PROGRAMMER'S CHOICE

André Willms

C++

Programmierung lernen

Anfangen, Anwenden, Verstehen

Klassiker

ADDISON-WESLEY

3 Die Grundrechenarten

In diesem Kapitel wollen wir uns unter anderem mit folgenden Themen beschäftigen:

- Operatoren für die Addition, Subtraktion, Multiplikation und Division
- Modulooperator, auch Restoperator genannt
- Besondere Zuweisungsoperatoren
- Berechnung von Ausdrücken bei der Ausgabe
- Inkrement- und Dekrementoperatoren
- Prä- und Postoperatoren

3.1 Die Operatoren +, –, *, /, (,)

Im vorigen Kapitel haben Sie gelernt, wie Sie Variablen definieren, initialisieren und ihnen Werte zuweisen. Wir werden uns nun die Operatoren ansehen, mit denen wir die hinlänglich bekannten Grundrechenarten auf die Inhalte unserer Variablen anwenden können.

In der folgenden Tabelle sehen Sie die C++-Operatoren für die vier Grundrechenarten:

Operator	Bedeutung
+	addiert zwei Werte
–	subtrahiert den rechten Wert vom linken
*	multipliziert zwei Werte miteinander
/	dividiert den linken Wert durch den rechten

Tabelle 3.1: Die C++-Operatoren für die vier Grundrechenarten

Sie können nun mit diesen Operatoren verschiedene Variablen verknüpfen und das Ergebnis einer anderen Variablen mithilfe des Zuweisungsoperators = zuweisen:

Syntax

```
x=a+b;
```

Die obige Anweisung bildet die Summe der Inhalte von *a* und *b* und weist das Ergebnis der Variablen *x* zu. Folgende Anweisung zeigt auch, dass das Gleichheitszeichen hier nicht wie in der Mathematik üblich betrachtet werden darf:

```
c=c*d;
```

Mathematisch betrachtet ist obige Anweisung schlichtweg falsch. In C++ bedeutet sie lediglich, dass die Inhalte von *c* und *d* multipliziert werden und das Ergebnis dann *c* zugewiesen wird. Dabei gilt grundsätzlich folgende Regel:

Zuerst wird der Ausdruck rechts vom Zuweisungsoperator komplett berechnet und dann wird das Ergebnis der Variablen links vom Zuweisungsoperator zugewiesen.

Bindungsstärke

Sie können die Rechenoperatoren in allen möglichen Kombinationen verwenden. Die Operatoren * und / binden stärker als + und –. Das heißt, 3+4*5 ergibt 23 und nicht 35[1].

Möchten Sie jedoch, dass zuerst 3+4 gerechnet und dann die Multiplikation mit 5 vorgenommen wird, müssen Sie den Ausdruck klammern. (3+4)*5 würde dann tatsächlich 35 ergeben, weil die Klammern stärker binden als die übrigen Rechenzeichen.

Runden

Da *int*-Variablen nur ganzzahlige Werte aufnehmen können, lässt C++ eventuell entstehende Ziffern hinter dem Komma einfach wegfallen. 3/2 ergibt dann 1 und –3/2 ergibt –1.

1 Diese Regel entspricht der »Punkt-vor-Strich«-Regel in der Mathematik.

! ! ! ACHTUNG

Fließkommazahlen werden an integer-Variablen durch Wegfall der Nachkommastellen angepasst. Es findet KEIN Auf- oder Abrunden statt.

Nachfolgend ein Beispiel, das die Rechenoperatoren benutzt:

× × × BEISPIEL

```
#include <iostream>

using namespace std;

int main()
{
int zahl1,zahl2,zahl3,zahl4;

  zahl1=3*2;                  //1
  zahl2=zahl3=zahl1*2;        //2
  zahl3=zahl3*2;              //3
  zahl4=zahl3/4;              //4
  zahl1=zahl2*zahl4+zahl3;    //5
  cout << "Das Ergebnis lautet : " << zahl1 << endl;
}
```

⊙ ⊙ ⊙ CD

Den Quellcode dieses Beispiels finden Sie auf der CD unter \KAP03\BSP01.CPP.

Sie können ja mal ausprobieren, ob Sie herausbekommen, welchen Wert das Programm ausgibt. Die Erklärung richtet sich nach der Nummerierung, die hinter den Zeilen als Kommentar angegeben ist.

1. *zahl1* bekommt den Wert 6 zugewiesen (Initialisierung von *zahl1*).
2. *zahl2* bekommt den Wert von *zahl3*, und *zahl3* bekommt den Wert von *zahl1**2, also 12 zugewiesen. *zahl2* und *zahl3* haben daher den Wert 12 (Initialisierung von *zahl2* und *zahl3*).
3. zahl3 bekommt den Wert zahl3*2, also 24.
4. zahl4 bekommt den Wert zahl3 durch 4, also 6 (Initialisierung von zahl4).
5. zahl1 bekommt den Wert zahl2*zahl4+zahl3, das ist 12*6+36, also 96.

   Das Programm muss daher den Wert 96 ausgeben. Sie können das Programm selbst kompilieren und starten, dann erhalten Sie den Wert ebenfalls.

3.2 Der %-Operator

Der %-Operator, der auch Modulooperator genannt wird, findet seine Verwendung bei Ganzzahlen und ermittelt dort den Rest einer Division.

Der Modulooperator % bestimmt den Rest einer Division von Ganzzahlen.

13%4 ergibt 1, weil bei Ganzzahlen 13/4 3 ergibt und somit 13–12=1 der Wert des Restes ist.

Mit diesem Operator lassen sich bestimmte Probleme ziemlich einfach lösen. Ein simples Beispiel: Stellen Sie sich vor, Sie haben acht Schubladen, die von 0 bis 7 durchnummeriert sind. In diese Schubladen sollen Sie Zahlen einsortieren. Die Zahl 0 kommt in Schublade 0, die Zahl 7 kommt in Schublade 7 und die Zahl 8 kommt wieder in Schublade 0. Demnach kommt die 12 in Schublade 4 und die 33 in Schublade 1. Ich denke, Sie erkennen die Regelmäßigkeit.

Wie lösen Sie dieses Problem rechnerisch? In C++ kein Problem. Sie nehmen eine Variable namens *schublade*, die die Schubladennummer enthalten soll, und eine Variable *zahl*, die die einzusortierende Zahl enthalten soll. Sie berechnen in einem C++-Programm die Schublade wie folgt:

Syntax

```
schublade = zahl % 8;
```

Die »8« wird verwendet, weil es sich ja um acht Schubladen handelt. Die acht Schubladen werden nummeriert von 0 bis 7. Die Leerzeichen zwischen den Operatoren und den Variablennamen dienen im Übrigen nur der Übersichtlichkeit und können auch weggelassen werden:

```
schublade=zahl%8;
```

Auch hier gilt wieder die altbekannte Regel, dass Programme nicht nur für Computer geschrieben werden, deshalb ist die Schreibweise mit Leerzeichen derjenigen ohne vorzuziehen.

3.3 Die Operatoren +=, –=, *=, /=, %=

Sie haben bisher die Operatoren für die Grundrechenarten kennengelernt. Ihnen ist vielleicht aufgefallen, dass solche Berechnungen fast immer mit einer Zuweisung einhergehen.

Deswegen wurden aus dem Zuweisungsoperator und den Rechenoperatoren neue Operatoren entworfen, die beide Funktionen übernehmen.

Diese Operatoren, auch Zuweisungsoperatoren genannt, dienen der Verkürzung von Zuweisungen. Sie können zum Beispiel folgende Verkürzungen vornehmen:

Ausdruck	Äquivalent
zahl=zahl+5;	zahl+=5;
wert=wert*23;	wert*=23;
x=x-wert;	x-=wert;
value=value/2;	value/=2;
x=x%8;	x%=8;
value=value+3*wert;	value+=3*wert;
zahl=wert/value+zahl;	zahl+=wert/value;

Tabelle 3.2: Äquivalenz von Ausdrücken

! ! ! ACHTUNG

Es ist jedoch wichtig, sich das über Zuweisungen Gesagte in Erinnerung zu rufen: Das Gleichheitszeichen ordnet der links von ihm stehenden Variablen den Wert zu, den der Ausdruck auf der rechten Seite ergibt. Falls der Ausdruck auf der rechten Seite eine Berechnung erfordert (z. B. a=b+c), wird er zuerst berechnet, und das Ergebnis dieser Berechnung wird dann der Variablen auf der linken Seite übergeben.

Bei den vier Zuweisungsoperatoren bedeutet dies ein Berechnen des Ausdrucks auf der rechten Seite mit anschließendem Verknüpfen des Ergebnisses mit der linken Seite. Die Art der Verknüpfung hängt dabei vom Zuweisungsoperator ab.

Das letzte Beispiel in der Tabelle ist nur deswegen äquivalent, weil die Division eine stärkere Bindung hat als die Addition.

```
wert=wert*3+value;
```

ist nicht gleich mit

```
wert*=3+value;
```

Bevor Sie weiterlesen, überlegen Sie einmal, warum die beiden Ausdrücke nicht gleich sind.

Betrachten wir zunächst die erste Zuweisung. Wie wir wissen, wird zuerst der Ausdruck auf der rechten Seite berechnet, der danach der Variablen auf der linken Seite zugewiesen wird. Da der Ausdruck nicht geklammert ist, wird zuerst die Multiplikation ausgeführt, weil sie Vorrang vor der Addition hat. Danach wird die Addition vorgenommen.

Die zweite Zuweisung analysieren wir genauso. Zuerst wird der Ausdruck auf der rechten Seite berechnet. Dort steht nur eine Addition und kann somit ausgeführt werden. Danach wird der rechte Ausdruck dem Zuweisungsoperator entsprechend mit dem linken verknüpft und in der linken Variablen gespeichert. Die Verknüpfung ist eine Multiplikation, also wird der berechnete Wert mit der Variablen *wert* multipliziert und dann in derselben gespeichert.

Wie wir sehen, wurde bei der ersten Zuweisung zuerst die Multiplikation und dann die Addition ausgeführt, bei der zweiten Zuweisung hingegen zuerst die Addition und dann die Multiplikation. Die beiden Zuweisungen können also nicht zu gleichen Ergebnissen führen.

! ! ! ACHTUNG

Achten Sie bei der Benutzung der Zuweisungsoperatoren auf die Bindungsstärke der verwendeten Rechenoperatoren.

Wenn Sie bei der Berechnung eines Ausdrucks nicht sicher sind, welche Operatoren denn nun Vorrang haben, dann klammern Sie einfach so, dass das Ergebnis Ihrem Wunsch entspricht. Klammern haben nämlich die höchste Priorität, und es ist grundsätzlich besser, zu viele Klammernpaare zu setzen als zu wenige.

3.4 Ausgabe von Ausdrücken

Des Weiteren ist es interessant, Folgendes zu wissen:

An jeder Stelle im Programm (außer links des Zuweisungsoperators), an der eine Variable stehen kann, kann auch ein berechenbarer Ausdruck stehen.

Sehen Sie einmal folgendes Fragment an:

× × × BEISPIEL

```
zahl3=zahl1+zahl2;
cout << zahl1 << " plus " << zahl2;
cout << " ist gleich " << zahl3 << endl;
```

Vorausgesetzt, dass die Variable *zahl3* nur für die Ausgabe der Summe gebraucht wird und in keinen weiteren Berechnungen auftritt, lassen sich die beiden Zeilen auf folgende Weise vereinfachen:

```
cout << zahl1 << " plus " << zahl2;
cout << " ist gleich " << (zahl1+zahl2) << endl;
```

Die Übersichtlichkeit erhöht sich und das Programm hat eine Variable weniger. C++ geht sogar noch weiter und erlaubt es, folgenden Satz aufzustellen:

Steht an einer Stelle im Programm ein berechenbarer Ausdruck, kann er auch an Ort und Stelle zugewiesen werden.

Dadurch entsteht eine Mischung aus den beiden oberen Beispielen:

```
cout << zahl1 << " plus " << zahl2;
cout << " ist gleich " << (zahl3=zahl1+zahl2) << endl;
```

Diese Zeile gibt nun die Summe von *zahl1* und *zahl2* aus und weist das Ergebnis zusätzlich noch *zahl3* zu, sodass im weiteren Verlauf *zahl3* tatsächlich die Summe der beiden Variablen enthält.

Des Weiteren gilt auch der folgende Satz, der eigentlich nicht erwähnt werden müsste:

An jeder Stelle im Programm, an der eine Variable stehen kann, kann auch eine Konstante stehen (außer links einer Zuweisung).

Dazu einmal ein sinnloses Beispiel:

```
cout << "Die Zahl heißt " << 231 << endl;
```

Sinnlos deshalb, weil in diesem Fall die Konstante direkt mit in den auszugebenden String geschrieben werden könnte:

```
cout << "Die Zahl heißt 231" << endl;
```

Sinnvoll wird das Einsetzen einer Konstanten erst bei Funktionsaufrufen mit Parameterübergabe, die wir später kennenlernen werden.

3.5 Die Operatoren ++ und --

C++ wäre nicht C++, wenn es für spezielle Fälle nicht wieder Abkürzungen parat hätte. Und tatsächlich gibt es noch einen Sonderfall der Addition und der Subtraktion: das *Inkrementieren* und das *Dekrementieren*. Eine Variable wird inkrementiert, wenn ihr Wert um 1 erhöht wird, und sie wird dekrementiert, wenn ihr Wert um 1 vermindert wird[2].

Sie fragen sich natürlich, warum diese beiden Operationen als Sonderfall aufgefasst werden, denn wir könnten doch ganz einfach eine Variable mit *+=1* inkrementieren und mit *–=1* dekrementieren. Das stimmt, aber je nachdem,

2 Eine Ausnahme bilden die Inkrement- und Dekrementoperatoren bei den Zeigern. Dort erhöhen bzw. vermindern sie nicht notwendigerweise um 1. Doch später dazu mehr.

welche der beiden Schreibweisen benutzt wird, können Sie auf den Verwendungszweck des Ausdrucks schließen. Des Weiteren können die Inkrement-/Dekrementoperatoren auf zwei verschiedene Weisen verwendet werden, auf die wir gleich zu sprechen kommen.

Zähler

Variablen, hauptsächlich ganzzahlige Variablen, können auf eine bestimmte Weise benutzt werden: nämlich als *Zähler*. Zähler werden im Allgemeinen nur um 1 erhöht oder um 1 vermindert. Deswegen sollten Sie zum Erhöhen und Vermindern, oder in der C++-Sprache: zum Inkrementieren und Dekrementieren, die entsprechenden Operatoren benutzen, wohingegen Sie beim bloßen Addieren oder Subtrahieren die übrigen Operatoren benutzen sollten.

* * * TIPP

Benutzen Sie zum Inkrementieren/Dekrementieren die Operatoren ++/-- und zum Addieren/Subtrahieren die Operatoren +/– bzw. +=/–=.

Doch nun ein Beispiel:

× × × BEISPIEL

```
wert=3;
cout << "Der Wert ist " << wert << endl;
wert++;
cout << "Der Wert ist " << wert << endl;
wert--;
cout << "Der Wert ist " << wert << endl;
```

Es wird Sie nicht überraschen, wenn die drei *cout*-Anweisungen 3, 4 und 3 ausgeben.

prä, post

Nun kann sowohl der Inkrement- als auch der Dekrementoperator in zwei Formen benutzt werden: erstens als **Postinkrement** und **Postdekrement** und zweitens als **Präinkrement** und **Prädekrement**.

Bei den Postoperatoren wird die Variable erst »benutzt« und dann entsprechend inkrementiert oder dekrementiert. Die Präoperatoren inkrementieren bzw. dekrementieren vor der Benutzung.

Das obige Beispiel verwendete Postinkrement und Postdekrement. Um die Unterschiede in der Schreibweise zu erklären, nun das gleiche Beispiel mit Präoperatoren:

× × × BEISPIEL

```
wert=3;
cout << "Der Wert ist " << wert << endl;
++wert;
cout << "Der Wert ist " << wert << endl;
--wert;
cout << "Der Wert ist " << wert << endl;
```

Nun ändert sich am Ablauf des Programms und an seiner Ausgabe durch die Änderung rein gar nichts. Wieso? Weil an der Stelle, an der inkrementiert bzw. dekrementiert wurde, keine Benutzung der Variablen stattfand. Doch wie wir schon gelernt haben, kann an der Stelle, an der ein berechenbarer Ausdruck steht, auch eine Zuweisung stehen. Dies gilt auch für die Inkrement- und Dekrementoperatoren, sodass wir sie nun in die *cout*-Anweisung einbetten.

× × × BEISPIEL

```
wert=3;
cout << "Der Wert ist " << wert++ << endl;
cout << "Der Wert ist " << wert-- << endl;
cout << "Der Wert ist " << wert << endl;
```

Schauen wir uns die erste *cout*-Anweisung an. Wir benutzen dort ein Postinkrement, sodass die Variable zuerst benutzt – was in diesem Fall eine Übergabe an die *cout*-Anweisung bedeutet – und dann imkrementiert wird. *cout* gibt also den Wert 3 aus, aber danach hat *wert* den Wert 4.

Das Gleiche gilt für die zweite *cout*-Anweisung. Zuerst wird der aktuelle Wert übergeben, nämlich 4, und dann wird die Variable dekrementiert. Die Werte, die das Programm ausgibt, lauten also 3, 4 und 3. Hier ist das Ergebnis noch identisch mit den vorherigen Versionen, doch benutzen wir nun die Präoperatoren:

× × × BEISPIEL

```
wert=3;
cout << "Der Wert ist " << ++wert << endl;
cout << "Der Wert ist " << --wert << endl;
cout << "Der Wert ist " << wert << endl;
```

In der ersten *cout*-Anweisung wird die Variable nun vor dem Benutzen inkrementiert, sodass die Variable *wert* den Wert 4 hat, wenn sie *cout* übergeben wird.

In der zweiten *cout*-Anweisung wird zuerst dekrementiert, also wird *printf* der Wert 3 übergeben. Die Werte der Ausgabe lauten 4, 3 und 3. Dies unterscheidet sich von den anderen Ergebnissen.

Bitte versuchen Sie, die Inkrement- bzw. Dekrementoperatoren nur für Zähler zu benutzen. Es ist natürlich auch erlaubt, sie in einem anderen Kontext anzuwenden, aber Sie sollten für sich selbst bestimmte Regeln aufstellen, an die Sie sich dann halten. Und je mehr sich diese Regeln mit den Regeln der Allgemeinheit decken, umso besser ist es.

Wenn Sie die Inkrement-/Dekrementoperatoren in Ausdrücken benutzen, ist es ratsam, ihre Wirkungsweise wie oben durchzuspielen, da unbedachter Einsatz derselben immer eine beliebte Fehlerquelle bildet.

3.6 Kontrollfragen

1. Welchen Wert ergibt *17/3*?
2. Welchen Wert ergibt 17%3?
3. Welchen Unterschied gibt es zwischen den Ausdrücken *3*4+5* und *5+4*3*?
4. Welchen Unterschied gibt es zwischen den Ausdrücken *(3*4)+5* und *3*(4+5)*?
5. Welche Nachteile ergeben sich, wenn ein zu berechnender Ausdruck direkt in der *cout*-Anweisung steht?
6. Sind *x+=2* und *x++2* das Gleiche?
7. Worin liegt der Vorteil von ++ gegenüber *+=1*?

3.7 Übungen

1. Schreiben Sie ein Programm, das Sie nach drei Zahlen fragt (auch negative Werte sollen erlaubt sein) und dann die Summe der drei Zahlen ausgibt. Nachdem die Summe ausgegeben wurde, soll nach einer neuen Zahl gefragt werden, mit der die Summe dann multipliziert wird. Dieses Ergebnis soll ebenfalls ausgegeben werden.
2. Schreiben Sie ein Programm, bei dem drei Zahlen multipliziert werden, das aber nur zwei Variablen benötigt. Lassen Sie das Ergebnis ausgeben. Entwerfen Sie das Programm so, dass das Produkt noch weiterverwendet werden könnte.

3. Schauen Sie sich folgendes Programmfragment an:

```
a=12;
a+=++a+a++;
a=a+a;
cout << a << endl;
```

   Welcher Wert wird ausgegeben?