Operatoren und Ausdrücke

- Priorität und Assoziativität von Operatoren
- Auswertungsreihenfolge der Operanden
- Implizite Typkonvertierung (Teil 1)
- Arithmetische Operatoren
- Bitoperatoren
- Vergleichsoperatoren
- Der Bedingungsoperator `?:`
- Implizite Typkonvertierung (Teil 2)
- Explizite Typkonvertierung
- Zuweisungsoperatoren
- Der Kommaoperator
- Konstantenausdrücke

Operatoren und Ausdrücke

Ein **Ausdruck** besteht aus

- Operanden,
- Operatoren und
- Klammern,

die zusammen eine Berechnungsvorschrift beschreiben.

Operanden können

- Variablen,
- Konstanten sowie
- Methodenaufrufe

sein.

Bei den **Operatoren** gibt es verschiedene Stelligkeiten:

- **Unäre** Operatoren wirken auf einen Operanden.
- **Binäre** Operatoren stehen zwischen ihren beiden Operanden.
- **Ternäre** Operatoren haben drei Operanden.

Operatoren und Ausdrücke

Priorität und Assoziativität von Operatoren

Die **Priorität** (oder der Vorrang) von Operatoren bestimmt ihre Auswertungsreihenfolge.

Beispiel: *„Punktrechnung vor Strichrechnung“*

- Der Ausdruck `5 * 3 - 3` ergibt den Wert `12` und nicht `0`. Der Operatorvorrang kann durch Klammerung geändert werden: `5 * (3 - 3)` ergibt `0`.

Erscheinen zwei Operatoren gleicher Priorität nebeneinander, entscheidet die **Assoziativität** (die Auswertungsrichtung) des Operators, der zuerst ausgeführt wird.

Beispiel:

- Weil Addition und Subtraktion linksassoziativ sind, ist der Ausdruck `9 - 5 + 2` äquivalent zu `(9 - 5) + 2` und ergibt den Wert 6 und nicht 2.

Operatoren und Ausdrücke

Priorität und Assoziativität von Operatoren

In der folgenden Tabelle sind alle verfügbaren Operatoren in absteigender Priorität aufgeführt.

Die Tabelle enthält

- der Priorität (Prio.),
- die Stelligkeit (Stl.) und
- die Auswertungsrichtung (Rtg.)

der Operatoren.

Bei der Auswertungsrichtung bedeutet

⇐ *„von rechts nach links“* (rechtsassoziativ) und
⇒ *„von links nach rechts“* (linksassoziativ).

Operatoren und Ausdrücke

Priorität und Assoziativität von Operatoren

Prio.	Operation	Stl.	Operator	Rtg.
0	Bereichsoperator		`::`	
1	Typumwandlung Laufzeit-Typinformation zur Laufzeit geprüfte Konvertierung zur Übersetzungszeit geprüfte Konv. ungeprüfte Konvertierung `const`-Konvertierung		**`Typname (Ausdruck)`** **`typeid`** **`dynamic_cast`** **`static_cast`** **`reinterpret_cast`** **`const_cast`**	
1	Elementselektion Indexoperator Funktionsaufruf Postinkrement Postdekrement	unär	**`., ->`** **`[]`** **`()`** **`++`** **`--`**	⇒

2	Objektgröße, Typgröße
	Präinkrement
	Prädekrement
	logische Negation
	Bitkomplement
	positives Vorzeichen
	negatives Vorzeichen
	Adressoperator
	Inhaltsoperator
	Erzeugung
	Feld-Erzeugung
	Freigabe
	Feld-Freigabe
	Typumwandlung
		unär	**`sizeof`**
			`++`
			`--`
			`!`
			`~`
			`+`
			`-`
			`&`
			`*`
			`new`
			`new[]`
			`delete`
			`delete[]`
			`(Typausdruck) Ausdruck`
				⇐

3	Elementselektion	binär	**`.*, ->*`**	⇒

Operatoren und Ausdrücke

Priorität und Assoziativität von Operatoren

Prio.	Operation	Stl.	Operator	Rtg.

4	Multiplikation
	Division
	Rest
		binär	`*`
			`/`
			`%`
				⇒

5	Addition Subtraktion	binär	`+` `-`	⇒
6	Linksschieben Rechtsschieben	binär	`<<` `>>`	⇒
7	Vergleiche	binär	`<, >, <=, >=`	⇒
8	Gleichheit Ungleichheit	binär	`==` `!=`	⇒
9	bitweises UND	binär	`&`	⇒
10	bitweises exklusives ODER	binär	`^`	⇒
11	bitweises ODER	binär	`|`	⇒
12	logisches UND	binär	`&&`	⇒
13	logisches ODER	binär	`||`	⇒
14	Zuweisungen **op** ∈ { `*, /, %, +, -,` `&, ^, |, <<, >>` }	binär	`=` `op=`	⇐
15	bedingter Ausdruck	ternär	`? :`	⇐
16	Ausnahme werfen		`throw`	
17	Kommaoperator (oder Sequenzoperator)	binär	`,`	⇒

Operatoren und Ausdrücke

Auswertungsreihenfolge der Operanden

Auf gleicher Prioritätsstufe wird ein Ausdruck von links nach rechts ausgewertet, mit Ausnahme der unären, des ternären und der Zuweisungsoperatoren, die von rechts abgearbeitet werden. Grundsätzlich werden jedoch zuerst die Klammern ausgewertet.

Beispiel:

```
a = b + c + d;   entspricht   a = ((b + c) + d);
a = b = c = d;   entspricht   a = (b = (c = d));
```

Die Reihenfolge der Auswertung von Unterausdrücken untereinander ist *undefiniert*. Daher sollten Ausdrücke vermieden werden, die einen Wert sowohl verändern als auch benutzen.

Beispiel:

```
int total = 0;
int sum = (total = 3) + (++total);
```

Der Wert von `sum` ist abhängig von der Reihenfolge, in der die Unterausdrücke in den Klammern berechnet werden.

Operatoren und Ausdrücke

Implizite Typkonvertierung

Alle Ausdrücke besitzen einen **Typ**.

Der Typ eines Ausdrucks wird von seinen Bestandteilen und der Semantik seiner Operatoren bestimmt.

Die meisten binären Operatoren verlangen als linken und rechten Operanden Werte desselben Typs.

Der C++-Compiler führt bei Operanden unterschiedlicher Datentypen intern eine Typkonvertierung durch, bei der die Operanden auf den Typ des „*kompliziertesten*“ Operanden erweitert werden.

Beispiele:

```
short + int      =>  int-Addition
int * long       =>  long-Multiplikation
double – long    =>  double-Subtraktion
```

Die folgenden Umwandlungen werden für Operanden eines zweistelligen Operators durchgeführt, um sie auf einen gemeinsamen Typ zu bringen, der dann als Ergebnis-Typ verwendet wird:

Operatoren und Ausdrücke

Implizite Typkonvertierung

1. Falls ein Operand den Typ `long double` hat, wird auch der andere in `long double` umgewandelt.

- Ansonsten wird, falls einer der Operanden den Typ `double` hat, auch der andere in `double` umgewandelt.
- Ansonsten wird, falls einer der Operanden den Typ `float` hat, auch der andere in `float` umgewandelt.
- Ansonsten wird für beide Operanden *ganzzahlige Promotion* durchgeführt.

2. Falls dann ein Operand den Typ `unsigned long` hat, wird auch der andere in `unsigned long` umgewandelt.

- Falls ansonsten einer der Operanden `long int` und der andere `unsigned int` ist, wird entweder `unsigned int` in `long int` umgewandelt (falls `long int` alle Werte von `unsigned int` darstellen kann) oder es werden beide in `unsigned long int` umgewandelt.
- Ansonsten wird, falls einer der Operanden den Typ `long` hat, auch der andere in `long` umgewandelt.
- Ansonsten wird, falls einer der Operanden den Typ `unsigned` hat, auch der andere in `unsigned` umgewandelt.
- Ansonsten sind beide Operanden vom Typ `int`.

Operatoren und Ausdrücke

Implizite Typkonvertierung

Ganzzahlige Promotions werden verwendet, um aus kürzeren ganzzahligen Typen `int`s zu erzeugen.

- Ein `char`, `signed char`, `unsigned char`, `short int` oder `unsigned short int` wird in einen `int` umgewandelt, falls dieser alle Werte des Quelltyps darstellen kann. Ansonsten werden sie in einen `unsigned int` umgewandelt.

- Ein `wchar_t` oder ein Aufzählungstyp werden in den ersten der Typen `int`, `unsigned int`, `long` oder `unsigned long` umgewandelt, der alle Werte des Quelltyps darstellen kann.

- Ein Bitfeld wird in einen `int` umgewandelt, wenn ein `int` alle Werte des Bitfelds darstellen kann. Ansonsten wird es, wenn dies ausreicht, in ein `unsigned int` umgewandelt. Ansonsten findet hier keine ganzzahlige Promotion statt.

- Ein `bool` wird in einen `int` umgewandelt, wobei `false` zu `0` und `true` zu `1` wird.

Operatoren und Ausdrücke

Implizite Typkonvertierung

evtl. Probleme bei Umwandlung ganzzahlig => reell

- Konvertierungen von ganzen Zahlen in Gleitkommazahlen sind mathematisch korrekt, sofern es die Hardware erlaubt.
- Wenn der ganzzahlige Wert nicht exakt als Wert des Gleitkommatyps dargestellt werden kann, geht Genauigkeit verloren.

Beispiel:

```
int i = 1234567890;
int j = float(1234567890);
cout << i << endl;
cout << j << endl;
```

Ausgabe:

```
1234567890
1234567936
```

`j` erhält den Wert `1234567936`, wenn eine Maschine sowohl für `int` als auch für `float` 32 Bit verwendet.

Operatoren und Ausdrücke

Im Folgenden wird für die einzelnen Operatoren angegeben, von welchen Datentypen die Operanden sein dürfen und welchen Datentyp dann das Ergebnis hat.

- Mit **ganz** werden die Datentypen `long`, `int`, `short`, `char` (alle sowohl `signed` als auch `unsigned`) sowie `bool` bezeichnet.

  Der Ergebnistyp ist wegen der ganzzahligen Promotions „mindestens“ `int`.
  (In den folgenden Tabellen bedeutet „**`>=int`**“ „mindestens vom Typ `int`“.)

- Mit **reell** werden die Datentypen `float` , `double` und `long double` bezeichnet.

  Der Ergebnistyp ist `long double`, wenn mindestens ein Operand vom Typ `long double` ist; `double`, wenn mindestens ein Operand vom Typ `double` ist; und sonst ist der Ergebnistyp immer `float`.

Operatoren und Ausdrücke

Arithmetische Operatoren

Pri.	Operator	Operanden	Ergebnis	Bemerkung
2	Vorzei-chen **+**	ganz reell	**>= int** reell	positives Vorzeichen
5	**+**	ganz, ganz reell, reell	**>= int** reell	Summe Summe
2	Vorzei-chen **-**	ganz reell	**>= int** reell	negatives Vorzeichen
5	**-**	ganz, ganz reell, reell	**>= int** reell	Differenz Differenz
4	*****	ganz, ganz reell, reell	**>= int** reell	Produkt Produkt
4	**/**	ganz, ganz reell, reell	**>= int** reell	ganzzahliger Quotient Quotient
4	**%**	ganz, ganz	**>= int**	ganzzahliger Rest

Bei der Division / und der Modulo-Operation % darf der zweite Operand nicht Null sein.

Falls dies doch der Fall ist, wird der Programmlauf mit einer entsprechenden Ausnahme abgebrochen.

Operatoren und Ausdrücke

Arithmetische Operatoren

Beispiel:

```
#include <iostream>
using namespace std;

int main()
{
  float quotient, z = 12.34E5f,
        q = -3e1f, rest, wert;
  int zaehler = 13;

  cout << zaehler / 4 << "   "
       << zaehler % 4 << endl;

  quotient = zaehler / 4;
  cout << quotient << endl;

  quotient = zaehler / 4.0F;
  cout << quotient << endl;

  // Fehler: modulo nur fuer
  // ganzzahlige Operanden definiert
  // rest = z % q;
}
```

Ausgabe:

```
3   1
3
3.25
```

Operatoren und Ausdrücke

Arithmetische Operatoren

Beispiel:

```
#include <iostream>
using namespace std;

int main()
{
    float zaehler = 2.5E27, nenner = 3.4e-12;
    int oben = 12, unten = 0;

    // Floatingpoint-Division durch Null
    cout << zaehler / 0.0;

    // Zahlenueberlauf bei float
    cout << zaehler / nenner;

    // Integer-Division durch Null
    cout << oben / unten;
}
```

Alle drei Divisionen führen zu Laufzeitfehlern.

Operatoren und Ausdrücke

Arithmetische Operatoren

Beachten Sie beim Rechnen mit ganzen Zahlen die *begrenzte Genauigkeit*.

Beispiel:

```
#include <iostream>
using namespace std;

int main()
{
    int a,b,c;

    // Annahme: 16 Bit pro int
    if(sizeof(int) == 2)
    {
        a = 50; b = 1000; c = a*b;
        cout << c;
        // Ausgabe -15536 statt 50000
    }

    // Annahme: 32 Bit pro int
    if(sizeof(int) == 4)
    {
        a = 50000; b = 1000000; c = a*b;
        cout << c;
        // Ausgabe -1539607552 statt 50000000000
    }
}
```

Das Resultat der Operation wird betragsmäßig zu groß, es tritt ein *Überlauf* (engl. *overflow*) auf.

Operatoren und Ausdrücke

Arithmetische Operatoren

Reelle Zahlen können in der Regel nicht exakt dargestellt werden. Die Folgen können sein:

- Bei der Subtraktion zweier fast gleich großer Werte heben sich die signifikanten Ziffern auf. Die Differenz wird ungenau (*numerische Auslöschung*)
- Die Division durch betragsmäßig zu kleine Werte ergibt einen *Überlauf* (engl. *overflow*).
- Wenn der Betrag des Ergebnisses zu klein ist, um mit dem gegebenen Datentyp darstellbar zu sein, tritt eine *Unterschreitung* (engl. *underflow*) auf. Das Ergebnis wird dann gleich 0 gesetzt.
- Ergebnisse können von der Reihenfolge der Berechnungen abhängen.

Beispiel:

```
#include <iostream>
using namespace std;

int main()
{
   float a=1.234567E-7, b=1.000000, c=-b, s1, s2;
   s1 = a + b;
   s1 += c;
   s2 = a;
   s2 += b + c;
   cout << s1 << '\n';     // 1.19209e-07
   cout << s2 << '\n';     // 1.23457e-07
}
```

Operatoren und Ausdrücke

Bitoperatoren

Schiebeoperatoren

Pri.	Operator	Operanden	Ergebnis	Bemerkung
6	**<<**	ganz, ganz	**>= int**	Linksschieben
6	**>>**	ganz, ganz	**>= int**	Rechtsschieben

Das Bitmuster des ersten Operanden wird um so viele Stellen nach links bzw. rechts geschoben, wie im zweiten Operanden angegeben ist.

- Beim Linksschieben werden von rechts Nullen nachgezogen.

- Beim Rechtsschieben werden von links Nullen nachgezogen, falls der erste Operand `unsigned` ist.

- Falls beim Rechtsschieben der Typ des ersten Operanden vorzeichenbehaftet ist, wird links entweder das Vorzeichenbit kopiert oder es werden Nullbits eingefügt - je nach C++-System.

Operatoren und Ausdrücke

Schiebeoperatoren

Beispiel:

```
#include <iostream>
using namespace std;

int main()
{
  int a, b;
  unsigned x, y;

  // Rechts-Schieben um n Stellen
  // entspricht Division durch 2 hoch n
  a = -12;
  b = a >> 2;   // -12/4
  cout << hex << a << "   " << b << endl;
  cout << dec << a << "   " << b << endl;

  x = -12;     // vorzeichenlos behandelt,
  y = x >> 2;  // daher positiv und recht gross
  cout << hex << x << "   " << y << endl;
  cout << dec << x << "   " << y << endl;

  // Links-Schieben um n Stellen
  // entspricht Multiplikation mit 2 hoch n
  a = 12;
  b = a << 4;   // 12*16
  cout << hex << a << "   " << b << endl;
  cout << dec << a << "   " << b << endl;
}
```

Operatoren und Ausdrücke

Schiebeoperatoren

Ausgabe:

```
fffffff4   fffffffd
-12   -3

fffffff4   3ffffffd
4294967284   1073741821

c   c0
12   192
```

Operatoren und Ausdrücke

Bitoperatoren

logische und bitweise Operatoren

Für einzelne Bits sind die logischen Operatoren wie folgt definiert:

A	B	NOT A	A AND B	A OR B	A XOR B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Interpretiert man `1` als `true` und `0` als `false`, so hat man die logischen Verknüpfungen für die Werte vom Typ `bool`.

Pri.	Operator	Operanden	Ergebnis	Bemerkung
2	**`!`**	beliebig	**`bool`**	logisches NOT
12	**`&&`**	beliebig, beliebig	**`bool`**	logisches AND
13	**`||`**	beliebig, beliebig	**`bool`**	logisches OR

Beispiel:

```
bool vertauscht;
...
if (!vertauscht)
    ...
```

Operatoren und Ausdrücke

logische und bitweise Operatoren

Bei **&&** wird der zweite Operand nicht mehr ausgewertet, wenn der erste `false` ergibt - dann ist das Ergebnis nämlich unabhängig vom zweiten Operanden sicher `false`.

Bei **||** wird der zweite Operand nicht mehr ausgewertet, wenn der erste `true` ergibt - dann ist das Ergebnis nämlich unabhängig vom zweiten Operanden sicher `true`.

Beispiel:

```
if (a < b && c != d) ...
```

bedeutet

```
if(a < b)
    if(c != d) ...
```

Operatoren und Ausdrücke

logische und bitweise Operatoren

Erweitert man die logischen Operatoren auf Folgen von Bits, so erhält man die bitweisen Operatoren.

Pri.	Operator	Operanden	Ergebnis	Bemerkung
2	**~**	ganz	**>= int**	Bitkomplement
9	**&**	ganz, ganz	**>= int**	bitweises AND
10	**^**	ganz, ganz	**>= int**	bitweises XOR
11	**|**	ganz, ganz	**>= int**	bitweises OR

Beispiel:

```
short k = 5, c = k << 2;

0000 0000 0000 0101      k  ist gleich  5
0000 0000 0001 0100      c  ist gleich  20
0000 0000 0000 0100      c  &  k  ist gleich  4

0000 0000 0000 0101      k  ist gleich  5
1111 1111 1111 1010      ~k  ist gleich  -6
```

Operatoren und Ausdrücke

Vergleichsoperatoren

Pri.	Operator	Operanden	Ergebnis	Bemerkung
7	**<**	ganz, ganz reell, reell	**bool**	kleiner als
7	**<=**	ganz, ganz reell, reell	**bool**	kleiner oder gleich
7	**>**	ganz, ganz reell, reell	**bool**	größer als
7	**>=**	ganz, ganz reell, reell	**bool**	größer oder gleich
8	**==**	ganz, ganz reell, reell	**bool**	gleich
8	**!=**	ganz, ganz reell, reell	**bool**	ungleich

Achtung:

Der Test auf Gleichheit wird mit zwei Gleichheitszeichen geschrieben (== statt =) !

Vermeiden Sie den Test auf Gleichheit von Gleitkomma-Werten !

```
double x, y; const double epsilon = 1e-5;

if(abs(x-y) < epsilon) ...  // o.k.

if(x == y) ...  // nein, nein, nein !!!
```

Operatoren und Ausdrücke

Vergleichsoperatoren

Vorsicht:

> Bei gemischter Verwendung der Datentypen `int` und `unsigned int` bei einem Vergleich führt der Compiler eine implizite Typumwandlung `int => unsigned int` durch.

Beispiel:

```
int i = -1;
unsigned int u = 0;

if (u < i)
  cout << u << " < " << i << endl;
```

Ausgabe:

```
0 < -1
```

Operatoren und Ausdrücke

Der Bedingungsoperator ? :

dient zur Formulierung eines Ausdrucks, der abhängig von einem booleschen Ausdruck einen von zwei Werten zurückliefert:

Bedingung ? Ausdruck$_1$: Ausdruck$_2$

Falls die *Bedingung* wahr ist, ergibt sich der Wert von *Ausdruck$_1$*, andernfalls der Wert von *Ausdruck$_2$*.

Beispiel:

```
int a = -12;
int betrag = a < 0 ? -a : a;
cout << betrag << endl;
cout << (a < 0 ? -a : a) << endl;
```

Die Wertzuweisung

```
min = (x <= y) ? x : y;
```

ist äquivalent zur if-Anweisung

```
if(x <= y)
  min = x;
else
  min = y;
```

Operatoren und Ausdrücke

Der Bedingungsoperator `?:`

In einem bedingten Ausdruck

***Bedingung* ? *Ausdruck*$_1$: *Ausdruck*$_2$**

müssen beide Ausdrücke zuweisungskompatible Typen haben. Der Typ des einen Ausdrucks muss dem Typ des anderen ohne explizite Typumwandlung zugewiesen werden können. Der Typ des Ergebnisses ist der allgemeinere der beiden Typen.

Im folgenden Beispiel:

```
double scale = halveIt ? 0.5 : 1;
```

sind die beiden Ausdrücke vom Typ `int` (`1`) bzw. `double` (`0.5`). Daher wird die `1` in `1.0` umgewandelt, und das Ergebnis des Bedingungsoperators ist vom Typ `double`.

Operatoren und Ausdrücke

Implizite Typkonvertierung (Teil 2)

In Java sind *implizite* Typkonvertierungen nur dann erlaubt, wenn keine Information verloren gehen kann.

Implizite Typanpassungen von einem „längeren“ in einen „kürzeren“ Datentyp weist der Java-Compiler zurück.

Beispiel:

Die Deklaration

```
double d = 7.99;
long l = d;
```

führt zu einer Fehlermeldung des Java-Compilers:

Incompatible type for declaration.
Explicit cast needed to convert double to long.

In Java schafft die *explizite* Typkonvertierung Abhilfe:

```
double d = 7.99;
long l = (long) d;
```

Operatoren und Ausdrücke

Implizite Typkonvertierung

C++ erlaubt auch implizite Typkonvertierungen, bei denen Informationen verloren gehen können.

Beispiel:

```
double d = 7.99;
long l = d;

void f(double d)
{
  char c = d;
}
```

Versuchen Sie, implizite Typumwandlungen, bei denen Informationen verloren gehen können, zu vermeiden.

Operatoren und Ausdrücke

Implizite Typkonvertierung

Ganze Zahlen, Gleitkommawerte und Zeiger können implizit in `bool` umgewandelt werden.

Ein Wert ungleich Null wird in `true`, der Wert Null in `false` umgewandelt.

Beispiel:

```
int i = 0;
double d = 1.2;
bool b1, b2;

b1 = i;
b2 = d;

cout.setf(ios_base::boolalpha);
cout << b1 << endl;
cout << b2 << endl;
```

Ausgabe:

```
false
true
```

Operatoren und Ausdrücke

Implizite Typkonvertierung

größer ganzzahlig => kleiner ganzzahlig

- Es werden lediglich so viele Bits von rechts her übernommen, wie der kleinere Typ aufnehmen kann. Dabei kann sich neben dem Zahlenwert auch das Vorzeichen ändern.
- Falls der Zieltyp `unsigned` ist, hat der resultierende Wert genauso viele Bits aus der Quelle, die auch in das Ziel passen. Höherwertige Bits werden ignoriert, falls notwendig.
- Falls das Ziel `signed` ist, bleibt der Wert erhalten, falls er im Zieltyp dargestellt werden kann. Ansonsten ist der Wert implementierungsabhängig.

Beispiel:

```
unsigned char uc = 1023;
// 1023 == 0x3ff    =>  uc == 0xff == 255

signed char sc = 1023;
// Das Ergebnis ist implementierungsabhängig.
// Plausible Ergebnisse sind 255 und –1.
```

Operatoren und Ausdrücke

Implizite Typkonvertierung

`double` => `float`

- Falls der Quellwert exakt mit dem Zieltyp dargestellt werden kann, ist das Ergebnis der ursprüngliche Wert.
- Falls der Quellwert zwischen zwei benachbarten Zielwerten liegt, wird einer der beiden als Zielwert verwendet.
- Ansonsten ist das Verhalten undefiniert.

Beispiel:

```
#include <values>
float f = FLT_MAX;    // größter float-Wert
double d = f;         // ok: d == f
float f2 = d;         // ok: f2 == f

double d3 = DBL_MAX;  // größter double-Wert
float f3 = d3;        // undefiniert, falls
                      // MAXFLOAT < MAXDOUBLE
```

Operatoren und Ausdrücke

Implizite Typkonvertierung

reell => ganzzahlig

- Wenn eine Gleitkommazahl in eine ganze Zahl gewandelt wird, wird der Teil hinter dem Komma abgeschnitten.
- Falls der so erhaltene Wert nicht im Zieltyp dargestellt werden kann, ist das Verhalten undefiniert.

Beispiel:

```
int i = 2.7;        // i wird 2
i = -2.7;           // i wird –2;
char b = 2000.7;    // für 8-Bit-chars undef.
```

Operatoren und Ausdrücke

Explizite Typkonvertierung

Falls Typkonvertierungen notwendig sind, bei denen Informationen verloren gehen könnten, sollten Sie die *explizite* Typkonvertierung verwenden.

Formen der expliziten Typumwandlung sind:

- **Typname (**Ausdruck**)**
- **(Typausdruck)** Ausdruck
- **static_cast<Typausdruck>(**Ausdruck**)**

Beispiel:

```
char c;
int i;

i = c;          // implizite Typanpassung
i = int(c);
i = (int)c;
i = static_cast<int>(c);

i = 66;
c = static_cast<char>(i);
cout << c;                  // 'B'
c = '1';
i = static_cast<int>(c);
cout << i;                  // 49
```

Operatoren und Ausdrücke

Explizite Typkonvertierung

Beispiel:

```
#include <iostream>
using namespace std;

int main()
{
  double d = 1.3e8;

  cout << "double : " << d << endl;
  cout << "float  : " << (float)d << endl;
  cout << "hexlong: " << hex << (long)d << endl;
  cout << "long   : " << dec << (long)d << endl;
  cout << "int    : " << (int)d << endl;
  cout << "short  : " << (short)d << endl;
  cout << "ushort : " << (unsigned short)d << endl;
  cout << "schar  : " << (signed char)d << endl;
  cout << "uchar  : " << (unsigned char)d << endl;
}
```

Ausgabe:

```
double : 1.3e+08
float  : 1.3e+08
hexlong: 7bfa480
long   : 130000000
int    : 130000000
short  : -23424
ushort : 42112
schar  : Ç
uchar  : Ç
```

Operatoren und Ausdrücke

Explizite Typkonvertierung

Beispiel:

```
#include <iostream>
using namespace std;

int main()
{
  float quotient;
  int zaehler = 14, nenner = 4;

  quotient = zaehler / nenner;
  cout << "ganzzahliges Ergebnis : "
       << quotient << endl;

  quotient = (float)zaehler / nenner;
  cout << "reelles Ergebnis      : "
       << quotient << endl;

  quotient = (float)(zaehler / nenner);
  cout << "ganzzahliges Ergebnis : "
       << quotient << endl;
}
```

Ausgabe:

```
ganzzahliges Ergebnis : 3
reelles Ergebnis      : 3.5
ganzzahliges Ergebnis : 3
```

Operatoren und Ausdrücke

Zuweisungsoperatoren

der einfache Zuweisungsoperator =

Pri.	Operator	Operanden	Ergebnis	Bemerkung
14	**=**	linke Seite, beliebig	Typ von linke Seite	Wertzuweisung

Wirkung von $E_1 = E_2$:

- Der rechte Operand E_2, der i. a. ein Ausdruck ist, wird ausgewertet.

- Der Datentyp von E_2 wird, falls nötig, implizit in den Datentyp von E_1 konvertiert.
  Der so erhaltene Wert wird E_1 zugewiesen.

- Die Zuweisung liefert als Wert den neuen Wert von E_1.

Operatoren und Ausdrücke

Zuweisungsoperatoren

Beispiel:

```
#include <iostream>
using namespace std;

int main()
{
  int x, y = 5, z = 7, u = -15;
  cout << "x = " << (x = y + z) << endl;
  // Wert von x nach Zuweisung ausgeben
  cout << "nochmal x = " << x << endl;
  // und noch einmal ausgeben
  y = (u = x + y) + z;
  // u wird die Summe x + y zugewiesen
  // danach wird y der Wert u + z zugewiesen
  cout << "u = " << u << ", y = " << y << endl;
  x = y = z = 15;
  // Mehrfachzuweisung: x, y, z erhalten den Wert 15
  cout << "x = " << x << endl;
}
```

Ausgabe:

```
x = 12
nochmal x = 12
u = 17, y = 24
x = 15
```

Operatoren und Ausdrücke

Zuweisungsoperatoren

kombinierte Zuweisungsoperatoren

Jeder Operator

op ∈ { ***, /, %, +, -, &, ^, |, <<, >>*** }

kann mit = verknüpft werden. Auf diese Weise erhält man die sogenannten kombinierten Zuweisungs-operatoren.

Pri.	Operator	Operanden	Ausdruck	entspricht
14	**op=**	Opd, Wert	Opd **op=** Wert	Opd **=** Opd **op** Wert

Beispiel:

```
feld[wo()] += 12;
```

ist dasselbe, wie

```
feld[wo()] = feld[wo()] + 12;
```

außer, dass der Ausdruck auf der linken Seite der Zuweisung auch nur einmal ausgewertet wird.

Operatoren und Ausdrücke

Zuweisungsoperatoren

kombinierte Zuweisungsoperatoren

Sind `var` eine Variable vom Typ `typ`, `expr` ein Ausdruck und *op* ein binärer Operator, so ist

```
var op= expr
```

äquivalent zu

```
var = (typ)((var) op (expr))
```

außer, dass `var` nur einmal ausgewertet wird.

Das bedeutet, dass *op*= nur gültig ist, wenn *op* auch für die beteiligten Typen gültig ist.

Beachten Sie die obige Klammerung. Der Ausdruck

```
a *= b + 1
```

ist gleichwertig mit

```
a = a * (b + 1)
```

aber nicht mit

```
a = a * b + 1
```

Operatoren und Ausdrücke

Zuweisungsoperatoren

Inkrement- und Dekrement-Operatoren

Häufig muss man den Wert einer Variablen um 1 erhöhen (inkrementieren) oder erniedrigen (dekrementieren). Hierzu gibt es den Inkrement-Operator ++ bzw. den Dekrement-Operator `--`.

Beide Operatoren sind unär und können entweder *vor* dem Operanden (Präfixform) oder *hinter* dem Operanden (Postfixform) stehen.

Pri.	Op.	Operanden	Ergebnis	Bemerkung
1	**++**	linke Seite	Typ von linker Seite	Postinkrement
1	**--**	linke Seite	Typ von linker Seite	Postdekrement
2	**++**	linke Seite	Typ von linker Seite	Präinkrement
2	**--**	linke Seite	Typ von linker Seite	Prädekrement

- Der Wert des Operanden wird bei ++ um 1 erhöht, bei `--` um 1 erniedrigt.

- Der Wert des Ausdrucks ist wie folgt festgelegt:

  - in der *Postfixform* der ursprüngliche Wert des Operanden,

  - in der *Präfixform* der um 1 erhöhte bzw. erniedrigte Wert des Operanden.

Operatoren und Ausdrücke

Zuweisungsoperatoren

Inkrement- und Dekrement-Operatoren

Beispiel:

```
#include <iostream>
using namespace std;

int main()
{
  int a = 12;
  cout << a++ << " , ";
  cout << ++a << " , ";
  cout << --a << " , ";
  cout << a-- << " , ";
  cout << --a << endl;
}
```

Ausgabe:

```
12 , 14 , 13 , 13 , 11
```

12	alter Wert von a, danach wird a erhöht
14	Wert 13 wird erst erhöht, dann ausgegeben
13	Wert 14 wird erst erniedrigt, dann ausgegeben
13	alter Wert von a, danach wird a erniedrigt
11	Wert 12 wird erst erniedrigt, dann ausgegeben

Operatoren und Ausdrücke

Der Kommaoperator

Der Kommaoperator wird gelegentlich in einer for-Schleife benutzt, um die Schleife kompakter zu schreiben.

Beispiel:

```
int i, sum;
for(i = 1, sum = 0; i <= 100; sum += i, i++);
cout << sum << endl;
```

Ausgabe:

```
5050
```

Der Kommaoperator kann in C++ auch anderweitig verwendet werden. Der Wert eines Ausdrucks

$expr_1, expr_2, expr_3, \ldots , expr_n$

ist der Wert des letzten Operanden.

Beispiel:

```
cout << (i++, --j, k += 1, l = k + j, 5)
     << endl;
```

Ausgabe:

```
5
```

Operatoren und Ausdrücke

Konstantenausdrücke

C++ erwartet an manchen Stellen, z.B. bei Feldgrenzen, `case`-Marken und Initialisierern für Aufzählungstypen, sogenannte Konstantenausdrücke. Das sind Ausdrücke, deren Operanden konstante Werte oder benannte Konstanten sind. Konstantenausdrücke werden bereits bei der Übersetzung berechnet.

Beispiel:

```
#include <iostream>
using namespace std;

int main()
{
  const int a = 12, b = -14, c = 3;
  int i = 3 * (a * c + b);
  cout << i << endl;
}
```

Der Compiler erzeugt folgende Deklaration:

```
int i = 66;
```