

Ausdrücke in C

Ein großes Teilgebiet der Sprache C im Zusammenhang
von Dieter Holzhäuser

Aus vielen Einzelheiten der Sprache C, die dem Einsteiger sonderbar erscheinen mögen, formt sich ein schlüssiges Gesamtbild, wenn man sie unter dem Begriff des Ausdrucks betrachtet. Das Verständnis von C-Ausdrücken ist ein Schlüssel zum Verständnis der Sprache.

Wer mit Ausdrücken experimentieren möchte, ist mit einer handlichen Entwicklungsumgebung für Windows-C-Programme gut beraten, die z.B. unter www.smorgasbordet.com/pellesec zum kostenlosen Download zur Verfügung steht.

Für die C-Programmierung von Mikrocontrollern (MC) sind Ausdrücke genauso bedeutsam wie in jedem anderen C-Programm, aber untersuchen kann man sie kaum, weil es keine Bildschirmausgabe gibt und das Programm für jeden Test auf den MC übertragen werden muss.

Ausdruck im Überblick

Ein Ausdruck besteht im einfachsten Fall aus einer Konstanten oder Variablen, wie z.B. `56` oder `z1`. Durch Operatoren verbunden, werden sie als Operanden bezeichnet und es entstehen komplexere Ausdrücke, wie `a + 8`, `-b`, `c1 - v / 3` oder `i1 > i2`. Den C-Operatoren ist ein eigener Abschnitt gewidmet.

Bei der Programmausführung wird jeder Ausdruck zahlenmäßig bewertet (es wird daraus eine Zahl errechnet). Wie jede Konstante oder Variable hat ein Ausdruck daher einen Typ.

Darüber hinaus werden in C die Wertzuweisung an eine Variable und der Funktionsaufruf mit Hilfe eines Ausdrucks durchgeführt. Für die Wertzuweisung (oder Varianten davon) wird der Operator `=` und für den Funktionsaufruf wird der Operator `()` verwendet. Die zahlenmäßige Bewertung solcher Ausdrücke fällt deshalb aber nicht unter den Tisch.

Beispielsweise wird ein Ausdruck wie `c1 = 6` mit `c1`, also mit 6 bewertet, was mehr eine formale und weniger eine praktische Bedeutung hat. Man kann das mit `printf ("Test %i \n", c1 = 6);` nachprüfen.

Ein Funktionsaufruf wird mit der Zahl bewertet, die er hinterlässt. Falls die Funktion vom Typ void ist (keine Rückgabe eines Wertes), wird sie in einem Ausdruck nicht akzeptiert.

Verwendung

Jeder Ausdruck kann an Stelle einer Zahl stehen. Auch die Klammer der if-Anweisung `if ()` verlangt im Grunde eine Zahl, für die in der Regel ein Ausdruck steht. Das heißt, der Zahlenwert des Ausdrucks entscheidet wie die if-Anweisung verzweigt. In C wird Zahlen eines arithmetischen Typs sowie Zeigern ungleich 0 der Wahrheitswert true zugeschrieben, folglich steht die Zahl 0 für den Wahrheitswert false. Im Gegensatz zu den folgenden Beispielen enthalten Ausdrücke in der

if-Klammer üblicherweise Vergleichsoperatoren. Erklären lassen sich die Beispiele durch die Interpretation einer Zahl als Wahrheitswert und die Bewertung des Ausdrucks als Zahl:

Die Anweisung `if (a)` ist syntaktisch korrekt, ebenso `if (1)`, eine Bedingung die immer erfüllt ist. Auch `if (c1 = 6)` wird korrekt ausgeführt (Wertzuweisung an c1), aber manche Compiler warnen. Es kommt oft irrtümlich zu dieser Wertzuweisung in der if-Klammer, wenn eigentlich `if (c1 == 6)` gemeint ist, also die Abfrage auf Gleichheit. Dagegen macht `if (c1 = c2)` sogar Sinn, und zwar als Kurzform für `c1 = c2; if (c1)`.

Ein andere Qualität der Verwendung hat die Ausdrucksanweisung. Sie entsteht, indem ein Ausdruck durch ein angehängtes Semikolon zu einer Anweisung gemacht wird. (Auch ohne einen Ausdruck ist das Semikolon eine Anweisung, nämlich eine, die nichts tut. Man kann sie sogar sinnvoll anwenden, z.B. in der Warteschleife `for (n = 0; n < 10000; n++) ;`)

Jeder Ausdruck kann zu einer Ausdrucksanweisung gemacht werden. Ausdrucksanweisungen wie `b3;` und `c1 - 2;` sind daher korrekt aber kaum sinnvoll, da mit dem jeweiligen Wert des Ausdrucks nichts geschieht. Auch akzeptiert der Compiler die Anweisung `f1;`, wenn `f1` als Funktion definiert ist, nur aufgerufen wird sie nicht.

Der eigentliche Zweck von Ausdrucksanweisungen ist, die Ausdrücke von Wertzuweisungen und Funktionsaufrufen zu eigenständigen Anweisungen zu machen, z.B. `c3 = 8;` und `f1 ();` Sollte `f1 ()` auch einen Wert zurückgeben, so wird er nicht verwendet. Funktionen, deren einziger Zweck die Rückgabe eines Wertes ist, sind daher als Ausdrucksanweisung sinnlos, wie z.B. die Anweisung `sin (x);`

C-Operatoren - Prioritäten und Richtung

Operatoren sind wesentliche Bestandteile von Ausdrücken. Üblicherweise verbindet man mit dem Begriff des Operators die Verarbeitung von Zahlen und Wahrheitswerten. C-Operatoren umfassen aber auch Wertzuweisungen, Funktionsaufrufe und Speicherzugriffe, die man zunächst nicht mit Zahlenverarbeitung in Verbindung bringt.

Tatsächlich kann man aber so gut wie alles, was mit C-Operatoren möglich ist, als Zahlen- bzw. Wahrheitswert-Verarbeitung bezeichnen. Zum Beispiel werden bei Funktionsaufrufen und Zugriffen Adressen, also Zahlen, verarbeitet, die Zeiger genannt werden. C kennt keine expliziten Wahrheitswerte, sondern interpretiert Zahlen als solche.

Die Bewertung von Ausdrücken wird nicht in der Reihenfolge vorgenommen, in der die Operatoren auftreten, sondern nach deren Priorität. Zum Beispiel hat die Multiplikation Vorrang vor der Addition, wie in der Mathematik auch. In der Tabelle unten ist die höchste Priorität (Pr) mit 1 und die niedrigste mit 15 bezeichnet.

Für jede Priorität ist festgelegt, in welcher Richtung zusammengefasst wird, wenn mehrere Operatoren gleicher Priorität auftreten. Der Ausdruck `3 - 5 + 8` hat 6 zum Ergebnis. Genau wie die Mathematik fasst C die Operanden von links her zusammen, also: `(3 - 5) + 8`. Von rechts her, entstünde das Ergebnis -10 aus `3 - (5 + 8)`. Die Richtung ist in der Tabelle mit den Buchstaben L (links) oder R (rechts) unter Ri angegeben. Mehrere Zuweisungen werden von rechts her ausgeführt: `a=b=c=d;` entspricht `(a = (b = (c = d))) ;`

Die Operatoren der Priorität 2 sind unäre Operatoren, d.h. sie verarbeiten nur einen Operanden. Die übrigen verbinden zwei Operanden miteinander und sind daher binär, bis auf die bedingte Bewertung `?:`, die ternär ist und drei Operanden benötigt.

Die Operatoren `+` `-` `*` `&` gibt es unär und binär. ANSI C99 hat für manche Operatoren alternative Schreibweisen eingeführt, z.B. `and` für `&&` und `not_eq` für `!=`

Pr	Ri	Op	Wirkung	Kategorie
1	L	<code>()</code>	Funktionsaufruf	Zugriff
		<code>[]</code>	Zugriff auf Vektorelement	Zugriff
		<code>-></code>	Zugriff auf Struktur/Unionkomp.	Zugriff
		<code>.</code>	Zugriff auf Struktur/Unionkomp.	Zugriff
2	R	<code>!</code>	Logisches Nicht	Logik
		<code>~</code>	Bitweises Nicht	Bit
		<code>++</code>	Inkrement	Arithmetik
		<code>--</code>	Dekrement	Arithmetik
		<code>+</code>	Wert von	Arithmetik
		<code>-</code>	Negativer Wert von	Arithmetik
		<code>(typ)</code>	Explizite Typwandlung (Cast)	Sonstige
		<code>*</code>	Verweisoperator	Zugriff
		<code>&</code>	Adresse von	Zugriff
<code>sizeof</code>	Größe in Bytes von	Sonstige		
3	L	<code>*</code>	Multiplikation	Arithmetik
		<code>/</code>	Division	Arithmetik
		<code>%</code>	Modulo Division	Arithmetik
4	L	<code>+</code>	Addition	Arithmetik
		<code>-</code>	Subtraktion	Arithmetik
5	L	<code><<</code>	Nach links schieben aller Bits	Bit
		<code>>></code>	Nach rechts schieben aller Bits	Bit
6	L	<code><</code>	Kleiner	Vergleich
		<code><=</code>	Kleiner oder gleich	Vergleich
		<code>></code>	Größer	Vergleich
		<code>>=</code>	Größer oder gleich	Vergleich
7	L	<code>==</code>	Gleich	Vergleich
		<code>!=</code>	Ungleich	Vergleich
8	L	<code>&</code>	Bitweises Und	Bit
9	L	<code>^</code>	Bitweises Exklusiv Oder	Bit
10	L	<code> </code>	Bitweises Oder	Bit
11	L	<code>&&</code>	Logisches Und	Logik
12	L	<code> </code>	Logisches Oder	Logik
13	R	<code>?:</code>	Bedingte Bewertung	Sonstige
14	R	<code>=</code>	Wertzuweisung	Zuweisung
		<code>+=</code> u.a.	Zus.ges. Wertzuweisung, auch: <code>-- *= %= /= &= ^= = <=> >>=</code>	Zuweisung
15	L	<code>,</code>	Sequenzoperator	Sonstige

In der beispielhaften Anweisung `a = ++b * 6 + f3();` wird das Ergebnis der Multiplikation (Priorität 3) zum Wert, den `f3` hinterlässt, addiert (Priorität 4). Zuletzt kommt die Wertzuweisung an die Variable `a` mit der Priorität 14. Das Inkrement von Variable `b` (Priorität 2) hat Vorrang vor der Multiplikation. Genauso hat der Aufruf der Funktion `f3` (Priorität 1) Vorrang vor der Addition. Die hohe Priorität des Operators `()` bedeutet jedoch nicht, dass bei der Bewertung des Ausdrucks als erstes die Funktion aufgerufen wird. Wann das geschieht, ist nicht festgelegt.

Wenn man es ganz anders haben möchte, verwendet man Klammern, aber auch, wenn man sich der Prioritäten nicht ganz sicher ist. Die Anweisung zuvor mit gleichem Ergebnis und allen Klammern lautet:

`a = (((++b) * 6) + (f3()));`. Die Klammern sollen in diesem Beispiel nur nachvollziehen wie die Prioritäten wirken. Für die Praxis ist eine solche Schreibweise nicht angebracht, da die Prioritäten dieser sehr häufig verwendeten Operatoren jedem Programmierer klar sein dürften. Ähnlich ist es mit `if ((a + b) > 5) ..`, was man getrost auch ohne die innere Klammer schreiben kann, und zwar `if (a + b > 5) ..`

C-Operatoren - die wichtigsten Kategorien

Arithmetik

Mit den Operatoren dieser Kategorie werden arithmetische Zahlentypen und Zeiger verrechnet. Typen, die kleiner sind als der Typ `int` (`char` u.a.), verwandelt C generell in den Typ `int`, bevor damit gerechnet wird. Wenn aber z.B. ein `float`-Typ und ein `int`-Typ addiert werden, wird der Ausdruck mit dem `float`-Typ bewertet, d.h. der Ausdruck erhält den komplexeren der auftretenden Typen. Auch wird zum Beispiel ein Ausdruck wie `a * b`, der nur `int`-Typen enthält, mit dem Typ `unsigned int` oder `long` bewertet, damit das Ergebnis darstellbar ist.

Bit

Auch die Operanden von Bit-Operatoren sind Ganzzahlen. Nur ist der Zahlenwert weniger von Bedeutung. Zum Beispiel ist der Zweck der Anweisung `b = b | 4;` Bit 3 von `b` zu setzen und die anderen Bits unverändert zu lassen (bitweises ODER). Jedes Bit von `b` wird mit dem entsprechenden Bit von der Zahl 4 ODER-verknüpft. Es ist ohne Bedeutung, dass `b` als Zahl um 4 erhöht wird, wenn Bit 3 von `b` zuvor gleich 0 war.

Logik

Gelangen Zahlen in eine Logik-Operation werden sie als Wahrheitswert behandelt. Wie das geschieht, wurde oben schon dargestellt. Als Ergebnis einer Logik-Operation entsteht ein Wahrheitswert, der eine Zahl ist, die entweder den Wert 0 oder den Wert 1 hat. Zum Beispiel ist das Ergebnis gleich 1, wenn im Ausdruck `a || b` (logisches ODER) mindestens ein Operand ungleich 0 ist. Besser: das Ergebnis ist `true`, wenn mindestens einer der Operanden `true` ist. Die Ausgabe mit `printf ("Ergebnis %i \n", 0 || 4);` oder mit `printf ("Ergebnis %i \n", 0 && 23);` zeigt es.

Vergleich

Ausdrücke mit Vergleichsoperatoren wie z.B. in `if (b1 > 4)` ergeben den Wahrheitswert `true` (Zahl 1), wenn der Vergleich zutrifft, sonst den Wahrheitswert `false` (Zahl 0). Welchen Zahlenwert z.B. der Vergleichsausdruck `5 > 4` hat, kann man mit `printf ("Ergebnis %i \n", 5 > 4);` nachprüfen.

Zuweisung

Für Schreibfaule gibt es neben der „normalen“ Zuweisung mit dem Operator `=` die zusammengesetzte Zuweisung. Zum Beispiel entspricht `a *= 3;` der Langform `a = a * 3;`

Zugriff

Bei der Kategorie Zugriff dreht sich so gut wie alles um Zeiger. Zeiger in C sind ein mächtiges und gefürchtetes Sprachelement. Um so wichtiger ist es, den Gedanken dahinter verstanden zu haben.

Die Operatoren `*` und `&` betreffen Zeiger. Während ein Variablenname in Ausdrücken mit dem Wert der Variablen bewertet wird, ergibt die Operation `&a` den Zeiger auf `a`. Das ist die Adresse, unter der sich `a` im Speicher befindet. Davon hat man aber erst etwas, wenn man den Zeiger anweist, den Wert der Variablen preiszugeben, auf die er zeigt. Das geschieht mit dem Verweisoperator `*`. Die Anweisung `b = *&a;` (Zusammenfassung von rechts) führt die gleiche Wertzuweisung durch wie `b = a;` Da heben sich Operatoren in ihrer Wirkung auf, was wie eine nutzlose Spielerei aussieht!

Ein Ausdruck wie `&a` ist eine Zeigerkonstante, denn die Adresse von `a` ist unveränderlich. Zeigervariablen dagegen kann man eine Zeigerkonstante zuweisen (oder den Inhalt einer anderen Zeigervariablen). Zeigervariablen müssen wie alle Variablen deklariert werden, beispielsweise so: `char *p1;` oder `char a, *pb, c, *p1;` (Es ist zweckmäßig, Namen für Zeigervariablen mit dem Buchstaben `p` beginnen zu lassen, abgeleitet von pointer.)

Alle Zeiger stehen für eine Adresse. Aber sie unterscheiden sich dennoch. Sie tragen nämlich den Typ des Objektes, auf das sie zeigen, mit sich herum, falls das nicht ausgeschlossen wird. Man spricht also von char-Zeiger, int-Zeiger, float-Zeiger usw. Ein int-Zeiger zum Beispiel kann nicht in einer float-Zeigervariablen gespeichert werden.

Es sei der char-Zeiger `pa` deklariert, dem mit `pa = &a;` die Adresse der char-Variablen `a` zugewiesen wird. Dann ist die Wertzuweisung `b = *pa;` gleichbedeutend mit `b = a;`

Das Spiel kann immer weiter getrieben werden, indem man einer „Zeiger-Zeiger-Variablen“, die z.B. mit `char **ppa;` deklariert ist, mit `ppa = &pa;` den Zeiger auf die char-Zeiger-Variablen `pa` zuweist, die ihrerseits durch `pa = &a;` auf die Variable `a` zeigt. Die Anweisung `b = **ppa;` wirkt dann genauso wie `b = a;`

Wird z.B. die Zahl `x` zu einem Zeiger addiert, erhöht sich die Adresse um das `x`-fache der Typ-Größe. Zeiger können auch subtrahiert und verglichen werden.

Auf Arrays (Vektoren) und den Programmcode von Funktionen greift C mit Hilfe konstanter Zeiger zu, die sich hinter dem jeweiligen Array- bzw. Funktionsnamen verbergen. Mehr dazu siehe Anwendung von Zeigern.

Anwendung von Zeigern

Der Hauptgrund dafür, dass Zeiger so vorteilhaft einzusetzen sind, ist die Möglichkeit sie in Zeigervariablen gleichen Typs zu speichern. Ein Programmteil, der die Zeigervariablen verwendet, verarbeitet also das Objekt, dessen Zeiger aktuell in der Zeigervariablen hinterlegt ist.

Dieser Programmteil kann z.B. eine Funktion sein, deren Aufrufparameter eine Zeigervariablen ist, die den Zeiger auf das jeweils zu verarbeitende Objekt übergibt (by reference). Insbesondere kann die Funktion dann schreibend auf das Objekt zugreifen, was bei der Parameterübergabe ohne Zeiger (by value) nicht möglich ist.

Beispiel: Wenn definiert ist: `char a,b; void f1 (char *x) {*x = 3;}`, weist der Aufruf `f1 (&a)`; der Variablen a den Wert 3 zu. Oder der Variablen b, falls der Aufruf lautet: `f1 (&b)`;

Eine andere wichtige Anwendung ist, die Reihenfolge von Datensätzen zu verändern, ohne dass sie im Speicher umkopiert werden müssen (Stichwort: verkettete Listen).

Kein Zeiger

Wenn es keinen passenden Zeiger gibt, der in einer Zeigervariablen gespeichert werden könnte, wird ihr die Zahl 0 zugewiesen. C behandelt einen Zeiger, der gleich 0 ist, nicht als Adresse. Zeiger, die 0 sein könnten, müssen überprüft werden, bevor sie verwendet werden, um einen Laufzeitfehler zu vermeiden, z.B. mit `if (p1)`

void-Zeiger

Die Bindung der Zeiger an einen Typ kann manchmal störend sein. In void-Zeigervariablen, z.B. deklariert als `void *p1`; können Zeiger beliebigen Typs gespeichert werden, die dann ihren ursprünglichen Typ verlieren und den „Typ“ void annehmen. Da sich mit void-Zeigern kaum etwas anfangen lässt, muss sich das Programm den ursprünglichen Typ auf irgendeine Art merken und ihn vor Verwendung des Zeigers mit einer expliziten Typwandlung (cast) wieder herstellen.

Wenn z.B. in einer Strukturkomponente Zeiger verschiedenen Typs unterzubringen sind, deklariert man diese Komponente als void-Zeiger und eine weitere, die den wirklichen Typ z.B. als Kennzahl enthält. Beim Zugriff erlaubt diese Kennzahl, den ursprünglichen Typ zurück zu gewinnen. Zum Beispiel wird der Zeiger in der void-Zeigervariablen p1 mit `p2 = (float*)p1`; zum float-Zeiger, der der float-Zeigervariablen p2 zugewiesen wird.

Zeiger und Arrays

Auch hinter Arrays verbergen sich Zeiger. Ja, man kann sagen, dass der Operator `[]` zum Zugriff auf die Elemente eines Arrays eine vereinfachte Zeigerarithmetik realisiert, weil der Name eines Arrays für einen konstanten Zeiger steht, nämlich für die Anfangsadresse des Arrays. Der Zugriff mit `[i]` auf das i. Element des Arrays kann daher genauso mit dem Verweisoperator `*` gemacht werden. Die folgenden Entsprechungen sind nur dazu gedacht, das Verständnis zu fördern. In der Praxis wird man für den Zugriff auf Elemente von Arrays immer den Operator `[]` verwenden.

Wenn z.B. v1 als Array deklariert ist, dann verweisen sowohl `v1[i]` als auch `*(v1+i)` auf das i. Element von v1. Die Zeigerarithmetik besteht darin, dass der Zeiger v1 vor dem Zugriff um das i-fache der Typ-Größe erhöht wird. Gut möglich, dass auch der Compiler den Operator `[]` in dieser Weise auflöst.

Umgekehrt kann der Operator `[]` auch unabhängig von einem Array den Operator `*` ersetzen. Wenn z.B. a eine long-Zahl ist, wird mit

`*((char*)&a + i)` auf das i. Byte von a zugegriffen.

Genauso geht das mit `((char*)&a)[i]`.

Die Wandlung des long-Zeigers `&a` in einen char-Zeiger ist erforderlich, weil auf einzelne Bytes von a zugegriffen werden soll.

Die Entsprechung wäre keine, wenn sie nicht auch für mehrdimensionale Arrays gelten würde:

Wenn das zweidimensionale Array

```
char txt [ ] [6] = { "abcd", "12345", "nmop" };
```

definiert ist, dann geben sowohl

```
printf("%c", txt[1][2] );
```

 als auch

```
printf("%c", *(*(txt+1)+2) );
```

 das Zeichen '3' aus.

Auch ohne Array funktioniert die mehrfache Anwendung von [] .

Mit den Definitionen

```
char a = 56; char* pa = &a; char** ppa = &pa;
```

 gibt

```
printf("%i", ppa[0][0]);
```

 die Zahl 56 aus,

genauso wie

```
printf("%i", *ppa[0]);
```

 und

```
printf("%i", **ppa );
```

Zeiger und Strings

Der konstante String steht in Deklarationen für den konstanten char-Zeiger, der auf sein erstes Zeichen zeigt. Also kann man z.B. definieren:

```
char * pno = "NO", *pyes = "YES" , *ptr ;
```

Je nach Situation wird zugewiesen

```
pstr = pno;
```

 oder

```
pstr = pyes;
```

 Dann genügt

```
printf ( pstr );
```

 für die Ausgabe des betreffenden Strings.

Zeiger auf Funktionen

Auch Funktionsnamen sind konstante Zeiger. Sie zeigen auf das erste Byte des Programmcodes der Funktion. Mit dem Operator () wird die Funktion aufgerufen.

Zeigervariablen für Zeiger auf Funktionen werden z.B. so deklariert:

```
int (*pf) ( char a );
```

 . (Achtung: die erste Klammer darf nicht vergessen werden, sonst würde eine Funktion pf deklariert, die int-Zeiger hinterlässt.)

Die konstanten Zeiger f1 und f2 können jeweils mit

```
pf = f1;
```

 oder

```
pf = f2;
```

 in pf gespeichert werden, wenn die Funktionen

```
int f1 ( char b)
```

 und

```
int f2 ( char c)
```

 definiert wurden. Die Funktionen müssen bezüglich ihrer Parameterliste und ihrem Rückgabotyp mit der Zeigervariablen übereinstimmen.

Mit

```
(*pf) (..)
```

 wird dann entweder

```
f1 ( )
```

 oder

```
f2 ( )
```

 aufgerufen. Achtung: die erste Klammer übertrumpft die Priorität 1 des Aufruf-Operators () . Ohne die Klammer würde er auf pf wirken, was aber der Compiler nicht zulässt.

Soweit der kleine Exkurs zu den Ausdrücken von C.