9

Überladen von Operatoren

Dieses Kapitel behandelt die folgenden Themen:

- Neue Bedeutungen für Operatorsymbole
- Gestaltung mathematischer und Index-Operatoren
- Unterschied zwischen ++x und x++ bei Neudefinition
- Smart Pointer
- Objekt als Funktion
- new und delete überladen / Eigene Speicherverwaltung

Den vordefinierten Operatorsymbolen in C++ kann man für Klassen neue Bedeutungen zuordnen. Damit können Operationen mit Objekten auf sehr einfache Weise ausgedrückt werden. Am bekannten Beispiel der rationalen Zahlen wird gezeigt, wie die arithmetischen Operatoren, die vorher nur für einige Grunddatentypen definiert waren, zur einfachen Formulierung von Rechenoperationen mit rationalen Zahlen benutzt werden. Um wie viel einfacher ist es, in einem Programm mit Matrizen `a`, `b` und `c`

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

zu schreiben, anstelle verschachtelter Schleifen unter genauer Beachtung der Randbedingungen wie Zeilen- und Spaltenzahlen! Um dieses Ziel zu erreichen, können in C++ Operatoren *überladen* werden, was nichts anderes bedeutet, als ihnen zusätzliche, problemabhängige Bedeutungen zu geben. Im obigen Fall würde der Operator `+` für eine Klasse `matrix` die Bedeutung bekommen, die Addition für alle Elemente der Matrix durchzuführen. In der C++-Standardbibliothek wird dieses Prinzip genutzt. Der Operator << ist

mehrfach überladen. Er dient nicht nur zum Verschieben von Bits, sondern wird in Verbindung mit `cout` zur Ausgabe von Daten verschiedenen Typs verwendet, sei es `char*`, `int`, `double` oder was sonst noch in Frage kommt.

Das Überladen von Operatoren funktioniert ähnlich wie das Überladen von Funktionen, wie es bereits in Abschnitt 3.2.5 (Seite 114) beschrieben wurde. Diesem Thema wurde dennoch ein eigenes Kapitel gewidmet, weil das Überladen von Operatoren im Unterschied zu Abschnitt 3.2.5 *Klassen voraussetzt*, die dort noch nicht behandelt wurden. Zudem zeigt die Möglichkeit des Überladens von Operatoren, dass darauf aufbauende Programme viel einfacher geschrieben werden können, wie das Beispiel am Kapitelanfang zeigt. Abbildung 9.1 zeigt die Syntax einer Operatordeklaration.

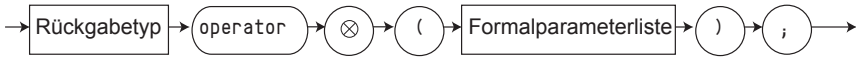


Abbildung 9.1: Syntaxdiagramm einer `operator`-Deklaration

⊗ steht für eines der möglichen C++-Operatorsymbole. Die Tabelle 9.1, auf die noch häufig Bezug genommen wird, zeigt, wie die Verwendung eines Operators vom Compiler in einen *Funktions*aufruf umgewandelt wird (⊗ sei das Operatorsymbol). Bestimmte Operatoren können nur als Elementfunktion auftreten, andere auch als globale Funktion.

Tabelle 9.1: Operator als Funktionsaufruf

Element-Funktion	Syntax	Ersetzung durch
nein	`x ⊗ y`	`operator⊗(x,y)`
	`⊗ x`	`operator⊗(x)`
	`x ⊗`	`operator⊗(x,0)`
ja	`x ⊗ y`	`x.operator⊗(y)`
	`⊗ x`	`x.operator⊗()`
	`x ⊗`	`x.operator⊗(0)`
	`x = y`	`x.operator=(y)`
	`x(y)`	`x.operator()(y)`
	`x[y]`	`x.operator[](y)`
	`x->`	`(x.operator->())->`
	`(T)x`	`x.operator T()`
	`static_cast<T>(x)`	`x.operator T()`
	`new T`	`T::operator new(sizeof(T))`
	`delete p`	`T::operator delete(p)`
	`new T[n]`	`T::operator new[](n * sizeof(T))`
	`delete[] p`	`T::operator delete[](p)`

`T` ist Platzhalter für einen Datentyp, `p` ist vom Typ `T*`.

Ein selbst definierter Operator *ist nichts als eine Funktion* in einer syntaktisch ansprechenderen Verkleidung! Einem in C++ vorhandenen Operator wird dabei eine neue, zusätzliche Bedeutung zugewiesen. Welche Bedeutung in einem Programmkontext die richtige ist, ermittelt der Compiler aus den Datentypen der Operanden. Eine Operatorfunktion unterscheidet sich von den bisher bekannten Funktionen nur in zweierlei Hinsicht:

1. Der Aufruf wird entsprechend der Tabelle 9.1 in einen Funktionsaufruf umgewandelt. Es macht keinen Unterschied, wenn man die Umwandlung selbst schon vornimmt und zum Beispiel `s = operator+(x,y)` statt `s = x + y` schreibt.
2. Der *Funktionsname* einer Operatorfunktion besteht aus dem Schlüsselwort `operator` und dem angehängten Operatorzeichen.

Aus Tabelle 9.1 wird ersichtlich, dass ein Überladen der unären Operatoren `++` und `--` eine Unterscheidung zwischen postfix- und präfix-Notation zulässt. Der Unterschied wird genauer auf Seite 334 beschrieben. Die Anzahl der Argumente ist um eins geringer, wenn der Operator eine Elementfunktion ist – unabhängig davon, ob es sich um einen unären oder binären Operator handelt. Sie ist um eins geringer, weil sich der Operator ohnehin schon auf ein Objekt bezieht. Es gibt einige Restriktionen für Operatorfunktionen:

- Es können die üblichen C++-Operatoren wie `=`, `==`, `+=` usw. überladen werden, nicht jedoch `.` `.*` `::` `?:` und andere Zeichen wie $ usw. Eine Definition von neuen Operatoren ist *nicht* möglich, auch nicht durch Kombination von Zeichen. Die Operatoren `new` und `delete` sowie `new []` und `delete []` können überladen werden.
- Die vorgegebenen Vorrangregeln können nicht verändert werden.
- Wenigstens ein Argument der Operatorfunktion muss ein Klassen-Objekt sein oder die Operatorfunktion muss eine Elementfunktion sein. Auf diese Weise wird sichergestellt, dass niemand die vorgegebene Bedeutung von Operatoren für Grunddatentypen umdefinieren kann.

9.1 Rationale Zahlen – noch einmal

In diesem Abschnitt sollen Operatoren für rationale Zahlen überladen werden, wobei auf das Beispiel in Abschnitt 4.4 (siehe Seite 162 und folgende) Bezug genommen wird. Ziel ist es, dass man anstelle des Aufrufs der Funktionen `add()`, `eingabe()` und `ausgabe()` direkt zum Beispiel

```
Rational a,b,c;
cin >> a;                    // überladener Operator
cin >> b;                    // überladener Operator
c = a + b;                   // überladener Operator
cout << c;                   // überladener Operator
```

schreiben kann. Ähnliches gilt für die anderen arithmetischen Funktionen.

9.1.1 Arithmetische Operatoren

Weil der binäre '+'-Operator symmetrisch ist und die Argumente des Operators selbst nicht verändert werden sollen, empfiehlt sich ein überladener Additionsoperator, der *keine* Elementfunktion der Klasse `Rational` ist. Eine gemischte Verwendung der Datentypen `long` und `Rational` soll natürlich möglich sein. Es gibt zwei Möglichkeiten in Abhängigkeit vom Vorhandensein des Typumwandlungskonstruktors (siehe die Diskussion in Abschnitt 4.4).

Zunächst muss überlegt werden, welche der möglichen Operatorvarianten aus Tabelle 9.1 ausgewählt werden sollte. Die Addition ist ein binärer Operator, sodass nur die Zeilen eins und vier der Tabelle in Frage kommen. Die Addition ist kommutativ, das heißt, $x + y = y + x$, also eine symmetrische Operation. Zu einer rationalen Zahl sollte auch eine ganze Zahl addiert werden können, die ja nur ein Spezialfall einer rationalen Zahl mit dem Nenner 1 ist. Damit lassen sich drei Fälle beschreiben:

1. Addition zweier rationaler Zahlen: $z = x + y$
   Umformulierung entsprechend Tabelle 9.1:
   A) `z = x.operator+(y)`
   B) `z = operator+(x,y)`
   Der Operator muss ein Objekt vom Typ `Rational` zurückgeben, welches `z` zugewiesen wird. Die Objekte `x` und `y` sollen nicht verändert werden. Lösung A) ist nicht zu empfehlen, weil sie auf den ersten Blick eine Änderung von `x` suggeriert.
2. Addition einer rationalen und einer ganzen Zahl, zum Beispiel $z = x + 3$
   Umformulierung entsprechend Tabelle 9.1:
   A) `z = x.operator+(3)`
   B) `z = operator+(x,3)`
   Beide Lösungen sind zunächst denkbar. Lösung A) ist aus demselben Grund wie oben abzulehnen.
3. Addition einer ganzen und einer rationalen Zahl, zum Beispiel $z = 3 + y$
   Umformulierung entsprechend Tabelle 9.1:
   A) `z = 3.operator+(y); // ?`
   B) `z = operator+(3,y);`
   Lösung A) ist ausgeschlossen, weil der selbst definierte Operator keine Elementfunktion einer ganzen Zahl sein kann! In C++ ist der Grunddatentyp `int` nicht als Klasse implementiert, und selbst wenn es so wäre, könnten wir nicht nachträglich eine neue Elementfunktion hinzufügen.

Aus dem Vergleich ergibt sich, dass die Lösung B) die einzig mögliche ist, wenn man die Symmetrie in der Schreibweise bei verschiedenen Datentypen erhalten will. Wenn entsprechend B) die Operatorfunktion `operator+()` *nicht* als Elementfunktion der Klasse `Rational` definiert werden soll, muss noch das Problem gelöst werden, dass die Operatorfunktion lesend auf die internen, privaten Daten von `Rational`-Objekten, also Zähler und Nenner, zugreifen darf. Wir benutzen die Methoden `getZaehler()` und `getNenner()`, die wegen ihrere inline-Eigenschaft schnell sind. Eine Alternative ist die Nutzung der Kurzformoperatoren, ähnlich wie schon auf Seite 167 gezeigt. Die folgende Zeile zeigt die Lösung zu Fall 3. B). Die Begründung für den `const`-Spezifizierer am Anfang der Zeile finden Sie als Hinweis auf Seite 165

```
const Rational operator+(const Rational&, const Rational&);
```

Ohne Typumwandlungskonstruktor

Um gemischte Datentypen zu erlauben, fügen wir zwei Deklarationen hinzu, mit denen Operationen wie $z = x + 3$ oder $z = 3 + y$ bearbeitet werden können:

```
const Rational operator+(long, const Rational&);
const Rational operator+(const Rational&, long);
```

Die Implementierung könnte wie folgt aussehen:

```
const Rational operator+(const Rational& a, const Rational& b) {
    return Rational(a.getZaehler()*b.getNenner() +
     b.getZaehler()*a.getNenner(), a.getNenner()*b.getNenner());
}

const Rational operator+(long a, const Rational& b) {
    Rational t(a, 1);  // t wird zu (a/1)
    return t + b;      // Aufruf von +(Rational, Rational)
}

const Rational operator+(const Rational& a, long b) {
    return b+a;        // Aufruf von +(long, Rational)
                       // durch vertauschte Reihenfolge
}
```

Hinweis

Bei der Rückgabe temporärer Objekte wird der Kopierkonstruktor aufgerufen – wenn der Compiler den Aufruf nicht wegoptimiert. Ab Seite 589 wird auf dieses Problem und mögliche Lösungen besonders eingegangen.

Mit Typumwandlungskonstruktor

Wenn ein Typumwandlungskonstruktor wie in Abschnitt 4.3.4 oder im Beispiel auf Seite 322 vorhanden ist, könnten alle Prototypen und Implementationen des Operators entfallen, die einen `long`-Parameter enthalten.

Implementierung mit Ausnutzen der Kurzformoperatoren

Mit Hilfe der Kurzformoperatoren lassen sich binäre Operatoren deutlich kürzer formulieren. Dazu muss zunächst `Rational operator+=(const Rational &)` in der Klassendeklaration nachgetragen werden. Die Definition dieses Operators sei dem Leser überlassen (siehe Übungsaufgaben Seite 323). `operator+()` kann dann leicht mithilfe von `+=` implementiert werden. Weil `operator+()` nicht auf private Daten eines `Rational`-Arguments zugreift, muss `operator+()` weder Element- noch `friend`-Funktion sein.

```
const Rational operator+(const Rational& a, const Rational& b) {
    Rational temp = a;
    return temp += b;      // Rückgabe von temp.operator+=(b)
}
```

oder noch kürzer durch impliziten Aufruf des Kopierkonstruktors

```
const Rational operator+(const Rational& a, const Rational& b) {
    return Rational(a) += b;
}
```

Der `+`-Operator ist in allen vorgestellten Formen sehr schnell, sofern der Compiler den Aufruf des Kopierkonstruktors bei der Rückgabe eliminiert, wie in Abschnitt 4.3.3, Seite 159, beschrieben.

9.1.2 Ausgabeoperator <<

Um eine rationale Zahl `r` mit einer einfachen Anweisung `cout << r;` ausgeben zu können, wird der Operator `<<` überladen. Aus der Syntax der Anweisung und der Tabelle 9.1 auf Seite 318 sehen wir, dass die Anweisung entweder als a) `cout.operator<<(r);` interpretiert werden würde oder b) als `operator<<(cout, r);`. Der Operator kann also keine Elementfunktion der Klasse `Rational` sein, sondern der Interpretation a) folgend höchstens eine der Klasse `ostream`, der `cout` angehört. Aber diese Möglichkeit kommt nicht in Frage, weil die Konstrukteure der Klasse `ostream` nicht jede mögliche Ausgabe einer benutzerdefinierten Klasse vorhersehen und einbauen konnten. Daher bleibt nur die Interpretation b), woraus sich ergibt, dass der Operator `<<` als globale Funktion mit zwei Argumenten formuliert werden muss:

```
std::ostream& operator<<(std::ostream& s, const Rational&);
```

Der Operator gibt eine Referenz auf das `ostream`-Objekt `s` zurück, sodass für das Ergebnis der Ausgabe wiederum `s` eingesetzt gedacht werden kann und eine Verkettung mehrerer Ausgaben möglich wird:

```
std::cout << "r = " << r << "r1= " << r1;
```

ist dann identisch mit

```
(((std::cout << "r = ") << r) << "r1= ") << r1;
```

Die Implementierung des Ausgabeoperators ähnelt der vorher benutzten Elementfunktion `ausgabe()`:

```
ostream& operator<<(ostream& ausgabe, const Rational& r) {
   ausgabe << r.getZaehler() << '/' << r.getNenner();
   return ausgabe;
}
```

Zum Abschluss wird die Definition der Klasse mit den bisher besprochenen Änderungen gezeigt, jedoch ohne die Implementierung der Methoden und Funktionen (siehe oben sowie Übungsaufgaben). Durch Operatoren ersetzte Elementfunktionen wie `add()` usw. wurden gestrichen.

Listing 9.1: Klasse Rational mit überladenen Operatoren

```
// cppbuch/k9/rational/ratioop.h
#ifndef RATIOOP_H
#define RATIOOP_H
#include<iostream>

class Rational {
  public:
    Rational() : zaehler(1), nenner(1) {}       // inline
    Rational(long z, long n);
    Rational(long);     // Typumwandlungskonstruktor

    // Abfragen inline
    long getZaehler() const { return zaehler;}
    long getNenner() const { return nenner;}
```

```
    // arithmetische Methoden
    Rational& operator+=(const Rational&);
    // weitere arithmetische Methoden weggelassen ...

    // weitere Methoden
    void set(long zaehler, long nenner);
    void kehrwert();
    void kuerzen();

  private:
    long zaehler, nenner;
};

// globaler Additionsoperator (andere Operatoren siehe Übungsaufgabe)
const Rational operator+(const Rational&, const Rational&);

// globale Funktionen zur Ein- und Ausgabe
std::ostream& operator<<(std::ostream&, const Rational&);
std::istream& operator>>(std::istream&, Rational&);
#endif        // RATIOOP_H
```

Übungen

9.1 Implementieren Sie den Operator >> zur Eingabe von Objekten der Klasse `Rational`.

9.2 Implementieren Sie den Operator `+=` für rationale Zahlen. Überlegen Sie vorher, ob der Operator als Element- oder als `friend`-Funktion realisiert werden sollte. Der Rückgabetyp könnte bei allein stehenden Operationen `void` sein. Aus dem Programmcode innerhalb `operator+()` auf Seite 321 ergibt sich aber ein Rückgabetyp `Rational` oder `Rational&`, der in der Klassendefinition oben verwendet wird. Welcher ist sinnvoll und warum?

9.3 Ersetzen Sie im Beispiel des Abschnitts 4.4 auch die Funktionen `sub()`, `mult()` und `div()` durch überladene Operatoren. Testen nicht vergessen!

9.4 Implementieren Sie den Operator `==` zum Vergleich zweier Rationalzahlen.

9.2 Eine Klasse für Vektoren

In den folgenden Abschnitten wird eine Klasse `Vektor` definiert, die ein eindimensionales Array wählbarer Größe bildet. Die Klasse soll zeigen, wie die Klasse `vector` der C++-Standardbibliothek im Prinzip arbeitet. Die Unterschiede zu einem C-Array sind:

- Der Arrayzugriff über den Index-Operator `[ ]` ist *sicher* (im Gegensatz zur Klasse `vector` der C++-Standardbibliothek, die geprüfte Zugriffe über die Methode `at()` anbietet). Eine Indexüberschreitung wird zur Laufzeit als Fehler gemeldet, es wird keine undefinierte Adresse angesprungen.

- Es wird ein Zuweisungsoperator definiert, sodass `v1 = v2;` geschrieben werden kann mit dem Ergebnis `v1[i] == v2[i]` für alle `i` im Bereich 0 bis (Länge von `v2` - 1).
- Die Klasse kann leicht um weitere Funktionen erweitert werden wie Bildung des Skalarprodukts, Addition zweier Vektoren und andere. Hier wird gezeigt, wie die Größe eines Vektors dynamisch, also zur Laufzeit, geändert werden kann. Er verhält sich dann wie ein dynamisches Array.

Die `private` Elemente der Klasse sind ein Zeiger `start` auf den Beginn eines Arrays der Größe `xDim`, das aus Objekten des Typs `T` besteht (Abbildung 9.2).

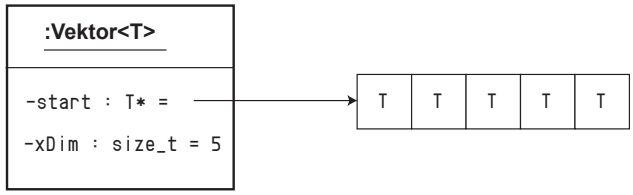


Abbildung 9.2: Ein Objekt der Klasse `Vektor<T>`

Listing 9.2: Template-Klasse Vektor

```
// cppbuch/k9/vektor/vektor.t
// dynamische Vektor-Klasse
#ifndef VEKTOR_T
#define VEKTOR_T
#include<stdexcept>

template<typename T>
class Vektor {
  public:
    Vektor(size_t x = 0);              // Allg. Konstruktor
    Vektor(size_t n, T t);             // Allg. Konstruktor 2:
                                         // n Elemente mit Wert t
    Vektor(const Vektor<T>& v);        // Kopierkonstruktor
    virtual ~Vektor() { delete [] start;} // Destruktor

    size_t size() const {return xDim;}
    void groesseAendern(size_t);       // dynamisch ändern

    // Indexoperator inline
    T& operator[](int index) {
        if( index < 0 || index >= (int)xDim) {
           throw std::out_of_range("Indexüberschreitung!");
        }
        return start[index];
    }
    // Indexoperator für nicht-veränderliche Vektoren
    const T& operator[](int index) const {
        if( index < 0 || index >= (int)xDim) {
           throw std::out_of_range("Indexüberschreitung!");
        }
```

```
        return start[index];
    }

    void swap(Vektor<T>& v); // Vektoren vertauschen
    // Zuweisungsoperator
    Vektor<T>& operator=(Vektor<T>); // Übergabe per Wert (siehe Impl. unten)

    // Zeiger auf Anfang und Position nach dem Ende für Vektoren
    // mit nichtkonstanten und konstanten Elementen
    T* begin() { return start;}
    T* end()  { return start + xDim;}
    const T* begin() const { return start;}
    const T* end() const { return start + xDim;}
  private:
    size_t xDim;                    // Anzahl der Datenobjekte
    T   *start;                     // Zeiger auf Datenobjekte
}; // #endif siehe Seite 330
```

Die Elemente des Arrays können beliebige kopierbare Objekte desselben Datentyps sein, zum Beispiel `Rational`-Objekte. Die Klassendeklaration enthält auch die Dinge, die erst in den späteren Abschnitten über die verschiedenen Operatoren benötigt werden.

Es ist unmittelbar einsichtig, dass Programme mit Vektoren bei Vorliegen der oben beschriebenen Funktionalität lesbarer und kürzer als mit C-Arrays geschrieben werden können – weswegen in diesem Buch von Anfang an Standardvektoren verwendet werden. Das Prinzip gilt natürlich nicht nur für Vektoren. Die anwendungsbezogene C++-Programmierung wird darauf hinauslaufen, eine Vielfalt vorgefertigter Klassen aus kommerziell erhältlichen und firmenspezifischen Bibliotheken einzusetzen, um die Produktivität der Programmierer und die Qualität ihrer Produkte zu erhöhen.

Die angegebenen Elementfunktionen und Operatoren werden nachfolgend erläutert. Es folgen die Implementationen der Konstruktoren:

```
template<typename T>
Vektor<T>::Vektor(size_t x)         // Allg. Konstruktor
: xDim(x), start(new T[x]) {
}
```

Der Konstruktor erzeugt zum Beispiel durch `Vektor<Rational> vRat(3);` einen `Rational`-Vektor `vRat` mit drei Elementen. Weil im Konstruktor mit `new` ein Array angelegt wird, muss im Destruktor `delete` mit eckigen Klammern angegeben werden. Der Kopier- oder Initialisierkonstruktor kopiert erst die Größe des Vektors, beschafft die benötigte Menge Speicherplatz und überträgt dann einzeln die Elemente:

```
template<typename T>
Vektor<T>::Vektor(const Vektor<T>& v) // Kopierkonstruktor
: xDim(v.xDim), start(new T[xDim]) {
    for(size_t i = 0; i < xDim; ++i) {
        start[i] = v.start[i];
    }
}
```

Man könnte fragen, ob nicht die C-Funktion `memcpy()` von Seite 881 das Kopieren des Vektors schneller als die gezeigte Schleife erledigen könnte, da der Speicherplatz für

das Array `start` zusammenhängend angelegt wird. Dies würde jedoch nur funktionieren, wenn der Typ `T` keine dynamischen Daten enthält, weil es sich um eine »flache« Kopie handelt (siehe auch Abbildung 6.2 auf Seite 236). Im Template ist aber nicht bekannt, ob es nur für solche Typen instanziiert werden wird. Der in Abschnitt 9.2.2 beschriebene Zuweisungsoperator stellt hingegen sicher, dass die Anweisung `start[i] = v.start[i];` eine vollständige Kopie des Vektor-Elements bewirkt.

Der vom System bereitgestellte Kopierkonstruktor wäre nicht ausreichend, weil ein Kopieren von `start` und `xDim` nicht genügt. Der Konstruktor zur Initialisierung des Vektors mit einem bestimmten Wert enthält eine einfache Schleife, die jedem Element des Vektors den Initialisierungswert zuweist:

```
template<typename T>
Vektor<T>::Vektor(size_t n, T wert)
: xDim(n), start(new T[n]) {
    for (size_t i = 0; i < xDim; ++i) {
        start[i] = wert;
    }
}
```

Die Methode `begin()` gibt einen Zeiger auf den Anfang des Vektors zurück; `end()` gibt einen Zeiger auf die Position *nach dem letzten* Vektorelement zurück. Diese Methoden werden erst später benötigt (Abschnitt 11.2).

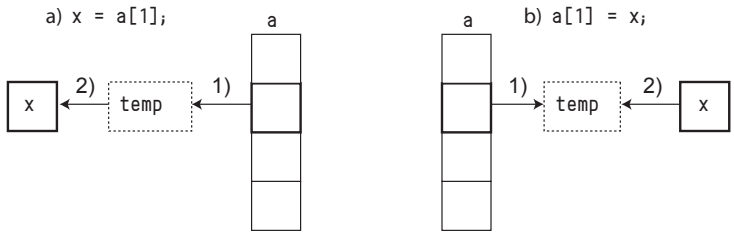
9.2.1 Index-Operator [ ]

Die Deklaration des Indexoperators zeigt, dass nicht ein Objekt, sondern eine *Referenz* auf ein Objekt des Datentyps `T` zurückgegeben wird. Warum? Gesucht ist eigentlich *ein Element* des Vektors. Betrachten wir dazu einfach die beiden Fälle (Rückgabe eines Objekts per Wert – Rückgabe einer Referenz) im Vergleich.

Falsch: Rückgabe per Wert

Die Deklaration würde dementsprechend lauten: `T operator[](int index);`. Da die Operatorfunktion eine Funktion wie jede andere ist, die nur einen speziellen Namen hat, gelten die gleichen Mechanismen wie sonst auch bei Funktionen. Das per Wert übergebene Argument `index` wird in die Funktion hineinkopiert, und die Anweisung `return irgendwas;` bewirkt, dass eine temporäre Kopie von `irgendwas` dem Aufrufer übergeben wird. Das Kopieren wird vom compilergenerierten Kopierkonstruktor oder einem für die Klasse `T` selbst geschriebenen übernommen. Abbildung 9.3 verdeutlicht den Vorgang. Wir müssen dabei zwei verschiedene Fälle unterscheiden, nämlich dass das anzusprechende Element auf der rechten oder auf der linken Seite (als L-Wert oder lvalue) einer Zuweisung stehen kann.

Betrachten wir den ersten Fall, in Abbildung 9.3 mit a) bezeichnet. Im ersten Schritt wird durch den Indexoperator eine temporäre Kopie des Elements Nr. 1 des Vektors `a` erzeugt. Diese Kopie wird der Variablen `x` zugewiesen, und anschließend wird die Kopie verworfen – genau wie wir es wünschen. Im zweiten Fall sieht es anders aus! Wie in Abbildung 9.3, Fall b), zu sehen ist, würde auch hier zunächst eine temporäre Kopie erzeugt. Die Variable `x` würde dann *dieser Kopie* zugewiesen werden, die danach »entsorgt« wird, *das ursprüngliche Vektorelement bliebe unverändert!* Damit wäre der Sinn der Anweisung



Reihenfolge: 1) Erzeugen einer temporären Kopie, 2) Zuweisung an x bzw. von x

Abbildung 9.3: Falsch: Zuweisung von Vektorelementen per Wert

`a[1] = x;` verfehlt. Glücklicherweise werden wir vom Compiler beim Auftreten dieser Anweisung darauf aufmerksam gemacht, dass der Operator nicht die ihm zugedachte Aufgabe erfüllt.

Richtig: Rückgabe per Referenz

Die Deklaration enthält nun den richtigen Rückgabetyp, sodass das richtige Objekt zurückgegeben wird:

```
T& operator[](int index);
```

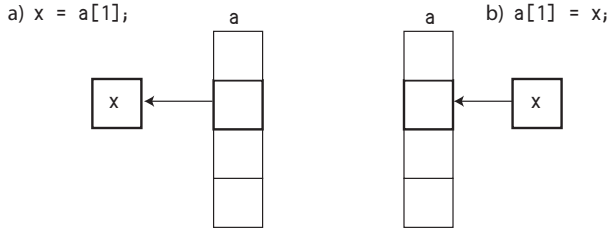


Abbildung 9.4: Richtig: Zuweisung von Vektorelementen per Referenz

Abbildung 9.4 zeigt in a) und b) die gleichen Fälle wie Abbildung 9.3, nur dass jetzt nicht mit einer temporären Kopie, sondern *direkt mit dem Original* gearbeitet wird. Eine Referenz ist ja nichts anderes als ein anderer Name, eine Alias-Bezeichnung, für ein und dasselbe Objekt. Der Kopierkonstruktor wird bei der Ergebnisrückgabe nicht aufgerufen, weil nichts zu kopieren ist und nur der Verweis auf das Original zurückgegeben wird. Das Ergebnis des Indexoperators ist nun das gewünschte Vektorelement und nicht nur eine temporäre Kopie.

Die Implementierung des Indexoperators wurde `inline` schon in der Klassendeklaration vorgenommen. Dabei wurde der Weg gewählt, dass das Programm sich mit einer Fehlermeldung beendet, wenn der zulässige Bereich für den Index über- oder unterschritten wird. Würden dem Aufrufer stattdessen ohne Abbruch irgendwelche Daten zurückgegeben, so arbeitete das Programm anschließend mit falschen Daten! Dadurch entstehende Folgefehler sind oft nur schwer auf die eigentliche Ursache der Indexüberschreitung zurückzuführen. Nach Auslieferung des Programms an einen Kunden sollte das Programm

allerdings *weder* abbrechen *noch* mit falschen Daten weiterrechnen (zur Fehlerbehandlung siehe Kapitel 8). Der zweite Indexoperator ist für den Zugriff auf konstante Objekte gedacht. Das erste `const` garantiert, dass ein per Referenz zurückgegebenes Element nicht verändert wird; das zweite `const` erlaubt die Anwendung des Operators auf konstante Vektor-Objekte ohne Beschwerden des Compilers. Wenn nun der Zugriff auf Arrayelemente ausschließlich über den Indexoperator geschieht, wird jede Bereichsüberschreitung gemeldet. Die nächsten Zeilen zeigen, wie der Operator in einem Programm eingesetzt wird.

```
Vektor<double> v(5);                    // Vektor mit 5 Elementen
v[0] = 1.00; v[1] = 1.234; v[2] = 2.22; // Elemente als Linkswerte
double x = v[2];                        // Element rechts

// Test auf Indexüberschreitung
v[99] = 400.2;  // Fehlermeldung zur Laufzeit!
x = v[100];     // Fehlermeldung zur Laufzeit!

// Definition und Initialisierung eines anderen Vektors
Vektor<double> vc = v;        // mit Kopierkonstruktor

// Element links und rechts
v[0] = vc[1];
```

Die Schreibweise unterscheidet sich in nichts von der Benutzung eines normalen Arrays. Nach außen hin ist die Wirkung die gleiche bis auf das sichere Verhalten bei Indexfehlern. Der sichere Zugriff kann bei dieser Vektor-Klasse mittels Zeigerarithmetik umgangen werden:

```
double *vp = &v[0];
*(vp+100) = 17.9;           // nicht entdeckter Indexfehler!
```

Gegen böswillige Manipulationen kann man sich ohnehin nicht vollständig schützen, aber die konsequente Benutzung eines wie oben definierten Indexoperators anstelle der Zeigerarithmetik trägt erheblich zur Sicherheit und Testbarkeit eines Programms bei.

9.2.2 Zuweisungsoperator =

Wenn ein spezieller Zuweisungsoperator nicht definiert ist, wird automatisch vom Compiler ein Zuweisungsoperator bereitgestellt (implizit generiert), der ein Objekt elementweise kopiert. Darunter ist zu verstehen, dass für jedes Element, das selbst wieder Objekt oder aber von einem Grunddatentyp sein kann, der zugehörige Zuweisungsoperator aufgerufen wird. Der Zuweisungsoperator für Grunddatentypen bewirkt eine bitweise Kopie. Falls Konstanten oder Referenzen vorhanden sind, geschieht dies jedoch nicht, weil sie nur initialisiert, aber nicht durch Zuweisung verändert werden könnten. In diesen Fällen ist ein besonderer '='-Operator notwendig. Im folgenden Beispiel sei die Existenz zweier Klassen `A` und `B` angenommen, die in der Klasse `C` benutzt werden:

```
class C : public A {
  public:
     // ...
  private:
    B einB;
```

```
    int x;
};
```

Eine Zuweisung der Art

```
C c1, c2;
c1 = c2; // d.h. c1.operator=(c2);
```

bewirkt implizit die folgenden Aufrufe, wobei keine Aussage darüber getroffen wird, ob der Zuweisungsoperator vom System generiert oder selbst geschrieben wurde:

```
c1.A::operator=(c2.A);    // Zuweisen des anonymen Subobjekts
c1.einB.operator=(c2.einB); // aggregiertes Objekt
c1.x = c2.x;              // Grunddatentyp
```

Wenn die Klassen `A` bzw. `B` selbst Elemente enthalten, gilt für diese dasselbe Prinzip.

Operatorfunktionen können wie andere Methoden vererbt werden. Die Ausnahme ist der Zuweisungsoperator '='. Der automatisch erzeugte Zuweisungsoperator würde ohnehin einen geerbten Zuweisungsoperator überschreiben. Aus diesem Grund ist ein Vererben des Zuweisungsoperators überflüssig. Typischerweise enthält eine abgeleitete Klasse zusätzliche Datenelemente. Die Restriktion sorgt dafür, dass nicht versehentlich die zusätzlichen Datenelemente beim Kopieren unter den Tisch fallen können.

Falls Zeiger als Elemente in einem Objekt vorhanden sind, werden diese ebenfalls kopiert. Dies kann zu Problemen führen, wenn die Zeiger auf dynamisch zugewiesene Speicherbereiche oder Objekte zeigen, weil dann nur Adressen kopiert werden (siehe dazu Abschnitt 4.3.3 und das Bild auf Seite 236). Wenn ein Objekt A einem Objekt B zugewiesen werden soll, ist folgende Reihenfolge einzuhalten:

1. Reservieren von Speicher für B in der Größe von Objekt A
2. Inhalt von A in den neuen Speicher kopieren
3. Freigabe des vorher belegten Speichers
4. Aktualisieren der Verwaltungsinformationen

Den 3. Schritt vor den ersten zu ziehen, um den gesamten Speicherbedarf zu reduzieren, wäre ein Fehler: Das Programm wäre nicht mehr exception-sicher (siehe dazu Abschnitt 8.3). Aber wenn wir den Swap-Trick von Seite 218 anwenden, können wir uns die Überlegungen über die Speicherbeschaffung und die Reihenfolge schenken: Der Zuweisungsoperator ist dann auf jeden Fall exception-sicher und auch viel einfacher zu schreiben. Er benutzt die Funktion `swap(Vektor<T>&)` der Klasse `Vektor`, die wiederum die Funktion `std::swap()` der Standardbibliothek nutzt. Sie vertauscht einfach nur ihre beiden Parameter.

```
// Vektoren vertauschen
template<typename T>
void Vektor<T>::swap(Vektor<T>& v) {
   std::swap(xDim, v.xDim);
   std::swap(start, v.start);
}
```

```
// Zuweisungsoperator
template<typename T>
Vektor<T>& Vektor<T>::operator=(Vektor<T> v) { // Übergabe per Wert! (siehe Text)
   swap(v);
   return *this;
}
```

Die Übergabe per Wert erspart eine Zeile zur Konstruktion eines temporären Objekts. Außerdem hat der Compiler so die Möglichkeit, den Kopierkonstruktor wegzuoptimieren – nämlich dann, wenn auf der rechten Seite der Zuweisung ein temporäres Objekt steht, etwa bei der Zuweisung eines Additionsergebnisses: `v1 = v2 + v3;`.

Damit können Vektoren mit *einem* Befehl im Programm zugewiesen werden, zum Beispiel `v2 = v`. Bisher bestehende Referenzen auf Elemente des linksseitigen Vektors werden bedeutungslos, weil der Adressbereich des Vektors nach der `new`-Operation im Kopierkonstruktor ganz woanders liegen kann! Eine syntaktisch mögliche Zuweisung auf sich selbst (zum Beispiel `v1 = v1`) wird wohl kaum jemand vornehmen – wenn doch, gäbe es nur einen leichten Performance-Verlust wegen der überflüssigen Kopie. Aufgrund dieser Überlegungen ergibt sich der folgende

Tipp

Verwenden Sie diese Form des Zuweisungsoperators für Objekte mit dynamischen Anteilen (mit `new` erzeugt). In den meisten anderen Fällen wird gar kein eigener Zuweisungsoperator gebraucht. Wenn doch, ist dann die Übergabe per `const&` vorzuziehen. Dieser Tipp gilt nicht für Zuweisungsoperatoren einer Vererbungshierarchie (vgl. Abschnitt 9.9).

Dynamisches Ändern der Vektorgröße

Nachdem die Wirkungsweisen von Kopierkonstruktor und Zuweisungsoperator bekannt sind, macht es nun keine Schwierigkeiten, eine Methode zum Ändern der Größe eines Vektors zu schreiben:

```
template<typename T>
void Vektor<T>::groesseAendern(size_t neueGroesse) {
    T *pTemp = new T[neueGroesse]; // neuen Speicherplatz besorgen
    // die richtige Anzahl von Elementen kopieren
    size_t kleinereZahl = neueGroesse > xDim ? xDim : neueGroesse;
    for(size_t i = 0; i < kleinereZahl; ++i) {
        pTemp[i] = start[i];
    }
    delete [] start;              // alten Speicherplatz freigeben
    start = pTemp;                // Verwaltungsdaten aktualisieren
    xDim = neueGroesse;
}
#endif // VEKTOR_T
```

Übungen

9.5 Schreiben Sie einen Zuweisungsoperator für die Klasse `NummeriertesObjekt` aus Abschnitt 6.2. Überlegen Sie vorher genau, was er eigentlich tun soll.

9.6 Schreiben Sie für die Klasse `MeinString` aus Abschnitt 6.1.1 Operatoren, die den Methoden `assign()`, `at()` und `ausgabe()` entsprechen.

9.7 Schreiben Sie für die Klasse `MeinStringdie` Operatoren `+=` und `+` zum Verketten von `MeinString`-Objekten.

Hinweis

Bei der Konstruktion des Zuweisungsoperators in einer Vererbungshierarchie sind besondere Bedingungen zu beachten. Ab Seite 365 wird darauf eingegangen.

9.2.3 Mathematische Vektoren

Die oben beschriebene Klasse `Vektor` kann beliebige kopierbare Objekte eines vorgegebenen Typs aufnehmen. Nun konstruieren wir uns eine maßgeschneiderte Klasse `MathVektor` für Objekte der Datentypen `int`, `float`, `complex`, `Rational` etc., die alle Eigenschaften der Klasse `Vektor` hat und noch ein paar mehr, nämlich dass mathematische Operationen mit Objekten der Klasse einfach möglich sind. Alternativ gibt es die Klasse `valarray` der C++-Standardbibliothek (siehe Seite 857 ff). Sie ist eine Klasse speziell für numerische Arrays und den mathematischen Operationen dazu, nur ist sie um einiges komplizierter. `MathVektor` *ist ein* `Vektor`, und diese Beziehung drückt sich in C++ durch Vererbung aus. Daher wird `MathVektor` von der Klasse `Vektor` des vorhergehenden Abschnitts abgeleitet. Normalerweise soll die Klasse `Vektor` nur einen Einblick in die Wirkungsweise einer Template-Klasse für Vektoren geben, nicht aber zum Standardwerkzeug werden – dafür ist im Allgemeinen die Klasse `vector` der C++-Standardbibliothek besser geeignet. Hier wird davon abgewichen, weil der Indexoperator den übergebenen Index prüfen soll. Als Beispiel für mathematische Operatoren sei hier nur der `*=`-Operator gezeigt.

Listing 9.3: Klasse für Vektoren mit mathematischen Operationen

```
// cppbuch/k9/mathvek/mvektor.t
#ifndef MVEKTOR_T
#define MVEKTOR_T
#include<iostream>
#include"../vektor/vektor.t"

template<typename T>
class MathVektor : public Vektor<T> {
  public:
    typedef Vektor<T> super; // Abkürzung für Oberklassentyp
    MathVektor(size_t=0);
    MathVektor(size_t n, T wert);
    void init(const T&); // alle Elemente setzen
    MathVektor& operator*=(const T&); // Operator *=
    // weitere Operatoren und Funktionen ...
}; // #endif folgt am Dateiende (nach der Implementierung)
```

Die Implementation der Konstruktoren ist entsprechend einfach. Sie initialisieren nur ein Basisklassensubobjekt vom Typ `Vektor`, sonst ist nichts weiter zu tun.

```
template<typename T>
MathVektor<T>::MathVektor(size_t x)
: Vektor<T>(x) {
}

template<typename T>
MathVektor<T>::MathVektor(size_t n, T wert)
: Vektor<T>(n, wert) {
}

template<typename T>
void MathVektor<T>::init(const T& wert) {
   for(size_t i = 0; i < super::size(); ++i) {
      super::operator[](i) = wert;
   }
}
```

Weil `MathVektor` keinerlei Attribute außer den geerbten hat, sind weder Destruktor, noch Kopierkonstruktor oder Zuweisungsoperator notwendig! Die vom System implizit generierten Funktionen Destruktor, Kopierkonstruktor und Zuweisungsoperator genügen, weil die entsprechenden Operationen für das Basisklassensubobjekt von der Klasse `Vektor` erledigt werden. Der Indexoperator wird von der Basisklasse geerbt, sodass die Klassendeklaration dadurch sehr einfach wird.

Bei dieser Diskussion wird vorausgesetzt, dass in einem Programm – wie in den Beispielen oben – Methoden über den Namen eines definierten Objekts aufgerufen werden, nicht über Referenzen oder Zeiger. Polymorphismus wird nicht betrachtet. Polymorphes Verhalten ist bei vektorähnlichen Objekten nicht üblich. Sollte es dennoch gewünscht sein, müsste der Zuweisungsoperator weiteren Bedingungen genügen, die in Abschnitt 9.9 erläutert werden. Auch wäre dann die Klasse `vector` als Oberklasse nicht geeignet, weil sie keinen virtuellen Konstruktor hat.

9.2.4 Multiplikationsoperator

Dieser Operator soll auf einfache Weise einen Vektor mit einem Skalar multiplizieren, indem zum Beispiel `v1 *= 1.23;` geschrieben wird. Der Operator muss dafür sorgen, dass alle Elemente von `v1` mit 1.23 multipliziert werden.

```
template<typename T>
MathVektor<T>& MathVektor<T>::operator*=(const T& zahl) {
   for(size_t i = 0; i < super::size(); ++i) {
      super::operator[](i) *= zahl; // elementweise Multiplikation
   }
  return *this;
}
```

Vielleicht fragen Sie sich, was die Angabe von `super::` soll, wo doch `MathVektor` von `vector` abgeleitet ist und `operator[]` bekannt sein sollte? Die Antwort: Es gibt beim Aufruf von `operator[]` keine Argumente, die von einem Template-Parameter abhängen. Damit ist eine Instanziierung der Funktion `operator[]` bezogen auf `T` nicht möglich. Der Compiler würde ohne `super::` eine Deklaration von `operator[]` vermissen. Mit `super::` ist jedoch klar, dass `operator[]` ein Element der Oberklasse ist; dort existiert die Deklaration. Natür-

lich könnte auch `Vektor<T>::operator[](i)` geschrieben werden. Eine weitere Alternative wäre die Verwendung von `this`, zum Beispiel

```
typedef Vektor<T> super;
// Verwendung:
super::operator[](i) *= zahl;        // gleichbedeutend mit:
Vektor<T>::operator[](i) *= zahl;    // gleichbedeutend mit:
this->operator[](i) *= zahl;         // gleichbedeutend mit:
(*this)[i] *= zahl;
```

Jetzt kann die Multiplikation eines beliebig großen Vektors mit einem Skalar sehr einfach geschrieben werden, zum Beispiel:

```
MathVektor<double> v1(100), v2(200);
//...
v1 *= 1.234567;     // alle Elemente von v1 multiplizieren
```

Aber was ist mit Fällen wie `v1 = v2 * 1.234567;` oder `v1 = 1.234567 * v2;`, in denen drei Beteiligte anstatt nur zwei vorhanden sind? Am Ende des Kapitels 9.1 wurde darauf hingewiesen, dass ein binärer Operator mithilfe des Kurzformoperators implementiert werden kann. Dabei müssen die binären Operatoren nicht einmal Elementfunktionen der Klasse sein, sofern sie nicht direkt auf private Daten zugreifen. Die Implementierung für die beiden Fälle ergibt in Analogie zum Vorschlag am Ende von Abschnitt 9.1:

```
// * Operator für den Fall v1 = zahl*v2;
template<typename T>
MathVektor<T> operator*(const T& zahl, MathVektor<T> mv) {
   return mv *= zahl; // Aufruf von operator*=()
}

// * Operator für den Fall v1 = v2*zahl; (vertauschte Reihenfolge der Argumente)
template<typename T>
MathVektor<T> operator*(const MathVektor<T>& v, const T& zahl) {
   return zahl*v; // Aufruf des obigen operator*()
}
```

Auf die Erweiterung mit zusätzlichen mathematischen oder anderen Operatoren wird nicht weiter eingegangen. Man beachte übrigens, dass sich die Länge eines Vektors durch Zuweisung ändern kann. In dem Beispiel

```
v1 = 1.234567 * v2;
```

spielt die Länge von `v1` *vor* der Zuweisung keine Rolle, und nach der Zuweisung haben `v1` und `v2` dieselbe Anzahl von Elementen. Dafür sorgt der Zuweisungsoperator der Klasse `Vektor`. Dieser Operator ist nicht vererbbar, er wird hier bei der elementweisen Kopie der Elemente eines `MathVektors` wirksam, d.h. bei der Kopie des Basisklassensubobjekts.

9.3 Inkrement-Operator ++

Der Inkrement-Operator ++ tritt in der Präfix- oder Postfix-Form auf, je nachdem, ob das Hochzählen vor oder nach der Verwendung einer Variablen geschehen soll. Um nicht nur ein ganz triviales Erhöhen um eins zu zeigen, wird eine Klasse `Datum` vorgestellt, in der das Hochzählen mit ++ das Weiterschalten des Tages auf das nächste Datum bedeutet. Dabei müssen Monatsübergänge, Schaltjahre und mehr berücksichtigt werden.

Nach Tabelle 9.1 (Seite 318) ist eine Unterscheidung von Präfix- und Postfix-Version durch ein Argument 0 bei der letzteren realisiert. Aus diesem Grund wird der Postfix-Inkrementoperator von der Präfix-Version durch einen zusätzlichen `int`-Parameter im Argument unterschieden, obwohl der Parameter sonst nicht weiter benötigt wird – ein vielleicht nicht besonders eleganter Kunstgriff. Um bereits *vor* der Konstruktion eines Datums Tag, Monat und Jahr überprüfen zu können, zum Beispiel nach einer Dialogeingabe, wird eine globale Funktion zur Überprüfung bereitgestellt. Die Funktion wird auch innerhalb der Methoden der Klasse benutzt. Sie benutzt die globale Funktion `istSchaltjahr()`, die auch von der Methode `Datum::istSchaltjahr()` gerufen wird.

Listing 9.4: Klasse Datum

```
// cppbuch/k9/datum/datum.h
#ifndef DATUM_H
#define DATUM_H

class Datum {
  public:
    Datum();                              // Standardkonstruktor
    Datum(int t, int m, int j);           // allgemeiner Konstruktor
    void set(int t, int m, int j);        // Datum setzen
    void aktuell();                       // Systemdatum setzen
    bool istSchaltjahr() const;
    Datum& operator++();                  // Tag hochzählen, präfix
    Datum operator++(int);                // Tag hochzählen, postfix
    int tag()  const;                     // lesende Methoden
    int monat() const;
    int jahr() const;
  private:
     int tag_, monat_, jahr_;
};
// Prototyp der globalen Schaltjahr-Funktion
bool istSchaltjahr(int jahr); // Implementation s.u.
// inline Methoden
// Um Code-Duplikation zu vermeiden, wird set() aufgerufen
inline Datum::Datum(int t, int m, int j) { set(t, m, j);}
inline int Datum::tag() const { return tag_; }
inline int Datum::monat() const { return monat_;}
inline int Datum::jahr() const { return jahr_; }
inline bool Datum::istSchaltjahr() const {
  return ::istSchaltjahr(jahr_); // globale Funktion mit ::-Operator
}
```

```
// globale Funktionen
bool istGueltigesDatum(int t, int m, int j);

inline bool istSchaltjahr(int jahr) {
    return (jahr % 4 == 0 && jahr % 100) || jahr % 400 == 0;
}
#endif  // DATUM_H
```

Implementierung zur Klasse Datum

Die folgend beschriebene Implementierung der Operatormethoden ist in der Datei *cppbuch/k9/datum/datum.cpp* der Beispiele zu finden. Der Standardkonstruktor initialisiert das Datum mit dem Systemdatum:

```
Datum::Datum() {
   aktuell(); // siehe unten
}

void Datum::set(int t, int m, int j) { // Datum neu setzen
    // Damit nur korrekte Datum-Objekte möglich sind:
    if(!istGueltigesDatum(t, m, j)) {
       // siehe Aufgabe am Ende des Abschnitts
    }
    tag_ = t;
    monat_ = m;
    jahr_ = j;
}
```

In der Funktion `aktuell()` treten vordefinierte Datentypen und Funktionen auf, die auf jedem C/C++-System zu finden sind (Einzelheiten siehe Seite 881). Die Datentypen und Funktionen `time_t`, `time()`, `tm` und `localtime()` sind in `<ctime>` deklariert, `time_t` beispielsweise als `long`. `time()` liefert die Anzahl der Sekunden, die seit dem 1. Januar 1970 bis zum Aufruf verstrichen sind, als Zahl vom Typ `time_t`. Die Funktion `localtime()` gibt einen Zeiger auf eine statische Struktur des Typs `tm` zurück, die ausgewertet wird.

```
// Systemdatum eintragen
void Datum::aktuell() {
    time_t now = time(NULL); // aktuelle Zeit in s seit 1.1.1970
    tm *z = localtime(&now); // Zeiger auf eine vordefinierte Struktur des Typs tm.
                             // Umwandlung mit localtime().
    jahr_ = z->tm_year + 1900;
    monat_ = z->tm_mon + 1;  // localtime() liefert 0..11
    tag_  = z->tm_mday;
}
```

Die folgende Methode dient schaltet den Tag weiter. Falls ein ungültiges Datum wie der 31. April gebildet wird, muss es sich um den Übergang zum nächsten Monat handeln:

```
Datum& Datum::operator++() { // Präfix (kein int-Argument)
   ++tag_;
   // Monatsende erreicht?
   if(!istGueltigesDatum(tag_, monat_, jahr_)) {
     tag_ = 1;
```

```
    if (++monat_ > 12) {
      monat_ = 1;
      ++jahr_;
    }
  }
  return *this;
}
```

Der Postfix-Inkrementierungsoperator folgt einem Muster, das für alle nachgestellten ++-Operatoren gültig ist. Zuerst wird der alte Zustand des Objekts »gerettet«, dann wird das Objekt verändert und der gerettete Wert zurückgegeben:

```
Datum Datum::operator++(int) { // Datum um 1 Tag erhöhen
                               // Parameter wird nicht gebraucht
  Datum temp(*this);
  ++*this;                    // Präfix ++ aufrufen
  return temp;
}
```

Anstelle von `++*this;` kann alternativ der Aufruf `operator++();` stehen. Die Angabe eines Variablennamens in der Parameterliste des Postfix-Operators ist nicht notwendig, weil die Variable nicht benötigt wird. Die Implementierung des Postfix-Operators ändert den Wert des Objekts mithilfe des Präfix-Operators, sodass eine Code-Duplikation vermieden wird, gibt aber mithilfe der Variablen `temp` den vorherigen Wert entsprechend der gewünschten Semantik zurück. Der Aufruf in einem Programm wird wie folgt interpretiert:

`++dat1;`	Aufruf wie `dat1.operator++();`
`dat1++;`	Aufruf wie `dat1.operator++(0);`

Damit ist das Rüstzeug zur Implementierung eigener Inkrement-Operatoren vorhanden. Die verzögerte Objektänderung in der Postfix-Form muss allerdings, wie gezeigt, selbst programmiert werden, sie wirkt nicht automatisch. Manchmal ist es von der Semantik her gleichgültig, ob ein Prä- oder Postfix-Operator genommen wird, wie etwa bei einer allein stehenden Anweisung `X++;` für ein beliebiges Objekt `X`. Jedoch kann eine temporäre Variable beim Postfix-Operator nicht vermieden werden, sodass in solchen Fällen aus Effizienzgründen ein Präfix-Operator vorzuziehen ist, das heißt, `++X;` ist besser. Alles in diesem Abschnitt über den Inkrement-Operator Gesagte gilt entsprechend natürlich auch für den Dekrement-Operator --.

Zum Schluss folgt die globale Funktion zur Überprüfung, ob ein Tag/Monat/Jahr-Tripel ein gültiges Datum darstellt. Die Jahreszahl soll größer als 1582 sein, weil in dem Jahr der bis heute gültige Gregorianische Kalender eingeführt worden war.

```
bool istGueltigesDatum(int t, int m, int j) {
  // Tage pro Monat (static vermeidet Neuinitialisierung):
  static int tmp[]={31,28,31,30,31,30,31,31,30,31,30,31};
  int letzterTagImMonat = tmp[m-1];
  if(m == 2 && istSchaltjahr(j)) {
    letzterTagImMonat = 29;
  }
  return ( m >= 1 && m <= 12 && j >= 1583 && j <= 2399 // oder mehr
       && t >= 1 && t <= letzterTagImMonat);
}
```

Übungen

9.8 Der Rückgabetyp der Präfix-Form des Inkrementoperators ist als Referenz formuliert. Warum?

9.9 Wäre eine Referenz als Rückgabetyp in der Postfix-Form sinnvoll?

9.10 Schreiben Sie einen Ausgabeoperator `<<` zur Ausgabe von Datum-Objekten in der üblichen Schreibweise, d.h. Tag, Monat und Jahr durch einen Punkt getrennt (zum Beispiel 31.12.2011).

9.11 Implementieren Sie die relationalen Operatoren `==`, `!=` und `<` für Objekte der Klasse Datum. Letzterer gibt an, welches Datum von zweien früher liegt.

9.12 Die Differenz zweier Datum-Objekte in Tagen soll durch eine globale Funktion mit der Schnittstelle `int DatumDifferenz(const Datum& a, const Datum& b)` berechnet werden.

9.13 Berechnen Sie, welches maximale Systemdatum auf Ihrem System möglich ist. Hinweis: siehe die obige Methode `aktuell()`.

9.14 Die Methode `set()` von Seite 334 soll eine von Ihnen geschriebene `UngueltigesDatumException` werfen, wenn das Datum nicht korrekt ist. `UngueltigesDatumException` soll von `runtime_error` erben. Prüfen Sie das Verhalten mit einem Testprogramm, indem Sie ein falsches Datum erzeugen, zum Beispiel

```
Datum einDatum;
try {
   einDatum.set(30, 2, 2012);
}
catch(const UngueltigesDatumException& e) {
   cout << e.what() << endl;
}
```

Die Ausgabe des Programms soll zum Beispiel lauten: »30.2.2012 ist ein ungültiges Datum« oder ähnlich.

9.4 Typumwandlungsoperator

Erinnern Sie sich an die `if`- und die `while`-Anweisung, in deren Bedingung nicht nur ein Ausdruck mit einem logischen oder arithmetischen Ergebnis, sondern auch ein Zeiger stehen darf? Ein Zeiger muss dann in einen äquivalenten Wert (0 oder 1, entsprechend `false` oder `true`) umgewandelt werden. Sie wissen, dass der Compiler Typumwandlungen automatisch vornimmt, wenn zum Beispiel einer `float`-Zahl eine `int`-Zahl zugewiesen wird. Bedingung für die Umwandlung von Klassenobjekten ist ein Typumwandlungskonstruktor (siehe Abschnitt 4.3.4) oder ein Typumwandlungsoperator, der implizit durch den Compiler oder explizit durch Angabe des Datentyps angewendet werden kann:

```
float f;
int i = 1234;
f = i;                          // implizit
f = static_cast<float>(i);      // explizit
```

Die Typumwandlung mit einem Konstruktor haben Sie in Abschnitt 4.3.4 kennen und in Abschnitt 4.4 anwenden gelernt. C++ erlaubt darüber hinaus die Definition von Typumwandlungsoperatoren für selbst geschriebene Klassen, um ein Objekt der Klasse in einen anderen Datentyp abzubilden. *Typumwandlungsoperatoren sind Elementfunktionen und haben weder eine Parameterliste noch einen Rückgabetyp* (siehe Syntax in Abbildung 9.5).

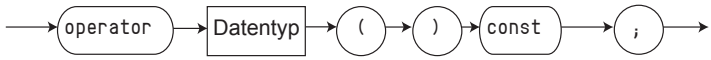


Abbildung 9.5: Syntax eines Typumwandlungsoperators

Hier wird als Beispiel die Umwandlung eines `Datum`-Objekts in einen String gezeigt, der dann weiterverarbeitet oder ausgegeben werden kann. Die Deklaration in der Klasse `Datum` sowie die Implementierung lauten:

```
// datum.h
#include<string>

class Datum {
    // .... wie vorher
    operator std::string() const;
};
```

```
// in datum.cpp:
Datum::operator std::string() const {
    std::string temp("tt.mm.jjjj");
    temp[0] = tag_/10 +'0';      // implizite Umwandlungen
    temp[1] = tag_%10 +'0';
    temp[3] = monat_/10 +'0';
    temp[4] = monat_%10 +'0';

    int pos = 9;                 // letzte Jahresziffer
    int j = jahr_;
    while(j > 0) {
       temp[pos] = j % 10 + '0'; // letzte Ziffer
       j /= 10;                  // letzte Ziffer abtrennen
       --pos;
    }
    return temp;
}
```

Es ist sowohl die implizite als auch die explizite Umwandlung möglich, wie die folgenden Zeilen zeigen.

```
using namespace std;
Datum d;
string s1 = d;                       // implizit oder
string s2 = (string)d;               // explizit
string s3 = static_cast<string>(d); // explizit
cout << s1 << endl;                  // Ausgabe
```

Empfehlung

Um die Typprüfung des Compilers nicht unnötig zu umgehen, sollte man Typumwandlungsoperatoren nur in begründeten Fällen einsetzen. Alternativ bieten sich Methoden an. In diesem Beispiel wäre eine Methode namens `toString()` anstelle des Typumwandlungsoperators die bessere Lösung, weil dann eine implizite Typumwandlung von vornherein unmöglich ist.

Übung

9.15 Ersetzen Sie den Typumwandlungsoperator durch eine Methode `toString()`.

9.5 Smart Pointer: Operatoren -> und *

Ein Schlüsselkonzept von C++ ist der Zugriff auf Objekte und deren Elementfunktionen über Zeiger, *Indirektion* genannt. Wer kennt die Fallstricke nicht, die mit der Verwendung von Zeigern einhergehen! Typische Fehler sind die Dereferenzierung von nicht initialisierten Zeigern oder versehentliche Versuche, `delete` mehr als einmal auf einen Zeiger anzuwenden. Andere Fehler sind hängende Zeiger und verwitwete Objekte (siehe Seite 203) – alles Fehler, die manchmal schwer zu finden sind. Gegen diese typische C-Krankheit bietet C++ eine Medizin: »intelligente« Zeiger (englisch *smart pointers*).

Die Vielseitigkeit von C++ erlaubt es, auch die Operatoren `->` und `*` zu überladen, sodass damit eine Klasse von »smart pointers« geschaffen werden kann, mit der Indirektion auf direkte, sichere und effiziente Art darstellbar ist. Weil Zeiger typischerweise für dynamische Objekte verwendet werden, werden im Folgenden nur smarte Pointer für Heap-Objekte betrachtet. Das Problem, versehentlich `delete` zu vergessen, existiert nicht für Objekte, die auf dem Stack abgelegt werden.

Was erwarten wir von einem intelligenten Zeiger?

1. Die Syntax der Benutzung soll möglichst der bekannten Schreibweise von »normalen« Zeigern gleichen.
2. Ein intelligenter Zeiger sollte ohne viel Aufwand für verschiedene Klassen möglich sein, ohne an Typsicherheit einzubüßen.
3. Gegenüber einem normalen Zeiger sollte kein Laufzeitmehrbedarf entstehen.
4. Ein smarter Pointer hat niemals einen undefinierten Wert. Er verweist entweder auf ein Objekt oder definitiv auf nichts, um hängende Zeiger zu vermeiden.
5. Die Dereferenzierung von nicht existenten Objekten soll nicht zu einem Programmabsturz, sondern zu einer Exception führen.
6. Das Objekt, auf das der Zeiger verweist, soll gelöscht werden, sobald der Zeiger nicht mehr gültig ist, um verwitwete Objekte zu vermeiden.

Des Weiteren sind noch viele Dinge denkbar, die ein intelligenter Zeiger bei der Dereferenzierung erledigen könnte. Beispiele: ein Objekt erst bei Bedarf vom Massenspeicher holen, in Abhängigkeit vom Zugriff auf konstante oder nichtkonstante Objekte unterschiedlich reagieren und anderes mehr. Um das folgende Beispiel nicht zu überfrachten, bleibt es auf die oben angeführten Punkte beschränkt. Bei der Verwendung ist zu beachten, dass eine gemischte Verwendung von »intelligenten« mit normalen C-Zeigern nicht sinnvoll ist, weil die Sicherheitsmaßnahmen der Klasse unterlaufen werden können.

Der bisher bekannte Operator -> ist für jede Adresse verwendbar, die vom Typ »Zeiger auf `struct` oder `class`« ist. Der überladene Operator verhält sich genauso: Seine Benutzung ist nichts anderes als der Aufruf der Funktion `operator->()` (siehe Tabelle 9.1, Seite 318), die eben so einen Zeiger zurückgibt – nur dass sie vorher noch etliche andere Dinge erledigen kann! Für Grunddatentypen ist ein Überladen allerdings nicht möglich.

Die Forderung nach Typsicherheit und gleichzeitiger Anwendbarkeit für verschiedene Klassen wird durch ein Klassen-Template erfüllt. Durch `inline`-Definitionen entsteht, verglichen mit einem normalen Zeiger, kein zusätzlicher Laufzeitaufwand für die Funktionalität. Nur darüber hinausgehende Extras wie die Überprüfung auf Gültigkeit kosten Laufzeit – nicht alles ist umsonst. Falls auf einen nicht-gültigen Zeiger zugegriffen wird, soll eine `NullPointerException` geworfen werden:

```
// cppbuch/k9/smartptr/NullPointerException.h
#ifndef NULLPOINTEREXCEPTION_H
#define NULLPOINTEREXCEPTION_H
#include<stdexcept>

class NullPointerException : public std::runtime_error {
 public:
   NullPointerException()
      : std::runtime_error("NullPointerException!") {
   }
};
#endif
```

Im folgenden Beispiel wird ein Klassen-Template `SmartPointer` für »intelligente« Zeiger auf *Heap-Objekte* vorgestellt. Die Klasse soll nicht für Zeiger gelten, die auf ein Array von Objekten verweisen.

Listing 9.5: Template für SmartPointer

```
// cppbuch/k9/smartptr/smartptr.t
#ifndef SMARTPTR_T
#define SMARTPTR_T
#include"NullPointerException.h"

template<typename T>
class SmartPointer {
   public:
      SmartPointer(T *p = 0);
      ~SmartPointer();  // nicht virtual: Vererbung ist nicht geplant
      T* operator->() const;
      T& operator*() const;
      SmartPointer& operator=(T *p);
```

```
    operator bool() const;
  private:                  // Ergänzungen siehe unten
    T* zeigerAufObjekt;
    void check() const;     // Prüfung auf nicht-Null
};
// ... hier die unten beschriebenen inline-Implementierungen einfügen
#endif    // SMARTPTR_T
```

Die Variable `zeigerAufObjekt` verweist auf ein Objekt eines beliebigen Typs `T`. Alle Zugriffe geschehen über die Operatoren `->` und `*`. Der Konstruktor stellt sicher, dass `zeigerAufObjekt` den Wert 0 bei Initialisierung erhält, wenn keine Objektadresse übergeben wird.

```
template<typename T>
inline SmartPointer<T>::SmartPointer(T *p)
: zeigerAufObjekt(p) {
}

template<typename T>
inline SmartPointer<T>::~SmartPointer() {
  delete zeigerAufObjekt;
}

template<typename T>
inline void SmartPointer<T>::check() const {
  if(!zeigerAufObjekt) {
    throw NullPointerException();
  }
}
```

Der Operator `->` gibt einen Zeiger auf das Objekt zurück, wobei im Unterschied zum bisher bekannten Pfeiloperator geprüft wird, ob er auf ein Objekt verweist. Falls nicht, gibt es eine Exception.

```
template<typename T>
inline T* SmartPointer<T>::operator->() const {
   check();
   return zeigerAufObjekt;
}
```

Der Operator `*` gibt eine Referenz auf das Objekt zurück, wobei im Unterschied zum bisher bekannten Dereferenzierungsoperator geprüft wird, ob `ZeigerAufObjekt` tatsächlich auf ein Objekt verweist. Falls nicht, gibt es eine Exception.

```
template<typename T>
inline T& SmartPointer<T>::operator*() const {
   check();
   return *zeigerAufObjekt;
}
```

Schließlich soll die Abfrage möglich sein, ob ein smarter Pointer auf ein Objekt zeigt. Dazu wird ein Typumwandlungsoperator benutzt, sodass Abfragen in `if`- oder `while`-Bedingungen mit der vertrauten Syntax `if(smartPointerObjekt) ...` möglich sind.

```
template<typename T>
inline SmartPointer<T>::operator bool() const {
   return bool(zeigerAufObjekt);
}
```

Die Aufgabe, ein Objekt zu löschen, sobald der zugehörige Zeiger nicht mehr gültig ist, übernimmt der Destruktor des smarten Pointers. Es muss in diesem Beispiel vermieden werden, dass mehrere `SmartPointer`-Objekt auf dasselbe Objekt verweisen. Bei dynamischen Objekten gäbe es andernfalls das Problem, dass das erste ungültig werdende `SmartPointer`-Objekt das referenzierte Objekt löscht mit der Folge, dass die anderen nicht mehr darauf zugreifen könnten, obwohl die internen Zeiger (`zeigerAufObjekt`-Variablen) ungleich 0 sind.

Es gibt Möglichkeiten, beliebig viele smarte Pointer auf ein Objekt verweisen zu lassen, ohne dass diese Schwierigkeiten auftreten. Sie setzen eine Buchführung voraus, die mitzählt, wie viele Zeiger auf ein Objekt verweisen. Die Benutzungszählung (englisch *reference counting*) wird von der Klasse `shared_ptr` (siehe Abschnitt 9.5.1) der C++-Standardbibliothek geleistet, übersteigt aber den Rahmen dieses einfachen Beispiels, weswegen hier schlicht verboten wird, dass mehr als ein smarter Pointer auf ein Objekt verweist. Das geschieht am einfachsten dadurch, dass sowohl der Zuweisungsoperator als auch der Initialisierungs- oder Kopierkonstruktor durch eine `private`-Deklaration unzugänglich gemacht werden:

```
  private:    // Ergänzung der Klassendefinition:
   void operator=(const SmartPointer& ); // Zuweisung p1 = p2; verbieten
   SmartPointer(const SmartPointer&); // Initialisierung mit SmartPointer verbieten
```

Damit ist zwar nicht jede, aber wenigstens eine Fehlerquelle gestopft. Eine Parameterübergabe per Wert ist die Erzeugung und Initialisierung eines Objekts, bei der dem Kopierkonstruktor eine Referenz auf das Original mitgegeben wird. Ein privater Kopierkonstruktor sorgt dafür, dass eine Übergabe per Wert nicht möglich ist und ein `SmartPointer` nur noch per Referenz übergeben werden kann. Obwohl sich da die Benutzung von `SmartPointer`-Objekten von der normaler Zeiger unterscheidet, ist dies ein erwünschter Effekt, denn: Eine Übergabe per Wert erzeugt eine temporäre Kopie des Parameters, deren Destruktor am Ende der Funktion aufgerufen wird. Der Destruktor würde das zugehörige Objekt löschen, obwohl es im aufrufenden Programm noch gebraucht wird!

Das folgende Programm zeigt einige Anwendungen von smarten Pointern. Es werden Objekte der Klassen `A` und `B` erzeugt, auf die mit smarten Pointern zugegriffen wird. Dabei erbt `B` von `A`, damit polymorphes Verhalten gezeigt werden kann. Die Destruktoren dokumentieren das Vergehen der Objekte, wenn die Zeiger ungültig werden.

Listing 9.6: Beispiel mit SmartPointer-Objekten

```
// cppbuch/k9/smartptr/main.cpp
#include"smartptr.t"
#include<iostream>
using namespace std;

class A {
   public:
     virtual void hi() { cout << "hier ist A::hi()" << endl;}
```

```
    virtual ~A()    { cout << "A::Destruktor" << endl; }
};

class B : public A {
   public:
     virtual void hi() { cout << "hier ist B::hi()" << endl; }
     virtual ~B()    { cout << "B::Destruktor" <<endl; }
};

// Parameterübergabe per Wert ist hier nicht möglich, wohl aber per Referenz:
template<typename T>
void perReferenz(const SmartPointer<T>& p) {
   cout << "Aufruf: perReferenz(const SmartPointer<T>&):";
   p->hi();   // d.h. (p.operator->())->hi();
}

int main() {
   cout << "Zeiger auf dynamische Objekte:" << endl;
   cout << "Konstruktoraufruf" << endl;
   SmartPointer<A> spA(new A);

   cout << "Operator ->" << endl;
   spA->hi();

   cout << "Operator *" << endl;
   (*spA).hi();

   cout << "Polymorphismus:" << endl;
   SmartPointer<A> spAB(new B); // zeigt auf B-Objekt
   spAB->hi();                  // B::hi()

   // Parameterübergabe eines SmartPointer
   perReferenz(spAB);

   // Die Wirkung der Sicherungsmaßnahmen im Vergleich zu einfachen
   // C++-Zeigern zeigen die folgenden Zeilen:
   SmartPointer<B> spUndef;
   try {
   if (!spUndef)        // = if(!(spUndef.operator bool()))
      cout << "undefinierter Zeiger:" << endl;
   // Zugriff auf nichtinitialisierten Zeiger bewirkt Laufzeitfehler:
   spUndef->hi();               // Laufzeitfehler!
   (*spUndef).hi();             // Laufzeitfehler!
   } catch(const NullPointerException& ex) {
      cout << "Laufzeitfehler: " << ex.what() << endl;
   }

   // alle folgenden Anweisungen bewirken Fehlermeldungen des Compilers!
   // Typkontrolle: ein A ist kein B!
   SmartPointer<B> spTyp(new A);      // Fehler!
   // keine Initialisierung mit Kopierkonstruktor
   SmartPointer<A> spY = spA;         // Fehler!
   // Zuweisung ist nicht möglich (privater =-Operator):
```

```
    SmartPointer<A> spA1;
    spA1 = spA;                          // Fehler!
}
```

9.5.1 Smart Pointer und die C++-Standardbibliothek

Ein `SmartPointer`, wie oben beschrieben, behält ein Objekt solange, bis der Destruktor beim Verlassen des Gültigkeitsbereichs wirksam wird. Abschnitt 33.1 beschreibt vordefinierte Klassen der C++-Standardbibliothek für smarte Zeiger. Eine davon ist die schon erwähnte Klasse `shared_ptr`. Es können mehrere Objekte dieser Klasse auf ein Objekt verweisen, weil sie eine Benutzungszählung implementiert. Im Gegensatz zur Klasse `SmartPointer` ist eine Parameterübergabe per Wert nicht schädlich, weil die Kopie nur den Benutzungszähler erhöht. Der Destruktor eines `shared_ptr`-Objekts wendet nur dann `delete` auf das referenzierte Objekt an, wenn der Benutzungszähler 1 ist, also kein anderes `shared_ptr`-Objekt (mehr) darauf verweist.

Mehr zu `shared_ptr` und Beispiele lesen Sie in Abschnitt 33.1.

9.6 Objekt als Funktion

In einem Ausdruck wird der Aufruf einer Funktion durch das von der Funktion zurückgegebene Ergebnis ersetzt. Die Aufgabe der Funktion kann von einem Objekt übernommen werden, eine Technik, die in den Algorithmen und Klassen der C++-Standardbibliothek häufig eingesetzt wird. Dazu wird der Funktionsoperator `()` mit der Operatorfunktion `operator()()` überladen. Ein Objekt kann dann wie eine Funktion aufgerufen werden. Ein algorithmisches Objekt dieser Art wird *Funktionsobjekt* oder *Funktor* genannt.

Funktoren sind Objekte, die sich wie Funktionen verhalten, aber alle Eigenschaften von Objekten haben. Sie können erzeugt, als Parameter übergeben oder in ihrem Zustand verändert werden. Die Zustandsänderung erlaubt einen flexiblen Einsatz, der mit Funktionen nur über zusätzliche Parameter möglich wäre. Wie funktioniert ein Funktor? Betrachten wir dazu eine Klasse, deren Objekte den Sinus eines Winkels berechnen. Den Objekten wird bei der Erzeugung bereits mitgeteilt, in welchen Einheiten der Winkel angegeben wird.

Das syntaktisch entscheidende Element ist der überladene Klammeroperator `operator()()`. Durch ihn ist es möglich, dass ein Objekt wie eine Funktion aufgerufen werden kann.

Listing 9.7: Beispiel für ein Funktionsobjekt

```
// cppbuch/k9/funktor/sinus.h
#ifndef SINUS_H
#define SINUS_H
#include<cmath> // sin(), Konstante M_PI für π
// manche Compiler stellen Konstanten wie M_PI nicht zur Verfügung
#ifndef M_PI
```

```
 #define M_PI 3.14159265358979323846
#endif

class Sinus {
   public:
     enum Modus { bogenmass, grad, neugrad};
     Sinus(Modus m = bogenmass)
     : berechnungsart(m) {
     }

     double operator()(double arg) const {
        double erg;
        switch(berechnungsart) {
          case bogenmass : erg = std::sin(arg);
                           break;
          case grad      : erg = std::sin(arg/180.0*M_PI);
                           break;
          case neugrad   : erg = std::sin(arg/200.0*M_PI);
                           break;
          default  : ;       // kann hier nicht vorkommen
        }
        return erg;
     }
 private:
     Modus berechnungsart;
#endif // SINUS_H
```

Die Algorithmen der C++-Standardbibliothek benutzen häufig Funktionsobjekte. Ein Beispiel dafür ist `setprecision` zum Festlegen der Anzahl ausgegebener Stellen bei der Ausgabe von Zahlen (siehe Abschnitt 10.3). Im Beispielprogramm werden Objekte für drei verschiedene Winkeleinheiten angelegt und als Funktionsobjekt benutzt. Ferner wird eins von den drei Objekten als Parameter übergeben. Was unterscheidet Funktionsobjekte von Funktionen?

- Sie erfüllen die Aufgabe einer Funktion, haben aber alle Eigenschaften von Objekten.
- Funktionsobjekte sind flexibler als Funktionen, weil virtuelle Funktionen und Vererbung möglich sind.

Listing 9.8: Funktor

```
// /cppbuch/k9/funktor/main.cpp
#include<iostream>
#include"sinus.h"
using namespace std;

void sinusAnzeigen(double arg, const Sinus& funktor) {
   cout << funktor(arg) << endl;
}

int main() {
   Sinus sinrad;                  // Funktoren anlegen
   Sinus sinGrad(Sinus::grad);
   Sinus sinNeuGrad(Sinus::neugrad);
```

```
  // Aufruf der Objekte wie eine Funktion
  cout << "sin(" << M_PI/4.0 <<" rad) = "    // π/4 = 0.785398...
       << sinrad(M_PI/4.0) << endl; // = sinrad.operator()(...);
  cout << "sin(0.7854 rad) = " << sinrad(0.7854) << endl;
  cout << "sin(45 Grad)   = " << sinGrad(45.0)  << endl;
  cout << "sin(50 Neugrad) = " << sinNeuGrad(50.0) << endl;

  // Übergabe eines Funktors an eine Funktion
  sinusAnzeigen(50.0, sinNeuGrad);
  // ...
}
```

- Sie ermöglichen eine einfachere Schnittstelle: Wenn ein Funktionsobjekt per Wert oder per Referenz an eine Funktion übergeben wird, kann es die verschiedensten Daten intern mit sich tragen. Im Beispiel oben ist es die Berechnungsart. Die vergleichbare Übergabe eines Funktionszeigers mit denselben Daten erfordert eine viel breitere Schnittstelle.

Funktionsobjekte sind durchaus nicht auf arithmetische Operationen beschränkt. Ihr Einsatz sollte immer dann überlegt werden, wenn Zeiger auf Funktionen verwendet werden.

9.6.1 Lambda-Funktionen

Mit Lambda-Ausdrücken können einfach namenlose Funktionsobjekte realisiert werden. Lambda-Funktionen oder -Ausdrücke sind anonyme (unbenannte) Funktionen, die direkt an der Stelle ihrer Verwendung definiert werden. Sie sind daher geeignet, wenn die Funktion nur einmal benötigt wird. Die Funktionen haben ihren Namen vom sogenannten Lambda-Kalkül, einem formalen System zur Erforschung einiger theoretischer Grundlagen der Mathematik und Informatik. Lambda-Funktionen werden wie ein Funktionsobjekt benutzt, nur dass man sich das Schreiben einer Klasse und der zugehörigen Funktion `operator()()` sparen kann. Der Compiler wandelt einen Lambda-Ausdruck in ein anonymes Funktionsobjekt um. Im folgenden Beispiel werden Vektorelemente nach dem Absolutbetrag sortiert. Gezeigt wird die Schreibweise ohne und mit Lambda-Funktion.

Listing 9.9: Nach Absolutbetrag sortieren, ohne Lambda-Funktion

```
struct Absolutbetrag {
  bool operator()(int x, int y) {
    return abs(x) < abs(y);
  }
};

int main() {
   vector<int> v = {-11, 3, 4, -7, 8, 1, 2, -4};
   sort(v.begin(), v.end(), Absolutbetrag());
}
```

Listing 9.10: Nach Absolutbetrag sortieren, mit Lambda-Funktion

```
// Auszug aus cppbuch/k9/funktor/lambda/main.cpp
vector<int> v = {-11, 3, 4, -7, 8, 1, 2, -4};
sort(v.begin(), v.end(), [](int x, int y) { return abs(x) < abs(y);});
```

Eine Lambda-Funktion beginnt mit dem Symbol `[]` am Anfang, dann folgen die Parameterliste in runden Klammern und abschließend eine durch geschweifte Klammern begrenzte Verbundanweisung. Das vom Compiler erzeugte Funktionsobjekt hat Zugriff nicht nur auf interne Variablen, sondern auch auf Variablen, die im selben Sichtbarkeitsbereich (*scope*) liegen. Im folgenden Beispiel wird die Summe aller Elemente eines Vektors berechnet:

```
// Auszug aus cppbuch/k9/funktor/lambda/main.cpp
double summe = 0.0;
for_each(v.begin(), v.end(), [&summe](double el) { summe += el; });
```

Wenn die Lambda-Funktion mit dem Symbol `[&]` beginnt, kann innerhalb der Verbundanweisung auf alle sichtbaren Variablen zugegriffen werden. Hier wird der Zugriff auf `summe` beschränkt. Die Menge der zugreifbaren Daten heißt in diesem Zusammenhang auch *Closure*. Die in den eckigen Klammern angegebenen externen Referenzen werden auf entsprechende Attribute des generierten temporären Funktionsobjekts abgebildet, sodass Closure und Objekt sich entsprechen. Wenn die Lambda-Funktion mit dem Symbol `[&]` beginnt, werden im Funktionsobjekt Referenzen angelegt; wenn stattdessen `[=]` genommen wird, werden alle Variablen kopiert.

9.7 new und delete überladen[1]

Wenn Sie für eine Klasse ein eigenes Memory-Management benötigen, können Sie die Operatoren `new` und `delete` (und natürlich `new[]` und `delete[]`) überladen. Um die Klasse der Anwendung nicht damit zu überfrachten, wird das Memory-Management in eine eigene Klasse ausgelagert, von der die Anwendungsklasse erbt. Die Anwendungsklasse sei hier eine einfache Klasse `Person`. Das `main`-Programm erzeugt zum Vergleich sowohl Objekte, die vom Laufzeitsystem auf dem Laufzeit-Stack abgelegt werden, als auch mit `new` erzeugte Objekte, die auf dem Heap landen.

Listing 9.11: main-Programm

```
// cppbuch/k9/new/new/main.cpp
#include"Person.h"
using namespace std;

int main() {
  Person person("Lena");                  // Stack-Objekt
  cout << "Name : " << person.getName() << endl;
  Person* ptr1 = new Person("Jens");      // Heap-Objekt
  cout << "Name : " << ptr1->getName() << endl;
  delete ptr1;                            // Löschen des Heap-Objekts

  size_t anz = 2;
  Person* arr = new Person[anz];          // dynamisches Array anlegen
```

[1] Dieser Abschnitt kann beim ersten Lesen übersprungen werden.

```
   for(size_t i = 0; i < anz; ++i) {
      cout << i << ": " << arr[i].getName() << endl;
   }
   delete[] arr;                          // dynamisches Array löschen
}                                         // Destruktor-Aufruf des Stack-Objekts
```

Um das Überladen kennenzulernen, wird ein Ansatz für ein eigenes Memory-Management zurückgestellt und zunächst nur dokumentiert, was beim Überladen von `new` und `delete` geschieht. Aus diesem Grund gibt es eine Meldung des Destruktors der Klasse `Person`:

Listing 9.12: Klasse Person

```
// cppbuch/k9/new/new/Person.h
#ifndef PERSON_H_
#define PERSON_H_
#include<iostream>
#include<string>
#include"Objekt.h"

class Person : public Objekt {
public:
   Person(const std::string& n = "N.N.")
      : name(n) {
   }

   ~Person() {
      std::cout << "Person-Destruktor aufgerufen ("
                << name << ")" << std::endl;
   }

   const std::string& getName() const {
      return name;
   }
private:
   std::string name;
};
#endif
```

Wenn `Person` nicht von `Objekt` erben würde, kämen die Standardoperatoren `new` und `delete` zum Einsatz. So aber wählt der Compiler die passenden Operatoren der Klasse `Objekt`:

Listing 9.13: Klasse Objekt

```
// cppbuch/k9/new/new/Objekt.h
#ifndef OBJEKT_H_
#define OBJEKT_H_
#include<iostream>

class Objekt {
public:
   virtual ~Objekt() {
      std::cout << "Objekt-Destruktor aufgerufen ("
                << this << ")" << std::endl;
   }
```

```
  static void *operator new(size_t size) {
    std::cout << "new aufgerufen. size=" << size << std::endl;
    return ::operator new(size);
  }
  static void operator delete(void* ptr, size_t size) {
    std::cout << "delete aufgerufen. size=" << size << std::endl;
    ::operator delete(ptr);
  }

  static void *operator new[](size_t size) {
    std::cout << "new[] aufgerufen. size=" << size << std::endl;
    return ::operator new[](size);
  }

  static void operator delete[](void* ptr, size_t size) {
    std::cout << "delete[] aufgerufen. size=" << size << std::endl;
    ::operator delete[](ptr);
  }
};
#endif
```

Die Funktionen der Klasse dokumentieren die Aufrufe und greifen dann auf die Standardoperatoren zurück. Dass andere Speicherbeschaffungsfunktionen möglich sind, wird unten gezeigt. Die Klasse `Objekt` hat einige Besonderheiten:

- Der Destruktor ist `virtual`. Dies ist zwingend erforderlich, soll es keine Speicherlecks geben, wie das Beispiel in Abschnitt 7.6.3 beweist.
- Alle Funktionen sind `static`. Das Schlüsselwort steht hier zur Dokumentation; tatsächlich kann es weggelassen werden. Aber ob es da steht oder nicht – `new` und `delete` sind grundsätzlich `static`. Der Grund liegt auf der Hand: Es wird kein Objekt modifiziert oder abgefragt, sondern es wird erzeugt bzw. gelöscht. Insofern kann auch ein Konstruktor als statische Funktion aufgefasst werden.
- Bei den `delete`-Funktionen kann der zweite Parameter `size` weggelassen werden. Ich habe ihn belassen, um die Größe des zu löschenden Bereichs zu dokumentieren.
- Der Rückgriff auf die Standardoperatoren wird durch den Scope-Operator `::` bewerkstelligt. Fehlt er, handelt man sich eine Kette rekursiver Aufrufe ein.
- Der Compiler wandelt zur Speicherbeschaffung die Anweisung `new Person("Jens");` im `main`-Programm in die Form `Objekt::operator new(sizeof(Person));` um. Das Objekt `Person("Jens")` wird an dieser Stelle konstruiert.
- Die Anweisung `new Person[anz];` im `main`-Programm wird vom Compiler in die Form `Objekt::operator new[](anz*sizeof(Person) + x);` umgewandelt. Das Array wird an dieser Stelle konstruiert. `x` ist eine implementationsabhängige Größe, um Verwaltungsinformationen abzulegen. Unten sehen Sie, wie `x` bestimmt wird.

Der Aufruf des `main`-Programms ergibt die Ausgabe (ohne die Zeilennummern), die auf Ihrem System nicht identisch sein muss:

```
1  Name : Lena
2  new aufgerufen. size=8
3  Name : Jens
4  Person-Destruktor aufgerufen (Jens)
```

```
 5  Objekt-Destruktor aufgerufen (0x804c038)
 6  delete aufgerufen. size=8
 7  new[] aufgerufen. size=20
 8  0: N.N.
 9  1: N.N.
10  Person-Destruktor aufgerufen (N.N.)
11  Objekt-Destruktor aufgerufen (0x804c02c)
12  Person-Destruktor aufgerufen (N.N.)
13  Objekt-Destruktor aufgerufen (0x804c024)
14  delete[] aufgerufen. size=20
15  Person-Destruktor aufgerufen (Lena)
16  Objekt-Destruktor aufgerufen (0xbfda7e0c)
```

Die Zeile 2 zeigt, dass ein `Person`-Objekt 8 Bytes benötigt. Davon sind 4 Bytes für den Zeiger auf die Tabelle virtueller Funktionen. Dies zeigt der Vergleich, wenn der Destruktor von `Objekt` fälschlicherweise *nicht* `virtual` sein sollte.

Die `delete ptr1`-Anweisung bewirkt die Destruktor-Aufrufe von Unter- und Oberklasse (Zeilen 4 und 5). Hier ist zu sehen, dass *zuerst* die Destruktoren aufgerufen werden und *danach* die dokumentierende Anweisung zum Zug kommt (Zeile 6)! Der Grund: Bei der Konstruktion wird zuerst der Speicher beschafft und dann das Objekt konstruiert. Bei der Destruktion muss der Ablauf umgekehrt sein: Erst der Destruktor-Aufruf und dann die Speicherfreigabe. Andernfalls würde der Destruktor in freigegebenem Speicher ablaufen – das darf nicht sein.

Obwohl das Array nur aus zwei Elementen besteht, werden 20 Bytes statt 2 * 8 = 16 angefordert (Zeile 7). Genau diese Menge wird wieder freigegeben (Zeile 14). Daraus ergibt sich der Wert von 4 Bytes für die oben genannte Hilfsgröße `x`.

Die Zeilen 10 bis 13 dokumentieren die Destruktion der beiden Array-Elemente. Auch hier sieht man (Zeile 14), dass die dokumentierende Anweisung *nach* den Destruktoren aufgerufen wird. Die Zeilen 15 und 16 zeigen die Destruktion des anfangs in `main()` angelegten Stack-Objekts.

9.7.1 Speichermanagement mit malloc und free

Oben wird bei den überladenen `new`- und `delete`-Operatoren auf die globalen Standardoperatoren zurückgegriffen. Eine andere Art ist die Speicherbeschaffung mit der C-Funktion `malloc(size_t size)` und die Freigabe mit `free(void* ptr)`. `malloc()` gibt einen Zeiger auf den Beginn des Speicherbereichs der Größe `size` Bytes zurück bzw. NULL, wenn kein Speicher zugewiesen werden kann. `free(ptr)` gibt den Speicher, auf den `ptr` verweist, wieder frei. `ptr` muss auf einen Speicherbereich zeigen, der vorher mit `malloc()`, `calloc()` oder `realloc()` (siehe unten) beschafft worden und noch nicht wieder freigegeben ist. Falls `ptr` jedoch gleich NULL ist, geschieht nichts. Um `malloc()` und `free()` in die Klasse `Objekt` einzubauen, müssen nur kleine Änderungen vorgenommen werden:

Listing 9.14: Klasse Objekt mit malloc/free

```
// cppbuch/k9/new/malloc/Objekt.h
#ifndef OBJEKT_H
#define OBJEKT_H
```

```
#include<iostream>
#include<cstdlib>              // malloc(), free()
#include<new>                  // bad_alloc

class Objekt {
public:
   virtual ~Objekt() {
      std::cout << "Objekt–Destruktor aufgerufen ("
                << this << ")" << std::endl;
   }

   static void *operator new(size_t size) {
      std::cout << "new aufgerufen. size=" << size << std::endl;
      void* ptr = malloc(size);
      if(!ptr) {
         throw std::bad_alloc();
      }
      return ptr;
   }

   static void operator delete(void *ptr, size_t size) {
      std::cout << "delete aufgerufen. size=" << size << std::endl;
      free(ptr);
   }

   static void *operator new[](size_t size) {
      std::cout << "new[] aufgerufen. size=" << size << std::endl;
      void* ptr = malloc(size);
      if(!ptr) {
         throw std::bad_alloc();
      }
      return ptr;
   }

   static void operator delete[](void *ptr, size_t size) {
      std::cout << "delete[] aufgerufen. size=" << size << std::endl;
      free(ptr);
   }
};
#endif
```

Alles andere kann bleiben. Falls die Speicherbeschaffung schief geht, wirft `malloc()` im Gegensatz zu `::new` keine Exception, sondern gibt einen Null-Zeiger zurück. Um dasselbe Verhalten wie mit `::new` zu erreichen, wird der zurückgegebene Wert überprüft und gegebenenfalls `bad_alloc` geworfen. Außer `malloc()` und `free()` gibt es weitere C-Funktionen zur Speicherverwaltung:

- `void* calloc(size_t n, size_t size)` beschafft Speicher für ein Array mit `n` Elementen jeweils der Größe `size`. Es wird ein Zeiger auf den Beginn des Arrays zurückgegeben bzw. NULL, wenn kein Speicher zugewiesen werden kann.
- `void* realloc(void* ptr, size_t size)` beschafft neuen Speicher der Größe `size`, kopiert maximal `size` Bytes des Objekts, das sich an der Adresse `ptr` befindet, in den

neuen Speicherbereich, gibt den Speicher bei `ptr` frei und gibt einen Zeiger auf den neu angelegten Speicher zurück. Anders gesagt, für den typischen Fall, dass `size` der Größe des Objekts entspricht: Das Objekt wird im Speicher verschoben, und der vorher belegte Platz wird freigegeben.

9.7.2 Unterscheidung zwischen Heap- und Stack-Objekten

Kann ein Objekt wissen, ob es sich auf dem Heap oder auf dem Stack befindet? Normalerweise nicht, aber wenn `new` und `delete` überladen werden, ergibt sich eine Möglichkeit. Im folgenden Beispiel werden die Objekte gezählt, wobei Heap- und Stack-Objekte unterschieden werden. Der »Trick« besteht darin, dass der `new`-Operator den erhaltenen Zeiger in eine Liste ein- und `delete` ihn wieder austrägt.

Wozu eine eigene Buchführung über Heap-Objekte?

Die werden Sie im Allgemeinen nicht benötigen. Die hier vorgestellte Technik ist jedoch interessant, weil sie in ähnlicher Form von manchen Bibliotheken eingesetzt wird. Die Qt-Bibliothek für grafische Benutzungsoberflächen (Abschnitt 14.2) verwaltet Heap-Objekte in einer Baumstruktur (nicht über den `new`-Operator, sondern über Konstruktoren) und sorgt dafür, dass bei Löschen eines Knotens mit `delete` alle davon abhängigen Knoten (Kind-Knoten) automatisch mit gelöscht werden. Ein Aufruf von `delete` für Kind-Knoten ist damit nicht mehr notwendig.

Im folgenden Beispiel wird statt einer Liste die Klasse `set` von Seite 787 verwendet. Die Feststellung, ob sich ein Element in einem Container befindet, wird von einem `set`-Container schneller beantwortet. Die Methode `count(x)` gibt die Anzahl der enthaltenen Elemente `x` zurück; bei einem Set kann das Ergebnis nur 0 oder 1 sein. Das Attribut `istHeapObjekt` wird entsprechend gesetzt. Die Klasse `Person` unseres Beispiels erbt nun nicht mehr von der Klasse `Objekt`, sondern von der Klasse `ZaehlendesObjekt`:

Listing 9.15: Klasse ZaehlendesObjekt

```
// cppbuch/k9/new/zaehlend/ZaehlendesObjekt.h
#ifndef ZAEHLENDESOBJEKT_H
#define ZAEHLENDESOBJEKT_H
#include<iostream>
#include<string>
#include<set>

class ZaehlendesObjekt {
public:
   ZaehlendesObjekt() {
      ++gesamt;
      istHeapObjekt = (1 == objekte.count(this));
      std::cout << (istHeapObjekt ? "Heap" : "Stack")
                << "-Objekt " << this << " erzeugt." << std::endl;
   }

   virtual ~ZaehlendesObjekt() {
      --gesamt;
      std::cout << (istHeapObjekt ? "Heap" : "Stack")
```

```
            << "–Objekt " << this << " zerstört." << std::endl;
  }

  static void status() {
    std::cout << "Es gibt " << gesamt << " Objekt(e), davon "
              << objekte.size() << " Heap–Objekt(e)" << std::endl;
  }

  static void *operator new(size_t size) {
    void* ptr = ::operator new(size);
    objekte.insert(ptr);
    return ptr;
  }

  static void operator delete(void *ptr) {
    objekte.erase(ptr);
    ::operator delete(ptr);
  }
private:
  bool istHeapObjekt;
  static std::set<void*> objekte;
  static int gesamt;
};
#endif
```

Die Instanziierung der statischen Variablen ist in der Implementierungsdatei:

Listing 9.16: Instanziierung statischer Variablen

```
// cppbuch/k9/new/zaehlend/ZaehlendesObjekt.cpp
#include"ZaehlendesObjekt.h"
std::set<void*> ZaehlendesObjekt::objekte;
int ZaehlendesObjekt::gesamt = 0;
```

Der `new`-Operator trägt den zurückzugebenden Zeiger in den Set `objekte` ein. Der direkt anschließend ausgeführte Konstruktor zählt die Gesamtzahl aller Objekte hoch und stellt fest, ob die eigene Adresse `this` im Set enthalten ist.

Der Destruktor vermindert die Gesamtzahl um eins. Der `delete`-Operator trägt den Zeiger des zu löschenden Heap-Objekts aus dem Set aus. Die Methode `status()` gibt die Gesamtzahl aller Objekte und den Anteil der Heap-Objekte aus. Letzter ergibt sich durch die Abfrage des Sets nach der Anzahl noch enthaltener Elemente mit der Methode `size()`.

In der Klasse `ZaehlendesObjekt` wird auf die Operatoren `new[]` und `delete[]` verzichtet. Man könnte die Anzahl der `new[]`-Aufrufe auf dieselbe Art ermitteln und auch ein fehlendes `delete[]` entdecken (siehe unten). Der Konstruktor könnte aber nicht mehr wie oben herausfinden, ob das Objekt ein Heap-Objekt ist, weil in dem Set nur der Beginn des Arrays abgelegt wäre, nicht jedoch die Adressen der einzelnen Array-Elemente.

9.7.3 Fehlende delete-Anweisung entdecken

Die Zählung der Heap-Objekte entsprechend dem vorhergehenden Abschnitt erlaubt es, fehlende `delete`-Anweisungen und damit verwitwete Objekte zu entdecken. Dazu wird

ein globales Objekt des Typs `Waechter` hinzugefügt (siehe Beispiel im Verzeichnis *cppbuch/k9/new/waechter*):

```
// cppbuch/k9/new/waechter/ZaehlendesObjekt.cpp
#include"ZaehlendesObjekt.h"
#include"Waechter.h"                          // neu
std::set<void*> ZaehlendesObjekt::objekte;
int ZaehlendesObjekt::gesamt = 0;
Waechter w;                                   // neu
```

Weil dieses Objekt global ist, wird sein Destruktor erst nach dem Ende von `main()` ausgeführt. Der Destruktor überprüft, ob der Set der Zeiger auf Heap-Objekte leer ist:

Listing 9.17: Klasse Waechter

```
// cppbuch/k9/new/waechter/Waechter.h
#ifndef WAECHTER_H
#define WAECHTER_H
#include"ZaehlendesObjekt.h"

class Waechter {
public:
   ~Waechter() {
      if(ZaehlendesObjekt::objekte.size() > 0) {
         std::cerr << "Es fehlen " << ZaehlendesObjekt::objekte.size()
                   << " delete-Anweisungen!" << std::endl;
      }
   }
};
#endif
```

Falls nicht, gibt es eine Fehlermeldung, und man kann sich auf die Suche begeben, an welcher Stelle die `delete`-Anweisung fehlt. Weil die Klasse `Waechter` auf das private Attribut `objekte` zugreift, wird der Zugriff erlaubt, indem `friend class Waechter;` in die Klassendefinition von `ZaehlendesObjekt` eingefügt wird. Die Wirkung wird sichtbar, wenn die Anweisung `delete ptr1;` in `main()` entfernt oder auskommentiert wird.

Listing 9.18: Fehlendes delete entdecken

```
// cppbuch/k9/new/waechter/main.cpp
#include"Person.h"
using namespace std;

int main() {
   Person person("Lena");
   cout << "Name : " << person.getName() << endl;
   ZaehlendesObjekt::status();
   Person* ptr1 = new Person("Jens");
   cout << "Name : " << ptr1->getName() << endl;
   ZaehlendesObjekt::status();
   delete ptr1;           // ggf. kommentieren, siehe Text
   ZaehlendesObjekt::status();
}
```

9.7.4 Eigene Speicherverwaltung

Eine universelle eigene Speicherverwaltung zu schreiben, die effizienter als die vorhandene ist, ist ein schwieriges Unterfangen. Weil die normale C++-Speicherverwaltung sehr gut ist, ist so ein Vorhaben auch *nicht empfehlenswert*. Nur für spezielle Zwecke kann eine eigene Speicherverwaltung sinnvoll sein, um zum Beispiel Fehler zu finden oder unter günstigen Umständen sehr schnell sein zu können. Im Folgenden werden solche günstigen Umstände vereinfachend vorausgesetzt. Die maximal von einem Programm benötigte Anzahl von Objekten (d.h. die Menge an Speicher) sei von vornherein bekannt. In so einem Fall kann der Speicher vorher statisch allokiert werden. Dies ist besonders bei eingebetteten Systemen (englisch *embedded systems*) wichtig, denen nur ein beschränkter Speicher zur Verfügung steht.

Ausrichtung an Speichergrenzen

Oben werden bestimmte Fälle wie das Löschen eines Null-Zeigers oder das Ausrichtung (englisch *alignment*) an Speichergrenzen von den benutzten Systemfunktionen automatisch gehandhabt. Mit der Ausrichtung an Speichergrenzen ist gemeint, dass Zeiger systemabhängig nur auf Speicheradressen mit bestimmten Eigenschaften verweisen dürfen. So kann es sein, dass eine Speicheradresse, an der ein `int`-Wert beginnt, durch 4 teilbar sein muss (falls 4 = `sizeof(int)` ist). Ein `double`-Wert kann systemabhängig an einer durch 8 oder durch 4 teilbaren Adresse beginnen. Der Grund liegt in der gewünschten hohen Verarbeitungsgeschwindigkeit. Wenn ein `int`-Wert erst aus einem Bereich »krummer« Byte-Adressen extrahiert werden muss, kostet das viel Laufzeit. Der Compiler sorgt dafür, dass Objekte diesen Anforderungen entsprechend angelegt werden. Sie können das leicht sehen, indem Sie sich `sizeof(char)` und `sizeof(double)` ausgeben lassen (auf meinem System 1 und 8), und dann eine aus beiden zusammengesetzte Struktur bilden:

```
struct SizeofTest {
   char x;
   double d;
};
```

`sizeof(SizeofTest)` ergibt auf meinem System 12 statt 9 – ein Hinweis, dass einige nach `x` liegende Bytes zugunsten eines schnelleren Zugriffs auf `d` verschenkt werden. Bei der eigenen Verwaltung von Speicherplatz müssen diese Anforderungen berücksichtigt werden, sofern der Compiler nicht dafür sorgt. In C++ ist eine Funktion `alignof(Typ)` vorgesehen, die die Stückelung des Speichers zurückgibt. `alignof(SizeofTest)` ergibt auf meinem System 4. Es könnten daher vier statt einer `char`-Variablen in der Struktur ohne weiteren Speicherbedarf untergebracht werden. Erst bei einer fünften `char`-Variablen würde der Speicherbedarf der Struktur um 4 auf 16 Bytes wachsen.

Konsequenzen für die Verwaltung des Speichers

Die oben vorgeschlagene Oberklasse `Objekt` zur Speicherverwaltung weiß nicht, von welchem Typ die Unterklasse ist, die die geerbten Funktionen `new` und `delete` aufruft. Damit gibt es keine Möglichkeit zur Bestimmung der Speicherausrichtung. Die Konsequenz: Die überladenen Operatoren `new` und `delete` müssen in die Anwendungsklasse verlegt werden, weil dort das Alignment bestimmt werden kann. Von dieser Klasse darf aus dem genannten Grund nicht geerbt werden. Dies sind natürlich Einschränkungen. Anderer-

seits sollte die Verwaltung selbst bereitgestellten Speichers ohnehin nur eine Ausnahme bleiben; normalerweise genügen `::new` und `::delete`. Zur Demonstration wird die Klasse `Person` entsprechend angepasst:

- Das Attribut `name` ist jetzt vom Typ `char[15]`. Damit liegt der Speicherbedarf fest, wie für ein eingebettetes System sinnvoll. Der Typ `string` impliziert `new`-Operationen und wird deshalb nicht gewählt. Der beschriebene Speicherverwaltungsmechanismus würde mit `string` auch funktionieren, nur wäre zusätzlicher Heap-Speicher notwendig, dessen Menge wegen der unvorhersehbaren Länge eines Strings nicht bezifferbar ist. Im Beispiel wird jeder String auf 14 Zeichen + Null-Byte gekürzt.
- Der Destruktor ist nicht `virtual`, weil von dieser Klasse nicht geerbt werden soll.
- Die Speicherverwaltung wird in eine eigene Klasse verlegt, die als Template gestaltet und damit auch für andere Typen nutzbar ist. Alle Funktionen sind `static`, weil die Klasse für alle `Person`-Objekte zuständig ist.
- Basis für die Speicherstückelung sind nicht die einzelnen Attribute der Klasse, sondern `sizeof(Person)`. Damit sorgt der Compiler für die richtige Ausrichtung.

Listing 9.19: Klasse Person mit eigener Speicherverwaltung

```
// Auszug aus cppbuch/k9/new/eigene/Person.h
#ifndef PERSON_H_
#define PERSON_H_
#include<algorithm>    // std::min()
#include<string>
#include<cstring>
#include"Speicherverwaltung.t"

class Person {
public:
   typedef Speicherverwaltung<Person, 100> Speicher;

   Person(const std::string& n = "N.N.") {
      strncpy(name, n.c_str(), sizeof(name)-1);
      size_t ende = std::min(n.length(), sizeof(name)-1);
      name[ende] = '\0';
   }

   const std::string getName() const {
      return name;
   }

   static void *operator new(size_t size) {
      return Speicher::getMemory(size);
   }

   static void operator delete(void* ptr) {
      Speicher::freeMemory(ptr);
   }

   static int freiePlaetze() { // gibt die Anzahl noch verfügbarer Plätze zurück
      return Speicher::freiePlaetze();
   }
```

```
private:
   char name[15];
};
#endif
```

Mit `typedef` werden Typ und maximale Anzahl der Objekte an einer Stelle festgelegt. Der Alias `Speicher` kann an jeder Stelle abkürzend anstelle des vollständigen Templatenamens angegeben werden. Der Konstruktor kopiert maximal `sizeof(name)-1` Zeichen und setzt das abschließende Null-Byte.

Die Klasse `Speicherverwaltung` hat nur statische Variablen. Wie aus Abschnitt 6.2 bekannt, geschieht die Initialisierung und Definition der klassenspezifischen Variablen außerhalb der Klassendeklaration. Bei statischen Template-Attributen braucht es keine Implementierungsdatei; die Definitionen werden nach der Klassendeklaration aufgeführt, wie unten gezeigt.

- `speicher` ist ein fixes `char`-Array, in dem die Objekte abgelegt werden. Weil `T` den Typ und `N` die maximale Anzahl der Objekte darstellen, beträgt die Anzahl der Arrayelemente `sizeof(T)*N`.
- `belegt` ist das Array zur Speicherverwaltung. Für jedes der `N` möglichen Objekte wird dort seine Adresse eingetragen. Das Array ist durch die Art der Speicherverwaltung stets dicht belegt; es gibt keine Lücken (siehe unten).
- `ersterFreierPlatz` gibt die Stelle im Array `speicher` an, wo das nächste Objekt eingetragen werden kann. `ersterFreierPlatz` verweist stets auf die Position direkt nach dem Ende des belegten Bereichs.

Listing 9.20: Klasse Speicherverwaltung

```
// cppbuch/k9/new/eigene/Speicherverwaltung.h
#ifndef SPEICHERVERWALTUNG_T
#define SPEICHERVERWALTUNG_T
#include<new>                  // bad_alloc
#include<stdexcept>            // invalid_argument

template<typename T, int N>
class Speicherverwaltung {
public:
   static void* getMemory(size_t size) {
      if(ersterFreierPlatz >= N) {
         throw std::bad_alloc();
      }
      if(size != sizeof(T)) {
         throw std::invalid_argument("getMemory(): Aufruf fuer falschen Typ!");
      }
      belegt[ersterFreierPlatz] = &speicher[sizeof(T)*ersterFreierPlatz];
      return belegt[ersterFreierPlatz++];
   }

   static void freeMemory(void* ptr) {
      if(ptr) {                // Null-Zeiger ignorieren
         for(int i = ersterFreierPlatz-1; i >=0; --i) {   // Stelle suchen
            if(belegt[i] == ptr) {
               // 'freimachen' durch Kopieren des letzten Eintrags
```

```
            // an diese Stelle und Anpassen von ersterFreierPlatz
            belegt[i] = belegt[--ersterFreierPlatz];
            break;
          }
        }
      }
    }

   static int freiePlaetze() {
     return N - ersterFreierPlatz;
   }
private:
   static int ersterFreierPlatz;
   static void* belegt[N];
   static char speicher[sizeof(T)*N];
};

// Definitionen
template<typename T, int N>
int Speicherverwaltung<T, N>::ersterFreierPlatz = 0;

template<typename T, int N>
void* Speicherverwaltung<T, N>::belegt[N];

template<typename T, int N>
char Speicherverwaltung<T, N>::speicher[sizeof(T)*N];
#endif
```

Die Funktion `getMemory()` überprüft zunächst, ob überhaupt noch Platz vorhanden ist. Falls nicht, wird `bad_alloc` geworfen. Falls unzulässigerweise von der Klasse `T` abgeleitet und `new` für die abgeleitete Klasse aufgerufen wird, reagiert `getMemory()` mit einer `invalid_argument`-Exception. Voraussetzung für die Erkennung ist, dass die abgeleitete Klasse zusätzliche Attribute zu den geerbten hat. Wenn alles klar geht, wird die nächste freie Adresse in das Feld `belegt` eingetragen und zurückgegeben. `ersterFreierPlatz` wird auf die nächste Position hochgezählt. Weil nicht erst eine freie Stelle gesucht werden muss, ist `getMemory()` sehr schnell.

`getMemory()` dauert hingegen geringfügig länger, weil die Stelle gesucht wird, wo die Adresse `ptr` eingetragen wurde. Zuletzt angelegte Objekte werden nicht immer, aber häufig zuerst wieder gelöscht. Deshalb beginnt die Suche am Ende des Bereichs. Wenn der Eintrag gefunden wird, kann er gelöscht werden. Statt einen Null-Zeiger einzutragen, ist es besser, den letzten Verweis an diese Stelle zu kopieren und `ersterFreierPlatz` herunterzuzählen. Dadurch werden zukünftige Suchvorgänge verkürzt, und für `new` ist gar kein Suchvorgang erforderlich.

9.7.5 Empfehlungen im Umgang mit new und delete

`new` und `delete` gehören zusammen. Daraus ergeben sich einige Empfehlungen:

- Überladen Sie `new` und `delete` *nicht*, es sei denn, Sie haben gute Gründe dafür. Wenn die mangelnde Performance wegen vieler `new`-Aufrufe ein Grund sein sollte, wäre zuerst zu überlegen, ob die Anzahl nicht reduziert werden kann, zum Beispiel

durch Wiederverwendung bereits angelegter Objekte. Sie könnten mit einer passenden Funktion neu initialisiert werden.

- Wenn Sie `new` überladen, überladen Sie auch `delete` und umgekehrt. Dasselbe gilt für `new[]` und `delete[]`.
- Überladene `delete`- und `delete[]`-Operatoren dürfen keine Exception werfen. Beide Funktionen werden oft innerhalb eines Destruktors aufgerufen und rufen selbst Destruktoren auf. Wenn im ersten Fall eine Exception geworfen würde, wäre der Destruktor nicht mehr exception-sicher. Einzelheiten dazu finden Sie in Abschnitt 20.2.7.
- Verwenden Sie nur zusammenpassende Operatoren und Funktionen. Das heißt, dass ein mit `new` angelegtes Objekt nicht mit `free()` freigegeben werden darf. `delete` darf nicht auf einen mit `malloc()` erzeugten Speicherbereich angewendet werden.
- Falls Sie die Speicherbeschaffung in eine Oberklasse verlagern, vergessen Sie nicht, den Destruktor der Oberklasse als `virtual` zu deklarieren.
- Bei der Verwaltung selbst bereitgestellten Speichers darf die Ausrichtung an Speichergrenzen nicht vergessen werden, ebenso nicht, dass `delete NULL` keine Auswirkung haben darf.

9.8 Mehrdimensionale Matrizen[2]

Neben eindimensionalen Feldern sind 2- und 3-dimensionale Matrizen in mathematischen Anwendungen verbreitet. Der Sprache C entsprechende ein- und mehrdimensionale Arrays in C++ sind aus den Abschnitten 5.4 und 5.7 bekannt. Mathematische Matrizen sind Spezialfälle von Arrays mit Elementen, die vom Datentyp `int`, `float`, `complex`, `rational` oder ähnlich sind. Die Klasse `MathVektor` (Abschnitt 9.2.3) ist eine eindimensionale Matrix in diesem Sinne, wobei die Klasse im Unterschied zu einem normalen C-Array einen sicheren Zugriff über den Indexoperator erlaubt, wie wir dies auch für zwei- und mehrdimensionale Matrixklassen erwarten. Der Zugriff auf Elemente eines ein- oder mehrdimensionalen Matrixobjekts sollte

- sicher sein durch eine Prüfung aller Indizes und
- über den Indexoperator (bzw. `[][]`, `[][][]` ...) erfolgen, um die gewohnte Schreibweise beizubehalten.

Das Überladen des Klammeroperators für runde Klammern () wäre alternativ möglich, entspräche aber nicht der üblichen Syntax. Nun kann man sich streiten, ob es ästhetischer ist, `a(5,6)` zu schreiben anstatt `a[5][6]`. Sicherlich ist es bei neu zu schreibenden Programmen gleichgültig, doch wenn man für die Wartung und Pflege von existierenden großen Programmen verantwortlich ist, die die `[]`-Syntax verwenden? Ein weiteres Argument: Eine Matrixklasse soll sich möglichst ähnlich wie ein konventionelles C-Array

[2] Dieser Abschnitt kann beim ersten Lesen übersprungen werden. Es geht darum, wie die Aufeinanderfolge verketteter Indexoperatoren `[][][]`, wie sie bei Matrizen üblich ist, mit dem überladenen Indexoperator realisiert wird, nicht um maximal effiziente Matrizen.

verhalten. Die auftretenden verketteten Aufrufe des Indexoperators werden im Einzelnen analysiert. Auch sollen die Indexoperatoren sicher sein. Etwas effizientere zweidimensionale Arrays, das heißt ohne Indexprüfung, werden an anderer Stelle behandelt (Seiten 215 und besonders 697 ff.).

Der Verzicht auf die erste Anforderung (Indexprüfung) wird oft mit Effizienzverlusten begründet. Dieses Argument ist nicht immer stichhaltig:

- Es ist wichtiger, dass ein Programm korrekt anstatt schnell ist. Indexfehler, das zeigt die industrielle Praxis, treten häufig auf. Das Finden der Fehlerquelle ist schwierig, wenn mit falschen Daten weitergerechnet und der eigentliche Fehler erst durch Folgefehler sichtbar wird.
- Der erhöhte Laufzeitbedarf durch einen geprüften Zugriff ist oft durchaus vergleichbar mit den weiteren Operationen, die mit dem Array-Element verbunden sind, und manchmal vernachlässigbar. Im Bereich der Natur- und Ingenieurwissenschaften gibt es einige Programme, bei denen sich die Indexprüfung deutlich nachteilig auswirkt, andererseits kommt es auf den Einzelfall an. Nur wenn ein Programm *wegen* der Indexprüfung zu langsam ist, sollte man nach gründlichen Tests erwägen, die Prüfung herauszunehmen.

Für rechenzeitoptimierte Vektoren und Matrizen sei auf entsprechende Bibliotheken verwiesen, von denen einige auf [OON] gelistet sind. C++ stellt die Klasse `valarray` zur Verfügung (Abschnitt 34, Seiten 857 ff). Zugunsten starker Optimierungsmöglichkeiten wurden beim Entwurf der Klasse `valarray` bewusst Einschränkungen bezüglich der Verständlichkeit und leichter Benutzbarkeit hingenommen. Eine schnelle 2-dimensionale Matrix wird in Abschnitt 24.10.1 diskutiert.

9.8.1 Zweidimensionale Matrix als Vektor von Vektoren

Was ist eine zweidimensionale Matrix, deren Elemente vom Typ `int` sind? Eine `int`-Matrix *ist ein* Vektor von mathematischen `int`-Vektoren! Diese Betrachtungsweise erlaubt eine wesentlich elegantere Formulierung einer Matrixklasse im Vergleich zur Aussage: Die Matrix *hat* beziehungsweise *besitzt* mathematische `int`-Vektoren. Die Formulierung der *ist-ein*-Relation als Vererbung zeigt die Klasse `Matrix`, wobei die bereits bekannten Klassen `vector` und `MathVektor` eingesetzt werden:

Listing 9.21: Matrix-Template

```
//cppbuch/k9/matrix/matrix.t
#ifndef MATRIX_T
#define MATRIX_T
#include"../mathvek/mvektor.t"

// Matrix als Vektor von MathVektoren
template<typename T>
class Matrix : public Vektor<MathVektor<T> > {
  protected:
    size_t yDim;
  public:
    typedef Vektor<MathVektor<T> > super;       // Oberklassentyp
    Matrix(size_t = 0, size_t = 0);             // Zeilen, Spalten
    size_t zeilen() const {return super::size(); } // Zu super siehe Seite 332.
```

```
    size_t spalten() const {return yDim; }
    void init(const T&);
    // mathematische Operatoren und Funktionen
    Matrix<T>& I(); // als Einheitsmatrix initialisieren
    Matrix<T>& operator*=(const T&);
    Matrix<T>& operator*=(const Matrix<T>&);
    void swap(Matrix<T>& rhs);        // Vertauschen
    // ... weitere Operatoren und Funktionen
};

template<typename T> // Implementierung des Konstruktors:
Matrix<T>::Matrix(size_t x, size_t y)
: Vektor<MathVektor<T> >(x), yDim(y) {
    MathVektor<T> temp(y);
    for(size_t i = 0; i < x; ++i) {
       super::operator[](i) = temp;
    }
}   // ... Fortsetzung folgt unten
```

Die Klasse `Matrix` erbt also von der Klasse `Vektor`, wobei jetzt der Datentyp der Vektorelemente durch ein Template beschrieben wird. Der Konstruktor initialisiert ein Basisklassensubobjekt des Typs `vector<MathVektor<T> >` mit der richtigen Größe `x`. Weil dem Basisklassensubobjekt in der Initialisierungsliste die zweite Dimension `y` nicht bekannt ist, legt es `xDim` eigene Vektorelemente vom Typ `MathVektor<T>` mit der Länge 0 an. Die korrekte Länge `yDim` aller Vektorelemente wird in der nachfolgenden Schleife durch Zuweisung eines temporären Objekts `temp` mit der richtigen Länge `y` erreicht.

`Matrix` hat keinerlei dynamische Daten außerhalb des Basisklassensubobjekts. Deshalb sind weder ein besonderer Destruktor, Kopierkonstruktor noch ein eigener Zuweisungsoperator notwendig. Die entsprechenden Operationen für das Basisklassensubobjekt werden von der `vector`-Klasse erledigt. Allerdings soll die Klasse `Matrix` nicht über Zeiger benutzt werden, zum Beispiel:

```
Matrix<double> *pMatrix = new Matrix<double>(10, 10);
// ... Benutzung ...
delete pMatrix;       // Speicherleck!
```

Der Grund liegt darin, dass die Klasse `vector` wie alle Container-Klassen der C++-Standardbibliothek keinen virtuellen Destruktor hat.

In Abschnitt 7.6.3 (Seite 280) finden Sie eine Begründung für den Fehler mit Beispiel.

Der Indexoperator wird geerbt; wie er funktioniert, wird unten beschrieben. Die Initialisierung der Matrix mit Werten (siehe Übungsaufgabe), der Kurzformoperator `*=` für die Multiplikation mit einem Skalar und die Erzeugung einer Einheitsmatrix sind ebenfalls einfach zu implementieren, wobei im `*=`-Operator geerbter Code wiederverwendet wird:

```
template<typename T>                  // Multiplikationsoperator
Matrix<T>& Matrix<T>::operator*=(const T& faktor) {
  for(size_t i = 0; i < super::size(); ++i) {
    super::operator[](i) *= faktor; // MathVektor<T>::operator*=()
  }
```

```
  return *this;
}

template<typename T>              // Einheitsmatrix
Matrix<T>& Matrix<T>::I() {   // keine Prüfung auf size() == yDim
   for(size_t i = 0; i < super::size(); ++i) {
     for(size_t j = 0; j < yDim ; ++j) {
       super::operator[](i)[j] = (i==j) ? T(1) : T(0);
     }
   }
   return *this;
}
// weitere Funktionen ...
#endif          // MATRIX_T
```

Nach diesem Schema können weitere Operatoren und Funktionen gebaut werden (siehe Übungsaufgaben). Nun fragt man sich, wie der Elementzugriff und die Indexprüfung in diesem Fall funktionieren? Betrachten wir folgendes Beispiel:

```
Matrix<float> matrix2D(100,55);
matrix2D[3][7] = 1.0162;
```

Der Zugriff ist sehr einfach, beide Indizes werden überprüft. Die Erklärung der Wirkungsweise ist jedoch nicht ganz so einfach. Um zu sehen, was geschieht, schreiben wir `matrix2D[3][7]` um und lösen dabei die Funktionsaufrufe auf:

```
(matrix2D.operator[](3)).operator[](7)
```

Die erste, von der Oberklasse geerbte Operatormethode ist

```
Vektor<MathVektor<float> >::operator[](int)
```

Das anonyme Basisklassensubobjekt ist ein `Vektor`, dessen `[]`-Operator mit dem Argument 3 aufgerufen wird. Die Elemente des Vektors sind vom Typ `mathVektor<float>`; zurückgegeben wird also eine Referenz auf den dritten (d.h. eigentlich vierten wegen der Zählung ab 0) `mathVektor` des Vektors. Bezeichnen wir den Rückgabewert zur Vereinfachung mit `X`, dann wird jetzt

```
X.operator[](7)
```

ausgeführt, was nichts anderes bedeutet, als die Indexoperation `operator[]()` für einen `mathVektor<float>` mit dem Ergebnis `float&` – also einer Referenz auf das gesuchte Element – auszuführen. In jedem dieser Aufrufe von Indexoperatoren werden Grenzen auf einheitliche Weise geprüft. Abgesehen von der äquivalenten Definition für konstante Objekte existiert nur eine *einzige* Definition des Indexoperators! Auf die Definition weiterer Operatoren und sinnvoller Elementfunktionen soll nicht eingegangen werden.

Übungen

9.16 Implementieren Sie einen Operator

```
Matrix<T>& Matrix::operator*=(const Matrix<T>&)
```

zur Multiplikation einer Matrix mit einer anderen.

9.17 Implementieren Sie einen binären Operator zur Multiplikation zweier zweidimensionaler Matrizen unter der Annahme, dass bereits ein Elementoperator `*=` existiert.

9.18 Schreiben Sie eine Methode `void init(const T&)` zur Initialisierung aller Matrixelemente. Sind dabei Methoden der Oberklasse einsetzbar?

9.8.2 Dreidimensionale Matrix

Das für zweidimensionale Matrizen benutzte Schema lässt sich nun zwanglos für Matrizen beliebiger Dimension erweitern. Hier sei nur noch abschließend das Beispiel für die dritte Dimension gezeigt. Was ist eine dreidimensionale Matrix, deren Elemente vom Typ `int` sind? Die Frage lässt sich leicht in Analogie zum vorhergehenden Abschnitt beantworten. Eine dreidimensionale `int`-Matrix *ist ein* `Vektor` von mathematischen zweidimensionalen `int`-Matrizen! Die Formulierung der *ist-ein*-Relation als Vererbung zeigt die Klasse `Matrix3D`:

Listing 9.22: 3-dimensionale Matrix

```
template<typename T>
class Matrix3D : public Vektor<Matrix<T> > {
  private:
    size_t yDim, zDim;
  public:
    typedef Vektor<Matrix<T> > super; // Oberklassentyp
    Matrix3D(size_t = 0, size_t = 0, size_t = 0);
    size_t xDIM() const { return super::size();}
    size_t yDIM() const { return yDim;}
    size_t zDIM() const { return zDim;}
    void init(const T&);            // Initialisierung
    Matrix3D<T>& I();               // Einheitsmatrix
    void swap(Matrix3D<T>& rhs);    // Vertauschen
    // mathematischer Operator:
    Matrix3D<T>& operator*=(const T&); // Multiplikation
    // weitere Operatoren und Funktionen ...
};

template<typename T>       // Implementierung des Konstruktors
Matrix3D<T>::Matrix3D(size_t x, size_t y, size_t z)
: Vektor<Matrix<T> >(x), yDim(y), zDim(z) {
    Matrix<T> temp(y, z);
    for(size_t i = 0; i < x; ++i) {
       super::operator[](i) = temp;
    }
}
```

Der Konstruktor initialisiert ein Basisklassensubobjekt des Datentyps `Vektor< Matrix<T> >` mit der richtigen Größe `x`. Weil dem Basisklassensubobjekt in der Initialisierungsliste die zweite und dritte Dimension nicht bekannt sind, legt es `xDim` eigene Elemente vom Typ `Matrix<T>` mit der Größe 0 mal 0 an. Die korrekte Größe aller Subobjekte wird in der nachfolgenden Schleife durch Zuweisung eines temporären Objekts `temp` erreicht. Weil `Matrix3D` (wie `Matrix`) keinerlei dynamische Daten außerhalb des Basisklassensubobjekts hat, ist weder ein besonderer Destruktor, Kopierkonstruktor noch ein eigener Zuweisungsope-

rator notwendig. Die entsprechenden Operationen für das Basisklassensubobjekt werden von der Klasse `Vektor` selbst erledigt. Der Indexoperator wird geerbt. Die Initialisierung, der Kurzformoperator `*=` und die Erzeugung einer Einheitsmatrix sind ähnlich wie oben beschrieben zu implementieren, wobei im `*=`-Operator geerbter Code wieder verwendet wird:

```
template<typename T>
void Matrix3D<T>::init(const T& wert) { // Initialisierung
    for(size_t i = 0; i < super::size(); ++i)
      for(size_t j = 0; j < yDim ; ++j)
        for(size_t k = 0; k < zDim ; ++k)
            super::operator[](i)[j][k] = wert;
}
```

Eine Alternative zur Initialisierung der einzelnen Elemente wie oben ist die Ausnutzung einer Methode der Klasse `Matrix`:

```
template<typename T>
void Matrix3D<T>::init(const T& wert) { // alternative Initialisierung
    for(size_t i = 0; i < super::size(); ++i) {
       super::operator[](i).init(wert);
       // operator[](i) ist vom Typ Matrix<T>
    }
}
```

Das kann ebenso vorteilhaft für die Bildung einer Einheitsmatrix ausgenutzt werden:

```
template<typename T>                          // Einheitsmatrix
Matrix3D<T>& Matrix3D<T>::I() { // keine Prüfung auf gleiche Dimensionen
   for(size_t i = 0 ; i< super::size(); ++i) {
      super::operator[](i).I();
   }
   return *this;
}

template<typename T>                        // Multiplikationsoperator
Matrix3D<T>& Matrix3D<T>::operator*=(const T& faktor) {
  for(size_t i = 0; i < super::size(); ++i)
    super::operator[](i) *= faktor; // Matrix<T>::operator*=()
  return *this;
}
#endif   // matrix_t
```

Nun können auf einfache Art dreidimensionale Matrizen definiert und benutzt werden, zum Beispiel:

```
Matrix3D<int> matrix3D(3,15,2);
for(size_t i = 0 ; i< matrix3D.xDIM(); ++i)
   for(size_t j = 0 ; j < matrix3D.yDIM() ; ++j)
      for(size_t k = 0 ; k < matrix3D.zDIM() ; ++k) {
          matrix3D[i][j][k] = was_auch_immer();
          cout << matrix3D[i][j][k];
      }
```

```
// Benutzung
matrix3D *= 1020;
// .... usw.
```

Die Wirkungsweise des Indexoperators ist in Analogie zur Klasse `Matrix` beschreibbar, es gibt nur einen verketteten Operatoraufruf mehr. Wir formulieren `matrix3D[1][2][3]` um und erhalten: `matrix3D.operator[](1).operator[](2).operator[](3)`

Der erste Operator gibt etwas vom Typ `Matrix<int>&` zurück oder genauer eine Referenz auf das erste Element des `Vektor`-Subobjekts von `matrix3D` (die dreidimensionale Matrix ist ein Vektor von zweidimensionalen Matrizen). Das zurückgegebene »Etwas« kürzen wir der Lesbarkeit halber mit `Z` ab und erhalten `Z.operator[](2).operator[](3)`

Sie wissen, dass eine Referenz nur ein anderer Name ist, sodass `Z` letztlich eine Matrix des Typs `Matrix<int>` repräsentiert. Sie sahen bereits, dass eine `Matrix<int>` ein Vektor ist, nämlich ein `Vektor< MathVektor<int> >`, von dem `operator[]()` geerbt wurde. Genau dieser Operator wird nun mit dem Argument 2 aufgerufen und gibt ein Ergebnis vom Typ `MathVektor<int>&` zurück, das hier der Kürze halber `X` heißen soll: `X.operator[](3)`

Der Rest ist leicht, wenn Sie an das Ende des Abschnitts über zweidimensionale Matrizen schauen. Auch hier ist wie bei der Klasse `Matrix` der Zugriff auf ein Element simpler als die darunterliegende Struktur.

Schlussbemerkung

Die Methode zur Konstruktion der Klassen für mehrdimensionale Matrizen kann leicht verallgemeinert werden: Eine n-dimensionale Matrix kann als Vektor von $(n-1)$-dimensionalen Matrizen aufgefasst werden. Die Existenz einer Klasse für $(n-1)$-dimensionale Matrizen sei dabei vorausgesetzt. In der Praxis werden jedoch vier- und höherdimensionale Matrizen selten eingesetzt. Indexoperator, Zuweisungsoperator, Kopierkonstruktor und Destruktor brauchen nicht geschrieben zu werden, sie werden von der Klasse `Vektor` zur Verfügung gestellt. Geschrieben werden müssen jedoch der Konstruktor, die Methoden zur Initialisierung und die gewünschten mathematischen Operatoren. Die beschriebenen Klassen zeigen die Kombination von Templates mit Vererbung. Beim Zugriff auf einzelne Matrix-Elemente wird kontrolliert, ob der Index im zulässigen Bereich liegt – das kostet natürlich Laufzeit.

9.9 Zuweisung bei Vererbung[3]

Wenn kein spezieller Zuweisungsoperator definiert ist, wird automatisch vom Compiler ein Zuweisungsoperator bereitgestellt, der ein Objekt elementweise kopiert. Elementweise Kopie meint, dass für jedes Element der zugehörige Zuweisungsoperator aufgerufen wird, sei er nun selbst definiert oder implizit generiert worden. Insbesondere wird bei einem Objekt einer abgeleiteten Klasse das anonyme Subobjekt der Oberklasse in diesem Sinn als Element aufgefasst. Dies ist alles einfach, wenn Polymorphismus aus dem Spiel bleibt.

[3] Dieser Abschnitt kann beim ersten Lesen übersprungen werden.

Der Tipp auf Seite 273 soll sicherstellen, dass sich die Bedeutung eines Methodenaufrufs nicht ändert, wenn auf eine Methode statt über den Objektnamen über Basisklassenzeiger bzw. -referenzen zugegriffen wird. Wie sieht es aus, wenn polymorphe Objekte einander zugewiesen werden sollen? Anhand der Klasse `Ober` und der abgeleiteten Klasse `Unter` soll das Problem verdeutlicht werden:

```
// Klassendefinitionen
class Ober {
   public:
     // ....
   private:
     int iO;                 // Daten der Klasse Ober
};

class Unter : public Ober {
   public:
     // ....
   private:
     int iU;                 // Daten der Klasse Unter
};
```

```
// main()-Programm, Auszug:
Unter U1, U2;
U1 = U2;          // kein Problem: Unter::operator=(const Unter&)
// polymorphe Objekte (statischer ≠ dynamischer Typ)
Ober& rO1 = U1;  // polymorphe Zuweisung
Ober& rO2 = U2;
```

Falls der Zuweisungsoperator nicht virtuell ist, werden nur statische Typen betrachtet:

```
Ober O1;
Ober O2;
O1  = O2;    // Ober::operator=(const Ober&) ok
```

Aus der Zuweisung

```
rO1 = U2;    // Ober::operator=(const Unter&) ?
```

versucht der Compiler, einen Zuweisungsoperator zu finden. In einer Oberklasse darf es so einen Operator nicht geben, weil nicht garantiert ist, dass in der Oberklasse Informationen über eine Unterklasse existieren. Der Compiler nimmt den nächst passenden Operator, nämlich `Ober::operator=(const Ober&)`, weil `rO1` vom statischen Typ `Ober` ist. Der Fehler: Dem Objekt `U1`, auf das die Referenz `rO1` verweist, wird *nur der Oberklassenanteil* von `U2` zugewiesen, das heißt, `U1` und `U2` unterscheiden sich nach der Zuweisung! Um den richtigen Zuweisungsoperator aufzurufen, wenn statischer und dynamischer Typ verschieden sind, muss der Zuweisungsoperator virtuell sein. Weil der implizit generierte Zuweisungsoperator niemals virtuell ist, heißt dies, dass sich der Softwareentwickler selbst dieser Mühe unterziehen muss.

Bei virtuellen Methoden müssen Name und Parameterlisten übereinstimmen, nur der Rückgabetyp darf etwas unterschiedlich sein: Wenn der Rückgabetyp einer virtuellen Funktion eine Referenz auf die Klasse ist, dann darf der Rückgabetyp der entsprechenden

Funktion in der abgeleiteten Klasse eine Referenz auf die abgeleitete Klasse sein. Ein Beispiel sind die Funktionen `vf4()` und `vf5()` auf Seite 274.

```
// Annahme: virtueller Zuweisungsoperator
O1  = O2;    // Ober::operator=(const Ober&) wie vorher
U1  = rO2;   // Unter::operator=(const Ober&)
rO1 = U2;    // Unter::operator=(const Ober&)
```

Leider tauchen bei der Implementierung nun weitere Probleme auf:

```
virtual Unter& Unter::operator=(const Ober& rs) {
                    // rs = rechte Seite der Zuweisung
    // geerbte Attribute der Oberklasse zuweisen:
    iO = rs.iO;        // Fehler! iO ist private.
    // Attribut der Klasse Unter zuweisen:
    iU = rs.iU;        // Fehler! rs.iU ist unbekannt!
    return *this;
}
```

Der erste Fehler kann leicht behoben werden: Wenn die Attribute der Oberklasse `protected` sind, ist die Zuweisung möglich. Nun wird dadurch der Zugriffsschutz aufgeweicht, und deshalb ist es besser, wenn die Oberklasse eine Methode zur lokalen Zuweisung ihrer Attribute bereitstellt, die die privaten Daten kopiert. Der zweite Fehler resultiert aus der Tatsache, dass der Compiler nur eine statische Analyse vornehmen kann: Danach hat das Objekt `rs`, das zur Klasse `Ober` gehört, kein Attribut `iU`. Benötigt wird jedoch zur Laufzeit die Information über den dynamischen Typ von `rs`, der sowohl `Ober` als auch `Unter` sein kann. Nur in letzterem Fall ist eine Zuweisung überhaupt sinnvoll.

Mit den bisher bekannten Mitteln ist dem Problem der Zuweisung bei Polymorphismus nicht beizukommen. Eine brutaler Downcast, also eine Typumwandlung der Art

```
    iU = static_cast<Unter&>(rs).iU; // gefährlich
```

verhindert zu erkennen, dass `rs` vielleicht doch den falschen Typ hat. Aus diesem Grund wurde der `dynamic_cast`-Operator eingeführt, der zur Laufzeit eine Typumwandlung vornimmt und dabei prüft, ob diese Typumwandlung erlaubt ist (siehe Abschnitt 7.9).

Man könnte denken, dass ein virtueller Zuweisungsoperator genügen würde. Das ist nicht der Fall, weil der Zuweisungsoperator anders als »normale« virtuelle Funktionen aufgerufen wird. Wenn nur ein virtueller Zuweisungsoperator geschrieben wurde, wird bei der Compilation der nicht-virtuelle automatisch auch noch erzeugt – und ggf. statt des virtuellen aufgerufen (siehe Abschnitt 13.5.3 in [ISOC++]). Das ist kein Problem, wenn die abgeleitete Klasse keinerlei dynamische Daten hat. Wenn es jedoch Zeigerattribute gibt, denen mit `new` etwas zugewiesen wird, ist ein selbstgeschriebener nicht-virtueller Zuweisungsoperator unumgänglich, weil der compilergenerierte nur den Zeiger kopieren würde, ohne Speicherplatz zu beschaffen. Im Zusammenspiel von nicht-virtuellem und virtuellem Zuweisungsoperator ergeben sich weitere Schwierigkeiten, auf die hier nicht näher eingegangen werden soll. Es gibt nur eine funktionierende Strategie, die im Folgenden beschrieben wird.

Bei dieser Strategie bekommt jede Klasse eine virtuelle Methode `assign()` und einen nicht-virtuellen `operator=()`. Bei polymorpher Zuweisung ruft der Zuweisungsoperator

`assign()` auf. Weil die Methode virtuell ist, wird sie automatisch passend zum Objekt aufgerufen.

Wenn dynamische Daten vorhanden sind, sind stets ein Kopierkonstruktor, ein Zuweisungsoperator und ein Destruktor erforderlich. Zur Demonstration hat jede Klasse des folgenden Beispiels sowohl ein Attribut vom Typ `int` als auch ein dynamisches Attribut vom Typ `char*` (C-String). Dem letzten Attribut wird mit `new` Speicherplatz zugewiesen, der im Destruktor freigegeben wird. Das Beispiel hat weitere Besonderheiten:

- Es gibt drei Klassen `A`, `B` und `C`. Dabei erbt `B` von `A`, und `C` erbt von `B`.
- Wegen der dynamischen Daten existiert für jede Klasse ein Kopierkonstruktor, ein Zuweisungsoperator und ein Destruktor.
- Der Zuweisungsoperator ruft in den abgeleiteten Klassen die virtuelle Methode `assign()`, in der die eigentliche Kopierarbeit geleistet wird. Damit der übergebene Typ stimmt, wird er mit `static_cast` umgewandelt. Ein `dynamic_cast` ist nicht notwendig, weil durch den polymorphen Aufruf der Methode `assign()` gewährleistet ist, dass der Typ zur Klasse passt.
- Jede Klasse hat eine swap-Methode, die nach bekanntem Muster innerhalb der Methode `assign()` eingesetzt wird.

Das `main`-Programm demonstriert mögliche Testfälle. Die Gleichheit der beteiligten Objekte wird mit `assert` geprüft. Dabei kommt der virtuelle Gleichheitsoperator `operator==()` zum Einsatz. Wegen der `virtual`-Eigenschaft ist er als Mitgliedsfunktion konzipiert. Das hat den Vorteil des direkten Zugriffs auf die Attribute, sodass getter-Funktionen in diesem Beispiel entfallen können.

Listing 9.23: main-Programm zum Testen der Zuweisungen

```
// cppbuch/k9/zuweisung_vererbung/zuwassign.cpp
#include<cassert>
#include<iostream>
#include"C.h"     // schließt A.h, B.h ein
using namespace std;

int main() {
  // Klasse A
  cout << "\nTest 1" << endl;
  A a1(1, "einsA");
  A a2(2, "zweiA");
  a1.ausgabe(); cout << endl;
  a1 = a2;
  cout << "a1 nach Zuweisung a1=a2:\n";
  a1.ausgabe(); cout << endl;
  assert(a1 == a2);

  // Klasse B
  cout << "\nTest 2" << endl;
  B b1(1, "einsA", 2, "einsB");
  B b2(3, "zweiA", 4, "zweiB");
  b1.ausgabe(); cout << endl;
  b1 = b2;
  cout << "b1 nach Zuweisung b1=b2:\n";
  b1.ausgabe(); cout << endl;
```

```
  assert(b1 == b2);

  cout << "\nTest 3 polymorphe Zuweisung"<< endl;
  B b3(5, "dreiA", 6, "dreiB");
  A& ar = b1;                    // Oberklassenreferenz
  ar = b3;
  cout << "ar nach Zuweisung ar=b3:\n";
  ar.ausgabe(); cout << endl;
  assert(ar == b3);

  // Klasse C
  cout << "\nTest 4" << endl;
  C c1(1, "einsA", 2, "einsB", 3, "einsC");
  C c2(4, "zweiA", 5, "zweiB", 6, "zweiC");
  c1.ausgabe(); cout << endl;
  c1 = c2;
  cout << "c1 nach Zuweisung c1=c2:\n";
  c1.ausgabe(); cout << endl;
  assert(c1 == c2);

  cout << "\nTest 5 : polymorphe Zuweisung A& = C"<< endl;
  C c3(7, "dreiA", 8, "dreiB", 9, "dreiC");
  A& arc = c2;                   // Oberklassenreferenz
  arc = c3;
  cout << "arc nach Zuweisung arc=c3:\n";
  arc.ausgabe(); cout << endl;
  assert(arc == c3);

  cout << "\nTest 6 : polymorphe Zuweisung B& = C"<< endl;
  B& brc(c2);                    // Oberklassenreferenz
  brc = c3;
  cout << "brc nach Zuweisung brc=c3:\n";
  brc.ausgabe(); cout << endl;
  assert(brc == c3);


  cout << "\nTest 7 : falscher Typ: b1 = c3"<< endl;
  try {
     b1 = c3;
  }
  catch(const bad_typeid& e) {
     cout << "Falscher Typ! Exception: " << e.what() << endl;
  }

  cout << "\nTest 8 : falscher Typ: a1 = c3"<< endl;
  try {
     a1 = c3;
  }
  catch(const bad_typeid& e) {
     cout << "Falscher Typ! Exception: " << e.what() << endl;
  }
  cout << "Test–Ende" << endl;
}
```

Es folgen die verwendeten Klassen, beginnend mit der obersten Basisklasse:

Listing 9.24: Basisklasse

```
// cppbuch/k9/zuweisung_vererbung/A.h
#ifndef A_H
#define A_H
#include<algorithm> // swap
#include<cstring>
#include<iostream>
#include<typeinfo>

using std::cout;
using std::endl;

class A {
public:
  A(int a_, const char* as_) : a(a_), as(new char[strlen(as_)+1]) {
    strcpy(as, as_);
  }

  A(const A& rhs) : a(rhs.a), as(new char[strlen(rhs.as)+1]) {
    strcpy(as, rhs.as);
  }

  virtual ~A() { delete [] as; }

  virtual A& assign(const A& rhs) {
    cout << "virtual A& A::assign(const A&)" << endl;
    if(typeid(*this) != typeid(rhs)) { // siehe Text unten
      throw std::bad_typeid();
    }
    A temp(rhs);
    swap(temp);
    return *this;
  }

  A& operator=(const A& rhs) {
    cout << "A& operator=(const A& rhs)" << endl;
    return assign(rhs);
  }

  virtual void ausgabe() const {
    cout << "A.a = " << a << ", A.as = " << as << " ";
  }

  void swap(A& rhs) {
    std::swap(a, rhs.a);
    std::swap(as, rhs.as);
  }

  virtual bool operator==(const A& arg) const {
    return typeid(*this) == typeid(arg) // siehe Text unten
      && a == arg.a && strcmp(as, arg.as) == 0;
```

```
  }
private:
   int a;
   char* as;
};
#endif
```

In C++ ist die Zuweisung `base = derived;` erlaubt. Dabei sei `base` ein Objekt der Basisklasse und `derived` ein Objekt einer abgeleiteten Klasse. Weil nur der Basisklassenanteil zugewiesen wird, wird so eine Anweisung nicht erwünscht sein. Aus diesem Grund wird mit `typeid` geprüft, ob der linke und der rechte Typ zur Laufzeit übereinstimmen. Falls nicht, wird eine Exception geworfen.

Ähnliches gilt für den Vergleichsoperator `==`. Bei dem Vergleich `base == derived;` wird normalerweise nur geprüft, ob die Basisklassenanteile im Objekt `derived` mit denen im Objekt `base` übereinstimmen. Um nicht zu ignorieren, dass es sich möglicherweise trotzdem um ganz verschiedene Typen handelt, wird `typeid` eingesetzt.

Weil der Vergleich bei abgeleiteten Klassen die Basisklasse einschließt, muss die Prüfung mit `typeid` in abgeleiteten Klassen nicht wiederholt werden. Das gilt nicht für `assign()`.

Listing 9.25: Abgeleitete Klasse B

```
// cppbuch/k9/zuweisung_vererbung/B.h
#ifndef B_H
#define B_H
#include"A.h"

class B : public A {
public:
   B(int a_, const char* as_, int b_, const char* bs_)
      : A(a_, as_),                              // Oberklassen-Subobjekt
        b(b_), bs(new char[strlen(bs_)+1]) {   // lokale Daten
      strcpy(bs, bs_);
   }

   B(const B& rhs)
      : A(rhs),                                  // Oberklassen-Subobjekt
        b(rhs.b), bs(new char[strlen(rhs.bs)+1]) { // lokale Daten
      strcpy(bs, rhs.bs);
   }

   ~B() { delete [] bs; }

   virtual B& assign(const A& rhs) {
      cout << "virtual B& B::assign=(const A&)" << endl;
      if(typeid(*this) != typeid(rhs)) {
         throw std::bad_typeid();
      }
      B temp(static_cast<const B&>(rhs));
      swap(temp);
      return *this;
   }
```

```
  B& operator=(const B& rhs) {
    cout << "B& operator=(const B& rhs)" << endl;
    return assign(rhs);
  }

  virtual void ausgabe() const {
    A::ausgabe();
    cout << "B.b = " << b << ", B.bs = " << bs << " ";
  }

  void swap(B& rhs) {
    A::swap(rhs);          // Oberklassendaten
    std::swap(b, rhs.b);   // lokale Daten
    std::swap(bs, rhs.bs); // lokale Daten
  }

  bool operator==(const A& arg) const {
    const B& rarg = static_cast<const B&>(arg);
    return A::operator==(arg) &&
      b == rarg.b && strcmp(bs, rarg.bs) == 0;
  }
private:
  int b;
  char* bs;
};
#endif
```

Listing 9.26: Abgeleitete Klasse C

```
// cppbuch/k9/zuweisung_vererbung/C.h
#ifndef C_H
#define C_H
#include"B.h"

class C : public B {
public:
  C(int a_, const char* as_, int b_, const char* bs_,
    int c_, const char* cs_)
    : B(a_, as_, b_, bs_),                // Oberklassen-Subobjekt
      c(c_), cs(new char[strlen(cs_)+1]) { // lokale Daten
    strcpy(cs, cs_);
  }

  C(const C& rhs)
    : B(rhs),                                  // Oberklassen-Subobjekt
      c(rhs.c), cs(new char[strlen(rhs.cs)+1]) { // lokale Daten
    strcpy(cs, rhs.cs);
  }

  ~C() { delete [] cs; }

  virtual C& assign(const A& rhs) {
    cout << "virtual C& C::assign=(const A&)" << endl;
```

```
    if(typeid(*this) != typeid(rhs)) {
      throw std::bad_typeid();
    }
    C temp(static_cast<const C&>(rhs));
    swap(temp);
    return *this;
  }

  C& operator=(const C& rhs) {
    cout << "C& operator=(const C& rhs)" << endl;
    return assign(rhs);
  }

  virtual void ausgabe() const {
    B::ausgabe();
    cout << "C.c = " << c << ", C.cs = " << cs << " ";
  }

  void swap(C& rhs) {
    B::swap(rhs);          // Oberklassendaten
    std::swap(c, rhs.c);   // lokale Daten
    std::swap(cs, rhs.cs); // lokale Daten
  }

  bool operator==(const A& arg) const {
    const C& rarg = static_cast<const C&>(arg);
    return B::operator==(arg) &&
      c == rarg.c && strcmp(cs, rarg.cs) == 0;
  }
private:
  int c;
  char* cs;
};
#endif
```

Aus diesem Abschnitt ergeben sich einige Empfehlungen, die beim Einsatz von Vererbung beherzigt werden sollten, es sei denn, Polymorphismus wird mit deutlichen Hinweisen und ausführlicher Dokumentation allen Benutzern einer Oberklasse verboten:

- Der nicht virtuelle Zuweisungsoperator muss existieren. Er ruft eine spezielle virtuelle Methode `assign()` auf, die die Zuweisung erledigt. Beispiel für eine Klasse `X`:
  `X& operator=(const X& rhs) { return assign(rhs);}`
- Jede Klasse hat praktischerweise eine `swap()`-Methode, die in der Zuweisung benutzt wird. Sie sorgt für die Vertauschung der Oberklassendaten und der lokalen Daten.
- Jede abgeleitete Klasse `Unterklasse`, egal an welcher Stelle der Vererbungshierarchie, sollte eine (in der obersten Basisklasse deklarierte) virtuelle Methode `assign()` mit der folgenden Struktur haben:

```
virtual Unterklasse& assign(const obersteBasisklasse& rhs) {
   Unterklasse temp(static_cast<const Unterklasse&>(rhs));
   swap(temp);
   return *this;
}
```

Die Methode `assign()` ist verantwortlich für die Zuweisung eines vollständigen Objekts der betreffenden Klasse und hat als Rückgabetyp eine Referenz dieser Klasse. Der Aufruf bewirkt bei dieser Struktur die Zuweisung aller anonymen Subobjekte der Oberklassen bis hin zur (obersten) Basisklasse.

- Nur wenn die Operationen `base = derived;` als Fehler betrachtet wird, müssen in den `assign()`-Methoden die Typen mit `typeid` geprüft werden.
- Nur wenn der Vergleich `(base == derived) false` ergeben soll, auch wenn die Basisklassenanteile links und rechts übereinstimmen, müssen im Vergleichsoperator der obersten Basisklasse die Typen mit `typeid` geprüft werden. Hier genügt die oberste Basisklasse, weil ein Vergleichsoperator einer Klasse den seiner direkten Oberklasse aufruft. Ohne Typprüfung ist der Vergleichsoperator nicht symmetrisch, d.h. `(base == derived)` kann `true` sein und das umgekehrte `false` (wobei dann ohne Typprüfung auf undefinierte Bereiche zugegriffen wird).

Manche empfehlen, vorher auf Identität zu prüfen, etwa:

```
virtual Unterklasse& assign(const obersteBasisklasse& rhs) {
   if(this != &rhs) {     // wirklich notwendig?
      Unterklasse temp(dynamic_cast<const Unterklasse&>(rhs));
      swap(temp);
   }
   return *this;
}
```

Eine Zuweisung der Art `x = x;` würde dann den Kopieraufwand vermeiden. Selbstzuweisung ist aber so selten, dass meiner Meinung nach gut auf die zusätzliche Abfrage verzichtet werden kann.

Schlussbemerkung

Das Beispiel zeigt auch, dass der Gleichheitsoperator virtuell sein kann, und dass ein nicht-virtueller Gleichheitsoperator nicht benötigt wird.

Der Zuweisungsoperator ist bei Vererbung und polymorpher Nutzung etwas komplizierter als ohne Vererbung. Bei Mehrfachvererbung ist noch zu beachten, dass die Subobjekte verschiedener Oberklassen richtig zugewiesen werden, und dass im Fall einer obersten Basisklasse (Stichwort virtuelle Vererbung) darauf geachtet werden muss, dass das Basisklassensubobjekt nur einmal zugewiesen wird. Dieses Thema ist aber zu speziell, um es hier zu vertiefen, und es gibt vermutlich nur sehr wenige Anwendungsfälle dafür. Wer trotzdem etwas darüber wissen möchte, findet in [Br] Informationen dazu.