C-Vorrangregeln

Version 1.0 — 14.11.2003

eMail: tb@ostc.de
Web: www.ostc.de

Schutzgebühr: 3 Euro

Hinweise auf inhaltliche Fehler, Schreibfehler und unklare Formulierungen sowie Ergänzungen, Kommentare, Wünsche und Fragen können Sie gerne an den Autor richten:

OSTC Open Source Training and Consulting GmbH
Thomas Birnthaler
eMail: tb@ostc.de
Web: www.ostc.de

Vorrangregeln

- Das Verständnis der Vorrangregeln der C-Operatoren ist wichtig. Bei falscher Interpretation von Ausdrücken sind die herrlichsten Fehler möglich. Zum schnellen Nachsehen kann z.B. die folgende Liste am Bildschirm angebracht werden.

Typ	Prio	Operatoren	Assoziativität
	15	`[] . -> ()`	links nach rechts
Einstellig	14	`! ~ ++ -- & * (type) sizeof + -`	**rechts nach links**
Arithmetisch	13	`* / %`	links nach rechts
	12	`+ -`	links nach rechts
	11	`<< >>`	links nach rechts
Vergleich	10	`< <= > >=`	links nach rechts
	9	`== !=`	links nach rechts
Bitweise	8	`&`	links nach rechts
	7	`^`	links nach rechts
	6	`|`	links nach rechts
Logisch	5	`&&`	links nach rechts
	4	`||`	links nach rechts
Bedingung	3	`?:`	**rechts nach links**
Zuweisung	2	`= += -= *= /= %= &= ^= |= <<= >>=`	**rechts nach links**
Sequenz	1	`,`	links nach rechts

- Jedem Operator ist eine **Priorität** zugeordnet, in einem Ausdruck benachbarte Operatoren unterschiedlicher Priorität werden nach abfallender Priorität ausgeführt. Benachbarte Operatoren gleicher Priorität werden gemäß ihrer **Assoziativität** von links nach rechts oder umgekehrt ausgeführt.

- Trotz der Vielzahl an Operatoren (45 bzw. 47 Stück) und der 15 Vorränge ist die Tabelle leicht zu behalten. Die Vorränge und die Assoziativitäten wurden nämlich (fast) so gewählt, wie man es intuitiv erwarten würde, bis auf 2 Ausnahmen: Die Bit-Operatoren `| & ^` und die Bit-Shift-Operatoren `<< >>`. Tip: wenn man diese **immer klammert**, dann passiert auch hier nichts.

- Beispiele für die Auswertungsreihenfolge:

```
a * b + c / d       entspricht     (a * b) + (c / d)
a < b && c > d      entspricht     (a < b) && (c > d)
*a[1]               entspricht     *(a[1])
*a++                entspricht     *(a++)
&a->b               entspricht     &(a->b)
*--a                entspricht     *(--a)
```

- **Assoziativität von links nach rechts** heißt: Sind hintereinanderstehende Ausdrücke über Operatoren der gleichen Priorität verknüpft (ohne explizite Klammerung), so wird mit der Auswertung beim am weitesten links stehenden Operator begonnen:

```
a + b - c + d       entspricht     ((a + b) - c) + d
a->b.c->d           entspricht     ((a->b).c)->d
a[1][2]             entspricht     (a[1])[2]
```

- Entsprechend gilt für **Assoziativität von rechts nach links**:

```
a = b = c = d       entspricht     a = (b = (c = d))
a ? b : c ? d : e   entspricht     (a ? b : (c ? d : e))
**a                 entspricht     *(*a)
```

- Die **Shift-Operatoren** << und >> können zwar als Multiplikation mit 2 und Division durch 2 betrachtet werden, haben aber einen geringeren Vorrang als die Additionsoperatoren + und −. Beim Rechnen mit geshifteten Werten also klammern:

```
a << 4 + b >> 8     entspricht     (a << (4 + b)) >> 8
```

- Ebenso sind die **Bit-Operatoren** &, | und ^ von geringerem Vorrang als die Vergleichsoperatoren. Beim Vergleich von isolierten Bits also klammern.

```
x & 0x11 > 0        entspricht     x & (0x11 > 0)
```

- Die Berechnung eines über die **Logik-Operatoren** && und || verknüpften logischen Ausdrucks erfolgt schrittweise von links nach rechts und wird sofort abgebrochen, wenn das Endergebnis *wahr* oder *falsch* feststeht (**verkürzte Auswertung, short circuit evaluation**). D.h. folgende for-Schleife ist korrekt formuliert (es findet kein Zugriff auf das nicht existierende Arrayelement arr[MAXLEN] statt):

```
char* arr[MAXLEN];

for (i = 0; i < MAXLEN && arr[i] != NULL; ++i)
    arr[i] = ...;
```

- Beim **Bedingten Ausdruck** ?: wird der Vergleichsausdruck und *genau einer* der beiden Wertausdrücke berechnet:

```
max = (a > b) ? ++a : ++b;   /* entweder a oder b inkrementiert */
```

  Dies entspricht der etwas längeren Formulierung:

```
if (a > b)
    ++a;
else
    ++b;
```

- Ein bedingter Ausdruck kann im Gegensatz zu einer if-Abfrage auch innerhalb eines anderen Ausdrucks oder als Funktionsargument verwendet werden:

```
cp = (x % 2 == 0) ? "gerade" : "ungerade";

printf("Zeiger cp zeigt auf %s\n, (cp == NULL) ? "(null)" : cp);
```

  Das Klammern der Bedingung in diesen Beispielen ist zwar nicht notwendig, erhöht aber die Übersichtlichkeit.

- Bei der Verwendung von Zeigern auf Datenstrukturen muss man beim Komponentenzugriff ebenfalls aufpassen. `*p_struct.elem` liefert wegen dem Vorrang `.` vor `*` nicht das richtige Ergebnis. Entweder man klammert `(*p).elem` oder man verwendet den „Pfeiloperator“ `p->elem`.

- Nur die Operatoren `&&`, `||`, `?:` und `,` (Komma) sowie die Zuweisungen garantieren eine **feste Auswertungsreihenfolge** ihrer Operanden. Bei allen anderen Operatoren ist die Auswertungsreihenfolge **compilerabhängig**. Insbesondere die Aufrufreihenfolge von als Operanden verwendeten Funktionen und die Auswertungsreihenfolge von Inkrement-/Dekrementoperationen ist compilerabhängig.

- Durch Aufspalten eines Ausdrucks in **Unterausdrücke mit Zwischenvariablen** kann eine bestimmte Auswertungsreihenfolge erzwungen werden. Ob z.B. im folgenden Beispiel zuerst *g()* und dann *h()* ausgewertet wird, bevor ihre beiden Ergebnisse addiert werden, oder umgekehrt, ist compilerabhängig.

  ```
  y = g(x) + h(x);
  ```

  Im folgenden Beispiel wird bei gleichem Ergebnis auf jeden Fall zuerst *g()* und dann *h()* ausgewertet.

  ```
  y = g(x);
  y += h(x);
  ```

- *Im Zweifelsfall komplexe Ausdrücke lieber in Teilausdrücke zerlegen oder lieber eine Klammer zuviel als zuwenig verwenden, um den gewünschten Vorrang auszudrücken. Allzu große Klammerngebirge sind allerdings auch nicht mehr überschaubar.*