C++-Vorrangregeln

Version 1.0 — 18.10.2014

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

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

Vorrangtabelle

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

Typ	Prio	Operatoren	Assoziativität
Bereichsauflösung	17	`::`	links → rechts
Einstellig	16	`[] . -> () ++ -- typeid *_cast`	links → rechts
	15	`! ~ ++ -- & * (TYP) sizeof + -`	**rechts → links**
		`new new[] delete delete[] alignof`	**rechts → links**
Zeiger auf Klassenmitglied-Auswahl	14	`.* ->*`	links → rechts
Arithmetisch	13	`* / %`	links → rechts
	12	`+ -`	links → rechts
Bitshift	11	`<< >>`	links → rechts
Vergleich	10	`< <= > >=`	links → rechts
	9	`== !=`	links → rechts
Bitweise	8	`&`	links → rechts
	7	`^`	links → rechts
	6	`|`	links → rechts
Logisch	5	`&&`	links → rechts
	4	`||`	links → rechts
Bedingter Ausdruck	3	`?:`	**rechts → links**
Initialisierung	2	`{...}`	**rechts → links**
Zuweisung		`= += -= *= /= %= &= ^= |= <<= >>=`	**rechts → links**
Ausnahme	1	`throw`	**rechts → links**
Sequenz	0	`,`	links → rechts

- Jedem Operator ist eine **Priorität** zugeordnet, in einem Ausdruck eingesetzte 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 60 Operatoren und 18 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 `<< >>`.
  Tipp: wenn man diese **immer klammert**, dann passiert auch hier nichts.
- Beispiele für die Auswertungsreihenfolge gemäß Priorität:

```
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 die **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)
```

Ungewöhnliche Operatoren

Ungewöhnlich gegenüber anderen Programmiersprachen sind folgende Operatoren:

- Der **logische Negations-Operator** `!` invertiert einen Booleschen Wert. Der **bitweise Invertierungs-Operator** `~` invertiert alle Bits eines Wertes:

```
a = !b;     /* wahr -> falsch, falsch -> wahr */
a = ~b;     /* 10001110 -> 01110001 */
```

- Die **Inkrement/Dekrement-Operatoren** `++ --` haben eine **Präfix-** und eine **Postfix-Form**. Werden beide Operatoren nur zusammen mit einer Variablen verwendet, besteht kein Unterschied zwischen ihnen:

```
++i;     /* i = i + 1 */
i++;     /* i = i + 1 */
--i;     /* i = i - 1 */
i--;     /* i = i - 1 */
```

  Werden die beiden Operatoren in einem **Ausdruck** verwendet, dann unterscheiden sich die Wirkungen der beiden Varianten:

```
a = ++i;       /* i = i + 1;   a = i     */
a = i++;       /* a = i;       i = i + 1 */
a = --i;       /* i = i - 1;   a = i     */
a = i--;       /* a = i;       i = i - 1 */
```

- Der **sizeof-Operator** `sizeof` ermittelt den Platzbedarf eines Datentyps oder einer Variablen in Byte:

```
int var;
i = sizeof(int);
i = sizeof(var);
```

- Der (explizite) **Cast-Operator** `(TYP)` oder `TYP(...)` (nur elementare Typen oder Typnamen) konvertiert einen Datentyp in einen anderen:

```
float f;
int i;
f  = (int) i;          /* int -> float */
f  = int(i);           /* int -> float */

char *cp;
int  *ip;
cp = (char*) ip;       /* char* -> int* */
```

- Der **Modulo-Operator** `%` ist nur auf Ganzzahlen anwendbar und errechnet den ganzzahligen **Divisionsrest**:

```
17 % 5      /* ergibt 2 wg. 17 : 5 = 3 Rest 2 */
```

- **Kleiner/Größer-Vergleiche** haben einen **höheren Vorrang** als die **Gleich/Ungleich-Vergleiche**. Dies erlaubt folgende Konstruktion (deren Bedeutung sich nicht unbedingt leicht erschließt):

```
a <= b == b <= c     /* (a <= b && b <= c) || (a > b && b > c) */
```

- Der **Bedingte Ausdruck** `?:` ist ein dreiwertiger Operator, er wertet den 2. oder 3. Ausdruck abhängig vom Wert des 1. Ausdrucks aus.

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

- Die 10 Operatoren `+ - * / % & ^ | << >>` sind als **verkürzte Operation** mit der **Zuweisung** kombinierbar:

```
i += 1;      /* i = i + 1 */
i -= 2;      /* i = i - 2 */
i *= 3;      /* i = i * 3 */
i /= 4;      /* i = i / 4 */
i %= 5;      /* i = i % 5 */
i &= 6;      /* i = i & 6 */
i ^= 7;      /* i = i ^ 7 */
i |= 8;      /* i = i | 8 */
i <<= 9;     /* i = i << 9 */
i >>= 10;    /* i = i >> 10 */
```

- Das **Komma `,`** ist ein **schwaches Semikolon**, das — im Gegensatz zu diesem — in einem Ausdruck erlaubt ist. Er sorgt dafür, dass erst der linke Ausdruck und dann der rechte ausgewertet wird. Ergebnis des ganzen Ausdrucks ist der Wert des rechten Ausdrucks. Da der Komma-Operator den niedrigsten Vorrang hat, werden alle anderen Operatoren vor ihm ausgewertet, solange nicht geeignet geklammert wird.

```
i = 10.2;           /* i = 10 (Dezimalpunkt!) */
i = 10,2;           /* i = 10   */
i = (10,2);         /* i = 2    */
j = (i = 10,2);     /* i = 10, j = 2    */
j = (i = (10,2));   /* i = 2, j = 2    */
```

Typischer Anwendungsfall des Komma-Operators sind `for`-Schleifen mit zwei oder mehr Laufvariablen oder Makros, die mehrere Anweisungen zusammenfassen.

```
for (i = 0, j = 0; i > 0 && j > 0; ++i, ++j) ...
#define  INCBOTH(i,j)  ++i, ++j
```

- Die **Bit-Operatoren** `& | ~` nicht mit den **Logischen/Booleschen Operatoren** `&& || !` verwechseln. Erstere arbeiten auf den einzelnen Bits der Werte, letztere arbeiten nur mit Wahrheitswerten nach folgender Logik:
  - **Falsch/False** = Wert 0
  - **Wahr/True** = Alle anderen Werte (ungleich 0).

Operatoren mit problematischer Priorität

- 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)
```

- Bei der Verwendung eines Zeiger `ps` auf eine Datenstruktur muss man beim **Komponentenzugriff** ebenfalls aufpassen. `*ps.elem` liefert wegen dem Vorrang `.` vor `*` nicht das richtige Ergebnis. Entweder man klammert `(*ps).elem` oder man verwendet den **Pfeiloperator** `ps->elem`.

Operatoren mit fixer Auswertungsreihenfolge

- Die Berechnung eines über die **Logik-Operatoren** `&&` und `||` verknüpften logischen Ausdrucks erfolgt schrittweise gemäß Vorrang und von links nach rechts und wird sofort abgebrochen, wenn das Endergebnis *wahr* oder *falsch* feststeht (**verkürzte Auswertung, shortcut/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 zuerst der Vergleichsausdruck und dann *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.

- Nur die Operatoren `&&`, `||`, `?:` und `,` (Komma) sowie die Zuweisungen `= += -= *= /= %= &= ^= |= <<= >>=` garantieren eine **feste Auswertungsreihenfolge** ihrer Operanden (in den folgenden Beispielen ist die garantierte Auswertungsreihenfolge durch 1 bzw. 2 angedeutet).

  ```
  1 && 2
  1 || 2
  1 ? 2a : 2b
  1, 2
  2 = 1
  2 *= 1
  ```

  Bei allen anderen Operatoren ist die Auswertungsreihenfolge **compilerabhängig**. Insbesondere die Aufrufreihenfolge von als Operanden verwendeten Funktionen und die Auswertungsreihenfolge von Inkrement-/Dekrementoperationen ist compilerabhängig.

  ```
  f() + g()        /* Aufruf-Reihenfolge von f und g unbestimmt */
  ++i + i--        /* Auswertung-Reihenfolge von ++i und i-- unbestimmt */
  ```

Auswertungs-Reihenfolge beeinflussen

- 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);
  ```

- **Tipp**: *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.*