Die Programmier-sprache C++

Inhaltsverzeichnis

Kapitel 1. Einführung

1.1. Historisches zu C und C++

„C++ makes it much harder to shoot yourself in the foot, but when you do, it blows off your whole leg.“ – Bjarne Stroustrup.

„In C we had to code our own bugs. In C++ we can inherit them.“ – Prof. Gerald Karam.

„Fifty years of programming language research, and we end up with C++ ???“ Richard A. O'Keefe, Computer scientist, concentrating on languages for logic programming and functional programming (including Prolog and Haskell).

„I invented the term 'Object-Oriented', and I can tell you I did not have C++ in mind.“ Alan Kay, creator of Smalltalk.

„C++: Hard to learn and built to stay that way.““

„Java is, in many ways, C++--.“ – Michael Feldman, Professor Emeritus at the George Washington University, Department of Computer Science, Washington.

„Writing in C or C++ is like running a chain saw with all the safety guards removed.“ – Bob Gray.

„Ever spend a little time reading comp.lang.c++ ? That's really the best place to learn about the number of C++ users looking for a better language.“ – R. William Beckwith.

„The evolution of languages: FORTRAN is a non-typed language. C is a weakly typed language. Ada is a strongly typed language. C++ is a strongly hyped language.“ – Ron Sercely.

„C(++) is a write-only, high-level assembler language.“ – Stefan Van Baelen.

„C++ : Where friends have access to your private members.“ – Gavin Russell Baker.

„C++ would make a decent teaching language if we could teach the ++ part without the C part.“ – Michael B. Feldman.

„The great thing about Object Oriented code is that it can make small, simple problems look like large, complex ones.“

„Hybrid ('half-assed') object languages like C++ are worst of all, as they unite the simplicity of Brainfuck with the inherent security of C and the speed of Perl.“ – Tony.

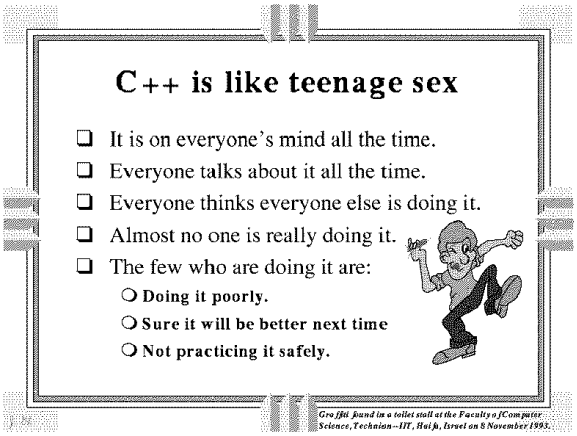

Abbildung 1.1. Toilettengraffiti am Computer Science Department der Technion (Israel Institute of Technology) in Haifa, Israel.

1.1.1. Entstehung von C

Die Wurzeln in der Entstehung der Programmiersprache C reichen bis in die Mitte der sechziger Jahre zurück, als *Martin Richards* die Programmiersprache BCPL (*Basic Combined Programming Language*) enwickelte. Sie wiederum inspirierte *Ken Thompson* von den AT&T Bell Labaratories im Jahre 1970 zur Entwicklung einer Programmiersprache B, einem Derivat von BCPL. Mit BCPL entwickelte Thompson im Jahre 1970 eine Version des UNIX-Betriebssystems für eine DEC-PDP-7.

BCPL und B sind typenlose Sprachen, die eine sehr große Nähe zu Assembler aufweisen. Aus diesem Grund entwickelt *Dennis MacAlistair Ritchie* – ebenfalls seit 1967 bei den Bell Laboratories im Computing Sciences Research Center beschäftigt – eine neue Version der Programmiersprache B, um damit ein UNIX-Betriebssystem zu schreiben, dieses Mal für eine DEC-PDP-11.

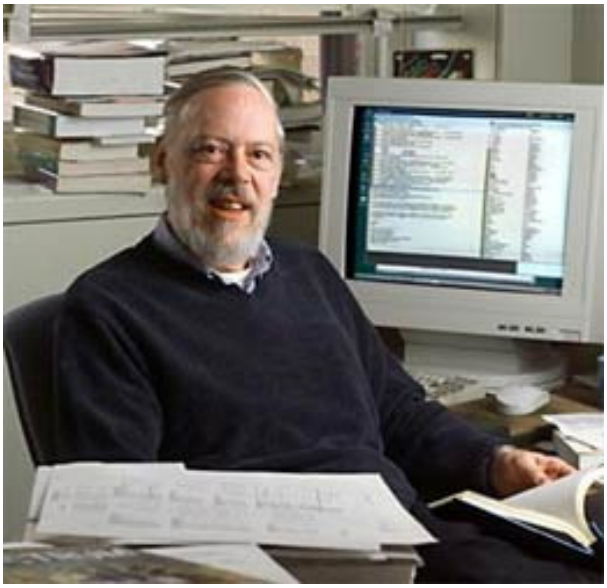

Abbildung 1.2. Der maßgebliche Erfinder von C: Dennis MacAlistair Ritchie.

Ritchie wurde am 9. September 1941 in Bronxville, New York geboren, studierte an der Harvard University Physik und Angewandte Mathematik und ist einer der Computer-Pioniere. Die mit Hilfe von Ritchie entstehende Version von UNIX ist zu über 90% in C geschrieben, insbesondere der C-Compiler selbst. Die Programmiersprache C war geboren, wir können ihre Genese dem Jahr 1972 zuordnen. Im Gegensatz zu B oder BCPL ist C jedoch nicht hardware- oder systemgebunden, und es ist nun erstmalig möglich, systemnahe Software rechnerunabhängig (portabel) schreiben zu können.

Neben Dennis M. Ritchie arbeitete auch *Brian W. Kernighan* an der Weiterentwicklung von C mit. Die beiden Entwickler bringen 1978 das ultimative Standardwerk zur Programmiersprache C auf den Markt. Es trägt den Namen „The C Programming Language“ und ist das erste Referenzbuch der von ihnen entwickelten Programmiersprache C. Im Jahr 1983 wird ein Kommitee des American National Standards Institute (genauer das Technische Komitee X3J11) einberufen, um einen einheitlichen Standard für die Programmiersprache C zu schaffen. Nach sechs Jahren Arbeit wird 1988 unter der Bezeichnung X3.159-1989 ein eindeutiger und maschinenunabhänger ANSI-C-Standard veröffentlicht. Dieser Standard erweitert beispielsweise das klassische „K&R“-C um Funktionsprototypen und den Modifizierer `const`.

1.1.2. Entstehung von C++

Abbildung 1.3. Der Erfinder von C++: Bjarne Stroustrup.

Die Programmiersprache C++ entsteht in den frühen 80er Jahren, wiederum in den Bell Laboratories. Ihre Sprachdefinition wird maßgeblich beeinflusst von *Bjarne Stroustrup*. Eine Pilotsprache „C mit Klassen" diente als Vorlage, sie griff auf Strukturen und Konzepte von C zurück und war um objektorientierte Elemente erweitert. „C mit Klassen" wiederum besaß eine Reihe von Anleihen bei *Simula-67*, die man wohl ale älteste objektorientierte Programmiersprache bezeichnen kann. Sie wurde im Jahre 1967 konzipiert, kannte bereits Konzept der Klassen, der abgeleiteten Klassen und der virtuellen Funktionen. Zur damaligen Zeit kamen diese Konzepte jedoch früh und Simula-67 konnte sich am Markt nicht durchsetzen. 1983 prägte *Rick Mascitti* den Begriff *C++* in Anlehnung an den Inkrementoperator ++ von C. In der Tat wurde auch der Name *D* diskutiert. Da C komplett in C++ als Teilmenge enthalten sein sollte, wollte man dies in der Bezeichnung des Namens auch zum Ausdruck bringen und verwarf den Namen *D*.

Im Jahe 1985 existierte in der Firma AT&T die erste Version eines C++-Compilers. In den darauffolgenden sechs Monaten glückte die kommerzielle Portierung dieses Übersetzers auf über 24 verschiedene Systeme. AT&T Bell Laboratories gestand Bjarne Stroustrup das Recht zu, das Referenzhandbuch seines Sprachentwurfs mit anderen zu diskutieren und es weiterzugeben. Durch Informationen und Diskussionen, die in der Newsgroup *comp.lang.c++* geführt wurden, kam es zu einer schnellen Verbreitung und Akzeptanz der neugeschaffenen Programmiersprache C++. In Papierform wurde C++ im Jahre 1986 durch das Referenzhandbuch *The C++ Programming Language* beschrieben.

Die Standardisierungsbemühungen des American National Standards Institutes (ANSI) wurden 1989 durch die Gründung des Kommitees X3J16 auf Initiative von Hewlett Packard vorangetrieben. Das standardisierte C++ soll auf die AT&T-Version 2.0 des C++-Compilers aufbauen sowie im wesentlichen ANSI C beinhalten. 1990 veröffentlichten *Margaret A. Ellis* und Bjarne Stroustrup das Referenzmanual *The Annotated C++ Reference Manual* im Addison-Wesley-Verlag. Ab diesem Zeitpunkt finden regelmäßig Konferenzen über C++ statt, die neue Entwicklungen der Sprache diskutieren. Im März 1996 veröffentlichen das Komitee X3J16 der ANSI und die Arbeitsgruppe WG21 der ISO in Austin, Texas ein Dokument, das als „*Draft C++ Standard*" bekannt wurde. Nach neun Jahren Arbeit verabschieden diese beiden Standardisierungsgremien den ANSI C++-Standard (ISO/IEC 14882).

1.1.3. Abgrenzung von anderen Programmiersprachen

Die starke Zunahme moderner Programmiersprachen (die alle ohne Rücksicht auf irgendwelche Verluste unzählige Sprachelemente gegenseitig voneinander abschreiben) erschwert es dem eigenen Gedächtnis manchmal ungemein, sich zu vergegenwärtigen, in welcher Programmiersprache man gerade denkt. Um einen Ausweg aus diesem Dilemma zu finden, bietet die Anwendung des folgenden Referenzhandbuchs eine praktische Hilfestellung, die auf die folgende Aufgabenstellung anzuwenden ist:

Aufgabe: *Shoot yourself in the foot.*

C: *You shoot yourself in the foot.*

C++: *You accidentally create a dozen instances of yourself and shoot them all in the foot. Providing emergency medical assistance is impossible since you can't tell which are bitwise copies and which are just pointing at others and saying, „That's me, over there."*

FORTRAN: *You shoot yourself in each toe, iteratively, until you run out of toes, then you read in the next foot and repeat. If you run out of bullets, you continue with the attempts to shoot yourself anyway because you have no exception-handling capability.*

Pascal: *The compiler won't let you shoot yourself in the foot.*

Ada: *After correctly packing your foot, you attempt to concurrently load the gun, pull the trigger, scream, and shoot yourself in the foot. When you try, however, you discover you can't because your foot is of the wrong type.*

COBOL: *Using a COLT 45 HANDGUN, AIM gun at LEG.FOOT, THEN place ARM.HAND.FINGER on HANDGUN.TRIGGER and SQUEEZE. THEN return HANDGUN to HOLSTER. CHECK whether shoelace needs to be re-tied.*

LISP: *You shoot yourself in the appendage which holds the gun with which you shoot yourself in the appendage which holds the gun with which you shoot yourself in the appendage which holds the gun with which you shoot yourself in the appendage which holds the gun with which you shoot yourself in the appendage which holds the gun with which you shoot yourself in the appendage which holds ...*

FORTH: *Foot in yourself shoot.*

Prolog: *You tell your program that you want to be shot in the foot. The program figures out how to do it, but the syntax doesn't permit it to explain it to you.*

BASIC: *Shoot yourself in the foot with a water pistol. On large systems, continue until entire lower body is waterlogged.*

Visual Basic: *You'll really only _appear_ to have shot yourself in the foot, but you'll have had so much fun doing it that you won't care.*

HyperTalk: *Put the first bullet of gun into foot left of leg of you. Answer the result.*

Motif: *You spend days writing a UIL description of your foot, the bullet, its trajectory, and the intricate scrollwork on the ivory handles of the gun. When you finally get around to pulling the trigger, the gun jams.*

APL: *You shoot yourself in the foot, then spend all day figuring out how to do it in fewer characters.*

SNOBOL: *If you succeed, shoot yourself in the left foot. If you fail, shoot yourself in the right foot.*

Unix:

```
% ls
foot.c foot.h foot.o toe.c toe.o
% rm * .o
rm:.o no such file or directory
% ls
%
```

Concurrent Euclid: *You shoot yourself in somebody else's foot.*

370 JCL: *You send your foot down to MIS and include a 400-page document explaining exactly how you want it to be shot. Three years later, your foot comes back deep-fried.*

Paradox: *Not only can you shoot yourself in the foot, your users can, too.*

Access: *You try to point the gun at your foot, but it shoots holes in all your Borland distribution diskettes*

instead.

Revelation: *You're sure you're going to be able to shoot yourself in the foot, just as soon as you figure out what all these nifty little bullet-thingies are for.*

Assembler: *You try to shoot yourself in the foot, only to discover you must first invent the gun, the bullet, the trigger, and your foot.*

Modula2: *After realizing that you can't actually accomplish anything in this language, you shoot yourself in the head.*

CLARION: *You tell your computer to create a program for shooting yourself in the foot.*

1.2. Wichtige Veränderungen und Erweiterungen gegenüber C

1.2.1. Kommentare

Zusätzlich zu den von C her bekannten Kommentaren, die durch `/*` und `*/` begrenzt werden, gibt es in C++ einzeilige Kommentare. Sie werden mit `//` eingeleitet, das Kommentarende ist implizit durch das Ende der Zeile gegeben:

```
int main ()
{
    int i;  // this is a single line comment
}
```

Der Beginn eines einzeiligen Kommentars muss nicht am Anfang der Zeile stehen, es können beliebige Deklarationen und Anweisungen davor platziert sein. Alle Zeichen auf der rechten Seite von `//` werden vom Compiler ignoriert.

1.2.2. Variablen, Zeigervariablen und Referenzen

In C gibt es mehrere Möglichkeiten, um auf einen im Speicher liegenden Wert zuzugreifen. In einer kurzen Wiederholung gehen wir hier auf die unterschiedlichen Möglichkeiten kurz ein.

1.2.2.1. Variante 1: Zugriff auf ein Datum direkt über eine Variable

Jedes im Speicher befindliche Datum wird klassischerweise bei Verwendung einer Hochsprache wie C durch eine *Variable* angesprochen. Mit einer Variablen sind drei Informationen assoziiert: Ein Name, ein Datentyp und eine Speicheradresse. Zum Beispiel wird durch

```
int n;
```

eine Variable definiert, die den Namen `n`, den Datentyp `int` und eine Adresse im Speicher besitzt, deren genaue Festlegung beim Programmstart oder auch erst zu einem späteren Zeitpunkt erfolgen kann. Nehmen wir an, dass diese Adresse beispielsweise `0x0128ABC0` lautet, dann könnte eine Visualisierung der Variablen `n` wie in Abbildung 1.4 gezeigt aussehen:

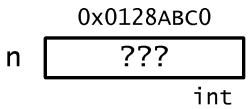

Abbildung 1.4. Visualisierung einer Variablen.

Die Variable selbst wird in Abbildung 1.4 durch ein Rechteck dargestellt. Der Name der Variablen `n` steht auf der linken Seite, die Speicheradresse (hier: `0x0128ABC0`) darüber und der Datentyp `int` darunter. Die Speicheradresse darf nicht darüber hinwegtäuschen, dass Variablen in der Regel mehrere Bytes im Speicher belegen. Für eine `int`-Variable werden (in Abhängigkeit vom Zielsystem) in der Regel 4 Bytes benötigt, die Speicheradresse ist somit die *Anfangsadresse* des Speicherbereichs, in dem der Wert komplett abgelegt ist.

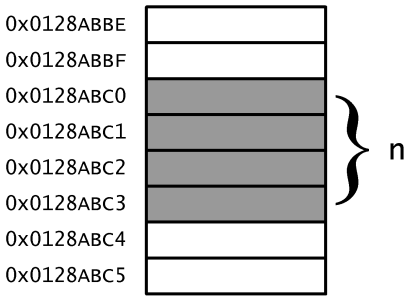

Abbildung 1.5. Ablage einer Variablen im Speicher.

Bekommt die Variable `n` einen Wert zugewiesen, zum Beispiel durch die Wertzuweisung

```
n = 123;
```

so wird dieser Wert in seiner binären Darstellung – typischerweise im Zweierkomplement – in den vier Speicherzellen mit den Adressen `0x0128ABC0`, `0x0128ABC1`, `0x0128ABC2` und `0x0128ABC3` abgelegt, wie wir in Abbildung 1.6 sehen.

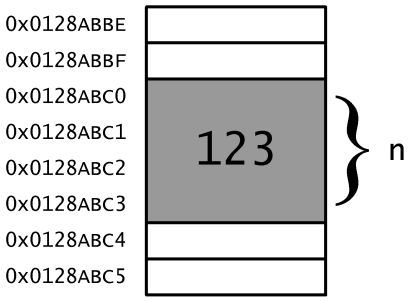

Abbildung 1.6. Ablage einer Variablen im Speicher mit Wert.

Der Einfachheit halber lässt man in den grafischen Darstellungen die Adresse der Variablen (die zum Übersetzungszeitpunkt ohnehin nicht bekannt ist) und den Typ der Variablen weg, so dass eine Variable häufig kompakter wie in Abbildung 1.7 dargestellt wird:

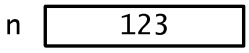

Abbildung 1.7. Vereinfachte Visualisierung einer Variablen mit Wert.

Nach diesen Vorbereitungen betrachten wir in Listing 1.1 ein kleines Beispiel:

```
01: int n;       // Variable vom Typ 'int'
02: int m = 1;   // Variable vom Typ 'int' mit Initialisierung
03: n = m;       // Wertzuweisung des Werts einer Variablen an eine andere
04: m = 2;       // Zuweisung eines Werts an eine Variable
```

Beispiel 1.1. Variablen, Initialisierung und Wertzuweisung.

Zum exakten Verständnis stellen wir den durch diese Anweisungen beschriebenen Sachverhalt in Abbildung 1.8 grafisch dar:

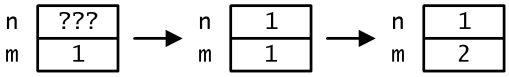

Abbildung 1.8. Zugriff auf ein Datum direkt über eine Variable.

1.2.2.2. Variante 2: Zugriff auf ein Datum über eine Zeigervariable

Da C eine sehr maschinennahe Programmiersprache ist, kann man neben dem Wert einer Variablen auch ihre Adresse zur Laufzeit bestimmen. Zu diesem Zweck gibt es den *Adressoperator* `&`. Der Ausdruck `&n` liefert für die Variable `n` ihre Anfangsadresse im Speicher zurück. Das Codefragment

```
int n;
n = 123;
printf ("%d\n", n);       // value of n
printf ("%08lX\n", &n);   // address of n
```

produziert auf meinem Rechner die Ausgabe

```
123
0012FE00
```

Die Adresse einer Variablen kann wiederum in einer Variablen abgelegt werden. Derartige Variablen rangieren unter der Bezeichnung *Zeigervariable* (engl. *pointer variable*). Formal wird eine Zeigervariable in C als *abgeleiteter Typ* eingeführt. In Bezug auf einen existierenden Typ *T* spricht man vom abgeleiteten Typ „Zeiger auf *T*“ und notiert diesen Datentyp syntaktisch mit *T**:

type pointer_variable;*

Damit können wir einfache Variablen und Zeigervariablen gemeinsam in Listing 1.2 an einem Beispiel studieren:

```
01: int  n;
02: int* pi;
03: n    = 123;   // assign value to variable n
04: pi   = &n;    // assign value to pointer variable pi
```

Beispiel 1.2. Variable versus Zeigervariable.

Der Wert einer Zeigervariablen kann immer nur eine Adresse sein – im letzten Beispiel die Adresse der Variablen `n`. Man sagt auch, dass die Zeigervariable `pi` zu einer anderen Variablen „zeigt“ oder die Variable `n` „referenziert“, wie Abbildung 1.10 verdeutlichen soll. Der `&`-Operator wird deshalb auch „Adress“- oder „Referenz“-Operator genannt.

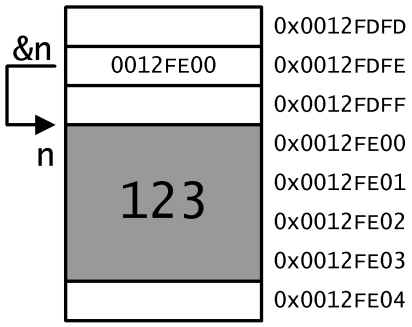

Abbildung 1.9. Ablage einer Zeigervariablen im Speicher.

Wie für Variablen hat sich auch für Zeigervariablen eine standardisierte Visualisierung etabliert (Abbildung 1.10). In den meisten Büchern wird ihr Inhalt durch einen Pfeil symbolisiert, der auf die Variable deutet, dessen Adresse in der Zeigervariablen abgelegt ist. Das heißt insbesondere, dass so gut wie nie der tatsächliche Inhalt der Zeigervariablen (also zum Beispiel die Adresse `0x0012FE00`) dargestellt wird, da ihr Wert in den seltensten Fällen von Bedeutung ist.

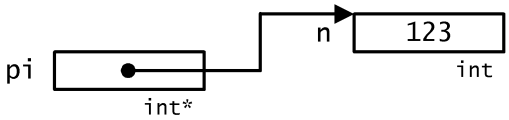

Abbildung 1.10. Visualisierung einer Zeigervariablen.

Mit Zeigervariablen kann man nicht nur auf die Adresse einer Variablen zugreifen, sondern vor allem auf deren Inhalt. Wenn `pi` eine Zeigervariable ist, die auf eine `int`-Variable `n` zeigt, so liefert der Ausdruck `*pi` den Wert von `n` zurück. Dieser Vorgang wird als „eine Zeigervariable dereferenzieren" bezeichnet, oder kurz „Dereferenzierung". Der `*`-Operator heißt entsprechend „Dereferenzierungs"-Operator – sofern er für eine Dereferenzierung verwendet wird und nicht für eine Multiplikation.

Wir führen das Dereferenzieren einer Variablen am Beispiel in Listing 1.3 vor:

```
01: int   n, m;
02: int*  pi;
03: pi    = &n;       // assign address of n to pointer pi
04: *pi   = 17;       // dereference pointer pi - write access
05: m     = *pi;      // dereference pointer pi - read access
```

Beispiel 1.3. Variablen und Dereferenzierung.

Das Beispiel verdeutlicht sehr prägnant, dass der direkte Zugriff auf eine Variable (hier: Variable `n`) ohne explizite Verwendung des Variablenbezeichners möglich ist. Der Ausdruck `*pi` fungiert in diesem Beispiel als Alias für die Variable `n`. Die Anweisungen des letzten Codefragments sind grafisch in Abbildung 1.11 illustriert:

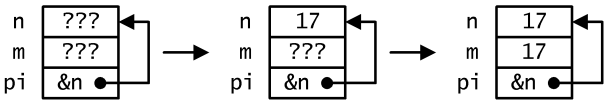

Abbildung 1.11. Zugriff auf ein Datum über eine Zeigervariable.

1.2.2.3. Variante 3: Zugriff auf ein Datum über eine Referenz

In C++ (nicht in C) ist es möglich, ein Datum im Speicher über eine *Referenz* anzusprechen. Eine Referenz ist nichts anderes als ein anderer Name für dieselbe Variable. Syntaktisch wird eine Referenz durch folgende Deklaration definiert:

type& *reference* = *variable*;

In dieser Deklaration wird eine Referenzvariable *reference* erstellt, über die im weiteren Programmablauf der Zugriff auf die Variable *variable* vom Datentyp *type* stattfindet. Ein Beispiel in Listing 1.4 soll dies verdeutlichen:

```
01: int n;
02: int& rn = n;
03: int m = 123;  // m <- 123
04: rn = m;       // n <- 123
```

Beispiel 1.4. Variablen und Referenzen.

Der Sachverhalt aus Listing 1.4 ist in Abbildung 1.12 grafisch dargestellt:

Abbildung 1.12. Zugriff auf ein Datum über eine Referenz.

Achtung

Der Referenz-Operator `&` kann in einem gültigen C++-Programm in zwei Situationen eingesetzt werden. Zum einen, um die Adresse einer Variablen zu ermitteln. In diesem Fall ist der Referenz-Operator *vor* dem entsprechenden Variablennamen zu platzieren. Im zweiten Fall wird der Referenz-Operator bei der Deklaration einer Referenzvariablen benötigt. Hier ist der Operator *nach* dem entsprechenden Datentyp aufzuführen.

1.2.2.3.1. Referenzen in Deklarationen

Beim Umgang mit Referenzen gilt es eine Regel zu beachten: Die Variable, die referenziert werden soll, kann der Referenz ausschließlich bei ihrer Deklaration zugewiesen werden. Jede weitere im Programmablauf folgende Zuweisung an eine Referenz bedeutet eine Zuweisung an die referenzierte Variable. Daraus folgt, dass eine Referenz bei ihrer Deklaration initialisiert werden *muss*, da dies zu einem späteren Zeitpunkt nicht mehr möglich ist:

```
int n;           // variable declaration (type 'int')
int& rn = n;     // variable declaration (type 'reference to int')
int m;           // variable declaration (type 'int')
n = 5;           // asssignment statement
m = 6;           // asssignment statement
rn = m;          // asssignment statement (!)
```

Alle Operationen oder Zugriffe, die auf eine Referenz angewendet werden, beziehen sich nach ihrer Deklaration immer auf die referenzierte Variable, siehe Abbildung 1.13.

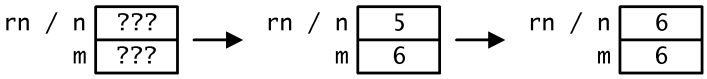

Abbildung 1.13. Deklaration versus Gebrauch einer Referenz.

1.2.2.3.2. Referenzen als Parameter

In C werden Parameter bei der Übergabe an eine Funktion immer auf den Laufzeitstapel kopiert, man spricht deshalb auch von der Parameterübergabe nach dem Prinzip *call-by-value*. Eine Variablenkopie ist aus Sicht der aufgerufenen Funktion immer eine lokale Variable, die übergebene Variable außerhalb der aufgerufenen Funktion kann auf diese Weise niemals verändert werden. Möchte man hingegen auch Variablen verändern, die nicht lokal zur aufgerufenen Funktion sind, muss man ihre Adresse übergeben. Wenngleich in dieser Variante die aufgerufene Funktion die Adresse einer Variablen erhält (und somit in der Lage ist, die Variable außerhalb der aufgerufenen Funktion verändern zu können), spricht man auch hier vom Übergabemechanismus *call-by-value*, da bei genauer Betrachtung eine *Kopie* der Variablenadresse übergeben wird. In C++ kommt nun eine weitere Variante hinzu: Die Übergabe von Parametern nach dem Prinzip *call-by-reference*.

Referenzen eröffnen die Möglichkeit, die Referenz eines Aktualparameters an eine Funktion zu übergeben. Eine Veränderung am Parameter ist nun gleichbedeutend mit einer Veränderung der außerhalb der aufgerufenen Funktion übergebenen Variablen. Ein Beispiel einer Funktion `Incr`, die ihren Aktualparameter um Eins erhöht, verdeutlicht diese Situation:

```
void Incr (int& n)
{
    n ++;
}
```

Durch den Referenzoperator `&` hinter der Typangabe `int` wird der formale Parameter `n` zu einer Referenz der Variablen, die beim Aufruf an `Incr` übergeben wird. Eine Veränderung von `n` ist gleichbedeutend mit einer Veränderung des Aktualparameters selbst:

```
int m = 1;
Incr (m);
printf ("%d\n", m);       // m = 2
```

Intern wird bei der *call-by-reference*-Parameterübergabe die Adresse des aktuellen Parameters übergeben. In der gerufenen Funktion wird bei jedem Zugriff auf den formalen Parameter der übergebene Zeiger dereferenziert, wodurch automatisch auf die übergebene Variable zugegriffen wird. Welchen Vorteil besitzen Referenzen als Parameter? Man erspart sich im Rumpf einer Funktion das Dereferenzieren mit dem `*`-Operator, die Anweisungen werden übersichtlicher und sind weniger fehleranfällig. Betrachten Sie zu diesem Zweck das klassische Beispiel einer Funktion `Swap`, die die Inhalte zweier Variablen vertauscht. In C müsste diese Funktion wie folgt geschrieben werden:

```
void Swap (int* n, int* m)
{
    int tmp = *n;
    *n = *m;
    *m = tmp;
}
```

Beim Aufruf von `Swap` müssen von den Aktualparametern die Adressen übergeben werden, also beispielsweise in der Form

```
int x = 1;
int y = 2;
Swap (&x, &y);
```

In C++ lässt sich diese Funktion unter Verwendung von Referenzen viel einfacher schreiben:

```
void Swap (int& n, int& m)
{
    int tmp = n;
    n = m;
    m = tmp;
}
```

Neben der `Swap`-Funktion selbst wird auch ihr Gebrauch einfacher:

```
int x = 1;
int y = 2;
Swap (x, y);
```

Wir kommen noch einmal auf die vorgestellte Funktion `Incr` zu sprechen. In einer zweiten Variante demonstrieren wir, wie sich Referenzen für die Definition einer Funktion eignen, die sowohl auf der linken als auch auf der rechten Seite einer Wertzuweisung stehen kann.

Zunächst zeigen wir an einer Reihe von Beispielen auf, welche Erwartungen wir an den Aufruf von `Incr` (und einer zweiten Funktion `Decr` zum Dekrementieren einer Variablen) stellen:

```
int x = 1;
int y = 2;

Incr (x);                       // contents of x is incremented
Decr (x);                       // contents of x is decremented

Incr (Incr (x));                // contents of x is incremented by 2 (!)
Decr (Decr (Decr (y)));         // contents of y is decremented by 3 (!)

Incr (Incr (Decr (Decr (x))));  // contents of x remains unchanged

int z = Incr (y);               // contents of y is incremented by 1
```

```
                                    // and assigned afterwards to z
```

Der Trick in der neuen Implementierung von `Incr` und `Decr` besteht darin, dass nach dem Inkrementieren (bzw. Dekrementieren) der spezifizierten Variablen ihre Referenz als Rückgabewert zurückzuliefern ist. Potentielle Aufrufer von `Incr` und `Decr` können so mit derselben Variablen weiterarbeiten (Listing 1.5):

```
01: int& Incr (int& n)
02: {
03:     n ++;
04:     return n;
05: }
06:
07: int& Decr (int& n)
08: {
09:     n --;
10:     return n;
11: }
```

Beispiel 1.5. Zwei Funktionen `Incr` und `Decr` im Stile der Operatoren `++` und `--`.

1.2.3. Der Gültigkeitsbereichsoperator : :

Auf den Hauptanwendungsbereich des Gültigkeitsbereichsoperators können wir erst bei der Diskussion von Namensräumen und Klassen eingehen, auf die wir später zu sprechen kommen. Einen ersten Eindruck von diesem Operator können wir trotzdem jetzt schon vermitteln, wir beschränken uns auf unsere Kenntnisse über globale Variablen und Funktionen in C. Lassen Sie uns einen Blick auf das folgende Codefragment werfen:

```
int x = 1;

void Func ()
{
    int x = 2;

    printf ("x: %d\n", x);
}
```

Welchen Wert wird der Aufruf von `printf` in der Konsole liefern? Es ist natürlich der Wert 2, da die lokale Variablendefinition vorrangig gegenüber global definierten Variablen ist. Um nun dennoch die Möglichkeit des Zugriffs auf die verdeckte, global definierte Variable `x` zu erhalten, kann man den Gültigkeitsbereichsoperator `::` ins Spiel bringen. Mit seiner Hilfe lässt sich die Blockstruktur von C++ durchbrechen und global definierte Variablen werden sichtbar. Die modifizierte `Func`-Funktion

```
int x = 1;

void Func ()
{
    int x = 2;

    printf ("x: %d\n", x);
    printf ("x: %d\n", ::x);
}
```

führt nun zur Ausgabe der Werte

```
x: 2
x: 1
```

1.2.4. Neue Cast-Operatoren

1.2.4.1. `const_cast`

1.2.4.2. `static_cast`

1.2.4.3. `reinterpret_cast`

1.2.4.4. `dynamic_cast`

1.2.5. Datentyp `bool`

Neben den von C her bekannten elementaren Datentypen wie `int`, `double` usw. gibt es in C++ auch noch einen *boolschen* Datentyp, dessen Variablen nur zwei Werte annehmen können: `true` und `false`. Der Name dieses Datentyps lautet sinnigerweise `bool`.

In der Regel kommen `bool`-Variablen zum Einsatz, wenn man das Resultat eines Vergleichs abspeichern möchte:

```
int x = 1;
int y = 2;
...
bool b = (x == y);
```

oder auch noch kürzer:

```
bool b = x == y;
```

Die direkte Wertzuweisung von `true` oder `false` an eine `bool`-Variable ist auch sinnvoll. Sie kommt typischerweise bei der Formulierung von Wiederholungsschleifen zum Einsatz:

```
bool flag = false;
while (flag)
{
    int x, y;
    ...
    if (x == y)
        flag = false;
    ...
}
```

Variablen des Typs `bool` lassen sich nach `int` konvertieren. Dabei wird `false` nach `0` und `true` nach `1` umgewandelt. Das Codefragment

```
bool flag;
flag = true;
printf ("true=%d\n", flag);
flag = false;
printf ("false=%d\n", flag);
```

führt zur Ausgabe

```
true=1
false=0
```

1.2.6. Funktionsprototypen

1.2.7. Default-Werte für Parameter

1.2.8. Dynamische Speicherverwaltung

Nicht immer ist es bei der Entwicklung eines Programms möglich, den benötigten Speicherplatz exakt vorherzuplanen. Eine – unökonomische – Strategie besteht darin, den maximal benötigten Speicher zum Start des Programms (beispielsweise in großen Arrays) zu allokieren. Besser ist es, den tatsächlich notwendigen Speicher erst zur Laufzeit (dynamisch) anzufordern. In C stehen zu diesem Zweck die zwei Funktionen `malloc` und `free` aus der C-Laufzeitbibliothek zur Verfügung. Ihr Pendant in C++ sind die beiden Operatoren `new` und `delete`.

Mit `new` kann man entweder Speicher für eine einzelne Variable oder für ein ganzes Array erzeugen:

```
int* p1 = new int;
int* p2 = new int[5];
```

`p1` ist eine Zeigervariable, die auf eine einzige `int`-Variable zeigt, `p2` wiederum verweist auf ein Array, bestehend aus 5 `int`-Variablen (Abbildung 1.14):

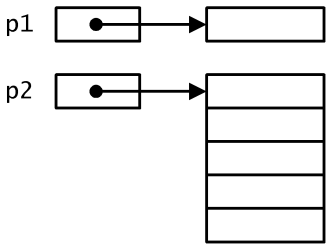

*Abbildung 1.14. Erzeugung einer **`int`**-Variablen mit **`new`** und eines **`int`**-Arrays mit **`new[]`**.*

Nach dem erfolgreichen Aufruf von `new` wird ein Zeiger des angeforderten Typs zurückgegeben. Andernfalls verursacht das C++-Laufzeitsystem das Werfen einer Ausnahme (mehr dazu in Kapitel XXX). Der dynamisch angelegte Speicher kann nun durch die Zeigervariable angesprochen werden:

```
*p1 = 99;
for (int i = 0; i < 5; i ++)
    p2[i] = i;
```

Bei einfachen Variablen ist es möglich, den dynamisch allokierten Speicherbereich zu initialisieren:

```
int* p1 = new int(99);
```

Ein mit `new` angeforderter Speicherbereich unterliegt nicht dem Gültigkeitsbereich der Zeigervariablen, die auf diesen Speicher verweist. Das C++-Laufzeitsystem verwaltet in Zusammenarbeit mit dem Betriebssystem einen großen, zusammenhängenden Speicherbereich namens *Halde* (engl. *Heap*). Dieser Speicherbereich kann ausschließlich durch Anforderungen mit dem `new`-Operator zugeteilt werden und ist mit dem `delete`-Operator wieder freizugeben, so dass er von neuem zur Verfügung steht.

In Entsprechung zu den Speicherplatzanforderungen mit `new` und `new[]` wird für die Speicherplatzfreigabe zwischen den Operatoren `delete` und `delete[]` unterschieden:

```
delete   p1;
delete[] p2;
```

Mit Hilfe des eckigen Klammernpaares `[]` muss der Compiler explizit informiert werden, ob der Zeiger auf eine einzelne Variable oder auf ein Array zeigt.

Im Zusammenspiel von `new` und `delete` gibt es einige Spielregeln zu beachten, die von Seiten des Compilers nicht überprüft werden (sofern dies überhaupt möglich wäre) und deren Nichtbeachtung zur Laufzeit unvorhersehbare Programmabstürze zur Folge haben könnte:

- Der `delete`-Operator darf auf `NULL`-Zeigervariablen angewendet werden. Natürlich bewirkt dieser Aufruf nichts, es ist aber sichergestellt, dass das laufende Programm dadurch nicht beeinträchtigt wird oder gar abstürzt.
- Mehrfache aufeinanderfolgende Aufrufe des `delete`-Operators mit derselben Zeigervariablen sind unzulässig. Es besteht die Gefahr eines Programmabsturzes!
- Der `delete`-Operator darf ausschließlich auf Zeigervariablen angewendet werden, die mit dem `new`-Operator erzeugt worden sind. Es besteht wiederum die Gefahr eines Programmabsturzes:

```
int n;
delete &n;    // Error!

void* ptr = malloc (5);
delete ptr;   // Error!

int* ip = new int(1);
delete ip;    // o.k.
```

Kapitel 2. Klassen und Objekte

2.1. Klassen und Objekte

2.1.1. Einführung und Motivation

2.1.2. Definition von Klasse und Objekt

Im Umfeld der objektorientierten Programmierung versteht man unter einer *Klasse* die Zusammenfassung von Daten und ihren korrespondierenden Funktionen in einem gemeinsamen Sprachkonstrukt. Dieses Konstrukt definiert einen abstrakten Datentyp, die Menge der gekapselten Daten und Funktionen leitet sich aus dem Problemkontext ab.

Wir betrachten als Beispiel die Rationalen Zahlen aus der Mathematik. Jede rationale Zahl besitzt einen Zähler und einen Nenner und eine Vielzahl von Funktionen, die auf diesen beiden ganzzahligen Werten operieren (wie beispielsweise die Addition und Subtraktion oder auch die Ausgabe einer rationalen Zahl in einem Konsolenfenster). Für eine Klasse `Fraction` besteht folglich der Datenteil aus zwei Variablen des Typs `int`. Für die Arithmetik rationaler Zahlen benötigen wir Funktionen wie `Add`, `Sub`, usw., der Datenteil der Klasse geht in die jeweilige Berechnung mit ein. In Listing 2.1 betrachten wir die erste Version einer Klassendeklaration in C++:

```
01: //
02: // class Fraction
03: //     Rational number data abstraction
04: //
05:
06: class Fraction
07: {
08: public:
09:     int     m_num;    // numerator
10:     int     m_denom; // denominator
11:
12:     // arithmetic methods
13:     void    Add (int, int);
14:     void    Sub (int, int);
15:
16:     // console output
17:     void    Print ();
18: };
```

Beispiel 2.1. Deklaration der Klasse `Fraction`.

Wir finden in Listing 2.1 wesentliche Merkmale zum Deklarieren einer Klasse `Fraction` vor wie beispielsweise zwei Funktionen `Add` und `Sub` zum Addieren und Subtrahieren rationaler Zahlen. Die beiden Funktionsprototypen spezifizieren – ganz in C-Manier – für ihre formalen Parameter nur den Datentyp (hier: `int`), es könnten auch zusätzlich Namen für die Formalparameter angegeben werden. In manchen Situationen kann dies die Lesbarkeit des Quellcodes durchaus erhöhen:

```
// arithmetic methods
void Add (int num, int denom);
void Sub (int num, int denom);
```

Tipp

Im Gegensatz zur Deklaration von Klassen in Java und C# weist die korrespondierende Syntax in C++ einen kleinen, subtilen Unterschied auf: Mit den geschweiften Klammern `{` und `}` gebildete Blöcke müssen zwingend mit einem Semikolon ‘;’ abgeschlossen werden. C++-Compiler sind an dieser Stelle sehr empfindlich. Wenn Sie das Semikolon vergessen, sind die daraus resultierenden Fehlermeldungen häufig sehr missverständlich und werden auch erst einige Zeilen später im Quellcode angezeigt.

Das Schlüsselwort `public` in Zeile 3 von Listing 2.1 gibt an, für wen die nachfolgenden Deklarationen sichtbar sind. Im Kapitel XXX gehen wir auf diesen Aspekt näher ein. In Listing 2.2 stellen wir eine Implementierung der `Fraction`-Klasse vor:

```
01: #include <stdio.h>
02: #include "Fraction.h"
03:
04: // arithmetic methods
05: void Fraction::Add (int num, int denom)
06: {
07:     m_num = m_num * denom + m_denom * num;
08:     m_denom = m_denom * denom;
09: }
10:
11: void Fraction::Sub (int num, int denom)
12: {
13:     m_num = m_num * denom - m_denom * num;
14:     m_denom = m_denom * denom;
15: }
16:
17: // console output
18: void Fraction::Print ()
19: {
20:     printf ("%d/%d\n", m_num, m_denom);
21: }
```

Beispiel 2.2. Implementierung der Klasse `Fraction`.

In Listing 2.2 gibt es einige Merkmale zu beachten:

- Kopplung von Funktion und definierender Klasse:

  Um für die `Add`-Funktion in Zeile 5 die Zugehörigkeit zur Klasse `Fraction` zum Ausdruck zu bringen, ist dem Funktionsnamen der Name der korrespondierenden Klasse voranzustellen. Die Klasse bildet einen Gültigkeitsbereich für ihre Elemente, man trennt deshalb den Klassen- und Funktionsnamen mit dem *Gültigkeitsbereichsoperator* '`::`' (engl. *scope operator*). Der Name einer Funktion wird mit Hilfe des Gültigkeitsbereichsoperators im Namensraum der Klasse angesiedelt, die Funktion ist *innerhalb* ihrer umgebenden Klasse definiert. Auf diese Weise ist es kein Problem – und häufig auch sinnvoll –, Funktionen mit demselben Namen in unterschiedlichen Klassen zu definieren.

- Kopplung von Funktion und Daten:

  In der Implementierung der `Add`-Funktion kommen neben den formalen Parametern `num` und `denom` zwei weitere Variablen `m_num` und `m_denom` zum Zuge. Es wird deutlich, dass neben den Formalparametern vor allem der Datenteil der korrespondierenden Klasse für die Ausführung der Operation zur Verfügung steht.

Jedes konkrete Exemplar einer Klasse wird als *Instanz* oder *Objekt* bezeichnet. Jede Instanz zieht eine konkrete Ausprägung des jeweiligen Datenbereichs im Speicher nach sich. Für Klassen wird im Gegensatz zu den Instanzen kein Speicher bereitgestellt, sie stellen ausschließlich eine Bauanleitung (Schablone) für ihre konkreten Ausprägungen dar. Objekte eines Klassentyps werden in C++ wie Variablen eines elementaren Datentyps in C angelegt:

```
// two rational numbers 'f1' and 'f2'
Fraction f1, f2;
```

Damit formulieren wir in Listing 2.3 eine erste Anwendung auf Basis des benutzerdefinierten Klassentyps `Fraction` :

```
01: #include "Fraction.h"
02:
03: // test frame
04: void main ()
05: {
06:     Fraction f, g;
07:     f.m_num = 1;
08:     f.m_denom = 2;
09:     g.m_num = 3;
10:     g.m_denom = 4;
11:
12:     f.Add(3, 7);
13:     f.Print();
14:     g.Print();
15: }
```

Beispiel 2.3. Anwendungsprogramm zur Klasse `Fraction`.

Die Ausführung des Programms führt zur Ausgabe von

```
13/14
3/4
```

auf der Konsole.

Beachten Sie bei der Implementierung der zwei Methoden `Add` und `Sub` in Listing 2.2: Eine Addition zweier rationaler Zahlen setzt – mathematisch betrachtet – einen linken und einen rechten Operanden voraus. In den Methodensignaturen von `Add` und `Sub` finden Sie jeweils nur einen Operanden vor, es handelt sich bei dieser Sprechweise um den *rechten* Operanden. Wo ist der linke Operand? Er wird stets durch ein Objekt der Klasse `Fraction` repräsentiert! In der Anweisung

```
f.Add(3, 7);
```

finden wir somit – in Bezug auf die binäre '+'-Operation rationaler Zahlen – den linken Operanden im Objekt `f` vor, der rechte Operand wird durch den Aktualparameter des `Add`-Methodenaufrufs spezifiziert.

Es folgen einige Hinweise zur Nomenklatur: In der objektorientierten Programmierung rangieren die einzelnen Datenelemente einer Klasse unter dem Begriff *Instanzvariable*. Alternativ haben sich auch die Begriffe *Attribut*, *Feld* oder speziell in C++ *Membervariable* eingebürgert. In Listing 2.1 finden wir in den Zeilen 9 und 10 die Deklaration zweier Instanzvariablen `m_num` und `m_denom` vor. In Zeile 6 von Listing 2.3 werden zwei Instanzen (Objekte) der Klasse `Fraction` mit den Namen `f` und `g` erzeugt (Abbildung 2.1). Jedes Objekt reserviert Speicher für seine Instanzvariablen `m_num` und `m_denom`. Der Zugriff auf eine Instanzvariable erfolgt mit dem Punktoperator '.' wie der Zugriff auf eine Strukturkomponente in Analogie zu C.

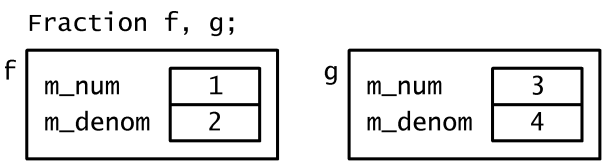

Abbildung 2.1. Ablage zweier `Fraction`-Instanzen im Speicher.

Der Begriff der *Funktion* wird in C++ durch den Terminus *Methode* oder – in konsequenter Fortführung des *Instanzvariablen*-Begriffs durch – *Instanzmethode* ersetzt. Eine Methode kann nur an einem Objekt aufgerufen werden. Vor dem Aufruf einer Methode muss zunächst eine Instanz der jeweiligen Klasse erzeugt werden. In den Zeilen 12, 13 und 14 finden wir Aufrufe der Methoden `Add` und `Print` an den Objekten `f` und `g` vor. Wie bei den Instanzvariablen erfolgt auch der Aufruf von Methoden mit dem Punktoperator.

Tipp

Die Regelung „eine Instanzmethode ist immer an einem Objekt aufzurufen“ besitzt eine Ausnahme: Ruft eine Instanzmethode eine andere Instanzmethode *derselben* Klasse auf, muss kein Objektbezeichner vor dem Methodenaufruf stehen. Es ist aus dem Kontext zweifelsfrei erkennbar, dass sich der Methodenaufruf auf das *aktuelle* Objekt bezieht. Wollten wir in der `Add`-Methode beispielsweise zu Testzwecken die `Print`-Methode aufrufen, könnte man dies so formulieren:

```
void Fraction::Add (int num, int denom)
{
    m_num = m_num * denom + m_denom * num;
    m_denom = m_denom * denom;
    Print ();   // o.k.
}
```

Objekte besitzen viele Gemeinsamkeiten mit Strukturvariablen in C. Sie können auf der linken und rechten Seite einer Wertzuweisung auftreten. In diesem Fall werden die Instanzvariablen einfach Byte für Byte zwischen den beteiligten Speicherbereichen umkopiert. Auch sind Objekte als Aktualparameter bei Methodenaufrufen zulässig bzw. können das Resultat eines Methodenaufrufs sein. Es wird – wiederum in Analogie zu den Strukturvariablen – eine Kopie des Objekts auf dem Laufzeitstapel abgelegt, die gerufene Methode kann das Originalobjekt nicht beeinflussen. Wir demonstrieren diese Aussagen an einer zweiten Version der Klasse `Fraction`, in Listing 2.4 beginnen wir mit der überarbeiteten Klassendeklaration:

```
01: class Fraction
02: {
03: public:
04:     // data area
05:     int         m_num;   // numerator
06:     int         m_denom; // denominator
07:
08:     // arithmetic methods
09:     void        Add      (Fraction);
10:     void        Sub      (Fraction);
11:
12:     // console output
13:     void        Print    ();
14: };
```

Beispiel 2.4. Alternative Deklaration der Klasse `Fraction` (Version 2).

In den Zeilen 9 und 10 können Sie erkennen, dass die zwei Methoden `Add` und `Sub` ihren Operanden in Gestalt eines `Fraction`-Objekts entgegennehmen. Die Anpassungen der `Add`- und `Sub`-Methode in der Implementierung finden Sie in Listing 2.5 vor:

```
01: void Fraction::Add (Fraction f)
02: {
03:     m_num = m_num * f.m_denom + m_denom * f.m_num;
04:     m_denom = m_denom * f.m_denom;
05: }
06:
07: void Fraction::Sub (Fraction f)
08: {
09:     m_num = m_num * f.m_denom - m_denom * f.m_num;
10:     m_denom = m_denom * f.m_denom;
11: }
```

Beispiel 2.5. Ausschnittsweise Implementierung der Klasse `Fraction` (Version 2).

Zum Testen legen wir in Listing 2.6 einen modifizierten Testrahmen zu Grunde.

```
01: void main ()
02: {
03:     Fraction f;
04:     f.m_num = 1;
05:     f.m_denom = 2;
06:     f.Print();
07:
08:     Fraction g;
09:     g.m_num = 3;
10:     g.m_denom = 2;
11:     f.Add(g);
12:     f.Print();
13:
14:     f = g;
15:     f.Print();
16: }
```

Beispiel 2.6. Anwendungsprogramm zur Klasse `Fraction` (Version 2).

Das Programm aus Listing 2.6 produziert das Resultat

```
1/2
8/4
3/2
```

Die Übergabe eines `Fraction`-Objekts im Ganzen entspricht jetzt weitaus mehr dem objektorientierten Charakter eines Methodenaufrufs in C++ als die Übergabe einzelner Objektbestandteile wie Zähler und Nenner. Trotzdem ist bei der Definition der Methodenschnittstelle Vorsicht angesagt. Die in Listing 2.5 praktizierte Übergabe der Aktualparameter basiert auf der *call-by-value*-Strategie. Dies bedeutet, dass für das zu übergebende Objekt exklusiv für jeden einzelnen Methodenaufruf eine vollständige Kopie auf dem Laufzeitstapel angelegt wird, die nach dem Aufruf überflüssig ist. In einer größeren Anwendung mit vielen Objekten und vielen Methodenaufrufen kommt es in dieser Situation zu Performanceproblemen. Eine Abhilfe schaffen wir mit dem bereits vorgestellten Sprachmittel der Referenz und der damit verbundenen Parameterübergabestrategie *call-by-reference*. Anstatt ein Objekt durch *call-by-value* für den Aufruf einer Methode zu kopieren, genügt es, den Zugang zum Objekt durch seine Referenz bereitzustellen. In diesem Fall wird keine zeitaufwändige Objektkopie erstellt, sondern einfach auf dem Laufzeitstapel nur die Referenz – sprich: die Adresse des Objekts – des Aktualparameters abgelegt. In Listing 2.7 finden wir die geringfügigen Anpassungen der `Fraction`-Klasse bei Deklaration und Implementierung gegenüber Listing 2.4 und Listing 2.5 vor:

```
01: class Fraction
02: {
03: public:
04:     // arithmetic methods
05:     void Add (Fraction&);
06:     void Sub (Fraction&);
07:     ...
08: };
09:
10: void Fraction::Add (Fraction& f)
11: {
12:     m_num = m_num * f.m_denom + m_denom * f.m_num;
13:     m_denom = m_denom * f.m_denom;
14: }
15:
16: void Fraction::Sub (Fraction& f)
17: {
18:     m_num = m_num * f.m_denom - m_denom * f.m_num;
19:     m_denom = m_denom * f.m_denom;
20: }
```

Beispiel 2.7. Klasse `Fraction`: Verwendung von Referenzen in der Parameterdeklaration (Version 3).

Im Gebrauch der Klasse `Fraction`-Klasse ändert sich nichts, das Anwendungsprogramm aus Listing 2.6 läuft unverändert. Wenngleich die Performance eines Anwendungsprogramms mit Methodenaufrufen auf *call-by-reference*-Basis wesentlich schneller ist, müssen wir noch eine kleine Sicherheitslücke schließen. Im Gegensatz zur *call-by-value*-Strategie ist es nun im Rumpf einer Methode mit Parametern vom Typ Referenz möglich, ein aktuelles Objekt bei der Parameterübergabe *direkt* anzusprechen. Der lesende Zugriff ist dabei weniger kritisch, es ist vielmehr auch ein schreibender Zugriff auf das Aktualobjekt möglich – es handelt sich eben nicht mehr um eine Kopie des betroffenen Objekts. Um den nicht erwünschten, schreibenden Zugriff im Kontext einer Methode auszuschließen, gibt es das Schlüsselwort `const`. Wir können auf diese Weise den effizienten Parameterübergabemechanismus der *call-by-reference*-Strategie so gestalten, dass die Integrität des aktuellen Objekts gewährleistet bleibt (Listing 2.8):

```
01: class Fraction
02: {
03: public:
04:     // arithmetic methods
05:     void Add (const Fraction&);
06:     void Sub (const Fraction&);
07:     ...
08: };
09:
10: void Fraction::Add (const Fraction& f)
11: {
12:     m_num = m_num * f.m_denom + m_denom * f.m_num;
13:     m_denom = m_denom * f.m_denom;
14: }
15:
16: void Fraction::Sub (const Fraction& f)
17: {
18:     m_num = m_num * f.m_denom - m_denom * f.m_num;
19:     m_denom = m_denom * f.m_denom;
20: }
```

Beispiel 2.8. Klasse `Fraction`: Konstante Referenzen zum Schreibschutz.

Die Auswirkungen in der Verwendung von `const` können wir am folgenden Codefragment studieren. Eine offensichtlich fehlerhafte Implementierung der Methode `Add`

```
void Fraction::Add (const Fraction& f)
{
    f.m_num = 1;
}
```

wird bereits zur Übersetzungszeit vom Übersetzer mit der Fehlermeldung *„l-value specifies const object“* abgewiesen.

2.1.3. Programmstruktur: Schnittstelle und Implementierung

Im letzten Abschnitt haben wir den gesamten Quellcode zur Entwicklung einer Klasse vorgestellt. In der Praxis – und vor allem in größeren Softwareprojekten – gibt es einige Regeln, wie der Quellcode geeignet auf mehrere Dateien zu verteilen ist. Dazu führen wir die Begriffe *Schnittstelle* und *Implementierung* ein. Unter der *Schnittstelle* einer Klasse verstehen wir alle Aspekte, *welche* Elemente eine Klasse für ihre Benutzung zur Verfügung stellt. Neben dem Namen der Klasse zählen dazu die Funktionsprototypen der Methoden und die Instanzvariablen, auf die ein Anwender zugreifen kann. Auch sind pro Methode alle Informationen relevant, die man zu ihrem Aufruf braucht. Beispielsweise die Anzahl der Parameter oder pro Parameter der Datentyp, so dass man für einen Aufruf die passenden Aktualparameter bereitstellen kann.

Bei der Frage, *wie* die Innereien einer Klasse konkret aussehen, kommt man auf den Begriff der *Implementierung* zu sprechen. Hierzu zählt die Realisierung der einzelnen Methoden der Klasse wie auch das Vorhandensein zusätzlicher Instanzvariablen, die nicht-öffentlich und damit für den Anwender nicht sichtbar sind – und deshalb in der *Schnittstelle* einer Klasse auch nicht erwähnt werden.

Im Regelfall wird der Quellcode einer Klasse in zwei Dateien abgelegt:

- *Header-Datei*:

  Die Header-Datei verwaltet die Klassendeklaration einer Klasse an einer zentralen Stelle. In der Header-Datei sollte immer der allgemein zugängliche Teil einer Klasse, also ihre Schnittstelle stehen. Header-Dateien besitzen im Regelfall die Endung *.h*, in anderen Entwicklungsumgebungen finden wir auch die Endungen *.hpp* oder *.hxx* vor.

- *Implementierungs-Datei*:

  In der Implementierungs-Datei (Endung *.cpp*, oder auch *.cxx* bzw. *.cc*) wird der implementierungsabhängige Quellcode einer Klasse abgelegt, vorzugsweise die Realisierung der Instanzmethoden. Auf diese Weise sind Schnittstelle und Implementierung einer Klasse in getrennten Quelldateien abgelegt und es wird dem Grundsatz der modularen Gestaltung des Quellcodes Rechnung getragen. Die Signatur einer implementierten Methode muss *identisch* mit der entsprechenden Signatur in der Klassendeklaration sein.

Legt man diese Aufteilung zu Grunde, kann eine Anwendung, die rationale Zahlen benutzt, mit Hilfe einer *#include*-Direktive auf die Header-Datei der Klasse `Fraction` zugreifen. Die Schnittstelle der Klasse ist auf diese Weise offen gelegt, ohne dass die Anwendung etwas von der Implementierung der Klasse wissen muss. Selbstverständlich muss beim Binden der Anwendung der Implementierungscode der `Fraction`-Klasse verfügbar sein. Dies muss datentechnisch aber nicht in Gestalt des Quellcodes erfolgen, es genügt eine Bibliotheksdatei, die die Implementierung der `Fraction`-Klasse in übersetzter Form enthält.

Um auf die Codefragmente aus dem letzten Abschnitt zu sprechen zu kommen: Die Deklaration der Klasse `Fraction` (Listing 2.1) sollte in einer Datei *Fraction.h* abgelegt sein, der Klassenname steht Pate für den Dateinamen. Für die Klassenimplementierung aus Listing 2.2 sollte der Dateiname *Fraction.cpp* lauten. Die Anwendung selbst sollte nicht in der Implementierungs-Datei einer Klasse programmiert werden, in diesem Buch wird in den meisten Fällen eine Datei *Demo.cpp* verwendet.

Tipp

Bei genauer Betrachtung der Header-Datei der Klasse `Fraction` können wir erkennen, dass diese nicht nur Informationen mit deklarierendem Charakter (wie etwa die Funktionsprototypen zweier Funktionen `Add` und `Sub`) enthält, sondern auch Details der Implementierung preis gibt. In den Zeilen 9 und 10 von Listing 2.1 wird beispielsweise festgelegt, dass Zähler und Nenner in Variablen des Typs `int` abgelegt sind. Wir sehen, dass der gute Vorsatz der Trennung zwischen der Schnittstelle und der Implementierung einer Klasse nicht konsequent bis in alle Details verfolgt wird. Neben der grauen Theorie stehen Entwickler auch in der Pflicht, pragmatische Aspekte wie beispielsweise eine sinnvolle Effizienz in das Design einer Programmiersprache mit einfließen zu lassen.

Die Ablage von Klassenschnittstelle und -implementierung in zwei getrennten Dateien führt zu einem weiteren entscheidenden Vorteil: Zu Beginn der Entwicklungsphase eines größeren Softwareprojekts steht für die beteiligten Entwickler der Entwurf der Datenschnittstellen im Vordergrund. Die Kapselung der *Implementierung* in separaten Dateien gestattet nachträgliche Änderungen in der Realisierung, ohne dass die Benutzer einer Klasse ihren Quellcode ändern müssen. Sollten sich Probleme in der *Schnittstelle* einer Klasse herausstellen, müssen sich natürlich alle Beteiligten

(Klassenentwickler und -anwender) an einen Tisch setzen, um an der Behebung der (ohnehin unvermeidlichen) Entwurfsfehler mitzuwirken.

Speziell bei Klassen mit sehr kleinen Methoden (oder der Einfachheit halber auch in einem Lehrbuch wie diesem ...) gibt es die Möglichkeit, den Rumpf einer Methode komplett innerhalb der Kassendeklaration auszuprogrammieren. Bei diesem Programmierstil entfällt der Vorteil der Geheimhaltung der Implementierung. Bei sehr einfachen Methoden fällt dieser Aspekt nicht sonderlich ins Gewicht, da kurze Methoden in der Regel eine Trivialimplementierung besitzen. In Listing 2.9 finden Sie die Header-Datei der Klasse `Fraction` mit drei ausprogrammierten Methoden vor:

```
01: class Fraction
02: {
03: public:
04:     // data area
05:     int m_num;   // numerator
06:     int m_denom; // denominator
07:
08:     // arithmetic methods
09:     void Add (int num, int denom)
10:     {
11:         m_num = m_num * denom + m_denom * num;
12:         m_denom = m_denom * denom;
13:     }
14:
15:     void Sub (int num, int denom)
16:     {
17:         m_num = m_num * denom - m_denom * num;
18:         m_denom = m_denom * denom;
19:     }
20:
21:     // console output
22:     void Fraction::Print () const
23:     {
24:         printf ("%d/%d\n", m_num, m_denom);
25:     }
26: };
```

Beispiel 2.9. Header-Datei der `Fraction`-Klasse mit ausprogrammierten Methoden.

Der in Listing 2.9 gezeigte Programmierstil sollte vorzugsweise bei Methoden erfolgen, die sehr kurz oder sehr zeitkritisch sind, da in diesem Fall bei der Codegenerierung an den Aufrufstellen der Code der Methode direkt eingesetzt wird. Auf diese Weise spart man sich den Overhead, der auf Maschinencodeebene beim Aufruf eines Unterprogramms entsteht (Ablage der Aktualparameter auf dem Laufzeitstapel, zusätzliche Sprung- und Rücksprungbefehle, usw.). Nachteilig an dieser Strategie ist natürlich, dass an *jeder* Stelle im Programm, wo eine solche Methode aufgerufen wird, der komplette Code der Methode eingesetzt wird. Der Maschinencode des Programms kann auf diese Weise erheblich länger werden.

Die soeben betrachtete Effizienzsteigerung kann auch unter Verwendung des `inline`-Schlüsselworts erzielt werden. Für Methoden, die in der Klassendeklaration mit `inline` gekennzeichnet werden, kann die Implementierung auch außerhalb der Klassendeklaration – aber wiederum nur innerhalb der aktuellen Header-Datei – stattfinden, da der C++-Übersetzer bei jedem Aufruf dieser Methode ihren Code kennen muss, um ihn an der Aufrufstelle einsetzen zu können. Damit formulieren wir in Listing 2.10 die letzte Alternative für eine optimierte Version der Klasse `Fraction`:

```
01: class Fraction
02: {
03: public:
04:     // data area
05:     int          m_num;   // numerator
06:     int          m_denom; // denominator
07:
08:     // arithmetic methods
09:     inline void  Add   (int, int);
10:     inline void  Sub   (int, int);
```

```
11:
12:     // console output
13:     inline void  Print () const;
14: };
15:
16: // (explicit) implementation of 'inline'-methods
17: inline void Fraction::Add (int num, int denom)
18: {
19:     m_num = m_num * denom + m_denom * num;
20:     m_denom = m_denom * denom;
21: }
22:
23: inline void Fraction::Sub (int num, int denom)
24: {
25:     m_num = m_num * denom - m_denom * num;
26:     m_denom = m_denom * denom;
27: }
28:
29: inline void Fraction::Print () const
30: {
31:     printf ("%d/%d\n", m_num, m_denom);
32: }
```

Beispiel 2.10. Header-Datei der `Fraction`-Klasse mit `inline`-Methoden.

Abschließend lässt sich feststellen, dass der vorgestellte Mechanismus zur Performancesteigerung von Methodenaufrufen entweder explizit mit dem `inline`-Schlüsselwort oder implizit mit direkt innerhalb der Klassendeklaration implementierten Methoden in Kraft treten „könnte“. Die letzte Aussage im Konjunktiv hat einen Grund. Ein C++-Compiler lässt sich bei der Codegenerierung nur zu einer Absichtserklärung überreden, er darf diese Regelung auch missachten! Der Grund liegt einfach darin, dass ein Übersetzer die Hoheit über alle Entscheidungen behalten möchte, die nach seiner Meinung zu einer sinnvollen Codegenerierung beitragen!

2.1.4. Konstante Objekte und Methoden

Objekte können wie einfache Variable in C als *konstant* deklariert werden. Soll eine Methode lesenden, aber keinen schreibenden Zugriff auf ihre Instanzvariablen haben, muss diese mit dem Schlüsselwort `const` deklariert werden.

In unserem Beispiel bietet sich die `Print`-Methode an, konstant deklariert zu werden. Im Zuge der Ausgabe ihrer Instanzvariablen greift sie lesend, aber nicht schreibend auf ihre Instanzvariablen zu. Am Ende der Methodendeklaration ist zu diesem Zweck das `const`-Schlüsselwort einzufügen:

```
class Fraction
{
    ...
    void  Print () const;
};
```

Die Implementierung der `Print`-Methode muss das `const`-Schlüsselwort ebenfalls miteinbeziehen:

```
// console output
void Fraction::Print () const
{
    printf ("%d/%d\n", m_num, m_denom);
}
```

Konstante Methoden besitzen das Privileg, an konstanten (und an nicht konstanten) Objekten aufgerufen werden zu dürfen:

```
const Fraction g;
g.Print ();
```

Eine konstante Methode besitzt den Vorteil, dass der Compiler zur Übersetzungszeit schreibende Zugriffe auf Instanzvariablen aufspüren kann. Eine – offensichtlich sehr – fehlerhafte Variante der `Print`-Methode in der Gestalt

```
// console output
void Fraction::Print () const
{
    m_num = 1;
}
```

wird vom Übersetzer mit der Fehlermeldung „*l-value specifies const object*“ abgewiesen.

Achtung

Die Kennzeichnung einer Methode mit dem `const`-Schlüsselwort kann auch zu Hindernissen führen. Möchte man beispielsweise ein Objekt zu Performance-Messungen mit einem Besuchszähler ausstatten (Instanzvariable `m_accessCount`), die pro Methodenaufruf an dem Objekt inkrementiert wird, stößt man auf das Problem, dass der Übersetzer diesen schreibenden Zugriff in allen mit `const` deklarierten Methoden ablehnt. Auf der anderen Seite ist es aber nachvollziehbar, dass ein Besuchszähler am eigentlichen Zustand des Objekts – wie etwa dem Zähler und Nenner einer rationalen Zahl – keine Änderung vornimmt. Aus diesem Grund kann man mit dem Schlüsselwort `mutable` selektiv den schreibenden Zugriff auf eine Instanzvariable in einer `const`-Methode herstellen.

Die Deklaration der Klasse `Fraction` würde mit einem Besuchszähler, der die `Print`-Methodenaufrufe zählt, so aussehen:

```
class Fraction
{
public:
    // data area
    int          m_num;          // numerator
    int          m_denom;        // denominator
    mutable int m_accessCount;  // access counter

    // c'tor
    Fraction ()
    {
        m_num = 0;
        m_denom = 1;
        m_accessCount = 0;
    }

    // console output
    void Fraction::Print () const
    {
        m_accessCount ++;
        printf ("%d/%d\n", m_num, m_denom);
    }
};
```

2.2. Initialisierung von Objekten

Um die Integrität eines Objekts für seine gesamte Lebensdauer zu gewährleisten, sind Instanzvariablen eines Objekts bei seiner Erzeugung sinnvoll zu initialisieren. Eine Möglichkeit haben wir bereits in Listing 2.3 kennen gelernt, die direkte Zuweisung von Werten an die Instanzvariablen eines Objekts unmittelbar *nach* seiner Erzeugung:

```
Fraction f;
f.m_num = 1;
f.m_denom = 7;
```

Diese Art der Initialisierung ist problematisch, da man nicht gezwungen ist, die Initialisierung des Objekts auch tatsächlich nach seiner Erzeugung vorzunehmen. Es könnten sich andere Anweisungen zwischen der Objekterzeugung und der Initialisierung einschleichen, das Objekt würde sich für diese Zeitdauer in einem undefinierten Zustand befinden.

Eine zweite Möglichkeit besteht in der Ergänzung der Klasse `Fraction` um eine spezielle Initialisierungsmethode, beispielsweise mit dem Namen `Init`:

```
class Fraction
{
public:
    int  m_num;   // numerator
    int  m_denom; // denominator

    // initialization method
    void Init (int num, int denom) { m_num = num; m_denom = denom; }
};
```

Der Anwender müsste dann – wiederum *nach* der Objekterzeugung – die `Init`-Methode explizit aufrufen:

```
void main ()
{
    // create instance
    Fraction f;

    // initialize instance
    f.Init (1, 7);
}
```

Auch in dieser Variante besteht noch immer das Problem, dass zwischen der Objekterzeugung und -initialisierung eine potentielle Lücke eintreten kann, in der das Objekt nicht korrekt initialisiert ist. Außerdem müsste man klären, ob der Methodenname `Init` in C++ als reservierter Bezeichner zu sehen ist – eine bei Methoden recht unübliche Vorgehensweise (mit Ausnahme der `main`-Methode, dem Einsprungpunkt einer C++-Anwendung). Die zuvor geschilderten Probleme der Objektinitialisierung sind den Entwicklern objektorientierter Programmiersprachen natürlich nicht verborgen geblieben, aus diesem Grund hat man den Mechanismus der *Konstruktoren* eingeführt.

2.2.1. Konstruktoren

Unter einem *Konstruktor* versteht man eine spezielle Art von Instanzmethode, die den gleichen Namen wie die Klasse hat, zu der sie gehört. Ein Konstruktor besitzt die Aufgabe, die Instanzvariablen eines Objekts vollständig mit sinnvollen Werten zu initialisieren. Umgangssprachlich kann man auch sagen, dass ein Konstruktor eine Instanz „konstruiert“. Die Anweisung zum Aufruf eines Konstruktors wird durch den Compiler im Zuge der Objekterzeugung *automatisch* abgesetzt. Auf diese Weise wird erreicht, dass Objekterzeugung und Konstruktorausführung zu einer Einheit verschmelzen.

Eine Klasse kann durchaus mehrere Konstruktoren besitzen. Wir stoßen wieder auf den Mechanismus des Überladens von Methoden, dieses Mal speziell im Umfeld von Konstruktoren. Aus diesem Grund ergänzen wir unsere Klasse `Fraction` in Listing 2.11 gleich um zwei Konstruktoren:

```
01: class Fraction
02: {
03: public:
04:     // constructors
05:     Fraction ();
06:     Fraction (int num, int denom);
07:
08:     ...
09: };
```

Beispiel 2.11. Klasse `Fraction` mit zwei Konstruktoren (Prototypen).

Die erste Version eines Konstruktors in Zeile 5 von Listing 2.11 besitzt keine Parameter. Seine Aufgabe ist es, den Instanzvariablen des Objekts sinnvolle Standardwerte zuzuordnen. Im Falle eines `Fraction`-Objekts bieten sich die Werte `0` für `m_num` und `1` für `m_denom` an. Der zweite Konstruktor (Zeile 6) initialisiert den Zähler und Nenner mit Werten, die der Anwender bei der Objektkonstruktion durch die Aktualparameter beliebig vorgeben kann. Wir bringen die Implementierung der Klasse `Fraction` in Listing 2.12 auf den neuesten Stand:

```
01: // constructors
02: Fraction::Fraction ()
03: {
04:     m_num = 0;
05:     m_denom = 1;
06: }
07:
08: Fraction::Fraction (int num, int denom)
09: {
10:     m_num = num;
11:     m_denom = (denom == 0) ? 1 : denom;
12: }
```

Beispiel 2.12. Implementierung der Konstruktoren der Klasse `Fraction`.

Mit der kleinen Sicherheitsüberprüfung in Zeile 11 von Listing 2.12 weisen wir exemplarisch darauf hin, dass die Konstruktion eines Objekts dafür verantwortlich ist, die Integrität des Objektzustands vom ersten Augenblick an zu gewährleisten.

Beachten Sie ferner in der Deklaration und Implementierung von Konstruktoren: Sie dürfen per Definition keinen Rückgabetyp besitzen – auch nicht den Datentyp `void`. In der Implementierung eines Konstruktors kann eine `return`-Anweisung, sofern sie überhaupt erforderlich ist, nur parameterlos auftreten.

Der Aufruf eines Konstruktors erfolgt bei der Erzeugung eines Objekts, der Entwickler muss den Aufruf nicht programmieren! Die aktuellen Parameter sind wie bei einem „normalen“ Methodenaufruf in runden Klammern hinter dem Namen der zu erzeugenden Instanz aufzuführen, zum Beispiel:

```
void main ()
{
    Fraction f1;         // c'tor Fraction() used
    Fraction f3(1, 2);   // c'tor Fraction(int, int) used
}
```

Achtung

Das letzte Codefragment umgeht eine beliebte Stolperfalle beim Gebrauch des parameterlosen Konstruktors. In Analogie zur Anwendung von Konstruktoren mit Parametern würde es sich anbieten, ein leeres Paar runder Klammern zu verwenden, um bei einer Objekterzeugung den Gebrauch des parameterlosen Konstruktors besonders hervorzuheben:

```
Fraction f();   // WARNING: Function Prototype !!!
```

Dies ist leider falsch: Die letzte Anweisung wird vom Übersetzer als Funktionsprototyp interpretiert, um eine Funktion `f` zu deklarieren, die keine Parameter besitzt und einen Wert vom Typ `Fraction` zurückliefert. Die korrekte Anwendung des parameterlosen Konstruktors erfolgt ohne Verwendung von runden Klammern!

2.2.2. Der Standardkonstruktor

Der parameterlose Konstruktor wird auch Standardkonstruktor (engl. *default constructor*) genannt. Ist `Type` ein Klassenbezeichner, dann lautet die Deklaration des Standardkonstruktors allgemein formuliert so:

```
class Type
{
public:
    Type ();  // default c'tor
};
```

Der Standardkonstruktor nimmt in der Menge aller Konstruktoren eine Sonderstellung ein: Wird durch den Anwender eine Klasse ohne Konstruktor definiert, ergänzt der C++-Compiler die Klassendefinition automatisch um eine Minimalversion des Standardkonstruktors. Leider ist dieses Geschenk mit dem Nachteil behaftet, dass die Instanzvariablen in diesem Fall mit undefinierten Werten vorbelegt werden.

Tipp

Warum sieht die Definition des Standardkonstruktors nicht vor, dass der Instanzvariablenbereich eines Objekts komplett mit Nullwerten vorbelegt wird – so wie dies beispielsweise in Java oder C# der Fall ist? Die Antwort ist sehr einfach: Wir sind auf einen Design-Fehler von C++ gestoßen. Zum Zeitpunkt der Definition von C++ stand das Thema „Performance“ weitaus mehr im Vordergrund als heute. Im Falle der Standardkonstruktoren wollte man einfach den Overhead einsparen, der zum Löschen des Objektspeichers erforderlich ist.

Die automatische Bereitstellung des Standardkonstruktors erfolgt nur, wenn in der Klasse keine anderen Konstruktoren definiert werden. Ein Codefragment in der Gestalt

```
class Fraction
{
public:
    int     m_num;   // numerator
    int     m_denom; // denominator

    Fraction (int num, int denom) { m_num = num; m_denom = denom; }
};

void main ()
{
    Fraction f;
}
```

wird deshalb in der `main`-Funktion vom Übersetzer mit der Fehlermeldung „*'Fraction' : no appropriate default constructor available*“ abgewiesen. Besitzt eine Klasse einen oder mehrere benutzerdefinierte Konstruktoren, so muss auch der Standardkonstruktor explizit definiert werden, wenn er für die Objekterzeugung zur Verfügung stehen soll. Falls nicht, könnte man ihn weglassen. Von wenigen Ausnahmen abgesehen gehört ein vom Anwender implementierter Standardkonstruktor immer zur vollständigen Definition einer Klasse.

Die Philosophie, die hinter dieser Regel steht, besagt: Für „sehr einfache“ Klassen – also Klassen, die keinen einzigen Konstruktor besitzen – entlastet der Compiler den Anwender von der Thematik *Objektinitialisierung* und definiert stellvertretend für ihn einen Standardkonstruktor. Bei allen anderen Klassen übernimmt der Compiler keine Verantwortung für die Integrität der Klasse und verlangt stattdessen, dass der Anwender alle erforderlichen Konstruktoren selbst beisteuert.

2.2.3. Initialisierungslisten

Viele Konstruktoren bestehen in ihrer Implementierung ausschließlich aus einer Folge von Wertzuweisungen, um die einzelnen Instanzvariablen zu initialisieren. Für diesen Fall kann man auf einen alternativen Mechanismus zur Objekterzeugung zurückgreifen, die so genannte *Initialisierungsliste*. Diese Liste folgt im Anschluss an die Formalparameterdeklaration eines Konstruktors, getrennt durch einen Doppelpunkt. Ein einzelnes Element der Initialisierungsliste besteht aus dem Namen einer Instanzvariablen, gefolgt von einem Wert in runden Klammern, der für die Vorbelegung der Instanzvariablen gewählt wird. Die einzelnen Elemente der Initialisierungsliste werden durch Komma voneinander getrennt:

```
// constructors
Fraction::Fraction () : m_num(0), m_denom(1) {}
Fraction::Fraction (int num, int denom) : m_num(num), m_denom(denom) {}
```

Wir sehen an diesem Beispiel, dass der Rumpf eines Konstruktors bei Gebrauch einer Initialisierungsliste sogar im Extremfall leer sein kann. Zur Laufzeit des Programms wird zuerst die Initialisierungsliste Element für Element abgearbeitet, danach kommen die verbleibenden Anweisungen im Konstruktorrumpf zur Ausführung.

Es gibt Situationen, in denen der Gebrauch einer Initialisierungsliste unumgänglich ist. Besitzt eine Klasse konstante Instanzvariablen, deren Werte durch einen Konstruktor übermittelt werden, müssen diese Instanzvariablen durch eine Initialisierungsliste initialisiert werden. Wir ziehen zur Veranschaulichung eine sich selbsterklärende Klasse `Pixel` zu Rate, deren Objekte nach ihrer Erzeugung den Ort und ihre Farbe nicht mehr verändern sollen. Eine entsprechende Deklaration der Klasse `Pixel` könnte so aussehen:

```
class Pixel
{
public:
    // data area
    const int m_x;
    const int m_y;
...
};
```

Ein Konstruktor der Gestalt

```
Pixel::Pixel (int x, int y)
{
    m_x = x;
    m_y = y;
}
```

liefert mit der resultierenden Fehlermeldungen *„Error: 'Pixel::m_x' : must be initialized in constructor base/member initializer list"* bereits den entscheidenden Hinweis für eine korrekte Implementierung. Eine alternative Realisierung in der Form

```
Pixel::Pixel (int x, int y) : m_x(x), m_y(y) {}
```

ist übersetzungsfähig und ermöglicht die Initialisierung von konstanten Instanzvariablen zum Erzeugungszeitpunkt des Objekts.

2.2.4. Zugriff einer Klasse auf sich selbst: Der `this`-Zeiger

Innerhalb einer Klasse kann auf Instanzmethoden und -variablen *derselben* Klasse direkt zugegriffen werden. Diese Eigenschaft haben wir uns in den bislang vorgestellten Instanzmethoden zu Nutze gemacht. In der `Add`-Methode der `Fraction`-Klasse kann man beispielsweise die beiden Instanzvariablen `m_num` und `m_denom` ohne zusätzliche Namensqualifizierung verwenden:

```
void Fraction::Add (int num, int denom)
{
    m_num = m_num * denom + m_denom * num;
    m_denom = m_denom * denom;
}
```

In speziellen Fällen benötigt man jedoch eine besondere Art der Namensqualifizierung, wie etwa bei dem Versuch, in der nachfolgenden Klasse `Fraction` einen benutzerdefinierten Konstruktor mit der vorgegebenen Signatur zu implementieren:

```
class Fraction
{
public:
    int num;   // numerator
    int denom; // denominator

    // constructor
    Fraction (int num, int denom)
    {
        // ToDo
    }
};
```

Erkennen Sie das Problem, auf das Sie in der Implementierung des Konstruktors stoßen? Im Rumpf des Konstruktors kommen für den Bezeichner `num` (wie auch `denom`) zwei Interpretationen in Betracht: Zum einen die als Instanzvariable der umgebenden Klasse und zum zweiten die als formaler Parameter des Konstruktors. Zu diesem Zweck besitzt jede Instanz einer Klasse eine implizit deklarierte Zeigervariable, die die Adresse der aktuellen Instanz enthält. Dieser Zeiger hat den Namen `this`. Jede Instanzmethode kann den `this`-Zeiger verwenden, ohne ihn vorher deklarieren zu müssen. Das nächste Codefragment demonstriert die Verwendung des `this`-Zeigers am Beispiel des zuvor beschriebenen Konstruktors:

```
Fraction::Fraction (int num, int denom)
{
    this->num = num;
    this->denom = denom;
}
```

Beachten Sie bitte, dass dieses Beispiel ohne den `this`-Zeiger nicht formulierbar wäre. Der `this`-Zeiger kann immer zur Adressierung der Instanzmethoden und -variablen der aktuellen Instanz verwendet werden. Formale Parameter und lokale Variablen von Methoden zählen nicht in die Kategorie der Instanzvariablen, der `this`-Zeiger ist also eine Hilfestellung, um derartige Konfliktfälle

aufzulösen.

Tipp

Manche Bücher der Informatikliteratur verwenden den `this`-Zeiger grundsätzlich, um den Zugriff auf Instanzmethoden und -variablen besonders hervorzuheben. Die `Add`-Methode aus unserer Beispielklasse `Fraction` würde dann so aussehen:

```
void Fraction::Add (int num, int denom)
{
    this->num = this->num * denom + this->denom * num;
    this->denom = this->denom * denom;
}
```

Ich habe mich für einen alternativen Programmierstil entschieden und versehe die Instanzelemente einer Klasse durchgehend mit dem Präfix „`m_`“ (steht für *member*).

2.2.5. Der Kopierkonstruktor

Neben dem Standardkonstruktor ist jede C++-Klasse mit einem weiteren Konstruktor ausgestattet, dem so genannten *Kopierkonstruktor*:

```
class Type
{
public:
    Type (const Type& t); // copy c'tor
}
```

Die Schnittstelle besitzt einen Parameter, um das zu kopierende Objekt zu spezifizieren. Dieses wird durch eine konstante Referenz spezifiziert, damit der Übersetzter versehentliche schreibende Zugriffe auf das Originalobjekt bereits zur Übersetzungszeit abweisen kann. Besitzt eine Klasse keinen Kopierkonstruktor, wird dieser – wie der Standardkonstruktor – automatisch vom Übersetzer generiert. Mit dem Kopierkonstruktor lässt sich eine Kopie eines Objekts erstellen:

```
Fraction f;      // using default c'tor
Fraction g = f;  // using copy c'tor
```

Der Aufruf eines Kopierkonstruktors bewirkt, dass der komplette Zustand eines Objekts in das neue Objekt umkopiert wird. Möchte man auf diesen Vorgang mehr Kontrolle ausüben können, muss man die Klasse um einen *benutzerdefinierten* Kopierkonstruktor ergänzen. Am Beispiel der Klasse `Fraction` sieht eine mögliche Implementierung des Kopierkonstruktors so aus:

```
Fraction::Fraction (const Fraction& f)
{
    m_num = f.m_num;
    m_denom = f.m_denom;
}
```

Tipp

Nebenbei bemerkt: Im Falle eines so einfachen Datentyps wie den der Klasse `Fraction` bräuchten wir keinen eigenen Kopierkonstruktor realisieren, da der vom System automatisch generierte just die zwei Instanzvariablen `m_num` und `m_denom` element- (bzw. byteweise) umkopieren würde. Im Falle der

`Fraction`-Klasse habe ich den Kopierkonstruktor nur geschrieben, um seine Implementierung an einem konkreten Beispiel vorführen zu können. Wir werden in den Folgekapiteln jedoch Klassen kennen lernen, bei denen die Definition eines benutzerdefinierten Kopierkonstruktors unumgänglich ist.

2.2.6. Destruktoren

Die Geburt eines Objekts wird stets, wie wir gesehen haben, durch einen Konstruktor begleitet. Diese Situation finden wir auch beim „Ableben“ eines Objekts vor. Die so genannten *Destruktoren* sind das Gegenstück zu Konstruktoren und werden am Ende der Lebensdauer eines Objekts automatisch aufgerufen. Destruktoren unterstützen im Regelfall ein Objekt bei der Aufgabe, seine zur Lebenszeit angeforderten Ressourcen wieder akkurat freizugeben. Im Gegensatz zu den Konstruktoren kann eine Klasse nur einen einzigen Destruktor besitzen. Der Methodenname eines Destruktors ist wie der von Konstruktoren identisch mit dem Klassennamen. Zur Unterscheidung von den Konstruktoren ist dieser mit einer vorangestellten Tilde kennzuzeichnen:

```
class Type
{
public:
    ~Type();  // d'tor
}
```

Die Schnittstellendefinition eines Destruktor ist parameterlos und besitzt (wie die eines Konstruktors) keinen Rückgabetyp.

Die Klasse `Fraction` ist auch kein prädestinierter Kandidat, um die Notwendigkeit für Destruktoren zu motivieren, da ihre Instanzen während ihrer Lebensdauer keine Ressourcen anfordern. Wir können allerdings die Gelegenheit nutzen, den Zeitpunkt des vom Laufzeitsystem garantierten Aufrufs eines Destruktors mit Hilfe von Trace-Ausgaben zu veranschaulichen. Zu diesem Zweck wechseln wir in Listing 2.13 vorübergehend zu folgender, vereinfachten Klassendefinition der `Fraction`-Klasse über:

```
01: class Fraction
02: {
03: public:
04:     // default c'tor and d'tor
05:     Fraction  ();
06:     ~Fraction ();
07: };
08:
09: // default c'tor
10: Fraction::Fraction()
11: {
12:     printf ("-> Fraction c'tor\n");
13: }
14:
15: // d'tor
16: Fraction::~Fraction()
17: {
18:     printf ("<- Fraction d'tor\n");
19: }
```

Beispiel 2.13. Klasse `Fraction` mit Trace-Ausgaben.

Der Destruktor eines Objekts wird grundsätzlich dann aufgerufen, wenn das Objekt am Ende seines Gültigkeitsbereichs angelangt ist. Für lokale Objekte ist dies das Ende des umgebenden Blocks, globale Objekte werden nach dem Verlassen der Anwendung beseitigt. Die Testanwendung aus Listing 2.14 bringt einige Konstruktoren- und Destruktorenaufrufe mit Hilfe von Trace-Ausgaben ans Tageslicht:

```
01: Fraction g;
```

```
02:
03: void main ()
04: {
05:     printf ("-> main\n");
06:
07:     Fraction x;
08:     printf (".. x is alive\n");
09:
10:     { Fraction y;
11:       printf (".. y is alive\n");
12:       printf (".. and dies\n");
13:     }
14:
15:     printf (".. between blocks\n");
16:
17:     { Fraction z;
18:       printf (".. z is alive\n");
19:       printf (".. and dies\n");
20:     }
21:
22:     printf (".. x dies\n");
23:     printf ("<- main\n");
24: }
```

Beispiel 2.14. Anwendung zum Testen von Konstruktoren- und Destruktorenaufrufen.

Die Ausgaben von Listing 2.14 lauten:

```
-> Fraction c'tor
-> main
-> Fraction c'tor
.. x is alive
-> Fraction c'tor
.. y is alive
.. and dies
<- Fraction d'tor
.. between blocks
-> Fraction c'tor
.. z is alive
.. and dies
<- Fraction d'tor
.. x dies
<- main
<- Fraction d'tor
<- Fraction d'tor
```

Jede C++-Klasse besitzt genau einen Destruktor. Wird der Destruktor in der Klassendefinition nicht explizit definiert, generiert der Compiler (wie im Falle des Standard- und des Kopierkonstruktors) automatisch einen Destruktor. In jedem Fall ist es ein nachahmenswerter Programmierstil, wenn die Deklaration einer Klasse grundsätzlich mit einem benutzerdefinierten Destruktor ausgestattet wird.

2.3. Zugriffsschutz

2.3.1. Zugriffsschutz

Ein zentrales Prinzip der objektorientierten Programmierung ist die so genannte *Kapselung*. Der Benutzer einer Klasse soll nur auf die Elemente ihrer *Schnittstelle* zugreifen können, die typischerweise durch die Konstruktoren und einen Ausschnitt der Instanzmethoden zum Ausdruck kommt. Die Daten sind im Gegensatz dazu vor dem Zugriff von außen zu schützen. Würde es keine Möglichkeit geben, den Zugriff auf die Daten (als auch Teile der Instanzmethoden) einzuschränken oder gar zu unterbinden, könnte man – beispielsweise mit der schon öfters zitierten Klasse `Date` – das folgende Codefragment formulieren:

```
Date d;
d.m_day = 99;
```

Wenn gleich dieser Fehler offensichtlich ist, da man den falschen Wert unmittelbar im Quellcode erkennen kann, resultiert daraus ein Objekt mit einem inkonsistenten Zustand. Eines der hehren Ziele objektorientierter Programmierung, die Integrität des Objektzustands für seine gesamte Lebensdauer zu wahren, wird dadurch verletzt. Zustandsverletzungen könnten natürlich auch die Folge weitaus weniger offensichtlicher Fehlerquellen sein, zum Beispiel in Folge des Aufrufs einer fehlerhaften Methode:

```
Date d;
Calender cal;
d.m_day = cal.ComputeXmasDay ();  // faulty method 'computeCurrentXmasDay'
```

Liegt in der Implementierung der `ComputeXmasDay`-Methode, die den Wochentag des Weihnachtsfestes berechnen soll, ein Fehler vor, hätten wir wiederum ein `Date`-Objekt in einen inkonsistenten Zustand versetzt. Zur Vermeidung derartiger Fehlerquellen gibt es in C++ das Konzept des *Zugriffsschutzes*. Es ermöglicht mit Hilfe der Zugriffsspezifizierer `public`, `protected` und `private` den eingeschränkten Zugriff auf die Instanzdaten und -methoden eines Objekts.

Steht innerhalb einer Klassendeklaration eines der zuvor erwähnten Schlüsselwörter (gefolgt von einem Doppelpunkt), so wird der Zugriff aller nachfolgend deklarierten Daten und Methoden wie folgt geschützt:

- Zugriffsklasse `private`:

  Alle nachfolgenden Deklarationen nehmen einen *privaten* Status innerhalb der Klasse an. Dies bedeutet, dass auf die so markierten Elemente der Klasse nur von Instanzmethoden *derselben* Klasse zugegriffen werden kann. Ausgenommen von dieser Regel sind befreundete Klassen und Funktionen, auf die wir in XXX noch zu sprechen kommen. Methoden, die nicht in dieser Klasse deklariert sind, haben folglich keinerlei Möglichkeit, auf die privat deklarierten Elemente dieser Klasse zuzugreifen!

- Zugriffsklasse `public`:

  Alle Deklarationen, die auf dieses Schlüsselwort folgen, nehmen einen *öffentlichen* Status an, d.h. sie sind von *überall* zugreifbar. Kurzum: Alle so deklarierten Elemente unterliegen in einem C++-Programm keinerlei Zugriffsbeschränkungen. Diese Kategorie des Zugriffsschutzes steht offensichtlich im Widerspruch zur Kapselung der sensiblen Daten eines Objekts. Aus diesem Grund werden in der Regel nur diejenigen Methoden und Konstruktoren einer Klasse mit `public` deklariert, die zur Schnittstelle der Klasse zählen.

- Zugriffsklasse `protected`:

  Die Anwendung des Zugriffsspezifizierers `protected` ist nur in einer Vererbungshierarchie sinnvoll, wir kommen auf seine Bedeutung im Kapitel über das Vererbungsprinzip zu sprechen.

Das folgende Beispiel (Listing 2.15) zeigt typische Möglichkeiten auf, wie die zwei Zugriffsspezifizierer `public` und `private` sinnvoll eingesetzt werden können:

```
01: class Date
02: {
03: private:
04:     // private member data
05:     int m_day;
06:     int m_month;
07:     int m_year;
08:
09: public:
10:     // c'tors
11:     Date ();
12:     Date (int day, int month, int year);
13:
14: public:
15:     // public interface
16:     void Print ();
17:
18: private:
19:     // internal helper methods
20:     bool IsValid (int day, int month, int year);
21: };
```

Beispiel 2.15. Klasse `Date` mit öffentlichen und privaten Elementen.

Für die Anwendung der Zugriffsspezifizierer gelten einige Regeln:

- Deklarationen mit `public`, `protected` und `private` dürfen beliebig oft aufeinander folgen. Sie gelten jeweils bis zum nächsten Deklarationsabschnitt mit einem Zugriffsspezifizierer oder aber bis zum Ende der Klassendeklaration.
- Beginnt eine Klassendeklaration nicht mit einem Zugriffsspezifizierer, so besitzen alle Elemente bis zum ersten Zugriffsspezifizierer den Status `private`.

Auf Basis der Klasse `Date` aus Listing 2.15 betrachten wir den folgenden Anwendungsfall:

```
void main ()
{
    Date d1;                        // ok
    Date d2 (1, 1, 2000);           // ok
    d1.m_day = 99;                  // error
    d1.IsValid (31, 2, 2000);       // error
    d2.Print ();                    // ok
}
```

Die mit *„ok“* kommentierten Zeilen sind fehlerfrei übersetzungsfähig, da die beiden Konstruktoren und die `Print`-Methode der `Date`-Klasse mit dem Zugriffsspezifizierer `public` deklariert sind. Alle Datenelemente sowie die Hilfsmethode `IsValid` sind sinnvollerweise in der Klasse (mit dem Zugriffsspezifizierer `private`) gekapselt, der Compiler weist jeglichen Zugriffsversuch mit der Fehlermeldung *„cannot access private member declared in class 'Date'“* ab.

2.3.2. „getter“- und „setter“-Methoden

Mit der Einführung der Sichtbarkeitsattribute haben wir den direkten Zugriff auf Instanzvariablen verloren, die mit dem Attribut `private` definiert sind. Legen wir wiederum die Klasse `Fraction` auf Basis der Definition

```
class Fraction
{
private:
    // data area
    int  m_num;          // numerator
    int  m_denom;        // denominator
    ...
};
```

zu Grunde, lässt sich beispielsweise das simple Codefragment

```
void main ()
{
    Fraction f;
    f.m_num = 1;
}
```

nicht mehr übersetzen. Der Compiler meldet zu recht: *„'Fraction::m_num' : cannot access private member declared in class 'Fraction'“*. Natürlich muss es dennoch möglich sein, den Zustand eines Objekts verändern zu können. Da der Ausgangspunkt für die Einführung der Sichtbarkeitsattribute in der Wahrung der Integrität eines Objekts liegt, ergänzen wir die Klasse `Fraction` um eine oder mehrere spezielle, mit dem Sichtbarkeitsattribut `public` definierte Zugriffsmethoden. Diese ermöglichen einerseits den Zugriff auf die – privat deklarierten – Instanzvariablen, behalten aber andererseits die Integrität des Objekts im Hinterkopf. Der Umweg über diese Zugriffsmethoden bewirkt somit einen ausschließlich *kontrollierten* Zugriff auf das Objekt:

```
01: class Fraction
02: {
03: private:
04:     // data area
05:     int  m_num;         // numerator
06:     int  m_denom;       // denominator
07:
08: public:
09:     // default c'tor
10:     Fraction ();
11:
12:     // getter / setter methods
13:     int GetNum () const { return m_num; };
14:     void SetNum (int num);
15:     int GetDenom () const { return m_denom; };
16:     void SetDenom (int denom);
17: };
18:
19: // c'tor
20: Fraction::Fraction ()
21: {
22:     m_num = 0;
23:     m_denom = 1;
24: }
25:
26: // setter methods
27: void Fraction::SetNum (int num)
28: {
29:     m_num = num;
30: }
31:
32: void Fraction::SetDenom (int denom)
33: {
34:     if (denom != 0)
35:         m_denom = denom;
36: }
```

Beispiel 2.16. Der Zugriff auf die Instanzvariablen eines Objekts wird in speziellen Zugriffsmethoden platziert.

Die in Listing 2.16 betrachteten Zugriffsmethoden (Zeilen 13 bis 16 und 27 bis 36) bezeichnet man im

Fachjargon als *getter*- oder *setter*-Methoden, da ihre Namen typischerweise mit dem Kürzel *Set* bzw. *Get* gebildet werden. Mit der Definition von Klasse `Fraction` aus Listing 2.16 lässt sich folgendes Beispielprogramm formulieren:

```
void main ()
{
    Fraction f;
    f.SetNum (1);
    f.SetDenom (7);
    int num = f.GetNum ();
    int denom = f.GetDenom ();
    printf ("numerator of f is %d, denominator is %d\n", num, denom);
}
```

Die Implementierung der *setter*-Methode für den Zähler kann jeden `int`-Wert ohne Wertebereichsüberprüfung akzeptieren. Beim Nenner müssen wir einen Null-Wert verhindern, diese Abprüfung erfolgt in Zeile 34 von Listing 2.16. Es ist mit Hilfe dieser Zugriffsmethoden nicht mehr möglich, ein `Fraction`-Objekt in einen inkonsistenten Zustand zu versetzen. Man beachte in Listing 2.16 ferner die Benutzung des `const`-Schlüsselworts bei den *getter*-Methoden. Auf diese Weise lassen sich *getter*-Methoden auch auf konstante `Fraction`-Objekte anwenden.

2.3.3. *Freunde* einer Klasse

Wendet man beim Entwurf einer Klasse die Regeln für den Zugriffsschutz an, gelten die Beschränkungen in der Sichtbarkeit der Klassenelemente für alle Klassenbenutzer. Gelegentlich ist es jedoch sinnvoll, dass die nicht zugänglichen Elemente einer Klasse ausnahmsweise doch zugänglich sind, und zwar entweder

- für eine globale Funktion,
- eine einzelne Instanzmethode einer anderen Klasse oder
- komplett für eine andere Klasse.

Für das gezielte und kontrollierte Aufheben des Zugriffsschutzes gibt es in C++ das Konzept des „Freundes“. Eine Klasse kann gezielt mit Hilfe des `friend`-Schlüsselworts für die zuvor aufgelisteten Sprachelemente den Zugriff auf die privaten Elemente einer Klasse einrichten. Als Beispiel wenden wir uns einer einfachen Klasse `A` zu, die eine ganzzahlige Instanzvariable `i` vom Typ `int` besitzt. Um `i` generell zu schützen, wird sie mit der Zugriffsklasse `private` deklariert:

```
class A
{
private:
    // data area
    int i;
};
```

Wollten wir gezielt einer globalen Methode `f`, einer Instanzmethode `b` aus der Klasse `B` oder der gesamten Klasse `C` den Zugriff auf die private Instanzvariable `i` gestatten, müssen wir die Klasse `A` um folgende `friend`-Deklarationen ergänzen:

```
class A
{
    friend void  f();       // global function f
    friend void  B::b();    // instance methode B::b
    friend class C;         // class C

    ...
};
```

Mit diesen Deklarationen sind jetzt folgende Implementierungen der „Freunde“ von Klasse `A` übersetzbar:

```
// class declarations
class B
{
public:
    void b();
};

class C
{
public:
    void c();
};

// implementation
void B::b() { A a; a.i = 97; }
void C::c() { A a; a.i = 98; }

// global function
void f()
{
    A a;
    a.i = 99;
}
```

Ohne das Konzept der Freunde würden die Anweisungen dieses Codefragments dreimal mit der Fehlermeldung *„Error 'A::i' : cannot access private member declared in class 'A'“* abgewiesen werden. Mit Hilfe der `friend`-Direktive kann der Datenschutz einer Klasse punktuell aufgehoben werden. Da eine Klasse selbst bestimmt, wer zu ihren Freunden zählt, wird das generelle Konzept des Zugriffsschutzes nicht ausgehebelt!

Achtung

`friend`-Direktiven können in einer Klassendeklaration an einer beliebigen Stelle aufgeführt werden. Sie werden von den Regeln des Zugriffsschutzes (`public`, `protected`, `private`) nicht berührt.

2.3.4. *Veränderbare* und *unveränderbare* Objekte

Beim Entwurf der Klasse `Fraction` können Sie für die Gestaltung der Methodensignaturen prinzipiell zwei Wege einschlagen:

- Entwurf einer Klasse mit *veränderbarem Verhalten*:

  Objekte mit *veränderbarem Verhalten* (engl. *mutable behaviour*) besitzen die Eigenschaft, dass ihr Zustand nach der Objekterzeugung veränderbar ist. Würden wir die Klasse `Fraction` nach dieser Richtlinie entwerfen, bietet sich für eine Methode `Add` folgende Realisierung an:

  ```
  void Fraction::Add (Fraction f)
  {
      m_num = m_num * f.m_denom + m_denom * f.m_num;
      m_denom = m_denom * f.m_denom;
  }
  ```

  Die `Add`-Methode besitzt kein explizites Ergebnis (Rückgabetyp `void`), das Resultat der Operation wird direkt am aufgerufenen Objekt abgelegt:

  ```
  Fraction f (1, 2);
  Fraction g (3, 2);
  ```

```
f.Add (g);  // f equals now f+g
```

Dieses Verhalten entspricht dem objektorientierten Paradigma in der Gestalt, dass ein Objekt durch eine Aktion – den Aufruf einer Methode – in einen neuen Zustand übergeht (Abbildung 2.2).

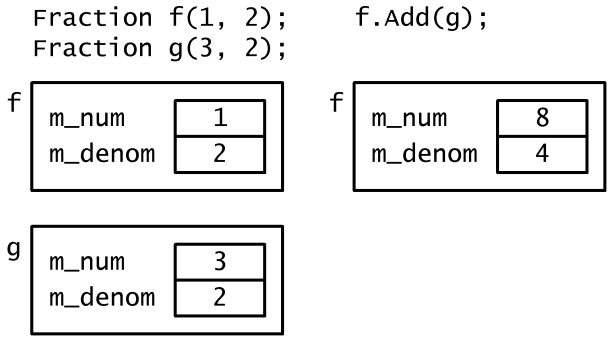

Abbildung 2.2. Aktionen an einem Objekt mit veränderbarem Verhalten ändern den Objektzustand.

Eine Nebenwirkung dieser Variante ist der Umstand, dass der alte Zustand des gerufenen Objekts nach dem Methodenaufruf nicht mehr existiert!

- Entwurf einer Klasse mit *unveränderbarem Verhalten*:

Objekte mit *unveränderbarem Verhalten* (engl. *immutable behaviour*) weisen die Eigenschaft auf, dass ihr Zustand nach der Objekterzeugung nicht mehr änderbar ist. In diesem Sinn können sie als *konstante* Objekte betrachtet werden. Die `Add`-Methode könnten wir in dieser Variante in der Form

```
Fraction Fraction::Add (Fraction f)
{
    int n = m_num * f.m_denom + m_denom * f.m_num;
    int m = m_denom * f.m_denom;
    return Fraction(n, m);
}
```

realisieren, also insbesondere mit Rückgabetyp `Fraction`. Für das Ergebnis wird ein neues Objekt erzeugt, dessen Kopie beim Verlassen der Methode dem Aufrufer als Resultat ausgehändigt wird (Abbildung 2.3), wie auch im nachfolgenden Beispiel demonstriert wird:

```
Fraction f (1, 2);
Fraction g (3, 2);
Fraction h = f.Add (g);
```

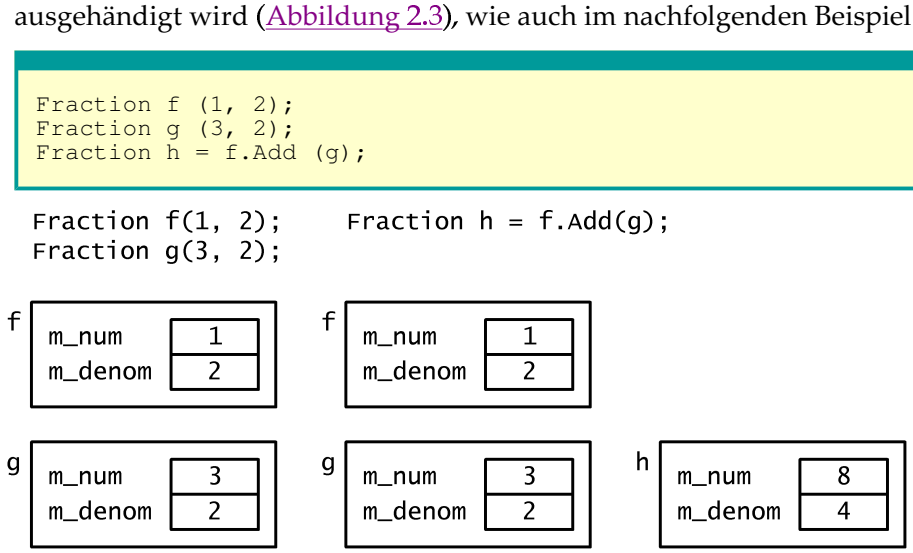

Abbildung 2.3. Objekte mit unveränderbarem Verhalten ändern bei Methodenaufrufen ihren Zustand nicht.

Zu beachten ist, dass der Zustand des gerufenen Objekts (hier: Instanz `f`) nicht verändert wird. Um an das Ergebnis des Methodenaufrufs zu gelangen, muss der Rückgabewert einem Objekt

zugewiesen werden. Dieses könnte auch das Objekt `f` sein!

Vorsicht: Ein Aufruf der Gestalt

```
f.Add(g);
```

ist syntaktisch korrekt (der Compiler generiert keine Fehlermeldung), das Resultat des `Add`-Methodenaufrufs geht allerdings aufgrund der fehlenden Wertzuweisung verloren.

Im Design von Klassen ist von Fall zu Fall zu entscheiden, für welche Strategie man sich in Bezug auf ihre Veränderbarkeit entscheidet. Die Klasse `Fraction` ist in den ersten Kapiteln veränderbar konzipiert, im Kapitel „Überladen von Operatoren“ werden wir zu einer unveränderbaren Variante überschwenken.

2.4. Klassenvariablen und -methoden

2.4.1. Klassenvariablen

Es gibt gelegentlich die Situation, dass man in einer Klasse eine Variable benötigt, die für alle Instanzen der Klasse nur einmal existiert. Ein einfacher Anwendungsfall hierfür ist die Aufgabe, die Anzahl der Instanzen einer Klasse während der Ausführung eines Programms zu einem bestimmten Zeitpunkt zu bestimmen. Man benötigt dazu eine einzige Zählervariable. Jedes Mal, wenn ein Objekt neu angelegt wird, kann man im Konstruktor diese Zählervariable inkrementieren. Wird das Objekt zerstört, wird die Zählervariable im Destruktor dekrementiert. Bleibt nur noch die Frage zu klären, wie und wo die Zählervariable geschickt definiert wird? Eine simple Lösung dieses Anwendungsfalls wäre natürlich der Einsatz einer globalen Variablen. C wie auch C++ gestatten die Definition globaler Variablen, die zur gesamten Laufzeit des Programms existieren und von überall aus erreichbar sind. Allerdings sind gerade globale Variablen die Hauptverursacher unübersichtlichen Quellcodes („Spaghetti-Code“). C++ bietet deshalb mit den so genannten *Klassenvariablen* eine alternative Möglichkeit.

Eine Klassenvariable ist innerhalb einer Klasse zu deklarieren. Zur Unterscheidung von einer „gewöhnlichen“ Instanzvariablen ist sie mit dem Schlüsselwort `static` zu kennzeichnen. Dies bewirkt, dass der Speicherplatz dieser Variablen nur einmal im Speicher angelegt wird. Insbesondere ist das Vorhandensein einer Klassenvariablen unabhängig von der Anzahl der Objekte, die von dieser Klasse existieren! Eine Klassenvariable ist kein Teil eines Objekts, dass von dieser Klasse existiert. Sie ist also durchaus von ihren Eigenschaften her mit einer klassischen globalen C-Variablen vergleichbar.

Wir kommen auf das Beispiel zum Zählen der Instanzen einer Klasse zurück, in Listing 2.17 erweitern wir die Klasse `Fraction` um eine Klassenvariable `counter`:

```
01: class Fraction
02: {
03: public:
04:     // keep track of 'Fraction' objects
05:     static int counter;
06:
07:     ...
08: };
```

Beispiel 2.17. Deklaration einer Klassenvariablen.

Die Deklaration einer Klassenvariablen in einer Klasse stellt noch keinen Speicherbereich für diese Variable bereit. Zu diesem Zweck muss sie zusätzlich außerhalb der Klasse definiert werden (Listing 2.18):

```
01: int Fraction::counter = 0;
```

Beispiel 2.18. Definition einer Klassenvariablen außerhalb der Klasse.

Die Definition einer Klassenvariablen außerhalb der Klasse ergibt durchaus Sinn. Da eine Klassenvariable nur einmal initialisiert werden darf, ein Konstruktor aber bei der Erzeugung eines Objekts jedes Mal von neuem aufgerufen wird, stehen Konstruktoren für die Initialisierung einer Klassenvariablen nicht zur Verfügung. Somit ist eine außerhalb der Klasse angesiedelte Klassenvariablendefinition der geeignete Ort, um eine Klassenvariable zu initialisieren.

Tipp

Die Qualifizierung von Klassenvariablen mit dem Schlüsselwort `static` ist mehr als unglücklich. In C (und C++) werden nämlich bereits lokale Variablen einer *Funktion*, die *globalen* Charakter besitzen, mit `static` markiert. Zusätzlich gibt es auch noch die Qualifizierung von globalen Variablen mit `static`, um ihre Sichtbarkeit auf die aktuelle Datei einzuschränken.

Natürlich darf in einem Konstruktor – wie auch in jeder anderen Instanzmethode der Klasse – auf eine Klassenvariable zugegriffen werden. Um beispielsweise die Anzahl der zur Laufzeit einer Anwendung existierenden `Fraction`-Instanzen korrekt zu erfassen, modifizieren wir den Konstruktor und Destruktor wie in Listing 2.19 gezeigt:

```
01: // c'tor
02: Fraction::Fraction ()
03: {
04:     Fraction::counter ++;
05:     ...
06: }
07:
08: // d'tor
09: Fraction::~Fraction ()
10: {
11:     Fraction::counter --;
12:     ...
13: }
```

Beispiel 2.19. Klasse `Fraction` mit Zählung ihrer Instanzen.

Wir testen diese Variante der `Fraction`-Klasse an folgendem Codefragment:

```
void main ()
{
    printf ("%d\n", Fraction::counter);
    {
        Fraction a;
        {
            Fraction b, c;
            printf ("%d\n", Fraction::counter);
        }
        printf ("%d\n", Fraction::counter);
    }
    printf ("%d\n", Fraction::counter);
}
```

Die Ausgabe dieses Codefragments lautet:

```
# Fraction objects: 0
# Fraction objects: 3
# Fraction objects: 1
# Fraction objects: 0
```

Die Ausführung dieses Beispiels verdeutlicht nebenbei, dass Klassenvariablen auch dann existieren, wenn von der Klasse noch kein einziges Objekt existiert.

Tipp

Wie wir gesehen haben, lassen sich Klassenvariablen durchaus mit globalen Variablen vergleichen. Es gibt allerdings ein signifikantes Merkmal, das den qualitativen Unterschied des Konzepts zwischen den beiden Variablen-Kategorien zum Ausdruck bringt. Globale Variablen in C oder C++ bestehen aus einem einzigen Namen. In größeren Programmen kann es deshalb zu Konflikten mit globalen Variablen desselben Namens kommen, die in unterschiedlichen Modulen angesiedelt sind. Da eine

Klassenvariable bei ihrer Deklaration einer Klasse zuzuordnen ist, ist diese bzgl. der Namensgebung in den Namensraum der Klasse eingebettet, sprich der Name einer Klassenvariablen besteht immer aus dem Paar Klassenname *plus* Variablenname. Auf diese Weise besitzen Klassenvariablen immer den konzeptionellen Bezug zu einer Klasse, der unstrukturierte Charakter globaler Variablen ist nicht mehr vorhanden.

Aus diesem Grund ist es auch möglich, in einem C++-Programm mehrere „globale" Variablen mit dem Namen `counter` zu definieren. Da Klassenvariablen erst durch den Namen der Klasse eindeutig werden, in deren Kontext sie definiert sind, lassen sich problemlos mehrere Klassenvariablen wie etwa `Rectangle::counter`, `Circle::counter` und `Line::counter` in einem C++-Programm definieren und unterscheiden. Es existieren potentiell unendlich viele Namensräume in C++, um Klassenvariablen nach logischen Gesichtspunkten zu sortieren.

2.4.2. Klassenmethoden

Nach der Lektüre des letzten Abschnitts drängt sich die Frage auf, ob es neben Klassenvariablen auch „Klassenmethoden" gibt. Wenn ja, welche besonderen Eigenschaften könnten eine Klassenmethode auszeichnen? Zur Beantwortung dieser Frage werfen wir in Listing 2.20 einen Blick auf die folgende Klasse `Arithmetic`:

```
01: class Arithmetic
02: {
03: public:
04:     int SumOfDigits (int);
05: };
06:
07: int Arithmetic::SumOfDigits (int n)
08: {
09:     int result = 0;
10:     while (n % 10 != 0)
11:     {
12:         result += n % 10;
13:         n /= 10;
14:     }
15:
16:     return result;
17: }
```

Beispiel 2.20. Klasse `Arithmetic` mit Methode `SumOfDigits`.

Um die Quersumme der Zahl 12345 zu berechnen, erzeugen wir eine Instanz der Klasse `Arithmetic` und rufen an diesem Objekt die `SumOfDigits`-Methode auf:

```
Arithmetic a;
int n = a.SumOfDigits (12345);
printf ("SumOfDigits (12345) = %d\n", n);
```

Die Ausführung dieser drei Anweisungen liefert natürlich das erwartete Ergebnis 15, dennoch drängt sich an diesem Beispiel die Frage auf: Auf welche Instanzvariablen der Klasse `Arithmetic` greift die `SumOfDigits`-Methode während ihrer Ausführung zu? Auf keine, die `Arithmetic`-Klasse besitzt überhaupt keine Instanzvariablen! Wozu benötigen wir dann eine Instanz der `Arithmetic`-Klasse, wenn die Ausführung der `SumOfDigits`-Methode die erzeugte Instanz links liegen lässt? Wir sind beim Konzept der *Klassenmethoden* angekommen und betrachten dazu Listing 2.21:

```
01: class Arithmetic
02: {
03: public:
04:     static int SumOfDigits (int);
05: };
```

```
06:
07: void main ()
08: {
09:     int n = Arithmetic::SumOfDigits (12345);
10:     printf ("SumOfDigits (12345) = %d\n", n);
11: }
```

Beispiel 2.21. Klassenmethoden werden an der Klasse aufgerufen.

In Listing 2.21 finden wir wieder das Schlüsselwort `static` vor, dieses Mal in Zeile 4 in der Schnittstellendefinition der `SumOfDigits`-Methode. Der Aufruf der `SumOfDigits`-Methode erfolgt in Zeile 9, wie bei den Klassenvariablen spezifiziert der Klassenname die Methode eindeutig. Die Implementierung der `SumOfDigits`-Methode bleibt von diesen Änderungen unberührt, wir legen die Realisierung aus Listing 2.20 zu Grunde.

Klassenmethoden besitzen wie Klassenvariablen gewisse Ähnlichkeiten mit „globalen Funktionen" von C bzw. C++: Eine Klassenmethode kann jederzeit – ohne die Existenz eines Objekts dieser Klasse – im laufenden Programm aufgerufen werden. Ihrem Wesen nach sind Klassenmethoden ähnlich zu globalen C-Funktionen, die eine bestimmte Berechnung oder allgemeiner formuliert, eine bestimmte Dienstleistung erbringen und nicht an den Zustand eines Objekts gekoppelt sind. Die Ausführung der Klassenmethode erfolgt autark vom Gedächtnis potentieller Objekte dieser Klasse.

Achtung

Mit folgender Testfrage können Sie Ihr Wissen bzgl. Klassenvariablen und -methoden überprüfen: Ist das Codefragment aus Listing 2.22 übersetzungsfähig oder nicht? Erkennen Sie das Problem?

```
01: class Test
02: {
03: public:
04:     // data member
05:     int m_i;
06:
07:     // c'tor
08:     Test () { m_i = 0; }
09:
10:     // public class method
11:     static void StaticMethod ()
12:     {
13:         int j = m_i;
14:     }
15: };
```

Beispiel 2.22. Klassenmethoden werden an der Klasse aufgerufen.

Die Klassenmethode `StaticMethod` greift in Zeile 13 auf die Instanzvariable `m_i` zu, was natürlich nicht möglich ist, da Klassenmethoden keinen Bezug zu Objekten der umgebenden Klasse besitzen – sofern solche überhaupt existieren. Der C++-Compiler quittiert die fehlerhafte Zeile mit der Fehlermeldung *„illegal reference to non-static member 'Test::m_i'"*. Aus demselben Grund kann deshalb der `this`-Operator niemals in einer Klassenmethode auftreten, da `this` ja eine Referenz des aktuellen Objekts repräsentiert.

2.5. Arrays von Objekten

2.6. Dynamische Objekte

2.6.1. Zeiger auf Objekte

2.6.2. Freigabe dynamischer Objekte

2.7. Namensräume

Wird ein größeres C++-Projekt in mehreren Quelltextdateien abgelegt, müssen die Namen aller global definierten Elemente (wie globale Variablen, Klassen und andere global definierte Datentypen wie Strukturen oder Aufzählungstypen) verschieden sein. Liegt eine Namensgleichheit vor, kommt es entweder bereits zum Übersetzungszeitpunkt oder aber spätestens beim Binden des Programms zu Fehlern. Der Grund dafür liegt einfach darin, dass alle global definierten Programmelemente einem gemeinsamen Namensbereich zugeordnet werden, dem sogenannten *globalen Namensraum* und deshalb für den Compiler unterschiedlich sein müssen.

Arbeiten mehrere Entwickler gemeinsam an einem umfangreicheren Programm, kann es durchaus unbeabsichtigter Weise zu einem Namenskonflikt kommen. Wir zeigen dies an einem einfachen Beispiel mit zwei Header-Dateien, die von zwei verschiedenen Teammitgliedern erstellt worden sind und beide – zufälligerweise – eine Klasse `Fraction` definieren:

```
// header_01.h:
// definition of class Fraction
class Fraction
{
};
...
```

bzw.

```
// header_02.h:
// a second definition of class Fraction
class Fraction
{
};
...
```

Werden in einer C++-Datei die beiden Headerdateien *header_01.h* und *header_02.h* inkludiert, generiert der C++-Compiler die Fehlermeldung *„'Fraction' : 'class' type redefinition“*. Mit Hilfe einer Namensraumdeklaration kann man einen Bezeichner definieren, der additiv zu allen Elementen des korrespondierenden Deklarationsabschnitts hinzugefügt wird. Auf diese Weise lässt sich – unterschiedliche Namensraumbezeichner vorausgesetzt – die Wahrscheinlichkeit verringern, dass die zusammengesetzten Bezeichner immer noch zu einem Namenskonflikt führen. Mit Hilfe unterschiedlicher Namensraumbezeichner ist es möglich, Datentypen mit demselben Namen in unterschiedlichen Namensräumen zu definieren und auch zu benutzen – sogar in dem Fall, dass im aktuellen Namensraum ein Element mit demselben Namen vorhanden ist. Wir untermauern dieses Sprachmittel mit einer Reihe von Beispielen. Zunächst ergänzen wir die zwei Header-Dateien um einen Namensraum, um als Erstes den Übersetzungskonflikt zu beseitigen:

```
// header_01.h:
// definition of class Fraction
namespace Library
{
    class Fraction
    {
    };
    ...
}
```

und

```
// header_02.h:
// definition of another class Fraction
namespace AnotherLibrary
```

```
{
    class Fraction
    {
    };
    ...
}
```

Es sind nun beide Header-Dateien in einer C++-Übersetzungseinheit benutzbar. Wenn wir die Klasse `Fraction` verwenden wollen, müssen wir den *voll-quantifizierten* Klassennamen verwenden, also die Kombination aus Klassennamen und Namensraum, in dem die Klasse definiert ist:

```
#include "Header_01.h"
#include "Header_02.h"

void main ()
{
    AnotherLibrary::Fraction f;
}
```

Mit Hilfe des Gültigkeitsbereichsoperators sprechen wir das Element eines Namensraums direkt an. Ohne die Zuhilfenahme eines Namensraumbezeichners ist in diesem Beispiel die Verwendung der `Fraction`-Klasse nicht möglich, da auf Grund der beiden *#include*-Direktiven zwei unterschiedliche `Fraction`-Klassendefinitionen in der Übersetzungseinheit vorliegen!

Namensräume sind auch schachtelbar. Programmiersprachliche Konstrukte wie Klassen oder Strukturen lassen sich auf diese Weise feingranularer in hierarchisch gegliederten Namensräumen ansiedeln:

```
namespace System
{
    namespace Windows
    {
        namespace Controls
        {
            class Button
            {
                ...
            };
        }
    }
}
```

Für den Zugriff auf derartige Konstrukte setzt man den Gültigkeitsbereichsoperator kaskadiert ein:

```
void main ()
{
    System::Windows::Controls::Button b;
}
```

Die kaskadierte Verwendung des Gültigkeitsbereichsoperators ist vielleicht nicht sehr elegant. Aus diesem Grund gibt es Möglichkeiten des vereinfachten Zugriffs, die wir im nächsten Abschnitt besprechen.

2.7.1. Die `using`-Direktive und die `using`-Anweisung

Wird in einem Programm häufig auf Elemente unterschiedlicher Namensräume zugegriffen, kann die Schreibweise mit kaskadierten Namensräumen ziemlich schnell recht unübersichtlich oder umständlich geraten. Mit Hilfe der `using`-Direktive teilt man dem Compiler mit, dass bei der Übersetzung einer Übersetzungseinheit alle Elemente eines Namensraums mit einzubeziehen sind. Die Notation mit den recht umständlichen kaskadierten Namensraumbezeichnern entfällt auf diese Weise:

```
void main ()
{
    using namespace System::Windows::Controls;

    Button b;
}
```

Die Namensraumdirektive macht alle Elemente eines Namensraums verfügbar. Durch die Angabe mehrerer `using namespace`-Direktiven ist es möglich, die Namen verschiedener Namensbereiche verfügbar zu machen, um gleichzeitig auf Namen in unterschiedlichen Namensräumen zugreifen zu können. Dabei treten Spielregeln in Kraft in Bezug auf die Reihenfolge, in der nach Namen gesucht wird: Zunächst wird die Tabelle aller lokalen Deklarationen zur Auflösung eines Namens durchsucht. Wird der gesuchte Name hier nicht gefunden, wird die Suche in umgebenden Blöcken – sofern vorhanden – fortgesetzt. Wird der Name auch dort nicht gefunden, werden schließlich alle importierten Namensräume nach dem gesuchten Namen durchsucht. Sollte dabei wiederum eine Mehrdeutigkeit entstehen, produziert der Übersetzung die Fehlermeldung, dass ein bestimmter Name ein *ambiguous symbol* darstellt. Es wird also vermieden, dass auf Grund des Imports von Namensräumen die Eindeutigkeit des Quellcodes verloren geht!

Die `using`-Direktive kann lokal in einer Funktion stehen (und beeinflusst dann auch nur den Übersetzungsverlauf dieser Funktion) oder aber sie ist global platziert:

```
using namespace System::Windows::Controls;

void main ()
{
    Button b;
}
```

Im letzten Beispiel beeinflusst die `using`-Direktive den Übersetzungsverlauf ab ihrer Platzierung im Quellcode. Möchte man mit der `using namespace`-Direktive verhindern, dass der Namensraum komplett in den Übersetzungslauf mit einfließt, kann man einzelne Namen eines Namensraums auch selektiv bekannt machen. Dazu verwendet man das `using`-Schlüsselwort im Kontext der `using`-Anweisung:

```
void main ()
{
    using System::Windows::Controls::Button;

    Button b;
}
```

Durch die Anweisung `using System::Windows::Controls::Button;` wird der Klassenname `Button` zu einem Synonym von `System::Windows::Controls::Button`. Im vorliegenden Beispiel kann der Klassenname `Button` allerdings nur lokal in der Funktion `main` verwendet werden, da die `using`-Anweisung lokal in dieser Funktion enthalten ist. Wie bei der `using namespace`-Direktive kann man auch `using`-Anweisungen außerhalb von Funktionen, also im globalen Namensraum, platzieren.

2.7.2. Unbenannte Namensräume und Aliasnamen

Um die Entwicklung eines Programms grundsätzlich gegen Kollisionen mit einem anderen Quelltext zu schützen, gibt es in C++ auch Namensräume ohne Namen. Ein unbenannter Namensraum sieht so aus:

```
namespace
{
    int x;
```

```
}
```

Der Zugriff auf die Elemente eines solchen Namensraums ist nur in der Übersetzungseinheit möglich, in der die Definition dieses Namensraums steht. Gibt es in unterschiedlichen Übersetzungseinheiten (also in verschiedenen Dateien) eines Projekts unbenannte Namensräume, so erzeugt der Compiler automatisch für jede Übersetzungseinheit, in der ein unbenannter Namensraum definiert wird, einen separaten Namen, so dass die unbenannter Namensräume in unterschiedlichen Dateien voneinander verschieden sind. Betrachten wir zum Beispiel eine Datei *File01.cpp* mit dem Inhalt

```
namespace
{
    int i;
}
```

und eine zweite Datei *File02.cpp* mit folgendem Inhalt:

```
extern int i;

void Test ()
{
    i = 1;
}
```

Die zwei Dateien *File01.cpp* und *File02.cpp* sind für sich separat fehlerfrei übersetzungsfähig. Beim Binden jedoch erhalten wir die Fehlermeldung *„NameSpaces error: unresolved external symbol "int i"“*: Der Compiler modifiziert den Variablennamen `i` aus dem unbenannten Namensraum mit einem für uns unbekannten Namenspräfix, so dass diese Variable von einer anderen Übersetzungseinheit nicht sichtbar ist. Würden wir hingegen die Variable `i` aus dem unbenannten Namensraum in den globalen Namensraum verschieben, wäre das Programm fehlerfrei übersetzbar!

Das Beispiel mit der Klasse `Button` aus dem Namensraum `System::Windows::Controls` hat es bereits gezeigt: Die Namen von Namensräumen, vorzugsweise bei geschachtelten Namensräumen, können leicht unübersichtlich geraten, wenn der voll-quantifizierte Name eines Elements zu lang wird. Um den Zugriff auf die Elemente eines Namensraums einfach zu gestalten, setzt man deshalb Aliasnamen für Namensräume ein. Wir kehren wieder zum Beispiel mit der Klasse `Button` zurück, und definieren den Aliasnamen `MyControls` für den Namensraum `System::Windows::Controls`:

```
void main ()
{
    namespace MyControls = System::Windows::Controls;

    MyControls::Button b;
}
```

Kapitel 3. Überladen von Operatoren

3.1. Einführung

Mit dem Sprachmittel der Klasse bietet C++ einem Entwickler die Möglichkeit, eigene, benutzerdefinierte Datentypen definieren zu können. Um diese Datentypen wie vordefinierte C++-Datentypen (`int`, `double` usw.) benutzen zu können, sollte es möglich sein, C++-Operatoren (zum Beispiel `+`, `-`, `*`, `/` oder `++` und `--`) auch in benutzerdefinierten Klassen eine Bedeutung zuordnen zu können. Zu diesem Zweck gibt es das so genannte *Überladen von Operatoren* (engl. *operator overloading*).

Die Bedeutung eines Operators in einem C++-Programm hängt davon ab, auf welchen Objekttyp er angewendet wird. So kann `a+b` bedeuten, dass zwei `int`-Variablen zu addieren sind, wenn `a` und `b` `int`-Variablen sind. Genauso gut ist es möglich, dass `a+b` das Verketten zweier Zeichenketten bedeutet, wenn `a` und `b` zwei Instanzen der Klasse `String` sind. Ebenso könnte `a+b` die Addition zweier rationaler Zahlen bewirken, wenn `a` und `b` Objekte einer Klasse `Fraction` sind. In jedem Fall ist es um ein Vielfaches intuitiver, in einem Programm mit rationalen Zahlen `a`, `b` und `c`

```
Fraction a = b + c / 2;
```

zu schreiben, anstelle auf der Basis von Methodenaufrufen die korrespondierende Anweisungsfolge

```
Fraction a = b.Add (c.Div (2));
```

formulieren zu müssen.

Achtung

Die Definition eines Operators in einer Klasse sollte in ihrer Bedeutung immer intuitiv sein. Aus diesem Grund ist die Implementierung des Inkrement-Operators `++` in einer Klasse `Student` fragwürdig, da die Semantik der `++`-Operation auf einen Studenten angewendet nicht klar ist:

```
Student stud;
stud ++;  // ???
```

Was bedeutet es, „einen Studenten zu inkrementieren“? Es ist wichtig, darauf zu achten, dass vom Einsatz des Operatoren-Überladens nur in sinnvoller Weise Gebrauch gemacht wird. Eine nicht empfehlenswerte Verwendung wäre auch das Überladen des `*`-Operators in der Klasse `Fraction`, um zwei rationale Zahlen zu *dividieren*. Technisch gesehen ist dies möglich. Das Ziel, zu einem besser lesbaren Programm zu gelangen, wird dadurch aber ins Gegenteil verkehrt.

Neben den geschilderten Vorzügen besitzt die Technik des Operator-Überladens auch einige Einschränkungen:

- Operatorsymbol:

  Es ist nicht möglich, neue Operatorsymbole zu definieren. Im Gegenteil, in der Sprachdefinition von C++ ist festgelegt, welche Standardoperatoren überladbar sind und welche nicht (Tabelle 3.1):

Kategorie	Operatoren
	+ (Vorzeichenoperator), - (Vorzeichenoperator), ++ (Präfix- und Variante), -- (Präfix-und Post-Variante), () (Typkonvertierung

Unäre Operatoren	(Adress-Operator, Referenzierung), * (Dereferenzierung)
Arithmetische Operatoren	+ (Addition), – (Subtraktion), * (Multiplikation), / (Division), % (Modu
Selektion	–> (direkte Selektion), –>* (indirekte Selektion)
Schiebeoperatoren	<<, >>
Vergleichsoperatoren	==, !=, <, >, <=, >=
Bitweise Operatoren	& (Bitweises Und), ^ (Bitweises exklusives Oder), | (Bitweises Ode (Bitweise Negation)
Logische Operatoren	&& (Logisches Und), || (Logisches Oder), ! (Logisches Nicht)
Zuweisungsoperatoren	=, +=, –=, *=, /=, %=, &=, |=, ^=, <<=, >>=
Sonstige Operatoren	f(x) (Methodenaufruf), [] (indizierter Zugriff), ++, --, new, delet (Komma-Operator)

Tabelle 3.1. Überladbare Operatoren in C++.

Wie wir Tabelle 3.1 entnehmen können, sind die meisten der C-Operatoren überladbar. Ausnahmen stellen im wesentlichen der .-Operator (Selektion), der .*-Operator (indirekte Selektion), der Gültigkeitsbereichsoperator :: und der einzige ternäre Operator von C und C++, der Bedingungsoperator ?:.

- Operatorstelligkeit:

  Die Stelligkeit eines Operators ist durch die Technik des Überladens nicht änderbar. Es ist nicht möglich, einen unären Operator in einer benutzerdefinierten Klasse mit einer binären Stelligkeit zu redefinieren oder umgekehrt.

- Standardimplementierung:

  Die Arbeitsweise der Operatoren für elementare Standarddatentypen (int, double, usw.) ist nicht änderbar.

- Operatorenvorrang und Assoziativität:

  Der Operatorenvorrang und die Assoziativität eines Operators können beim Überladen nicht geändert werden. Sind für das sinnvolle Arbeiten mit einem überladenen Operator andere Einstellungen nötig, kann man mit Hilfe zusätzlicher Klammern Einfluss auf den Operatorenvorrang und seine Assoziativität nehmen.

3.2. Unäre und binäre Operatoren

3.2.1. Unäre Operatoren überladen

Wir kommen nun auf die technischen Details zum Überladen eines Operators zu sprechen. Grundsätzlich lassen sich dabei zwei Ansätze verfolgen:

1. Definition einer speziellen Instanzmethode:

   Für das Überladen des Operators in einer Klasse ist eine Methode zu definieren, deren Name sich aus dem Schlüsselwort `operator`, gefolgt vom Symbol des Operators zusammensetzt. Zwischen dem `operator`-Schlüsselwort und dem Operatorsymbol sind Leerzeichen zulässig, '`operator+`' und '`operator +`' sind zwei korrekte Schreibweisen eines derartigen Methodenbezeichners. Die Anzahl der Parameter dieser Methode muss gleich der Stelligkeit des Operators minus Eins sein. Der erste („linke") Operand wird immer durch das Objekt gestellt, auf das die Operatormethode angewendet wird. Für unäre Operatoren muss die Operatormethode folglich parameterlos sein. Binäre Operatoren führen zu einer Operatormethode mit einem Parameter, er repräsentiert den „rechten" Operanden.

2. Definition einer globalen Funktion:

   Alternativ zu Instanzmethoden können Operatoren auch durch eine globale Funktion definiert werden. Bezüglich der Namensgebung der globalen Funktion gelten dieselben Spielregeln wie in der ersten Variante. Da die Funktion in den allermeisten Fällen Zugriff zu den (privaten) Instanzvariablen des beteiligten Operandentyps benötigt, ist diese ein Freund des Operandentyps (`friend`). Ein Vorteil dieser Variante ist, dass die Anzahl der Parameter mit der Stelligkeit des Operators übereinstimmt.

Achtung

Es gibt einige wenige Operatoren, für die das Überladen nur in einer Variante möglich ist. Wir werden in den nachfolgenden Abschnitten gesondert auf diese Sonderfälle hinweisen.

Es gibt keine festen Regeln, welche der beiden Strategien bei einer konkreten Realisierung zu verfolgen ist. Soll die Ausführung eines bestimmten Operators den Zustand seines (seiner) Operanden ändern („*veränderbares Verhalten*"), spricht einiges für die Realisierung mit einer Instanzmethode, da diese von ihrem Wesen her prinzipiell zur Zustandsänderung eines Objekts animiert. Für Operatoren mit unveränderbarem Verhalten bietet sich im Gegensatz dazu eine globale Funktion an. In dieser Variante kommt als Vorteil hinzu, dass die automatische Typkonversion für alle Aktualparameter abläuft, auf die wir in XXX noch zu sprechen kommen. Um es noch einmal hervorzuheben: Diese Beobachtungen mögen als Richtschnur oder Leitfaden dienen, aus rein technischer Sicht sind beide Varianten gleichwertig.

Bei der Klasse `Fraction` bieten sich zwei unäre Operatoren an: Das unäre Minus zum Wechseln des Vorzeichens mit dem '`-`'-Operator und das Invertieren einer rationalen Zahl (Vertauschen von Zähler und Nenner), wir wählen die Tilde '`~`' als Operatorsymbol.

Um an einem Beispiel die zwei Strategien für das Operator-Überladen gegenüberstellen zu können, stellen wir beide Realisierungen für diese zwei Operatoren vor. Zuvor müssen wir allerdings festlegen, ob die Implementierung ihren Operanden verändern soll oder nicht. Der Ansatz des unveränderbaren Verhaltens erscheint mir sinnvoller.

Genug der Vorrede, in Listing 3.1 finden Sie die Deklaration der zwei Instanzmethoden '`operator-`' und '`operator~`' in der Klasse `Fraction` vor:

```
01: class Fraction
02: {
03: public:
04:     // unary arithmetic operators (definition inside class)
05:     Fraction operator- ();
06:     Fraction operator~ ();
07:
08:     ...
09: };
10:
11: // implementation of unary arithmetic operators
12: Fraction Fraction::operator- ()
13: {
14:     return Fraction (-m_num, m_denom);
15: }
16:
17: Fraction Fraction::operator~ ()
18: {
19:     return Fraction (m_denom, m_num);
20: }
```

Beispiel 3.1. Überladen unärer Operatoren (innerhalb der Klasse).

Mit Hilfe der Implementierung aus Listing 3.2 haben wir das Ziel erreicht, das Vorzeichen und das Invertieren rationaler Zahlen mit Operatorschreibweise durchführen zu können (Listing 3.2):

```
01: Fraction f (1, 2);
02: f = -f;
03: f.Print();
04: f = ~f;
05: f.Print();
```

Beispiel 3.2. Anwendung unärer Operatoren.

Die Ausgabe des Programms aus Listing 3.2 lautet:

```
-1/2
-2/1
```

Anmerkung

Die Anwendung eines überladenen Operators wird vom C++-Compiler auf den entsprechenden Aufruf einer Instanzmethode umgesetzt. Zur Verifikation dieser Behauptung können Sie spaßeshalber alternativ zur Operatorenschreibweise auch die entsprechende Instanzmethode in ihrer „klassischen“ Schreibweise aufrufen. Das Codefragment

```
Fraction f (1, 2);
f = f.operator-();
f.Print();
f = f.operator~();
f.Print();
```

ist übersetzungsfähig und funktionell identisch zum Codefragment aus Listing 3.2.

Die zweite Alternative im Überladen von Operatoren erfolgt außerhalb der Klasse mit geeigneten `friend`-Funktionen (Listing 3.3):

```
01: class Fraction
02: {
03:     // unary arithmetic operators (definition outside class)
04:     friend Fraction operator- (const Fraction&);
05:     friend Fraction operator~ (const Fraction&);
06:
07:     ...
08: };
09:
10: // implementation of unary arithmetic operators
11: Fraction operator- (const Fraction& f)
12: {
13:     return Fraction (-f.m_num, f.m_denom);
14: }
15:
16: Fraction operator~ (const Fraction& f)
17: {
18:     return Fraction (f.m_denom, f.m_num);
19: }
```

Beispiel 3.3. Überladen unärer Operatoren (außerhalb der Klasse).

Der Testrahmen aus Listing 3.2 ist unverändert mit dieser Variante übersetzungsfähig.

Tipp

Auch bei dieser Variante des Operatoren-Überladens können wir die Beobachtung machen, dass hinter den Kulissen der C++-Sprachoberfläche die Anwendung des überladenen Operators auf den entsprechenden Aufruf der globalen C++-Funktion umgesetzt wird. Dieses Mal sieht das Codefragment aus Listing 3.2 in seiner alternativen Formulierung so aus:

```
Fraction f (1, 2);
f = ::operator-(f);
f.Print();
f = ::operator~(f);
f.Print();
```

Prinzipiell könnte die zweite Variante auch mit globalen Funktionen umgesetzt werden, die *keine* Freunde der Klasse `Fraction` sind:

```
01: // unary arithmetic operators (definition outside class, no friend relationship)
02: Fraction operator- (const Fraction&);
03: Fraction operator~ (const Fraction&);
04: ...
05:
06: // implementation of unary arithmetic operators
07: Fraction operator- (const Fraction& f)
08: {
09:     return Fraction (-f.GetNum(), f.GetDenom ());
10: }
11:
12: Fraction operator~ (const Fraction& f)
13: {
14:     return Fraction (f.GetDenom (), f.GetNum());
15: }
```

Beispiel 3.4. Überladen unärer Operatoren außerhalb der Klasse ohne Freundschaftsbeziehung.

Wenn gleich diese Variante konzeptionell gleichwertig mit globalen Funktionen ist, die zum Freundeskreis der betroffenen Klasse zählen, erkennen wir in Listing 3.4 einen Nachteil: Da jeglicher Zugriff auf die privaten Elemente der `Fraction`-Klasse nicht mehr möglich ist, müssen wir in der Implementierung der Funktionen auf die öffentlichen „getter“-Methoden der Klasse zurückgreifen. Ob

der daraus tatsächlich resultierende Maschinencode ineffizienter ist als im Vergleich zu globalen `friend`-Funktionen, lässt sich nicht behaupten, da die Codegeneratoren moderner Compiler bei kleinen Funktionen immer auf den Kunstgriff der Codesubstitution zurückgreifen. An Stelle eines Funktionsaufrufs finden wir im Code den tatsächlichen Rumpf der Funktion eingefügt vor, so dass potentielle Laufzeitnachteile nicht vorhanden sind. Da das Konzept der Freunde gerade vor dem Hintergrund des Überladens von Operatoren entworfen wurde, ist eine Reduktion auf echt globale Funktionen (ohne Freundschaftsbeziehung) allerdings nicht notwendig.

3.2.2. Binäre Operatoren überladen

Nach den unären Operatoren kommen wir auf binäre Operatoren zu sprechen. Es stehen wie gehabt die zwei Strategien „Instanzmethode“ und „globale Funktion“ zur Auswahl. Exemplarisch gehen wir die zwei Operationen der Addition und der Subtraktion näher ein. Da bei der Strategie auf Basis von `friend`-Funktionen die Anzahl der Funktionsparameter und die Operatorstelligkeit übereinstimmen, gebe ich dieses Mal dieser Strategie den Vorzug.

Die Frage bezüglich der Veränderbarkeit der Operanden ist jetzt in der Entwurfsphase zu beantworten. In einem Ausdruck der Form

```
Fraction a, b, c;
...
a = b + c;
```

sollte die Ausführung des ‘+‘-Operators mit Sicherheit keine Auswirkung auf die beiden Operanden `b` und `c` nach sich ziehen. Wir legen folglich das Prinzip des unveränderbaren Verhaltens in den nachfolgenden Codefragmenten zu Grunde.

Nach diesen Vorüberlegungen kommen wir in Listing 3.5 auf die Deklaration und Implementierung des ‘+‘- und ‘−‘-Operators zu sprechen:

```
01: class Fraction
02: {
03:     // binary arithmetic operators (definition outside class)
04:     friend Fraction operator+ (const Fraction&, const Fraction&);
05:     friend Fraction operator- (const Fraction&, const Fraction&);
06:
07:     ...
08: };
09:
10: // implementation of binary arithmetic operators
11: Fraction operator+ (const Fraction& f1, const Fraction& f2)
12: {
13:     int num = f1.m_num * f2.m_denom + f1.m_denom * f2.m_num;
14:     int denom = f1.m_denom * f2.m_denom;
15:     return Fraction (num, denom);
16: }
17:
18: Fraction operator- (const Fraction& f1, const Fraction& f2)
19: {
20:     int num = f1.m_num * f2.m_denom - f1.m_denom * f2.m_num;
21:     int denom = f1.m_denom * f2.m_denom;
22:     return Fraction (num, denom);
23: }
```

Beispiel 3.5. Überladen binärer Operatoren (außerhalb der Klasse).

Das folgende Codefragment zeigt einige Möglichkeiten auf, die mit dieser Klassendefinition möglich sind:

```
void Demo ()
{
    Fraction a (1, 7);
```

```
    Fraction b (3, 7);
    Fraction c;

    c = a + b;
    c.Print ();

    c = a + a + a;
    c.Print ();

    c = a - b - a;
    c.Print ();
}
```

Die Ausgabe des letzten Codefragments lautet:

```
4/7
3/7
-3/7
```

Die Implementierung der weiteren binären Operatoren überlasse ich Ihnen zur Übung.

3.3. Überladung des Wertzuweisungsoperators

Wir haben bereits die Wertzuweisung von Objekten angesprochen. Der C++-Compiler legt in einer benutzerdefinierten Klasse – ohne weitere Eingriffe des Entwicklers – automatisch eine Standardimplementierung des Wertzuweisungsoperators an. Auf der anderen Seite gestattet C++, so wie es beim Standardkonstruktor, Kopierkonstruktor und Destruktor der Fall ist, auch eine explizite Definition dieses Operators. In manchen Klassen ist diese explizite Definition sogar unumgänglich, wie wir in diesem Kapitel noch sehen werden.

In Listing 3.6 starten wir einen ersten Versuch, den =-Operator explizit in der Klasse `Fraction` zu überladen:

```
01: class Fraction
02: {
03: public:
04:     // assigment operator
05:     void operator= (const Fraction&);
06:
07:     ...
08: };
09:
10: // assigment operator
11: void Fraction::operator= (const Fraction& f)
12: {
13:     m_num = f.m_num;
14:     m_denom = f.m_denom;
15: }
```

Beispiel 3.6. Überladung des Wertzuweisungsoperators: Erster Versuch.

Tipp

Im Gegensatz zu den meisten anderen überladbaren Operatoren kann der Wertzuweisungsoperator nur *innerhalb* der Klasse (mit einer Instanzmethode) überladen werden. Seine schematische Definition in einer fiktiven Klasse `Type` lautet

```
class Type
{
public:
    Type& operator= (const Type& type);  // assignment operator
};
```

Die alternative Variante durch eine globale (`friend`-)Funktion entfällt beim Wertzuweisungsoperator. Auf diese Weise wird sichergestellt, dass es sich beim linken Operand immer um einen so genannten *lvalue* handelt. Hiermit soll ausgedrückt werden, dass dem linken Operanden ein Wert zuweisbar ist, d.h. der Operand muss für einen Speicherplatz (ein Objekt) stehen, das modifizierbar ist.

Die Implementierung dürfte auf den ersten Blick unseren Erwartungen entsprechen, es werden alle Instanzvariablen des Objekts auf der rechten Seite der Wertzuweisung (durch Aktualparameter `f` spezifiziert) in das Objekt auf der linken Seite umkopiert (Objekt, an dem der Aufruf der Instanzmethode erfolgt). Das folgende Minimalbeispiel

```
Fraction a (1, 2);
Fraction b (3, 2);
a = b;
```

ist sowohl übersetzungsfähig, der Wert `3/2` von `a` nach der Wertzuweisung überzeugt uns von der Korrektheit der Implementierung. Trotzdem haben wir ein kleines Problem übersehen. C gestattet neben der einfachen Wertzuweisung auch die Verkettung mehrerer Zuweisungen, also beispielsweise eine Anweisung in der Form

```
int a, b, c;
a = b = c = 1;
```

In dieser Anweisung wird der Wert `1` zunächst der Variablen `c` zugewiesen, dann der Variablen `b` und schließlich `a`. Wollten wir diese Zuweisung mit rationalen Zahlen formulieren, erhalten wir die Fehlermeldung *„Error: binary '=' : no operator found which takes a right-hand operand of type 'void'"*. Der Punkt ist, dass unsere aktuelle Implementierung des =-Operators keinen Rückgabewert besitzt. Würde sie eine Referenz des aktuellen Objekts, an dem die Wertzuweisung ausgeführt wird, zurückliefern, könnte man mehrere Wertzuweisungen miteinander verketten. Die korrekte Definition und Implementierung des Wertzuweisungsoperators sieht somit wie in Listing 3.7 gezeigt aus:

```
01: class Fraction
02: {
03: public:
04:     // assigment operator
05:     Fraction& operator= (const Fraction&);
06:     ...
07: };
08:
09: // assigment operator
10: Fraction& Fraction::operator= (const Fraction& f)
11: {
12:     m_num = f.m_num;
13:     m_denom = f.m_denom;
14:     return *this;
15: }
```

Beispiel 3.7. Korrekte Überladung des Wertzuweisungsoperators.

Beachten Sie Zeile 14 von Listing 3.7: Damit mehrere Aufrufe des =-Operators miteinander verknüpfbar sind, muss ein Aufruf der implementierenden Methode die Referenz des Objekts zurückliefern, an dem der Aufruf erfolgt, also das durch `this` referenzierte Objekt. Da die Schnittstellendefinition keinen Zeiger, sondern eine Referenz des aktuellen Objekts erwartet, lautet der korrekte Rückgabewert `*this`.

3.4. Wertzuweisung versus Initialisierung

Der Gebrauch des =-Operators bei der Erzeugung eines neuen Objekts mit Hilfe des Kopierkonstruktors lässt immer wieder die Vermutung aufkommen, dass hier der – potentiell überladene – Wertzuweisungsoperator zum Einsatz kommt. Dies ist nicht der Fall, die Zuweisung eines existierenden Objekts an ein anderes sowie die Erzeugung eines neuen Objekts mit Hilfe des Kopierkonstruktors sind zwei grundsätzlich verschiedene Aktionen. Eine *Initialisierung* geht immer Hand in Hand mit der Erzeugung eines *neuen* Objekts. Eine *Zuweisung* auf der anderen Seite geschieht immer zwischen zwei bereits *vorhandenen* Objekten. Eine Initialisierung der Gestalt

```
Fraction f;
...
Fraction g = f;  // copy c'tor
```

ist niemals eine Zuweisung und demzufolge wird hier auch niemals der Zuweisungsoperator des Klassentyps `Fraction` aufgerufen. Bei einer Wertzuweisung, wie etwa der Gestalt

```
Fraction f;
Fraction g;
...
g = f;  // assignment
```

kommt entweder der vom System bereitgestellte Standardwertzuweisungsoperator zum Zuge oder aber der benutzerdefinierte Wertzuweisungsoperator, sofern dieser im fraglichen Klassentyp definiert ist.

Achtung

Der Kopierkonstruktor kommt nicht zum Einsatz, wenn für die Erzeugung eines neuen Objekts ein benutzerdefinierter Konstruktor herangezogen wird. Im Beispiel

```
Fraction f = Fraction (1, 2);
```

wird zwar ein temporäres Objekt erzeugt und in das neu zu erzeugende Objekt umkopiert, der Kopierkonstruktor bleibt dabei aber außen vor. Syntaktisch präziser sollte man diese Anweisung mit

```
Fraction f (1, 2);
```

formulieren, da dann der Einsatz des benutzerdefinierten Konstruktors mit zwei `int`-Parametern besser erkennbar ist.

3.5. Überladung der arithmetischen Wertzuweisungsoperatoren

In Ergänzung zu den arithmetischen Operatoren und dem Wertzuweisungsoperator kennt C auch das Zusammenspiel dieser beiden Operatoren, die so genannten *arithmetischen Wertzuweisungsoperatoren*. Mit ihrer Hilfe können Anweisungen der Gestalt

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

auch kompakter durch

```
a += b;
```

programmiert werden. In C++ können die arithmetischen Wertzuweisungsoperatoren auch explizit überladen werden. Dabei ist – wie beim Wertzuweisungsoperator – zu beachten, dass arithmetische Wertzuweisungen auch schachtelbar sind. Eine Anweisung der Form

```
a += b += c;
```

ist also ebenfalls korrekt. Mit diesen Hinweisen können wir direkt zur Ergänzung der Klasse `Fraction` um den `+=`- und `-=`-Operator in Listing 3.8 überleiten:

```
01: class Fraction
02: {
03:     // addition-assignment operators (definition outside class)
04:     friend const Fraction& operator+= (Fraction&, const Fraction&);
05:     friend const Fraction& operator-= (Fraction&, const Fraction&);
06:
07:     ...
08: };
09:
10: // addition-assignment operators
11: const Fraction& operator+= (Fraction& f1, const Fraction& f2)
12: {
13:     f1.m_num = f1.m_num * f2.m_denom + f1.m_denom * f2.m_num;
14:     f1.m_denom = f1.m_denom * f2.m_denom;
15:     f1.Reduce ();
16:     return f1;
17: }
18:
19: const Fraction& operator-= (Fraction& f1, const Fraction& f2)
20: {
21:     f1.m_num = f1.m_num * f2.m_denom - f1.m_denom * f2.m_num;
22:     f1.m_denom = f1.m_denom * f2.m_denom;
23:     f1.Reduce ();
24:     return f1;
25: }
```

Beispiel 3.8. Überladung der arithmetischen Wertzuweisungsoperatoren += und -=.

Auch wenn es vielleicht selbstverständlich klingen mag: Bei der Realisierung der arithmetischen Wertzuweisungsoperatoren ist darauf zu achten, dass die unterschiedlichen Operatoren einer Klasse sich zueinander konsistent verhalten. Darunter verstehen wir, dass beispielsweise die beiden Anweisungen

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

dasselbe Resultat erzielen, obwohl verschiedene Operatoren zur Ausführung gelangen. Wir testen unsere Implementierung an folgendem Beispielszenario:

```
void Demo ()
{
    Fraction a (1, 7);
    Fraction b (3, 7);
    Fraction c (5, 7);

    a += b += c;
    a.Print ();
    b.Print ();
    c.Print ();

    c -= b -= a;
    a.Print ();
    b.Print ();
    c.Print ();
}
```

Die Ausgabe des letzten Codefragments lautet:

```
9/7
8/7
5/7
9/7
-1/7
6/7
```

Beziehen wir in die Implementierung der arithmetischen Wertzuweisungsoperatoren die zu Grunde liegenden arithmetischen Operatoren mit ein, können wir die Realisierung kompakter fassen:

```
01: const Fraction& operator+= (Fraction& f1, const Fraction& f2)
02: {
03:     f1 = f1 + f2;
04:     f1.Reduce ();
05:     return f1;
06: }
07:
08: const Fraction& operator-= (Fraction& f1, const Fraction& f2)
09: {
10:     f1 = f1 - f2;
11:     f1.Reduce ();
12:     return f1;
13: }
```

Beispiel 3.9. Überladung der arithmetischen Wertzuweisungsoperatoren += und -=, 2. Variante.

3.6. Überladung der Vergleichsoperatoren

Das Überladen der Vergleichsoperatoren kann in Analogie zu den arithmetischen Operatoren erfolgen. Insgesamt fallen in diese Kategorie sechs Operatoren (<, <=, >, >=, == und !=), allerdings sollte sich eine geschickte Implementierung zahlreiche Synergieeffekte zu Nutze machen. In Listing 3.10 können wir erkennen, dass tatsächlich nur drei Operatoren im eigentlichen Sinne implementiert sind, die verbleibenden drei werden auf vorhandene Implementierungen zurückgeführt:

```
01: class Fraction
02: {
03:     // comparison operators
04:     friend bool operator<= (const Fraction&, const Fraction&);
05:     friend bool operator<  (const Fraction&, const Fraction&);
06:     friend bool operator>= (const Fraction&, const Fraction&);
07:     friend bool operator>  (const Fraction&, const Fraction&);
08:     friend bool operator== (const Fraction&, const Fraction&);
09:     friend bool operator!= (const Fraction&, const Fraction&);
10:
11:     ...
12: };
13:
14: // comparison operators
15: bool operator<= (const Fraction& f1, const Fraction& f2)
16: {
17:     return f1.m_num * f2.m_denom <= f1.m_denom * f2.m_num;
18: }
19:
20: bool operator< (const Fraction& f1, const Fraction& f2)
21: {
22:     return f1.m_num * f2.m_denom < f1.m_denom * f2.m_num;
23: }
24:
25: bool operator>= (const Fraction& f1, const Fraction& f2)
26: {
27:     return ! (f1 < f2);
28: }
29:
30: bool operator> (const Fraction& f1, const Fraction& f2)
31: {
32:     return ! (f1 <= f2);
33: }
34:
35: bool operator== (const Fraction& f1, const Fraction& f2)
36: {
37:     return f1.m_num * f2.m_denom == f1.m_denom * f2.m_num;
38: }
39:
40: bool operator!= (const Fraction& f1, const Fraction& f2)
41: {
42:     return ! (f1 == f2);
43: }
```

Beispiel 3.10. Überladung der Vergleichsoperatoren.

Etwaige Überlegungen in punkto „Veränderbarkeit des Objektzustands“ sind bei den Vergleichsoperatoren natürlich obsolet. Der Vergleich zweier Objekte stellt eine Bedingung dar und zieht ein Resultat vom Typ `bool` nach sich, die beteiligten Objekte erfahren trivialerweise keine Änderung ihres Zustands.

3.7. Typumwandlungen

3.7.1. Typumwandlungen mit Operatoren

Elementare Standarddatentypen von C/C++ wie `int`, `double` usw. und benutzerdefinierte Datentypen stehen häufig in einer engen Beziehung zueinander. Die ganzen Zahlen `1`, `2`, `3`,... könnten beispielsweise auch als rationale Zahlen `1/1`, `2/1`, `3/1`,... interpretiert werden. Damit ist ein Codefragment

```
f = f + 1;
```

mit einem Objekt `f` der Klasse `Fraction` nicht nur sinnvoll, sondern auch wünschenswert. Ein Ansatz, um den C++-Compiler zur Übersetzung derartiger Ausdrücke zu überreden, besteht darin, die Menge der Überladungen des +-Operators entsprechend zu erweitern. Da wir dann auch eine Anweisung der Gestalt

```
f = 1 + f;
```

unterstützen sollten, führen die zwei Operatoren + und − gleich zu vier zusätzlichen Deklarationen:

```
friend Fraction operator+ (const Fraction&, int n);
friend Fraction operator+ (int n, const Fraction&);
friend Fraction operator- (const Fraction&, int n);
friend Fraction operator- (int n, const Fraction&);
```

Listing 3.11 demonstriert eine mögliche Implementierung:

```
01: // additional binary arithmetic operators
02: Fraction operator+ (const Fraction& f, int n)
03: {
04:     return Fraction (f.m_num + n * f.m_denom, f.m_denom);
05: }
06:
07: Fraction operator+ (int n, const Fraction& f)
08: {
09:     return f + n;
10: }
11:
12: Fraction operator- (const Fraction& f, int n)
13: {
14:     return Fraction (f.m_num - n * f.m_denom, f.m_denom);
15: }
16:
17: Fraction operator- (int n, const Fraction& f)
18: {
19:     return Fraction (n * f.m_denom - f.m_num, f.m_denom);
20: }
```

Beispiel 3.11. Vervollständigung der Überladungen binärer Operatoren.

Die Implementierung der zusätzlichen binären Operatoren ist vergleichsweise einfach, nichtsdestotrotz stellt die Vielzahl der überladenen Operatoren einen Nachteil dar. Eine Abhilfe ist durch das Einführen zusätzlicher Konstruktoren möglich, worauf wir im nächsten Abschnitt eingehen werden.

3.7.2. Typumwandlungen mit Konstruktoren

Ein Typumwandlungskonstruktor nimmt die Umwandlung anderer (elementarer oder

benutzerdefinierter) Datentypen in einen benutzerdefinierten Datentyp vor. Wir erhalten auf diese Weise Vorschriften, wie Werte eines bestimmten Typs erzeugbar sind. Wird ein bestimmter Typ benötigt, und gibt es einen Konstruktor, der aus dem vorliegenden Typ den geforderten Typ erzeugt, kommt dieser Konstruktor automatisch zum Einsatz. Wir sprechen in dieser Situation von einer *impliziten Typumwandlung*, sie findet beispielsweise bei der Übergabe von Parametern an Funktionen (Methoden), in Wertzuweisungen und bei der Bildung von Rückgabewerten in Funktionen (Methoden) statt.

Als Beispiel ergänzen wir die `Fraction`-Klasse in Listing 3.12 um einen Typumwandlungskonstruktor, der die Wandlung eines ganzzahligen `int`-Werts in eine rationale Zahl durchführt:

```
01: class Fraction
02: {
03: public:
04:     // conversion c'tor
05:     Fraction (int);
06:
07:     ...
08: };
09:
10: // conversion c'tor
11: Fraction::Fraction (int num)
12: {
13:     m_num = num;
14:     m_denom = 1;
15: }
```

Beispiel 3.12. Typumwandlungskonstruktor `int` nach `Fraction`.

Neben einfachen Wertzuweisungen in der Form

```
Fraction a;
...
a = 1;
```

können nun alle binären arithmetischen Operatoren entfallen, die einen `int`-Parameter besitzen. In Anweisungen wie

```
Fraction a;
...
a = a + 1;
a = 1 + a;
```

wird jeder Ausdruck vom Typ `int` mit Hilfe des passenden Typumwandlungskonstruktors in ein `Fraction`-Objekt umgewandelt, so dass im Anschluss daran der binäre Additionsoperator (mit zwei `Fraction`-Operanden) aufgerufen werden kann.

3.7.3. Explizite Typumwandlungen

Implizite Typkonvertierungen mit Hilfe von Konstruktoren können in manchen Fällen auch unerwünscht sein oder zu Interpretationen des Übersetzers führen, die nicht im Sinne des Anwenders sind. Sehen wir uns zu diesem Zweck den Entwurf einer Klasse `String` für Zeichenketten an. Zur Verwaltung eines internen Puffers für die einzelnen Zeichen empfiehlt sich ein Konstruktor, mit dem man die Puffergröße einstellen kann. Kennt man von einer Zeichenkette deren maximale Länge, so kann man auf diese Weise verhindern, dass zur Laufzeit beim Überschreiten der aktuellen Puffergröße ein neuer Puffer angelegt werden muss und so unnötige Kopieraktionen stattfinden. Das folgende Codefragment beschreibt eine Klasse `String`, die in der Instanzvariablen `m_cardinality` die Puffergröße verwaltet:

```
class String
{
private:
    int m_cardinality;

public:
    // c'tors
    String (int);

    ...
};

String::String (int cardinality)
{
    m_cardinality = cardinality;
}
```

Problematisch an diesem Design ist, dass für eine Instanz `s` der Klasse `String` der C++-Compiler auch zum Beispiel folgende Anweisung übersetzt:

```
s = 'A';
```

Für einen Entwickler, der den Quellcode oder die Dokumentation der `String`-Klasse nicht verfügbar hat, drängt sich hier der Eindruck auf, dass das Objekt `s` eine Zeichenkette der Länge 1 zugewiesen bekommen soll, dessen erstes (und einziges) Zeichen ein `'A'` ist. Die Wirklichkeit sieht aber ganz anders aus. Zunächst wird die `char`-Konstante `'A'` durch eine Standardkonvertierung in ihr ganzzahliges Pendant konvertiert (Datentyp `int`). Anschließend wird bei einem Aufruf des Konstruktors ein interner Puffer der Länge `'A'` angelegt!

Zur Vermeidung derartiger Missverständnisse zwischen Entwickler und Compiler gibt es das `explicit`-Schlüsselwort. Versieht man einen Typumwandlungskonstruktor mit `explicit`, wird erreicht, dass der Konstruktor nur noch explizit eingesetzt werden kann. Anweisungen, die auf die implizite Unterstützung des Typumwandlungskonstruktors setzen, werden mit einer Fehlermeldung abgewiesen:

```
String s1;
s1 = '1';        // Error: binary '=' : no operator found which takes
                 // a right-hand operand of type 'char'
                 // (or there is no acceptable conversion)

String s2 (64);  // o.k.
```

Damit das letzte Codefragment vom C++-Compiler wie in den Kommentaren beschrieben übersetzt wird, muss die Definition der `String`-Klasse jetzt so aussehen:

```
class String
{
private:
    int m_cardinality;

public:
    // c'tors
    String ();
    explicit String (int);
};
```

3.7.4. Konvertierung mit Operatoren

Die bislang vorgestellten Typumwandlungskonstruktoren hatten als Ziel der Umwandlung einen benutzerdefinierten Datentyp. Es sind aber auch Umwandlungen in der entgegengesetzten Richtung möglich. Damit könnte man zum Beispiel `Fraction`-Objekte in den Typ `double` umwandeln. Diese

Aufgabe obliegt den so genannten *Konvertierungsoperatoren*.

Ein Konvertierungsoperator besitzt im Gegensatz zu den anderen überladenen Operatoren eine leicht abgewandelte Syntax. Sein Name besteht aus dem Schlüsselwort `operator` gefolgt von einem Typnamen, danach schließt sich eine leere Parameterliste an. In Listing 3.13 ist ein Konvertierungsoperator aufgeführt, um rationale Zahlen in Gleitpunktwerte zu wandeln:

```
01: class Fraction
02: {
03: public:
04:     // type conversion operator (Fraction -> double)
05:     operator double ();
06:
07:     ...
08: };
09:
10: // conversion operator (Fraction -> double)
11: Fraction::operator double ()
12: {
13:     return (double) m_num / (double) m_denom;
14: }
```

Beispiel 3.13. Konvertierungsoperator `Fraction` nach `double`.

Beachten Sie in Zeile 5 von Listing 3.13: Der Typ des Konvertierungsoperators wird implizit durch den Namen des Operators festgelegt, eine explizite Angabe vor dem `operator`-Schlüsselwort wird mit einem Fehler abgewiesen:

```
double operator double ();  // ERROR: user-defined conversion
                            // cannot specify a return type
```

Mit Hilfe des vorgestellten Konvertierungsoperators von `Fraction` nach `double` ist nun das folgende Codefragment übersetzungsfähig:

```
Fraction f (1, 7);
double d = f;
printf ("d: %g\n", d);
```

Die Ausführung liefert das korrekte Ergebnis:

```
d: 0.142857
```

3.7.5. Mehrdeutigkeiten bei Typumwandlungen

Ist eine Klasse sowohl im Besitz von Konstruktoren als auch Operatoren zur Typumwandlung, können leicht Mehrdeutigkeiten entstehen. Wir kommen zur Erläuterung eines Beispiels wieder auf die Klasse `Fraction` zurück, dieses Mal mit der Definition aus Listing 3.14:

```
01: class Fraction
02: {
03: private:
04:     // private member data
05:     int m_num;    // numerator
06:     int m_denom;  // denominator
07:
08: public:
09:     // c'tors
10:     Fraction ();
11:     Fraction (int, int);
12:
```

```
13:     // conversion c'tor (int -> Fraction)
14:     Fraction (int);
15:
16:     // type conversion operator (Fraction -> double)
17:     operator double ();
18:
19:     // binary arithmetic operators (definition outside class)
20:     friend Fraction operator+ (const Fraction&, const Fraction&);
21:     friend Fraction operator- (const Fraction&, const Fraction&);
22:
23:     ...
24: };
```

Beispiel 3.14. Klasse `Fraction` mit Typumwandlungskonstruktor und -operator.

Auf Basis der Klassendefinition von Listing 3.14 sind nun Ausdrücke formulierbar, die für den Compiler nicht eindeutig übersetzbar sind. Der Ausdruck

```
f + 1;
```

kann einerseits als Addition zweier rationaler Zahlen interpretiert werden, wenn der zweite Operand `1` vor der Anwendung des +-Operators mit Hilfe des Typumwandlungskonstruktors in ein `Fraction`-Objekt umgewandelt wird. Alternativ könnte man den ersten Operanden `f` mit Hilfe des Typumwandlungsoperators in einen `double`-Wert konvertieren und anschließend zwei `double`-Werte addieren. Um in solchen Fällen die Fehlermeldung *„'operator+' : 2 overloads have similar conversions"* des Compilers zu umgehen, muss man mit expliziten Typumwandlungen für Klarheit sorgen:

```
f + (Fraction) 1;
```

oder

```
(double) f + 1;
```

3.8. „The Big Three“: Kopierkonstruktor, Wertzuweisungsoperator und Destruktor

Die bislang im Mittelpunkt stehende Klasse `Fraction` ist vergleichsweise einfach strukturiert: Sie verwaltet in ihren Instanzvariablen keinerlei Daten, die dynamisch zu allokieren sind. Wir kommen in diesem Abschnitt auf die Besonderheiten in der Implementierung einer Klasse zu sprechen, die zur Verwaltung ihrer Zustandsdaten den `new`-Operator benötigt.

Zeichenketten im Stile von C (Array von `char`-Elementen mit einem ASCII-Nullwert am Ende) sind in ihrer Hantierung ziemlich umständlich und fehleranfällig. Ist beispielsweise beim Kopieren einer C-Zeichenkette mit der C-Runtime-Systemfunktion

```
char* strcpy (char* dest, const char* src);
```

der Zielpuffer `dest` kleiner wie die zu kopierende Zeichenkette `src`, wird beim Kopieren nicht zur Verfügung stehender Speicher unkontrolliert überschrieben. In einem objektorientierten Umfeld ist die Existenz einer Klasse für Zeichenketten unabdingbar, um ihre Verarbeitung zu vereinfachen und sicherer zu machen. Wir stellen im folgenden die ersten Überlegungen zu einer Klasse `String` vor, wie sie in der C++-Standardbibliothek vorhanden ist.

Die Leistungsfähigkeit unserer Klasse `String` ist vergleichsweise einfach und umfasst nur einen kleinen Teil der im Standard vorgesehenen Funktionalität. Zunächst müssen wir festlegen, wie in unserer Realisierung der `String`-Klasse eine Zeichenkette im Speicher ausgebreitet werden soll. Da Zeichenketten unterschiedlich lang sein können, sind statische Datenstrukturen wie ein Array ungeeignet. Es liegt nahe, statt dessen einen Pointer vom Typ `char` zu verwenden, der mit dem `new`-Operator den jeweils benötigten Speicherplatz einer Zeichenkette zugewiesen bekommt. Die aktuelle Länge der Zeichenkette legen wir in einer zweiten Instanzvariablen ab. Da wir zu Demonstrationszwecken bei der Implementierung ohne die C-Runtime-Systembibliothek auskommen wollen, legen wir ferner fest, dass die Zeichenkette kein terminierendes Null-Zeichen besitzt. Abbildung 3.1 zeigt auf der Basis dieser Vorüberlegungen exemplarisch eine Instanz der Klasse `String`.

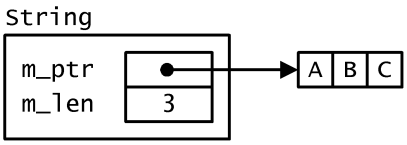

Abbildung 3.1. Ein Objekt der Klasse `String` zur Beschreibung der Zeichenkette „ABC“.

Der mit dem `new`-Operator reservierte Speicherplatz muss natürlich wieder freigegeben werden, wenn das Objekt nicht mehr benötigt wird. Diese Aufgabe obliegt dem Destruktor der `String`-Klasse. Wir erkennen schlagartig, dass die Bedeutung des Destruktors bei Klassen mit dynamischen Daten um ein Vielfaches größer wird! In Listing 3.15 stellen wir die erste Version für die Deklaration der Klasse `String` vor:

```
01: class String
02: {
03: private:
04:     char* m_ptr;  // buffer
05:     int  m_len;   // buffer length
06:
07: public:
08:     // c'tors and d'tor
09:     String ();
10:     String (const char*);
11:     ~String ();
```

```
12: };
```

Beispiel 3.15. Schnittstelle der Klasse `String` (1. Version).

Eine gangbare Implementierung der Klassenschnittstelle aus Listing 3.15 finden wir in Listing 3.16 vor:

```
01: String::String ()
02: {
03:     // empty string
04:     m_len = 0;
05:     m_ptr = (char*) 0;
06: }
07:
08: String::String (const char * s)
09: {
10:     // length of string
11:     m_len = 0;
12:     while (s[m_len] != '\0')
13:         m_len ++;
14:
15:     // allocate buffer
16:     m_ptr = new char[m_len];
17:
18:     // copy argument
19:     for (int i = 0; i < m_len; i ++)
20:         m_ptr[i] = s[i];
21: }
22:
23: String::~String ()
24: {
25:     delete[] m_ptr;
26: }
```

Beispiel 3.16. Implementierung der Klasse `String` (1. Version).

Zu unserer Überraschung müssen wir feststellen, dass die ersten Anwendungen mit unserer Implementierung bereits bei einem so einfachen Codefragment wie

```
void Demo ()
{
    String s1 ("ABC");
    String s2 ("12345");
    s1 = s2;
}
```

versagen. Das Laufzeitsystem vermeldet einen Fehler „*Debug Assertion Failure*“ (Abbildung 3.2).

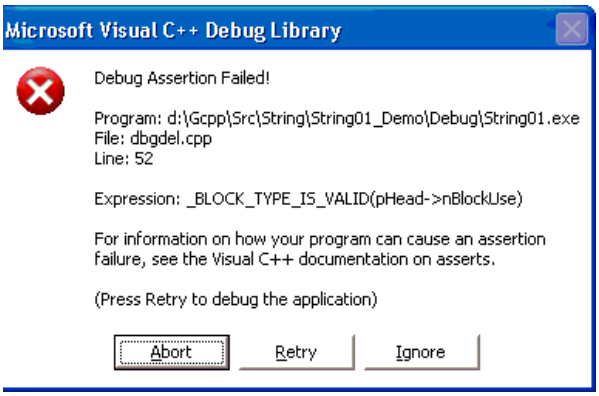

Abbildung 3.2. Typischer Laufzeitfehler in einem C++-Programm mit dynamisch allokierten Daten.

Erkennen Sie das Problem? Die Anweisung

```
s1 = s2;
```

ist sowohl übersetzungs- – wie auch lauffähig, allerdings verwendet sie den Zuweisungsoperator in der vom Compiler automatisch generierten Standardversion. Die dieser Variante zu Grunde liegende Implementierungsstrategie firmiert unter der Bezeichnung *flache Kopie* (engl. *shallow copy*). Bei ihr werden alle Instanzvariablen des Originalobjekts in das neue Objekt einfach umkopiert! Handelt es sich dabei um Variablen elementaren Datentyps (wie `int`, `double`, usw.), wird der Wert bitweise kopiert. Im Falle einer Zeigervariablen wird der Zeiger kopiert (wenn Sie so wollen: ebenfalls bitweise), aber *nicht* der Speicherbereich, auf den der Zeiger verweist. Dies hat zur Folge, dass die Zeigervariablen im Originalobjekt und im kopierten Objekt auf denselben Speicherbereich verweisen (siehe Abbildung 3.3), eine in den meisten Fällen unpassende Eigenschaft einer Objektkopie.

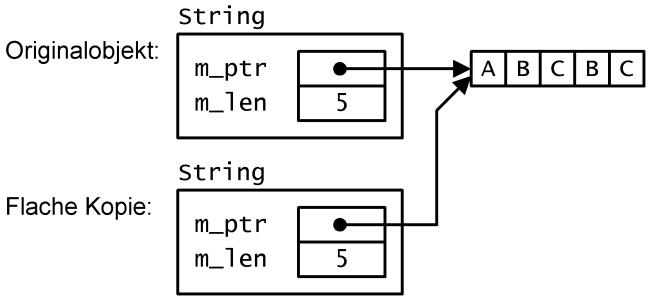

Abbildung 3.3. Strategie des flachen Kopierens für `String`-Objekte.

Auf das Problem des Programmabsturzes stoßen wir nun, wenn die diskutierte `Demo`-Methode verlassen wird. Es gelangt an beiden `String`-Objekten der Destruktor zur Ausführung. Da die `m_ptr`-Instanzvariable in beiden Objekten auf denselben Speicherbereich zeigt, kommt es zu dem Versuch, einen einmalig mit `new` angelegten Speicherbereich *zweimal* freizugeben. Der zweite Aufruf des `delete`-Operators führt zum Absturz des Programms!

Bei der Strategie des *tiefen Kopierens* wird bei allen Zeigervariablen eine Kopie des referenzierten Speicherbereichs angelegt. Dieser Vorgang wird rekursiv so lange wiederholt, bis für alle Unterobjekte des Originals neue Speicherbereiche in der Objektkopie angelegt sind. Original und Kopie bestehen folglich in dieser Variante komplett aus unterschiedlichen Speicherbereichen, siehe Abbildung 3.4.

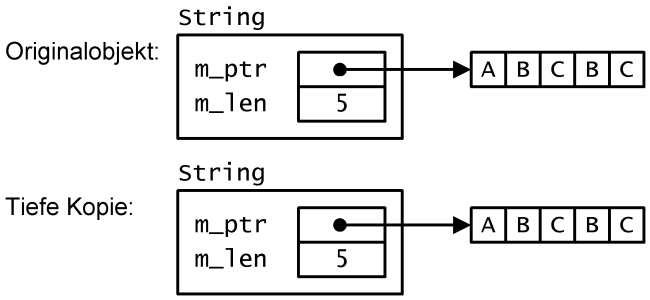

Abbildung 3.4. Strategie des tiefen Kopierens für `String`-Objekte.

Die Strategie des tiefen Kopierens kann kein Compiler für uns automatisch generieren, wir müssen diese selbst implementieren. In Bezug auf unsere `String`-Klasse heißt das, dass die benutzerdefinierte Realisierung des Wertzuweisungsoperators zwingend erforderlich ist:

```
// assignment operator
String& String::operator= (const String& s)
{
    // delete old string
    delete[] m_ptr;

    // allocate new buffer
    m_len = s.m_len;
    m_ptr = new char[m_len];

    // deep copy
    for (int i = 0; i < m_len; i ++)
        m_ptr[i] = s.m_ptr[i];

    return *this;
```

```
}
```

In der Implementierung des Wertzuweisungsoperators muss daran gedacht werden, dass die Daten des Objekts, das auf der linken Seite steht, zuerst freigegeben werden. Nach der Freigabe kann dann der erforderliche Speicherbereich für die neuen Daten allokiert werden, um die Daten des Objekts, das auf der rechten Seite steht, im Sinne der tiefen Kopierstrategie umzukopieren.

Die vorgestellte Realisierung ist prinzipiell fehlerfrei, lässt aber außer Acht, dass die Zuweisung eines Objekts an sich selbst zwar keinen Sinn ergibt, syntaktisch aber zulässig ist:

```
s1 = s1;
```

In diesem Fall würde der Aufruf des `delete`-Operators die Daten des aktuellen Objekts vor dem eigentlichen Kopieren selbst zerstören! Aus diesem Grund wird der `this`-Zeiger des Objekts, das auf der linken Seite steht, zuerst mit der Adresse des Objekts auf der rechten Seite verglichen. Sind die Adressen gleich, handelt es sich auf beiden Seiten der Wertzuweisung um dasselbe Objekt und es ist nichts zu tun:

```
01: // assignment operator
02: String& String::operator= (const String& s)
03: {
04:     if (this != &s)
05:     {
06:         // delete old string
07:         delete[] m_ptr;
08:
09:         // allocate new buffer
10:         m_len = s.m_len;
11:         m_ptr = new char[m_len];
12:
13:         // deep copy
14:         for (int i = 0; i < m_len; i ++)
15:             m_ptr[i] = s.m_ptr[i];
16:     }
17:
18:     return *this;
19: }
```

Beispiel 3.17. Implementierung des Wertzuweisungsoperators in der Klasse `String` (2. Version).

Wir halten als erstes Zwischenergebnis fest, dass in Klassen, die dynamisch allokierte Daten besitzen, die Wertzuweisung von Objekten mit dem automatisch generierten Zuweisungsoperator nicht korrekt abläuft! Es wird die Strategie des flachen Kopierens verfolgt, bei der Freigabe derartiger Objekte resultieren Programmabstürzen bei den Destruktoraufrufen.

Wir kommen auf eine zweite Anomalie zu sprechen und ergänzen zu diesem Zweck unsere `String`-Klasse um eine Methode `ToUpperCase`:

```
String String::ToUpperCase ()
{
    char* buffer = new char[m_len + 1];
    for (int i = 0; i < m_len; i++)
        buffer[i] = (m_ptr[i] >= 'a' && m_ptr[i] <= 'z') ?
            m_ptr[i] + ('A' - 'a') :
            m_ptr[i];
    buffer[m_len] = '\0';

    String s(buffer);
    delete[] buffer;
    return s;
}
```

`ToUpperCase` wandelt in einer `String`-Instanz Buchstaben mit Kleinschreibung in Großschreibung

um. Trotz fehlerfreier Implementierung stürzt die einfache Testmethode

```
void Demo ()
{
    String s1("abcdef");
    String s2;
    s2 = s1.ToUpperCase();
}
```

wiederum mit der in Abbildung 3.2 gezeigten Fehlermeldung ab.

In der Tat ist die Fehlerursache dieses Mal nicht ohne weiteres erkennbar. Da der `delete`-Operator ein Kandidat für den Absturz sein könnte, versehen wir zur Fehlersuche den beteiligten Konstruktor und den Destruktor mit entsprechenden Testausgaben. Wir stoßen bei der Ausführung des Programms auf eine Überraschung:

```
c'tor (char*)
c'tor ()
c'tor (char*)
d'tor
d'tor
d'tor
d'tor
```

Wir finden zunächst drei Konstruktorenaufrufe vor, deren Ursprung wir nachvollziehen können: In der Testroutine `Demo` werden zwei `String`-Objekte erzeugt, zusätzlich legt die `ToUpperCase`-Methode ein lokales `String`-Objekt an. Wie lassen sich aber *vier* Destruktorenaufrufe erklären, wenn nur drei `String`-Objekte erzeugt werden? Wir betrachten zu diesem Zweck die `ToUpperCase`-Methode etwas genauer: Das Resultatobjekt ist ein lokales Objekt, es ist folglich nach dem Verlassen der Methode nicht mehr existent. Wie kommt es dann dazu, dass dennoch ein Objekt für den Konstruktoraufruf in der Anweisung

```
s2 = s1.ToUpperCase();
```

zur Verfügung steht? Es ist der Kopierkonstruktor, der sich beim Verlassen der `ToUpperCase`-Methode dafür verantwortlich zeichnet, eine Kopie des Resultatobjekts auf dem Stapel anzulegen! Sie dient dem nachfolgenden Wertzuweisungsoperator als Aktualparameter. Wir verifizieren diese Behauptung, in dem wir in der `String`-Klasse einen Kopierkonstruktor mit folgender Trivialimplementierung ergänzen:

```
String::String (const String& s)
{
    cout << "c'tor (const String&)" << endl;

    // shallow copy
    m_len = s.m_len;
    m_ptr = s.m_ptr;
}
```

An den Programmausgaben können wir nun erkennen, dass zum einen der Kopierkonstruktor zur Ausführung gelangt und zum anderen die Anzahl der Konstruktoren- und Destruktoraufrufe jetzt symmetrisch ist:

```
c'tor (char*)
c'tor ()
c'tor (char*)
c'tor (const String&)
d'tor
d'tor
d'tor
```

```
d'tor
```

Das Programm stürzt in dieser Variante immer noch ab, der Kommentar in der Implementierung des Kopierkonstruktors führt uns auf die richtige Spur: Der Kopierkonstruktor erzeugt in seiner Standardimplementierung eine Objektkopie mit der Strategie des flachen Kopierens. Bei Objekten mit dynamischen Instanzvariablen kommt es wieder zum bereits vorgestellten Absturz des `delete`-Operators. Implementieren wir den Kopierkonstruktor mit der tiefen Kopierstrategie (Listing 3.18), können wir einen korrekten Verlauf der `Demo`-Methode ohne Absturz nachvollziehen:

```
01: String::String (const String& s)
02: {
03:     // allocate new buffer
04:     m_len = s.m_len;
05:     m_ptr = new char[m_len];
06:
07:     // deep copy
08:     for (int i = 0; i < m_len; i ++)
09:         m_ptr[i] = s.m_ptr[i];
10: }
```

Beispiel 3.18. Implementierung des Kopierkonstruktors mit der tiefen Kopierstrategie.

Der Wertzuweisungsoperator, der Kopierkonstruktor und der Destruktor werden zusammen *„The Big Three“* genannt. Dahinter verbirgt sich die einfache Regel, dass in Klassen, die einen Destruktor benötigen, der Kopierkonstruktor und der Wertzuweisungsoperator ebenfalls vorhanden sein müssen. In Klassen, die Speicher von der Halde in Anspruch nehmen, müssen diese drei Klassenelemente immer implementiert sein. Die Gemeinsamkeiten zwischen dem Wertzuweisungsoperator und dem Kopierkonstruktor sollten auf der Hand liegen. Beide Elemente initialisieren ein Objekt mit den Daten eines existierenden Objekts. Im Wertzuweisungsoperator ist zusätzlich der Speicherplatz des aktuellen Objekts freizugeben, bevor es mit den neuen Werten versorgt wird.

Die drei Klassenelemente lassen sich prinzipiell aus zwei elementareren Grundfunktionen beschreiben. Wenn die Hilfsmethode `Init` die Werte ihres Arguments in das aktuelle Objekt (auf Basis der tiefen Strategie) kopiert (bzw. eine parameterlose `Init`-Methode die Zeigervariablen der Klasse passend vorbelegt) und die Hilfsmethode `Free` allen dynamisch allokierten Speicher wieder freigibt, so können wir *„The Big Three“* für eine Klasse `Type` schematisch wie folgt darstellen (Listing 3.19):

```
01: class Type
02: {
03: public:
04:     Type ();                              // default c'tor
05:     Type (const Type& type);              // copy c'tor
06:     ~Type();                              // d'tor
07:     Type& operator= (const Type& type);   // assignment operator
08: };
09:
10: // default c'tor
11: Type::Type ()
12: {
13:     Init ();
14: }
15:
16: // copy c'tor
17: Type::Type (const Type& right)
18: {
19:     Init (right);
20: }
21:
22: // d'tor
23: Type::~Type ()
24: {
25:     Free ();
26: }
27:
28: // assignment operator
29: Type& Type::operator= (const Type& right)
```

```
30: {
31:     if (this != &right)
32:     {
33:         Free ();
34:         Init (right);
35:     }
36:
37:     return *this;
38: }
```

Beispiel 3.19. Schematische Darstellung der „The Big Three“-Spielregeln.

3.9. Überladung spezieller Operatoren

3.9.1. Überladung des Inkrement-Operators ++ und des Dekrement-Operators --

Die beiden unären Operatoren ++ und -- üben einen direkten Einfluss auf ihren Operanden aus, wie beispielsweise in dem klassischen C-Codefragment:

```
int i = 0;
i ++;
```

Aus diesem Grund sollten Überladungen in benutzerdefinierten Klassen dasselbe Verhalten aufweisen, sprich die Inkrement- und Dekrement-Operatoren sind stets im Sinne des „veränderbaren Verhaltens“ zu implementieren. Zusätzlich dürfen wir nicht außer Acht lassen, dass die beiden Operatoren in zwei Versionen existieren. Da gibt es zum einen die Präfix-Version, die als Ergebnis des Ausdrucks

```
++ i
```

den um Eins inkrementierten Wert besitzt. Die Postfix-Version im Gegensatz besitzt als Resultat des Ausdrucks

```
i ++
```

den Wert des Operanden *vor* der Anwendung der Inkrement-Operation. Diese Eigenschaft müssen wir selbstverständlich auf die Implementierung eines benutzerdefinierten Überladungen übertragen.

Bei der Deklaration des Inkrement- oder Dekrement-Operators haben wir das Problem, die jeweilige Version erkennen zu müssen. Hier hat man einen kleinen Kunstgriff angewendet und sich an die Technik des Überladens von Methoden in einer Klasse erinnert. Zur Unterscheidung der Präfix- und der Postfix-Version wurde die Methodensignatur der Postfix-Version mit einem zusätzlichen *Dummy*-Parameter (von Typ int) ausgestattet. Dieser Parameter ist zur Laufzeit absolut überflüssig, da weder ein Wert übergeben noch in der Implementierung auf diesen Parameter zugegriffen wird. Wenn wir – um bei unserem einheitlichen Stil zu bleiben – den Inkrement-Operator als globale Funktion definieren, müssen wir die `Fraction`-Klasse um die folgenden zwei Deklarationen erweitern:

```
01: // increment/decrement operators (prefix/postfix version)
02: friend Fraction& operator++ (Fraction&);           // prefix increment
03: friend const Fraction operator++ (Fraction&, int); // postfix increment
04: friend Fraction& operator-- (Fraction&);           // prefix decrement
05: friend const Fraction operator-- (Fraction&, int); // postfix decrement
```

Beispiel 3.20. Überladung des Inkrement-Operators ++ und des Dekrement-Operators -- mit Präfix- und Postfix-Version.

Das nächste Listing veranschaulicht die unterschiedlichen Vorgehensweisen, die bei der Implementierung einer Präfix- und Postfix-Version zu beachten sind:

```
01: // increment operator: prefix version
02: Fraction& operator++ (Fraction& f)
03: {
04:     f += 1;
05:     return f;
06: }
```

```
07:
08: // decrement operator: prefix version
09: Fraction& operator-- (Fraction& f)
10: {
11:     f -= 1;
12:     return f;
13: }
14:
15: // increment operator: postfix version
16: const Fraction operator++ (Fraction& f, int)
17: {
18:     Fraction tmp = f; // construct a copy
19:     ++ f;             // increment number
20:     return tmp;       // return the copy
21: }
22:
23: // decrement operator: postfix version
24: const Fraction operator-- (Fraction& f, int)
25: {
26:     Fraction tmp = f; // construct a copy
27:     -- f;             // decrement number
28:     return tmp;       // return the copy
29: }
```

Beispiel 3.21. Implementierung Inkrement- und Dekrement-Operator in der Präfix- und Postfix-Version.

Auffällig sind in Listing 3.21 die unterschiedlichen Rückgabetypen. Die Postfix-Version besitzt das Problem, dass sie als Resultat den ursprünglichen Wert ihres Operanden vor Ausführung der Operation zurückliefern muss. Aus diesem Grund werden in den Zeilen 18 bzw. 26 mit dem Kopierkonstruktor jeweils Temporärobjekte mit dem Originalzustand angelegt, um diese in Zeile 20 (bzw. 28) als Ergebnis der Operation zurückzugeben. Der Rückgabetyp der Präfix-Version wird durch ein anderes Kriterium beeinflusst. Dieses kann man zunächst allgemein damit umschreiben, dass eine Implementierung überladener Operatoren konsistent zum Verhalten der Standardoperatoren sein muss. Werfen wir zu diesem Zweck zunächst einen Blick auf die Inkrement-Operatoren bei Variablen des Typs `int`. In der Präfix-Version ist eine Schachtelung der Inkrement-Operatoren zulässig, bei der Postfix-Version hingegen macht dies keinen Sinn:

```
int i = 0;
++ ++ i;     // o.k.
i ++ ++ ;    // '++' needs l-value
```

Dieses Verhalten sollten auch überladene Operatoren aufweisen. Aus diesem Grund ist das Resultat der beiden Postfix-Version in Zeile 16 und 24 mit `const` deklariert. Die Präfix-Version wiederum liefert (Zeile 2 und 9) eine Referenz ihres Operanden zurück, so dass geschachtelte Ausführungen möglich sind!

Wir schließen diesen Abschnitt mit einem kleinen Beispiel ab. Das Codefragment

```
Fraction f (1, 2);
Fraction g;

g = f ++;
g.Print ();

g = ++ f;
g.Print ();

g = f --;
g.Print ();

g = -- f;
g.Print ();
```

ist übersetzungsfähig und liefert das Ergebnis

```
1/2
5/2
5/2
1/2
```

3.9.2. Überladung des Index-Operators `[]` und des Funktionsoperators `()`

3.9.3. Überladung der Stream-Operatoren << und >>

Kapitel 4. Komposition und Vererbung

4.1. Einführung

Eine große Stärke der objektorientierten Programmierung besteht darin, neue Klassen aus bestehenden Klassen zu bilden. Die Mechanismen der *Komposition* und *Vererbung* sind zwei Möglichkeiten, um dieses Ziel zu erreichen.

4.2. Komposition

Wir starten dieses Kapitel mit einer Betrachtung der Frage, wie Klassen zueinander in Beziehung stehen können. Betrachten wir deshalb zwei Klassen `Point` und `Line`, zum Beispiel

```
class Point
{
public:
    int m_x;
    int m_y;
};
```

und

```
class Line
{
public:
    Point m_start;
    Point m_end;
};
```

Man kann in dieser Situation feststellen, dass die Klasse `Line` aus einem Anfangs- und einem Endpunkt besteht, oder etwas simpler formuliert: Ein `Line`-Objekt »hat einen« Anfangs- und einen Endpunkt. Die `Line`-Klasse des obigen Beispiels demonstriert auf anschauliche Weise den Typus der »hat-ein«-Beziehung (Abbildung 4.1): Die Elemente einer Klasse werden aus Objekten einer anderen Klasse gebildet oder – etwas akademischer formuliert – *komponiert*. Wir sind bei der *Komposition* (engl. *Composition*) angekommen, sie repräsentiert eine Möglichkeit, um mit Hilfe existierender neue Klassen zu definieren.

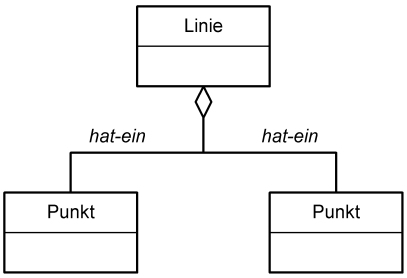

Abbildung 4.1. Die hat-ein Beziehung an einem Beispiel.

4.2.1. Konstruktion und Destruktion

Auch für komponierte Objekte müssen wir der Frage nach ihrer korrekten Initialisierung nachgehen. Da sowohl das umgebende wie auch die enthaltenen Objekte jeweils durch einen Konstruktoraufruf zu initialisieren sind, stoßen wir auf folgende Fragen:

- In welcher Reihenfolge werden die beteiligten Konstruktoren aufgerufen?
- Wie können aktuelle Parameter an die Konstruktoren der enthaltenen Unterobjekte übergeben werden?

Wir studieren diese Fragen am Beispiel einer Klasse `Student`:

```
class Student
{
public:
    String m_firstName;
    String m_lastName;
    Date m_birth;
```

```
};
```

Wird von der Klasse `Student` ein Objekt angelegt, müssen gleichzeitig zwei `String`-Objekte und ein `Date`-Objekte erzeugt werden. Die Antwort auf die erste Frage lautet: Es werden zunächst die Konstruktoren der enthaltenen Objekte aufgerufen und zum Abschluss der Konstruktor des umgebenden Objekts. Damit kommen wir auf die enthaltenen Objekte zu sprechen, für sie gilt: Ihre Konstruktoren werden in der Reihenfolge ihrer Deklaration innerhalb der umgebenden Klasse aufgerufen.

Zur Verifikation dieser Aussagen ergänzen wir Trivialimplementierungen der Klassen `String`, `Date` und `Student` jeweils um einen Standardkonstruktor mit einer einfachen Testausgabe:

```
class String
{
public:
    String () { printf ("c'tor String\n"); }
};

class Date
{
public:
    Date () { printf ("c'tor Date\n"); }
};

class Student
{
public:
    Student () { printf ("c'tor Student\n"); }

public:
    String m_first;
    String m_last;
    Date m_birth;
};
```

Wir erhalten auf diese Weise die Ausgaben

```
c'tor String
c'tor String
c'tor Date
c'tor Student
```

wenn wir eine Instanz der `Student`-Klasse anlegen. Um die Antwort der zweiten Frage zu überprüfen, stellen wir den Deklarationsabschnitt der `Student`-Klasse wie folgt um:

```
class Student
{
public:
    Date m_birth;
    String m_firstName;
    String m_lastName;

    ...
};
```

Die Testausgaben lauten nun tatsächlich

```
c'tor Date
c'tor String
c'tor String
c'tor Student
```

Wie lassen sich die Konstruktoren der enthaltenen Objekte mit Aktualparametern versorgen? Die Lösung ist die Übergabe der Parameter mit Hilfe einer speziellen Syntax, der so genannten

Initialisierungsliste (Listing 4.1):

```
01: class Student
02: {
03: public:
04:     // c'tor
05:     Student (char* first, char* last, int day, int month, int year)
06:         : m_first(first), m_last(last), m_birth(day, month, year)
07:         {}
08:
09: private:
10:     String m_first;
11:     String m_last;
12:     Date m_birth;
13: };
```

Beispiel 4.1. Konstruktor mit Initialisierungsliste (Definition innerhalb der Klasse).

Im Anschluss an die Parameterliste des Konstruktors der umgebenden Klasse (hier: Klasse `Student`) werden hinter einem Doppelpunkt die einzelnen Unterobjekte aufgeführt (hier: `m_first`, `m_last` und `m_birth`). In runden Klammern hinter dem Namen der Unterobjekte stehen dann die Parameter für den gewünschten Konstruktor dieser Objekte. Wie bereits erwähnt: Die Reihenfolge der Konstruktoraufrufe der Unterobjekte wird *nicht* durch ihre Position innerhalb der Initialisierungsliste bestimmt, sondern durch die Reihenfolge, in der die einzelnen Unterobjekte in der umgebenden Klasse definiert sind.

Die Spezifikation einer Initialisierungsliste kann auch außerhalb der Klasse erfolgen. Eine zweite Gestaltung eines Konstruktors der `Student`-Klasse könnte wie in Listing 4.2 gezeigt aussehen:

```
01: class Student
02: {
03: public:
04:     // c'tor
05:     Student (char*, char*, int, int, int);
06:
07: private:
08:     String m_first;
09:     String m_last;
10:     Date m_birth;
11: };
12:
13: ...
14:
15: Student::Student (char* first, char* last, int day, int month, int year)
16:     : m_first(first), m_last(last), m_birth(day, month, year)
17:     { }
```

Beispiel 4.2. Konstruktor mit Initialisierungsliste (Definition außerhalb der Klasse).

Wir erkennen an den vorangegangenen Beispielen, dass der verbleibende Anweisungsblock des Konstruktors im Extremfall sogar leer bleiben kann, also nur durch eine öffnende und schließende geschweifte Klammer gebildet wird („`{}`“).

Diese Art der Initialisierung ist für Instanzvariablen beliebigen Typs anwendbar, also auch für Variablen elementaren Datentyps. Ein Konstruktor der Klasse `Date` könnte alternativ zur Form

```
Date::Date (int day, int month, int year)
{
    m_day = day;
    m_month = month;
    m_year = year;
}
```

damit auch kompakter durch

```
Date::Date (int day, int month, int year)
    : m_day(day), m_month(month), m_year(year)
    {}
```

definiert werden.

Nach der Konstruktion von Objekten kommen wir nun auf das Ende ihrer Lebenszeit zu sprechen. Für jedes beteiligte Objekt in einem komponierten Objekt kommt natürlich ein Destruktor zu Ausführung. Wie sieht es mit der Reihenfolge der einzelnen Destruktoraufrufe aus? Es gilt die einfache Regel, dass alle Destruktoren in der umgekehrten Reihenfolge wie die korrespondierenden Konstruktoren ausgeführt werden. Um auch diese Aussage mit einem einfachen Codefragment zu belegen, ergänzen wir die `Student`-Klasse sowie ihre unterlagerten Klassen um geeignete Testausgaben in den Destruktoren:

```
class String
{
public:
    ~String () { printf ("d'tor String\n"); }
};

class Date
{
public:
    ~Date () { printf ("d'tor Date\n"); }
};

class Student
{
public:
    ~Student () { printf ("d'tor Student\n"); }

public:
    String m_first;
    String m_last;
    Date m_birth;
};
```

Wir beobachten die Testausgaben

```
d'tor Student
d'tor Date
d'tor String
d'tor String
```

wenn die Lebensdauer eines `Student`-Objekts erlischt.

4.2.2. Komposition und Arrays

Wir haben uns bislang ausschließlich mit der Konstruktion eines einzigen Objekts beschäftigt. Mit der Komposition sind zusätzliche Objekte ins Spiel gekommen, diese wurden allerdings von der umhüllenden Klasse versteckt. Wie sieht es aus, wenn wir ein Array von Objekten anlegen, etwa in der Form

```
Test data[5];
```

mit einer beliebigen Klasse `Test`. Bemerkenswert ist, dass im Falle von Arrays, die aus Objekten bestehen, für *jedes* Objekt des Arrays ein Konstruktor aufgerufen wird. Definieren wir die Klasse `Test` zu Demonstrationszwecken durch

```
class Test
{
public:
    Test () { printf ("c'tor Test\n"); }
};
```

dann erhalten wir auf der Konsole die Ausgaben

```
c'tor Test
c'tor Test
c'tor Test
c'tor Test
c'tor Test
```

Ähnliche Beobachtungen können wir machen, wenn das Array durch die Speicherverwaltung freigegeben wird. Für jedes Objekt des Arrays wird der Destruktor aufgerufen. Noch offen ist, in welcher Reihenfolge die Aufrufe der Konstruktoren und Destruktoren erfolgen. Hierzu modifizieren wir die Definition der Klasse `Test` in Listing 4.3 wie folgt:

```
01: class Test
02: {
03: private:
04:     static int g_count;
05:
06: public:
07:     Test ()
08:     {
09:         m_count = ++ Test::g_count;
10:         printf ("c'tor Test [%d]\n", m_count);
11:     }
12:
13:     ~Test () { printf ("d'tor Test [%d]\n", m_count); }
14:
15: private:
16:     int m_count;
17: };
18:
19: int Test::g_count = 0;
```

Beispiel 4.3. Klasse `Test` zur Analyse von Konstruktor- und Destruktoraufrufen.

In dieser Version besitzt die Klasse `Test` eine Klassenvariable `g_count` (Zeile 4 von Listing 4.3). Mit ihrer Hilfe initialisieren wir die einzige Instanzvariable `m_count` bei der Erzeugung von `Test`-Objekten mit einem aufsteigenden Wert. Die Traceausgaben machen deutlich, dass das Aufräumen der Objekte in einem Array in der umgekehrten Reihenfolge wie ihre Initialisierung erfolgt:

```
c'tor Test [1]
c'tor Test [2]
c'tor Test [3]
c'tor Test [4]
c'tor Test [5]
d'tor Test [5]
d'tor Test [4]
d'tor Test [3]
d'tor Test [2]
d'tor Test [1]
```

4.3. Vererbung

4.3.1. Spezialisierung und Generalisierung

Wir schlagen einen zweiten Weg zur Erstellung neuer Klassen auf Basis existierender Klassen wiederum mit der Fragestellung des letzten Kapitels ein, wie Klassen zueinander in Beziehung stehen können. Zur Illustration legen wir dieses Mal zwei Klassen `Person` und `Employee` zu Grunde. Von einer Person erwarten wir, dass sie einen Vornamen, einen Nachnamen und ein Alter besitzt. Ein Angestellter in einem Unternehmen besitzt unter anderem ein Gehalt und eine Personalnummer. In welcher *Beziehung* stehen die beiden Klassen `Person` und `Employee` zueinander?

Ein zweiter, sehr häufiger Beziehungstypus in der realen Welt ist die *ist-ein* Beziehung (engl. *is-a* relationship). Wenn wir sagen, dass ein Angestellter *eine* Person *ist*, dann bringen wir damit zum Ausdruck, dass wir einen Angestellten als *Spezialisierung* einer Person auffassen (Abbildung 4.2). Ein Angestellter hat alle Eigenschaften einer Person (einen Vor- und Nachnamen, ein Alter und weitere Eigenschaften, die eine Person besitzt), und darüber hinaus eine Reihe zusätzlicher Eigenschaften, die eben einen Angestellten ganz spezifisch auszeichnen (wie etwa eine Personalnummer, ein Gehalt, eine Betriebszugehörigkeitsdauer, usw.). Ein Werkstudent ist ebenfalls eine Person. Wir gehen deshalb davon aus, dass er eine Reihe von Eigenschaften mit einem Angestellten gemeinsam hat, die durch ihr gemeinsames Wesen als Person resultieren. Auf der anderen Seite besitzt ein Werkstudent auch eine Reihe von Eigenschaften, die nur auf Werkstudenten zutreffen, aber trotzdem eine Spezialisierung in Bezug auf eine Person gestatten.

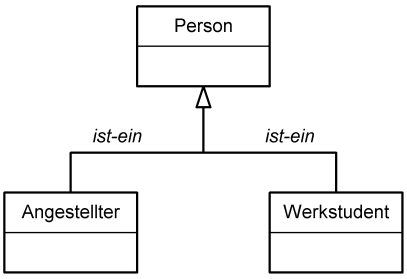

Abbildung 4.2. Die ist-ein Beziehung an einem Beispiel.

Die Beziehung der Spezialisierung besitzt ein Gegenstück, das *Verallgemeinerung* oder *Generalisierung* heißt. Angestellte und Werkstudenten sind Spezialisierungen einer Person, eine Person wiederum ist die Verallgemeinerung eines Angestellten oder eines Werkstudenten. Etwas formaler betrachtet kann man die beiden Beziehungen der Spezialisierung und Generalisierung als zueinander *reziprok* und *hierarchisch* einordnen. Sie sind reziprok, da die Spezialisierung die entgegengesetzte Operation zu einer Generalisierung ist und umgekehrt. Den hierarchischen Charakter der Beziehungen kann man sich durch eine Baumstruktur vorstellen, in der die spezielleren Typen von den allgemeineren Typen abzweigen. Man kann in dieser Hierarchie nach oben in die Richtung der allgemeineren Typen steigen. Bewegt man sich in dem Baum nach unten, erfolgt eine Spezialisierung.

4.3.2. Basisklasse und abgeleitete Klasse

Würden wir zwei Klassen `Person` und `Employee` ohne Berücksichtigung ihrer existierenden Beziehungseigenschaft entwerfen, dann könnten zwei rudimentäre Deklarationen so aussehen:

```
class Person
{
public:
    char* m_first;
    char* m_last;
    int   m_age;

    void Print ();
};
```

```
class Employee
{
public:
    char*  m_first;
    char*  m_last;
    int    m_age;
    double m_salary;
    int    m_number;
};
```

Die Gemeinsamkeiten der beiden Klassentypen `Person` und `Employee` können nun durch ein besonderes Sprachelement unterstützt werden. Wir wenden das Prinzip der *Vererbung* an, das in einer objektorientierten Programmiersprache wie C++ zur Umsetzung der Spezialisierung eingesetzt wird. Die Betrachtung der Klasse `Employee` als Spezialfall der Klasse `Person` bringt man in C++ dadurch zum Ausdruck, dass man im Anschluss an die Klassenklausel den Namen der allgemeineren Klasse anhängt und die beiden Namen durch einen Doppelpunkt voneinander trennt (auf das Schlüsselwort `public` kommen wir in diesem Kapitel später noch zu sprechen):

```
class Employee : public Person
```

Auf diese Weise übernimmt die Klasse `Person` die Rolle der so genannten *Basisklasse* (auch *Vaterklasse* oder *Oberklasse* genannt), die Klasse `Employee` hingegen wird *abgeleitete Klasse* oder auch *Unterklasse* genannt. Den Doppelpunkt in der Klassenklausel kann man als „*leitet sich von ... ab*“ lesen.

Durch den Mechanismus der Vererbung erbt die abgeleitete Klasse die Instanzvariablen der Vaterklasse. In den Instanzvariablendeklarationen von Klasse `Employee` dürfen folglich die drei Variablen `first`, `last` und `age` *nicht* mehr auftauchen. Alle Elemente der Basisklasse sind jetzt automatisch Elemente der abgeleiteten Klasse:

```
class Employee : public Person
{
public:
    double m_salary;
    int    m_number;
};
```

Beispiel 4.4. Klasse `Employee` als Spezialisierung der Klasse `Person`.

Das Prinzip der Vererbung ist natürlich auch auf Methoden anwendbar. Ist in der Vaterklasse beispielsweise eine Methode `Print` vorhanden, müssen wir von dieser Methode keine Kopie mehr in der abgeleiteten Klasse aufnehmen. Im Gegenteil, alle öffentlichen Methoden der Vaterklasse stehen in der abgeleiteten Klasse automatisch zur Verfügung. Diese Spielregel gilt *nicht* für Konstruktoren und Destruktoren, wir betrachten diese speziellen Methoden einer Klasse in den noch folgenden Erläuterungen separat. Für das Erste belassen wir es bei der Aussage, dass eine abgeleitete Klasse bei der Erzeugung von Objekten den Standardkonstruktor ihrer Basisklasse in Anspruch nimmt.

Wir führen die bislang diskutierten Aspekte an einem kleinen Beispiel vor:

```
001: class Person
002: {
003: public:
004:     // data
005:     char m_first[20];
006:     char m_last[20];
007:     int  m_age;
008:
009: public:
010:     // c'tor and d'tor
011:     Person ();
012:     ~Person ();
013:
```

```
014:     // methods
015:     void SetFirstName (char *);
016:     void SetLastName (char *);
017:     void SetAge (int);
018:     void Print () ;
019: };
020:
021: class Employee : public Person
022: {
023: private:
024:     double m_salary;
025:     int    m_number;
026:
027: public:
028:     // c'tor and d'tor
029:     Employee ();
030:     ~Employee ();
031:
032:     // methods
033:     void SetSalary (double);
034: };
035:
036: Person::Person ()
037: {
038:     printf ("c'tor Person()\n");
039:     strcpy (m_first, "<empty>");
040:     strcpy (m_last, "<empty>");
041:     m_age = -1;
042: }
043:
044: void Person::SetFirstName (char * name)
045: {
046:     int i = 0;
047:     while (name[i] != '\0')
048:     {
049:         m_first[i] = name[i];
050:         i ++;
051:     }
052:     m_first[i] = '\0';
053: }
054:
055: void Person::SetLastName (char * name)
056: {
057:     int i = 0;
058:     while (name[i] != '\0')
059:     {
060:         m_last[i] = name[i];
061:         i ++;
062:     }
063:     m_last[i] = '\0';
064: }
065:
066: void Person::SetAge (int age)
067: {
068:     m_age = age;
069: }
070:
071: void Person::Print ()
072: {
073:     printf ("Name: %s %s, Age: %d\n", m_first, m_last, m_age);
074: }
075:
076: Person::~Person ()
077: {
078:     printf ("d'tor Person\n");
079: }
080:
081: Employee::Employee ()
082: {
083:     printf ("c'tor Employee\n");
084:     m_salary = 0.0;
085:     m_number = 1;
086: }
087:
088: Employee::~Employee ()
089: {
090:     printf ("d'tor Employee\n");
091: }
```

Beispiel 4.5. Das Prinzip der Vererbung an einem einfachen Beispiel aufgezeigt.

Zum Testen verwenden wir folgenden Testrahmen:

```
void main ()
{
    Employee e;
    e.Print ();

    e.SetFirstName ("Anton");
    e.SetLastName ("Meier");
    e.m_age = 44;
    e.Print ();
}
```

Die Ausgaben lauten:

```
c'tor Person()
c'tor Employee
Name: <empty> <empty>, Age: -1
Name: Anton Meier, Age: 44
d'tor Employee
d'tor Person
```

Bevor wir auf die einzelnen Aspekte der Vererbung detaillierter eingehen, fassen wir die bislang erarbeiteten Kernaussagen zusammen:

- Jedes Objekt vom Typ `Employee` enthält einen Speicherbereich, der alle Daten der Basisklasse (hier: `Person`) enthält. Dieser Speicherbereich wird noch vor der Initialisierung eines `Employee`-Objekts durch einen impliziten Aufruf des Basisklassenstandardkonstruktors initialisiert. Durch diesen Mechanismus wird erreicht, dass alle Instanzvariablen des Basisklassenobjekts *vor* dem Objekt der abgeleiteten Klasse einen wohldefinierten Zustand annehmen. Letzteres könnte ja bei seiner Initialisierung bereits auf geerbte Elemente der Basisklasse zugreifen, aus diesem Grund ist die Reihenfolge bei der Initialisierung von Basisklasse und abgeleiteter Klasse essentiell.

- Destruktoren werden in der umgekehrten Reihenfolge (automatisch) aufgerufen, also zuerst für die abgeleitete Klasse und dann für die Basisklasse, da der Destruktor der abgeleiteten Klasse noch den Zugriff auf die Instanzvariablen der Basisklasse besitzt.

- Jede Instanzvariable der Basisklasse steht in Objekten der abgeleiteten Klasse zur Verfügung, sofern die Variable mit der Zugriffsklasse `public` deklariert ist. Im letzten Beispiel ist das Element `m_age` sowohl in `Person`- wie auch in `Employee`-Objekten vorhanden, obwohl es in der `Employee`-Klasse nicht deklariert worden ist. Der Zugriff auf eine Instanzvariable an einem Objekt lässt nicht erkennen, in welcher Klasse (korrespondierende Klasse oder eine Basisklasse davon) das Element definiert ist.

- Jede Methode der Basisklasse kann auf ein Objekt der abgeleiteten Klasse angewendet werden, sofern die Methode mit der Zugriffsklasse `public` deklariert ist. Im letzten Beispiel kann die `Print`-Methode sowohl von `Person`- wie auch von `Employee`-Objekten benutzt werden, obwohl sie in der `Employee`-Klasse nicht vorhanden ist. Der Aufruf einer Methode an einem Objekt lässt nicht erkennen, ob sie zur korrespondierenden Klasse oder zur Basisklasse gehört.

- Abgeleitete Klassen können zusätzliche Instanzvariablen und Methoden enthalten, die keinen Bezug zur Basisklasse haben. Ein Gehalt (Instanzvariable `m_salary`) ist kennzeichnend für einen Angestellten, aber nicht für eine Person.

- Eine Klasse ist ein Datentyp in C++. Eine abgeleitete Klasse kann als Teiltyp der Oberklasse aufgefasst werden. Ein Objekt der abgeleiteten Klasse ist zuweisungskompatibel zu einem Objekt der Basisklasse. Die Anweisung

```
Employee e;
Person p;
p = e;
```

kopiert den Inhalt aller Instanzvariablen des `Employee`-Objekts, die im Basisklassenobjekt des Typs `Person` ebenfalls enthalten sind, um. Die nur zur abgeleiteten Klasse gehörenden Instanzvariablen werden *nicht* umkopiert, da in der Basisklasse dafür kein Platz vorhanden ist.

Eine umgekehrte Wertzuweisung

```
e = p;
```

ist *nicht* möglich, da auf diese Weise die nur in der Basisklasse exisitierenden Instanzvariablen undefiniert bleiben würden.

4.3.3. Der Zugriffsspezifizierer `protected`

Im letzten Abschnitt ist in der Basisklasse `Person` ausnahmslos die Zugriffsklasse `public` zum Einsatz gekommen. Hätten wir in gewohnter Manier die Instanzvariablen dieser Klasse mit `private` vor jeglichem Zugriff von außen geschützt, würden einige Folgeprobleme auftreten. Beispielsweise wird der Konstruktor der `Person`-Klasse des folgenden Codefragments mit der Fehlermeldung *„'Person::m_age' : cannot access private member declared in class 'Person'“* abgewiesen:

```
class Person
{
private:
    // data
    char m_first[20];
    char m_last[20];
    int m_age;
};

class Employee : public Person
{
private:
    double m_salary;
    int m_number;

public:
    // c'tor
    Employee () { m_age = 21; }  // Error: 'Person::m_age' :
                                 // cannot access private member
                                 // declared in class 'Person'
};
```

Auf welche Instanzvariablen und -methoden einer Basisklasse hat eine abgeleitete Klasse Zugriff? Eine abgeleitete Klasse hat *keinen* Zugriff auf die mit `private` deklarierten Instanzvariablen und -methoden der Basisklasse. Eine mögliche Abhilfe ist die Ergänzung der Basisklasse um geeignete öffentliche Methoden (hier: Methode `SetAge`), die den Zugriff auf die privaten Elemente kontrolliert durchführt:

```
class Person
{
private:
    // data
    char m_first[20];
    char m_last[20];
    int m_age;

public:
    // methods
    void SetAge (int age) { m_age = age; }
};

class Employee : public Person
```

```
{
private:
    double m_salary;
    int m_number;

public:
    // c'tor
    Employee () { SetAge (21); }
};
```

Alternativ kann eine Basisklasse eine abgeleitete Klasse zum Freund erklären:

```
class Person
{
    friend class Employee;
private:
    // data
    char m_first[20];
    char m_last[20];
    int m_age;
};

class Employee : public Person
{
private:
    double m_salary;
    int m_number;

public:
    // c'tor
    Employee () { m_age = 21; }
};
```

Beide Behebungen des Problems sind nicht optimal. Die ideale Lösung des Problems besteht im Gebrauch einer dritten Kategorie des Zugriffsschutzes, die eine Balance zwischen dem öffentlichen und privaten Zugriffsschutz darstellt. Das korrespondierende Schlüsselwort lautet `protected`, der so genannte *geschützte* Zugriffsschutz ist in einer Klasse genau dann zu verwenden, wenn man ihr die Rolle als Basisklasse zuteil kommen lassen möchte. Mit `protected` deklarierte Klassenelemente sind für eine abgeleitete Klasse wie öffentliche (`public`) Elemente sichtbar, außerhalb der Klasse verhalten sie sich wie privat deklarierte Elemente.

Zusammenfassend lässt sich sagen:

- Private Klassenelemente sind nur innerhalb der Klasse und für befreundete Klassen sichtbar.
- Geschützte Klassenelemente sind innerhalb der Klasse, für befreundete Klassen sichtbar, für abgeleitete Klassen und wiederum für deren Freunde sichtbar.
- Öffentliche Klassenelemente unterliegen keinen Zugriffsbeschränkungen und sind von überall aus sichtbar.

Klassenelemente ohne Zugriffsklasse besitzen per Voreinstellung den privaten Zugriffsschutz.

4.3.4. Öffentliche, geschützte und private Vererbung

Die Schlüsselwörter `private`, `protected` und `public` dienen dazu, die Sichtbarkeit von Elementen einer Klasse festzulegen. Fehlt die Angabe einer Zugriffsklasse vor einem Klassenelement, ist das Element automatisch privat. Bei der Vererbung ist es möglich, den Sichtbarkeitsbereich aller Elemente einer Basisklasse an einer zentralen Stelle gegenüber der abgeleiteten Klasse zu verändern. Dazu ist bei der Deklaration der abgeleiteten Klasse eines der drei Schlüsselwörter `private`, `protected` oder `public` zu platzieren:

```
class B : public A {... }     // öffentliche Vererbung
```

oder

```
class B : protected A {... }  // geschützte Vererbung
```

oder

```
class B : private A {... }    // private Vererbung
```

Welche Auswirkungen ergeben sich auf diese Weise für die Elemente der abgeleiteten Klasse? Kommt in der Deklaration der abgeleiteten Klasse das Schlüsselwort `public` zum Einsatz, sind alle öffentlichen (`public`-)Elemente der Basisklasse auch in der abgeleiteten Klasse öffentlich. Ebenso bleiben geschützte Elemente der Basisklasse auch in der abgeleiteten Klasse geschützt. Die privaten Elemente spielen bei dieser Betrachtung keine Rolle, da sie in der abgeleiteten Klasse ohnehin nicht sichtbar sind.

Wird bei der Spezialisierung einer Klasse das `protected`-Schlüsselwort verwendet, sind die öffentlichen Elemente der Basisklasse in der abgeleiteten Klasse nur noch geschützt sichtbar. Die geschützten Elemente der Basisklasse bleiben auch in der abgeleiteten Klasse geschützt.

Bei einer privaten Ableitung schließlich werden alle öffentlichen und geschützten Elemente der Basisklasse zu privaten Elemente der abgeleiteten Klasse. Wie fassen diese Regeln in Abbildung 4.3 zusammen:

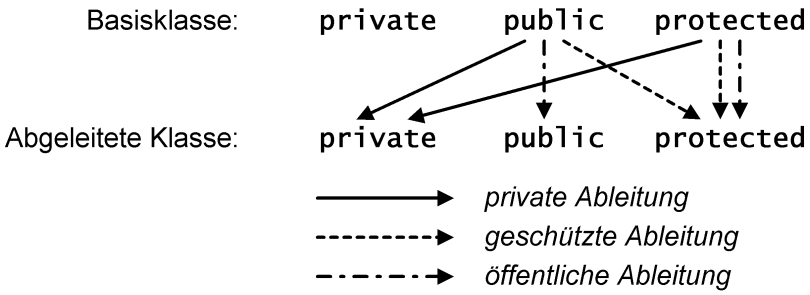

Abbildung 4.3. Einschränkungen der Sichtbarkeit bei Vererbung

Aus den Spielregeln aus Abbildung 4.3 können wir ablesen, dass die Zugriffsrechte auf diese Weise nur eingeschränkt, nicht aber erweiterbar werden. Das folgende Codefragment demonstriert die Einschränkungen der Sichtbarkeit an Hand einiger Beispiele:

```
class A
{
private:
    int m_a;

protected:
    int m_b;

public:
    int m_c;
};

class B : protected A
{
public:
    void f();
};

class C : private B
{
public:
    void g();
};

void B::f()
```

```
{
    m_a ++;  // No: cannot access private member 'm_a' declared in class 'A'
    m_b ++;  // o.k. - m_b is still protected
    m_c ++;  // o.k. - m_c is now protected
}

void C::g()
{
    m_a ++;  // No: cannot access private member 'm_a' declared in class 'A'
    m_b ++;  // o.k. - m_b is now private
    m_c ++;  // o.k. - m_c is now private
}
```

Zusätzlich zur Möglichkeit, alle Elemente einer Basisklasse in der abgeleiteten Klasse in ihrer Sichtbarkeit einzuschränken, kann man auch einzelne Elemente gezielt beeinflussen. In der Deklaration einer abgeleiteten Klasse werden die entsprechenden Elemente der Basisklasse noch einmal aufgeführt, allerdings unter Verwendung des Gültigkeitsbereichsoperators:

```
class A
{
public:
    int m_a;
    int m_b;
    int m_c;
};

class B : private A
{
protected:
    A::m_a;  // m_a is now protected instead of private

public:
    A::m_b;  // m_b is now public instead of private
};

void main ()
{
    B b;
    b.m_a ++;  // NO: cannot access protected member 'm_a' declared in class 'B'
    b.m_b ++;
    b.m_c ++;  // NO: 'A::m_c' not accessible
               //     because 'B' uses 'private' to inherit from 'A'
}
```

Die Kommentare des letzten Codefragments verdeutlichen, wie die Sichtbarkeitsangabe im Kopf der abgeleiteten Klasse durch Einzelangaben überschrieben werden kann. Nach wie vor ist es nicht möglich, private Elemente der Basisklasse zu öffentlichen oder geschützten Elementen der abgeleiteten Klasse zu machen.

4.3.5. Konstruktoren und Destruktoren bei Vererbung

Der automatische Aufruf des Basisklassenstandardkonstruktors beim Erzeugen eines Objekts einer abgeleiteten Klasse entspricht nicht immer dem Wunsch des Entwicklers. Häufig besitzt eine Klasse mehrere Konstruktoren (siehe das Stichwort „*Überladen von Methoden*“) und es ist die Frage zu klären, wie ein Konstruktor einer abgeleiteten Klasse einen bestimmten Basisklassenkonstruktor direkt ansprechen kann. Wir stoßen wieder – wie bei der Komposition von Objekten – auf einen Anwendungsfall der Initialisierungsliste. Dieses Mal erfolgt im Anschluss an die Liste der Parameter – durch einen Doppelpunkt getrennt – der Name der Basisklasse, wiederum gefolgt mit den aktuellen Parametern eines bestimmten Konstruktors der Basisklasse. Wir betrachten diese Aussage am besten in Listing 4.6 an einem Beispiel mit Personen und Angestellten:

```
01: class Person
02: {
03: public:
04:     Person (char*, char*, int);
```

```
05:
06: private:
07:     // data
08:     char m_first[20];
09:     char m_last[20];
10:     int m_age;
11: };
12:
13: Person::Person (char* first, char* last, int age)
14: {
15:     strcpy (m_first, first);
16:     strcpy (m_last, last);
17:     m_age = age;
18: }
19:
20: class Employee : public Person
21: {
22: private:
23:     double m_salary;
24:     int m_number;
25:
26: public:
27:     // c'tor
28:     Employee (char*, char*, int, double, int);
29: };
30:
31: Employee::Employee (char* first, char* last, int age, double salary, int number)
32:     : Person (first, last, age)
33: {
34:     m_salary = salary;
35:     m_number = number;
36: }
```

Beispiel 4.6. Delegation eines Konstruktoraufrufs

Wir erkennen in Zeile 32 von Listing 4.6, dass der Konstruktor der Klasse `Employee` dazu herangezogen wird, einen Konstruktor der Basisklasse `Person` geeignet mit aktuellen Parametern zu versorgen. Die syntaktische Anordnung des Basisklassenkonstruktoraufrufs vor dem Rumpf des aktuellen Konstruktors soll zum Ausdruck bringen, dass immer zuerst der Konstruktor der Basisklasse aufgerufen wird und erst anschließend der Konstruktor der abgeleiteten Klasse.

Um die korrekte Initialisierung der Basisklasse zu erreichen, ist es nicht möglich, in einer Klassenhierarchie den Konstruktor einer indirekten Basisklasse anzusprechen. Die Initialisierung der unmittelbaren Basisklasse könnte auf diese Weise nicht garantiert werden. Ein Codefragment der Gestalt

```
class A
{
public:
    A() {}
};

class B : A
{
};

class C : B
{
public:
    C() : A() {}
};
```

wird aus diesem Grund beim Übersetzen der Klasse `C` mit der Fehlermeldung *„'C' : illegal member initialization: 'A' is not a base or member“* abgewiesen!

4.3.6. Klassenhierarchien

Von einer abgeleiteten Klasse können wiederum beliebig viele weitere Klassen abgeleitet werden. Graphisch wird die Vererbung durch einen Pfeil dargestellt, der von der abgeleiteten Klasse zur Basisklasse zeigt (Abbildung 4.4).

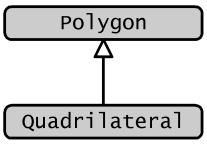

Abbildung 4.4. Graphische Darstellung der Vererbung.

Eine abgeleitete Klasse kann mehrere Basisklassen besitzen. Wir sprechen in diesem Fall von der so genannten *Mehrfachvererbung* (Abbildung 4.5), auf die wir in Kapitel XXX noch näher eingehen werden:

```
class Circle
{
    /* ... */
};

class Triangle
{
    /* ... */
};

class Cone : public Circle, public Triangle
{
    /* ... */
};
```

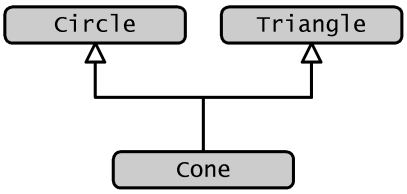

Abbildung 4.5. Die Mehrfachvererbung.

Durch wiederholtes Anwenden des Vererbungsprinzips lassen sich beliebig komplexe Klassenhierarchien aufbauen (Abbildung 4.6). Dabei ist allerdings zu beachten, dass in Klassenhierarchien keine geschlossenen Graphen bzw. rekursive Abhängigkeiten auftreten dürfen.

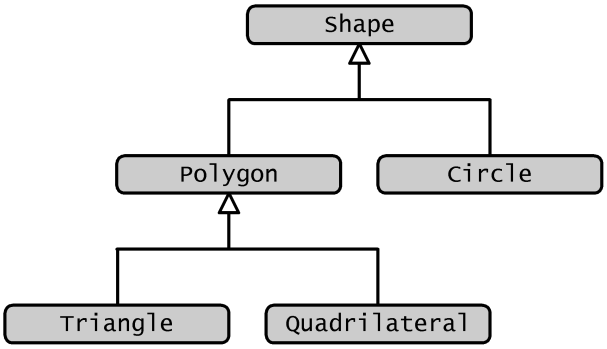

Abbildung 4.6. Beispiel einer komplexeren Klassenhierarchie.

Die Klassenhierarchie aus Abbildung 4.6 besitzt als Wurzelelement die Klasse `Shape` als allgemeinen Repräsentanten für geometrische Figuren. In dieser Klasse könnten zentrale Operationen wie `Draw`, `Move` oder `Rotate` zum Zeichnen, Verschieben und Rotieren definiert werden, da sie für beliebige geometrische Figur Sinn ergeben. Für `Shape`-Objekte selbst ergeben diese Methoden keinen Sinn. Sie können in der Implementierung in dieser Klasse entweder eine Fehlermeldung ausgeben oder einfach einen leeren Rumpf besitzen. Im Kapitel über abstrakte Klasse lernen wir eine weitere Variante in der Definition von Methoden kennen, die ganz ohne eine Implementierung auskommt.

Die Klasse `Polygon` ist von der Klasse `Shape` abgeleitet. Ein `Polygon`-Objekt verwaltet ein Array mit einer beliebigen Anzahl von Eckpunkten. Die Methoden `Draw`, `Move` oder `Rotate` werden in der `Polygon`-Klasse mit einer konkreten Implementierung überschrieben. Die Klassen `Triangle` (Dreieck) und `Quadrilateral` (Viereck) wiederum sind spezielle `Polygon`-Objekte mit drei bzw. vier Eckpunkten. Die Methoden zum Zeichnen, Verschieben und Rotieren werden von der Basisklasse

`Polygon` geerbt, sie sollten ohne weitere Anpassungen auch für Dreiecke und Vierecke taugen.

4.3.7. Instanzvariablen verdecken

Durch das Prinzip der Vererbung kommt eine abgeleitete Klassen in den Besitz der öffentlichen und geschützten Instanzvariablen ihrer Vaterklasse. Wie sieht es aus, wenn in der abgeleiteten Klasse eine Instanzvariable definiert ist, deren Name bereits für eine Instanzvariable in der Vaterklasse vergeben wurde? Dies ist in der Tat zulässig, man bezeichnet diese Eigenschaft der Programmiersprache C++ als *Verdecken* einer Instanzvariablen (Listing 4.7):

```
01: class A
02: {
03: public:
04:     int m_x;
05: };
06:
07: class B : public A
08: {
09: public:
10:     int m_x;
11: };
```

Beispiel 4.7. Verdecken einer Instanzvariablen.

In Listing 4.7 wird die Instanzvariable `m_x` der Vaterklasse `A` (Zeile 4) durch eine Instanzvariable desselben Namens in der abgeleiteten Klasse `B` (Zeile 10) verdeckt. Das Verdecken einer Instanzvariablen bedeutet nicht, dass in der abgeleiteten Klasse die Instanzvariable der Vaterklasse nicht mehr ansprechbar ist! Mit Hilfe des Gültigkeitsbereichsoperators kann man an einem Objekt der abgeleiteten Klasse auf verdeckte Instanzvariablen der Vaterklasse zugreifen, wie das Beispiel in Listing 4.8 demonstriert:

```
01: void main ()
02: {
03:     A a;
04:     B b;
05:
06:     a.m_x = 1;
07:     b.m_x = 2;
08:     b.A::m_x = 3;
09: }
```

Beispiel 4.8. Direkter Zugriff auf eine verdeckte Instanzvariable.

Abbildung 4.7 visualisiert den Zustand der zwei Objekte `a` und `b`, wie er beim Verlassen der `main`-Methode aus Listing 4.8 aussieht.

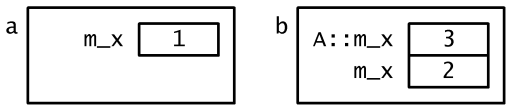

Abbildung 4.7. Abgeleitete Klasse mit verdeckenden Instanzvariablen.

Der Gültigkeitsbereichsoperator kann auch in Methoden angewendet werden, um im Falle einer Variablenverdeckung Konflikte im Zugriff zu lösen. Für diese Situation betrachten wir ein Beispiel in Listing 4.9:

```
01: class A
02: {
03: public:
04:     A () { m_x = 1; }
05:
```

```
06: protected:
07:     int m_x;
08: };
09:
10: class B : public A
11: {
12: protected:
13:     int m_x;
14:
15: public:
16:     B () { m_x = 2; }
17:
18: public:
19:     void f ()
20:     {
21:         printf ("m_x = %d\n", m_x);
22:         printf ("A::m_x = %d\n", A::m_x);
23:     }
24: };
```

Beispiel 4.9. Zugriff auf eine verdeckte Instanzvariable durch eine Instanzmethode.

Ein Aufruf der Methode `f` an einem Objekt der Klasse `B` führt zu den Ausgaben

```
m_x = 2
A::m_x = 1
```

Tipp

In wieweit das Verdecken einer Instanzvariablen eine sinnvolle Angelegenheit ist oder nur versehentlich passieren sollte, ist eine legitime Frage. Ein Entwickler, der eine existierende Klasse, die nicht von ihm implementiert wurde, spezialisiert, ist ja nicht ohne weiteres in der Lage, die Menge aller geerbten Instanzvariablen im Überblick zu haben. Nichts desto Trotz gilt die Aussage, dass das gezielte Verdecken von Instanzvariablen eher die Ausnahme darstellt und deshalb tunlichst vermieden werden sollte.

4.3.8. Instanzmethoden überschreiben

Die Erläuterungen zum Verdecken von Variablen lassen sich unmittelbar auf Methoden übertragen. Besitzt eine Klasse `A` eine Methode `f`, darf eine Spezialisierung der Klasse `A` ebenfalls eine Methode `f` mit derselben Schnittstelle definieren. In den meisten Büchern über C++ wird der Terminus des *Überschreibens* für diesen Vorgang gewählt. Das Verb *verdecken* wäre auch sinnvoll, da wir ähnliche Beobachtungen wie beim Verdecken von Instanzvariablen vorfinden werden:

```
01: class A
02: {
03: public:
04:     void f () { printf ("A::f\n"); }
05: };
06:
07: class B : public A
08: {
09: public:
10:     void f () { printf ("B::f\n"); }
11: };
```

Beispiel 4.10. Überschreiben (Verdecken) von Methoden.

Mit Hilfe des Gültigkeitsbereichsoperators kann man wieder steuern, auf welche Methode man – bei Namensgleichheit – gezielt zugreifen möchte. Ein Beispiel verdeutlicht die Details:

```
01: void main ()
02: {
03:     A a;
04:     B b;
05:
06:     a.f ();      // invokes f() defined in A
07:     b.f ();      // invokes f() defined in B
08:     b.A::f ();   // invokes f() defined in A
09: }
```

Beispiel 4.11. Zugriff auf verdeckte Instanzmethoden.

Die Kommentare in Listing 4.11 bzw. die Ausführung des Testfragments illustrieren den konkreten Methodenaufruf: Wird `f` an einem Objekt `o` aufgerufen, entscheidet zunächst der Objekttyp, welche Methode zur Ausführung gelangt. Ist `o` vom Typ `A`, handelt es sich um die Methode `f` aus Klasse `A`. Ist `o` vom Typ einer abgeleiteten Klasse, wird die in der abgeleiteten Klasse überschriebene Methode aufgerufen. Methoden einer Basisklasse gehen durch Überschreibung nicht verloren, mit Hilfe des Gültigkeitsbereichsoperators kann man diese nach wie vor erreichen, wie die Ausführung des letzten Codefragments illustriert:

```
A::f
B::f
A::f
```

Der Zugriff auf eine überschriebene Methode ist auch in der dafür verantwortlichen Methode möglich. Wenden wir uns einem etwas praxisrelevanteren Beispiel in Listing 4.12 zu:

```
01: class Person
02: {
03: public:
04:     void Print () { printf ("Person::Print\n"); }
05: };
06:
07: class Employee : public Person
08: {
09: public:
10:     void Print ()
11:     {
12:         Person::Print ();
13:         printf ("Employee::Print\n");
14:     }
15: };
```

Beispiel 4.12. Zugriff auf die verdeckte Instanzmethode.

Zeile 12 von Listing 4.12 demonstriert, dass der Gültigkeitsbereichsoperators auch innerhalb der verdeckten Instanzmethode selbst angewendet werden kann, um die verdeckte Instanzmethode zu erreichen.

Das Überschreiben von Instanzmethoden ist wie das Verdecken von Instanzvariablen technisch möglich, sollte aber in der zuvor beschriebenen Manier prinzipiell nicht erfolgen. Im nächsten Kapitel lernen wir eine Variation in der Definition von Methoden kennen. Wir werden sehen, dass das Überschreiben von Methoden dort in einem ganz anderen Licht erscheint und sehr viel Sinn ergibt.

Kapitel 5. Polymorphismus

5.1. Einführung

5.1.1. Konvertierung zwischen Klassentypen

Wir leiten diesen Abschnitt mit der Fragestellung ein, inwieweit Objekte eines bestimmten Klassentyps in einen anderen Klassentyp konvertierbar sind. Bei Variablen elementaren Datentyps ist der Hintergrund dieser Frage leicht zu erkennen. Die Werte einer `int`-Variablen könnten beispielsweise auch in einer `long`-Variablen abgelegt werden. Bei Klassentypen gibt es einen ähnlichen Zusammenhang genau dann, wenn die beiden Klassentypen in der *ist-ein*-Beziehung zueinander stehen. Aus diesem Grund ist, wenn wir beispielsweise die beiden Klassen `Account` und `CreditAccount` betrachten, folgende Typkonvertierung naheliegend und zulässig:

```
CreditAccount ca;
Account* a;
a = &ca;
```

Da jedes Girokonto auch ein Konto „ist“, kann einer Zeigervariablen des Typs `Account` stets die Adresse eines `CreditAccount`-Objekts zugewiesen werden. Welche Wirkung geht von dieser Wertzuweisung aus? Allgemein gesprochen übernimmt eine abgeleitete Klasse alle (öffentlichen und geschützten) Elemente ihrer Basisklasse. Somit kann ein Objekt vom Typ der abgeleiteten Klasse als Objekt der Basisklasse angesehen werden. Zu berücksichtigen ist allerdings, dass mit dem Zeiger der Basisklasse kein Zugriff auf Elemente der abgeleiteten Klasse möglich ist. Eine Anweisung der Gestalt

```
double limit = a -> m_limit;
```

wird vom Compiler mit der Fehlermeldung *„'m_limit' : is not a member of 'Account'“* abgewiesen. Die Instanzvariable `m_limit` ist über eine Zeigervariable des abgeleiteten Typs erreichbar. Über Basisklassenzeigervariablen sind ausschließlich die Elemente des Basisklassentyps erreichbar, wie in Abbildung 5.1 dargestellt wird.

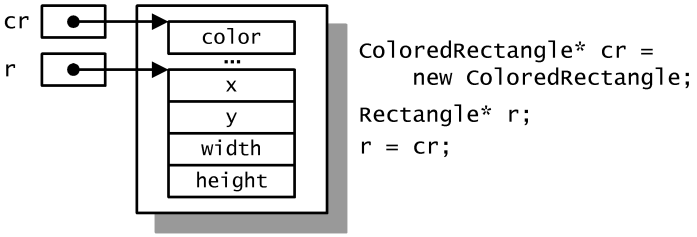

Abbildung 5.1. Implizite Typkonvertierung in den Basisklassentyp.

Damit lautet die erste Aussage für die Konvertierung zwischen Klassentypen: Stehen zwei Klassentypen in der *ist-ein* Beziehung zueinander, so kann eine Zeigervariable des Basisklassentyps stets auf ein Objekt eines abgeleiteten Klassentyps verweisen. Man bezeichnet Typkonvertierungen von der Unterklasse zur Oberklasse auch als *Upcast*, sie finden immer *implizit* statt (siehe Abbildung 5.2).

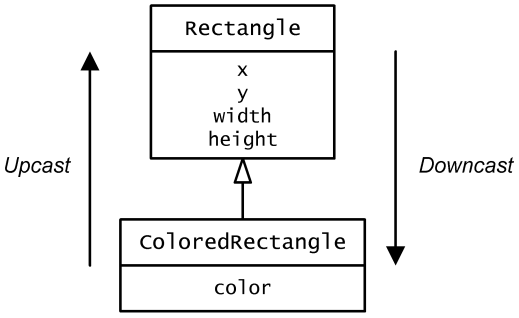

Abbildung 5.2. Upcast und Downcast: Typkonvertierung von der Unterklasse zur Oberklasse bzw.

umgekehrt.

Wie sieht es im umgekehrten Fall aus, sprich kann ich einer Zeigervariablen eines abgeleiteten Klassentyps ein Objekt des Basistyps zuweisen? Zur besseren Vorstellung dieses Sachverhalts betrachten wir das folgende Codefragment:

```
Account a;
CreditAccount* ca;
ca = &a;
```

Da ein `Account`-Objekt nicht notwendigerweise auch ein `CreditAccount`-Objekt sein muss, gestattet C# diese Typkonvertierung nicht. Der C#-Compiler lehnt die Anweisung `ca = &a;` mit der Fehlermeldung *„'=' : cannot convert from 'Account*' to 'CreditAccount*'“* ab. Auf der anderen Seite können wir als Programmersteller mehr Informationen über ein Programm besitzen, beispielsweise über sein Verhalten zur Laufzeit – im Gegensatz zur Übersetzungszeit, die für den Compiler maßgeblich ist. Deshalb lässt sich eine Typkonvertierung von einer Basisklasse zu einer abgeleiteten Klasse mit Hilfe eines Cast-Operators durchführen. Wir überzeugen den Compiler auf diese Weise von der Richtigkeit der Typkonvertierung. In diesem Fall sprechen wir von einer *expliziten* Typkonvertierung bzw. von einem *Downcast* (Typkonvertierung von der Oberklasse zur Unterklasse):

```
Account a;
CreditAccount* ca;
ca = (CreditAccount*) &a;
```

Natürlich stellt sich im letzten Codefragment die Frage, wie eine solche Typkonvertierung konkret ausgeführt werden soll, wenn zur Laufzeit das tatsächlich vorliegende Objekt nicht vom abgeleiteten Typ ist, sondern eine Instanz des Basisklassentyps ist (also im betrachteten Beispiel ein `Account`-Objekt vorliegt). Die Antwort ist vergleichsweise einfach: Diese Typkonvertierung ist generell nicht durchführbar, das Programm setzt die Ausführung an dieser Stelle unkontrolliert fort. Auf meinem Rechner bedeutet dies zum Beispiel, dass eine Anweisung der Gestalt

```
printf ("%g\n", ca -> m_limit);
```

das Resultat

```
3.59807e-217
```

produziert. Es hat ein lesender Zugriff auf einen Speicherbereich statt gefunden, der weder zulässig ist und unter gewissen Umständen auch zu einem Abbruch des Programms führt. Downcasts spielen in der Praxis eine eher untergeordnete Rolle. Wir gehen statt dessen im nächsten Abschnitt noch einmal vertieft auf den Upcast ein.

5.1.2. Virtuelle Methoden

Für unsere Betrachtungen zur Typkonvertierung eines Objekts betrachten wir das folgende Codefragment:

```
CreditAccount ca;
Account* a = &ca;
a -> Withdrawal (0.0);
```

Die Typkonvertierung des `CreditAccount`-Objekts in den Basisklassentyp hat in dieser Situation geklappt, wir stehen nun vor der Frage: Welche Version der `Withdrawal`-Methode wird zur Laufzeit eigentlich aufgerufen? Wir wollen dabei zu Grunde legen, dass diese Methode zum Abheben eines

bestimmten Geldbetrags in beiden Klassen definiert ist:

```
01: class Account
02: {
03: public:
04:     virtual bool Withdrawal (double amount);
05: };
06:
07: bool Account::Withdrawal (double amount)
08: {
09:     printf ("Account::Withdrawal\n");
10:     return true;
11: }
12:
13: class CreditAccount : public Account
14: {
15: public:
16:     double m_limit;
17:
18: public:
19:     bool Withdrawal (double amount);
20: };
21:
22: bool CreditAccount::Withdrawal (double amount)
23: {
24:     printf ("CreditAccount::Withdrawal\n");
25:     return true;
26: }
```

Beispiel 5.1. Eine Methode `Withdrawal` in zwei Ausprägungen.

Auf der einen Seite ist die Methode `Withdrawal` aus der Klasse `Account` ein Kandidat, da die Zeigervariable `a` laut ihrer Definition im Quelltext vom Typ `Account*` ist. Auf der anderen Seite wäre auch ein Aufruf der `Withdrawal`-Methode aus der Klasse `CreditAccount` naheliegend, da die Zeigervariable `a` zur Laufzeit ja auf ein Objekt dieses Typs verweist. Die richtige Antwort lautet: *Beide* Varianten sind möglich! Entscheidend für diese Antwort ist der Sachverhalt, wie die `Withdrawal`-Methode in ihrer *Basisklasse* definiert ist. Haben wir eine Definition der Gestalt

```
public:
    bool Withdrawal (double amount);
```

vorliegen (siehe Zeile 4 von Listing 5.1), wird bei der Anweisung

```
a -> Withdrawal (0.0);
```

die `Withdrawal`-Methode der `Account`-Klasse aufgerufen. Es werden in dieser Variante die vom Compiler zur Übersetzungszeit bekannten Informationen für die Codegenerierung zu Grunde gelegt (hier: Zeigervariable `a` ist vom Typ `Account*`, ergo ist ein Aufruf der `Withdrawal`-Methode dieser Klasse zu generieren). Da die Definition einer Variablen bereits zur Übersetzungszeit bekannt ist und damit keinen Bezug zur Laufzeit besitzt, tituliert man dieses Laufzeitverhalten auch als *statische Bindung* (engl. *early-binding*), sprich die Bindung des Methodenaufrufs an ein Objekt erfolgt zur Übersetzungszeit anhand der im Quelltext vorliegenden Informationen.

Sie vermuten es bereits, das *„early-binding“* besitzt in der Form des *„late-bindings“* ein Gegenstück, das im deutschen als *dynamische Bindung* bezeichnet wird. In dieser Facette der Vererbung ist für den Aufruf einer Methode nicht die Definition des Zeigertyps (zur Übersetzungszeit) verantwortlich, sondern der zur Laufzeit tatsächlich vorliegende Typ des Objekts. Dazu müssen wir allerdings an der Definition der `Withdrawal`-Methode in der Basisklasse eine Änderung vornehmen (siehe Zeile 2 von Listing 5.2), sie muss nun

```
01: public:
02:     virtual bool Withdrawal (double amount);
```

Beispiel 5.2. Virtuelle Definition der Methode `Withdrawal`.

lauten. Diese Variante der Methodendefinition fußt auf dem so genannten *virtuellen Methodenaufrufmechanismus*. Wie beim Verdecken von Methoden lassen sich auch mit `virtual` markierte Methoden in einer abgeleiteten Klasse neu definieren (siehe Zeile 4 von Listing 5.3)

```
01: class Account
02: {
03: public:
04:     virtual bool Withdrawal (double amount);
05: };
```

Beispiel 5.3. Überschreiben einer virtuellen Methode.

Wir sprechen in dieser Variante nun vom *überschreiben* (engl. *to override*) einer geerbten Methode. Bitte beachten Sie das englische Wort genau: Es handelt sich nicht, wie irrtümlicherweise häufig angenommen wird, um das Wort *to overwrite*, das man mit *überschreiben* übersetzen könnte. *To override* bedeutet *etw. außer Kraft setzen* oder *sich über etwas hinwegsetzen*. Diese Übersetzung trifft den Vorgang des Überschreibens einer Methode in seiner eigentlichen Bedeutung.

In einer Anweisung der Gestalt

```
a -> Withdrawal (0.0);
```

ist nicht mehr der (zur Übersetzungszeit bekannte) Zeigervariablentyp für den Methodenaufruf entscheidend, sondern der Typ des zur Laufzeit vorliegenden Objekts. Im konkreten Beispiel wird die `Withdrawal`-Methode der Klasse `CreditAccount` aufgerufen. Wie beim Verdecken kann auch beim Überschreiben einer Methode die in der Basisklasse vorliegende Implementierung weiterhin mit Hilfe des Gültigkeitsbereichsoperators benutzt werden.

Das Verhalten virtueller Methoden tritt auch ans Tageslicht, wenn der Methodenaufruf durch eine Basisklassenreferenzvariable geschieht:

```
CreditAccount ca;
Account& a = ca;
a.Withdrawal (0.0);
```

In beiden Fällen lautet die Ausgabe des Codefragments

```
CreditAccount::Withdrawal
```

5.1.3. Polymorphismus

Polymorphismus ist einer der Grundpfeiler der objektorientierten Programmierung. Dieser Begriff ist zugegebenermaßen nicht ganz einfach zu erklären. Vielleicht erkennen Sie erst an einem konkreten Beispiel, was man sich unter diesem Prinzip vorzustellen hat. In kurzen Worten eignet sich folgende prägnante Aussage zu seiner Beschreibung: „Unter *Polymorphismus* oder *Vielgestaltigkeit* versteht man die Fähigkeit existierenden Programmcodes, neuen Code aufrufen zu können“. Dank dieser Eigenschaft kann man mit diesem Mechanismus ein bestehendes System erweitern oder verbessern, ohne den vorhandenen Programmcode anfassen oder modifizieren zu müssen.

Zur Demonstration der Leistungsfähigkeit des Polymorphismus stellen wir uns exemplarisch die Aufgabe, die grafische Darstellung einer Reihe von einfachen grafischen Grundelementen in einer möglichst einfachen und eleganten Form durchzuführen. Wir wollen dazu Rechtecke und Kreise verwenden, die alle eine gemeinsame Basisklasse `Shape` spezialisieren. Die Implementierung der Basisklasse `Shape` finden Sie in Listing 5.4 vor. Zu beachten ist in Zeile 12 die virtuell deklarierte Methode `Draw`. Sie besitzt eine Realisierung und ist mit dem Schlüsselwort `virtual` markiert:

```
01: class Shape
02: {
03: protected:
04:     int m_top;
05:     int m_left;
06:
07: public:
08:     Shape (int top, int left)
09:     {
10:         m_top = top;
11:         m_left = left;
12:     }
13:
14: public:
15:     virtual void Draw ()
16:     {
17:         printf ("Not Implemented !\n");
18:     }
19: };
```

Beispiel 5.4. Basisklasse `Shape` mit einer virtuellen Methode `Draw`.

Der Quellcode für die zwei abgeleiteten Klassen `Rectangle` und `Circle` folgt in Listing 5.5. Wesentlich ist wiederum die `Draw`-Methode, die in diesen beiden Klassen überschrieben wurde:

```
01: class Rectangle : public Shape
02: {
03: public:
04:     Rectangle (int top, int left) : Shape(top, left) {}
05:
06: public:
07:     void Draw ()
08:     {
09:         printf ("Drawing a Rectangle !\n");
10:     }
11: };
12:
13: class Circle : public Shape
14: {
15: public:
16:     Circle (int top, int left) : Shape(top, left) {}
17:
18: public:
19:     void Draw ()
20:     {
21:         printf ("Drawing a Circle !\n");
22:     }
23: };
```

Beispiel 5.5. Die Klassen `Rectangle` und `Circle`: Zwei Spezialisierungen der Basisklasse `Shape`.

Das Ziel unserer Betrachtungen soll eine Menge unterschiedlicher Figuren sein, die möglichst einfach und kompakt gezeichnet werden – wobei wir natürlich an Aufrufe der unterschiedlichen `Draw`-Methoden denken. Zu diesem Zweck legen wir zunächst ein Array von `Shape`-Zeigervariablen an, das Objekte vom Typ `Shape` oder Spezialisierungen dieser Klasse aufnehmen kann:

```
Shape* shapes[3];
```

Danach erzeugen wir drei Instanzen der Klassen `Rectangle` und `Circle`:

```
Rectangle r1 (10, 20);
Circle c1 (30, 40);
Rectangle r2 (50, 60);
```

Um diese drei Objekte kompakt ansprechen zu können, legen wir ihre Adressen im Array `shapes` ab:

```
// fill array with graphical objects
shapes[0] = &r1;
shapes[1] = &c1;
shapes[2] = &r2;
```

Da ein `Rectangle`-Objekt *ein* `Shape`-Objekt *ist* wie auch ein `Circle`-Objekt *ein* `Shape`-Objekt *ist*, sind diese drei Wertzuweisungen korrekt. Damit haben wir alle Vorbereitungen getroffen, um die `Draw`-Methode ins Spiel zu bringen. Was halten Sie von folgender Wiederholungsanweisung?

```
for (int i = 0; i < 3; i ++)
    shapes[i] -> Draw ();
```

Da die `Draw`-Methode in der Basisklasse `Shape` virtuell deklariert ist und die abgeleiteten Klassen `Rectangle` und `Circle` jeweils ihre eigene Implementierung bereitstellen, gelangen in der Wiederholungsschleife in der Tat drei *unterschiedliche* Realisierungen zur Ausführung:

```
Drawing a Rectangle !
Drawing a Circle !
Drawing a Rectangle !
```

Um das faszinierende Merkmal des letzten Codefragments noch einmal auf den Punkt zu bringen: Sie können der Anweisung

```
shapes[i] -> Draw ();
```

zwar entnehmen, dass ein Aufruf einer `Draw`-Methode ansteht, aber: An *welchem* Objekt diese Methode aufgerufen wird, ist für den Leser des Quelltextes *nicht* nachvollziehbar, da das konkret vorliegende Objekt erst zur Laufzeit bekannt ist. Virtuelle Methoden verlagern die Entscheidung, welche Methode ausgeführt wird, von der Übersetzungs- zur Laufzeit. Wäre die `Draw`-Methode bei ihrer Definition in der Basisklasse *nicht* virtuell deklariert, würde in dem letzten Codefragment dreimal die `Draw`-Methode der `Shape`-Klasse ausgeführt werden!

Die Fähigkeit, in einer Anweisung unterschiedliche Methoden mit demselben Namen an verschiedenen Objekten auszuführen, wird als *Polymorphismus* bezeichnet. Polymorphismus ist ein griechisches Wort und setzt sich aus den beiden Teilen *Poly*, zu deutsch *viel*, und *Morphismus*, zu deutsch etwa *Gestalt* oder *Form*, zusammen. Polymorphismus ist identisch mit der bereits zitierten dynamischen Bindung von Methoden. In einer Basisklasse setzt Polymorphismus die Definition einer oder mehreren virtueller Methoden voraus, die in abgeleiteten Klassen zu überschreiben sind. Für den Aufruf ist eine Zeigervariable oder eine Referenz des Basisklassentyps zu verwenden. Der zur Laufzeit tatsächlich vorliegende Typ des Objekts entscheidet, welche Methode aufgerufen wird.

Damit dürfte auch verständlich geworden sein, wie es zur Aussage „existierender Programmcode ruft neuen Programmcode auf“ kommt. Wollten wir beispielsweise zu einem späteren Zeitpunkt ein zusätzliches grafisches Grundelement in das System einbringen, muss man dieses nur als Spezialisierung der `Shape`-Klasse definieren und die virtuell deklarierten Methoden passend überschreiben. Da zusätzliche Grundelemente durch einen Upcast wiederum als `Shape`-Objekte

anzusehen sind, können sie problemlos in einem `Shape`-Array abgelegt werden. Auf diese Weise werden neue Objekttypen nahtlos in bestehenden Programmcode integriert, ohne dass Anpassungen an den neuen Objekttypen erforderlich werden!

5.1.4. Vererbung anschaulich betrachtet

Durch die Fähigkeit, in den Methoden einer Unterklasse Methoden der Oberklasse aufrufen zu können, gelangen wir zu einer neuen Betrachtungsweise, wie sich das Vererben von Methoden für den Entwickler anschaulich betrachtet darstellen kann. Ausgangspunkt unserer Überlegungen ist eine Basisklasse `A`, die eine Methode `f` durch

```
class A
{
public:
    void f () {}
};
```

oder

```
class A
{
public:
    virtual void f () {}
};
```

definiert. Spezialisieren wir Klasse `A` durch eine Klasse `B`, ergeben sich für Methode `f` grundsätzlich die drei in Tabelle 5.1 beschriebenen Betrachtungsweisen – unabhängig davon, ob die `f`-Methode virtuell definiert ist oder nicht:

Strategie	Beschreibung

Erbschaft	Methode `f` wird in der abgeleiteten Klasse nicht überschrieben:
	`class B : public A`
	`{`
	`};`
	Diese Variante rangiert unter der klassischen Bezeichnung der *Vererbung*: Klasse `B` nimmt die Implementierung der `f`-Methode uneingeschränkt für sich in Anspruch. Es ist nicht erforderlich, in der abgeleiteten Klasse mit Hilfe des Verdeckens oder Überschreibens Modifikationen irgendwelcher Art an der geerbten Methode vorzunehmen.

Ersetzung	Methode `f` wird in der abgeleiteten Klasse überschrieben, es erfolgt *kein* Aufruf der Implementierung von `f` in der Vaterklasse:
	`class B : public A`
	`{`
	`  void f () {}`
	`};`
	Die in der Vaterklasse existierende Implementierung wird – aus welchen Gründen auch immer – ignoriert. Man spricht nun vom *Ersatz* der Methode, da durch das Verdecken oder Überschreiben der Methode die in der Vaterklasse vorliegende Implementierung durch eine andere ausgetauscht wird.

Änderung	Methode `f` wird in der abgeleiteten Klasse überschrieben *und* es erfolgt ein Aufruf der Implementierung von `f` aus der Vaterklasse:
	`class B : public A`
	`{`
	`  void f ()`
	`  {`
	`  // ...`
	`  A::f();`
	`  // ...`
	`  }`
	`};`
	
	In der dritten Variante liegt eine Mischung aus den ersten beiden Varianten vor. Zum einen wird die Methode in der abgeleiteten Klasse überschrieben (oder verdeckt), die in der Vaterklasse vorliegende Implementierung bedarf also einer Anpassung in der Unterklasse. Zum anderen wird die existierende Implementierung aus der Vaterklasse nicht ignoriert, sondern in der überschriebenen Methode durch ihren Aufruf mit eingebunden. Wir sprechen nun von einer *Änderung* oder *Modifikation* der vorhandenen Implementierung.

Tabelle 5.1. Drei unterschiedliche Betrachtungsweisen für das Überschreiben von Methoden.

5.2. Abstrakte Klassen

Wir fahren mit unseren Betrachtungen zum Polymorphismus fort und wenden uns der Klasse `Shape` aus dem Beispiel dieses Abschnitts zu (siehe Listing 5.4). Die Klasse `Shape` beherbergte in dieser Realisierung zum einen eine Implementierung der Methode `Draw`, die allerdings mangels Kenntnis realer geometrischer Figuren nur eine Fehlermeldung ausgibt. Zum anderen war die Methode virtuell definiert und stellte damit eine Aufforderung für abgeleitete Klassen dar, diese Methode mit einer sinnvollen Implementierung zu überschreiben. Möchte man den Stellenwert, den bestimmte Methoden im Sinne eines Kontrakts für abgeleitete Klassen besitzen, besonders hervorheben, kann man solche Methoden auch in einer Basisklasse ohne Implementierung definieren. Zu diesem Zweck ersetzt man den Rumpf durch die Zeichenfolge „`= 0;`“. Methoden ohne Implementierung nennt man *abstrakte* Methoden oder im Fachjargon von C++ auch *pure virtuell*. Abstrakte Methoden siedelt man in einer Oberklasse an, wenn ihre Implementierung für alle Unterklassen sinnvoll und erwünscht ist (Listing 5.6):

```
01: class Shape
02: {
03: protected:
04:     int m_top;
05:     int m_left;
06:
07: public:
08:     // c'tor
09:     Shape (int top, int left)
10:     {
11:         m_top = top;
12:         m_left = left;
13:     }
14:
15: public:
16:     // method(s)
17:     virtual void Draw () = 0;
18: };
```

Beispiel 5.6. Definition einer Klasse `Shape` mit einer abstrakten Methode `Draw`.

Besitzt eine Klasse mindestens eine abstrakte Methode, so bezeichnet man auch die Klasse als *abstrakt*. Natürlich sind abstrakte Methoden implizit virtuell. Sie sind ja dazu da, in den abgeleiteten Klassen mit einer konkreten Implementierung überschrieben zu werden. Die beiden Klassen `Rectangle` und `Circle` aus Listing 5.5 können unverändert herangezogen werden, um eine konkrete Spezialisierung der `Shape`-Klasse zu bilden.

Von einer abstrakten Klasse lassen sich keine Objekte anlegen, der C++-Compiler lehnt ein derartiges Ansinnen ab („*'Shape' : cannot instantiate abstract class*“). Da eine abstrakte Klasse nicht vollständig implementiert ist, macht dies auch keinen Sinn. Eine abstrakte Klasse stellt ausschließlich eine Vorlage für potentielle Unterklassen dar. Diese Klassen stehen dann in der Pflicht, die noch „offenen Punkte“ in der Definition der Oberklasse zu konkretisieren, sprich sie müssen die abstrakten Methoden ihrer Oberklasse überschreiben. Den Umstand, dass man zu einer abstrakten Klasse kein Objekt anlegen kann, darf man nicht damit verwechseln, dass Zeigervariablen oder Referenzen eines abstrakten Klassentyps definierbar sind, also beispielsweise

```
Rectangle r (10, 20);
Shape* ps = &r;
Shape& rs = r;
```

Da Objekte von Unterklassen des Typs `Shape` auch als `Shape`-Objekte angesehen werden können, kann man diese einer Zeigervariablen eines abstrakten Klassentyps zuweisen. Welchen Vorteil gewinnt man dadurch? Wir können das Beispiel aus dem letzten Abschnitt zum universellen Zeichnen aller

Figuren nun auf der Basis einer abstrakten Vaterklasse umformulieren (Listing 5.7):

```
01: Shape* shapes[3];
02:
03: Rectangle r1 (10, 20);
04: Circle c1 (30, 40);
05: Rectangle r2 (50, 60);
06:
07: shapes[0] = &r1;
08: shapes[1] = &c1;
09: shapes[2] = &r2;
10:
11: for (int i = 0; i < 3; i ++)
12:     shapes[i] -> Draw ();
```

Beispiel 5.7. Polymorphismus auf Basis einer abstrakten Klasse `Shape`.

Die Ausgabe des Programms aus Listing 5.7 lautet wiederum

```
Drawing a Rectangle !
Drawing a Circle !
Drawing a Rectangle !
```

und ist damit identisch zum letzten Beispiel.

5.3. Virtuelle Destruktoren

Im Abschnitt über „Polymorphismus“ sind wir auf die Eigenschaft der dynamischen Bindung von Methoden eingegangen. Dieses Merkmal dürfen wir auch dann nicht außer Acht lassen, wenn es um die Aufräumarbeiten eines Objekts geht, sprich um die Ausführung des Destruktors. Sollten wir – wie bislang in diesem Buch praktiziert – den Destruktor nicht virtuell definiert haben, kann es im Umfeld des virtuellen Methodenaufrufmechanismus passieren, dass bei Verwendung von Zeigervariablen nicht der „richtige“ Destruktor aufgerufen wird. Wir demonstrieren diese potentielle Fehlerquelle an einem Beispiel (Listing 5.8), das zu diesem Zweck zwei vereinfachte Implementierungen der Klassen `Shape` und `Rectangle` besitzt:

```
01: class Shape
02: {
03: public:
04:     // c'tor
05:     Shape () { printf ("c'tor Shape\n"); }
06:
07:     // d'tor
08:     ~Shape () { printf ("d'tor Shape\n"); }
09: };
10:
11: class Rectangle : public Shape
12: {
13: public:
14:     // c'tor
15:     Rectangle () { printf ("c'tor Rectangle\n"); }
16:
17:     // d'tor
18:     ~Rectangle () { printf ("d'tor Rectangle\n"); }
19: };
```

Beispiel 5.8. Destruktoren im Umfeld des virtuellen Methodenaufrufmechanismus.

Welcher Destruktor wird in der folgenden Anwendung der Klasse `Rectangle` aufgerufen?

```
void main ()
{
    Shape* ps = new Rectangle;
    delete ps;
}
```

Es wird ausschließlich der Destruktor der Basisklasse `Shape` ausgeführt, wie wir in der Ausführung des Codefragments erkennen können:

```
c'tor Shape
c'tor Rectangle
d'tor Shape
```

Obwohl die Klasse `Rectangle` einen Destruktor besitzt und eine Instanz dieses Klassentyps angelegt wurde, wird der Destruktor diese Klasse ignoriert! Wir müssen deshalb die Definition des Destruktors in der Basisklasse um das Schlüsselwort `virtual` erweitern:

```
01: class Shape
02: {
03: public:
04:     // c'tor
05:     Shape () { printf ("c'tor Shape\n"); }
06:
07:     // d'tor
08:     virtual ~Shape () { printf ("d'tor Shape\n"); }
```

```
09: };
```

Beispiel 5.9. Definition eines virtuellen Destruktors.

Mit dieser Klassendefinition erhalten wir nun genau das Verhalten, das wir zur Laufzeit erwarten: Es wird sowohl der Destruktor von `Rectangle` als auch der Destruktor von `Rectangle` aufgerufen, bevor der Speicher des `Rectangle`-Objekts freigegeben wird:

```
c'tor Shape
c'tor Rectangle
d'tor Rectangle
d'tor Shape
```

Beabsichtigen Sie ein Objekt einer abgeleiteten Klasse durch einen Basisklassenzeiger zu löschen, so muss die Basisklasse immer einen virtuellen Destruktor besitzen. Stellen Sie sicher, dass in Basisklassen der Destruktor immer virtuell definiert ist.

Kapitel 6. Ein- und Ausgabe

6.1. Einführung

In Analogie zur Programmiersprache C besitzt auch C++ keine eigenen Sprachmittel für die Ein- und Ausgabe. Diese werden durch spezielle Klassen der C++-Standardklassenbibliothek zur Verfügung gestellt. Daneben unterstützt C++ prinzipiell auch weiterhin die aus der *C-Runtime-Library* stammenden Bibliotheksfunktionen wie `printf`, `scanf` usw. Diese klassischen Funktionen geben jedoch sehr einfach Anlass zu fehlerhafter Software, wie am Beispiel der trivialen Anweisung

```
printf ("Value of PI: %s", 3.14);
```

erkennbar ist. Die nicht vorhandene Überprüfung einer Formatanweisung (im Beispiel: `%s`) und ihrem korrespondierendem Argument (im Beispiel: `3.14`) auf Übereinstimmung ist eine der großen Schwachstellen bei der Ein-/Ausgabe von C. Aus diesem Grund bietet C++ eine völlig neue Ein- und Ausgabeschnittstelle an, die erstmals 1984 von Bjarne Stroustrup vorgestellt und 1989 von Jerry Schwarz im Rahmen der ISO-Standardisierung grundlegend überarbeitet wurde. Sie basiert auf einer Reihe von Klassen, die im Namensraum `std` deklariert sind und durch die Include-Dateien `iostream` erreichbar sind. Die beiden wichtigsten Klassen lauten `istream` und `ostream`, sie sind für die Ein- und Ausgabe auf einer Konsole verantwortlich. Um beispielsweise eine Ausgabe auf dem Standardausgabegerät `cout` durchzuführen, lautet ein minimales C++-Programm

```
#include <iostream>
void main ()
{
    std::cout.write ("ABC", 3);
}
```

oder ohne die etwas störende Qualifizierung mit `std::` auch

```
#include <iostream>
using namespace std;
void main ()
{
    cout.write ("ABC", 3);
}
```

Bevor wir auf die Details bei der Ein- und Ausgabe von C++ eingehen, erläutern wir zunächst den zentralen Begriff des *Datenstroms*.

6.2. Ein- und Ausgabe mit Datenströmen

6.2.1. Datenströme

Die Ein- und Ausgabe in C++ basiert auf dem Konzept eines *Datenstroms* (engl. *stream*). Die Ein- und Ausgabe sämtlicher – auch noch so komplex zusammengesetzter – Daten wird im Datenstrom-Modell auf Bytefolgen abgebildet. Diese fließen von einer Datenquelle (*data source*) zu einer Datensenke (*data sink*). In seiner allgemeinen Form enthält der Datenstrom noch keine Beschreibung der Datenquelle (woher kommen die Daten) und der Datensenke (wohin gehen sie). Dies wird erst durch entsprechende Spezialisierung einer verallgemeinerten, abstrakten Datenstromklasse erreicht.

Softwaretechnisch werden Datenströme über spezielle Objekte (*stream*-Objekt) angesprochen, deren Eigenschaften (Zustände und Methoden) durch Klassen beschrieben werden. Ein Datenstromobjekt lässt sich sowohl Dateien als auch Geräten (Bildschirm, Tastatur, usw.) zuordnen. Auf diese Weise ist der Zugriff auf Dateien und Geräte völlig gleichartig in der Hantierung. In Abbildung 6.1 sind drei Datenstromobjekte für die wichtigsten Standard-Ein-/Ausgabe-Geräte illustriert:

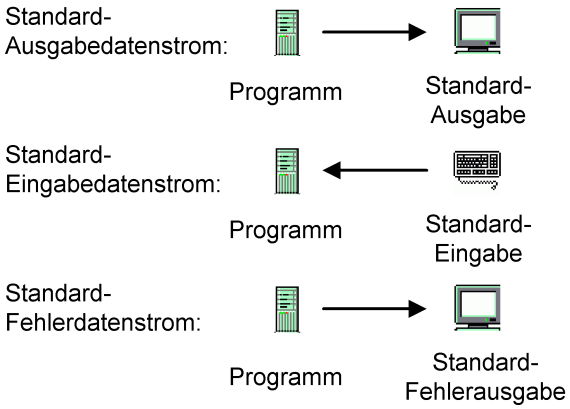

Abbildung 6.1. Standard-Datenströme.

Die C++-Klassenbibliothek kennt folgende Klassen für Datenströme:

- `istream` – Eingabedatenstrom (engl. *input stream*)

  Die Klasse `istream` (genauer: die durch Instanziierung der Template-Klasse mit einem Zeichentyp konkretisierte Klasse) stellt alle notwendigen Operationen zum Lesen der Standard-Datentypen bereit.

- `ostream` – Ausgabedatenstrom (engl. *output stream*)

  Die Klasse `ostream` (genauer: die durch Instanziierung der Template-Klasse mit einem Zeichentyp konkretisierte Klasse) stellt alle notwendigen Operationen zum Schreiben der Standard-Datentypen bereit.

- `iostream` – Ein- und Ausgabedatenstrom (engl. *input/output stream*)

  Die Klasse `iostream` ist von den beiden Klassen `istream` und `ostream` abgeleitet und dient als Basisklasse für weitere Datenstromklassen

In Erweiterung des Betriebssystems UNIX/Linux stellt die C++-Klassenbibliothek acht vordefinierte Objekte für die Standardein- und -ausgabe zur Verfügung. Diese Objekte kann man in zwei Gruppen einteilen: Objekte, die einen byte-orientierten Datentransfer durchführen und solche Objekte, die UNICODE-Zeichen (*wide characters*) transferieren.

- `cin` und `wcin` – Standardeingabe (`istream`-Datenstrom), in der Regel die Eingabe über eine Tastatur.

- `cout` und `wcout` – Standardausgabe (`ostream`-Datenstrom), in der Regel der Bildschirm.
- `cerr` und `wcerr` – ungepufferte Standardfehlerausgabe (`ostream`-Datenstrom)
- `clog` und `wclog` – gepufferte Standardfehlerausgabe (`ostream`-Datenstrom)

Die drei Datenstromkategorien *in*, *out* und *err* entsprechen den drei Standarddateizeigern `stdin`, `stdout` und `stderr` in C. Eine gepufferte Fehlerausgabe gibt es in C nicht. Ferner ist zu beachten, dass `cin` und `cout` (bzw. `cin` und `cout`) miteinander verbunden sind, sprich wenn eine Eingabe von `cin` gelesen werden soll, wird vorher der Puffer von `cout` entleert.

Diese Objekte sind dem Namensraum `std` zugeordnet. Ohne weitere `namespace`-Direktiven sind die deshalb über `std::cin`, `std::cout` bzw. `std::cerr` anzusprechen.

6.2.2. Hierarchie der wichtigsten Datenstrom-Klassen

Viele Klassen der C++-Ein- und Ausgabe sind in Wirklichkeit Instanziierungen einer Klassenschablone. In Abbildung 2.1 finden Sie deshalb zunächst einen Überblick über die wichtigsten Datenstromklassen und -Schablonen der C++-Standardklassenbibliothek vor. Mit Ausnahme der Basisklasse `ios_base` handelt es sich ausschließlich um Schablonen.

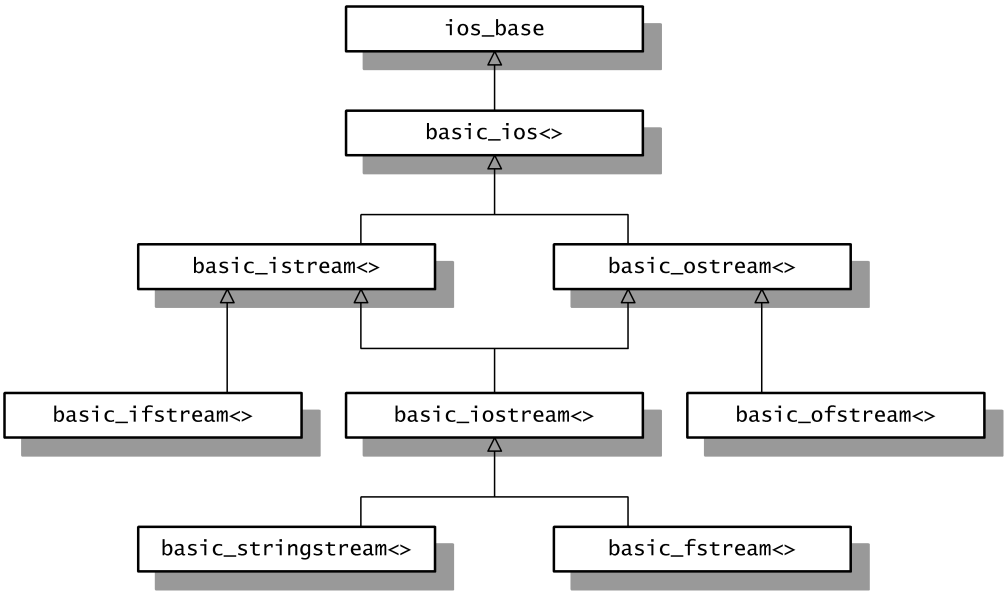

Abbildung 6.2. Klassendiagramm der Datenstromklassen und -Schablonen der C++-Standardklassenbibliothek.

Wir widmen uns nun der Aufgabe, für die konkreten Klassen, die Sie bei Ein- oder Ausgabe verwenden, eine Zuordnung zur benutzten Datenstromschablone zu treffen. Dabei legen wir in Tabelle 6.1 als Parameter der Schablone immer den Datentyp `char` zu Grunde:

Name	Typ	Beschreibung	Header-Datei	Konkrete Klasse
`ios_base`	Basisklasse	Definition prinzipieller Eigenschaften aller Datenstrom-Klassen, die unabhängig vom Zeichenformat und den daraus abgeleiteten Eigenschaften sind (z.B. Status- u. Formatflags).	`<xiosbase>`	-
`basic_ios<>`	abgeleitete Klassen-	Definition prinzipieller Eigenschaften aller Datenstrom-		

	Schablone	Klassen, die vom Zeichenformat abhängig sind (z.B. Erzeugung des Datenpuffers, dieser ist ein Objekt der Klasse `basic_streambuf<>`)	`<ios>`	`ios`
`basic_istream<>`	abgeleitete Klassen-Schablone	Beschreibung von Klassen für Objekte, die das Lesen von Zeichen aus Datenströmen ermöglichen.	`<istream>`	`istream`
`basic_ostream<>`	abgeleitete Klassen-Schablone	Beschreibung von Klassen für Objekte, die das Schreiben von Zeichen in Datenströme ermöglicht.	`<ostream>`	`ostream`
`basic_iostream<>`	abgeleitete Klassen-Schablone	Beschreibung von Klassen für Objekte, die das gleichzeitige Lesen und Schreiben von Zeichen eines Datenstroms ermöglicht.	`<iostream>`	`iostream`
`basic_ifstream<>`	abgeleitete Klassen-Schablone	Beschreibung von Klassen für Objekte, die das Lesen aus Dateien ermöglichen.	`<fstream>`	`ifstream`
`basic_ofstream<>`	abgeleitete Klassen-Schablone	Beschreibung von Klassen für Objekte, die das Schreiben in Dateien ermöglichen.	`<fstream>`	`ofstream`
`basic_fstream<>`	abgeleitete Klassen-Schablone	Beschreibung von Klassen für Objekte, die das gleichzeitige Lesen und Schreiben einer Datei ermöglichen.	`<fstream>`	`fstream`

Tabelle 6.1. Die wichtigsten Datenstrom-Klassen der C++-Standardbibliothek

Auf einer tieferen Ebene besitzt jedes Datenstrom-Objekt einen Zeichenpuffer zum Lesen bzw. Schreiben (`streambuf`), in den die Ausgabe der formatierten Daten erfolgt. Das `streambuf`-Objekt abstrahiert sozusagen das physikalische Medium, über das die Ein- und Ausgabe abgewickelt wird, während die von `istream` bzw. `ostream` abgeleiteten Klassen die Formatierung besorgen. Zum binären, unformatierten Schreiben lassen sich Variableninhalte auch direkt (und sehr schnell) in diesen Puffer schreiben. Wir stellen auch diese Klassen in einer kleinen Übersicht in Tabelle 6.2 vor:

Name	**Typ**	**Beschreibung**	**Header-Datei**	**Konkrete Klasse**
`basic_streambuf<>`	Basisklassen-Schablone	Beschreibung von Klassen zur Realisierung eines Datenpuffers für den Datenstrom-Zugriff. Datenpuffer-Objekte sind für die Durchführung des eigentlichen Datentransfers zuständig.	`<streambuf>`	`streambuf`

`basic_filebuf<>`	abgeleitete Klassen-Schablone	Beschreibung von Klassen zur Realisierung eines Datenpuffers für ein Datenstrom-Objekt einer Datei. Objekte dieses Typs sind für die Durchführung des eigentlichen Dateizugriffs zuständig.	`<fstream>`	`filebuf`

Tabelle 6.2. Datenpufferklassen der C++-Standardbibliothek

6.3. Ausgabe

6.3.1. Standard-Ausgabe

Die Standard-Ausgabe in C++ erfolgt über vordefinierte Objekte der Klassen `ostream`. Um es präziser zum Ausdruck zu bringen: In Wirklichkeit ist die Klasse `ostream` eine Spezialisierung der Schablone `basic_ostream` auf Basis des aktuellen Parameters `char` sowie einer Spezialisierung einer zweiten Schablone `char_traits`, die ebenfalls mit dem Datentyp `char` instanziiert wird:

```
typedef basic_ostream <char, char_traits<char> > ostream;
```

Die Aufgabe der `char_traits`-Schablone besteht darin, alle Eigenschaften eines konkreten Elements innerhalb eines Datenstroms soweit zu abstrahieren, dass dieser letztlich nur an Hand eines `char_traits`-Objekts instanziierbar ist. Zu diesem Zweck werden die Eigenschaften von Zeichen durch deren Zeichenmerkmale (engl. *character traits*) definiert. Eine solche Beschreibung ist die Spezialisierung der Schablone `char_traits`:

```
template <class char_type> struct char_traits { ... };
```

Diese Klasse enthält neben typedefs wie `pos_type` oder `char_type` auch Funktionen, die einzelne Zeichen oder auch Zeichenketten vergleichen (Gleichheit und Kleiner-Relation), die Zeichen und Zeichenketten effizient kopieren, verschieben und auffinden, die die Länge von Zeichenketten bestimmen, die nach bzw. von `int` wandeln und die das spezielle Zeichen `EOF` (*end of file*) definieren. Dank dieser Flexibilität eines Zeichenmerkmalobjekts können ein-byte und multi-byte Datenströme aus einer gemeinsamen Templateklasse instanziiert werden, es lassen sich sogar auch problemlos eigene Stromklassen erstellen.

Zurück zur Klasse `ostream` bzw. zur Schablone `basic_ostream`. Der Einfügeoperator << (engl. *insertion operator*) ist das Kernelement der Klasse `ostream` bzw. der zu Grunde liegenden Schablone `basic_ostream`. Er deutet bei seiner Anwendung an, dass alles, was rechts davon steht, zum betroffenen Ausgabestromobjekt fließt, also zum Beispiel

```
cout << 123;
```

Die Einfügeoperator ist für nahezu alle elementaren C/C++-Datentypen in der `basic_ostream`-Schablone bereits vorhanden, siehe Listing 6.1:

```
01: basic_ostream& operator<< (bool val);
02: basic_ostream& operator<< (short val);
03: basic_ostream& operator<< (unsigned short val);
04: basic_ostream& operator<< (int val);
05: basic_ostream& operator<< (unsigned int val);
06: basic_ostream& operator<< (long val);
07: basic_ostream& operator<< (unsigned long val);
08: basic_ostream& operator<< (float val);
09: basic_ostream& operator<< (double val);
10: basic_ostream& operator<< (long double val);
11: basic_ostream& operator<< (const void* val);
12:
13: basic_ostream& operator<< (basic_ostream& (*pfn)(basic_ostream&));
14: basic_ostream& operator<< (ios_base& (*pfn)(ios_base&));
15: basic_ostream& operator<< (basic_ios<Elem, Tr>& (*pfn)(basic_ios<Elem, Tr>&));
16: basic_ostream& operator<< (basic_streambuf<Elem, Tr> *strbuf);
```

Beispiel 6.1. Überladungen des Einfügeoperators << in der Klasse `basic_ostream`.

In Listing 6.1 sind einige Spezialfälle enthalten. Die Überladung in Zeile 13 ist für den `endl`-Manipulator konzipiert, die zwei anderen Überladungen in Zeile 14 und 15 für andere Manipulatoren wie beispielsweise `hex`. Der Parameter `pfn` ist ein Funktionszeiger, der konform zum jeweiligen Manipulatortyp ist. Auf Manipulatoren gehen wir am Ende dieses Kapitels intensiver ein.

Die Überladung aus Zeile 16 entnimmt die Elemente aus einem `stream_buf`-Objekt und fügt sie in

den Datenstrom ein. Die Operation ist beendet, wenn im `stream_buf`-Objekt das *end-of-file*-Kriterium gegeben ist. In Zeile 11 ist eine Version des Einfügeoperators für nicht typisierte Zeiger enthalten (`const void*`). Mit dieser Überladung kann man den Wert einer Zeigervariablen als hexadezimale Adressen ausgegeben. Zeiger auf Zeichen `char*` werden dagegen als C-Zeichenketten aufgefasst. Möchte man die Adresse einer C-Zeichenkette ausgeben, muss man ihren Typ vorher nach `void*` casten:

```
char* cp = "Hello C++!";
cout << cp << endl;          // prints string
cout << (void*) cp << endl; // prints address of string
```

Ausgabe:

```
Hello C++!
004177A8
```

Die Zeilen 1 bis 10 von Listing 6.1 sind auf alle elementaren Datentypen von C++ zugeschnitten. Für ganzzahlige Datentypen liegen jeweils zwei Überladungen vor, um zwischen einer vorzeichenbehafteten und einer vorzeichenlosen Ausgabe unterscheiden können:

```
int n = 0;
unsigned int m = 0;
n --;
m --;
cout << n << '\n';
cout << m << '\n';
```

Ausgabe:

```
-1
4294967295
```

Der Clou bei der Definition des Einfügeoperators liegt darin, dass dieser eine Referenz auf ihren linken Operanden zurückliefert. So ist eine mehrfache Anwendung der <<-Operation hintereinander möglich. Beispielsweise bedeutet

```
cout << i << j;
```

ausführlicher

```
(cout << i) << j;
```

Das Ergebnis von „`cout << i`“ ist das Datenstromobjekt `cout` im Zustand nach der Ausführung der <<-Operation. Dieser Datenstrom ist dann der linke Operand für die Insertion von `j`. Die einzelnen Werte, die mit dem Einfügeoperator ausgegeben werden sollen, können natürlich auch unterschiedlichen Typs sein, sie sind ausschließlich durch den <<-Operator zu trennen. Die Anweisung

```
cout << "Value of pi: " << 3.14159 << " ...";
```

gibt auf der Konsole

```
Value of pi: 3.14159 ...
```

aus. Ungeachtet von dieser komfortablen Schreibweise darf man natürlich nicht übersehen, dass alle C/C++-Operatoren ihre Priorität und Assoziativität beibehalten. Im Bedarfsfall muss man Klammern verwenden, wenn man von den Standardregeln abweichen möchte. Möchte man die binäre Darstellung der Zahl 1 um 3 Bits nach links schieben und den resultierenden Wert auf der Konsole ausgeben, dann führt die Anweisung

```
cout << "Value of 1 << 3: " << 1 << 3 << '.';
```

nicht zum Ziel. Die Ausgabe lautet

```
Value of 1 << 3: 13.
```

und nicht wie erwartet

```
Value of 1 << 3: 8.
```

Folgende Modifikation der Anweisung führt hingegen zum gewünschten Resultat:

```
cout << "Value of 1 << 3: " << (1 << 3) << '.';
```

Neben dem Einfügeoperator besitzt die Klasse `ostream` auch eine Reihe von Methoden (Tabelle 6.3):

Methode	Beschreibung
put	`basic_ostream& put(char_type ch);` Legt ein einzelnes Zeichen im zu Grunde liegenden Datenstrom ab.
flush	`basic_ostream& flush();` Leert den Ausgabepuffer, indem alle darin vorhandenen Zeichen ausgegeben werden.
write	`basic_ostream& write(const char_type *str, streamsize count);` Legt alle Zeichen ab der Adresse `str` im Datenstrom ab. Die Anzahl der Zeichen wird durch den zweiten Parameter `count` festgelegt.

Tabelle 6.3. Ein- und Ausgabemethoden der Klasse ***`ostream`***

In Tabelle 6.3 ist vor allem die Methode `write` hervorzuheben. Da `write` auch die Anzahl der auszugebenden Zeichen erwartet, kann sie nicht nur Null-terminierte Zeichenketten, sondern auch binäre Daten ausgeben, also zum Beispiel `char`-Arrays, die binäre Nullen enthalten. Die `write`-Methode bietet somit einen größeren Funktionsumfang an als der korrespondierende Einfügeoperator:

```
char charsIncludingZero[] = { '1', '2', '3', 0, '4', '5', '6' };
cout.write (charsIncludingZero, 7);
cout.put ('.');
```

Das vorstehende Codefragment führt in der Konsole zur Ausgabe

```
123 456.
```

Man erkennt, dass alle 7 Zeichen des `char`-Arrays ausgegeben werden, die binäre Null wird durch ein Leerzeichen vertreten.

6.3.2. Ausgabe benutzerdefinierter Datentypen

Für die Ausgabe benutzerdefinierter Datentypen wäre es elegant, wenn wie bei der Ausgabe der Standarddatentypen der Einfügeoperator zum Zuge kommen könnte. Es bietet sich also an, den `<<`-Operator zu diesem Zweck außerhalb einer benutzerdefinierten Klassen als globale Funktion zu definieren. Wir demonstrieren diese Vorgehensweise am Beispiel der Klasse `Fraction`:

```
// output operator
ostream& operator<< (ostream& os, const Fraction& f)
{
    os << f.GetNum() << '/' << f.GetDenom();
    return os;
}
```

Beachten Sie, dass in der Realisierung auf die existierenden Implementierungen des Einfügeoperators bei den Standarddatentypen zurückgegriffen wird. Mit dieser globalen Funktion lassen sich nun `Fraction`-Objekte auf dieselbe Weise wie Standarddatentypen ausgeben:

```
extern ostream& operator<< (ostream& os, const Fraction& f);
...

Fraction f (1, 2);
cout << "f: " << f << endl;
```

Wir erkennen nebenbei in dem exemplarischen Codefragment, dass auch bei benutzerdefinierten Datentypen mehrere Aufrufe des Ausgabeoperators in einer Anweisung beliebig häufig miteinander kombinierbar sind:

```
f: 1/2
```

Die vorgestellten Implementierungen des Ausgabeoperators ist nicht in der Lage, auf die privaten Instanzvariablen der Klasse `Fraction` zuzugreifen. Da Zähler und Nenner am Beispiel der Klasse `Fraction` über *getter*- und *setter*-Methoden erreichbar sind, konnten wir diese Klippe umschiffen. Häufig ist jedoch empfehlenswerter, dem Einfügeoperator den Zugang zum privaten Zustand der betroffenen Klasse zu gestatten, ihn also als Freund der Klasse zu deklarieren:

```
01: class Fraction
02: {
03:     // output operator
04:     friend ostream& operator<< (ostream&, const Fraction&);
05:
06:     ...
07: };
```

Beispiel 6.2. `Fraction`-Klasse mit Ausgabeoperator als `friend`-Funktion.

In Listing 6.3 erkennen wir einen weiteren Vorteil, wenn der Ausgabeoperator als `friend`-Funktion realisiert wird: Die Implementierung ist effizienter, da der direkte Zugriff auf die privaten Instanzvariablen des betroffenen Objekts möglich ist:

```
01: // output operator
02: ostream& operator<< (ostream& os, const Fraction& f)
03: {
04:     os << f.m_num << '/' << f.m_denom;
05:     return os;
06: }
```

Beispiel 6.3. Realisierung der Ein- und Ausgabeoperatoren als `friend`-Funktion.

Achtung

Im Gegensatz zu vielen anderen Operatoren kann der Ausgabeoperator niemals als Elementmethode innerhalb einer Klasse definiert werden. In einer Anweisung der Gestalt

```
Fraction f (1, 2);
cout << f;
```

ist der linke Operand des `<<`-Operators immer vom Typ `ostream`. Bei Operatoren, die im Kontext einer Klasse überladen werden, kann der linke Operand immer nur vom Typ der umgebenden Klasse sein!

6.4. Eingabe

6.4.1. Standard-Eingabe

Eingaben werden – analog zu den Ausgaben – mit einer überladenen Version des Extraktionsoperators `>>` durchgeführt oder durch den Aufruf einer Methode der Klasse `istream`:

```
typedef basic_istream<char, char_traits<char> > istream;
```

Für die elementaren Datentypen liegen in der Klasse `istream` zahlreiche Überladungen vor (Listing 6.4):

```
01: basic_istream& operator>> (bool& val);
02: basic_istream& operator>> (short& val);
03: basic_istream& operator>> (unsigned short& val);
04: basic_istream& operator>> (int& val);
05: basic_istream& operator>> (unsigned int& val);
06: basic_istream& operator>> (long& val);
07: basic_istream& operator>> (unsigned long& val);
08: basic_istream& operator>> (void*& val);
09: basic_istream& operator>> (float& val);
10: basic_istream& operator>> (double& val);
11: basic_istream& operator>> (long double& val);
12:
13: basic_istream& operator>> (basic_istream& (*pfn)(basic_istream&));
14: basic_istream& operator>> (ios_base& (*pfn)(ios_base&));
15: basic_istream& operator>> (basic_ios<Elem, Tr>& (*pfn)(basic_ios<Elem, Tr>&));
16: basic_istream& operator>> (basic_streambuf<Elem, Tr>* strbuf);
```

Beispiel 6.4. Überladungen des Extraktionsoperators >> in der Klasse `basic_istream`.

Es finden sich auch hier einige spezielle Überladungen des Extraktionsoperators vor. Die Überladung aus Zeile 13 ist für den `ws`-Manipulator konzipiert, die zwei Überladungen aus den Zeilen 14 und 15 für andere Manipulatoren wie `hex` beispielsweise. Liegen die einzulesenden Daten in einem `stream_buf`-Objekt vor, ist die Überladung aus Zeile 16 anzuwenden. Alle anderen Überladungen dienen der formatierten Eingabe.

Entsprechend dem Einfügeoperator geben auch die Extraktionsoperatoren eine Referenz auf ihren linken Operanden zurück, so dass man sie auf die gleiche Art und Weise verketten kann.

Alle Extraktionsoperatoren überspringen Whitespaces vor dem Lesen. Beim Lesen von Zeichenketten werden Whitespaces übersprungen, dann Zeichen gelesen bis zum nächsten Whitespace. Grundsätzlich muss man am Extraktionsoperator für C-Zeichenketten kritisieren, dass die Länge der Zeichenkette nicht geprüft wird (nicht geprüft werden kann!), so dass beim Lesen jederzeit Speicher hinter dem physikalischen Ende der Zeichenkette überschrieben werden kann. Das kann man beispielsweise so verhindern:

```
char buf[10];
cout << "Enter string:" << endl;
cin >> setw(sizeof(buf)) >> buf;
cout << ">>>" << buf << "<<<" << endl;
```

`setw` ist ein Manipulator, der in der Include-Datei *iomanip* definiert ist. Er begrenzt die maximale Anzahl von Zeichen, die in der folgenden Operation (hier: Extraktion der Zeichenkette) übertragen werden:

```
Enter string:
12345678901234567890
>>>123456789<<<
```

Einfacher ist natürlich die Verwendung „richtiger“ Strings der C++ Standardbibliothek (`string`-Objekte), die nicht überlaufen können.

6.4.2. Eingabe benutzerdefinierter Datentypen

In Analogie zum Einfügeoperator lässt sich der Extraktionsoperator für die Eingabe benutzerdefinierter Datentypen überladen. Für die Standardtypen ist der >>-Operatoren vordefiniert. Am Beispiel der Klasse `Fraction` kann die Eingabe so aussehen:

```
// class interface
class Fraction
{
    // input operator
    friend istream& operator<< (istream&, Fraction&);

    ...
};

// input operator implementation
istream& operator>> (istream& is, Fraction& f)
{
    is >> f.m_num >> f.m_denom;
    f.Reduce();
    return is;
}

void main()
{
    Fraction f;
    cout << "enter num and denom:" << endl;

    cin >> f;
    cout << "f: " << f << endl;
}
```

Wie bei der Ausgabe sind auch mehrere Aufrufe des Eingabeoperators in einer Anweisung beliebig häufig miteinander kombinierbar. Die einzelnen einzugebenden Werte sind in diesem Fall durch *whitespaces* (Leerzeichen, Tabulatoren, Zeilenende) abzugrenzen. Der folgende Abdruck

```
enter num and denom:
3 12
g: 1/4
```

verdeutlicht eine mögliche Ausführung des Codefragments. Die hervorgehobenen Werte stellen die Eingabe von der Tastatur dar, die Werte `3` und `12` sind durch ein Leerzeichen voneinander getrennt.

6.5. Fortgeschrittene Konzepte zur Ein- und Ausgabe

6.5.1. Zustand eines Datenstroms: Status-Flags

Ein Datenstrom besitzt zu jedem Zeitpunkt einen Zustand, der durch mehrere Werte definiert ist. Der Zustand hängt vom Erfolg bzw. Misserfolg der letzten Ein-/Ausgabeoperation ab, man kann ihn nach jeder Operation abfragen oder auch verändern.

Zur Definition des Zustands gibt es in der Klasse `ios` ein Statuswort, in dem einzelne Bits (Flags) bestimmte Fehlersituationen kennzeichnen (Fehlerflags). Die betreffenden Flags sind durch Konstanten in der Klasse `ios` festgelegt (Tabelle 6.4):

Status-Flag	Beschreibung
`ios::goodbit`	Es ist kein Fehler aufgetreten. Wenn der Wert des Statuswortes gleich `ios::goodbit` ist, dann ist kein Fehlerflag gesetzt.
`ios::eofbit`	Dieses Bit ist gesetzt, falls bei der letzten Leseoperation das Dateiende erreicht wurde, d.h. wenn keine weiteren Daten mehr gelesen werden können. Wenn versucht wird, über das Dateiende hinaus zu lesen, wird zusätzlich auch das Fehlerflag `ios::failbit` gesetzt.
`ios::failbit`	Dieses Bit ist gesetzt, falls bei der letzten Ein-/Ausgabeoperation ein Fehler aufgetreten ist. Prinzipiell handelt es sich um einen nicht gravierenden I/O-Fehler. Die letzte I/O-Operation konnte zwar nicht korrekt abgeschlossen werden, der Datenstrom ist aber prinzipiell noch in Ordnung (es könnte beispielsweise ein Formatfehler beim Lesen aufgetreten sein, beim Lesen von Zahlen standen etwa Buchstaben im Datenstrom).
`ios::badbit`	Ein fataler Ein-/Ausgabefehler ist aufgetreten. Es können insbesondere keine weiteren Operationen auf diesem Datenstrom erfolgreich ausgeführt werden (z.B. Datenträger ist voll).

Tabelle 6.4. Status-Flags eines Datenstroms (Fehlerzustands-Konstanten).

Die genauen Werte der einzelnen Fehlerzustands-Konstanten und damit die zugehörigen Bits (Fehlerflags) im Statuswort sind implementierungsabhängig. Das folgende Beispiel führt beim Microsoft Visual C++ Compiler zur Ausgabe

```
int n = ios::goodbit;
cout << "ios::goodbit: " << n << endl;
n = ios::eofbit;
cout << "ios::eofbit:  " << n << endl;
n = ios::failbit;
cout << "ios::failbit: " << n << endl;
n = ios::badbit;
cout << "ios::badbit:  " << n << endl;
```

führt beim Microsoft Visual C++ Compiler zur Ausgabe

```
ios::goodbit: 0
ios::eofbit:  1
ios::failbit: 2
ios::badbit:  4
```

In jedem Datenstrom gibt es ein *Statuswort*, das den aktuellen Zustand des Datenstroms enthält. Dieses Statuswort kann nicht direkt gelesen werden (Zugriffsklasse private), es gibt jedoch eine Reihe von Methoden, die lesend als auch schreibend darauf zugreifen können (Tabelle 6.5):

Methode	Beschreibung

good	`bool good() const;` Liefert `true` zurück, wenn im Statuswort kein Fehlerflag gesetzt ist.
bad	`bool bad() const;` Liefert `true` zurück, wenn im Statuswort das Status-Flag `badbit` gesetzt ist.
eof	`bool eof() const;` Liefert `true`, wenn im Statuswort das Flag `eofbit` gesetzt ist.
fail	`bool fail() const;` Liefert `true`, wenn im Statuswort die Flags `failbit` oder `badbit` gesetzt sind.
bad	`bool bad() const;` Liefert `true`, wenn im Statuswort das Flag `badbit` gesetzt ist.
rdstate	`iostate rdstate() const;` Liefert den Zustand des Datenstroms (komplettes Statuswort) zurück.
clear	`void clear (iostate state = goodbit);` Mittels dieser Methode kann man im Statuswort alle Bits löschen (default-Parameter) oder aber auch ein neues Statuswort setzen. Setzt das Statuswort des Datenstroms auf den Wert des Parameters `status`. Dieser besitzt per Voreinstellung (Aufruf ohne Parameter, Default-Parameter) den Wert `goodbit`, also fehlerfrei.

setstate	`void setstate (iostate state);`
	Setzt die zusätzlichen Flags des Parameters `state` im Statuswort. Die gesetzt gewesenen Flags bleiben gesetzt. Ein Aufruf von `setstate` ist äquivalent zum Aufruf von
	`clear (rdstate() | state);`

Tabelle 6.5. Zugriffsmethoden für das Statuswort.

Wir demonstrieren die Anwendung der Zugriffsmethoden aus Tabelle 6.4 am folgenden Codefragment:

```
01: bool b = cout.bad( );
02: cout << b << endl;   // ok
03: cout.clear(ios::badbit);
04: b = cout.bad();
05: cout << b << endl;   // (!)
06: b = cout.fail();
07: cout << b << endl;   // (!)
08: cout.setstate(ios::failbit);
09: b = cout.fail();
10: cout.clear();
11: cout << b << endl;   // ok
```

Beispiel 6.5. Test der Statuswort-Zugriffsmethoden.

Da im Beispiel aus Listing 6.5 vier `cout`-Anweisungen enthalten sind, erwarten wir auch eine entsprechende Ausgabe von vier `bool`-Werten. Das Resultat der Ausführung ist überraschend:

```
0
1
```

Der `clear`-Methodenaufruf in Zeile 3 ist die Ursache für dieses Verhalten. Da nach dem Aufruf im Statuswort des `cout`-Objekts das `ios::badbit`-Fehlerflag gesetzt ist, werden weitere Ein- und Ausgabeoperationen in diesem Datenstrom nicht mehr ausgeführt. Die beiden `cout`-Anweisungen in den Zeilen 5 und 7 ziehen deshalb keine Ausgaben nach sich. Erst nach dem Löschen aller Fehlerflags

mit dem Aufruf von `clear` in Zeile 10 ist der Datenstrom wieder funktionstüchtig.

Neben den Methoden aus Tabelle 6.4 sind in der Klasse `basic_ios` die zwei Operatoren `operator!` und `operator void*` überladen, um eine vereinfachte Abfrage des Statusworts in arithmetischen Ausdrücken zu ermöglichen (Tabelle 6.6):

Operator	Beschreibung
`operator!() const;`	• Wert `true`: Der Datenstrom ist in einem echten Fehlerzustand, d.h. das Flag `ios::failbit` und/oder das Flag `ios::badbit` sind gesetzt. • Wert `false`: Der Datenstrom ist in keinem echten Fehlerzustand.
`operator void* () const;`	• `NULL`-Zeiger: Der Datenstrom ist in einem echten Fehlerzustand. • Zeiger ungleich `NULL`: Der Datenstrom ist in keinem echten Fehlerzustand.

Tabelle 6.6. Zugriffsoperatoren für das Statuswort.

Mit Hilfe des logischen Negationsoperators aus Tabelle 6.6 kann die Eingabe eines ganzzahligen Werts so erfolgen:

```
int n = 0;
if (cin >> n)
    cout << "Input: " << n << endl;
else
    cout << "Wrong input !!!" << endl;
```

6.5.2. Formatierung der Ein- und Ausgabe: Format-Flags

Möchte man bei der Ein- und Ausgabe in C++ nicht mit den Standardformatierungen arbeiten, kann man mit Hilfe der *Format-Flags* sowohl das Ausgabeformat wie auch das Eingabeformat vielfältig beeinflussen. Innerhalb des Format-Flags werden die meisten Format-Eigenschaften durch einzelne Bits gekennzeichnet und in einem Datenwort zusammengefasst.

Bei manchen Format-Optionen ist es nicht ausreichend, ein einzelnes Flag zu setzen. Kollidierende Flags müssen zum gleichen Zeitpunkt zurückgesetzt werden. Sollen beispielsweise die Ausgaben von ganzzahligen Werten nicht mehr dezimal, sondern hexadezimal erfolgen, muss man einerseits das Format-Flag `ios_base::hex` setzen. Auf der anderen Seite müssen die Flags `ios_base::dec` und `ios_base::oct` zurückgesetzt werden. Für solche Fälle, in denen eine Gruppe von zusammengehörenden Flags eine Ausgabeoption beschreiben, gibt es drei vordefinierte Bitmasken, die in Tabelle 6.7 zusammengestellt sind:

Bitmaske	Beschreibung
`adjustfield`	Positionierung der Ausgabe innerhalb der durch `ios_base::width()` (siehe Tabelle 6.10) festgelegten Breite des Feldes.
`basefield`	Umschaltung zwischen dezimaler, oktaler und hexadezimaler Zahlendarstellung.
`floatfield`	Einstellung der Notation von Gleitkommazahlen.

Tabelle 6.7. Vordefinierte Bitmasken für Gruppen zusammengehörender Format-Flags.

In Tabelle 6.8 finden Sie nun eine Beschreibung aller Format-Flags aus der Klasse `ios_base` vor:

Formatflag	Beschreibung	Bitfeld

`left`
`right`
`internal`
	Linksbündige Ausgabe, eventuell verbleibende Leerräume der aktuellen Feldbreite werden mit dem Füllzeichen ausgefüllt.
	Rechtsbündige Ausgabe, eventuell verbleibende Leerräume der aktuellen Feldbreite werden mit dem Füllzeichen ausgefüllt.
	Auffüllen verbleibender Leerräume in Bezug auf die aktuelle Feldbreite zwischen dem Vorzeichen und dem numerischen Wert.
		`adjustfield`

`dec`
`oct`
`hex`
	Ein- oder Ausgabe in dezimalem Format
	Ein- oder Ausgabe in oktalem Format
	Ein- oder Ausgabe in hexadezimalem Format
		`basefield`

`showbase`	Ausgabe mit Zahlensystemkennung, das die Basis des ganzzahligen Werts erkennen lässt (`0` für oktal, `0x` für hexadezimal)	
`showpoint`	Ausgabe immer mit Dezimalpunkt und gegebenenfalls Nullen nach dem Punkt (bei Gleitpunktzahlen).	
`showpos`	Explizite Ausgabe eines `'+'`-Zeichens bei positiven Zahlen.	
`uppercase`	Ausgabe von Großbuchstaben bei Hexadezimalzahlen und Gleitpunktzahlen in Exponentialdarstellung (`'E'` statt `'e'`, `'X'` statt `'x'` usw.).	
`fixed` `scientific`	Ausgabeformat von Gleitkommazahlen in Dezimalbruchdarstellung. Ausgabeformat von Gleitkommazahlen in Exponentialdarstellung.	`floatfield`
`skipws`	(Führende) Leerzeichen, Tabulatoren und Zeilenumbrüche (*white spaces*) werden bei der Eingabe übersprungen.	
`unitbuf`	Explizites Leeren des Ausgabepuffers nach jeder Ausgabeoperation.	
`boolalpha`	Ausgabe von `bool`-Werten mit den Zeichenketten „`false`“ bzw. „`true`“ an Stelle numerischer Werte	

Tabelle 6.8. Flags des Formatstatus.

Die Format-Flags werden – ähnlich wie das Statuswort eines Datenstroms – in einer Variablen vom Typ `ios::fmtflags` gespeichert. Bei den meisten C++-Implementationen handelt es dabei um eine `int`-Variable:

```
typedef int fmtflags;
```

Um diese Details der Implementierung nicht beachten müssen, greift man besser über eine Methodenschnittstelle auf einzelne Format-Flags zu (Tabelle 6.9):

Methode	Beschreibung
`flags`	`fmtflags flags() const;` Liefert den aktuellen Zustand der Format-Flags in einer Variablen des Typs `fmtflags` zurück.
`flags`	`fmtflags flags(fmtflags mask);`

	Setzt alle Formatflags auf die im Parameter `mask` enthaltenen Werte, d.h. die alten Formatflags werden überschrieben. Die alten Formatflags werden jedoch als Funktionswert zurückgegeben.
`setf`	`void setf(fmtflags mask);` Setzt die im Parameter `mask` gesetzten Formatflags, die übrigen Formatflags werden nicht beeinflusst. Die alten Formatflags werden jedoch als Funktionswert zurückgegeben.
`setf`	`fmtflags setf(fmtflags mask, fmtflags unset);` Setzt die durch den zweiten Parameter `unset` festgelegten Formatflags zurück und setzt anschließend die im ersten Parameter `mask` gesetzten Formatflags. Typische Anwendung: Setzen von Formatflags innerhalb eines Bitfelds unter gleichzeitigem Rücksetzen der übrigen Flags dieses Bitfelds (innerhalb eines der definierten Bitfelder darf immer nur ein Flag gesetzt sein). Die übrigen Formatflags (außerhalb des durch `unset` festgelegten Bitfelds) werden nicht beeinflußt. Die alten Formatflags werden als Funktionswert zurückgegeben.
`unsetf`	`void unsetf(fmtflags mask);` Setzt die im Parameter `mask` gesetzten Formatflags zurück, die übrigen Formatflags werden nicht beeinflusst. Die alten Formatflags werden als Funktionswert zurückgegeben.

Tabelle 6.9. Methoden für Format-Flags.

Wir fahren mit einigen Beispielen zu den Formatflags fort. Um beispielsweise einen ganzzahligen Wert der Reihe nach im dezimalen, hexadezimalen, oktalen und zum Abschluss wieder dezimalen Format auszugeben, kann man so vorgehen:

```
int n = 255;
cout << n << endl;
cout.setf(ios::hex, ios::basefield);
cout << n << endl;
cout.setf(ios::oct, ios::basefield);
cout << n << endl;
cout.setf(ios::dec, ios::basefield);
cout << n << endl;
```

Die Ausgabe lautet

```
255
ff
377
255
```

Die Ausgabe ganzer Werte mit oder ohne Vorzeichen kann man so gestalten:

```
int n = 255;
cout << n << endl;
cout.setf(ios::showpos);
cout << n << endl;
cout.unsetf (ios::showpos);
cout << n << endl;
```

Die entsprechende Ausgabe sieht so aus:

```
255
+255
255
```

Bei der Ausgabe boolscher Werte lässt sich die Lesbarkeit erhöhen, wenn man das Formatflag `ios::boolalpha` einsetzt:

```
bool b = true;
cout << "True: " << b << ", False: " << !b << endl;
cout.setf(ios::boolalpha);
cout << "True: " << b << ", False: " << !b << endl;
```

Enscheiden Sie selbst, welche Darstellung Ihnen besser gefällt:

```
True: 1, False: 0
True: true, False: false
```

Zum Abschluss stellen wir die Ausgabe von Gleitpunktzahlen in Dezimalbruchdarstellung und in Exponentialdarstellung gegenüber:

```
#define _USE_MATH_DEFINES
#include <math.h>
...
double pi = M_PI;
cout << pi << endl;
ios::fmtflags old = cout.setf(ios::fixed, ios::floatfield);
cout << pi << endl;
cout.setf(ios::scientific, ios::floatfield);
cout.setf(ios::uppercase);
cout << pi << endl;
cout.flags(old);
cout << pi << endl;
```

Ausgabe:

```
3.14159
3.141593
3.141593E+000
3.14159
```

Tipp

In der letzten Ausgabe wird die Gleitpunktzahl `pi` mit den unterschiedlichen Werten `3.14159` und `3.141593` ausgegeben. Dies ist kein Fehler, sondern wird nach der Lektüre des Abschnitts „Genauigkeit bei Gleitpunktzahlen“ verständlich!

6.5.3. Weitere Formateigenschaften

Zu den weiteren beeinflussbaren Format-Eigenschaften zählen

- die Feldbreite,
- die Genauigkeit bei Gleitpunktzahlen und
- Füllzeichen.

6.5.3.1. Feldbreite

Für die Ausgabe von Werten gilt prinzipiell, dass die für die Darstellung des Ausgabewertes benötigte Anzahl von Zeichen (so genannte *Feldbreite*) zum Zuge kommt. Diese kann auch auf einen anderen Wert gesetzt werden. Dabei ist zu beachten, dass eine Änderung der Feldbreite für die genau darauf folgende Ausgabe eine Wirkung erzielt und nicht etwa bis zu einer neuen, anderen Einstellung.

Zur Darstellung der Feldbreite dient der Datentyp `streamsize`, er ist ein Synonym eines ganzzahligen vorzeichenbehafteten Standard-Datentyps:

```
typedef int streamsize;
```

Zur Definition der Methode `width` in der Klasse `ios_base` siehe nun Tabelle 6.10:

Methode	Beschreibung

`width`	`streamsize width() const;`
	`streamsize width(streamsize wide);`
	Die erste Überladung liefert die aktuell eingestellte Feldbreite zurück. Mit der zweiten Überladung setzt man die Feldbreite für die nächste Ausgabe auf den Wert `wide`, der alte Wert wird zurückgegeben. Eine zu kleine Änderung der Feldbreite übt keinen Einfluss

	auf die Ausgabe aus!

Tabelle 6.10. Methode `width` für die Einstellung der Feldbreite.

Das Codefragment

```
cout << '!';
cout.width(10);
cout << 123 << '!' << endl;
```

produziert die Ausgabe

```
!       123!
```

Benötigt die Ausgabe eines Wertes tatsächlich mehr Zeichen wie durch die aktuelle Feldbreite vorgesehen sind, so wird die Feldbreite entsprechend erhöht. Das Codefragment

```
cout << '!';
cout.width (3);
cout << 123456789 << '!' << endl;
```

führt zur Ausgabe von

```
!123456789!
```

Das letzte Beispiel verdeutlicht den Aspekt, dass eine Änderung der Feldbreite nur eine Auswirkung auf die unmittelbar nachfolgende Ausgabe besitzt. Die Feldbreite wird nach jeder Ausgabe wieder auf ihren Voreinstellungswert 0 gesetzt:

```
cout.width(10);
cout << cout.width() << endl;
cout << cout.width() << endl;
```

Ausgabe:

```
        10
0
```

Die Feldbreite hat auch bei Eingaben eine Wirkung, wenn man Zeichenketten mit dem Extraktionsoperators >> einliest. Ist die Feldbreite auf einen Wert *n* eingestellt, werden maximal *n*-1 Zeichen eingelesen und mit dem `'\0'`-Zeichen abgeschlossen. Tritt vorher ein *whitespace*-Zeichen auf, werden entsprechend weniger Zeichen gelesen. Die Feldbreite legt also eine maximale Eingabefeldgröße für Zeichenketten fest. Per Voreinstellung werden immer alle Zeichen bis zum nächsten *whitespace*-Zeichen eingelesen:

Beispiel:

```
char text[16];
cin.width(8);
cin >> text;
cout << '!' << text << '!' << endl;
cin.width(6);
cin >> text;
cout << '!' << text << '!' << endl;
```

Es folgen verschiede Szenarien der Ein- und Ausgabe, die von der Tastatur eingegebenen Zeichen sind dabei fett hervorgehoben:

```
123456789012345
!1234567!
!89012!
```

Ein-/Ausgabe:

```
1234 1234
!1234!
!1234!
```

Ein-/Ausgabe:

```
12 123456789
!12!
!12345!
```

Da nach dem Einlesen einer Zeichenkette die Feldgröße wieder auf ihren Voreinstellungswert 0 gesetzt

wird, muss sie für jeden weiteren einzulesenden String erneut gesetzt werden. Beim Einlesen von Zeichenketten mit dem Extraktionsoperators sollte sicherheitshalber immer die Feldbreite auf die Größe des den String aufnehmenden Speicherbereichs begrenzt werden:

```
char text[30];
cin.width(sizeof(text));
cin >> text;
```

Tipp

Alternativ zum Aufruf der `width`-Methode gibt es auch einen Manipulator `setw`. Manipulatoren sind Thema eines noch folgenden, gleichnamigen Abschnitts!

6.5.3.2. Genauigkeit bei Gleitpunktzahlen

Gleitkommazahlen werden prinzipiell auf zwei verschiedene Arten ausgegeben: entweder in der Dezimalbruchdarstellung oder in der wissenschaftlichen Schreibweise, d. h. in der Exponentialdarstellung. Die *Genauigkeit* bei Gleitpunktzahlen legt für ihre Ausgabe sowohl in der Dezimalbruchdarstellung als auch in der Exponentialdarstellung die **Anzahl der Nachpunktstellen** fest – sofern die Darstellungsart über eines der zwei Formatflags `ios::fixed` bzw. `ios::scientific` explizit eingestellt wurde. Diese beiden Flags bilden zusammen das Bitfeld `ios::floatfield`, sie schließen sich also gegenseitig aus. Die letzte Stelle wird in diesem Fall gegebenenfalls gerundet. Abschließende Nullen nach dem Dezimalpunkt werden mit ausgegeben.

Ist *keines* der Formatflags für die Darstellungsart gesetzt, so bestimmt gemäß dem ANSI/ISO-Standard die Genauigkeit die **Gesamtanzahl der Stellen**. Wenn mit dieser Stellenzahl die Zahl als Dezimalbruch dargestellt werden kann, wird diese Darstellungsart gewählt, andernfalls die Exponentialdarstellung. Folgen nach dem Dezimalpunkt keine von `'0'` verschiedenen Ziffern und ist das Formatflag `ios::showpoint` nicht gesetzt, so wird der Dezimalpunkt weggelassen und die Anzahl der auszugebenden Stellen auf die Vorpunktstellen begrenzt.

Der Voreinstellungswert für die Genauigkeit beträgt 6. Er kann durch einen beliebigen anderen Wert ersetzt werden. Eine einmal eingestellte Genauigkeit gilt solange für alle folgenden Gleitpunktzahl-Ausgaben, bis sie explizit geändert wird. Die Genauigkeit wird mit der `precision`-Methode eingestellt (Tabelle 6.11):

Methode	Beschreibung

`precision`	`streamsize precision() const;`
	`streamsize precision(streamsize prec);`
	Die erste Überladung liefert die aktuell eingestellte Genauigkeit zurück. Mit der zweiten Überladung wird die Genauigkeit auf den Wert `prec` gesetzt, der alte Wert wird zurückgegeben.

Tabelle 6.11. Methode `precision` zur Einstellung der Genauigkeit von Gleitkommazahlen.

Wir demonstrieren die vorgestellten Aussagen zur Genauigkeit von Gleitkommazahlen am folgenden Beispiel, dessen Kommentare Sie bitte detailliert studieren:

```
double d = 12345.6789;
cout << d << endl;      // display 6 significant digits before and after the decimal point
                        // (conforming to ANSI/ISO-Standard)
cout.precision(5);      // change precision
cout << d << endl;      // display 5 significant digits before and after the decimal point
                        // (conforming to ANSI/ISO-Standard)
cout.setf(ios::fixed); // set 'fixed point' notation
cout << d << endl;      // now display 5 significant just after the decimal point
cout.setf(ios::scientific, ios::floatfield);
                        // set 'exponential' notation
cout << d << endl;      // now display 5 significant again just after the decimal point
```

```
cout.unsetf(ios::fixed | ios::scientific);
                        // clear all 'floatfield' bits, enable ANSI/ISO-Standard
cout.precision(6);      // set standard precision: 6 digits, conforming to ANSI/ISO-Standa
cout << d << endl;      // display 6 significant digits before and after the decimal point
```

Ausgabe:

```
12345.7
12346
12345.67890
1.23457e+004
12345.7
```

Ohne eines der Flags `ios::fixed` und `ios::scientific` gibt die Genauigkeit die Anzahl Stellen an, die verwendet wird. Ist `ios::showpoint` gesetzt, so wird diese Anzahl immer angezeigt, auch wenn gar kein Dezimalpunkt nötig wäre oder wenn die weiteren Stellen nur Nullen sind. Ohne `ios::showpoint` ist es die maximale Anzahl von zu benutzenden Stellen, es können aber – nach Notwendigkeit – auch weniger sein.

Ist die Anzahl der Vorkommastellen größer als mit der Genauigkeit darstellbar, wird auf die Exponentialdarstellung umgestellt. Die Mantisse bleibt erhalten, es kommen 4 Zeichen für den Exponenten hinzu: das `e`, das Vorzeichen und 2 Ziffern. Gleiches geschieht, wenn die Anzahl Stellen nicht mehr darstellbar ist ohne Exponent und ohne die Gesamtbreite der Darstellung von Genauigkeit plus 4 Zeichen zu sprengen. Bei erzwungener Festkommadarstellung mit `ios::fixed` gibt die Genauigkeit die Anzahl der Nachkommastellen an. Ohne `ios::showpoint` ist es die maximale Anzahl Stellen, mit `ios::showpoint` wird immer genau diese Anzahl Nachkommastellen verwendet. Sehr kleine Zahlen sind dann nicht mehr von Null unterscheidbar, sehr große Zahlen führen zu immer größerer Vorkomma-Stellenanzahl bei immer noch voller Nachkommastellen-Anzeige, auch wenn diese Stellen keinerlei Signifikanz mehr haben. Bei erzwungener Exponentialdarstellung mit `ios::scientific` werden immer die angegebene Anzahl Nachkommastellen und eine Vorkommastelle angezeigt, dazu ein Exponent – auch wenn dieser Null sein sollte. Hier ist die Anzahl Zeichen, die benötigt wird, garantiert für jeden Wert gleich.

Wir runden die Betrachtungen der Gleitpunktzahlen mit einem zweiten Beispiel ab. Zu Übungszwecken können Sie versuchen, die Ausgabe des folgenden Codefragments ohne seine Ausführung zu ermitteln:

```
double d = 123.4567;
double e = 12;

cout << "Default:                                    " << d << ", " << e << endl;
ios::fmtflags standard = cout.setf (ios::showpoint);
cout << "ios::showpoint:                             " << d << ", " << e << endl;
// set ios::fixed
cout.setf (ios::fixed);
cout << "ios::showpoint+ios::fixed:                  " << d << ", " << e << endl;
// clear all 'floatfield' bits, set ios::scientific
cout.setf (ios::scientific, ios::floatfield);
cout << "ios::showpoint+ios::scientific:             " << d << ", " << e << endl;
// change precision to 8 digits
int old_prec = cout.precision(8);
cout << "ios::showpoint+ios::scientific+Precision=8: " << d << ", " << e << endl;
// clear all 'floatfield' bits, set ios::fixed
cout.setf (ios::fixed, ios::floatfield);
cout << "ios::showpoint+ios::fixed+Precision=8:      " << d << ", " << e << endl;
```

Ausgabe:

```
Default:                                    123.457, 12
ios::showpoint:                             123.457, 12.0000
ios::showpoint+ios::fixed:                  123.456700, 12.000000
ios::showpoint+ios::scientific:             1.234567e+002, 1.200000e+001
ios::showpoint+ios::scientific+Precision=8: 1.23456700e+002, 1.20000000e+001
ios::showpoint+ios::fixed+Precision=8:      123.45670000, 12.00000000
```

6.5.3.3. Füllzeichen

Ist die Feldbreite für die nachfolgende Ausgabe größer als die Anzahl der auszugebenden Zeichen, wird in die nicht belegten Stellen ein *Füllzeichen* ausgegeben. Als Voreinstellung ist ein Leerzeichen als Füllzeichen eingestellt. Es kann durch ein beliebiges anderes Zeichen ersetzt werden, siehe dazu

Tabelle 6.12. Ein einmal eingestelltes Füllzeichen gilt solange, bis es explizit durch ein anderes ersetzt wird.

Methode	Beschreibung

`fill`	`char_type fill() const;`
	`char_type fill(char_type ch);`
	Die erste Überladung liefert das aktuelle Füllzeichen als Funktionswert zurück. Mit der zweiten Überladung wird das Füllzeichen auf den Wert `ch` gesetzt, das alte Füllzeichen wird als Funktionswert zurückgegeben.

Tabelle 6.12. Methode `fill` zum Setzen des Füllzeichens.

Der in Tabelle 6.12 verwendete Datentyp `char_type` ist in der Schablone `basic_ios<>` definiert:

```
typedef Elem char_type;
```

Dabei erfolgt einfach eine direkte Zuordnung des Schablonenparameters `Elem` zum Datentyp `char_type`:

```
template <class Elem, class Traits>
class basic_ios : public ios_base
{ ...
}
```

Das folgende Beispiel demonstriert mehrere Anwendungen der `fill`-Methode, unter anderem in Verbindung mit links- und rechtsbündiger Platzierung der auszugebenden Zeichen:

```
int iwert = -1234;
cout.width(10);
cout << iwert << '\n';
cout.fill('*');
cout.width(10);
cout << "Hallo" << '\n';
cout.width(10);
cout << iwert << '\n';
cout.setf(ios::left, ios::adjustfield);
cout.width(10);
cout << iwert << '\n';
cout.setf(ios::internal, ios::adjustfield);
cout.width(10);
cout << iwert << '\n';
```

Ausgabe:

```
     -1234
*****Hallo
*****-1234
-1234*****
-*****1234
```

6.5.4. Manipulatoren

Unter dem Konzept der Manipulatoren versteht man spezielle Operanden in Ein-/Ausgabe-Ausdrücken, die nicht eingelesen bzw. ausgegeben werden, sondern in irgendeiner Art und Weise das betroffene Datenstrom-Objekt beeinflussen (*manipulieren*). Mit Hilfe von Manipulatoren kann man beispielsweise Formateinstellungen vornehmen oder den Ausgabepuffer leeren. Da sie wie normale Ein-/Ausgabe-Operanden verwendet werden, können sie auch in zusammengesetzten (verketteten) Ein-/Ausgabe-Ausdrücken auftreten. Dies erlaubt eine wesentlich effektivere Realisierung der Ein-/Ausgabe-Formatierung als die explizite Verwendung der in Tabelle 6.9 vorgestellten Formatierungsmethoden. Die Ausgabe zweier Zahlen, wobei die erste dezimal und die zweite hexadezimal dargestellt werden soll, lässt sich mit Hilfe von Manipulatoren so formulieren:

```
cout << dec << 17 << endl;
cout << hex << 17 << endl;
```

Denkbar ist sogar eine noch kompaktere Formulierung, die nur aus einer einzigen Anweisung besteht:

```
cout << dec << 17 << endl << hex << 17 << endl;
```

Wir erkennen an diesen zwei Beispielen, dass die Formateinstellungen innerhalb einer *shift*-Kette vorgenommen werden können, ohne diese zu unterbrechen.

Manipulatoren lassen sich in unterschiedlichen Kategorien einteilen. Man kann zum einen zwischen (vordefinierten) Standard-Manipulatoren und benutzerdefinierten Manipulatoren unterscheiden. Auf die Vorgehensweise zur Erstellung eines benutzerdefinierten Manipulatoren gehen wir am Ende dieses Kapitels ein. Manipulatoren können auch aus der Sicht ihrer Parameter betrachten werden: Es gibt parameterlose Manipulatoren und Manipulatoren mit Parametern. In Tabelle 6.13 finden Sie alle parameterlosen Standard-Manipulatoren vor:

Manipulator	Beschreibung	Ein-/Ausgabe
`endl`	Einen Zeilenvorschub (`'\n'`) wird ausgeben, der Ausgabepuffer wird anschließend geleert.	Ausgabe
`ends`	Ein terminierendes Nullzeichen (`'\0'`) wird ausgeben. Der Manipulator dient zum Abschließen einer C-Zeichenkette.	Ausgabe

`dec`
`oct`
`hex`
	Ein- oder Ausgabe ganzer Zahlen im dezimalen, oktalen bzw. hexadezimalen Format.	Ein-/Ausgabe

`ws`	Leerzeichen, Tabulatoren und Zeilenenden (*whitespaces*) werden in der Eingabe überlesen.	Eingabe
`flush`	Ausgabepuffer leeren.	Ausgabe

`left`
`right`
`internal`
	Die Ausgabe erfolgt entweder linksbündig, rechtsbündig oder im Falle von `internal` linksbündig bzgl. des Vorzeichens und rechtsbündig bzgl. des Werts.	Ausgabe

`fixed` `scientific`	Ausgabe von Gleitkommazahlen in Dezimalbruchdarstellung bzw. Exponentialdarstellung.	Ausgabe
`boolalpha`	Die Konstanten `true` und `false` alphabetisch ausgeben (oder lesen). Anders ausgedrückt: Die Konstante `ios::boolalpha` setzen.	Ein-/Ausgabe
`noboolalpha`	Die Konstanten `true` und `false` numerisch ausgeben (`1` bzw. `0`). Anders ausgedrückt: Die Konstante `ios::boolalpha` rücksetzen.	Ein-/Ausgabe
`showpos`	Explizite Ausgabe eines `'+'`-Zeichens bei positiven Zahlen (die Konstante `ios::showpos` setzen).	Ausgabe
`noshowpos`	Keine Ausgabe eines `'+'`-Zeichens bei positiven Zahlen (die Konstante `ios::showpos` rücksetzen).	Ausgabe
`uppercase`	Ausgabe von hexadezimalen Darstellungen und Exponential-Formaten in Großbuchstaben (die Konstante `ios::uppercase` setzen).	Ausgabe
`nouppercase`	Ausgabe von hexadezimalen Darstellungen und Exponential-Formaten in Kleinbuchstaben (die Konstante `ios::uppercase` rücksetzen).	Ausgabe

showbase	Ausgabe eines Zeichens, das die Basis des ganzzahligen Werts erkennen lässt (die Konstante ios::showbase setzen).	Ausgabe
noshowbase	Keine Basis bei ganzzahligen Werten anzeigen (die Konstante ios::showbase rücksetzen).	Ausgabe
showpoint	Ausgabe immer mit Dezimalpunkt und gegebenenfalls Nullen nach dem Punkt bei Gleitpunktzahlen (die Konstante ios::showpoint setzen).	Ausgabe
noshowpoint	Keine Ausgabe eines Dezimalpunkts bei Gleitpunktzahlen, deren Nachkommawert gleich Null ist (die Konstante ios::showpoint rücksetzen).	Ausgabe
skipws	(Führende) Leerzeichen, Tabulator und Zeilenumbrüche (*white spaces*) werden bei der Eingabe übersprungen (die Konstante ios::skipws setzen).	Eingabe
noskipws	(Führende) Leerzeichen, Tabulator und Zeilenumbrüche (*white spaces*) werden bei der Eingabe gelesen (die Konstante ios::skipws rücksetzen).	Eingabe

Tabelle 6.13. Parameterlose Standard-Manipulatoren.

Wie funktioniert eigentlich so ein Manipulator? Zur Beantwortung dieser Frage beschreiten wir einfach den Weg, einen parameterlosen Manipulator selbst zu definieren. Wir benennen den Manipulator hexadecimal, er soll wie der Standard-Manipulator hex arbeiten. Zunächst vereinbaren wir die Schnittstelle einer globalen Funktion hexadecimal wie folgt:

```
ios_base& hexadecimal (ios_base& stream);  // manipulator for hexadecimal output
```

Die Implementierung der hexadecimal-Funktion, die in einer separaten Datei liegen könnte, gestalten wir so:

```
ios_base& hexadecimal (ios_base& stream)
{
    stream.setf (ios_base::hex, ios_base::basefield);
    return stream;
}
```

Nun ist die Frage zu klären, wie es bei einer Anweisung der Gestalt

```
cout << hexadecimal << 17 << endl;
```

dazu kommt, dass der selbst definierte Manipulator aufgerufen wird? In dieser Situation hilft uns die Definition der Klasse ostream weiter, sie besitzt eine Überladung des <<-Operators mit folgendem Aussehen:

```
ostream& operator<< (ios_base& (*pfn) (ios_base&))
{
    (*pfn) (* (ios_base*) this);  // call manipulator function
    return (*this);
}
```

Diese Version des <<-Operators erwartet einen Zeiger auf eine Funktion, die einen Parameter vom Typ ios_base& besitzt und als Resultat einen Wert ebenfalls vom Typ ios_base& zurückliefert. Genau auf diese Weise ist unsere Funktion hexadecimal aber definiert, so dass die Anweisung

```
cout << hexadecimal << 17 << endl;
```

in unserem Sinne interpretiert wird, da der Name der Funktion hexadecimal vom Compiler als ein Zeiger auf diese Funktion interpretiert wird. Zur Hervorhebung dieses Ablaufs könnten wir das letzte Codefragment auch wie folgt umformulieren:

```
cout.operator << (&hexadecimal) << 17 << endl;
```

So erkennen wir deutlich, wie die Adresse der Funktion `hexadecimal` an die Operatorfunktion `<<` übergeben wird. Bei der Definition der Manipulatorfunktion kann man natürlich auch direkt die abgeleitete Klasse `ostream` verwenden, wenn man die Manipulatorfunktion nicht verallgemeinert definieren möchte:

```
ostream& hexadecimal (ostream& os)
{
    os.setf (ios_base::hex, ios_base::basefield);
    return os;
}
```

Neben den bisher betrachteten Manipulatoren, die alle ausnahmlos ohne Parameter waren, sind auch solche realisierbar, die Parameter besitzen. Beispielsweise wäre es doch sehr angenehm, über einen Manipulator zu verfügen, der eine bestimmte Anzahl von Leerzeichen in die Ausgabe einfügt:

```
cout << '1' << blanks(3) << '2' << endl;
```

Das letzte Codefragment sollte die Ausgabe

```
1   2
```

nach sich ziehen. Die Realisierung eines Manipulators mit Parametern ist etwas komplizierter, wir schlagen zu diesem Zweck einen Weg mit vier Etappen ein:

- Etappe 1: Definition einer Klasse `blanks`:

  Wir benötigen zunächst eine Klasse, die für den Standardoperator `()` eine Überladung besitzt. Der Hintergrund dieses Ansatzes liegt darin begründet, dass auf diese Weise eine Instanz dieser Klasse als Funktion angesehen werden kann. Diese Vorgehensweise dürfte im folgenden noch verständlicher werden, wir fahren zunächst mit der ersten – noch unvollständigen – Definition der Klasse `blanks` fort:

  ```
  class blanks
  {
  private:
      int m_num;

  public:
      blanks (int num) : m_num (num)
      {
      }
  };
  ```

  Die Klasse `blanks` besitzt einen Konstruktor mit einem Parameter. Damit können `blanks`-Objekte angelegt werden, die die Anzahl der Leerzeichen für eine bestimmte Ausgabe in ihrem Status herumtragen.

- Etappe 2: Übersetzung des Ausdrucks `cout << blanks(3)`:

  Zunächst einmal wird vom Compiler standardmäßig ein Ausdruck der Gestalt

  ```
  cout << blanks(3);
  ```

  in die äquivalente Darstellung

  ```
  cout.operator<< (blanks(3));
  ```

  transformiert. In dieser Darstellung erkennen wir, dass zunächst ein Objekt vom Typ `blanks` erzeugt wird, das mit dem benutzerdefinierten Konstruktor `blanks(int)` – hier mit dem aktuellen Parameter `3` – initialisiert wird. Dieses Objekt wird als Parameter an den Operator `<<` weitergereicht, den wir nun zu diesem Zweck geeignet überladen müssen:

  ```
  ostream& operator<< (ostream& os, const blanks& b)
  {
      return b(os);
  }
  ```

- Etappe 3: Implementierung des Operators `()` in der Klasse `blanks`:

  In der Überladung des `<<`-Operators aus der zweiten Etappe erkennen wir, dass wir in der Klasse `blanks` den `()`-Operator benötigen. Dazu gehen wir zu folgender, modifizierten

Definition der `blanks`-Klasse über:

```
class blanks
{
private:
    int m_num;

public:
    blanks (int num) : m_num (num)
    {
    }

    ostream& operator() (ostream& os) const
    {
        for (int i = 0; i < m_num; i ++)
            os.put (' ');
        return os;
    }
};
```

Mit dieser Ergänzung der Klasse `blanks` um den `()`-Operator ist nun der `<<`-Operator aus Etappe 2 übersetzungsfähig. Der Ausdruck `b(os)` wird vom Übersetzer zu `b.operator()(os)` umgewandelt.

Wir sind am Ziel angekommen: Die Platzierung des `blanks`-Klassennamens mit Parameterliste in einer Ausgabeoperation bewirkt zunächst, dass (mit dem einzigen vorhandenen Konstruktor) ein `blanks`-Objekt erzeugt wird. Dieses wiederum wird (per Referenz) an den überladenen `<<`-Operator weitergereicht, der an diesem Objekt den `()`-Operator aufruft. Als Argument wird der aktuelle Datenstrom (per Referenz) übergeben. Damit ist das `blanks`-Objekt im Besitz eines Ausgabedatenstroms und kann die gewünschte Anzahl von Leerzeichen (mit der `put`-Methode) in diesen Strom ausgeben.

- Etappe 4: Verkettung mit weiteren Operatoren:

  In der Verkettung des `<<`-Operators aus Etappe 2 und des `()`-Operators aus Etappe 3 wird jedes Mal die Referenz des übergebenen `ostream`-Objekts wieder zurückgegeben. Deshalb ist eine Verkettung dieses Manipulators auch mit weiteren Operatoren möglich.

Natürlich gibt es auch standardmäßig vorhandenen parametrisierte Manipulatoren, die drei bekanntesten finden Sie in Tabelle 6.14 näher erläutert vor:

Manipulator	Beschreibung
`setw`	Setzen der Feldbreite für die nächste Ausgabe (Parametertyp: `int`).
`setfill`	Setzen des Füllzeichens. In der Voreinstellung wird das Leerzeichen genommen (Parametertyp: `char`).
`setprecision`	Setzen der Weite für die Ausgabe von Gleitkommazahlen. Falls die Anzahl der notwendigen Ziffern den aktuellen Wert des Parameters übersteigt, wird auf die wissenschaftliche Notation, also die Ausgabe mit Exponent, umgeschaltet (Parametertyp: `int`).

Tabelle 6.14. Die wichtigsten Standard-Manipulatoren mit Parameter.

6.6. Dateien

6.7. Zeichenketten

6.8. Aufgaben

6.8.1. Manipulator für Währungsanzeigen

Erstellen Sie einen Manipulator `currency`, der dafür sorgt, dass die nachfolgend ausgegebene Gleitkommazahl mit 2 Nachkommastellen in der Ausgabe erscheint. Vor dem Wert ist die Währungsbezeichnung auszugeben.

Das folgende Codefragment demonstriert zwei Anwendungsfälle des Manipulators:

```
double money;
money = 10.99;
cout << currency("DM") << money << endl;
money = 20;
cout << currency("Euro") << money << endl;
```

Ausgabe:

```
DM: 10.99
Euro: 20.00
```

6.9. Lösungen

6.9.1. Manipulator für Währungsanzeigen

Zur Implementierung des Manipulators benötigen wie eine Klasse `currency` mit einer Überladung des Einfüge- und des `()`-Operators, wie in Listing 6.6 gezeigt:

```
01: class currency
02: {
03: private:
04:     char* m_name;
05:
06: public:
07:     // c'tor
08:     currency (char* name) : m_name (name)
09:     {
10:     }
11:
12:     // operator()
13:     ostream& operator() (ostream& os) const
14:     {
15:         os << m_name << ": ";
16:         os << fixed << setprecision (2);
17:         return os;
18:     }
19:
20:     // insertion operator
21:     friend ostream& operator<< (ostream& os, const currency& cur)
22:     {
23:         return cur(os);
24:     }
25: };
```

*Beispiel 6.6. Klasse **currency**.*

Kapitel 7. Templates

7.1. Einführung

Häufig macht man als Entwickler die Beobachtung, dass bestimmte Funktionen oder Operationen auf Werte unterschiedlichen Datentyps anwendbar sind. So sind beispielsweise die Addition und Multiplikation für komplexe Zahlen (oder auch quadratische Matrizen) genauso wie für reelle Zahlen oder ganze Zahlen definiert. Die Vorgehensweise, die `Add`-Funktion für einen Datentyp zu implementieren und danach für alle anderen Datentypen zu kopieren und anzupassen, ist nicht nur umständlich, sondern auch sehr fehleranfällig. Sind in einer Ausprägung der `Add`-Funktion Änderungen notwendig, so müssten diese in allen anderen Ausprägungen händisch nachgezogen werden. Die Beseitigung von Fehlern ist ein weiterer Aspekt, der sich negativ auf alle vorhandenen Funktionsausprägungen auswirkt. Wir erkennen, dass sich Algorithmen oder auch Datenstrukturen, die auf gemeinsamen Eigenschaften aufbauen, unabhängig vom speziellen Datentyp formulieren lassen sollten.

Mit dem C++-Sprachmittel der *Schablone* (engl. *template*) ist es möglich, die Parameter einer Funktion (und das Funktionsergebnis) ohne konkrete Festlegung ihres Datentyps zu spezifizieren. Für den unbestimmten Datentyp wird zunächst ein Platzhalter verwendet. Genau wie für eine einzelne Funktionen kann man auch für ganze Klassen eine Schablone definieren, aus der dann bei Bedarf eine tatsächliche Klassen erzeugt wird, dazu später in diesem Kapitel noch mehr. Erst bei der tatsächlichen Verwendung der Funktion (oder Klasse), nicht bereits aber bei deren Definition oder Deklaration, muss dann der tatsächlich benötigte Datentyp festgelegt werden.

Diese Betrachtungsweise ist vergleichbar mit der Vorgehensweise bei Funktionen, die in ihrer Definition *formale* Parameter verwenden. Mit Hilfe der Formalparameter lassen sich die Anweisungen in einem Funktionsrumpf auf einer abstrakten Basis formulieren. Erst beim Aufruf der Funktion mit *aktuellen* Parametern kommt es zur konkreten Manipulation von real existierenden Werten. In diesem Sinne stellt auch eine Schablone die Generalisierung einer Funktion (oder einer Klasse) dar: Dieses Mal sind die Aktualparameter keine *Werte*, sondern *Datentypen*.

7.2. Funktions-Templates

Die Deklaration einer parametrisierten Funktion `Max`, die das Maximum zweier Werte – zunächst noch unbekannten Datentyps – ermittelt, sieht so aus:

```
template <class T>
T Max (T x, T y);
```

Am Anfang der Deklaration erkennen wir als erstes das Schlüsselwort *template*. Hierdurch wird zum Ausdruck gebracht, dass die Definition einer Schablone folgt. C++ kennt grundsätzlich zwei Kategorien von Schablonen: *Funktions*- und *Klassen*-Schablonen, in diesem Abschnitt gehen wir nur auf die Funktionsschablonen ein. Nach dem *template*-Schlüsselwort folgt – eingeschlossen durch die zwei Klammern < und > – die Liste der Datentypparameter. Im aktuellen Beispiel ist es nur ein einziger Parameter, dessen Name frei wählbar ist. Es hat sich jedoch die Konvention eingebürgert, bei Schablonen mit einem einzigen Parameter den Bezeichner `T` zu wählen. Zur besseren Lesbarkeit ist jedem Datentypbezeichner das Schlüsselwort `typename` voranzustellen.

Die Schnittstelle und der Rumpf der parametrisierten Funktion sind nun wie eine normale Funktion zu programmieren. An jeder Stelle im Quellcode, die einen unbestimmten Datentyp erwartet, ist statt dessen der Bezeichner `T` (bzw. bei mehreren Parametern der entsprechende Parametername) zu verwenden, siehe Listing 7.1:

```
01: template <class T>
02: T Max (T x, T y)
03: {
04:     if (x > y)
05:         return x;
06:     else
07:         return y;
08: }
```

Beispiel 7.1. Funktionsschablone `Max`.

In der Realisierung der Funktionsschablone `Max` in Listing 7.1 erkennen wir, dass der unbestimmte Datentypname `T` in Zeile 2 wie ein beliebiger konkreter Datentyp verwendet wird.

Ein C++-Compiler erzeugt beim Übersetzen der in Listing 7.1 vorgestellten Funktionsschablone noch keinen Code. Dies ist auch gar nicht möglich, da eine Generierung konkreter Maschinencode-Instruktionen nicht auf der Basis eines unbestimmte Datentyps möglich ist. Diese erfolgt erst, wenn eine konkrete Ausprägung der Funktionsschablone, also die `Max`-Funktion für einen konkreten Datentyp wie `int`, `double`, usw. verwendet wird. Man sagt jetzt, dass der Compiler eine *Instanziierung* der Funktionsschablone vornimmt. Im folgenden Codefragment wird eine Instanziierung der Funktionsschablone für den Datentyp `int` vorgenommen, da wir `Max` konkret mit zwei Variablen des Typs `int`-aufrufen:

```
int x = 1;
int y = 1;
int z = Max(1, 2);
```

Der Aufruf der `Max`-Funktion führt zur Instanziierung der benötigten Schablonenfunktion. Der C++-Compiler erzeugt jetzt Maschinencode für eine `Max`-Funktion auf der Basis des bestimmten Datentyps `int`. Die einmal erzeugte Instanziierung der Funktionsschablone kommt von nun an immer zum Zug, wenn ein `Maximum`-Funktionsaufruf mit einem `int`-Array als Aktualparameter erfolgt.

Die `Max`-Funktionsschablone kann natürlich auch kompakter aufgerufen werden, wie das folgende

Codefragment demonstriert:

```
cout << "Max (5.0, 4.0): " << Max (5.0, 4.0) << endl;
```

Der C++-Compiler erkennt in diesem Fragment, dass die Schablone mit zwei Konstanten des Typs `double` aufgerufen wird und instanziiert eine entsprechende Schablone. Bei Zweifeln an der Beschaffenheit der konkret vorliegenden Instanziierung der Funktionsschablone kann man auch das *Watch*-Fenster des Visual Studios zu Rate ziehen:

```
Templates.exe!Max<double>(double x=5.0000000000000000, double y=4.0000000000000000)
```

Es lassen sich nicht nur einfache Variablen des Typs `T` deklarieren, man kann auch einen Zeiger der eine Referenz vom Typ `T` oder gar ein Array mit Werten des Typs `T` usw. vereinbaren. In Listing 7.2 finden Sie eine mögliche Realisierung einer Funktionsschablone `Maximum` zur Bestimmung des maximalen Werts in einem Array mit Werten unbestimmten Datentyps vor.

```
01: template <typename T>
02: T Maximum (T values[], int num)
03: {
04:     T max = values[0];
05:
06:     for (int i = 1; i < num; i ++)
07:         if (values[i] > max)
08:             max = values[i];
09:
10:     return max;
11: }
```

Beispiel 7.2. Funktionsschablone `Maximum`.

Am Beispiel aus Listing 7.2 erkennen wir, dass eine Funktionsschablone zwei Arten von Parametern haben kann:

- *Typ*-Parameter:

  Parameter dieser Kategorie werden durch das Schlüsselwort `typename` (veraltet auch `class`) gekennzeichnet und stehen für einen unbestimmten Datentyp.

- *Nicht-Typ*-Parameter:

  Solche Parameter sind identisch mit den regulären, formalen Parametern einer C-Funktion und legen den bestimmten Datentyp für die Aktualparameter fest.

Kehren wir zur `Maximum`-Schablone aus Listing 7.2 zurück: Mit dem Parameter `Maximum` wird der (unbestimmte) Typ der Elemente eines Arrays festgelegt, der Parameter `num` ist vom Typ `int` und spezifiziert die Größe des Arrays.

Zum Abschluss dieses Abschnitts betrachten wir in Listing 7.3 eine Funktionsschablone, deren Parameter Referenzen eines unbestimmten Datentyps sind:

```
01: template <typename T>
02: void Swap (T& x, T& y)
03: {
04:     T tmp = x;
05:     x = y;
06:     y = tmp;
07: }
```

Beispiel 7.3. Funktionsschablone `Swap`.

Beim Aufruf einer konkreten Instanziierung der `Swap`-Schablone ist natürlich darauf zu achten, dass die aktuellen Parameter auch konform zur Schablonendefinition sind. Die Anweisung

```
Swap(1, 2);
```

ist weder sinnvoll (wie soll der *Wert* zweier Konstanten vertauscht werden?), der C++-Compiler reagiert auch etwas akademischer mit der Fehlermeldung *„'Swap' : cannot convert parameter 1 from 'int' to 'int &'"*. Das folgende Codefragment

```
int x = 1;
int y = 2;
cout << "x: " << x << ", y: " << y << endl;
Swap (x, y);
cout << "x: " << x << ", y: " << y << endl;
```

hingegen ist übersetzungsfähig und produziert das gewünschte Resultat

```
x: 1, y: 2
x: 2, y: 1
```

7.2.1. Implizite Instanziierung

Steht der C++-Compiler vor der Aufgabe, eine Schablone zu instanziieren, analysiert er zunächst die verursachende Aufrufstelle. An Hand der Funktionsargumente versucht der Übersetzer nun, passende aktuelle Templateparameter zu identifizieren. Betrachten wir zu diesem Zweck die folgende Anwendung der `Maximum`-Schablone aus Listing 7.2:

```
double dvalues[] = { 2.0, 4.0, 6.0, 8.0 };
double dmax = Maximum (dvalues, 4);
```

Der aktuelle Parameter `dvalues` ist vom Typ `double[]`, damit zieht der Aufruf eine Instanziierung der `Maximum`-Funktionsschablone in der Ausprägung

```
double Maximum (double values[], int num)
{
    double max = values[0];

    for (int i = 1; i < num; i ++)
        if (values[i] > max)
            max = values[i];

    return max;
}
```

nach sich. Wir bezeichnen diese Art der Instanziierung von Funktionsschablonen als *implizit*, da sie automatisch dann erfolgt, wenn wir im Quellcode eine entsprechende Verwendung vorfinden und der Compiler in der Lage ist, selbständig die richtige Spezialisierung der Schablone zu finden.

7.2.2. Explizite Instanziierung

Es gibt auch Situationen, in denen der C++-Compiler nicht in der Lage ist, selbständig die richtige Spezialisierung einer Schablone zu finden. Betrachten Sie beispielsweise den folgenden Anwendungsfall der `Max`-Schablone aus Listing 7.1:

```
int i = 1;
double d = 2.5;
cout << Max (i, f) << endl;
```

Der Compiler quittiert die letzte Zeile mit der Fehlermeldung *„'T Max(T,T)': template parameter 'T' is ambiguous"*, da für den unbestimmten Datentyp T sowohl `int` wie auch `double` in Frage kommen können. Durch eine erweiterte Syntax lassen sich die Parameter eine Schablone explizit angegeben:

```
int i = 1;
double d = 2.5;
cout << Max<double> (i, d) << endl;
```

In diesem Fall wird eine konkrete Funktion auf Basis des Datentyps `double` instanziiert, die Übersetzung und Ausführung erfolgen problemlos. Es ist auch eine Instanziierung der Schablone mit dem Datentyp `int` denkbar:

```
int i = 1;
double d = 2.5;
cout << Max<int> (i, d) << endl;
```

Diese Form der Instanziierung ist prinzipiell machbar, natürlich gelten die üblichen Regeln der Programmiersprache C++ aber nach wie vor. In diesem Fall bedeutet das konkret, dass der Funktionsaufruf vom Übersetzer mit der Warnung *„conversion from 'double' to 'int', possible loss of data"* begleitet wird. Das Resultat der Funktionsaufruf ist der ganzzahlige Wert 2, da der aktuelle Parameter `2.5` vor dem Aufruf auf den ganzzahligen Wert 2 abgerundet wird.

7.2.3. Explizite Spezialisierung

Die bislang betrachteten Instanziierungen einer Funktionsschablone basierten immer auf derselben Schablonendefinition. In Analogie zum Überladen einer Funktion (Methode) kommt auch bei Schablonen der Wunsch auf, für bestimmte Parameterkonstellationen einer Schablone eine alternative Implementierung bereitstellen zu können. Wir motivieren diese Aussage durch das folgende Codefragment:

```
cout << Max("Eins", "Zwei") << endl;
```

Die bisherige Implementierung der `Max`-Schablone aus Listing 7.1 ist für `const char*`-Parameter nicht sinnvoll, da hier nur die Zeigeradressen verglichen werden:

```
Max<char const *> (const char* x = 0x004b41e4, const char* y = 0x004b41ec)
{
    if (0x004b41e4 > 0x004b41ec)
        return 0x004b41e4;
    else
        return 0x004b41ec;
}
```

Wir definieren aus diesem Grund eine explizite Spezialisierung der `Max`-Schablone für den Datentyp `const char*`:

```
01: #include <cstring>
02:
03: template <>
04: const char* Max<const char*> (const char* x, const char* y)
05: {
06:     return (strcmp(x, y) < 0) ? x : y;
07: }
```

Beispiel 7.4. Explizite Spezialisierung der Funktionsschablone `Max`.

Mit dieser Implementierung der `Max`-Schablone werden nun Zeichenketten bezüglich ihrer lexikografischen Anordnung verglichen (basierend auf dem zu Grunde liegenden Zeichensatz) und nicht ihre Adressen im Speicher. Alternativ zu Listing 7.4 gibt es auch eine Syntax mit abgekürzter Schreibweise:

```
01: #include <cstring>
02:
03: template <>
04: const char* Max (const char* x, const char* y)
05: {
06:     return (strcmp(x, y) < 0) ? x : y;
07: }
```

Beispiel 7.5. Explizite Spezialisierung der Funktionsschablone `Max` in abgekürzter Schreibweise.

Der Vollständigkeit halber erwähnen wir noch eine dritte Schreibweise. Diese gilt aber als veraltet und sollte dementsprechend nicht mehr angewendet werden:

```
01: #include <cstring>
02:
03: const char* Max (const char* x, const char* y)
04: {
05:     return (strcmp(x, y) < 0) ? x : y;
06: }
```

Beispiel 7.6. Explizite Spezialisierung der Funktionsschablone `Max` in veralteter Schreibweise.

Die dritte Schreibweise besitzt keinerlei Merkmale einer Schablone. Es handelt sich hierbei um das gewöhnliche Überladen einer Funktion, das eben auch dann zulässig ist, wenn bereits Funktionsschablonen mit demselben Namen existieren.

Achtung

Das Thema der Spezialisierung von Funktionsschablonen ist – wie wir gerade gesehen haben – sehr wesensverwandt mit dem Überladen von Funktionen. Da es zu einer Funktionsschablone prinzipiell unendlich viele konkrete Spezialisierungen geben kann, je nachdem, mit welchem konkreten Datentyp die Schablonenparameter ersetzt werden, produziert eine Funktionsschablone in dieser Betrachtungsweise beliebig viele Überladungen einer Funktion. Die auf diese Weise entstehenden Instanziierungen einer Funktionsschablone konkurrieren nun auf der einen Seite mit den expliziten Spezialisierungen der Schablone und dazu noch andererseits mit gewöhnlichen, nicht parametrisierten Funktionen, die alle denselben Namen besitzen. Ist der Compiler mit dem Aufruf einer Funktion konfrontiert, steht er vor dem Problem, dass er gegebenenfalls aus bis zu drei Kategorien eine konkrete Funktion auswählen kann. Um potentielle Mehrdeutigkeiten bei der Übersetzung aufzulösen, geht der C++-Compiler bei der Wahl einer passenden Funktion in folgender Reihenfolge vor:

1. Aufruf einer gewöhnlichen Funktion.
2. Aufruf einer expliziten Spezialisierung einer Funktionsschablone.
3. Aufruf einer durch Instanziierung einer Funktionsschablone erzeugten Funktion.

7.2.4. Ein Beispiel

Aktionen wie Suchen und Sortieren sind generell unabhängig von der genauen Art dessen, was gesucht oder sortiert wird. Elemente sehr vieler Typen kann man mit den gleichen Mechanismen suchen und sortieren. Sie müssen nur miteinander verglichen werden können. Wenn die Elemente eines Typs `T` verglichen werden können, dann können sie auch sortiert und gesucht werden. Das sind ideale Voraussetzungen zum Einsatz von Schablonen, wie wir am folgenden Beispiel zum generischen Sortieren eines Arrays demonstrieren werden.

Konkret wollen wir auf möglichst universelle Art entweder `int`-Werte, C-Zeichenketten, also `'\0'`-terminierte `char`-Arrays oder Instanzen einer Klasse `Date`-Werte sortieren. Dabei legen wir jeweils zu Grunde, dass die einzelnen Variablen oder Objekte in einem Array abgelegt sind und dass die Länge dieses Arrays bekannt ist. Zum Sortieren greifen wir auf den *BubbleSort*-Algorithmus zurück, dessen Technik aber nicht Thema dieses Abschnitts ist. Wir können an diesem Beispiel sehr schön die Aspekte von Funktionsschablonen, ihrer Instanziierung und ihrer Spezialisierung studieren.

Um die Aufgabenstellung möglichst modular anzugehen, entwerfen wir in Listing 7.7 zunächst die Schablonen `Swap`, `Compare`, `Print` und `Sort`, die wir flexibel kombiniert zum Einsatz bringen wollen:

```
01: template <typename T>
02: void Swap (T& x, T& y)
03: {
04:     T tmp = x;
05:     x = y;
06:     y = tmp;
07: }
08:
09: template <typename T>
10: int Compare (const T& x, const T& y)
11: {
12:     return (x < y) ? -1 : ((x == y) ? 0 : 1);
13: }
14:
15: template <typename T>
16: void Sort (T x[], int num)
17: {
18:     for (int i = 1; i < num; i ++)
19:         for (int j = 0; j < num - i; j ++)
20:             if (Compare (x[j+1], x[j]) < 0)
21:                 Swap (x[j], x[j+1]);
22: }
23:
24: template <typename T>
25: void Print (T x[], int num)
26: {
27:     cout << "{ ";
28:     for (int i = 0; i < num; i ++)
29:     {
30:         cout << x[i];
31:         if (i < num - 1)
32:             cout << ", ";
33:     }
34:     cout << " }" << endl;
35: }
```

Beispiel 7.7. Unterstützende Funktionsschablonen zum Sortieren eines Arrays.

Wir wenden diese Schablonen nun an, um in einem Array mit `int`-Werten die aufsteigende Reihenfolge herzustellen. Wir müssen nichts weiter tun, die Anweisungsfolge

```
int numbers[] = { 9, 7, 5, 3, 1 };
Print (numbers, 5);
Sort (numbers, 5);
Print (numbers, 5);
```

führt uns bereits zum Ziel, wie eine Programmausführung verifiziert::.

```
{ 9, 7, 5, 3, 1 }
{ 1, 3, 5, 7, 9 }
```

Wenn wir mit einem Array von C-Strings analog vorgehen, bekommen wir beim Übersetzen des Codefragments

```
char* strings[] = { "Franz", "Beata", "Peter", "Werner", "Susan" };
Print (strings, 5);
Sort (strings, 5);
Print (strings, 5);
```

keinerlei Schwierigkeiten. Bei der Ausführung der Anweisungen müssen wir aber feststellen, dass die Zeichenketten nicht in der erwarteten, sortierten Reihenfolge vorliegen. Erkennen Sie das Problem? In der Funktionsschablone `Compare` werden nicht, wie erwartet, die Zeichenketten in Bezug auf ihre lexikografische Ordnung verglichen, sondern die *Adressen* der Zeichenketten. Aus diesem Grund müssen wir für den Datentyp `char*` zu einer Spezialisierung der `LessThan`-Funktionsschablone übergehen:

```
01: template <>
02: int Compare<char*> (char* const & x, char* const & y)
03: {
04:     return strcmp (x, y);
05: }
```

Jetzt werden Zeichenketten nach unserer Vorstellung verglichen und auch das Resultat der Programmausführung passt:

```
{ Franz, Beata, Peter, Werner, Susan }
{ Beata, Franz, Peter, Susan, Werner }
```

Kommen wir jetzt auf Objekte des Typs `Date` zu sprechen. Für die Belange dieses Kapitels gehen wir zunächst mit folgender Minimalimplemetierung der Klasse `Date` ins Rennen:

```
01: // prerequisites
02: class Date
03: {
04: private:
05:     int m_day, m_month, m_year;
06:
07: public:
08:     // c'tors
09:     Date ();
10:     Date (const Date&);
11:     Date (int day, int month, int year);
12:
13:     // comparison operators
14:     friend bool operator== (const Date&, const Date&);
15:     friend bool operator<  (const Date&, const Date&);
16:     friend bool operator>  (const Date&, const Date&);
17:
18:     // assignment operator
19:     Date& operator= (const Date&);
20:
21:     // output
22:     friend ostream& operator<< (ostream&, const Date&);
23: };
24:
25: // c'tors
26: Date::Date ()
27: {
28:     cout << "default c'tor" << endl;
29:     m_day = 1;
30:     m_month = 1;
31:     m_year = 2000;
32: }
33:
34: Date::Date (const Date& d)
35: {
36:     cout << "copy c'tor" << endl;
37:     m_day = d.m_day;
38:     m_month = d.m_month;
39:     m_year = d.m_year;
40: }
41:
```

```
42: Date::Date (int day, int month, int year)
43: {
44:     cout << "c'tor (int, int, int)" << endl;
45:     m_day = day;
46:     m_month = month;
47:     m_year = year;
48: }
49:
50: // assignment operator
51: Date& Date::operator= (const Date& d)
52: {
53:     cout << "operator=" << endl;
54:     m_day = d.m_day;
55:     m_month = d.m_month;
56:     m_year = d.m_year;
57:
58:     return *this;
59: }
60:
61: // comparison operators
62: bool operator== (const Date& d1, const Date& d2)
63: {
64:     return (d1.m_year == d2.m_year &&
65:             d1.m_month == d2.m_month &&
66:             d1.m_day == d2.m_day);
67: }
68:
69: bool operator< (const Date& d1, const Date& d2)
70: {
71:     if (d1.m_year < d2.m_year)
72:         return true;
73:
74:     if (d1.m_year == d2.m_year && d1.m_month < d2.m_month)
75:         return true;
76:
77:     if (d1.m_year == d2.m_year &&
78:         d1.m_month == d2.m_month &&
79:         d1.m_day < d2.m_day)
80:             return true;
81:
82:     return false;
83: }
84:
85: bool operator> (const Date& d1, const Date& d2)
86: {
87:     return ! (d1 < d2 || d1 == d2);
88: }
89:
90: // output operator
91: ostream& operator<< (ostream& os, const Date& d)
92: {
93:     os << d.m_day << '.' << d.m_month << '.' << d.m_year;
94:     return os;
95: }
```

Die Funktionsschablonen `Print` und `Sort` sind auch für ein Array des Typs `Date` problemlos übersetzbar. Zu Testzwecken geben wir die Aufrufe aller Konstruktorenaufrufe sowie alle Wertzuweisungen von `Date`-Objekten aus, so dass man sich einen Überblick über die Aktivitäten in den einzelnen Schablonen verschaffen kann:

```
Date d1 (1, 1, 2000);
Date d2 (12, 12, 2004);
Date d3 (7, 1, 2002);
Date dates[3] = { d1, d2, d3 };
Print (dates, 3);
Sort (dates, 3);
Print (dates, 3);
```

Die Ausgabe des Codefragments lautet:

```
c'tor (int, int, int)
c'tor (int, int, int)
c'tor (int, int, int)
copy c'tor
copy c'tor
copy c'tor
```

```
{ 1.1.2000, 12.12.2004, 7.1.2002 }
copy c'tor
operator=
operator=
{ 1.1.2000, 7.1.2002, 12.12.2004 }
```

7.3. Klassen-Templates

Genauso wie wir für Funktionen Schablonen definiert haben, gibt es auch Schablonen für Klassen. Eine Klassen-Template ist eine Schablone für eine Klasse mit einem oder mehreren unbestimmten Datentypparametern, die als Typ für einzelne Bestandteile der Klasse verwendet werden. In Analogie zu den Funktionen nennt man solche Schablonen auch *parametrisierte Klassen*. Datenstrukturen wie Vektoren oder Listen eignen sich sehr gut dazu, als parametrisierte Klasse entwickelt zu werden. Da derartige Objekte Elemente unterschiedlichen Typs enthalten können, genügt es, nur eine (parametrisierte) Klasse zu entwickeln. Sie kann anschließend für beliebige konkrete Datentypen eingesetzt werden.

7.3.1. Allgemeines

Wir stellen das Prinzip der Klassen-Schablone am einfachen Beispiel eines Stapels in Listing 7.8 vor. In einer ersten Version setzen wir zur Vereinfachung voraus, dass der Stapel maximal 20 Elemente aufnehmen kann. Ferner verwendet die implementierende Klassen-Schablone in ihrer Realisierung keine dynamisch allokierten Variablen, so dass der *Big-Three*-Aspekt (Destruktor, Kopier-Konstruktor und Wertzuweisungsoperator) zunächst außer Acht gelassen werden können.

```
01: // class interface
02: template <typename T>
03: class Stack
04: {
05: public:
06:     // c'tors
07:     Stack ();
08:
09:     // public interface
10:     void Push(const T e);
11:     T Pop();
12:     bool IsEmpty() const;
13:     bool IsFull() const;
14:     int Size() const;
15:
16: private:
17:     // data
18:     static const int MaxSize = 20;
19:     T m_stack[MaxSize];
20:     int m_top;
21: };
22:
23: // class implementation
24: template <typename T>
25: Stack<T>::Stack()
26: {
27:     m_top = 0;
28: }
29:
30: template <typename T>
31: bool Stack<T>::IsEmpty() const
32: {
33:     return m_top == 0;
34: }
35:
36: template <typename T>
37: bool Stack<T>::IsFull() const
38: {
39:     return m_top == m_size;
40: }
41:
42: template <typename T>
43: int Stack<T>::Size() const
44: {
45:     return m_top;
46: }
47:
48: template <typename T>
49: void Stack<T>::Push(const T e)
50: {
51:     m_stack[m_top] = e;
```

```
52:     m_top ++;
53: }
54:
55: template <typename T>
56: T Stack<T>::Pop()
57: {
58:     m_top --;
59:     return m_stack[m_top];
60: }
```

Beispiel 7.8. Der generische Klassentyp `Stack`.

Dem Namen einer Klassenschablone ist das Schlüsselwort `template` voranzustellen, gefolgt in den spitzen Klammern von den formalen Parametern für die Schablonendefinition. Hängt die Schablone nur von einem einzigen Parameter ab, besitzt dieser in der Regel den Namen `T`. Dieser Parameter kann im Rumpf der Schablone wie ein gewöhnlicher, konkreter Typ verwendet werden, also zum Beispiel in der Deklaration

```
T m_stack[MaxSize];
```

um ein Feld bestehend aus `T`-Elementen zu deklarieren. Eine Schablone kann auch von mehreren Parametern abhängen, wir werden dazu später ein Beispiel demonstrieren (TODO). Nun kommen wir auf die syntaktischen Besonderheiten zu sprechen, die bei der Definition einer Klassenschablone zu beachten sind. Jede Definition einer Methode, die außerhalb der Klasse erfolgt, muss mit dem Vorspann

```
template <typename T>
```

eingeleitet werden, wie zum Beispiel

```
template <typename T>
void Stack<T>::Push (T e)
{
    m_elems[m_top] = e;
    m_top ++;
}
```

Ferner ist jedes Vorkommen des Schablonennamens außerhalb seiner Definition mit den Schablonenparametern in spitzen Klammern zu versehen, wie zum Beispiel

```
template <typename T>
void Stack<T>::Push (T e)
{
    m_elems[m_top] = e;
    m_top ++;
}
```

In den Parameterlisten einer Schablonenmethode sowie im Rumpf von Methodendefinitionen müssen die Schablonenparameter nicht explizit aufgeführt werden, wenn der Schablonenname verwendet wird. Diese syntaktischen Regeln resultieren aus den Grenzen der Gültigkeitsbereiche: Die Parameterliste und der Methodenrumpf sind aus der Sicht der Blockstruktur dem Methodennamen untergeordnet und „erben“ dessen Namensraum. Der Ergebnistyp einer Methode steht außerhalb des Methodenrumpfes, so daß hier jeder Templatename ausdrücklich parametrisiert werden muß.

Die Klassen-Schablone ist nun die Grundlage für die Erzeugung einer realen Stack-Klasse. Dazu muss in einem Programm der Schablonenparameter durch einen wirklichen Typ (elementarer Standarddatentyp oder benutzerdefinierter Klassentyp) ersetzt werden. Dies geschieht, indem wir den benötigten Typ in spitzen Klammern <> hinter dem Schablonennamen aufführen:

```
01: #include <iostream>
02: using namespace std;
03:
04: #include "Stack.h"
05:
06: void main()
07: {
08:     Stack<int> istack;
09:     for (int i = 0; i < 5; i ++)
10:         istack.Push (i);
11:     cout << "# elements: " << istack.Size() << endl;
12:     while (! istack.IsEmpty ())
```

```
13:         cout << istack.Pop() << " ";
14:     cout << endl;
15: }
```

Beispiel 7.9. Ein Beispiel einer konkreten Stack-Klasse, bestehend aus `int`-Werten.

Eine wesentliche Einschränkung der in Listing 7.8 angeführten `Stack<T>`-Klasse ist die feste Größe des Feldes, das zur Aufnahme der Datenelemente dient. Wir wollen jetzt dem Benutzer beim Anlegen des Stacks ermöglichen, dessen Maximalkapazität festzulegen. Wir müssen dazu den Konstruktor der Klasse abändern, einen Destruktor ergänzen und mit Hilfe entsprechender `new[]`- und `delete[]`-Aufrufe Speicher für einen flexibel großen Stapel anfordern oder wieder freigeben. Die Festlegung der Schablonenschnittstelle und die Implementierung ist dieses Mal auf zwei Dateien aufgeteilt (Listing 7.10 und Listing 7.11):

```
01: template<typename T>
02: class Stack
03: {
04: public:
05:     // c'tors / d'tor
06:     Stack (int size = 10);
07:     Stack (const Stack&);
08:     ~Stack ();
09:
10:     // public interface
11:     void Push(const T e);
12:     T Pop();
13:     bool IsEmpty() const;
14:     bool IsFull() const;
15:     int Size() const;
16:
17:     // operators
18:     Stack& operator=(const Stack&);
19:
20: private:
21:     int m_top;
22:     int m_size;
23:     T* m_stack;
24: };
```

Beispiel 7.10. Schnittstelle einer Stack-Schablone mit dynamischer Speicherverwaltung.

Es folgt die Implementierung der Schablonenschnittstelle:

```
01: template <typename T>
02: Stack<T>::Stack(int size = 10) : m_size(size), m_top(0)
03: {
04:     m_stack = new T[m_size];
05: }
06:
07: template <typename T>
08: Stack<T>::Stack(const Stack<T>& s) : m_size(s.m_size), m_top(s.m_top)
09: {
10:     // allocate buffer
11:     m_stack = new T[m_size];
12:
13:     // copy elements
14:     for (int i = 0; i < m_top; i ++)
15:         m_stack[i] = s.m_stack[i];
16: }
17:
18: template <typename T>
19: Stack<T>::~Stack()
20: {
21:     delete[] m_stack;
22: }
23:
24: // operators
25: template <typename T>
26: Stack<T>& Stack<T>::operator=(const Stack<T>& s)
27: {
```

```
28:     // prevent self-assignment
29:     if (this == &s)
30:         return *this;
31:
32:     // delete old stack
33:     delete[] m_stack;
34:
35:     // allocate new stack
36:     m_size = s.m_size;
37:     m_top = s.m_top;
38:     m_stack = new T[m_size];
39:
40:     // copy stack
41:     for (int i = 0; i < m_top; i ++)
42:         m_stack[i] = s.m_stack[i];
43:
44:     return *this;
45: }
46:
47: template <typename T>
48: bool Stack<T>::IsEmpty() const
49: {
50:     return m_top == 0;
51: }
52:
53: template <typename T>
54: bool Stack<T>::IsFull() const
55: {
56:     return m_top == m_size;
57: }
58:
59: template <typename T>
60: int Stack<T>::Size() const
61: {
62:     return m_top;
63: }
64:
65: template <typename T>
66: void Stack<T>::Push(T e)
67: {
68:     m_stack[m_top] = e;
69:     m_top ++;
70: }
71:
72: template <typename T>
73: T Stack<T>::Pop()
74: {
75:     m_top --;
76:     return m_stack[m_top];
77: }
```

Beispiel 7.11. Implementierung einer Stack-Schablone mit dynamischer Speicherverwaltung.

Da wir in der Klassendefinition nun dynamisch allokierte Speicherbereiche vorfinden, müssen jetzt zusätzlich noch der Zuweisungsoperator, der Kopierkonstruktor und ein Destruktor ergänzt werden. Das Paradigma der *Big-Three* ist natürlich auch bei Klassenschablonen zu beachten.

Zum Testen der modifizierten Klassenschablone legen wir folgenden Testrahmen zu Grunde:

```
void main()
{
    Stack<char> cstack(6);
    for (char c = 'A'; c <= 'F'; c ++)
        cstack.Push(c);
    cout << "# elements: " << cstack.Size() << endl;

    Stack<char> cstack2 = cstack;

    while (!cstack.IsEmpty())
        cout << cstack.Pop() << " ";
    cout << endl;

    while (!cstack2.IsEmpty())
        cout << cstack2.Pop() << " ";
    cout << endl;
}
```

Die Ausgabe lautet

```
# elements: 6
F E D C B A
F E D C B A
```

7.3.2. Schablonen und Freunde

Wie wir im Kapitel über Freundschaftsbeziehungen bereits gesehen haben, ist es manchmal erforderlich, Zugriffe auf interne Variablen einer Klasse durch eine `friend`-Deklaration für eine externe Funktion zuzulassen. Dabei gilt es zunächst einmal festzuhalten: Eine `friend`-Funktion ist prinzipiell keine Template-Funktion. Eine `friend`-Deklaration mit der bislang vorgestellten Syntax befördert das betroffene Element zum Freund aller Instanziierungen der umgebenden Klassenschablone. Dies ist nicht immer erwünscht: In der Regel sollte die Funktion mit der korrespondierenden Template-Klasse auf der Basis gemeinsamer Datentypen zusammenarbeiten, es ist folglich sinnvoll, auch die `friend`-Funktion als Schablone zu definieren.

Wir diskutieren die getroffenen Aussagen am Beispiel des Ausgabeoperators `<<` für die Klassenschablone `Stack<T>`. Der `<<`-Operator wird im folgenden Beispiel als Funktionsschablone im Kontext einer Klassenschablone definiert:

```
template <typename T>
class Stack
{
    // output operator
    template<typename T>
    friend ostream& operator<< (ostream&, const Stack<T>&);

    ...
};
```

Die Definition der Funktionsschablone ist noch nachzutragen. Auf Basis der Klasse `Stack<T>` aus Listing 7.8 könnte diese so aussehen:

```
template <typename T>
ostream& operator<< (ostream& os, const Stack<T>& s)
{
    for (int i = 0; i < s.m_top; i ++)
        os << i << ":   " << s.m_stack[i] << endl;
    return os;
}
```

Wir erhalten auf diese Weise eine Version der Funktionsschablone für jede Instanz der Klassenschablone. Diese Vorgehensweise ist vor allem dann empfehlenswert, wenn eine Funktionsschablone von denselben Parametern wie die umgebende Klassenschablone abhängt. Die Ausgabe eines Stapels mit `int`-Werten kann nun vergleichsweise kompakt so erfolgen:

```
Stack<int> istack;
istack.Push(11);
istack.Push(12);
cout << istack << endl;
```

Die Ausgabe des Codefragments lautet

```
0: 11
1: 12
```

7.4. Schablonen und Typnamen: Das Schlüsselwort `typename`

Im folgenden Codefragment finden Sie die Definition einer Klasse `AnyType` vor. An sich ist an dieser Deklaration nichts auffälliges zu entdecken, die Klassendefinition sieht syntaktisch korrekt aus. Sie enthält als Elemente eine Instanzvariable `m_v` des generischen Tpys `vector<T>` und die Schnittstellenfestlegung einer Methode `AnyMethod`:

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

template<typename T>
class AnyType
{
private:
    vector<T> m_v;

public:
    // method
    typename vector<T>::iterator AnyMethod ();
};
```

Selbst wenn wir die Implementierung der Methode `AnyMethod` ergänzen, schaffen wir es nicht, das Programm fehlerfrei zu übersetzen:

```
template<typename T>
vector<T>::iterator AnyType<T>::AnyMethod ()
{
    return m_v.begin();
}
```

Der Microsoft C++-Compiler reagiert beim Übersetzen zweimal mit der Fehlermeldung *„'std::vector<T>::iterator': dependent name is not a type"* – sowohl bei der Schnittstellenfestlegung wie auch bei der Implementierung der Methode `AnyMethod`. Der Punkt ist, dass nach dem C++-Sprachstandard ein Name, der in einer Schablonendefinition benutzt wird und der von einem Schablonenparameter abhängt (wie in unserem Beispiel der Name `vector<T>::iterator`), nicht als Typname interpretiert wird! Einen Grund für dieses Compilerverhalten kann man am folgenden Beispiel erkennen:

```
template<class T>
void AnyMethod (T& v)
{
    T::x(y);  // (1)
    T::x y;   // (2)
}
```

Da bei der Übersetzung die Instantiierung von `T` noch nicht bekannt ist, kann `T::x` sowohl ein Typname als auch ein Funktionsname sein. Um diese mehrdeutige Situation zu vermeiden, nimmt der Compiler an, dass `T::x` *kein* Typname ist. Der Compiler nimmt also `T::x` als Funktionsaufruf und übersetzt (1) fehlerfrei und liefert bei (2) eine Fehlermeldung. Soll ein Name, der von einem Schablonenparameter abhängt, vom Compiler als Typname verstanden werden, so ist dazu das Schlüssetwort `typename` als Präfix des Typname zu verwenden. Das obere Beispiel muss damit wie folgt lauten:

```
template<typename T>
class AnyType
{
private:
    vector<T> m_v;

public:
    // method
```

```
    typename vector<T>::iterator AnyMethod ();
};

template<typename T>
typename vector<T>::iterator AnyType<T>::AnyMethod ()
{
    return m_v.begin();
}
```

7.5. Übungen

7.5.1. Ein Stapel dynamischer Länge

In den Beispielen dieses Kapitels wurde aus naheliegenden Gründen die Klasse `Stack` mit einer möglichst einfachen Implementierung betrachtet. Ziel dieser Übung ist es, eine Realisierung eines Stapels zu entwerfen, die auch in größeren Programmen einsetzbar ist.

Legen Sie Zu diesem Zweck die Elemente des Stapels in einer einfach verketteten Liste ab. Auf diese Weise können beliebig viele Elemente im Stapel ohne nennenswerte Kopiervorgänge abgelegt werden. Der Zugriff auf das oberste Element des Stapels kann ebenfalls zeitoptimal gestaltet werden, da dieses bei einer verketteten Liste im Wurzelelement abgelegt werden kann. Die beiden Methoden `push` und `pop` sind ohne Performanceeinbußen implementierbar.

Das nachfolgende Codefragment sollte mit Hilfe der Klasse `Stack<T>` übersetzbar sein:

```
void main()
{
    Stack<int> is;
    for (int i = 0; i < 5; i ++)
        is.push (2 * i);
    cout << "Size: " << is.size() << endl;
    while (! is.empty())
        cout << "  " << is.pop() << endl;
}
```

7.5.2. Restklassenarithmetik mit Klassenschablonen

Wenn man mit Restklassen rechnet, so bedeutet dies, dass man nur mit dem „Rest“ rechnet, der nach einer ganzzahligen Teilung durch eine bestimmte Zahl übrigbleibt. Diese Zahl, durch die geteilt wird, nennt man den „Modul“ oder die „Modulzahl“. Wenn wir beispielsweise mit dem Teiler oder der Modulzahl 5 rechnen, sagen wir auch, wir rechnen „modulo 5“. Beim Rechnen mit Restklassen addieren und teilen wir Zahlen also nach den normalen Regeln der Alltagsarithmetik, verwenden dabei jedoch immer nur den Rest nach der Teilung. Um anzuzeigen, dass wir nach den Regeln der Modulo-Arithmetik und nicht nach denen der üblichen Arithmetik rechnen, schreibt man den Modul dazu. Man sagt dann zum Beispiel „4 *modulo* 5“, schreibt aber kurz „4 *mod* 5“. Also an einem Beispiel demonstriert:

$4 \; mod \; 5 + 3 \; mod \; 5 = 7 \; mod \; 5 = 2 \; mod \; 5$

Dieselben Regeln gelten auch für das Multiplizieren sowie die anderen Grundrechenarten:

$4 \; mod \; 5 * 2 \; mod \; 5 = 8 \; mod \; 5 = 3 \; mod \; 5$

7.5.3. Eine Array-Klasse

Bei dem in C/C++ eingebauten Datentyp des Arrays werden zur Laufzeit keine Überprüfungen in Bezug auf die Gültigkeit aktueller Indices durchgeführt. Entwerfen Sie eine Klassenschablone `Array<T>`, die beim lesenden und schreibenden Zugriff auf ihre Elemente eine entsprechende Überprüfung durchführt. Achten Sie darauf, dass Arrayobjekte bei gleicher Größe komplett (mit dem Zuweisungsoperator =) einander zugewiesen werden können. Dies ist ein weiterer Vorteil gegenüber Standardarrays in C/C++.

7.6. Lösungen

7.6.1. Ein Stapel dynamischer Länge

```
01: // interface of class Stack<T>
02: template<class T>
03: class Stack
04: {
05:     // output operator
06:     template<typename T>
07:     friend ostream& operator<< (ostream&, const Stack<T>&);
08:
09: private:
10:     // internal data type 'StackItem'
11:     typedef struct StackItem
12:     {
13:         T m_item;
14:         StackItem* m_next;
15:     }
16:     StackItem;
17:
18:     // private member data
19:     StackItem* m_root;
20:     size_t m_items;
21:
22:     // private helper methods
23:     void release();
24:     StackItem* copy(const StackItem* list);
25:
26: public:
27:     // c'tors/d'tor
28:     Stack();
29:     Stack(const Stack& s);
30:     ~Stack();
31:
32:     // operators
33:     Stack& operator=(const Stack& s);
34:     T Stack<T>::operator[](size_t) const;
35:
36:     // public interface
37:     void push(T item);
38:     T pop();
39:     bool empty() const { return m_root == 0; }
40:     size_t size() const { return m_items; }
41: };
```

Beispiel 7.12. Schnittstelle der Klasse ***`Stack<T>`***.

```
001: // default c'tor
002: template <typename T>
003: Stack<T>::Stack()
004: {
005:     m_root = (StackItem*) 0;
006:     m_items = 0;
007: }
008:
009: // copy c'tor
010: template <typename T>
011: Stack<T>::Stack(const Stack& s)
012: {
013:    m_root = copy(s.m_root);
014:    m_items = s.m_items;
015: }
016:
017: // d'tor
018: template <typename T>
019: Stack<T>::~Stack()
020: {
021:     release();
022: }
023:
024: // operators
025: template <typename T>
```

```
026: Stack<T>& Stack<T>::operator=(const Stack<T>& s)
027: {
028:     if (this != &s)
029:     {
030:         // release left side
031:         this -> ~Stack();
032:
033:         // copy right side
034:         m_root = copy(s.m_root);
035:         m_items = s.m_items;
036:     }
037:
038:     return *this;
039: }
040:
041: template<class T>
042: T Stack<T>::operator[](size_t index) const // read-only
043: {
044:     StackItem* elem = m_root;
045:     while (index != 0)
046:     {
047:         elem = elem -> m_next;
048:         -- index;
049:     }
050:     return elem -> m_item;
051: }
052:
053: // public interface
054: template <typename T>
055: void Stack<T>::push(T item)
056: {
057:    StackItem* elem = new StackItem;
058:    elem -> m_item = item;
059:    elem -> m_next = m_root;
060:    m_root = elem;
061:    ++ m_items;
062: }
063:
064: template <typename T>
065: T Stack<T>::pop()
066: {
067:     StackItem* elem = m_root;
068:     m_root = m_root -> m_next;
069:     T item = elem -> m_item;
070:     delete elem;
071:     -- m_items;
072:     return item;
073: }
074:
075: // private interface
076: template <typename T>
077: void Stack<T>::release()
078: {
079:     while (!empty())
080:         pop();
081: }
082:
083: template<class T>
084: typename Stack<T>::StackItem* Stack<T>::copy(const StackItem* list)
085: {
086:     StackItem* first;
087:     StackItem** last = &first;
088:
089:     while (list != (StackItem*) 0)
090:     {
091:         *last = new StackItem;
092:         (*last) -> m_item = list -> m_item;
093:         last = &(*last) -> m_next;
094:         list = list -> m_next;
095:     }
096:
097:     *last = (StackItem*) 0;
098:     return first;
099: }
100:
101: // ouput operator
102: template <typename T>
103: ostream& operator<< (ostream& os, const Stack<T>& s)
104: {
105:     os << '[';
106:
107:     typename Stack<T>::StackItem* current = s.m_root;
108:     while (current != (Stack<T>::StackItem*) 0)
109:     {
```

```
110:         os << current -> m_item;
111:         if (current -> m_next != (typename Stack<T>::StackItem*) 0)
112:             os << ',';
113:
114:         current = current -> m_next;
115:     }
116:     os << ']';
117:
118:     return os;
119: }
```

Beispiel 7.13. Implementierung der Klasse **`Stack<T>`**.

7.6.2. Restklassenarithmetik mit Klassenschablonen

In Listing 7.14 finden Sie eine Klassenschablone `Z_mod_nZ` vor, die die Regeln der Restklassenarithmetik beherrscht:

```
001: #include <iostream>
002: using namespace std;
003:
004: template <int n>
005: class Z_mod_nZ
006: {
007: public:
008:     // c'tors
009:     Z_mod_nZ(int z = 0);
010:
011: private:
012:     // member data
013:     int m_z;
014:
015:     // private helper methods
016:     void Normalize();
017:
018: public:
019:     // conversion operator
020:     operator int() const { return m_z; }
021:
022:     // public arithmetic assignment operators
023:     template<typename T>
024:     friend Z_mod_nZ& operator += (Z_mod_nZ<n>&, int);
025:     template<typename T>
026:     friend Z_mod_nZ& operator -= (Z_mod_nZ<n>&, int);
027:     template<typename T>
028:     friend Z_mod_nZ& operator *= (Z_mod_nZ<n>&, int);
029:     template<typename T>
030:     friend Z_mod_nZ& operator /= (Z_mod_nZ<n>&, int);
031:
032:     // increment/decrement operators (prefix/postfix version)
033:     template<typename T>
034:     friend Z_mod_nZ& operator++ (Z_mod_nZ&);      // prefix increment
035:     template<typename T>
036:     friend Z_mod_nZ  operator++ (Z_mod_nZ&, int); // postfix increment
037:     template<typename T>
038:     friend Z_mod_nZ& operator-- (Z_mod_nZ&);      // prefix decrement
039:     template<typename T>
040:     friend Z_mod_nZ  operator-- (Z_mod_nZ&, int); // postfix decrement
041:
042:     // output
043:     template<typename T>
044:     friend ostream& operator << (ostream &o, const Z_mod_nZ<n>& z);
045: };
046:
047: // implementation of c'tors
048: template <int n>
049: Z_mod_nZ<n>::Z_mod_nZ(int z = 0) : m_z(z)
050: {
051:     Normalize();
052: }
053:
054: // implementation of friend operators
055: template <int n>
056: Z_mod_nZ<n>& operator += (Z_mod_nZ<n>& z, int i)
057: {
058:     z.m_z += i;
059:     z.Normalize();
```

```
060:     return z;
061: }
062:
063: template <int n>
064: Z_mod_nZ<n>& operator -= (Z_mod_nZ<n>& z, int i)
065: {
066:     z.m_z -= i;
067:     z.Normalize();
068:     return z;
069: }
070:
071: template <int n>
072: Z_mod_nZ<n>& operator *= (Z_mod_nZ<n>& z, int i)
073: {
074:     z.m_z *= i;
075:     z.Normalize();
076:     return z;
077: }
078:
079: template <int n>
080: Z_mod_nZ<n>& operator /= (Z_mod_nZ<n>& z, int i)
081: {
082:     z.m_z /= i;
083:     z.Normalize();
084:     return z;
085: }
086:
087: template <int n>
088: void Z_mod_nZ<n>::Normalize()
089: {
090:     m_z = (m_z < 0) ?
091:         (n - ((-m_z) % n)) :
092:         (m_z % n);
093: }
094:
095: // increment/decrement operators (prefix/postfix version)
096: template <int n>
097: Z_mod_nZ<n>& operator++ (Z_mod_nZ<n>& z)     // prefix increment
098: {
099:     ++ z.m_z;
100:     z.Normalize();
101:     return z;
102: }
103:
104: template <int n>
105: Z_mod_nZ<n> operator++ (Z_mod_nZ<n>& z, int) // postfix increment
106: {
107:     Z_mod_nZ<n> tmp = z;
108:     z.m_z++;
109:     z.Normalize();
110:     return tmp;
111: }
112:
113: template <int n>
114: Z_mod_nZ<n>& operator-- (Z_mod_nZ<n>& z)     // prefix decrement
115: {
116:     -- z.m_z;
117:     z.Normalize();
118:     return z;
119: }
120:
121: template <int n>
122: Z_mod_nZ<n> operator-- (Z_mod_nZ<n>& z, int) // postfix decrement
123: {
124:     Z_mod_nZ<n> tmp = z;
125:     z.m_z --;
126:     z.Normalize();
127:     return tmp;
128: }
129:
130: // implementation of global operators
131: template <int n>
132: Z_mod_nZ<n> operator+ (Z_mod_nZ<n> &a, const Z_mod_nZ<n>& b)
133: {
134:     Z_mod_nZ<n> c(a);
135:     c += b;
136:     return c;
137: }
138:
139: template <int n>
140: Z_mod_nZ<n> operator- (Z_mod_nZ<n> &a, const Z_mod_nZ<n>& b)
141: {
142:     Z_mod_nZ<n> c(a);
143:     c -= b;
```

```
144:     return c;
145: }
146:
147: template <int n>
148: Z_mod_nZ<n> operator* (Z_mod_nZ<n> &a, const Z_mod_nZ<n>& b)
149: {
150:     Z_mod_nZ<n> c(a);
151:     c *= b;
152:     return c;
153: }
154:
155: template <int n>
156: Z_mod_nZ<n> operator/ (Z_mod_nZ<n> &a, const Z_mod_nZ<n>& b)
157: {
158:     Z_mod_nZ<n> c(a);
159:     c /= b;
160:     return c;
161: }
162:
163: // output operator
164: template <int n>
165: ostream& operator<< (ostream& os, const Z_mod_nZ<n>& z)
166: {
167:     os << z.m_z << '(' << n << ')';
168:     return os;
169: }
```

Beispiel 7.14. Restklassenarithmetik mit Klassenschablonen.

Es folgt eine einfache Anwendung der `Z_mod_nZ`-Klasse:

```
01: void main ()
02: {
03:     Z_mod_nZ<7> a(3);
04:     Z_mod_nZ<7> b(12);
05:
06:     cout  <<  a  <<  '+'  <<  b  <<  '='  <<  a+b  <<  endl;
07:     cout << (a+b) << endl;
08:
09:     cout  <<  a  <<  '-'  <<  b  <<  '='  <<  a-b  <<  endl;
10:     cout << (a-b) << endl;
11: }
```

Die Ausgabe lautet

```
3(7)+5(7)=1(7)
1(7)
3(7)-5(7)=5(7)
5(7)
```

Bemerkung: Die Anregung zu diesem Beispiel stammt aus *http://www.math.uni-wuppertal.de/~axel/skripte/oop/oop16_3_2.html*.

7.6.3. Eine Array-Klasse

```
001: #include <iostream>
002: using namespace std;
003:
004: #include "Point.h"
005:
006: // interface
007: template <class T>
008: class Array
009: {
010: private:
011:     // private member data
012:     int m_size;
013:     T*  m_data;
014:
015: public:
016:     // c'tors / d'tor
017:     Array();
018:     Array(int size);
019:     Array(const Array<T>&);
```

```
020:     ~Array();
021:
022:     // public interface
023:     int Size() const;
024:     T& operator [] (int) const;
025:     Array<T>& operator= (const Array<T>&);
026:
027:     // output operator
028:     template<typename T>
029:     friend ostream& operator<< (ostream&, const Array<T>&);
030:
031: private:
032:     // private helper method
033:     void Allocate();
034: };
035:
036: // implementation: c'tors / d'tor
037: template <class T>
038: Array<T>::Array() : m_size(0)
039: {
040:     Allocate();
041: }
042:
043: template <class T>
044: Array<T>::Array(int size) : m_size(size)
045: {
046:     Allocate();
047: }
048:
049: template <class T>
050: Array<T>::Array(const Array<T>& a) : m_size(a.m_size)
051: {
052:     Allocate();
053:     for (int i = 0; i < m_size; i ++)
054:         m_data[i] = a.m_data[i];
055: }
056:
057: template <class T>
058: Array<T>::~Array()
059: {
060:     delete[] m_data;
061: }
062:
063: // implementation: public interface
064: template <class T>
065: void Array<T>::Allocate()
066: {
067:     m_data = new T[m_size];
068:     for (int i = 0; i < m_size; i ++)
069:     {
070:         m_data[i] = T();
071:     }
072: }
073:
074: template <class T>
075: int Array<T>::Size() const
076: {
077:     return m_size;
078: }
079:
080: template <typename T>
081: Array<T>& Array<T>::operator= (const Array<T>& a)
082: {
083:     if (m_size != a.m_size)
084:     {
085:         cout << "Error: Index out of bounds!";
086:     }
087:     else if (this != &a)
088:     {
089:         T* tmp = new T[m_size];
090:         for (int i = 0; i < m_size; i ++)
091:             tmp[i] = a.m_data[i];
092:
093:         delete m_data;
094:         m_data = tmp;
095:     }
096:
097:     return *this;
098: }
099:
100: template <typename T>
101: T& Array<T>::operator[] (int i) const
102: {
103:     if (i < 0 || i >= m_size)
```

```
104:         cout << "Error: Index out of bounds!";
105:
106:     return m_data[i];
107: }
108:
109: // output operator
110: template <typename T>
111: ostream& operator<< (ostream& os, const Array<T>& a)
112: {
113:     os << "[";
114:     for (int i = 0; i < a.m_size; i ++)
115:     {
116:         os << a.m_data[i];
117:         if (i < a.m_size - 1)
118:             os << ',';
119:     }
120:     os << "]";
121:     return os;
122: }
```

Beispiel 7.15. Eine Array-Klasse mit integrierter Indexüberprüfung.

Es folgt eine einfache Anwendung der `Array`-Klasse:

```
01: void main ()
02: {
03:     // testing assignment- and index-operator
04:     Array<int> a(10);
05:     Array<int> b(10);
06:
07:     for (int i = 0; i < a.Size(); i ++)
08:         a[i] = i;
09:     cout << a << endl;
10:     cout << b << endl;
11:
12:     b = a;
13:     cout << a << endl;
14:     cout << b << endl;
15: }
```

Die Ausgabe lautet

```
[0,1,2,3,4,5,6,7,8,9]
[0,0,0,0,0,0,0,0,0,0]
[0,1,2,3,4,5,6,7,8,9]
[0,1,2,3,4,5,6,7,8,9]
```

Bemerkung: Die Anregung zu diesem Beispiel stammt aus http://www.math.uni-wuppertal.de/~axel/skripte/oop/oop16_3_1.html.

Kapitel 8. Übungen

8.1. Uhrzeit: Die Klasse `Time`

8.1.1. Aufgabe

Entwerfen Sie für Objekte, die eine Uhrzeit mit Stunde, Minute und Sekunde repräsentieren, eine geeignete Klasse `Time`. Die Stunde kann einen Wert zwischen 0 und 23 einnehmen, die Minuten und Sekunden Werte zwischen 0 und 59. Die Gestaltung der einzelnen Klassenelemente entnehmen Sie bitte den folgenden Codefragmenten:

Codefragment zum Testen der Konstruktoren:

```
// testing c'tors
Time t1;
cout << t1 << endl;
Time t2 (0, 30, 12);
cout << t2 << endl;
Time t3 ("09:30:00");
cout << t3 << endl;
Time t4 (24*60*60 - 1);
cout << t4 << endl;
```

Ausgabe:

```
00:00:00
12:30:00
09:30:00
23:59:59
```

Codefragment zum Testen der `Add`-Methode:

```
// testing 'Add'
Time t1 (0, 0, 12);
Time t2 (33, 33, 3);
for (int i = 0; i < 5; i ++)
{
    t1.Add (t2);
    cout << t1 << endl;
}
```

Ausgabe:

```
15:33:33
19:07:06
22:40:39
02:14:12
05:47:45
```

Codefragment zum Testen der `Increment`-Methode:

```
// testing 'Increment'
Time t (55, 59, 23);
for (int i = 0; i < 8; i ++)
{
    t.Increment ();
    cout << t << endl;
}
```

Ausgabe:

```
23:59:56
23:59:57
23:59:58
23:59:59
00:00:00
00:00:01
00:00:02
00:00:03
```

Codefragment zum Testen der `Diff`-Methode:

```
// testing 'Diff'
Time t1;
Time t2("23:59:59");
Time t3 = t1.Diff(t2);
cout << t3 << endl;
t3 =  t2.Diff(t1);
cout << t3 << endl;
```

Ausgabe:

```
23:59:59
23:59:59
```

Codefragment zum Testen der arithmetischen Operatoren:

```
// testing operators
Time t1 (15, 30, 6);
Time t2 = t1 + t1;
cout << t2 << endl;
t2 += t1;
cout << t2 << endl;
t2 -= 120;
cout << t2 << endl;
t2 -= t1;
cout << t2 << endl;
```

Ausgabe:

```
13:00:30
19:30:45
19:28:45
12:58:30
```

Codefragment zum Testen des Inkrement- und Dekrement-Operators:

```
// testing increment/decrement operator
Time t1 (0, 0, 12);
Time t2 = t1++;
cout << t2 << endl;
t2 = ++t1;
cout << t2 << endl;
t2 = t1--;
cout << t2 << endl;
t2 = --t1;
cout << t2 << endl;
```

Ausgabe:

```
12:00:00
12:00:02
12:00:02
12:00:00
```

Codefragment zum Testen des `int`-Konvertierungsoperators:

```
// testing conversion operator
Time t;
t = 60*60 + 60 + 1;
cout << "Time:    " << t << endl;
cout << "Seconds: " << (int) t << endl;
```

Ausgabe:

```
Time:    01:01:01
Seconds: 3661
```

Codefragment zum Testen der Ein- und Ausgabe (Konsoleneingaben fett gedruckt):

```
Time t;
cin >> t;
cout << t << endl;
```

Ausgabe:

```
Hours [hh]: 12
Minutes [mm]: 34
Seconds [ss]: 56
12:34:56
```

8.1.2. Lösung

```
01: class Time
02: {
03: private:
04:     int m_seconds;
05:     int m_minutes;
06:     int m_hours;
```

```
07:
08: public:
09:     // c'tors
10:     Time ();
11:     Time (int seconds, int minutes, int hours);
12:     Time (const Time&);  // copy c'tor
13:     Time (int seconds);  // conversion c'tor
14:     Time (char*);        // conversion c'tor
15:
16:     // getter // setter
17:     int GetSeconds () const { return m_seconds; };
18:     int GetMinutes () const { return m_minutes; };
19:     int GetHours () const { return m_hours; };
20:     void SetSeconds (int seconds);
21:     void SetMinutes (int minutes);
22:     void SetHours (int hours);
23:
24:     // public interface
25:     void Reset ();
26:     void Add (const Time&);
27:     void Sub (const Time&);
28:     Time Diff (const Time&) const;
29:     void Increment ();
30:     void Decrement ();
31:
32:     // standard operators
33:     Time& operator= (const Time&);
34:
35:     // arithmetic operators
36:     Time operator+ (const Time&) const;
37:     Time operator- (const Time&) const;
38:
39:     // arithmetic-assignment operators
40:     Time operator+= (const Time&);
41:     Time operator-= (const Time&);
42:     Time operator+= (int seconds);
43:     Time operator-= (int seconds);
44:
45:     // increment/decrement operators (prefix/postfix version)
46:     friend Time& operator++ (Time&);           // prefix increment
47:     friend const Time operator++ (Time&, int); // postfix increment
48:     friend Time& operator-- (Time&);           // prefix decrement
49:     friend const Time operator-- (Time&, int); // postfix decrement
50:
51:     // comparison operators
52:     friend bool operator== (const Time&, const Time&);
53:     friend bool operator!= (const Time&, const Time&);
54:     friend bool operator<= (const Time&, const Time&);
55:     friend bool operator<  (const Time&, const Time&);
56:     friend bool operator>= (const Time&, const Time&);
57:     friend bool operator>  (const Time&, const Time&);
58:
59:     // conversion operator
60:     operator int();
61:
62:     // input / output
63:     friend ostream& operator<< (ostream&, const Time&);
64:     friend istream& operator>> (istream&, Time&);
65:
66: private:
67:     // helper methods
68:     int TimeToSeconds ();
69:     void SecondsToTime (int seconds);
70: };
```

Beispiel 8.1. Klasse `Time`: Schnittstelle.

```
001: // c'tors
002: Time::Time () : m_seconds(0), m_minutes(0), m_hours (0) {}  // default c'tor
003:
004: Time::Time (int seconds, int minutes, int hours)  // user-defined c'tor
005: {
006:     m_seconds = (0 <= seconds && seconds < 60) ? seconds : 0;
007:     m_minutes = (0 <= minutes && minutes < 60) ? minutes : 0;
008:     m_hours = (0 <= hours && hours < 24) ? hours : 0;
009: }
010:
011: Time::Time (char* s)  // user-defined c'tor
012: {
013:     // expecting format "hh:mm:ss"
```

```
014:     int hours = 10 * (s[0] - '0') + (s[1] - '0');
015:     m_hours = (0 <= hours && hours < 24) ? hours : 0;
016:     s += 3;  // skip hours and colon
017:     int minutes = 10 * (s[0] - '0') + (s[1] - '0');
018:     m_minutes = (0 <= minutes && minutes < 60) ? minutes : 0;
019:     s += 3;  // skip minutes and colon
020:     int seconds = 10 * (s[0] - '0') + (s[1] - '0');
021:     m_seconds = (0 <= seconds && seconds < 60) ? seconds : 0;
022: }
023:
024: Time::Time (const Time& t)   // copy c'tor
025: {
026:     m_seconds = t.m_seconds;
027:     m_minutes = t.m_minutes;
028:     m_hours = t.m_hours;
029: }
030:
031: Time::Time (int seconds)     // conversion c'tor
032: {
033:     if (0 <= seconds && seconds <= 24 * 60 * 60)
034:     {
035:         SecondsToTime (seconds);
036:     }
037:     else
038:     {
039:         m_seconds = m_minutes = m_hours = 0;
040:     }
041: }
042:
043: // getter // setter
044: void Time::SetSeconds (int seconds)
045: {
046:     m_seconds = (0 <= seconds && seconds < 60) ? seconds : 0;
047: }
048:
049: void Time::SetMinutes (int minutes)
050: {
051:     m_minutes = (0 <= minutes && minutes < 60) ? minutes : 0;
052: }
053:
054: void Time::SetHours (int hours)
055: {
056:     m_hours = (0 <= hours && hours < 24) ? hours : 0;
057: }
058:
059: // public interface
060: void Time::Reset ()
061: {
062:     m_seconds = 0;
063:     m_minutes = 0;
064:     m_hours = 0;
065: }
066:
067: void Time::Add (const Time& t)
068: {
069:     m_seconds += t.m_seconds;
070:     m_minutes += t.m_minutes;
071:     m_hours += t.m_hours;
072:
073:     // normalize object
074:     m_minutes += m_seconds / 60;
075:     m_hours += m_minutes / 60;
076:     m_seconds = m_seconds % 60;
077:     m_minutes = m_minutes % 60;
078:     m_hours = m_hours % 24;
079: }
080:
081: void Time::Sub (const Time& t)
082: {
083:     int seconds =
084:         (m_hours * 3600 + m_minutes * 60 + m_seconds) -
085:         (t.m_hours * 3600 + t.m_minutes * 60 + t.m_seconds);
086:
087:     if (seconds < 0)
088:         seconds += 24 * 60 * 60;
089:
090:     // transform total seconds into hour, minutes and seconds
091:     SecondsToTime (seconds);
092: }
093:
094: Time Time::Diff (const Time& t) const
095: {
096:     Time tmp;
097:     if (*this <= t)
```

```
098:         tmp = t - *this;
099:     else
100:         tmp = *this - t;
101:
102:     return tmp;
103: }
104:
105: void Time::Increment ()
106: {
107:     m_seconds ++;
108:     if (m_seconds >= 60)
109:     {
110:         m_seconds = 0;
111:         m_minutes ++;
112:         if (m_minutes >= 60)
113:         {
114:             m_minutes = 0;
115:             m_hours ++;
116:             if (m_hours >= 24)
117:             {
118:                 m_hours = 0;
119:             }
120:         }
121:     }
122: }
123:
124: void Time::Decrement ()
125: {
126:     m_seconds --;
127:     if (m_seconds < 0)
128:     {
129:         m_seconds = 59;
130:         m_minutes --;
131:         if (m_minutes < 0)
132:         {
133:             m_minutes = 59;
134:             m_hours --;
135:             if (m_hours < 0)
136:             {
137:                 m_hours = 23;
138:             }
139:         }
140:     }
141: }
142:
143: // standard operators
144: Time& Time::operator= (const Time& t)
145: {
146:     if (&t != this)
147:     {
148:         m_seconds = t.m_seconds;
149:         m_minutes = t.m_minutes;
150:         m_hours = t.m_hours;
151:     }
152:
153:     return *this;
154: }
155:
156: // arithmetic operators
157: Time Time::operator+ (const Time& t) const
158: {
159:     Time tmp(*this);
160:     tmp.Add(t);
161:     return tmp;
162: }
163:
164: Time Time::operator- (const Time& t) const
165: {
166:     Time tmp(*this);
167:     tmp.Sub(t);
168:     return tmp;
169: }
170:
171: Time Time::operator+= (const Time& t)
172: {
173:     Add(t);
174:     return *this;
175: }
176:
177: Time Time::operator-= (const Time& t)
178: {
179:     Sub(t);
180:     return *this;
181: }
```

```
182:
183: Time Time::operator+= (int seconds)
184: {
185:     for (int i = 0; i < seconds; i ++)
186:         Increment ();
187:
188:     return *this;
189: }
190:
191: Time Time::operator-= (int seconds)
192: {
193:     for (int i = 0; i < seconds; i ++)
194:         Decrement ();
195:
196:     return *this;
197: }
198:
199: // increment operator: prefix version
200: Time& operator++ (Time& t)
201: {
202:     t += 1;
203:     return t;
204: }
205:
206: // decrement operator: prefix version
207: Time& operator-- (Time& t)
208: {
209:     t -= 1;
210:     return t;
211: }
212:
213: // increment operator: postfix version
214: const Time operator++ (Time& t, int)
215: {
216:     Time tmp (t);  // construct a copy
217:     ++ t;          // increment number
218:     return tmp;    // return the copy
219: }
220:
221: // decrement operator: postfix version
222: const Time operator-- (Time& t, int)
223: {
224:     Time tmp (t);  // construct a copy
225:     -- t;          // decrement number
226:     return tmp;    // return the copy
227: }
228:
229: // comparison operators
230: bool operator== (const Time& t1, const Time& t2)
231: {
232:     return
233:         t1.m_seconds == t2.m_seconds &&
234:         t1.m_minutes == t2.m_minutes &&
235:         t1.m_hours == t2.m_hours;
236: }
237:
238: bool operator!= (const Time& t1, const Time& t2)
239: {
240:     return ! (t1 == t2);
241: }
242:
243: bool operator<= (const Time& t1, const Time& t2)
244: {
245:     return t1 < t2 || t1 == t2;
246: }
247:
248: bool operator<  (const Time& t1, const Time& t2)
249: {
250:     if (t1.m_hours < t2.m_hours)
251:         return true;
252:
253:     if (t1.m_hours == t2.m_hours && t1.m_minutes < t2.m_minutes)
254:         return true;
255:
256:     if (t1.m_hours == t2.m_hours && t1.m_minutes == t2.m_minutes && t1.m_seconds <
257:         return true;
258:
259:     return false;
260: }
261:
262: bool operator>= (const Time& t1, const Time& t2)
263: {
264:     return ! (t1 < t2);
265: }
```

```
266:
267: bool operator>  (const Time& t1, const Time& t2)
268: {
269:     return ! (t1 <= t2);
270: }
271:
272: // conversion operator
273: Time::operator int()
274: {
275:     return TimeToSeconds();
276: }
277:
278: // input / output
279: istream& operator>> (istream& is, Time& t)
280: {
281:     cout << "Hours [hh]: ";
282:     is >> t.m_hours;
283:     cout << "Minutes [mm]: ";
284:     is >> t.m_minutes;
285:     cout << "Seconds [ss]: ";
286:     is >> t.m_seconds;
287:
288:     return is;
289: }
290:
291: ostream& operator<< (ostream& os, const Time& t)
292: {
293:     os << setw(2) << setfill('0') << t.m_hours << ":";
294:     os << setw(2) << setfill('0') << t.m_minutes << ":";
295:     os << setw(2) << setfill('0') << t.m_seconds;
296:
297:     return os;
298: }
299:
300: // helper methods
301: int Time::TimeToSeconds ()
302: {
303:     return m_hours * 3600 + m_minutes * 60 + m_seconds;
304: }
305:
306: void Time::SecondsToTime (int seconds)
307: {
308:     m_hours = seconds / 3600;
309:     seconds = seconds % 3600;
310:     m_minutes = seconds / 60;
311:     m_seconds = seconds % 60;
312: }
```

Beispiel 8.2. Klasse `Time`: Implementierung.

8.2. Rationale Zahlen: Die Klasse `Fraction`

8.2.1. Aufgabe

In dieser Übung wollen wir uns der Klasse `Fraction` in einer zusammenfassenden Betrachtung annehmen. Einige Aspekte dieser Klasse haben wir in unserem Studium der Programmiersprache C++ schon betrachtet, abschließend werden die folgenden Eigenschaften und Verbesserungen durchgeführt:

- Objekte der Klasse `Fraction` sollen die von ihr verwaltete rationale Zahl immer in einer optimal gekürzten Form verwalteten. Schreiben Sie dazu eine Methode `Reduce`, die den Bruch optimal kürzt.
- Rationale Zahlen können sowohl positiv als auch negativ sein. Um die klasseninterne Arbeitsweise der einzelnen Methoden zu vereinfachen, soll die Regel gelten, dass der Nenner einer rationalen Zahl immer positiv ist. Damit kann der Zähler eines `Fraction`-Objekts in Abhängigkeit vom Vorzeichen der rationalen Zahl entweder positiv oder negativ sein. Achten Sie darauf, dass bei allen Änderungen am Objektzustand (zum Beispiel nach einer Subtraktion) der Nenner positiv ist.
- Gängige Konstruktoren zur Objekterzeugung, Zuweisungsoperator.
- Implementieren Sie die folgenden mathematischen Rechenoperationen:
  - Grundrechenoperationen (`+`, `-`, `*` und `/`).
  - Grundrechenoperationen in Verbindung mit dem Zuweisungsoperator (`+=`, `-=`, `*=` und `/=`).
  - Inkrement- und Dekrementoperator (`++` und `--`).
  - Inverse einer rationalen Zahl (Operator `~`).
  - Unärer Minus-Operator (`-`).
  - Vergleichsoperatoren (`==`, `!=`, `<`, `<=`, `>` und `>=`).
- Methode `Gcd` (*greatest common divisor*) zur Bestimmung des größten gemeinsamen Teilers von Zähler und Nenner (ggT). Die `Gcd`-Methode kann von der `Reduce`-Methode verwendet werden.
- Operatoren zur Typumwandlung. Ein `Fraction`-Objekt soll in einem arithmetischen Ausdruck auch dann verwendet werden können, wenn auf Grund des Kontextes ein `float`- oder `double`-Wert erwartet wird.
- Ein- und Ausgabe. Eine rationale Zahl soll in der Form „*zähler* / *nenner*“ ausgegeben werden. Für Eingaben ist dasselbe Format zu Grunde zu legen. Zwischen den numerischen Werten und dem Schrägstrich sind beliebige Leerzeichen und Tabulatoren erlaubt.

8.2.2. Lösung

```
01: #include <iostream>
02: using namespace std;
03:
04: class Fraction
05: {
06: private:
07:     // private member data
08:     int m_num;    // numerator
09:     int m_denom;  // denominator
10:
```

```
11: public:
12:     // c'tors
13:     Fraction ();
14:     Fraction (int, int);
15:     Fraction (const Fraction&);
16:
17:     // conversion c'tor
18:     Fraction (int);
19:
20:     // getter / setter
21:     int GetNum () const { return m_num; };
22:     void SetNum (int num);
23:     int GetDenom () const { return m_denom; };
24:     void SetDenom (int denom);
25:
26:     // unary arithmetic operators
27:     friend Fraction operator- (const Fraction&);
28:     friend Fraction operator~ (const Fraction&);
29:
30:     // binary arithmetic operators
31:     friend Fraction operator+ (const Fraction&, const Fraction&);
32:     friend Fraction operator- (const Fraction&, const Fraction&);
33:     friend Fraction operator* (const Fraction&, const Fraction&);
34:     friend Fraction operator/ (const Fraction&, const Fraction&);
35:
36:     // arithmetic-assignment operators
37:     friend const Fraction& operator+= (Fraction&, const Fraction&);
38:     friend const Fraction& operator-= (Fraction&, const Fraction&);
39:     friend const Fraction& operator*= (Fraction&, const Fraction&);
40:     friend const Fraction& operator/= (Fraction&, const Fraction&);
41:
42:     // assigment operator
43:     Fraction& operator= (const Fraction&);
44:
45:     // increment/decrement operators (prefix/postfix version)
46:     friend Fraction& operator++ (Fraction&);           // prefix increment
47:     friend const Fraction operator++ (Fraction&, int); // postfix increment
48:     friend Fraction& operator-- (Fraction&);           // prefix decrement
49:     friend const Fraction operator-- (Fraction&, int); // postfix decrement
50:
51:     // comparison operators
52:     friend bool operator<= (const Fraction&, const Fraction&);
53:     friend bool operator<  (const Fraction&, const Fraction&);
54:     friend bool operator>= (const Fraction&, const Fraction&);
55:     friend bool operator>  (const Fraction&, const Fraction&);
56:     friend bool operator== (const Fraction&, const Fraction&);
57:     friend bool operator!= (const Fraction&, const Fraction&);
58:
59:     // type conversion operator (Fraction -> double)
60:     operator double ();
61:
62:     // input / output operators
63:     friend ostream& operator<< (ostream&, const Fraction&);
64:     friend istream& operator>> (istream&, Fraction&);
65:
66: private:
67:     // private helper methods
68:     void CheckSigns ();
69:     void Reduce ();
70:
71: public:
72:     // class helper methods
73:     static int Gcd (int n, int m);
74: };
```

Beispiel 8.3. Klasse `Fraction`: Schnittstelle.

```
001: // c'tors
002: Fraction::Fraction () : m_num (0), m_denom (1)
003: {
004: }
005:
006: Fraction::Fraction (int num, int denom)
007: {
008:     m_num = num;
009:     m_denom = (denom == 0) ? 1 : denom;
010:     CheckSigns ();
011:     Reduce ();
012: }
013:
```

```
014: Fraction::Fraction (const Fraction& f)
015: {
016:     m_num = f.m_num;
017:     m_denom = f.m_denom;
018: }
019:
020: // conversion c'tor
021: Fraction::Fraction (int num)
022: {
023:     m_num = num;
024:     m_denom = 1;
025: }
026:
027: // setter
028: void Fraction::SetNum (int num)
029: {
030:     m_num = num;
031:     Reduce ();
032: }
033:
034: void Fraction::SetDenom (int denom)
035: {
036:     if (denom != 0)
037:     {
038:         m_denom = denom;
039:         CheckSigns ();
040:         Reduce ();
041:     }
042: }
043:
044: // implementation of unary arithmetic operators
045: Fraction operator- (const Fraction& f)
046: {
047:     return Fraction (-f.m_num, f.m_denom);
048: }
049:
050: Fraction operator~ (const Fraction& f)
051: {
052:     return Fraction (f.m_denom, f.m_num);
053: }
054:
055: // implementation of binary arithmetic operators
056: Fraction operator+ (const Fraction& f1, const Fraction& f2)
057: {
058:     int num = f1.m_num * f2.m_denom + f1.m_denom * f2.m_num;
059:     int denom = f1.m_denom * f2.m_denom;
060:     return Fraction (num, denom);
061: }
062:
063: Fraction operator- (const Fraction& f1, const Fraction& f2)
064: {
065:     int num = f1.m_num * f2.m_denom - f1.m_denom * f2.m_num;
066:     int denom = f1.m_denom * f2.m_denom;
067:     return Fraction (num, denom);
068: }
069:
070: Fraction operator* (const Fraction& f1, const Fraction& f2)
071: {
072:     int num = f1.m_num * f2.m_num;
073:     int denom = f1.m_denom * f2.m_denom;
074:     return Fraction (num, denom);
075: }
076:
077: Fraction operator/ (const Fraction& f1, const Fraction& f2)
078: {
079:     int num = f1.m_num * f2.m_denom;
080:     int denom = f1.m_denom * f2.m_num;
081:     return Fraction (num, denom);
082: }
083:
084: // assigment operator
085: Fraction& Fraction::operator= (const Fraction& f)
086: {
087:     m_num = f.m_num;
088:     m_denom = f.m_denom;
089:     return *this;
090: }
091:
092: // addition-assignment operators
093: const Fraction& operator+= (Fraction& f1, const Fraction& f2)
094: {
095:     f1.m_num = f1.m_num * f2.m_denom + f1.m_denom * f2.m_num;
096:     f1.m_denom = f1.m_denom * f2.m_denom;
097:     f1.Reduce ();
```

```
098:     return f1;
099: }
100:
101: const Fraction& operator-= (Fraction& f1, const Fraction& f2)
102: {
103:     f1.m_num = f1.m_num * f2.m_denom - f1.m_denom * f2.m_num;
104:     f1.m_denom = f1.m_denom * f2.m_denom;
105:     f1.Reduce ();
106:     return f1;
107: }
108:
109: // comparison operators
110: bool operator<= (const Fraction& f1, const Fraction& f2)
111: {
112:     return f1.m_num * f2.m_denom <= f1.m_denom * f2.m_num;
113: }
114:
115: bool operator< (const Fraction& f1, const Fraction& f2)
116: {
117:     return f1.m_num * f2.m_denom < f1.m_denom * f2.m_num;
118: }
119:
120: bool operator>= (const Fraction& f1, const Fraction& f2)
121: {
122:     return ! (f1 < f2);
123: }
124:
125: bool operator> (const Fraction& f1, const Fraction& f2)
126: {
127:     return ! (f1 <= f2);
128: }
129:
130: bool operator== (const Fraction& f1, const Fraction& f2)
131: {
132:     return f1.m_num * f2.m_denom == f1.m_denom * f2.m_num;
133: }
134:
135: bool operator!= (const Fraction& f1, const Fraction& f2)
136: {
137:     return ! (f1 == f2);
138: }
139:
140: // conversion operator (Fraction -> double)
141: Fraction::operator double ()
142: {
143:     return (double) m_num / (double) m_denom;
144: }
145:
146: // increment operator: prefix version
147: Fraction& operator++ (Fraction& f)
148: {
149:     f += 1;
150:     return f;
151: }
152:
153: // decrement operator: prefix version
154: Fraction& operator-- (Fraction& f)
155: {
156:     f -= 1;
157:     return f;
158: }
159:
160: // increment operator: postfix version
161: const Fraction operator++ (Fraction& f, int)
162: {
163:     Fraction tmp (f); // construct a copy
164:     ++ f;             // increment number
165:     return tmp;       // return the copy
166: }
167:
168: // decrement operator: postfix version
169: const Fraction operator-- (Fraction& f, int)
170: {
171:     Fraction tmp (f); // construct a copy
172:     -- f;             // decrement number
173:     return tmp;       // return the copy
174: }
175:
176: // private helper method
177: void Fraction::CheckSigns ()
178: {
179:     // normalize fraction
180:     if (m_denom < 0)
181:     {
```

```
182:         m_num *= -1;
183:         m_denom *= -1;
184:     }
185: }
186:
187: void Fraction::Reduce ()
188: {
189:     int sign = (m_num < 0) ? -1 : 1;
190:     int gcd = Fraction::Gcd (sign * m_num, m_denom);
191:     m_num /= gcd;
192:     m_denom /= gcd;
193: }
194:
195: int Fraction::Gcd (int n, int m)
196: {
197:     if (n != m)
198:     {
199:         // calculate greatest common divisor of numerator and denominator
200:         while (m > 0)
201:         {
202:             int rem = n % m;
203:             n = m;
204:             m = rem;
205:         }
206:     }
207:
208:     return n;
209: }
210:
211: // output operator
212: ostream& operator<< (ostream& os, const Fraction& f)
213: {
214:     os << f.m_num << '/' << f.m_denom;
215:     return os;
216: }
217:
218: // input operator
219: istream& operator>> (istream& is, Fraction& f)
220: {
221:     is >> f.m_num >> f.m_denom;
222:     f.Reduce();
223:     return is;
224: }
```

Beispiel 8.4. Klasse `Fraction`: Implementierung.

```
001: #include "Fraction.h"
002:
003: void DemoUnaryOperators()
004: {
005:     Fraction f (1, 2);
006:     f = -f;
007:     cout << f << endl;
008:     f = ~f;
009:     cout << f << endl;
010: }
011:
012: void DemoBinaryOperators()
013: {
014:     Fraction a (1, 7);
015:     Fraction b (3, 7);
016:     Fraction c;
017:
018:     c = a + b;
019:     cout << c << endl;
020:
021:     c = a + a + a;
022:     cout << c << endl;
023:
024:     c = a - b - a;
025:     cout << c << endl;
026:
027:     c = a * b;
028:     cout << c << endl;
029:
030:     c = a / b;
031:     cout << c << endl;
032: }
033:
034: void DemoAssignmentOperators01()
```

```
035: {
036:     Fraction a (1, 2);
037:     Fraction b (3, 2);
038:     a = b;
039:     cout << a << endl;
040: }
041:
042:
043: void DemoAssigmentOperators02()
044: {
045:     Fraction a (1, 7);
046:     Fraction b (3, 7);
047:     Fraction c (5, 7);
048:
049:     a.operator= (b.operator= (c));
050:     cout << a << endl;
051:
052:     a = b = c;
053:     cout << a << endl;
054: }
055:
056: void DemoArithmeticAssigmentOperators()
057: {
058:     Fraction a (1, 7);
059:     Fraction b (3, 7);
060:     Fraction c (5, 7);
061:
062:     a += b += c;
063:     cout << a << endl;
064:     cout << b << endl;
065:     cout << c << endl;
066:
067:     c -= b -= a;
068:     cout << a << endl;
069:     cout << b << endl;
070:     cout << c << endl;
071: }
072:
073: void DemoComparisonOperators()
074: {
075:     Fraction f (1, 2);
076:     Fraction g (1, 3);
077:
078:     bool b = f < g;
079:     printf ("%d\n", b);
080:
081:     b = f <= g;
082:     printf ("%d\n", b);
083:
084:     b = f > g;
085:     printf ("%d\n", b);
086:
087:     b = f >= g;
088:     printf ("%d\n", b);
089:
090:     b = f == g;
091:     printf ("%d\n", b);
092:
093:     b = f != g;
094:     printf ("%d\n", b);
095: }
096:
097:
098: void DemoTypeConversionOperators()
099: {
100:     Fraction a;
101:
102:     a = 1;
103:     cout << a << endl;
104:
105:     a = a + (Fraction) 1;
106:     a = 1 + (int) (double) a;
107:     cout << a << endl;
108: }
109:
110: void DemoTypeConversionOperator02()
111: {
112:     Fraction f (1, 7);
113:     double d = f;
114:     printf ("d: %g\n", d);
115: }
116:
117: void DemoIncrementOperators()
118: {
```

```
119:     Fraction f (1, 2);
120:     Fraction g;
121:
122:     g = f ++;
123:     cout << g << endl;
124:
125:     g = ++ f;
126:     cout << g << endl;
127:
128:     g = f --;
129:     cout << g << endl;
130:
131:     g = -- f;
132:     cout << g << endl;
133: }
134:
135: void DemoGcd()
136: {
137:     int n = Fraction::Gcd (9, -1);
138:     printf ("Gcd: %d\n", n);
139:
140: }
141:
142: void DemoInputOutput()
143: {
144:     Fraction f (1, 2);
145:     cout << "f: " << f << endl;
146:
147:     Fraction g;
148:     cout << "enter num and denom:" << endl;
149:
150:     cin >> g;
151:     cout << "g: " << g << endl;
152: }
153:
154: void main ()
155: {
156:     DemoUnaryOperators();
157:     DemoBinaryOperators();
158:     DemoAssignmentOperators01();
159:     DemoAssigmentOperators02();
160:     DemoArithmeticAssigmentOperators();
161:     DemoComparisonOperators();
162:     DemoTypeConversionOperators();
163:     DemoTypeConversionOperator02();
164:     DemoIncrementOperators();
165:     DemoGcd();
166:     DemoInputOutput ();
167: }
```

Beispiel 8.5. Klasse `Fraction`: Testrahmen.

Der Testrahmen aus Listing 8.5 führt bei der Ausführung zu folgenden Resultaten:

```
-1/2
-2/1
4/7
3/7
-3/7
3/49
1/3
3/2
5/7
5/7
9/7
8/7
5/7
9/7
-1/7
6/7
0
0
1
1
0
1
1/1
3/1
d: 0.142857
1/2
```

```
5/2
5/2
1/2
Gcd: 9
f: 1/2
enter num and denom:
3 12
g: 1/4
```

8.3. Komplexe Zahlen: Die Klasse `Complex`

8.3.1. Aufgabe

Die Theorie der komplexen Zahlen ist auf den Mathematiker Carl Friedrich Gauß zurückzuführen, der diese erfand, um beispielsweise Gleichungen der Gestalt

(1) $x^2 = -1$

lösen zu können. Mit Hilfe einer genialen Idee, der Erfindung der imaginären Zahlen, lassen sich Gleichungen lösen, für die im Bereich der reellen Zahlen keine Lösung existiert. Der „Urvater" aller imaginären Zahlen ist die Zahl *i* mit der Eigenschaft

$i * i = -1$.

Die Gleichung (1) ist nun mit Hilfe von *i* lösbar, es sind $x_1 = i$ und $x_2 = -i$ die beiden Lösungen. Alle imaginären Zahlen sind Vielfache der Zahl *i* und werden auch so geschrieben, zum Beispiel $3i = 3 * i$ oder $0.99999i = 0.99999 * i$. Komplexe Zahlen sind nun Zahlen, die sich aus einer reellen und einer imaginären Zahl zusammensetzen, man spricht hier vom Real- und Imaginärteil einer komplexen Zahl und stellt sie in der Form

$z = x + iy$

dar, wobei *x* den realen und *y* den durch das *i* gekennzeichneten imaginären Anteil darstellen. Beispiele komplexer Zahlen sind $1 + i2$ oder $1.11111 + i2.22222$ oder aber auch nur $i6$ (sprich diese komplexe Zahl besitzt keinen Realteil) oder aber 5.5 (komplexe Zahl, deren imaginärer Anteil gleich Null ist). Die reellen Zahlen werden als Teilmenge der komplexen Zahlen aufgefasst. Für die grafische Darstellung komplexer Zahlen hat Gauß die nach ihm benannte komplexe (Gaußsche) Zahlenebene eingeführt (siehe Abbildung 8.1). Die Punkte der *x*-Achse eines normalen, kartesischen Koordinatensystems, der *reellen* Achse, entsprechen den reellen, die Punkte der *y*-Achse, der *imaginären* Achse, den rein imaginären unter den komplexen Zahlen.

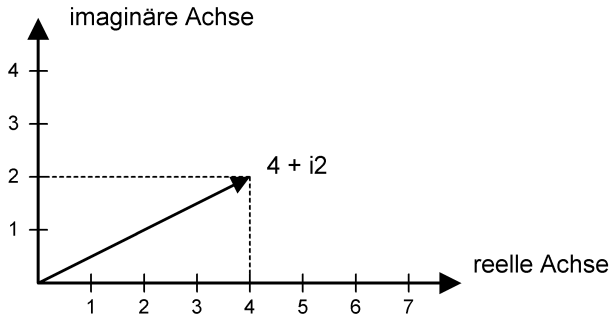

Abbildung 8.1. Jede komplexe Zahl entspricht einem Punkt in der Ebene.

In Abbildung 8.1 erkennen Sie die Darstellung der komplexen Zahl $4 + i2$ durch einen entsprechenden Ortsvektor in der komplexen Zahlenebene. Für das Rechnen mit komplexen Zahlen hat man die Grundrechenarten +, –, * und / definiert, sie lauten für zwei komplexe Zahlen $z_1 = x_1 + iy_1$ und $z_2 = x_2 + iy_2$ wie folgt:

Addition:	$z_1 + z_2 =$	$(x_1 + iy_1) + (x_2 + iy_2) = (x_1 + x_2) + i(y_1 + y_2)$
Subtraktion:	$z_1 - z_2 =$	$(x_1 + iy_1) - (x_2 + iy_2) = (x_1 + x_2) - i(y_1 + y_2)$
Multiplikation:	$z_1 * z_2 =$	$(x_1 + iy_1) * (x_2 + iy_2) = x_1x_2 + ix_1y_2 + ix_2y_1 - y_1y_2 =$
		$x_1x_2 - y_1y_2 + i(x_1y_2 + x_2y_1)$

Tabelle 8.1. Die Grundrechenarten +, - und * komplexer Zahlen.

Auf die Beschreibung der Division verzichten wir, da wir sie für die noch folgenden Beispiele in diesem Buch nicht benötigen. Neben diesen elementaren Rechenoperationen gibt es noch den Betrag einer komplexen Zahl. Darunter verstehen wir in der komplexen Zahlenebene den Abstand einer komplexen Zahl zum Ursprung. Nach dem Satz von Pythagoras gilt für das Betragsquadrat einer komplexen Zahl $z = x + iy$

$|z|^2 = x^2 + y^2$.

Mit der konjugiert komplexen Zahl schließen wir diesen Ausflug in die Theorie komplexer Zahlen ab, in Bezug auf eine komplexe Zahl $z = x + iy$ besitzt sie den Wert $x - iy$.

Erstellen Sie eine Klasse `Complex` unter Berücksichtigung von Tabelle 8.2:

Element	Beschreibung

Konstruktoren	`Complex ();`
	`Complex (double, double);`
	`Complex (double);`
	
	Standard- und benutzerdefinierter Konstruktor. Der dritte Konstruktor ist ein Typkonvertierungskonstruktor.

`operator+`
`operator-`
`operator*`
`operator/`
	`friend Complex operator+ (const Complex&, const Complex&);`
	`friend Complex operator- (const Complex&, const Complex&);`
	`friend Complex operator* (const Complex&, const Complex&);`
	`friend Complex operator/ (const Complex&, const Complex&);`
	
	Addition, Subtraktion, Multiplikation und Division zweier komplexer Zahlen.

`operator-`	`friend Complex operator- (const Complex&);` Negation einer komplexen Zahl (unäres Minus).

`operator==`
`operator!=`
	`friend bool operator== (const Complex&, const Complex&);`
	`friend bool operator!= (const Complex&, const Complex&);`
	
	Vergleichsoperatoren.

setter- und *getter*-Methoden	`void SetReal (double);`
	`double GetReal ();`
	`void SetImag (double);`
	`double GetImag ();`
	
	Lesender und schreibender Zugriff auf den Real- und Imaginärteil einer komplexen Zahl.

`Abs`	`double Abs();` Betrag einer komplexen Zahl.

`operator!`	`friend Complex operator! (const Complex&);` Liefert die konjugierte komplexe Zahl zurück.

Ein- und Ausgabe	`friend ostream& operator<< (ostream&, const Complex&);`
	`friend istream& operator>> (istream&, Complex&);`
	
	Gibt eine komplexe Zahl auf der Konsole im vektoriellen Format `[x,y]` aus bzw. liest eine komplexe Zahl von der Konsole ein.

*Tabelle 8.2. Konstruktoren, Methoden und Operatoren der Klasse **`Complex`**.*

8.3.2. Lösung

Die Lösung dieser Aufgabe entnehmen Sie bitte Listing 8.6 (Schnittstelle), Listing 8.7 (Implementierung) und Listing 8.8 (Testrahmen).

```
01: class Complex
02: {
03: private:
04:     double m_real;
05:     double m_imag;
06:
07: public:
08:     // c'tors
09:     Complex ();
10:     Complex (double, double);
11:
12:     // conversion constructor
13:     Complex (double);
14:
15:     // setter / getter methods
16:     void SetReal (double real) { m_real = real; }
17:     double GetReal () { return m_real; }
18:     void SetImag (double imag) { m_imag = imag; }
19:     double GetImag () { return m_imag; }
20:
21:     // unary arithmetic operators
22:     friend Complex operator- (const Complex&);
23:
24:     // binary arithmetic operators
25:     friend Complex operator+ (const Complex&, const Complex&);
26:     friend Complex operator- (const Complex&, const Complex&);
27:     friend Complex operator* (const Complex&, const Complex&);
28:     friend Complex operator/ (const Complex&, const Complex&);
29:
30:     // comparison operators
31:     friend bool operator== (const Complex&, const Complex&);
32:     friend bool operator!= (const Complex&, const Complex&);
33:
34:     // complex conjunction operator
35:     friend Complex operator! (const Complex&);
36:
37:     // input / output operators
38:     friend ostream& operator<< (ostream&, const Complex&);
39:     friend istream& operator>> (istream&, Complex&);
40: };
```

*Beispiel 8.6. Klasse **`Complex`**: Schnittstelle.*

```
01: // c'tors
02: Complex::Complex ()
03: {
04:     m_real = 0.0;
05:     m_imag = 0.0;
```

```
06: }
07:
08: Complex::Complex (double real, double imag)
09: {
10:     m_real = real;
11:     m_imag = imag;
12: }
13:
14: // conversion constructor
15: Complex::Complex (double real)
16: {
17:     m_real = real;
18:     m_imag = 0.0;
19: }
20:
21: // unary arithmetic operators
22: Complex operator- (const Complex& c)
23: {
24:     return Complex (c.m_real, -c.m_imag);
25: }
26:
27: // binary arithmetic operators
28: Complex operator+ (const Complex& c1, const Complex& c2)
29: {
30:     Complex tmp = Complex (c1);
31:     tmp.m_real += c2.m_real;
32:     tmp.m_imag += c2.m_imag;
33:     return tmp;
34: }
35:
36: Complex operator- (const Complex& c1, const Complex& c2)
37: {
38:     Complex tmp = Complex (c1);
39:     tmp.m_real -= c2.m_real;
40:     tmp.m_imag -= c2.m_imag;
41:     return tmp;
42: }
43:
44: Complex operator* (const Complex& c1, const Complex& c2)
45: {
46:     double real = c1.m_real * c2.m_real - c1.m_imag * c2.m_imag;
47:     double imag = c1.m_real * c2.m_imag + c1.m_imag * c2.m_real;
48:     return Complex (real, imag);
49: }
50:
51: Complex operator/ (const Complex& c1, const Complex& c2)
52: {
53:     double real =
54:         (c2.m_real * c1.m_real + c2.m_imag * c1.m_imag) /
55:         (c2.m_imag * c2.m_imag + c1.m_imag * c1.m_imag);
56:
57:     double imag =
58:         (c2.m_real * c1.m_real - c2.m_imag * c1.m_imag) /
59:         (c2.m_imag * c2.m_imag + c1.m_imag * c1.m_imag);
60:
61:     return Complex (real, imag);
62: }
63:
64: // comparison operators
65: bool operator== (const Complex& c1, const Complex& c2)
66: {
67:     if (c1.m_real == c2.m_real && c1.m_imag == c2.m_imag)
68:         return true;
69:
70:     return false;
71: }
72:
73: bool operator!= (const Complex& c1, const Complex& c2)
74: {
75:     return !(c1 == c2);
76: }
77:
78: // complex conjunction operator
79: Complex operator! (const Complex& c)
80: {
81:     return Complex (c.m_real, - c.m_imag);
82: }
83:
84: // output operator
85: ostream& operator<< (ostream& os, const Complex& c)
86: {
87:     char* sign = (c.m_imag >= 0) ? "+" : "";
88:     os << "[" << c.m_real << sign << c.m_imag << "*i]";
89:     return os;
```

```
90: }
91:
92: istream& operator>> (istream& is, Complex& c)
93: {
94:     is >> c.m_real >> c.m_imag;
95:     return is;
96: }
```

Beispiel 8.7. Klasse `Complex`: Implementierung.

```
01: void main ()
02: {
03:     // testing c'tors
04:     Complex c1 = Complex ();
05:     Complex c2 = Complex (3.0, 4.0);
06:     cout << "c1: " << c1 << endl;
07:     cout << "c2: " << c2 << endl;
08:
09:     // testing input operator
10:     cout << "enter complex number (e.g. \"1.5 3.5\"):" << endl;
11:     cin >> c1;
12:     cout << "c1: " << c1 << endl;
13:
14:     // testing setter and getter methods
15:     c2.SetReal (5);
16:     cout << "c2: " << c2 << endl;
17:     double imag = c2.GetImag ();
18:     cout << "imaginary part of c2: " << imag << endl;
19:
20:     // testing unary arithmetic operators
21:     Complex c3 = -c1;
22:     cout << "-c1: " << c3 << endl;
23:
24:     // testing binary arithmetic operators
25:     c3 = c1 + c2;
26:     cout << "c1 + c2: " << c3 << endl;
27:     c3 = c1 - c2;
28:     cout << "c1 - c2: " << c3 << endl;
29:     c3 = c1 * c2;
30:     cout << "c1 * c2: " << c3 << endl;
31:     c3 = c1 / c2;
32:     cout << "c1 / c2: " << c3 << endl;
33:
34:     // testing conversion constructor
35:     c3 = c1 + 5.0;
36:     cout << "c1 + 5.0: " << c3 << endl;
37:     c3 = 5 + c1;
38:     cout << "5 + c1: " << c3 << endl;
39:     c3 = 10.0;
40:     cout << "10.0: " << c3 << endl;
41:
42:     // testing conjunction operator
43:     cout << "c1: " << c1 << endl;
44:     c3 = !c1;
45:     cout << "!c1: " << c3 << endl;
46:
47:     // testing copy c'tor
48:     Complex c4 (c1);
49:     cout << "c4: " << c4 << endl;
50:
51:     // testing assignment operator
52:     c4 = c3;
53:     cout << "c4: " << c4 << endl;
54: }
```

Beispiel 8.8. Klasse `Complex`: Testrahmen.

Der Testrahmen aus Listing 8.8 führt bei der Ausführung zu folgenden Resultaten:

```
c1: [0+0*i]
c2: [3+4*i]
enter complex number (e.g. "1.5 3.5"):
1 2
c1: [1+2*i]
c2: [5+4*i]
```

```
imaginary part of c2: 4
-c1: [1-2*i]
c1 + c2: [6+6*i]
c1 - c2: [-4-2*i]
c1 * c2: [-3+14*i]
c1 / c2: [0.65-0.15*i]
c1 + 5.0: [6+2*i]
5 + c1: [6+2*i]
10.0: [10+0*i]
c1: [1+2*i]
!c1: [1-2*i]
c4: [1+2*i]
c4: [1-2*i]
```

8.4. Zeichenketten in C++: Die Klasse `string`

8.4.1. Aufgabe

In einer objektorientierten Sprache wie C++ gehört eine Klasse `String` für den komfortablen Umgang mit Zeichenketten zum Pflichtbestandteil der Klassenbibliothek eines jeden C++-Compilers. Da in C++ auch standardisierte Klassenbibliotheken existieren, finden wir die gesuchte Klasse – unter dem Namen `string` – tatsächlich auch in der STL („Standard Template Library“) vor. Um Übung in der Anwendung der Programmiersprache C++ zu erlangen, beschäftigen wir uns im folgenden mit der Realisierung unserer eigenen Klasse `String`, siehe dazu die Beschreibung der öffentlichen Schnittstelle dieser `String`-Klasse in Tabelle 8.3.

Bei der Implementierung der Klasse `String` ist der Speicherbereich für die einzelnen Zeichen der Zeichenkette dynamisch zu allokieren. Der Einfachheit halber legen wir zusätzlich zu Grunde, dass der Umfang dieses Speicherbereichs exakt an die Länge der Zeichenkette angepasst wird, siehe Abbildung 8.2. Den Overhead im Arbeitsaufwand der einzelnen Methoden nehmen wir in billigend Kauf, da wir mit dieser Vorschrift eine einfachere Realisierung verbuchen können.

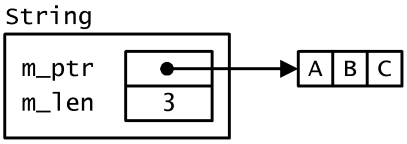

Abbildung 8.2. Instanzdatenbereich eines `String`-Objekts mit einem dynamisch allokierten Datenpuffer.

Element	Schnittstelle und Beschreibung
Konstruktor	`String ();` Standard-Konstruktor
Konstruktor	`String (const String& s);` Kopierkonstruktor
Konstruktor	`String (const char* cp);` Benutzerdefinierter Konstruktor. Als Argument wird eine Folge von `char`-Elementen erwartet, die mit `'\0'` abgeschlossen ist (so genannte „C-Zeichenkette“).
Destruktor	`~String ();` Destruktor
Methode `Length`	`int Length () const;` Liefert die Länge der Zeichenkette zurück.

Methode `Insert`	`bool Insert (const String& s, int ofs);` Fügt die Zeichenkette `s` in die aktuelle Zeichenketten-Instanz an der Position `ofs` ein.
Methode `Append`	`void Append (const String& s);` Hängt die Zeichenkette `s` am Ende des aktuellen Zeichenkettenobjekts an.
Methode `Remove`	`bool Remove (int ofs, int num)` Entfernt `num` Zeichen an der Position `ofs` des aktuellen Zeichenkettenobjekts.
Methode `SubString`	`String SubString (int ofs, int num)` Extrahiert eine Teilzeichenkette (beginnend an der Position Position `ofs` mit `num` Zeichen) aus dem aktuellen Zeichenkettenobjekt. Das Ergebnis wird in Gestalt eines `String`-Objekts zurückgeliefert.
Methode `Find`	`int Find (const String& s);` Sucht nach der Zeichenkette `s` im aktuellen Zeichenkettenobjekt. Wird die Zeichenkette gefunden, wird der Index (ihres ersten Vorkommens) innerhalb des aktuellen Zeichenkettenobjekts zurückgeliefert, andernfalls der Wert `-1`.
Methode `ToUpper`	`void ToUpper ();` Wandelt alle Kleinbuchstaben im aktuellen Zeichenkettenobjekt in Großbuchstaben um.
Methode `ToLower`	`void ToLower ();` Wandelt alle Großbuchstaben im aktuellen Zeichenkettenobjekt in Kleinbuchstaben um.
Methode `Left`	`String Left (int num);` Liefert die ersten `num` Zeichen des aktuellen Zeichenkettenobjekts in Gestalt eines eigenständigen Zeichenkettenobjekts zurück.
Methode `Right`	`String Right (int);` Liefert die letzten `num` Zeichen des aktuellen Zeichenkettenobjekts in Gestalt eines eigenständigen Zeichenkettenobjekts zurück.

Operator =	`String& operator= (const String& s);` Wertzuweisungsoperator
Operator []	`char operator[] (int index) const;` Indexoperator

Operator +
Operator +=
	`friend String operator+ (const String& s1, const String& s2);`
	`friend const String& operator+= (String& s1, const String& s2);`
	Verknüpfung von zwei Zeichenketten in Operatarenschreibweise als Alternative zur `Append`-Methode. Der +-Operator liefert als Resultatobjekt die Verkettung der zwei Zeichenketten `s1` und `s2` zurück, d.h. ihre Hintereinanderschreibung. Die Objekte `s1` und `s2` bleiben bei dieser Operation unverändert. Der +=-Operator hängt die Zeichenkette `s2` an `s1` an, das Ergebnis der Verkettung kommt folglich in Objekt `s1` zum Tragen.

Operator ==
Operator !=
	`friend bool operator== (const String& s1, const String& s2);`
	`friend bool operator!= (const String& s1, const String& s2);`
	Der ==-Operator vergleicht zwei Zeichenkettenobjekte auf inhaltliche Gleichheit, der !=-Operator auf ihre Ungleichheit.

Eingabeoperator <<	`friend ostream& operator<< (ostream& os, const String& s);` Eingabe einer Zeichenkette von der Konsole. In der Implementierung darf man eine maximale Anzahl für die einzulesenden Zeichen voraussetzen.
Ausgabeoperator >>	`friend istream& operator>> (istream& is, String& s);` Ausgabe einer Zeichenkette. Die Zeichenkette `"ABC"` sollte im Format `"ABC"[3]` ausgegeben werden, die Länge der Zeichenkette ist dabei in eckigen Klammern aufgeführt.

Tabelle 8.3. Öffentliche Elemente der Klasse `String`.

Die vorgestellte Implementierung der Klasse `String` verzichtet auf jegliche Unterstützung aus der *C Runtime Library* (*CRT*). Vor allem bietet es sich zu Lehrzwecken an, die Realisierung einer Klasse ohne jegliche Unterstützung durch eine externe Bibliothek zu betrachten. In der Praxis würde man zur Realisierung die *CRT* mit einbeziehen, um das Rad für eine Reihe elementarer Operationen wie Zeichenkette kopieren, vergleichen, usw. nicht zweimal erfinden zu müssen.

8.4.2. Lösung

In Listing 8.9 finden Sie die Schnittstelle der Klasse `String` vor, in Listing 8.10 ihre Implementierung und in Listing 8.11 einen dazu passenden Testrahmen:

```
01: class String
02: {
03: private:
04:     char* m_ptr; // buffer
05:     int m_len;   // buffer length
06:
07: public:
08:     // c'tors and d'tor
09:     String ();
10:     String (const char*);
11:     String (const String&);
12:     ~String ();
13:
14:     // public methods
15:     int Length () const;
16:     bool Insert (const String&, int);
17:     bool Remove (int, int);
18:     void Append (const String&);
19:     String SubString (int, int) const;
20:     int Find (const String&) const;
21:     void ToUpper ();
22:     void ToLower ();
23:     String Left (int) const;
24:     String Right (int) const;
25:
26:     // assignment operator
27:     String& operator= (const String&);
28:
29:     // subscript operator
30:     char& operator[] (int index);
31:
32:     // string concatenation
33:     friend String operator+ (const String&, const String&);
34:     friend const String& operator+= (String&, const String&);
35:
36:     // comparison operators
37:     friend bool operator== (const String&, const String&);
38:     friend bool operator!= (const String&, const String&);
39:
40:     // input/output
41:     friend ostream& operator<< (ostream&, const String&);
42:     friend istream& operator>> (istream&, String&);
43: };
```

Beispiel 8.9. Klasse `String`: Schnittstelle.

```
001: // c'tors and d'tor
002: String::String ()
003: {
004:     // empty string
005:     m_len = 0;
006:     m_ptr = (char*) 0;
007: }
008:
009: String::String (const String& s)
010: {
011:     // allocate buffer
012:     m_len = s.m_len;
013:     m_ptr = new char[m_len];
014:
015:     // copy object
016:     for (int i = 0; i < m_len; i ++)
017:         m_ptr[i] = s.m_ptr[i];
018: }
019:
020: String::String (const char* s)
021: {
022:     // length of string
023:     m_len = 0;
024:     while (s[m_len] != '\0')
025:         m_len ++;
026:
```

```
027:     // allocate buffer
028:     m_ptr = new char[m_len];
029:
030:     // copy string
031:     for (int i = 0; i < m_len; i ++)
032:         m_ptr[i] = s[i];
033: }
034:
035: String::~String ()
036: {
037:     delete[] m_ptr;
038: }
039:
040: // getter
041: int String::Length () const
042: {
043:     return m_len;
044: }
045:
046: char& String::operator[] (int index)
047: {
048:     // check parameter
049:     if (index < 0 || index >= m_len)
050:         throw out_of_range ("Wrong index");
051:
052:     return m_ptr[index];
053: }
054:
055: // public methods
056: bool String::Insert (const String& s, int ofs)
057: {
058:     if (ofs > m_len)
059:         return false;
060:
061:     // allocate new buffer
062:     char* tmp = new char[m_len + s.m_len];
063:
064:     for (int i = 0; i < ofs; i ++)      // copy first part
065:         tmp[i] = m_ptr[i];
066:     for (int i = 0; i < s.m_len; i ++)  // copy string to insert
067:         tmp[ofs + i] = s.m_ptr[i];
068:     for (int i = ofs; i < m_len; i ++)  // copy second part
069:         tmp[s.m_len + i] = m_ptr[i];
070:
071:     delete[] m_ptr;   // release old buffer
072:     m_ptr = tmp;      // switch buffer
073:     m_len += s.m_len; // adjust buffer length
074:
075:     return true;
076: }
077:
078: void String::Append (const String& s)
079: {
080:     Insert (s, m_len);
081: }
082:
083: bool String::Remove (int ofs, int num)
084: {
085:     if (ofs + num > m_len)
086:         return false;
087:
088:     // allocate new buffer
089:     char* tmp = new char[m_len - num];
090:
091:     // copy remaining parts
092:     for (int i = 0; i < ofs; i ++)            // first part
093:         tmp[i] = m_ptr[i];
094:     for (int i = ofs + num; i < m_len; i ++)  // second part
095:         tmp[i - num] = m_ptr[i];
096:
097:     delete[] m_ptr; // release old buffer
098:     m_ptr = tmp;    // switch buffer
099:     m_len -= num;   // adjust buffer length
100:
101:     return true;
102: }
103:
104: String String::SubString (int ofs, int num) const
105: {
106:     String empty;
107:     if (ofs < 0)
108:         return empty;
109:     if (ofs > m_len)
110:         return empty;
```

```
111:     if (ofs + num > m_len)
112:         return empty;
113:
114:     // allocate temporary buffer
115:     char* tmp = new char[num + 1];
116:
117:     // copy partial string
118:     for (int i = 0; i < num; i ++)
119:         tmp[i] = m_ptr[ofs + i];
120:     tmp[num] = '\0'; // terminate buffer
121:
122:     // create result object
123:     String s(tmp);
124:     delete[] tmp; // release temporary buffer
125:     return s;
126: }
127:
128: int String::Find (const String& s) const
129: {
130:     for (int i = 0; i < m_len - s.m_len + 1; i ++)
131:     {
132:         int k = 0;
133:         while (k < s.m_len)
134:         {
135:             if (m_ptr[i + k] != s.m_ptr[k])
136:                 break;
137:
138:              k ++;
139:         }
140:         if (k == s.m_len)
141:             return i;
142:     }
143:
144:     return -1;
145: }
146:
147: void String::ToUpper ()
148: {
149:     for (int i = 0; i < m_len; i ++)
150:         if (m_ptr[i] >= 'a' && m_ptr[i] <= 'z')
151:             m_ptr[i] -= ('a' - 'A');
152: }
153:
154: void String::ToLower ()
155: {
156:     for (int i = 0; i < m_len; i ++)
157:         if (m_ptr[i] >= 'A' && m_ptr[i] <= 'Z')
158:             m_ptr[i] += ('a' - 'A');
159: }
160:
161: String String::Left (int num) const
162: {
163:     return SubString (0, num);
164: }
165:
166: String String::Right (int num) const
167: {
168:     return SubString (Length() - num, num);
169: }
170:
171: // assignment operator
172: String& String::operator= (const String& s)
173: {
174:     if (this != &s)
175:     {
176:         // delete old string
177:         delete[] m_ptr;
178:
179:         // allocate new buffer
180:         m_len = s.m_len;
181:         m_ptr = new char[m_len];
182:
183:         // deep copy
184:         for (int i = 0; i < m_len; i ++)
185:             m_ptr[i] = s.m_ptr[i];
186:     }
187:
188:     return *this;
189: }
190:
191: // string concatenation
192: String operator+ (const String& s1, const String& s2)
193: {
194:     String s(s1);
```

```
195:     s.Append (s2);
196:     return s;
197: }
198:
199: const String& operator+= (String& s1, const String& s2)
200: {
201:     s1.Append (s2);
202:     return s1;
203: }
204:
205: // comparison operators
206: bool operator== (const String& s1, const String& s2)
207: {
208:     if (s1.m_len != s2.m_len)
209:         return false;
210:
211:     for (int i = 0; i < s1.m_len; i++)
212:         if (s1.m_ptr[i] != s2.m_ptr[i])
213:             return false;
214:
215:     return true;
216: }
217:
218: bool operator!= (const String& s1, const String& s2)
219: {
220:     return ! (s1 == s2);
221: }
222:
223: // output operator
224: ostream& operator<< (ostream& os, const String& s)
225: {
226:     os << '"';
227:     for (int i = 0; i < s.m_len; i++)
228:         os << s.m_ptr[i];
229:     os << "\"[" << s.m_len << ']';
230:
231:     return os;
232: }
233:
234: // input operator
235: istream& operator>> (istream& is, String& s)
236: {
237:     char line[256];
238:     is.getline(line, 256);
239:
240:     // calculate length of string
241:     int len = 0;
242:     while (line[len] != '\0')
243:         len ++;
244:
245:      // release old buffer, if any
246:     delete[] s.m_ptr;
247:
248:     // allocate new buffer and copy string
249:     s.m_ptr = new char[len];
250:     for (int i = 0; i < len; i ++)
251:         s.m_ptr[i] = line[i];
252:
253:     // adjust buffer length
254:     s.m_len = len;
255:
256:     return is;
257: }
```

Beispiel 8.10. Klasse `String`: Implementierung.

```
001: #include <iostream>
002: using namespace std;
003:
004: #include "String.h"
005:
006: void TestingCtorsDtor ()
007: {
008:     // testing c'tors
009:     String s1;
010:     cout << "s1: " << s1 << endl;
011:     String s2 ("12345");
012:     cout << "s2: " << s2 << endl;
013:     String s3 (s2);
014:     cout << "s3: " << s3 << endl;
```

```
015: }
016:
017: void TestingInsert ()
018: {
019:     String s1 ("12678");
020:     s1.Insert ("345", 2);
021:     cout << "s1: " << s1 << endl;
022:
023:     String s2 ("ABCD");
024:     s2.Insert (s1, 2);
025:     cout << "s2: " << s2 << endl;
026:
027:     s2.Insert ("!", 13);
028:     cout << "s2: " << s2 << endl;
029:
030:     s2.Insert (".", 12);
031:     cout << "s2: " << s2 << endl;
032:
033:     s2.Insert (".", 0);
034:     cout << "s2: " << s2 << endl;
035: }
036:
037: void TestingAppend ()
038: {
039:     String s1;
040:     s1.Append ("123");
041:     cout << "s1: " << s1 << endl;
042:
043:     String s2 ("ABC");
044:     s2.Append ("DEF");
045:     cout << "s2: " << s2 << endl;
046: }
047:
048: void TestingRemove ()
049: {
050:     String s1 ("12345");
051:     s1.Remove (1, 3);
052:     cout << "s1: " << s1 << endl;
053:
054:     String s2 ("ABC");
055:     s2.Remove (0, 3);
056:     cout << "s2: " << s2 << endl;
057: }
058:
059: void TestingSubString ()
060: {
061:     String s2 ("ABCDE");
062:     String s;
063:     s = s2.SubString (1, 3);
064:     cout << "s: " << s << endl;
065:     s = s2.SubString (0, 5);
066:     cout << "s: " << s << endl;
067:     s = s2.SubString (0, 6);
068:     cout << "s: " << s << endl;
069:     s = s2.SubString (0, 0);
070:     cout << "s: " << s << endl;
071: }
072:
073: void TestingOperators ()
074: {
075:     // testing '=' operator
076:     String s1;
077:     String s2 ("12345");
078:     s1 = s2;
079:     cout << "s1: " << s1 << endl;
080:
081:     // testing '==' operator
082:     cout << "s1 == s2: " << (s1 == s2) << endl;
083:     s1.Remove (0, 1);
084:     cout << "s1 == s2: " << (s1 == s2) << endl;
085:     cout << "s1 != s2: " << (s1 != s2) << endl;
086: }
087:
088: void TestingInput ()
089: {
090:     String s;
091:     cout << "Enter string: ";
092:     cin >> s;
093:     cout << "String: " << s << '.' << endl;
094: }
095:
096: void TestingLeftRight ()
097: {
098:     String s("12345");
```

```
099:     String s1;
100:     String s2;
101:
102:     s1 = s.Left (3);
103:     cout << "Left(3):  " << s1 << '.' << endl;
104:     s2 = s.Right(2);
105:     cout << "Right(2): " << s2 << '.' << endl;
106:
107:     s1 = s.Left (5);
108:     cout << "Left(5):  " << s1 << '.' << endl;
109:     s2 = s.Right(5);
110:     cout << "Right(5): " << s2 << '.' << endl;
111:
112:     s1 = s.Left (6);
113:     cout << "Left(6):  " << s1 << '.' << endl;
114:     s2 = s.Right(6);
115:     cout << "Right(6): " << s2 << '.' << endl;
116: }
117:
118: void TestingToUpperToLower ()
119: {
120:     String s("aBcDeFgHiJkLmNoPqRsTuVwXyZ");
121:     cout << "s:" << s << '.' << endl;
122:     s.ToUpper ();
123:     cout << "s:" << s << '.' << endl;
124:     s.ToLower ();
125:     cout << "s:" << s << '.' << endl;
126: }
127:
128: void TestingFind ()
129: {
130:     String s("ABCDEFGHIJKLMN");
131:     int i = s.Find ("IJK");
132:     cout << "i: " << i << endl;
133:
134:     i = s.Find ("ABCDEFGHIJKLMN");
135:     cout << "i: " << i << endl;
136:
137:     i = s.Find ("IJKZ");
138:     cout << "i: " << i << endl;
139:
140:     i = s.Find ("N");
141:     cout << "i: " << i << endl;
142:
143:     i = s.Find ("Z");
144:     cout << "i: " << i << endl;
145:
146:     i = s.Find ("");
147:     cout << "i: " << i << endl;
148: }
149:
150: void TestingSubsriptOperator ()
151: {
152:     String s("ABCDE");
153:     cout << "s[0]: " << s[0] << endl;
154:     cout << "s[4]: " << s[4] << endl;
155:
156:     s[0] = '<';
157:     s[4] = '>';
158:     cout << "s: " << s << endl;
159:
160:     try
161:     {
162:         s[5] = '!';
163:     }
164:     catch (out_of_range& e)
165:     {
166:         cout << e.what () << endl;
167:     }
168: }
169:
170: void TestingStringConcatenation ()
171: {
172:     String s1("123");
173:     String s2("ABC");
174:     String s3;
175:
176:     s3 = s1 + s2;
177:     cout << "s1:" << s1 << endl;
178:     cout << "s2:" << s2 << endl;
179:     cout << "s3:" << s3 << endl;
180:
181:     s1 += s2 += "789";
182:     cout << "s1:" << s1 << endl;
```

```
183:     cout << "s2:" << s2 << endl;
184: }
185:
186: void main ()
187: {
188:     TestingCtorsDtor ();
189:     TestingInsert ();
190:     TestingAppend ();
191:     TestingRemove ();
192:     TestingSubString ();
193:     TestingOperators ();
194:     TestingInput ();
195:     TestingLeftRight ();
196:     TestingToUpperToLower ();
197:     TestingFind ();
198:     TestingSubsriptOperator ();
199:     TestingStringConcatenation ();
200: }
```

Beispiel 8.11. Klasse `String`: Testrahmen.

Der Testrahmen aus Listing 8.11 führt bei der Ausführung zu folgenden Resultaten:

```
s1: ""[0]
s2: "12345"[5]
s3: "12345"[5]
s1: "12345678"[8]
s2: "AB12345678CD"[12]
s2: "AB12345678CD"[12]
s2: "AB12345678CD."[13]
s2: ".AB12345678CD."[14]
s1: "123"[3]
s2: "ABCDEF"[6]
s1: "15"[2]
s2: ""[0]
s: "BCD"[3]
s: "ABCDE"[5]
s: ""[0]
s: ""[0]
s1: "12345"[5]
s1 == s2: 1
s1 == s2: 0
s1 != s2: 1
Enter string: 123456
String: "123456"[6].
Left(3):  "123"[3].
Right(2): "45"[2].
Left(5):  "12345"[5].
Right(5): "12345"[5].
Left(6):  ""[0].
Right(6): ""[0].
s:"aBcDeFgHiJkLmNoPqRsTuVwXyZ"[26].
s:"ABCDEFGHIJKLMNOPQRSTUVWXYZ"[26].
s:"abcdefghijklmnopqrstuvwxyz"[26].
i: 8
i: 0
i: -1
i: 13
i: -1
i: 0
s[0]: A
s[4]: E
s: "<BCD>"[5]
Wrong index
s1:"123"[3]
s2:"ABC"[3]
s3:"123ABC"[6]
s1:"123ABC789"[9]
s2:"ABC789"[6]
```

8.5. Dynamische Datenstrukturen: Die verkettete Liste

8.5.1. Aufgabe

Rekursive Datentypen bilden gegenüber den statischen Datentypen wie Arrays und Strukturen das Fundament für Datenmengen, deren Umfang zur Laufzeit potentiell unendlich groß werden kann. Der bekannteste Vertreter dieser so genannten *dynamischen Datenstrukturen* ist die *verkettete Liste*. Der Name der Datenstruktur bezieht sich auf ihren internen Aufbau. Die einzelnen Listenelemente werden wie in einer Kette der Reihe nach aneinander gekettet. Pro Element der Liste ist neben den Nettodaten zusätzlich eine Zeigervariable erforderlich, um das nächste Element in der Liste zu identifizieren. Der Anwender einer verketteten Liste besitzt nur eine Adresse des ersten Listenelements, dieses wird in der Regel als `root` gekennzeichnet, siehe Abbildung 8.3.

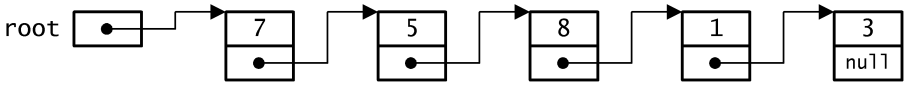

Abbildung 8.3. Schematische Darstellung einer verketteten Liste mit ganzzahligen Elementen.

Die Operationen einer verketteten Liste müssen ihrer internen Struktur Rechnung tragen. Um beispielsweise in einer Liste ein bestimmtes Element zu suchen, muss man die Liste Element für Element so lange traversieren, bis das Element gefunden wird. Diese Art des Zugriffs auf eine dynamische Datenstruktur ist natürlich nicht so laufzeitoptimal wie beispielsweise der indizierte Zugriff auf ein Array. Wir bezahlen diesen Preis für die Fähigkeit, eine unbestimmte Anzahl von Elementen elegant verwalten zu können.

Im folgenden betrachten wir die Realisierung einer verketteten Liste, die ganzzahlige Elemente aufnehmen kann. Ein einzelnes Element dieser Liste wird durch eine Instanz der Klasse `ListItem` gebildet:

```
01: class ListItem
02: {
03: friend class LinkedList;
04:
05: friend ostream& operator<< (ostream&, ListItem&);
06: friend ostream& operator<< (ostream&, LinkedList&);
07:
08: friend bool operator== (const LinkedList&, const LinkedList&);
09:
10: private:
11:     int m_val;
12:     ListItem * m_next;
13:
14: public:
15:     // c'tor
16:     ListItem (int);
17: };
18:
19: // c'tor
20: ListItem::ListItem (int val)
21: {
22:     m_val = val;
23:     m_next = (ListItem*) 0;
24: }
25:
26: ostream& operator<< (ostream& os, ListItem& li)
27: {
28:     os << li.m_val;
29:     return os;
30: }
```

Beispiel 8.12. Die Klasse `ListItem`: Einzelnes Element einer verketteten Liste.

Zum Arbeiten mit einer verketteten Liste benötigt man im einfachsten Fall Methoden zum Einfügen, Löschen und Suchen einzelner Elemente. Wir legen die Methoden aus Tabelle 8.4 als Grundlage

unserer Realisierung fest:

Methode	Beschreibung

Konstruktoren, Destruktor	`LinkedList ();`
	`LinkedList (const LinkedList&);`
	`~LinkedList ();`
	
	Standardkonstruktor, Kopierkonstruktor und Destruktor.

Methode `Size`	`int Size ();` Liefert die Anzahl der Elemente in der Liste zurück.
Methode `IsEmpty`	`bool IsEmpty ();` Liefert `true` zurück, wenn die Liste leer ist, andernfalls wird `false` zurückgegeben.
Methode `AddHead`	`bool AddHead (int val);` Fügt ein neues Element `val` am Anfang in der Liste ein.
Methode `AddTail`	`void AddTail (int val);` Fügt ein neues Element `val` am Ende der Liste ein.
Methode `Insert`	`bool Insert (int val, int pos);` Fügt ein neues Element `val` in der Liste an der durch `pos` spezifizierten Position ein. Der Index `pos` ist null-basiert.
Methode `Contains`	`bool Contains (int val);` Sucht das spezifizierte Element `val` in der Liste. Ist das Element nicht in der Liste enthalten, lautet der Rückgabewert `false`, andernfalls wird `true` zurückgegeben.
Methode `RemoveItemAtPosition`	`bool RemoveItemAtPosition (int pos);` Entfernt ein Element an der spezifizierten Position `pos` in der Liste. Spezifiziert `pos` eine ungültige Position, lautet der Rückgabewert `false`, andernfalls wird `true` zurückgegeben.
Methode `RemoveItem`	`bool RemoveItem (int val);`

	Entfernt ein Element `val` in der Liste. Ist das Element nicht in der Liste enthalten, lautet der Rückgabewert `false`, andernfalls wird `true` zurückgegeben.
Methode `RemoveAll`	`void RemoveAll ();` Entfernt alle in der Liste enthalten Elemente.

Tabelle 8.4. Methoden der Klasse ***`LinkedList`***.

Neben den Methoden zählen auch eine Reihe von Operatoren zu einem gelungenen Klassendesign, um die Anwenderschnittstelle der Klasse intuitiver zu gestalten. In Tabelle 8.5 finden Sie eine Beschreibung aller Operatoren der Klasse `LinkedList` vor:

Operator	Beschreibung
`+`	`friend LinkedList operator+ (const LinkedList& l1, const LinkedList& l2);` Konkatenation zweier verketteter Listen. Die durch das Anhängen von `l2` an `l1` entstehende verkettete Liste ist in einem neuem Objekt als Resultat zurückzuliefern.
`+=`	`friend LinkedList& operator+= (LinkedList& l1, const LinkedList& l2);` Operatorenschreibweise für die Methode `Concat`: Das Resultat kommt im ersten Parameter `l1` zu liegen.
`+=`	`friend LinkedList& operator+= (LinkedList& l, int val);` Operatorenschreibweise für die Methode `AddTail`: Der durch `val` spezifizierte `int`-Parameter ist am Ende der Liste `l` anzuhängen.
`-=`	`friend LinkedList& operator-= (LinkedList& l, int val);` Operatorenschreibweise für die Methode `RemoveItem`: Der durch `val` spezifizierte `int`-Parameter ist aus der Liste `l` zu entfernen.
`==`	`friend bool operator== (const LinkedList& l1, const LinkedList& l2);` Test zweier verketteter Listen auf Gleichheit. Zwei verkettete Liste sind genau dann gleich, wenn sie dieselbe Länge besitzen und an jeder Position dasselbe Datenelement vorhanden ist.
`==`	`friend bool operator!= (const LinkedList& l1, const LinkedList& l2);` Test zweier verketteter Listen auf Ungleichheit.

=	`LinkedList& operator= (const LinkedList& l);` Wertzuweisung zweier verketteter Listen.
<<	`ostream& operator<< (ostream&, LinkedList&);` Gibt eine verkettete Liste in der Form „`[1, 2, 3]`“ auf dem Ausgabestrom aus.

Tabelle 8.5. Operatoren der Klasse ***`LinkedList`***.

8.5.2. Lösung

Eine Umsetzung der Listenspezifikation aus Tabelle 8.4 und Tabelle 8.5 finden Sie in Listing 8.13 vor. Die Klasse `LinkedList` ist die Realisierung der Liste, ihre einzelnen Elemente sind vom Typ `ListItem`:

```
001: #include <iostream>
002: using namespace std;
003:
004: class LinkedList
005: {
006: friend ostream& operator<< (ostream&, LinkedList&);
007:
008: private:
009:     // member data
010:     ListItem * m_root;
011:     int m_count;
012:
013: public:
014:     // c'tors and d'tor
015:     LinkedList ();
016:     LinkedList (const LinkedList&);
017:     ~LinkedList ();
018:
019:     // public interface
020:     void AddHead (int);              // insert item at begin of list
021:     void AddTail (int);              // insert item at end of list
022:     bool Insert (int, int);          // insert item at a specified position
023:     bool RemoveItemAtPosition (int); // remove item at specified position
024:     bool RemoveItem (int);           // remove specified item
025:     void RemoveAll ();               // remove all items
026:     bool Contains (int);             // find item
027:     int  Size ();                    // retrieve length of linked list
028:     bool IsEmpty ();                 // is list empty
029:     void Concat (const LinkedList&); // concatenation of two lists
030:     void Reverse ();                 // reverse list
031:
032:     // additional operators
033:     friend LinkedList operator+ (const LinkedList&, const LinkedList&);
034:     friend LinkedList& operator+= (LinkedList&, const LinkedList&);
035:     friend LinkedList& operator+= (LinkedList&, int);
036:     friend LinkedList& operator-= (LinkedList&, int);
037:
038:     // comparison operators
039:     friend bool operator== (const LinkedList&, const LinkedList&);
040:     friend bool operator!= (const LinkedList&, const LinkedList&);
041:
042:     // assignment operator
043:     LinkedList& operator= (const LinkedList&);
044: };
045:
046: // c'tors and d'tor
047: LinkedList::LinkedList ()
048: {
049:     m_root = (ListItem*) 0;
050:     m_count = 0;
051: }
052:
053: LinkedList::LinkedList (const LinkedList& l)
```

```
054: {
055:     m_root = (ListItem*) 0;
056:     m_count = 0;
057:
058:     ListItem* item = l.m_root;
059:     while (item != (ListItem*) 0)
060:     {
061:         AddTail (item -> m_val);
062:         item = item -> m_next;
063:     }
064: }
065:
066: LinkedList::~LinkedList ()
067: {
068:     RemoveAll ();
069: }
070:
071: // public interface
072: void LinkedList::AddHead (int val)
073: {
074:     // create a new node
075:     ListItem* node = new ListItem (val);
076:
077:     if (m_root == (ListItem*) 0)
078:     {
079:         m_root = node;
080:     }
081:     else
082:     {
083:         node -> m_next = m_root;
084:         m_root = node;
085:     }
086:
087:     // increment node counter
088:     m_count ++;
089: }
090:
091: bool LinkedList::Insert (int val, int pos)
092: {
093:     // verify params
094:     if (pos < 0 || pos > m_count)
095:         return false;
096:
097:     // create a new node
098:     ListItem* node = new ListItem (val);
099:
100:     if (pos == 0)
101:     {
102:         node -> m_next = m_root;
103:         m_root = node;
104:     }
105:     else // i >= 1
106:     {
107:         ListItem* current = m_root;
108:         while (pos - 1 > 0)
109:         {
110:             current = current -> m_next;
111:             pos --;
112:         }
113:
114:         node -> m_next = current -> m_next;
115:         current -> m_next = node;
116:     }
117:
118:     // increment node counter
119:     m_count ++;
120:
121:     return true;
122: }
123:
124: void LinkedList::AddTail (int val)
125: {
126:     // create a new node
127:     ListItem* node = new ListItem (val);
128:
129:     if (m_root == (ListItem*) 0)
130:     {
131:         m_root = node;
132:     }
133:     else
134:     {
135:         // search end of list
136:         ListItem* last = m_root;
137:         while (last -> m_next != (ListItem*) 0)
```

```
138:             last = last -> m_next;
139:
140:         // append node
141:         last -> m_next = node;
142:     }
143:
144:     // increment node counter
145:     m_count ++;
146: }
147:
148: bool LinkedList::RemoveItemAtPosition (int pos)
149: {
150:     if (pos < 0 || pos >= m_count)
151:         return false;
152:
153:     if (pos == 0)
154:     {
155:         // store root pointer temporary
156:         ListItem* item = m_root;
157:
158:         // move pointer root to second element
159:         m_root = m_root -> m_next;
160:
161:         // delete first element
162:         delete item;
163:     }
164:     else
165:     {
166:         // move temporary pointer to predecessor of element to be removed
167:         ListItem* pred = m_root;
168:         for (int i = 0; i < pos - 1; i ++)
169:             pred = pred -> m_next;
170:
171:         // need pointer for later removal of list element
172:         ListItem* item = pred -> m_next;
173:
174:         // link predecessor and successor of questionable element together
175:         pred -> m_next = pred -> m_next -> m_next;
176:
177:         // now release list element
178:         delete item;
179:     }
180:
181:     // decrement node counter
182:     m_count --;
183:
184:     return true;
185: }
186:
187: bool LinkedList::RemoveItem (int val)
188: {
189:     // search specified item
190:     int pos = 0;
191:     ListItem* current = m_root;
192:     while (current != (ListItem*) 0)
193:     {
194:         if (current -> m_val == val)
195:             break;
196:
197:         pos ++;
198:         current = current -> m_next;
199:     }
200:
201:     // element not found
202:     if (current == (ListItem*) 0)
203:         return false;
204:
205:     RemoveItemAtPosition (pos);
206:     return true;
207: }
208:
209: void LinkedList::RemoveAll ()
210: {
211:     ListItem* item = m_root;
212:
213:     // delete each single element
214:     while (item != (ListItem*) 0)
215:     {
216:         // store current node pointer
217:         ListItem* current = item;
218:
219:         // advance to next node
220:         item = item -> m_next;
221:
```

```
222:         // delete 'current' node pointer
223:         delete current;
224:     }
225:
226:     // reset instance data of list
227:     m_root = (ListItem*) 0;
228:     m_count = 0;
229: }
230:
231: bool LinkedList::Contains (int val)
232: {
233:     ListItem* current = m_root;
234:     while (current != (ListItem*) 0)
235:     {
236:         if (current -> m_val == val)
237:             return true;
238:
239:         current = current -> m_next;
240:     }
241:
242:     // element not found
243:     return false;
244: }
245:
246: int LinkedList::Size ()
247: {
248:     return m_count;
249: }
250:
251: bool LinkedList::IsEmpty ()
252: {
253:     return m_count == 0;
254: }
255:
256: // assignment operator
257: LinkedList& LinkedList::operator= (const LinkedList& l)
258: {
259:     // delete current instance data of this object
260:     RemoveAll ();
261:
262:     // make a copy of the provided object
263:     m_root = (ListItem*) 0;
264:
265:     ListItem* item = l.m_root;
266:     while (item != (ListItem*) 0)
267:     {
268:         AddTail (item -> m_val);
269:         item = item -> m_next;
270:     }
271:
272:     return *this;
273: }
274:
275: void LinkedList::Concat (const LinkedList& l)
276: {
277:     // search last item to avoid recursion
278:     ListItem* item = l.m_root;
279:     ListItem* last;
280:     while (item != (ListItem*) 0)
281:     {
282:         last = item;
283:         item = item -> m_next;
284:     }
285:
286:     // append second list to current list
287:     item = l.m_root;
288:     while (item != (ListItem*) 0)
289:     {
290:         this -> AddTail (item -> m_val);
291:
292:         // end of original list reached
293:         if (item == last)
294:             break;
295:
296:         // advance to next node
297:         item = item -> m_next;
298:     }
299: }
300:
301: void LinkedList::Reverse ()
302: {
303:     // create a new root node
304:     ListItem* root = (ListItem*) 0;
305:
```

```
306:     // traverse current list and build reverse list
307:     ListItem* item = m_root;
308:     while (item != (ListItem*) 0)
309:     {
310:         ListItem* node = new ListItem (item -> m_val);
311:
312:         if (root == (ListItem*) 0)
313:         {
314:             root = node;
315:         }
316:         else
317:         {
318:             node -> m_next = root;
319:             root = node;
320:         }
321:
322:         item = item -> m_next;
323:     }
324:
325:     // release current list
326:     int count = m_count;
327:     RemoveAll ();
328:
329:     // switch to new list
330:     m_root = root;
331:     m_count = count;
332: }
333:
334: // additional operators
335: LinkedList operator+ (const LinkedList& l1, const LinkedList& l2)
336: {
337:     LinkedList item = l1;
338:     item.Concat (l2);
339:     return item;
340: }
341:
342: LinkedList& operator+= (LinkedList& l1, const LinkedList& l2)
343: {
344:     l1.Concat (l2);
345:     return l1;
346: }
347:
348: LinkedList& operator+= (LinkedList& l, int val)
349: {
350:     l.AddTail (val);
351:     return l;
352: }
353:
354: LinkedList& operator-= (LinkedList& l, int val)
355: {
356:     l.RemoveItem (val);
357:     return l;
358: }
359:
360: bool operator== (const LinkedList& l1, const LinkedList& l2)
361: {
362:     ListItem* item1 = l1.m_root;
363:     ListItem* item2 = l2.m_root;
364:
365:     while (item1 != (ListItem*) 0 && item2 != (ListItem*) 0)
366:     {
367:         if (item1 -> m_val != item2 -> m_val)
368:             return false;
369:
370:         item1 = item1 -> m_next;
371:         item2 = item2 -> m_next;
372:     }
373:
374:     if (item1 == (ListItem*) 0 && item2 == (ListItem*) 0)
375:         return true;
376:
377:     return false;
378: }
379:
380: bool operator!= (const LinkedList& l1, const LinkedList& l2)
381: {
382:     return ! (l1 == l2);
383: }
384:
385: // output operator
386: ostream& operator<< (ostream& o, LinkedList& l)
387: {
388:     ListItem* item = l.m_root;
389:
```

```
390:      o << "[";
391:      while (item != (ListItem*) 0)
392:      {
393:          if (item != l.m_root)
394:              o << ",";
395:
396:          // o << item -> m_val;
397:          o << *item;
398:          item = item -> m_next;
399:      }
400:      o << "] (" << l.m_count << " elements)";
401:
402:      return o;
403: }
```

Beispiel 8.13. Einfache Realisierung einer verketteten Liste für `int`-Variablen.

Die Algorithmen aus Listing 8.13 erfordern viel Sorgfalt im Umgang mit Zeigern. Für das Löschen eines Elements in der Liste ist zunächst der Vorgänger des zu löschenden `ListItem`-Objekts zu lokalisieren. Nun kann das `m_next`-Element des Vorgänger-Objekts mit der Adresse des Nachfolgeobjekts des zu löschenden Objekts überschrieben werden. Wir erkennen in Abbildung 8.4, dass das zu löschende `ListItem`-Objekt nach dem Umhängen des Zeigers in der Liste nicht mehr enthalten ist – wenngleich das `ListItem`-Objekt im Speicher als solches noch existiert. Es ist nun die Aufgabe des `delete`-Operators, diesen Speicherplatz wieder freizugeben.

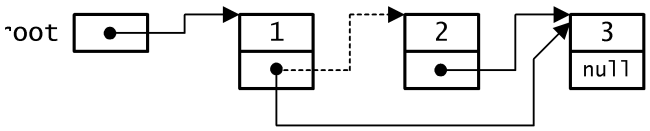

Abbildung 8.4. Das Löschen eines Listenelements (hier: `ListItem`-Objekt mit der Zahl 2) erfolgt durch das Umhängen einer Zeigervariablen.

Das Einfügen eines neuen Elements in eine verkettete Liste erfolgt nach einem ähnlichen Prinzip. Es ist wiederum zunächst die Position in der Liste zu lokalisieren, an der das neue Element einzufügen ist. Nun sind, wie in Abbildung 8.5 gezeigt wird, zwei Zeigervariablen anzupassen, und das Element ist in der Liste eingeklinkt.

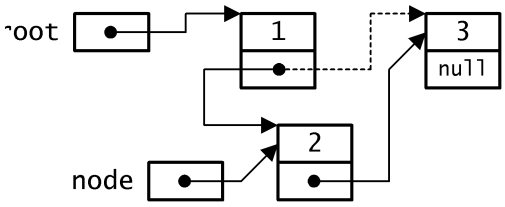

Abbildung 8.5. Das Einfügen eines neuen Listenelements (hier: `ListItem`-Objekt mit der Zahl 2) erfordert das Umhängen zweier Zeigervariablen.

Eine Methode aus Listing 8.13 bedarf noch einer zusätzlichen Erläuterung. Die Realisierung von `Concat` in den Zeilen 275 bis 299 sieht auf den ersten Blick vergleichsweise umständlich aus. Warum tut es eine einfachere Implementierung, etwa in der Gestalt

```
void LinkedList::Concat (const LinkedList& l)
{
    // append second list to current list
    ListItem* item = l.m_root;
    while (item != (ListItem*) 0)
    {
        this -> AddTail (item -> m_val);
        item = item -> m_next;
    }
}
```

nicht auch? Das Problem besteht einzig und allein bei einem Aufruf dieser Methode in der Form

```
LinkedList l;
...
l.Concat (l);
```

Erkennen Sie das Problem? Beim Anhängen einer Liste an sich selbst gerät die vorgestellte `Concat`-

Implementierung in eine Endlosschleife! Es wird zwar eine neue Liste, ausgehend von dem temporären Wurzelelement `item` aufgebaut. Die sukzessiven Aufrufe von `AddTail` am selben Listenobjekt führen aber dazu, dass dieses Objekt pro Iterationsschritt (in der `while`-Schleife) um ein Element länger wird und so das Ende-Kriterium niemals erreicht werden kann. In der Realisierung von Listing 8.13 wird deshalb vor dem Eintritt in die `while`-Schleife das letzte Listenelement ermittelt, um so einen zusätzlichen *Wächter* (*guard*) für das korrekte Ende der Wiederholungsschleife zu besitzen. Wir schließen diesen Abschnitt in Listing 8.14 mit einer kleinen Anwendung zum Testen der Realisierung ab:

```
01: void main ()
02: {
03:     LinkedList l;
04:
05:     // test insert
06:     for (int i = 0; i < 5; i ++)
07:         l.AddHead (2*i);
08:
09:     // test << operator
10:     cout << l << endl;
11:
12:     // test insert at a specified position
13:     l.Insert (98, 5);
14:     cout << l << endl;
15:
16:     // AddTail element
17:     l.AddTail (99);
18:     cout << l << endl;
19:
20:     // test find
21:     cout << "searching number 13: " << l.Contains (13) << endl;
22:     cout << "searching number 98: " << l.Contains (98) << endl;
23:
24:     // test remove
25:     l.RemoveItemAtPosition (6);
26:     cout << l << endl;
27:
28:     l.RemoveItemAtPosition (0);
29:     cout << l << endl;
30:
31:     // testing copy c'tor
32:     LinkedList l2 (l);
33:     cout << "l:  " << l << endl;
34:     cout << "l2: " << l2 << endl;
35:
36:     // testing assignment operator
37:     l2.AddHead (50);
38:     l2.AddHead (51);
39:     l2.AddHead (52);
40:     cout << "l2: " << l2 << endl;
41:
42:     l = l2;
43:     cout << "l:  " << l << endl;
44:
45:     l.RemoveAll ();
46:     cout << l << endl;
47: }
```

Beispiel 8.14. Einfacher Testrahmen für die Realisierung der verketteten Liste.

Wenn uns kein Fehler unterlaufen ist, sollte die `Main`-Methode aus Listing 8.14 die Ausgabe

```
[8,6,4,2,0] (5 elements)
[8,6,4,2,0,98] (6 elements)
[8,6,4,2,0,98,99] (7 elements)
searching number 13: 0
searching number 98: 1
[8,6,4,2,0,98] (6 elements)
[6,4,2,0,98] (5 elements)
l:  [6,4,2,0,98] (5 elements)
l2: [6,4,2,0,98] (5 elements)
l2: [52,51,50,6,4,2,0,98] (8 elements)
l:  [52,51,50,6,4,2,0,98] (8 elements)
[] (0 elements)
```

produzieren.