführen, solange alle Operanden Integer sind. Eine implizite Umwandlung in eine Gleitpunkt-Zahl findet nicht statt.



Die obengenannte Regel bildet alle wichtigen Punkte über implizite arithmetische Umwandlungen ab. Ein Spezialfall sollte jedoch noch besprochen werden: Die Typ-Promotionen.

9.10.2 Die int- und double-Erweiterung (Promotion)

Die Typen `int` und `double` werden in C als Default-Typen angenommen. Für den Fall, dass beispielsweise Parameter bei Auslassungspunkten `...` übergeben werden oder bei binären Operatoren, welche die verwendeten Operanden-Typen nicht unterstützen, wird vom Compiler automatisch versucht, die Operanden zu dem Default-Typ zu erweitern, welcher den Wertebereich des Ursprungstyps abzubilden vermag. Dies wird als „**Typ-Erweiterung**“ (auf Englisch „**type promotion**“) bezeichnet und verläuft nach folgenden Regeln:

Wenn der vorzeichenbehaftete Typ `signed int` den Wertebereich des ursprünglichen Typs vollständig abbilden kann, wird der ursprüngliche Typ automatisch in den Typ `signed int` erweitert. Falls dies nicht der Fall ist, stattdessen jedoch der vorzeichenlose Typ `unsigned int` den Wertebereich des ursprünglichen Typs vollständig abbilden kann, wird der ursprüngliche Typ automatisch in den Typ `unsigned int` erweitert.

Wenn eine Gleitpunkt-Zahl als Parameter der Auslassungspunkte `...` einer Funktion übergeben wird, wird jedes Auftreten des Typs `float` automatisch zum Typ `double` erweitert.

9.11 Konvertierungsvorschriften

Im Folgenden werden die Wandlungsvorschriften zwischen verschiedenen Typen behandelt. Ein Compiler versucht grundsätzlich, die Konvertierung zwischen verschiedenen Typen – wenn immer möglich – werterhaltend zu gestalten. Sprich: Der umgewandelte Wert im Ziel-Typ entspricht – wenn immer möglich – dem ursprünglichen Wert im Ursprungstyp.

Wann immer der Wertebereich eines Ziel-Typs nicht ausreichend ist, um den zu konvertierenden Wert aufzunehmen, werden Fehler passieren, welche je nach Standard und Compiler implementationsabhängig sein können.

Hier ist zu beachten, dass diese Vorschriften sowohl bei impliziten, als auch expliziten Typumwandlungen mithilfe des Casting-Operators (siehe dazu Kapitel → 9.9.5) angewendet werden.

In den folgenden Unterkapiteln werden die verschiedenen Fälle behandelt und anschließend durch Beispiele veranschaulicht.

9.11.1 Umwandlungen zwischen Integer-Typen

Eine Konvertierung zwischen Integer-Typen ergibt genau dann den ursprünglichen arithmetischen Wert, wenn der Ziel-Typ den ursprünglichen Wert abbilden kann.

Andernfalls kann bei heutigen Compilern normalerweise davon ausgegangen werden, dass die höherwertigen Bits eines nicht abbildbaren Wertes einfach abgeschnitten werden.

In allen anderen Fällen ist das Resultat implementationsspezifisch.

9.11.2 Umwandlungen zwischen Integer- und Gleitpunkt-Typen

Wenn ein Wert aus einem Integer-Typ in einen Gleitpunkt-Typ umgewandelt wird, so werden im Prinzip als Nachkommastellen Nullen eingesetzt. Sehr große Zahlen können jedoch möglicherweise nicht exakt dargestellt werden. Das Resultat ist dann entweder der nächst höhere oder der nächst niedrigere darstellbare Wert.

Bei der Wandlung einer Gleitpunkt-Zahl in eine Integer-Zahl werden die Stellen hinter dem Komma abgeschnitten (Dies entspricht einer Rundung hin zu null). Ist die Zahl zu groß und kann nicht im Wertebereich des Integer-Typs dargestellt werden, so ist der Effekt der Umwandlung nicht definiert.

Sollen negative Gleitpunkt-Zahlen in unsigned Integer-Werte umgewandelt werden, so ist der Effekt nicht definiert.

9.11.3 Umwandlungen zwischen Gleitpunkt-Typen

Die Umwandlung zwischen verschiedenen Gleitpunkt-Typen wird den Wert erhalten, wann immer dies möglich ist. Ist es nicht exakt möglich, so wird der nächst kleinere oder größere Wert genommen. Gibt es keinen solchen Wert, so ist das Resultat nicht definiert.

9.11.4 Zwei Beispiele für Typumwandlungen

Dieses erste Beispiel demonstriert die Zuweisung eines float-Wertes an eine int-Variable:

truncate.c

```
#include <stdio.h>

int main(void) {
    double x = 4.2;
    int zahl;

    zahl = x;
    printf("zahl = %d\n", zahl);
    return 0;
}
```

Bei der Zuweisung `zahl = x` wird der Teil nach dem Dezimalpunkt abgeschnitten. Darauf weist eine Warnung des Compilers hin. Wenn das Programm dennoch ausgeführt wird, ist die Ausgabe 4.

Das zweite Beispiel zeigt implizite Typumwandlungen bei der Verwendung von `signed` und `unsigned int`:

typConvert.c

```
#include <stdio.h>

int main(void) {
    int i = -1;
    unsigned int u = 2;
    printf("%u\n", u * i);

    i = u * i;
    printf("%d\n", i);
    return 0;
}
```

Gemäß den Regeln des Standards wird vor der Produktbildung `u * i` der Operand `i` in `unsigned int` umgewandelt, da dieser als der mächtigere Typ gilt. Entsprechend ist die erste Ausgabe 4294967294.

Bei der Zuweisung `i = u * i` ist der Operand rechts somit ebenfalls vom Typ `unsigned int`. Er muss jedoch in den Typ links des Gleichheitszeichens, also in `int` gewandelt werden. Somit ist die Ausgabe hier -2.

9.12 Übungsaufgaben

Aufgabe 1: Arithmetische Operatoren

Schreiben Sie ein Programm, welches die Zahlen 1 bis 10 ausgibt und dabei zu jeder Zahl folgende Informationen mitschreibt:

- Die Summe mit der vorherigen Zahl.
- Die Differenz der Zahl zur Zahl 5.
- Das Quadrat der Zahl.
- Die Zahl geteilt durch 4.
- Ob die Zahl ohne Rest durch 3 teilbar ist.

Aufgabe 2: Vergleichs-Operatoren

Schreiben Sie ein Programm, welches die Zahlen 1 bis 10 ausgibt und dabei zu jeder Zahl ausgibt, ob sie kleiner, gleich oder größer als 5 sind. Die Zahl 7 soll jedoch nicht ausgegeben werden!

Aufgabe 3: Logische Operatoren

Schreiben Sie ein Login-Programm für eine Bank, welches eine gegebene Kontonummer (vom Typ `int`) und ein Passwort (ebenfalls vom Typ `int`) mit fest im Programmcode verankerten Werten gegenprüft. Nutzen Sie hierfür folgendes Gerüst:

kontoAbfrage.c

```
#include <stdio.h>

#define KONTONUMMER 6284
#define PIN 3141

int main(void) {
    int kontonummerEingabe = 1234;
    int pinEingabe = 777;

    // ...

    return 0;
}
```

Die beiden Variablen `kontonummerEingabe` und `pinEingabe` stellen die gegebenen Werte dar, die Konstanten `KONTONUMMER` und `PIN` sind die im Programmtext fest verankerten Werte.

- a) Verfassen Sie das Programm zuerst so, dass die beiden Eingabewerte gleichzeitig geprüft werden, sodass am Ende eine Nachricht über ein erfolgreiches Login oder Fehlschlag ausgegeben wird.
- b) Verändern Sie das Programm so, dass die beiden Eingabewerte einzeln nacheinander geprüft werden und je eine Fehlermeldung ausgegeben wird, wenn etwas davon nicht stimmt.
- c) Bauen Sie in das Programm zwei zusätzliche Kontonummern und Passwörter gemäß folgendem Code ein:

```
#define ANZAHL 3
int kontonummern[ANZAHL] = {6284, 7243, 1614};
int pins[ANZAHL] = {3141, 5974, 7777};
```

Das Programm muss für alle korrekten Kombinationen funktionieren und alle inkorrekten Kombinationen als Fehler anzeigen.

- d) Bankangestellte sind vergessliche Menschen und brauchen ein Hintertürchen. Bauen Sie die zusätzliche Kontonummer 1337 ein, welche jedes Passwort akzeptiert.
- e) Das Hintertürchen darf nicht von Hackern verwendet werden. Hacker zeichnen sich seltsamerweise dadurch aus, dass ihre Passwörter stets durch 7 teilbar sind. Verunmöglichen Sie es den Hackern, auf ein Bankkonto zuzugreifen!

Aufgabe 4: Bedingungs-Operator

Gegeben sei das folgende Programm:

condition.c

```
#include <stdio.h>

int main(void) {
    int i;
    int x = 1;
    int y = 2;
    x == y ? (i = 1) : (i = 0);
    printf("i = %d\n", i);
    return 0;
}
```

- Schreiben Sie die Zeile mit dem Bedingungs-Operator so um, dass sie direkt als Definition `int i = ...` funktioniert.
- Ersetzen Sie den Bedingungs-Operator durch eine `if`-Anweisung mit `else`-Teil.

Aufgabe 5: Der Subtraktions-Zuweisungs-Operator

Erläutern Sie, was das folgende Programm tut. Überzeugen Sie sich durch einen Programmlauf. Variieren Sie die beiden Variablen `a` und `b`.

subtraction.c

```
#include <stdio.h>

int main(void) {
    int a = 53;
    int b = 11;

    while (a >= b) {
        a -= b;
    }
    printf("??????? ist : %d\n", a);
    return 0;
}
```

Aufgabe 6: Notenrechner

- a) Analysieren Sie das folgende Programm. Variieren Sie den Wert der Variablen `note` und prüfen sie das ausgegebene Resultat.
- b) Ersetzen Sie die `else if`-Anweisungen durch den Operator `||` für das logische ODER, sodass nur ein `else` im Programm steht.
- c) Vereinfachen Sie das Programm weiter: Verwenden Sie den Bedingungs-Operator, um die Variable `bestanden`, sowie die beiden Definitionen `JA` und `NEIN` vollständig zu entfernen.

NotenRechner.c

```
#include <stdio.h>

#define SPITZE          1
#define GUT             2
#define BEFRIEDIGEND   3
#define AUSREICHEND     4
#define DURCHGEFALLEN  5
#define JA              1
#define NEIN            0

int main(void) {
    int note = 4;

    int bestanden = NEIN;
    if (note == SPITZE) bestanden = JA;
    else if (note == GUT) bestanden = JA;
    else if (note == BEFRIEDIGEND) bestanden = JA;
    else if (note == AUSREICHEND) bestanden = JA;
    else bestanden = NEIN;

    printf(bestanden ? "JA, " : "NICHT ");
    printf("WEITER SO !\n");

    return 0;
}
```


Aufgabe 7: Gebrauch verschiedener Operatoren

Gegeben sei:

```
int a = 2;  
int b = 1;  
int* ptr = &b;
```

Finden Sie ohne C-Compiler heraus, welcher Wert der Variablen `a` in den einzelnen Anweisungen zugewiesen wird. Beachten Sie dabei genau die Priorität der entsprechenden Operatoren (siehe Kapitel [→ 9.2](#)). Erläutern Sie, wie Sie auf das Ergebnis kommen. Verifizieren Sie ihr theoretisch ermitteltes Ergebnis gegebenenfalls durch einen Programmlauf.

```
a = b = 2;  
a = 5 * 3 + 2;  
a = 5 * (3 + 2);  
a *= 5 + 5;  
a %= 2 * 3;  
a = !(--b == 0);  
a = 0 && 0 + 2;  
a = b++ * 2;  
a = - 5 - 5;  
a = -(b++);  
a = 5 == 5 && 0 || 1;  
a = ((((((b + b) * 2) + b) && b || b)) == b);  
a = b ? 5 - 3 : b;  
a = 5 ** ptr;  
a = b + (++b);
```