Kapitel 4

Arbeiten mit den eingebauten Typen

Sie kennen nun die grundlegenden Datentypen von C++. Damit können wir Berechnungen durchführen – wir starten mit einfachen Operatoren aus der Mathematik. Dabei betrachten wir auch gleich den Inkrement- und Dekrementoperator. Diese Kurzformen von Operationen sind für die Programmierung besonders nützlich.

Danach werden wir die Frage klären, was passiert, wenn man die Datentypen miteinander vermischt, und welche Regeln in C++ für die Typumwandlung gelten.

4.1 Arithmetische Operatoren

Wir starten mit arithmetischen Operatoren, wie beispielsweise in `3+5`. Der Operator `+` addiert seine beiden Operanden `3` und `5` und gibt das Ergebnis als Wert zurück. Der Operator `+` hat hier zwei (lateinisch: *bi*) Operanden und gehört damit zu den *binären Operatoren*. Das Ergebnis des Ausdrucks kann direkt in einer Initialisierung oder Zuweisung verwendet werden.

```
int a = 3+5;  // entspricht int a = 8;
a = 9-4;      // entspricht a = 5;
```

Die beiden Operatoren + und – arbeiten aber auch als sogenannte *unäre Operatoren* mit nur einem Operanden, wenn Sie sie als Vorzeichen nutzen. Die Art des Operators ergibt sich hier aus dem Kontext. Gegebenenfalls muss ein Ausdruck in Klammern gesetzt werden, um dem Compiler klar anzuzeigen, welche Anwendung gemeint ist:

```
int c = -3;
int d = +4-(-7);  // entspricht int d = 11;
```

Tabelle 4.1 veranschaulicht einige arithmetische Operatoren für mathematische Operationen, die in C++ zur Verfügung stehen.

Operator	Beispiel	Bedeutung
+	x+y	Addiert zwei Werte.
-	x-y	Subtrahiert zwei Werte.
*	x*y	Multipliziert zwei Werte.
/	x/y	Dividiert zwei Werte. Eine Division durch 0 ist nicht erlaubt und sollte vermieden werden.
%	x%y	Rest einer Division (Modulo). Funktioniert nur mit Ganzzahlen, nicht mit Fließkommazahlen als Datentyp. Eine Modulo-Division durch 0 ist nicht erlaubt und sollte vermieden werden.
+	+x	Positives Vorzeichen
-	-x	Negatives Vorzeichen

Tabelle 4.1 Darstellung von binären und unären arithmetischen Operatoren in C++

Für arithmetische Operatoren gelten die klassischen Punkt-vor-Strich-Regeln. Das bedeutet, `100 + 100 * 2` ergibt 300. Klammern binden stärker als die Rechenzeichen: `(100 + 100) * 2` ergibt den Wert 400.

Hier sehen Sie ein einfaches Beispiel, das arithmetische Operationen in der Praxis zeigt:

```
00  // listings/04/listing01.cpp
...
01  int main() {
02    int wert1 = 0, wert2 = 0, wert3 = 0;
03    int sum = 0, mul = 0, div = 0, rest = 0;
04    std::cout << "Wert 1: ";
05    std::cin >> wert1;
06    std::cout << "Wert 2: ";
07    std::cin >> wert2;
08    std::cout << "Wert 3: ";
```

```
09    std::cin >> wert3;
10    sum = wert1 + wert2 + wert3;  // Alles addieren
11    mul = wert1 * wert2 * wert3;  // Alles multiplizieren
12    div = sum / wert3;            // Ganzzahldivision
13    rest = sum % wert3;           // Rest ermitteln
14    std::cout << wert1 << "+" << wert2 << "+" << wert3 << "="
15              << sum << '\n';
16    std::cout << wert1 << "*" << wert2 << "*" << wert3 << "="
17              << mul << '\n';
18    std::cout << sum << "/" << wert3 << "=" << div << ", Rest "
19              << rest << "\n";
20  }
```

Das Programm bei der Ausführung:

```
Wert 1: 2
Wert 2: 4
Wert 3: 5
2+4+5=11
2*4*5=40
11/5=2, Rest 1
```

In Zeile (**03**) bis (**09**) lesen Sie die Werte für die anschließenden Berechnungen über den Eingabestream `std::cin` ein. In Zeile (**10**) werden alle eingegebenen Werte addiert und das Ergebnis des Ausdrucks mit dem Zuweisungsoperator (`=`) der Variablen `sum` zugewiesen. In Zeile (**11**) wird das Ergebnis der Multiplikation der drei Werte der Variablen `mul` zugewiesen. Die Zeile (**12**) ermittelt das Ergebnis der Division von `sum` durch `wert3`. Der erste Operand ist ein Ganzzahltyp, daher kommt die Ganzzahldivision zum Einsatz. Mithilfe des Modulo-Operators wird in Zeile (**13**) der Rest einer Ganzzahldivision berechnet. In den Zeilen (**14**) bis (**19**) erfolgt die Ausgabe der berechneten Werte.

Eingabe von Anwendern prüfen

Machen Sie im Programm zu *listing01.cpp* eine falsche Eingabe, beispielsweise einen Buchstaben statt einer Zahl, ist die weitere Abarbeitung des Programms nicht mehr vorhersehbar. In der Praxis sollten Sie alle Eingaben prüfen. Dazu können Sie den Eingabestream `std::cin` abfragen, ob es einen Fehler gegeben hat. Die Eingabe eines Wertes, der nicht zum geforderten Datentyp passt, ist ein solcher Fehler. Hier soll das Programm nach einer Fehlermeldung direkt beendet werden.

Zur Implementierung einer Fehlerbehandlung greifen wir den ersten Abschnitten des Kapitel 5 vor. Praktisch sieht die Fehlerbehandlung in *listing02.cpp* dann so aus:

```
if (std::cin.fail()) {
  std::cerr << "Fehler bei der Eingabe\n";
  return EXIT_FAILURE;
}
```


In der ersten Zeile wird ermittelt, ob es bei der letzten Eingabe mit `std::cin` einen Fehler gegeben hat. Wenn das der Fall ist, wird eine Fehlermeldung ausgegeben und das Programm wird mit `return EXIT_FAILURE` beendet.

Wie mit den Ganzzahltypen lässt sich auch mit Fließkommatypen rechnen. Hierzu betrachten wir ein Programm, das nach Angabe eines Radius den Umfang des entsprechenden Kreises ermittelt. Der Kreisumfang berechnet sich zu *U = pi *2*r*:

```
00  // listings/04/listing02.cpp
...
01  int main() {
02    constexpr double pi = 3.141592;
03    double r = 0.0;
04    std::cout << "Kreisumfangsberechnung\n";
05    std::cout << "Radius eingeben: ";
06    std::cin >> r;
07    if (std::cin.fail()) {
08      std::cerr << "Fehler bei der Eingabe\n";
09      return EXIT_FAILURE;
10    }
```

```
11    const double U = 2 * r * pi;
12    std::cout << "Der Umfang betraegt " << U << "\n";
13    return EXIT_SUCCESS;
14  }
```

Das Programm bei der Ausführung:

```
Kreisumfangsberechnung
Radius eingeben: 1.2
Der Umfang betraegt 7.53982
```

Als Fließkommatyp haben wir `double` gewählt und den Wert von `pi` gleich bei der Definition als konstanten Datentyp festgelegt, der bereits zur Übersetzungszeit bekannt ist. Nachdem wir den Wert für den Radius in Zeile (**06**) eingelesen haben, wird in Zeile (**11**) die Berechnung durchgeführt und das Ergebnis der Konstanten `U` zugewiesen, die in Zeile **(12)** ausgegeben wird.

Variablen, const und constexpr

Der Wert von `U` wird im Programmverlauf nicht mehr verändert, daher ist `U` mit `const` als Konstante angelegt worden. Das ist korrekt, in der Praxis wird allerdings in solchen Fällen oft auch eine Variable verwendet. Die Verwendung von `constexpr` hingegen hätte an dieser Stelle zu einem Fehler geführt, denn aufgrund der Benutzereingabe von `r` kann der Wert noch nicht zur Übersetzungszeit ermittelt werden.

In Zeile (**07**) bis (**10**) wird die Eingabe überprüft. Wird ein Fehler festgestellt, werden die Anweisungen innerhalb des Anweisungsblocks `{}` von `if` ausgeführt, also eine Fehlermeldung über den Stream `std::cerr` ausgegeben und das Programm direkt mit `return EXIT_FAILURE` beendet.

4.1.1 Kurzschreibweise arithmetischer Operatoren

Bestimmte Konstrukte kommen in der Programmierung besonders oft vor, beispielsweise das schrittweise Erhöhen eines Wertes:

```
wert = wert + schrittweite;
```


Für diesen und ähnliche Ausdrücke gibt es die gleichwertige (und kürzere) Darstellung mit dem `+=`-Operator:

```
wert += schrittweite;
```

Alle binären arithmetischen Operatoren aus dem vorherigen Abschnitt lassen sich in einer solchen Kurzschreibweise verwenden, korrekt sind also auch die Anweisungen:

`var1-=var2`, `var1*=var2`, `var1/=var2` und `var1 %=var2`.

4.1.2 Inkrement- und Dekrementoperator

Besonders oft wird der Wert von Variablen um 1 erhöht oder verringert. Für dieses sogenannte Inkrementieren und Dekrementieren gibt es eigene Operatoren. Wie diese Operatoren in C++ geschrieben werden, veranschaulicht Tabelle 4.2.

Operator	Bedeutung
`++`	Inkrementoperator (die Variable wird um 1 erhöht.)
`--`	Dekrementoperator (die Variable wird um 1 verringert.)

Tabelle 4.2 Inkrement- und Dekrementoperator

Anwendungsgebiete des Inkrement- und Dekrementoperators

Die beiden Operatoren werden meist bei Schleifen verwendet. Sie sind beide unär, brauchen also nur einen Operanden.

Für die Verwendung dieser beiden Operatoren, die sich neben Ganzzahlen auch auf Fließkommazahlen anwenden lassen, gibt es jeweils zwei Möglichkeiten, wie Tabelle 4.3 zeigt.

Anwendung	Bedeutung
`var++`	Postfix-Schreibweise; erhöht den Wert von `var`, gibt noch den alten Wert an den aktuellen Ausdruck weiter.

Tabelle 4.3 Postfix- und Präfix-Schreibweisen

Anwendung	Bedeutung
`++var`	Präfix-Schreibweise; erhöht den Wert von `var` und gibt diesen sofort an den aktuellen Ausdruck weiter.
`var--`	Postfix-Schreibweise; reduziert den Wert von `var`, gibt noch den alten Wert an den aktuellen Ausdruck weiter.
`--var`	Präfix-Schreibweise; reduziert den Wert von `var` und gibt diesen sofort an den aktuellen Ausdruck weiter.

Tabelle 4.3 Postfix- und Präfix-Schreibweisen (Forts.)

Hierzu ein einfaches Beispiel, das den Inkrementoperator (`++`) in beiden Varianten demonstriert. Der Dekrementoperator (`--`) verhält sich analog.

```
00  // listings/04/listing03.cpp
...
01  int main() {
02    int ivar = 1;
03    std::cout << "ivar = " << ivar << '\n';
04    ivar++;
05    std::cout << "ivar = " << ivar << '\n';
06    std::cout << "ivar = " << ivar++ << '\n';
07    std::cout << "ivar = " << ivar << '\n';
08    std::cout << "ivar = " << ++ivar << '\n';
11  }
```

Das Programm bei der Ausführung:

```
ivar = 1
ivar = 2
ivar = 2
ivar = 3
ivar = 4
```

Der Inkrementoperator wird in Zeile (**04**) erstmals verwendet, und zwar in der Postfix-Schreibweise. Hier wird der Wert der Variablen `ivar` um eins erhöht. Die Inkrementierung steht als Ausdruck allein. Ob Sie die Postfix- oder Präfix-Schreibweise verwenden, ist daher an dieser Stelle egal. In Zeile (**05**) wird der zuvor auf 2 inkrementierte Wert von `ivar` ausgegeben.

In Zeile (**06**) wird es interessanter: Die Ausgabe aus dieser Zeile ist immer noch 2, da der noch unveränderte Wert von `ivar` an den aktuellen Ausdruck weitergegeben wird. Der aktuelle Ausdruck endet am Semikolon, erst hier wirkt sich das Postfix-Inkrement aus dem Ausdruck aus. Der nächste Ausdruck in Zeile (**07**) verwendet den erhöhten Wert 3. Wollen Sie den Wert einer Variablen sofort innerhalb eines Ausdrucks inkrementieren, verwenden Sie statt der Postfix-Schreibweise die Präfix-Schreibweise, wie hier in Zeile (**08**). Dort wird das Ergebnis des Inkrementoperators direkt an den aktuellen Ausdruck gegeben.

Seiteneffekt

Anhand des Beispiels haben Sie gesehen, wie der Wert einer Variablen mithilfe der Inkrement- und Dekrementoperatoren innerhalb eines Ausdrucks verändert wird. Dies bezeichnet man als *Seiteneffekt*. Innerhalb eines Ausdrucks sollten Sie eine Variable auf keinen Fall mehrfach ändern, beispielsweise wie folgt:

```
int ivar = 1;
std::cout << ivar++ << ivar++ << ivar++ << "\n";
```

Die Reihenfolge, in der diese Ausdrücke ausgewertet werden, ist laut Standard unspezifiziert. Daher kann das Programm auf unterschiedlichen Systemen unterschiedliche Ergebnisse erzeugen. Auf einem System mag die Ausgabe gleich »123« sein, auf anderen wiederum »321«. Laut Standard sind beide Ergebnisse möglich. Je nach Einstellung kann Ihr Compiler hier eine Warnung ausgeben, aber vermeiden Sie diese Konstellation lieber.

4.2 Ungenaue Fließkommazahlen

Die Fließkomma-Arithmetik ist nur so genau, wie es die interne Darstellung der Fließkommazahlen erlaubt. Das klingt vielversprechender, als es ist. Das liegt daran, dass reelle Zahlen im Fließkommaformat nicht immer exakt dargestellt werden können. Das Fließkommaformat besteht aus einem Vorzeichen, einem Dezimalbruch und einem Exponenten:

$$\pm f.fff \times 10^{\pm e}$$

Zunächst finden Sie mit ± das Vorzeichen, gefolgt vom Dezimalbruch mit den vier Stellen (f.fff), und am Ende den Exponenten mit einer Stelle ($^{\pm e}$). Die Zahlen werden in der Regel im wissenschaftlichen E-Format (±f.fffE $^{\pm e}$) geschrieben. So wird beispielsweise die Zahl 1.0 im E-Format mit +1.000E+0 dargestellt, die 0.0 mit +0.000E+0 und die –0.006321 mit –6.321E-3. Die Zahl nach dem E gibt dabei an, um wie viele Stellen der Dezimalpunkt verschoben wird. Bei negativen Zahlen nach dem E wird er nach links, bei positiven nach rechts verschoben.

Diese interne Darstellung ist für viele Fälle gut geeignet, hat aber prinzipbedingt auch Schwächen. Rechnen Sie selbst beispielsweise 2/6 + 2/6, kommen Sie auf 4/6. Hier geht aber das Dilemma mit den Rundungsfehlern schon los. 2/6 entspricht im E-Format +3.333E–1. Addieren Sie nun +3.333E–1 und +3.333E–1, erhalten Sie als Ergebnis +6.666E–1 (beziehungsweise 0.6666). Gut, aber leider ist dies nicht das, was Sie erhalten, wenn Sie 4/6 berechnen, denn das sind im E-Format +6.667E–1 und nicht wie berechnet +6.666E–1. Derartige Rundungsfehler können an verschiedenen Stellen vorkommen. Das liegt auch nicht an einer fehlerhaften Implementierung, die man einfach verbessern könnte, sondern an der Form, wie die Fließkommawerte intern behandelt werden. Um unter diesen Voraussetzungen möglichst gut zu arbeiten, geben wir folgenden Rat für die Arbeit mit Fließkommazahlen:

- Vergleichen Sie Fließkommaergebnisse nie auf exakte Gleichheit. Überprüfen Sie stattdessen, ob Ihr Wert im Bereich einer gewissen Toleranz liegt. Statt mit

  ```
  // variable gleich 1.33
  (var == 1.33)
  ```

  prüfen Sie besser so:

  ```
  // var  groesser als 1.3299 und var kleiner als 1.3301
  (var > 1.3299 && var < 1.3301)
  ```

  Die passende Toleranz hängt dabei von den erwarteten Werten ab.

- Falls Sie eine Software entwickeln, die Geldbeträge verwaltet, verwenden Sie zum Speichern dieser Beträge niemals Fließkommazahlen. Bezogen auf die europäische Währung sollten Sie als Grundlage Eurocent und für die Geldbeträge geeignete Ganzzahlen (`int`, `long` oder `long long`) verwenden. Statt mit 9,99 Euro würden Sie also mit 999 Eurocent rechnen.

4.3 Typumwandlung

Sie kennen jetzt unterschiedliche Datentypen. Wir haben aber noch nicht behandelt, was passiert, wenn Typen gemischt werden. Es ist möglich und auch üblich, Elemente unterschiedlicher Datentypen miteinander zu verwenden. Dabei ist es oft erforderlich, die Werte eines Datentyps in einen anderen umzuwandeln. Diese Typumwandlung, auch *Type-Casting* genannt, wird oft automatisch durch den Compiler vorgenommen. Diesen Vorgang bezeichnet man als *implizite Umwandlung*. Er tritt auf, wenn ein Wert eines Typs an einer Stelle verwendet wird, an der ein anderer, kompatibler Datentyp erwartet wird. Typische Situationen für eine implizite Umwandlung sind:

- Initialisierung und Zuweisung
- Binäre Operatoren (arithmetische Ausdrücke und Vergleiche)
- Funktionsaufrufe (Funktionen werden ab Kapitel 8 behandelt.)

Eine Typumwandlung kann aber auch durch den Entwickler explizit angefordert werden. Auf beide Vorgehensweisen gehen wir in den folgenden Abschnitten näher ein.

4.3.1 Implizite Umwandlung durch den Compiler

Der Compiler nimmt eine erlaubte Standard-Typumwandlung automatisch vor, wenn der Typ eines Ausdrucks nicht mit dem erwarteten Typ übereinstimmt. Betrachten wir zuerst einen einfachen Fall:

```
short svar = 1024;
int ivar = 0;
ivar = svar;  // ivar = 1024
int ivar2 = svar;  // ivalr2 = 1024
```

Sowohl bei der Zuweisung an `ivar` als auch bei der Initialisierung von `ivar2` wird der Wert vom Typ `short` vom Compiler in einen `int` umgewandelt und zugewiesen. Der Wertebereich des Datentyps `int` schließt den von `short` komplett ein. Daher ist es garantiert möglich, den Wert unverändert in dem anderen Datentyp darzustellen. Diese Art der impliziten Umwandlung nennt sich *Promotion* oder auch *Anhebung*.

Hier handelt es sich um eine sogenannte Integer-Promotion (im Englischen auch als *integral promotion* bezeichnet), die von links nach rechts in der Folge dieser Datentypen immer möglich ist:

`char`, `short`, `int`, `long`, `long long`

Analog zur Integer-Promotion gibt es auch eine Fließkomma-Promotion, mit der Werte vom Datentyp `float` bei Bedarf implizit auf den Typ `double` angehoben werden. (Achtung, eine Konvertierung zum `long double` ist keine Promotion!)

In den obigen Fällen ist aufgrund der Promotion garantiert, dass der Zieldatentyp den erhaltenen Wert komplett darstellen kann. Wir können aber leicht Fälle finden, in denen das nicht mehr möglich ist. So kann eine Variable vom Typ `short` nur einen Bruchteil der Werte aufnehmen, die als `long int` dargestellt werden können. Auch ein `int` kann nur einen Teil der ganzzahligen Werte eines `double` aufnehmen und die Nachkommastellen überhaupt nicht. Diese Fälle betrachten wir gleich.

Arithmetische Konvertierung

Neben der Konvertierung bei Initialisierung und Zuweisung gibt es einen weiteren wichtigen Fall der automatischen Typumwandlung.

Bei arithmetischen Operationen sowie bei der Nutzung der Vergleichsoperatoren, die wir in Kapitel 5 behandeln werden, müssen die Operanden der arithmetischen Operatoren vom gleichen Typ sein. Dies wird automatisch durch die sogenannte *arithmetische Konvertierung* erreicht. Initialisieren Sie beispielsweise `auto`-Variablen auf die folgende Art und Weise, ergibt sich jeweils ein anderer Datentyp:

```
// listings/04/listing04.cpp
auto auto1 = 1 + 3;     // auto1 vom Typ int
auto auto2 = 1 + 3l;    // auto2 vom Typ long
auto auto3 = 1 + 3.0;   // auto3 vom Typ double
auto auto4 = 1 + 3.0f;  // auto4 vom Typ float
short short1 = 0; long long1 = 0;
auto auto5 = short1 + long1;  // auto5 vom Typ long
```


Die Datentypen beider Operanden werden in einen gemeinsamen Datentyp umgewandelt, und zwar jeweils in den »größeren«. Dieser ist dann auch der Datentyp des Ergebnisses.

Diese »Vereinheitlichung« zu einem Datentyp bei Datentypen unterschiedlicher Größe liegt noch nahe. Die arithmetische Konvertierung sorgt allerdings auch dafür, dass auf den Datentyp `int` angehoben wird, wenn an einer Operation nur integrale Datentypen »kleiner« als `int` beteiligt sind sind, also `char` und `short` (jeweils `signed` und `unsigned`):


```
auto auto6 = 'A' + 'B'; // auto6 vom Typ int
auto auto7 = 'A' + '\0'; // auto7 vom Typ int
```

Diese Umwandlung ist weniger naheliegend und überrascht manchen Entwickler.

Integral-Fließkomma-Typumwandlung

Sie können einem integralen Typ einen Fließkommatyp zuweisen, der integrale Typ kann aber keine Nachkommastelle speichern. Diese Information geht bei der Konvertierung verloren, die Nachkommastelle wird abgeschnitten (es wird nicht abgerundet). Einen Informationsverlust bei der Konvertierung bezeichnet man als *Narrowing*.

```
// listings/04/listing05.cpp
double fwert = 3.14567;
// Fliesskomma-Integral-Umwandlung -> aus 3.14567 wird 3
int iwert = fwert;
```

Lässt sich der Wert ohne den Nachkommaanteil nicht als integraler Wert darstellen, beispielsweise, weil er außerhalb des Wertebereiches liegt, ist das Ergebnis der Umwandlung undefiniert und das Resultat ist abhängig von der Compiler-Implementierung.

Der umgekehrte Fall – wenn Sie einen integralen Typ in einen Fließkommatyp umwandeln oder diesem zuweisen – ist weniger problematisch. Das Ergebnis entspricht einer Gleitkommarepräsentation.

Die automatische Umwandlung kann willkommen sein, aber auch eine Fehlerquelle darstellen. Sie sollten die automatischen Umwandlungen daher kennen und wissen, wie Sie damit umgehen.

Fließkomma-Typumwandlung

Eine Fließkommazahl kann auch von `double` in `float` umgewandelt werden. Sofern allerdings der Wert der zu wandelnden Fließkommazahl in dem Zieldatentyp nicht komplett abgedeckt werden kann, ist auch hier das Ergebnis wieder undefiniert und hängt vom eingesetzten Compiler ab.

Integral-Typumwandlung und die Vorzeichen

Wir haben schon im vorigen Kapitel vor der gemischten Verwendung von Datentypen mit und ohne Vorzeichen gewarnt. Ein `unsigned int` kann negative Werte aus einem `int` oder `signed int` nicht aufnehmen. Auf der anderen Seite sind die maximalen Werte, die ein `unsigned int` darstellen kann, nicht mehr im Gültigkeitsbereich des Datentyps `int`.

Die Regeln für die Typumwandlung zwischen `signed int` und `unsigned int` oder andersherum sind zwar seit C++20 vollständig definiert, aber das Risiko für Fehler bei der Verwendung ist hoch, wenn Sie nicht immer alle Wertebereiche im Blick haben. Wenn Sie die gemeinsame Verwendung dieser Datentypen nicht vermeiden können, sollten Sie sich vorab detailliert mit den erwarteten Wertebereichen und der Umwandlung dieser Datentypen auseinandersetzen.

4.3.2 Automatische Typumwandlung beschränken

Bei der Initialisierung von Variablen können Sie mithilfe der in C++11 eingeführten einheitlichen Initialisierung mit den geschweiften Klammern die versehentliche »Verengung« (*Narrowing*) von Datentypen verhindern, die beispielsweise eintritt, wenn Sie einer Ganzzahl eine Fließkommazahl zuweisen:

```
00  // listings/04/listing06.cpp
01    double dvar = 3.14;
02    int ivar{dvar};  // Fehler !!! Narrowing
03    short svar1{1234};
04    short svar2{svar1 + 1};  // Fehler !!! Narrowing
05    long lval = {svar1 + 1};  // OK!
```


In Zeile (**02**) findet ein Narrowing statt. Die Daten von `dval` passen nicht komplett in `ival`, und somit ist es jetzt ein Fehler. Der Compiler quittiert den Übersetzungsversuch mit einer Fehlermeldung. Das Gleiche gilt für Zeile (**04**), in der ein `int` aus der arithmetischen Konvertierung nicht sicher aufgenommen werden kann. Die Konvertierung in Zeile (**05**) hingegen ist problemlos möglich.

4.3.3 Explizite Typumwandlung

Sie haben gesehen, wie der Compiler automatisch implizite Umwandlungen durchführt. Als Entwickler können Sie Typumwandlungen mit den C++-Typumwandlungsoperatoren auch selbst veranlassen.

C++-Typumwandlungsoperatoren

Die Operatoren, die C++ zur expliziten Typumwandlung stellt, wirken sich beabsichtigt kleinteilig auf die Typumwandlung aus. Auch die Absicht der Umwandlung soll durch den Operator deutlich dargestellt werden.

Den in der Praxis am meisten verwendeten Operator `static_cast<TYP>(ausdruck)` setzen Sie ein, um zwischen verwandten Ganzzahl- und Fließkommatypen zu konvertieren. `TYP` ist dabei der Typ, in den Sie den Ausdruck in den runden Klammern konvertieren wollen:

```
int z = 10, n = 4;
std::cout << (z/n) << '\n'; // Ausgabe: 2
std::cout << (static_cast<double>(z)/n) << '\n'; // Ausgabe: 2.5
```

Durch die Umwandlung des Wertes der Variablen `z` in den Typ `double` wird die Berechnung in der dritten Zeile nicht, wie ohne die Umwandlung, als Ganzzahldivision ausgeführt, sondern als Division von zwei `double`-Werten. (`n` wird aufgrund der arithmetischen Konvertierung bei der Ausführung ebenfalls wie ein `double` behandelt.)

Wenn Sie den `static_cast`-Operator für alle typischen Standard-Typumwandlungen verwenden – auch für die Umwandlungen, die der Compiler ohnehin so ausführen würde, dann bringen Sie in Ihrem Code die Absicht zum Ausdruck, genau diese Umwandlungen wirklich zu wollen, und zeigen, dass hier nicht eine Standard-Typumwandlung versehentlich durchgeführt wird:

```
// listings/04/listing07.cpp
  double dwert = 9.1;
  // explizite Typumwandlung von double nach int
  int iwert = static_cast<int>(dwert);
  // explizite Typumwandlung von int nach float
  float fwert = static_cast<float>(iwert);
  // Konvertiert das Ergebnis in einen int-Wert
  int idoppelt = static_cast<int>(dwert * dwert);
  // Typumwandlungsbeschränkung aufheben
  int ivar{static_cast<int>(dwert)};
  // Mit auto geht es auch
  auto auto1{static_cast<int>(dwert)};
```

Neben dem `static_cast` gibt es in C++ weitere »benannte« Cast-Operatoren, beispielsweise:

```
const_cast<TYP>(ausdruck)
dynamic_cast<TYP>(ausdruck)
reinterpret_cast<TYP>(ausdruck)
```

Insbesondere die explizite Typumwandlung per `reinterpret_cast` und der `dynamic_cast` sind ein relativ kompliziertes und weitreichendes Thema. Sie werden daher an dieser Stelle nicht weiter behandelt.

»Function-Style«-Cast und C-Style-Casts

Es gibt in C++ weiterhin den sogenannten *C-Style-Cast*. Der Name deutet schon an, dass er aus der Sprache C übernommen wurde. Wir zeigen ihn nur, damit Sie ihm aus dem Weg gehen können:

```
double dval = 123.123;
int ival = (int)dval; // C-Style-Cast, nicht verwenden
```

Dem C-Style-Cast sehr ähnlich ist der sogenannte *Function-Style-Cast* von C++:

```
double fval = 123.123;
int ival = int(fval); // Function-Style-Cast
```


Vermeiden Sie diese Casts in C++-Projekten unbedingt, auch wenn der Code erfreulich kompakt aussieht. Beide Casts erlauben eine extreme Bandbreite von Umwandlungen. Es lässt sich praktisch jede Konvertierung zwischen verwandten, nicht verwandten oder höheren Typen erzwingen. Die Folgen gehen bis zu Programmabstürzen, auch wenn wir für einen solchen Fall hier noch keine Beispiele aufführen können.

Bei diesen Casts wird nicht direkt deutlich, was alles konvertiert werden wird, zudem sind sie mit einer Textsuche nur schwer aufzufinden. Mit den C++-Casts muss die vom Programmierer verlangte explizite Typumwandlung zumindest ausgeschrieben werden. Die explizite Schreibweise der C++-Casts sorgt zusätzlich dafür, dass sie sich mit einer Textsuche leicht lokalisieren lassen.

4.4 Kontrollfragen und Aufgaben

1. Was sind Typumwandlungen, und welche Arten von Umwandlungen gibt es?
2. Welcher Wert wird in den folgenden Zeilen ausgegeben?

   ```
   int i = 1;
   std::cout << i-- << '\n';
   std::cout << ++i << '\n';
   std::cout << i++ << '\n';
   std::cout << ++i << '\n';
   ```

3. Welche Werte werden hier ausgegeben?

   ```
   int ival1, ival2 = 2;
   double dval1 = 3.5;
   ival1 = dval1;
   std::cout << ival1 << '\n';
   std::cout << dval1 / ival2 << '\n';
   std::cout << static_cast<int>(dval1 / ival2) << '\n';
   std::cout << static_cast<double>(dval1 / ival2) << '\n';
   ```

4. Schreiben Sie ein Listing, das eine Strecke in Kilometern (`km`) abfragt. Diesen eingelesenen Wert rechnen Sie in das Streckenmaß Meilen (`ml`) um (1 Meile = 1,60934 km). Verwenden Sie `double` als Basisdatentyp.

Kapitel 13

Operatoren überladen

Neben Funktionen und Methoden (vergleiche Abschnitt 8.1.8, »Funktionen überladen«) können in C++ auch Operatoren überladen werden. Sie können damit die Bedeutung von Operatoren in Verbindung mit benutzerdefinierten Datentypen (`enum`, `struct` und `class`) anpassen. Ein Beispiel für die Verwendung von überladenen Operatoren ist das Zusammenfügen von Strings (auch *konkatenieren* oder *concat* genannt und oft als `cat` abgekürzt):

```
std::string s1 {"Hallo "}, s2{"Welt\n"};
std::cout << s1+s2; // Ausgabe: Hallo Welt
```

Meist ist diese sogenannte *Infix-Notation* mit dem Operator-Zeichen `+` zwischen den Operatoren leichter zu lesen als beispielsweise die Notation als Methode `s1.cat(s2)` oder als Funktion `cat(s1, s2)`.

Das obige Beispiel, in dem zwei Strings mit dem Operator + zu einem neuen String »addiert« werden, wird den meisten Verwendern gut lesbar und relativ natürlich erscheinen. In diesem Fall ähnelt die Bedeutung auch stark der des arithmetischen Operators +. Die Funktion eines überladenen Operators kann aber auch von der ursprünglichen Bedeutung abweichen.

Ein Großteil der Operatoren kann überladen werden. Sie können auch die Operatoren << und >> so überladen, dass sie Ihre Objekte geeignet ausgeben und einlesen und, ohne dass wir es ausführlich erläutert hätten, haben Sie mit der Verwendung von `[]` und `()` für den Zugriff auf Elemente von `std::vector` schon weitere Überladungen gesehen.

Die in Tabelle 13.1 gelisteten Operatoren können Sie überladen, die wenigen Ausnahmen sind in Tabelle 13.2 aufgeführt.

Operatoren	Bedeutung
`+ - * / % -(unär) + -- += -= *= /= %=`	arithmetische Operatoren
`&& || !`	logische Operatoren
`== != < <= > >=`	Vergleichsoperatoren
`<=>`	Drei-Wege-Vergleich (engl. wegen der Form auch *spaceship operator*), seit C++20
`=`	Zuweisungsoperator
`& | ^ ~ >> << &= |= ^= >>= <<=`	Bit-Operatoren
`""`	benutzerdefiniertes Literal
`* -> ->* , () [] & new delete new[] delete[]`	sonstige Operatoren

Tabelle 13.1 Operatoren, die überladen werden können

Operatoren	Bedeutung
`.`	Zugriffsoperator
`.*`	Zugriffsoperator (Elementzeiger)
`::`	Scope-Operator (Bereichsoperator)
`?:`	ternärer Operator (bedingte Auswertung)
`sizeof`	Größe von Objekten

Tabelle 13.2 Operatoren, die nicht überladen werden können

13.1 Das Schlüsselwort »operator«

Dass in C++ Operatoren überladen werden können, verdanken wir den sogenannten *Operatorfunktionen*. Dies sind Funktionen, die mit dem Schlüsselwort `operator` und dem eigentlichen Operatorensymbol vom Compiler aufgerufen werden. Die Operatoren, die in Tabelle 13.1 aufgelistet sind, können im Quellcode auf zwei unterschiedliche Varianten verwendet werden. Die übliche Verwendung `objekt1 + objekt2`, um beispielsweise zwei Werte zu addieren, kennen Sie ja bereits. Wenn mindestens einer der beteiligten Operanden `objekt1` oder `objekt2` einen benutzerdefinierten Datentyp hat, ruft der Compiler unter der Haube die Funktion `operator+(objekt1, objekt2)` oder die Methode `objekt1.operator+(objekt2)` mit der passenden Signatur auf. Es handelt sich hierbei um die *Operatorfunktionen*, d. h. Funktionen, die mit dem Schlüsselwort `operator`, gefolgt von dem entsprechenden Operator, benannt sind. Diese Funktionen und Methoden können Sie wie gewohnt überladen und damit das gewünschte Verhalten für eigene Operatoren implementieren.

Die Syntax einer solchen Operatorfunktion oder -methode sieht so aus:

```
Rückgabewert [Klassenname::]operator@ ( Parameter )
```


Das Zeichen `@` hinter dem Schlüsselwort `operator` müssen Sie durch einen gültigen, überladbaren Operator ersetzen (siehe Tabelle 13.1). Zwischen dem Schlüsselwort `operator` und dem Symbol darf auch Whitespace, beispielsweise ein Leerzeichen, stehen. Die Anzahl der Parameter (keiner, einer oder zwei) hängt von dem zu überladenden Operator ab.

Die Regeln bei der Operatorüberladung

Mit der Operatorüberladung können Sie vieles machen. Es gibt aber durchaus einige Einschränkungen:

- Operatorüberladungen finden im Zusammenhang mit benutzerdefinierten Datentypen statt. Operatoren, die nur auf eingebauten Basisdatentypen arbeiten, können Sie nicht »umbiegen«.
- Sie können nur Operatoren überladen, die bereits existieren. Es lassen sich keine neuen Operatorsymbole einführen.

- Die Anzahl der zu einem Operator gehörigen Operanden kann nicht verändert werden. Ein binärer Operator hat nach wie vor zwei Operanden und ein unärer einen. Der einzige ternäre Operator `?:` kann nicht überladen werden.
- Auch die Priorität und Links-nach-rechts- bzw. Rechts-nach-links-Bindung der Operatoren verändert sich nicht. Zum Beispiel: Der Operator `*` besitzt eine höhere Priorität als der Operator `+` (Punkt-vor-Strich-Regelung).
- Überladene logische Operatoren, also `&&` (UND) sowie `||` (ODER), haben nicht mehr die in Kapitel 5, »Kontrollstrukturen«, beschriebene Kurzschlussauswertung.
- Operatoren dürfen keine Standardargumente erhalten.

13.2 Zweistellige (arithmetische) Operatoren überladen

Zur Demonstration der Operatorüberladung verwenden wir in diesem Kapitel die Klasse `Dint`. Sie ist eine simple Klasse ohne besondere Bedeutung, daher können wir die Funktion unserer Operatoren sehr leicht nach Wunsch definieren. Die Klasse nutzt je ein Attribut vom Typ `int` und vom Typ `double`, um Werte zu speichern. Hier sehen Sie einen Ausschnitt der Klassendefinition:

```
00  // listings/13/dint01/dint.h
01  class Dint {
02   public:
03    Dint();
04    Dint(double d, int i);
05    void print() const;
...
06   private:
07    double dval = 0.0;
08    int ival = 0;
09  };
```


Wir beginnen mit der Überladung der beiden Operatoren + und +=. Das hier gezeigte Prinzip können Sie aber auch auf andere arithmetische Operatoren anwenden.

Wir wollen erreichen, dass die folgenden Operationen möglich sind:

```
01  Dint dint01{1.1, 100};
02  Dint dint02{2.2, 200};
03  Dint dint03 = dint01 + dint02;
04  Dint dint04 = dint01 + 3.3;
05  Dint dint05 = dint02 += dint03;
```

Im Augenblick würde jede dieser Zeilen eine Fehlermeldung des Compilers verursachen.

Aus den Anweisungen lassen sich bereits einige Erwartungen herauslesen, die an die Operatoren gestellt werden. In den Zeilen **(03)** und **(04)** zeigt sich die Erwartung, dass das Ergebnis des Additionsoperators ein neues Objekt ist, mit dem beispielsweise initialisiert werden kann. In Zeile **(05)** erkennt man, dass die Zuweisungsaddition `+=` nicht nur `dint02` ändern soll, sondern dass das Resultat ebenfalls für eine Initialisierung verwendet werden soll.

Operatoren nachvollziehbar verwenden

Achten Sie bei Operatorüberladungen darauf, dass sie bei der Verwendung nachvollziehbar und leicht handhabbar ist. Wie oben gezeigt, erwarten die meisten Nutzer, dass Operationen verkettet werden können, beispielsweise `a = b + c += d;`, und dass bei einer Addition `a + b` die Operanden `a` und `b` nicht verändert werden. Überladen Sie beispielsweise `==`, sollten Sie auch sein Gegenstück `!=` überladen.

Es ist möglich, bei der Implementierung überladener Operatoren gegen solche Erwartungen zu verstoßen. Damit machen Sie allerdings deren Verwendung schwierig und fehleranfällig. Geht die Logik der Operatorüberladung nicht vollständig auf, ist es häufig sinnvoller, (zusätzlich) eine Methode zu implementieren.

Was die Operatoren in unserer Klasse inhaltlich machen sollen, ist ganz uns überlassen. Wir werden sie für unsere Klasse `Dint` so implementieren, dass die einzelnen Attribute eines Objekts der Klasse `Dint` mit den entsprechenden Attributen des zweiten Operanden addiert werden.

13.2.1 Operatorüberladung als Methode einer Klasse

Wir wollen für die Operatoren `+` und `+=` also je eine Variante implementieren, die zwei Argumente vom Typ `Dint` bekommt sowie eine Variante mit `Dint` und `double`.

Die Addition mit dem `+` soll als Ergebnis ein neues `Dint`-Objekt zurückgeben. Der Operator `+=` gibt eine Referenz `Dint&` zurück, damit die Operation wie bei den Basisdatentypen verkettet werden kann. Wir implementieren die Operatoren als Methoden der Klasse `Dint`. Der erste Parameter ist dabei immer vom Typ `Dint` und entspricht der Instanz, für die die Methode aufgerufen wird. Die Headerdatei *dint.h* sieht damit wie folgt aus:

```
01  // listings/13/dint01/dint.h
02  class Dint {
03   public:
...
04    Dint operator+(const Dint& other);
05    Dint operator+(const double d);
06    Dint& operator+=(const Dint& other);
07    Dint& operator+=(const double d);
08   private:
09    double dval = 0.0;
10    int ival = 0;
11  };
```

Bevor wir Ihnen zeigen, wie Sie diese überladenen Operatoren in der Praxis einsetzen können, möchten wir hier noch die Definitionen aus der Quelldatei *dint.cpp* wiedergeben:

```
00  // listings/13/dint01/dint.cpp
...
01  Dint Dint::operator+(const Dint& other) {
02    Dint tmp{*this};
```



```
03    tmp.dval += other.dval;
04    tmp.ival += other.ival;
05    return tmp;
06  }
07  Dint Dint::operator+(const double d) {
08    Dint tmp{*this};
09    tmp.dval += d;
10    return tmp;
11  }
12  Dint& Dint::operator+=(const Dint& other) {
13    this->dval += other.dval;
14    this->ival += other.ival;
15    return *this;
16  }
17  Dint& Dint::operator+=(const double d) {
18    this->dval += d;
19    return *this;
20  }
```

Der Aufruf von `dint01+dint02` entspricht dem Aufruf `dint01.operator+(dint02);`.

Dieser Operator wird in den Zeilen **(01)** bis **(06)** definiert. Dazu wird in Zeile **(02)** eine Kopie erzeugt und mit dem aufrufenden Objekt (`*this`) initialisiert. Die übergebenen Werte des zweiten Operanden `other` werden entsprechend zu den Werten des kopierten Objekts addiert, das dann als Rückgabewert dient.

Das praktisch gleiche Vorgehen finden Sie auch in Zeile **(07)** bis **(11)**. Allerdings wird hier vor der Rückgabe nur ein `double`-Wert als Argument zur Klassenvariablen `tmp` hinzuaddiert.

Beim Überladen beider Varianten von `operator+=` in Zeile **(12)** bis **(20)** wird kein lokales Objekt anlegt, die sogenannte Zuweisungsaddition soll den ersten Operanden ja ausdrücklich verändern. Dieser erste Operand entspricht dem Objekt (`*this`), das jeweils verändert wird. Das bereits geänderte Objekt wird in Zeile **(15)** und **(19)** jeweils als Referenz zurückgegeben. Mit dieser Implementierung können die Operatoren verwendet werden.

Dabei ist die Langform möglich, die ja auch konkret so implementiert worden ist, beispielsweise mit dem Aufruf:

```
Dint dint03 = dint01.operator+(dint02);
```

Glücklicherweise müssen Sie nicht diese Form verwenden. Nutzen Sie stattdessen die Infix-Notation mit dem Operator zwischen den Operanden die der Compiler durch die überladene Variante ersetzt:

```
Dint dint03 = dint01 + dint02;
```

Sie sehen, dass man bei geeigneter Auswahl und Implementierung mit der Operatorüberladung hervorragende Schnittstellen anbieten kann. Sie sollten den Einsatz aber auf uneingeschränkt geeigneten Fälle begrenzen.

Zum Schluss dieses Abschnitts noch das Hauptprogramm, das die Verwendungsmöglichkeiten der überladenen Operatoren demonstriert:

```
00  // listings/13/dint01/main.cpp
01  int main() {
02    Dint dint01 {1.1, 100};
03    Dint dint02 {2.2, 200};
04    Dint dint03 = dint01 + dint02;
05    dint01.print();
06    dint02.print();
07    dint03.print();
08    Dint dint04 = dint01 + 3.3;
09    dint04.print();
10    dint02 += dint03;
11    dint02.print();
12    return EXIT_SUCCESS;
13  }
```

Die **fett** hervorgehobenen Stellen verwenden Operatorüberladungen.

Besonderheiten und Empfehlungen

- Zweistellige Operatoren wie `x+y` sollten ein neues Objekt zurückliefern. Damit vermeiden Sie schwer auffindbare Fehler.

- Wenn Sie einen zweistelligen Operator wie `x+y` überladen haben, sollten Sie auch die zusammengesetzte Version `+=` implementieren.
- Zusammengesetzte Operatoren wie `+=` sollten Sie, wie hier gezeigt, als Methode implementieren, da die notwendige Änderung des linken Operanden hier einfacher zu bewerkstelligen ist als bei der Implementierung als Hilfsfunktion, die Sie noch kennenlernen werden.
- Bevorzugen Sie als Parameter für die Überladung als Argumenttyp dieselbe Klasse wie beispielsweise in `dint11+=dint2`. Eine solche Verwendung ist sehr gängig. Wenn es sinnvoll ist, können Sie auch Überladungen für andere Typen hinzufügen, wie in `dint3+=33.33`.

13.2.2 Operatorüberladung als globale Hilfsfunktion

Sie können die Operatorüberladung nicht nur als Methode einer Klasse implementieren. Sie können sie stattdessen auch als globale Hilfsfunktion programmieren. Sofern Sie dabei auf die privaten Daten einer Klasse zugreifen wollen, müssen Sie diese globale Hilfsfunktion als `friend` deklarieren (siehe Abschnitt 12.2, »Freundfunktionen (›friend‹)«).

Operatoren mit Anbindung an die Klasse

Den Zuweisungsoperator `=`, den Operator `()`, den Indexoperator `[]` und den Zeigeroperator `->` können Sie nicht als globale Funktion überladen.

Wir möchten Ihnen gleich einen Fall präsentieren, in dem Sie die Operatorüberladung nicht als Methode der Klasse implementieren können. Bei dieser Verwendung des +-Operators

```
Dint dint01 {1.1, 100};
Dint dint02 = 6.6 + dint01;
```

wird sich der Compiler beschweren, dass es keinen Operator mit der entsprechenden Signatur gibt. Bei der bisher verwendeten Operatorüberladung in der Klasse ist der linke Operand immer als Objekt der Klasse implementiert. Im aktuellen Beispiel ist der linke Operand aber ein Literal

des Typs `double`, für den wir keine Methoden implementieren können. Wir können hier allerdings eine Operatorüberladung als globale Funktion mit den entsprechenden zwei Parametern erstellen. Die Deklaration in der Headerdatei *dint.h* sieht dann folgendermaßen aus:

```
00  // listings/13/dint02/dint.h
01  class Dint {
02    public:
03    Dint();
...
04      friend Dint operator+(double d, const Dint& di);
05    private:
06    double dval = 0.0;
07    int ival = 0;
08  };
```

Da wir für die Operatorüberladung innerhalb der globalen Funktion `operator+` auf private Daten der Klasse zugreifen wollen, kennzeichnen wird sie mit dem Schlüsselwort `friend`. Mit der Implementierung einer öffentlichen `get_dval()`-Methode könnte dieser Operator aber auch ohne `friend` implementiert werden.

Aufrufen können wir diese globale Funktion jetzt zum Beispiel wie folgt:

```
Dint dint02 = operator+(6.6, dint01);
```

Glücklicherweise können Sie auch hier die wesentlich lesefreundlichere Version mit derselben Bedeutung verwenden:

```
Dint dint02 = 6.6 + dint01;
```

Die Implementierung für dieses Beispiel in der Quelldatei *dint.cpp* entspricht im Wesentlichen der Implementierung der Operatorüberladung als Methode:

```
00  // listings/13/dint02/dint.cpp
...
01  Dint operator+(double d, const Dint& di) {
02    Dint tmp = di;
```



```
03    tmp.dval += d;
04    return tmp;
05  }
```

Die Funktion erhält als ersten Parameter für den linken Operanden einen `double`-Wert und als zweiten Parameter für den rechten Operanden eine Referenz auf ein konstantes `Dint`-Objekt. Sie erzeugt aus dem rechten Operanden ein temporäres Objekt vom Typ `Dint`, zu dessen `dval`-Wert der `double`-Wert des linken Operanden addiert wird, bevor die Funktion das temporäre Objekt als Ergebnis zurückliefert.

Jetzt fehlt nur noch ein Hauptprogramm, das die Verwendung der globalen Operatorüberladungsfunktion in der Praxis demonstriert:

```
00  // listings/13/dint02/main.cpp
...
01  #include "dint.h"
02  int main() {
03    Dint dint01 {1.1, 100};
04    Dint dint02 = operator+(5.5, dint01);
05    Dint dint03 = 6.6 + dint01;
06    dint02.print();
07    dint03.print();
08  }
```


13.3 Einstellige Operatoren überladen

Die unären Inkrement- und Dekrementoperatoren `++` und `--` stellen einen Spezialfall unter den Operatoren dar, da sie in zwei verschiedenen Varianten auftreten, nämlich in der Präfix- (`++val`) und in der Postfix-Schreibweise (`val++`).

Wir werden für beide Varianten eine Überladung erstellen. Beide haben den gleichen Bezeichner; die Postfix-Signatur der Überladung unterscheidet sich durch einen Dummy-Parameter von der Präfix-Signatur.

Wir beginnen mit dem einfacheren Fall, der Präfix-Variante (`++val`). Die Deklaration in der Klassendefinition für die Präfix-Schreibweise des `++`-Operators bzw. des `---`Operators sieht so aus:

```
00  // listings/13/dint03/dint.h
...
01  class Dint {
02    public:
...
03      // Präfixversionen
04      Dint& operator++();  // ++objekt
05      Dint& operator--();  // --objekt
...
06    private:
07    double dval = 0.0;
08    int ival = 0;
09  };
```

Bezogen auf unsere Klasse `Dint` verwenden wir die Überladung der Inkrement- und Dekrementoperatoren, um beide Klassenvariablen `dval` und `dint` um den Wert 1 zu erhöhen beziehungsweise zu verringern.

Die Definition des Quellcodes zur Präfix-Methode in *dint.cpp* sieht wie folgt aus:

```
00  // listings/13/dint03/dint.cpp
...
01  Dint& Dint::operator++() {
02    dval += 1.0;
03    ival += 1;
04    return *this;
05  }
06  Dint& Dint::operator--() {
07    dval -= 1.0;
08    ival -= 1;
09    return *this;
```

Die Verwendung der Präfix-Schreibweise birgt keine Überraschungen. Zunächst werden in Zeile **(02)** und **(03)** beziehungsweise **(07)** und **(08)** die

beiden Klassenvariablen `dval` und `ival` um den Wert 1 inkrementiert bzw. dekrementiert. Anschließend wird das aufrufende Objekt als Ganzes zurückgegeben.

Etwas anders müssen Sie bei der Postfix-Schreibweise (`val++`) vorgehen. Zuerst einmal ist hier die Schnittstelle unterschiedlich. Die Überladungen der Postfix-Schreibweise unterscheiden sich in der Signatur durch einen zusätzlichen Parameter vom Typ `int`. Wir haben den Parameter in der Schnittstelle auch gar nicht benannt: Er wird nicht verwendet, sondern dient nur dazu, eine eigene Signatur zur Unterscheidung der Varianten zu generieren.

Hier sehen Sie die Deklaration zum Überladen der Operatoren in der Postfix-Schreibweise in *dint.h*:

```
00  // listings/13/dint03/dint.h
...
01  class Dint {
02    public:
...
03      // Postfix-Versionen
04      Dint operator++(int);  // objekt++
05      Dint operator--(int);  // objekt--
...
06    private:
07    double dval = 0.0;
08    int ival = 0;
09  };
```

Hier folgen die Definitionen der Postfix-Versionen des Inkrement- und Dekrementoperators in der Quelldatei *dint.cpp*:

```
00  // listings/13/dint03/dint.cpp
...
01  Dint Dint::operator++(int){
02    Dint tmp{*this};
03    dval+=1.0;
04    ival+=1;
05    return tmp;
```

```
06  }
07  Dint Dint::operator--(int) {
08    Dint tmp{*this};
09    dval-=1.0;
10    ival-=1;
11    return tmp;
12  }
```

Zunächst sichern Sie bei beiden Überladungen die aufrufende Instanz in einem temporären Objekt in Zeile **(02)** bzw. **(08)**. Dann verändern Sie den Inhalt des *aufrufenden* Objekts (`*this`) in Zeile **(03)** und **(04)** bzw. in Zeile **(09)** und **(10)**. Am Ende wird das unveränderte temporäre Objekt in Zeile (**05**) bzw. **(11)** zurückgegeben. Dieser Umweg über das temporäre Objekt ist nötig, damit die Regeln der Postfix-Notation eingehalten werden.

Zum Schluss noch ein Hauptprogramm, das die überladenen Inkrement- und Dekrementoperatoren für die Klasse `Dint` in der Praxis zeigt:

```
00  // listings/13/dint03/main.cpp
01  #include "dint.h"
02  int main() {
03    Dint dint01 {1.1, 100};
04    dint01.print();      // = 1.1 100
05    (dint01++).print();  // = 1.1 100
06    dint01.print();      // = 2.1 101
07    (++dint01).print();  // = 3.1 102
08    (--dint01).print();  // = 2.1 101
09    return EXIT_SUCCESS;
10  }
```

Bevorzugen Sie die Präfix-Varianten

Die Präfix-Varianten `operator++()` und `operator--()` verändern das Objekt mit `++a` und `--a` sofort und liefern es auch als Referenz zurück. Bei den Postfix-Varianten `operator++(int)` und `operator--(int)` für `a++` und `a--` muss erst eine Kopie des alten Wertes erstellt und zurückgeliefert werden. Daher sollten Sie in der Praxis die Präfix-Varianten bevorzugen, in denen kein unnötiges Dummy-Objekt erzeugt werden muss.