4

Objektorientierung 1

Dieses Kapitel behandelt die folgenden Themen:

- Klasse und Objekt
- Konstruktion und Initialisierung von Objekten
- Destruktoren
- Der Weg von der Problemstellung zu Klassen und Objekten
- Gegenseitige Abhängigkeit von Klassen

Zunächst lernen Sie den Begriff *Abstrakter Datentyp* kennen, der die Grundlage für die anschließend erläuterten Begriffe *Klasse* und *Objekt* bildet. Ein Objekt vom Typ `Ort` dient als durchgängiges Beispiel. Objekte müssen vor ihrer Verwendung erzeugt werden, und sie müssen sinnvolle Daten enthalten. Die Erzeugung und Initialisierung ist die Aufgabe der verschiedenen *Konstruktoren*. Ein vollständiges Beispiel zum Rechnen mit rationalen Zahlen demonstriert das bis dahin Gelernte. *Destruktoren* sind zum Aufräumen da – sie zerstören nicht mehr benötigte Objekte.

4.1 Abstrakte Datentypen

Bisher haben wir Datentypen und Funktionen kennengelernt. In der Einführung auf Seite 27 wurde der Unterschied zwischen Klassen und Objekten beschrieben und darauf hingewiesen, dass Daten und Funktionen eines Objekts zusammengehören. Die Programmierung, wie wir sie bis jetzt kennengelernt haben, erlaubt es durchaus, unzulässige Funktionen auf Daten anzuwenden, zum Beispiel eine Buchung auf ein Konto unter Umgehung von Kontrollmechanismen vorzunehmen. Um unzulässige Zugriffe und damit auch versehentliche Fehler zu vermeiden, sollten die Daten *gekapselt* werden, indem man zusammengehörige *Daten* und *Funktionen* zusammenfasst. Das dadurch entstehende Gebilde heißt *Abstrakter Datentyp*.

Der Sinn liegt darin, den richtigen Gebrauch der Daten sicherzustellen. Die *tatsächliche* Implementierung der Datenstrukturen ist nach außen nicht sichtbar. Deshalb werden Datenstrukturen eines Abstrakten Datentyps ausschließlich durch die mit diesen Daten möglichen Operationen beschrieben. Von der internen Darstellung wird abstrahiert.

Mit »Funktion« ist hier *nicht* die konkrete Implementierung gemeint, das heißt, *wie* die Funktion im Einzelnen auf die Daten wirkt. Zur Verwendung eines Abstrakten Datentyps reicht die Spezifikation der Zugriffsoperation aus.

Abstrakter Datentyp = Datentypen + Funktionen

Ferner sind logisch zusammengehörige Dinge an einem Ort konzentriert. Die zusammen mit Daten gekapselten Funktionen heißen im Folgenden Elementfunktionen (englisch *member functions*). Der Zugriff auf die Daten soll *nur über Elementfunktionen* (auch Methoden genannt) möglich sein. Die Begriffe *Elementfunktion* und *Methode* werden synonym verwendet, obwohl der aus der Programmiersprache *Smalltalk* stammende Begriff *Methode* eigentlich besser auf die später zu besprechenden virtuellen Funktionen von C++ passt. Zum Vergleich sei der Zugriff auf zwei Koordinaten `x` und `y` eines Punktes auf verschiedene Arten gezeigt.

- **Unstrukturierter Zugriff**

```
int x = 100; // x-Koordinate eines Punktes
int y = 0;   // y-Koordinate eines Punktes
```

  Der Nachteil besteht darin, dass an jeder Stelle im Programm `x` und `y` ungeschützt verändert werden können. Eine Erweiterung der Funktionalität, zum Beispiel Protokollierung der Änderungen in einer Datei, muss an jeder Stelle nachgetragen werden.

- **Strukturierter Zugriff**
  Hier werden die Daten in eine Struktur gepackt (siehe Seite 88); der verändernde Zugriff geschieht über eine Funktion:

```
struct Punkt {
    int x, y;
} einPunkt;

void aendern(Punkt& p, int x, int y) {
     // ... Plausibilitätsprüfung
     p.x = x;
     p.y = y;
```

```
    // ... Protokollierung
  }

// Aufruf
aendern(einPunkt, 10, 800);
```

Der Vorteil besteht in der Gruppierung logisch zusammengehöriger Daten und der Änderung über eine Funktion. Eine Erweiterung der Funktionalität ist leicht realisierbar. Der Nachteil besteht darin, dass auch hier ein zusätzlicher Zugriff an der Funktion vorbei möglich ist, zum Beispiel

```
einPunkt.x = -3000;
```

- **Abstrakter Datentyp**
  Die Daten werden, unterstützt durch die Programmiersprache, so gekapselt, dass ein Zugriff *ausschließlich* über eine Funktion geschieht. Abbildung 4.1 verdeutlicht das Prinzip. Die Funktion als öffentliche Schnittstelle gehört zur Datenkapsel und ist der einzige Zugang. Eine direkte Änderung der Daten unter Umgehung der Funktion ist unmöglich.

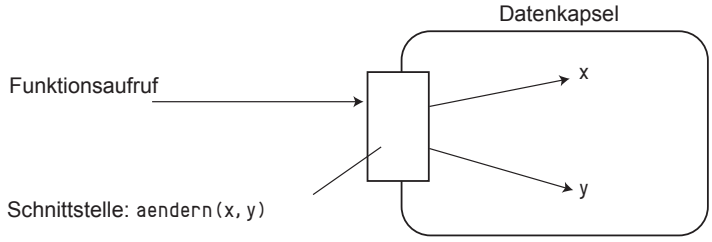

Abbildung 4.1: Abstrakter Datentyp

4.2 Klassen und Objekte

Eine Klasse ist ein Datentyp, genauer: ein Abstrakter Datentyp, der in einer Programmiersprache formuliert ist. Eine Klasse ist auch die *Beschreibung* von Objekten oder, anders ausgedrückt, *die Abstraktion von ähnlichen Eigenschaften und Verhaltensweisen ähnlicher Objekte*. Eine Klasse definiert die Struktur aller nach ihrem Muster erzeugten Objekte. In C++ dient die Klasse dazu, dem Compiler die Beschreibung von später zu definierenden Objekten mitzuteilen.

Ein Objekt hat einen inneren Zustand, der durch andere Objekte oder Elemente der in der Programmiersprache vorgegebenen Datentypen dargestellt wird. Der Zustand kann sich durch Aktivitäten des Objekts ändern, das heißt, durch Operationen, die auf Objektdaten ausgeführt werden. Die von jedem benutzbaren Operationen bilden die *öffentliche* Schnittstelle, in C++ gekennzeichnet durch das Schlüsselwort `public`.

Ein Objekt ist die konkrete Ausprägung einer Klasse, es belegt im Gegensatz zur Klasse Bereiche im Speicher, die mit definierten Bitmustern belegt sind, die Werte von Objekteigenschaften darstellen, wie der Betrag »15« der Wert in € der Eigenschaft Kontostand sein kann oder rot der Wert der Eigenschaft Farbe. Eigenschaften werden auch *Attribute* genannt, die bestimmte *Werte* besitzen. Der Zustand eines Objekts, der durch die zu den Attributen gehörenden Werte beschrieben wird, ist im Sinne des Abstrakten Datentyps nicht direkt änderbar, sondern nur über öffentliche Methoden. Die Attribute werden deshalb normalerweise mit dem Schlüsselwort `private` deklariert.

Ein Objekt besitzt eine *Identität*, die es unterscheidbar macht von einem beliebigen anderen Objekt, selbst wenn beide genau gleiche Daten enthalten. Die Identität zu einem bestimmten Zeitpunkt wird in C++ durch eine eindeutige Position im Speicher nachgebildet; zwei Objekte können niemals dieselbe Adresse haben, es sei denn, ein Objekt ist im anderen enthalten. Eine in C++ formulierte Klasse hat eine typische Gestalt:

```
class Klassenname {
  public:
  Typ elementfunktion1();
  Typ elementfunktion2();
  // und weitere ...
  private:
  Typ attribut1;
  Typ attribut2;
  // und weitere ...
};
```

Die reine Deklaration einer Elementfunktion wird auch *Prototyp* genannt. Zunächst betrachten wir ein einfaches Beispiel eines abstrakten Datentyps, nämlich einen *Ort*, der aus den X- und Y-Koordinaten besteht, sofern man sich auf zwei Dimensionen beschränkt. An Operationen seien vorgesehen:

`getX()`	X-Koordinate zurückgeben
`getY()`	Y-Koordinate zurückgeben
`aendern()`	X- und Y-Koordinaten ändern

Die Klasse wird `Ort1` genannt, weil sie noch Änderungen unterliegt. Sie ist wie folgt definiert:

Listing 4.1: Klasse Ort1, Version 1

```
// cppbuch/k4/ort1/ort1.h
#ifndef ORT1_H
#define ORT1_H
class Ort1 {                              // Version 1
   public:
     int getX() const;
     int getY() const;
     void aendern(int x,  int y);  // x,y = neue Werte
   private:
     int xKoordinate;
     int yKoordinate;
};                                // Semikolon nicht vergessen!
#endif     // ORT1_H
```

Daten *und* Methoden werden in einer Klasse zusammengefasst. Nach dem Schlüsselwort `class` folgen der Klassenname und ein durch geschweifte Klammern begrenzter Block mit den Deklarationen von Daten (Attributen) und Funktionen (Methoden). Der Gültigkeitsbereich von Klassenelementen ist lokal zu der Klasse: Die Daten sind privat, das heißt, sie sind von außen nicht zugreifbar, sodass Anweisungen wie `xKoordinate = 13;` unmöglich sind. Das Schlüsselwort `private` könnte entfallen, weil die Voreinstellung ohnehin `private` ist, wenn die privaten Daten *vor* dem `public`-Bereich stehen würden. Alles nach dem Schlüsselwort `public` Deklarierte ist öffentlich zugänglich. Die Funktionen heißen Elementfunktionen oder Methoden, von denen es private und öffentliche geben kann. Eine Datenänderung kann ausschließlich über eine geeignete Methode erfolgen, zum Beispiel `aendern()`. Das Schlüsselwort `const` oben drückt aus, dass die damit ausgezeichneten Elementfunktionen das Objekt nicht verändern können und deswegen auch auf konstante Objekte anwendbar sind. `getX()` und `getY()` geben nur Zahlen zurück, ohne die privaten Daten zu verändern, während `aendern()` die vorherigen Werte der privaten Variablen überschreibt (mehr zu `const` folgt in Abschnitt 4.5).

Objekterzeugung und -benutzung

Wie kann nun ein Ort in einem Programm benutzt werden? Zunächst muss ein Objekt erzeugt werden, denn die obige Klassendeklaration *beschreibt* nur ein Objekt. Wir nehmen an, dass die Deklaration der Klasse in einer Datei *ort1.h* vorliegt. Der Mechanismus der Objekterzeugung ist die übliche Variablendefinition, wobei die Klasse den Datentyp darstellt:

Listing 4.2: Ort-Objekt benutzen

```
// cppbuch/k4/ort1/ort1main.cpp
#include"ort1.h"            // Definition der Klasse einlesen
#include<iostream>
using namespace std;

int main() {                // Anwendung der Ort1-Klasse
  Ort1 einOrt1;             // Objekt erzeugen
  einOrt1.aendern(100, 200);
  cout << "Der Ort hat die Koordinaten x = "
       << einOrt1.getX() << " und y = "
       << einOrt1.getY() << endl;
}
```

Im Sinne der Aufteilung von Schnittstelle und Implementation werden Klassen wie Strukturen (`struct`) behandelt, wie in Abschnitt 3.3.4 (Seite 126 f.) beschrieben. Die Implementation der Funktionen sei in der Datei *ort1.cpp* abgelegt (Beschreibung siehe unten), die eine Zeile `#include"ort1.h"` enthält. Bei der Variablendefinition wird für ein Objekt Speicherplatz bereitgestellt. Um dies zu bewerkstelligen, wird beim Ablauf des Programms an der Stelle der Definition eine besondere Klassenfunktion aufgerufen, die *Konstruktor* genannt wird. Oben wird ein Objekt namens `einOrt1` definiert, das durch den impliziten Aufruf eines *Konstruktors* erzeugt wird und Speicherplatz für die Daten belegt. Der Programmcode der Funktionen ist selbstverständlich nur einmal für alle erzeugten `Ort1`-Objekte vorhanden.

Der Konstruktor wird vom System automatisch bereitgestellt, kann aber auch selbst definiert werden, wie Sie ab Seite 154 sehen. Zunächst hat das Objekt keinen definierten Zustand. Der wird im Beispiel erst mit dem Aufruf `einOrt1.aendern(100, 200);` hergestellt. Entsprechend der schon erläuterten Notation *Objektname.Anweisung(* gegebenenfalls *Daten)* erhält das Objekt den Auftrag, die Koordinaten auf (100, 200) zu ändern. *Wie* das Objekt diese Dienstleistung erbringt, ist in der Methode `aendern()` versteckt.

Im informationstechnischen Sprachgebrauch heißen Dinge (Objekte, Rechner etc.), die eine Dienstleistung erbringen, *Server*. Die Dienstleistung wird erbracht für einen *Client* (deutsch: Klient, Kunde), der selbst ein Rechner oder Objekt sein kann. Im obigen Programm ist das `main()`-Programm der Client, und das Objekt `einOrt1` ist der Server, der beauftragt wird, eine Koordinatenänderung durchzuführen.

Die Methode `aendern()` soll abgesehen von der Änderung der Koordinaten eine protokollierende Meldung auf den Bildschirm schreiben. Damit sind Änderungen unter Umgehung der Protokollierung unmöglich. Die Datei *ort1.cpp* enthält die Implementation der Methoden:

Listing 4.3: Implementierung von Ort1

```
// cppbuch/k4/ort1/ort1.cpp
#include"ort1.h"
#include<iostream>
using namespace std;

int Ort1::getX() const { return xKoordinate;}
int Ort1::getY() const { return yKoordinate;}

void Ort1::aendern(int x, int y) {
   xKoordinate = x;
   yKoordinate = y;
   cout << "Ort1-Objekt geändert! x = "
            << xKoordinate << " y = "
            << yKoordinate << endl;
}
```

Durch den Klassennamen und den Bereichsoperator `::` wird die Methode als zur Klasse `Ort1` gehörig gekennzeichnet. Daher darf *innerhalb* der Funktion auf die privaten Daten zugegriffen werden.

Das lauffähige Programm entsteht durch Übersetzen der Dateien `ort1.cpp` und `ort1main.cpp` und Linken der durch die Übersetzung entstandenen Dateien `ort1.o` und `ort1main.o`. Die Datei `ort1.h` wird während der Übersetzung der `*.cpp`-Dateien gelesen. Wenn sich alle zusammengehörigen Dateien in einem Verzeichnis befinden, werden Übersetzen und Linken mit dem Befehl *make* ausgeführt, falls keine Entwicklungsumgebung benutzt wird. Voraussetzung ist eine Datei namens *makefile*, die die Übersetzung steuert (mehr dazu siehe Kapitel 17). In den Beispielen auf der DVD ist dies stets der Fall.

4.2.1 inline-Elementfunktionen

Im Abschnitt 3.5 (Seite 139) haben wir den `inline`-Mechanismus für kleine Funktionen kennengelernt. Die Programmierung mit Klassen verwendet typischerweise viele kleine

Funktionen, sodass mit `inline` ein erheblicher Effizienzgewinn möglich ist. Dabei gibt es drei Möglichkeiten:

1. Deklaration *und* Definition innerhalb der Klasse. Beispiel:

```
// in ort1.h
class Ort1 {                              // Version 2
  public:
    int getX() const { return xKoordinate;}
    int getY() const { return yKoordinate;}
    void aendern(int x, int y) {
      xKoordinate = x;
      yKoordinate = y;
      std::cout << "Ort1-Objekt geändert! x = "
                << xKoordinate << " y = "
                << yKoordinate << std::endl;
    }
  private:
    int xKoordinate,
        yKoordinate;
};
```

2. Deklaration und Definition innerhalb der Header-Datei:

```
// in ort1.h
class Ort1 {                              // Version 3
   public:                 // nur Prototypen
     int getX() const;
     int getY() const;
     void aendern(int x, int y); // x,y = neue Werte
   private:
     int xKoordinate,
         yKoordinate;
};

// ========= inline - Implementierung ===========
inline int Ort1::getX() const { return xKoordinate;}
inline int Ort1::getY() const { return yKoordinate;}
inline void Ort1::aendern(int x, int y) {
   // ... wie oben
}
```

   Diese Variante hat den Vorteil, dass die Implementierung der Methoden nicht direkt innerhalb der Klassendeklaration sichtbar ist. Dadurch kann eine Klassendeklaration übersichtlicher werden.

3. `inline`-Deklaration außerhalb der Klasse. Dies ist fehlerhaft und daher nicht empfehlenswert (siehe Text unten). Beispiel:

```
// in ort1.h
class Ort1 {                              // Version 4
     // ... wie oben
     int getX() const;
};
```

```
// in ort1.cpp!
inline int Ort1::getX() const {  // Fehler!
   return xKoordinate;
}
```

Was im Beispiel 3, Version 4 auf den ersten Blick als möglich erscheint, ist aus den in Abschnitt 3.5 aufgeführten Gründen nicht praktikabel. Siehe dazu insbesondere den Hinweis auf Seite 139. Deshalb sollten nur die *ersten zwei* der drei beschriebenen Möglichkeiten benutzt werden.

4.3 Initialisierung und Konstruktoren

Objekte können während der Definition initialisiert, also mit sinnvollen Anfangswerten versehen werden. Es wurde bereits erwähnt, dass eine besondere Elementfunktion namens *Konstruktor* diese Arbeit neben der Bereitstellung von Speicherplatz übernimmt. Die Syntax von Konstruktoren ähnelt der von Funktionen, nur dass der Klassenname den Funktionsnamen ersetzt. Außerdem haben Konstruktoren keinen Return-Typ, auch nicht `void`. Im Sinn der Abstrakten Datentypen sind die Methoden einer Klasse für sinnvolle und konsistente Änderungen eines Objekts zuständig. Konstruktoren haben die Verantwortung, dass sich ein Objekt vom Augenblick der Entstehung an in einem korrekten Zustand befindet. Nun gibt es mehrere Arten von Initialisierungen, unterschieden durch verschiedene Arten von Konstruktoren, die im Folgenden beschrieben werden.

4.3.1 Standardkonstruktor

Falls kein Konstruktor angegeben wird, wird einer vom System automatisch erzeugt (implizite Konstruktordeklaration). Die Daten des Objekts enthalten dann unbestimmte Werte. Dieser vordefinierte Konstruktor (englisch *default constructor*) kann auch selbst geschrieben werden, um Attribute bei Anlage des Objekts zu initialisieren. Der Standardkonstruktor hat keine Argumente. Für eine Klasse `X` wird er einfach mit `X();` deklariert. Bei der Definition wird der Bezugsrahmen der Klasse angegeben, um dem Compiler mitzuteilen, dass es sich um eine Methode der Klasse `X` handelt, also `X::X() {...}`. In unserem Beispiel soll erreicht werden, dass bei Erzeugung eines `Ort1`-Objekts sofort gültige Koordinaten eingetragen werden. In der Implementationsdatei *ort1.cpp* wird der Standardkonstruktor definiert mit der Wirkung, dass jedes neue `Ort1`-Objekt sofort mit den Nullpunktkoordinaten initialisiert wird:

```
Ort1::Ort1() {              // neuer Standardkonstruktor
   xKoordinate = 0;         // Koordinaten des Nullpunkts
   yKoordinate = 0;
}
```

Die Klassendeklaration in *ort1.h* muss im `public`-Teil noch um die Zeile `Ort1();` ergänzt werden. Die Wirkung in einem Anwendungsprogramm wird in diesem Beispiel deutlich:

```
Ort1 einOrtObjekt;
cout << einOrtObjekt.getX() << endl // 0
     << einOrtObjekt.getY();      // 0
```

Hinweis

Bei den Konstruktoren des nächsten Abschnitts können Parameter in Klammern übergeben werden. Ein Standardkonstruktor muss stets *ohne* Klammern aufgerufen werden.

```
Ort1 nochEinOrtObjekt();  // Fehler!
```

Diese Zeile wird vom Compiler nämlich nicht als Definition eines neuen Objekts, sondern als *Deklaration* einer Funktion `nochEinOrtObjekt()` verstanden, die keine Argumente braucht und ein `Ort1`-Objekt zurückgibt.

4.3.2 Allgemeine Konstruktoren

Allgemeine Konstruktoren können im Gegensatz zu Standardkonstruktoren *Argumente* haben, und sie können genau wie Funktionen *überladen* werden, das heißt, dass es mehrere allgemeine Konstruktoren mit unterschiedlichen Parameterlisten geben kann. Die zu den nachstehenden Definitionen gehörigen Prototypen sind in der Klassendeklaration nachzutragen. Wenn mindestens ein allgemeiner Konstruktor definiert worden ist, wird vom System kein Standardkonstruktor erzeugt, das heißt, es gibt keinen, wenn man ihn nicht selbst geschrieben hat. Eine versehentliche Initialisierung mit unbestimmten Daten ist damit ausgeschlossen.

```
Ort1::Ort1(int x, int y) { // Allgemeiner Konstruktor
   xKoordinate = x;
   yKoordinate = y;
}
```

Konstruktion mit Parameter- oder Initialisierungsliste

Aufruf des Konstruktors heißt Definition des Objekts `nochEinOrt`:

```
Ort1 nochEinOrt1(70, 90); // Objektdefinition = Konstruktoraufruf (Parameterliste)
Ort1 nochEinOrt2 = {70, 90}; // Alternative mit externer Initialisierungsliste {}
Ort1 nochEinOrt3 {70, 90}; // dito ohne =
```

Wenn es mehrere allgemeine Konstruktoren gibt, sucht sich der Compiler den passenden heraus, indem er Anzahl und Datentypen der Argumente der Parameterliste der Konstruktordefinition mit der Angabe im Aufruf vergleicht.

Vorgegebene Parameterwerte in Konstruktoren

Das auf Seite 113 (Abschnitt 3.2.4) beschriebene Verfahren, Parametern von Funktionen einen Wert vorzugeben, ist auf Konstruktoren übertragbar. Die vorgegebenen Werte müssen in der Deklaration angegeben werden, hier also in der Datei *ort1.h*:

```
// in ort1.h
class Ort1 {                              // Version 5
```

```
    // ... wie oben
    Ort1(int x, int y = 100);
};
```

Dieser Konstruktor erlaubt zum Beispiel

```
Ort1 nochEinOrt(70);   // xKoordinate = 70, yKoordinate = 100!
Ort1 nochEinOrt(70, 90); // Vorgabewert wird überschrieben; oder allg. Konstruktor? (s.u.)
```

Dabei ist wie bei Funktionen darauf zu achten, dass eine Koexistenz von überladenen Konstruktoren stets zu eindeutigen Aufrufen führt. Version 5 kann nicht zusammen mit dem oben angegebenen allgemeinen Konstruktor verwendet werden, weil Aufrufe mit zwei Parametern nicht eindeutig einem der beiden zugeordnet sein können. Man kann den allgemeinen Konstruktor und den Standardkonstruktor kombinieren, indem alle Parameter mit Vorgabewerten versehen werden.

Initialisierung mit konstruktorinterner Liste

Was geschieht beim Aufruf des Konstruktors? Zunächst – noch vor Betreten des Blocks `{...}` – wird Speicherplatz für die Datenelemente `xKoordinate` und `yKoordinate` beschafft. Dann wird im zweiten Schritt der Programmcode innerhalb der geschweiften Klammern ausgeführt und die Aktualparameter werden zugewiesen. Es gibt eine Möglichkeit, beide Vorgänge in *einem* Schritt zusammenzufassen, der deshalb besonders bei größeren oder sehr vielen Objekten Laufzeitvorteile bringt. Der Weg führt über eine *Initialisierungsliste*, die noch vor dem Block angegeben und abgearbeitet wird:

```
// in ort1.h
class Ort1 {                              // Version 6
    // ... wie oben
    Ort1(int x, int y)                    // allgemeiner Konstruktor, inline
    : xKoordinate(x), yKoordinate(y){ // (interne) Initialisierungsliste
        // leerer Block
    }
};
```

Der Codeblock `{...}` kann sogar leer sein, hier bei Verzicht auf die Plausibilitätskontrolle. Die Initialisierung lässt sich auch aufteilen, indem zum Beispiel »`, yKoordinate(y)`« aus der Initialisierungsliste entfernt und der Codeblock um »`yKoordinate = y;`« ergänzt würde. Die Reihenfolge der Initialisierung ist:

1. Zuerst wird die Liste abgearbeitet. Die Reihenfolge der Initialisierung richtet sich nach der Reihenfolge innerhalb der Klassendeklarationen, nicht nach der Reihenfolge in der Liste. `xKoordinate` wird zuerst initialisiert. Wenn eine Initialisierung auf dem Ergebnis einer anderen aufbaut, wäre eine falsche Reihenfolge verhängnisvoll. Um solche Fehler zu vermeiden, sollen alle Elemente der Initialisierungsliste in der Reihenfolge ihrer Deklaration aufgeführt werden.
2. Danach wird der Codeblock `{...}` ausgeführt.

Die vorgezogene Abarbeitung der Liste wird auch benutzt, um Objekte oder Größen zu initialisieren, die innerhalb des Codeblocks *konstant* sind, wie wir noch sehen werden (Abschnitt 6.2). Die Klasse Ort wird weiter unten gebraucht. Deswegen wird sie hier unter dem neuen Namen `Ort` (statt `Ort1`) vollständig aufgeführt.

Listing 4.4: Klasse Ort

```
// cppbuch/include/ort.h
#ifndef ORT_H
#define ORT_H
#include<cmath>              // wegen sqrt()
#include<iostream>

class Ort {
   public:
     Ort(int einX = 0, int einY = 0)
     : xKoordinate(einX), yKoordinate(einY) {
     }

     int getX() const { return xKoordinate;}

     int getY() const { return yKoordinate;}

     void aendern(int x, int y) {
         xKoordinate = x;
         yKoordinate = y;
     }
   private:
     int xKoordinate;
     int yKoordinate;
};

inline void anzeigen(const Ort& o) { // globale Funktion
   std::cout << '(' << o.getX() << ", " << o.getY() << ')';
}

// globale Funktion zur Berechnung der Entfernung zwischen zwei Orten
inline double entfernung(const Ort& ort1, const Ort& ort2) {
  double dx = static_cast<double>(ort1.getX() - ort2.getX());
  double dy = static_cast<double>(ort1.getY() - ort2.getY());
  return std::sqrt(dx*dx + dy*dy);
}
#endif    // ORT_H
```

In der Methode `aendern()` wird auf die Kontrollausgabe verzichtet. Warum?

- Zwei verschiedene Dinge sollen nicht von derselben Methode erledigt werden.
- Eine Ausgabe (oder Eingabe) in einer Methode, die eigentlich eine andere Aufgabe hat, verhindert den universellen Einsatz. Zum Beispiel ließe sich die Methode nicht ohne Weiteres in einem System mit grafischer Benutzungsoberfläche verwenden.

Eine Kontrolle sollte die Benutzung nicht beeinträchtigen und sollte daher anders realisiert werden – wie, hängt vom Anwendungsfall ab. Alle Methoden der Klasse `Ort` sind sehr kurz und der Einfachheit halber `inline` deklariert worden. Der Konstruktor definiert einen Ort (0, 0), sofern keine Koordinaten angegeben werden. Die globale Funktion `entfernung()` wird noch benötigt, und `anzeigen()` gibt die Koordinaten im Format (x, y) aus.

4.3.3 Kopierkonstruktor

Ein Kopierkonstruktor wird im Englischen *copy constructor* oder treffender *copy initializer* genannt. Er dient dazu, ein Objekt mit einem anderen zu *initialisieren*. Das erste (und im Allgemeinen einzige) Argument des Kopierkonstruktors ist eine *Referenz auf ein Objekt derselben Klasse*. Die Deklaration eines Kopierkonstruktors der Klasse `X` lautet `X(X&);`. Weil ein Objekt, das dem Kopierkonstruktor als Argument dient, nicht verändert werden soll, ist es sinnvoll, es als Referenz auf `const` zu übergeben: `X(const X&);`. Falls kein Kopierkonstruktor vorgegeben wird, wird für jede Klasse bei Bedarf vom System einer erzeugt, der die einzelnen Elemente des Objekts kopiert. Die Elemente können selbst wieder Objekte sein, deren Kopierkonstruktor dann wiederum aufgerufen wird, sei es ein selbst definierter oder der vom System bereitgestellte. Die Kopie jedes Grunddatentyps ist eine bitweise Abbildung des Speicherbereichs. Der Kopierkonstruktor der Klasse `Ort` wird wie der Standardkonstruktor in den `public`-Bereich der Klasse `Ort` geschrieben. Er ist wie folgt definiert:

```
Ort(const Ort& einOrt)          // Kopierkonstruktor
  // Kopie der einzelnen Elemente:
 : xKoordinate(einOrt.xKoordinate),
   yKoordinate(einOrt.yKoordinate) {
     // Anzeige des Aufrufs nur zur Demonstration
     std::cout << "Kopierkonstruktor aufgerufen\n";
}
```

Eigentlich bräuchten wir keinen eigenen Kopierkonstruktor für die schlichten Elemente der Klasse `Ort`, weil der vom System erzeugte genügen würde. Er wurde nur geschrieben, um den Aufruf auf dem Bildschirm dokumentieren zu können. *Wenn* schon ein Kopierkonstruktor mit besonderen Aktionen geschrieben wird, darf die Kopie der einzelnen Elemente nicht vergessen werden. Manchmal sind andere Operationen als die Kopie notwendig; entsprechende Beispiele werden Sie kennenlernen. Die Syntax der Initialisierung unterscheidet sich nicht vom Üblichen:

```
Ort einOrt(19, 39);
Ort derZweiteOrt = einOrt; // Aufruf des Kopierkonstruktors
// gleichwertig ist: Ort derZweiteOrt(einOrt);
cout << derZweiteOrt.getX() << ''
     << derZweiteOrt.getY(); // 19 39
```

Diese Definition erzeugt ein Objekt `derZweiteOrt`, das mit den Werten des bereits vorhandenen Objekts `einOrt` initialisiert wird. Auch wenn das Gleichheitszeichen verwendet wird, handelt es sich hier um eine *Initialisierung* und *nicht* um eine *Zuweisung*, zwei Dinge, die in ihrer Bedeutung streng unterschieden werden.

Ein Kopierkonstruktor wird nur dann benutzt, wenn ein *neues* Objekt erzeugt wird, aber *nicht* bei Zuweisungen, also Änderungen von Objekten.

Bei *Zuweisungen* wird der vom System bereitgestellte Zuweisungsoperator benutzt, sofern kein eigener definiert wurde – auch das ist möglich, wie Sie sehen werden. Der Kopierkonstruktor wird nicht aufgerufen bei Zuweisungen oder Initialisierungen, bei denen ein temporär erzeugtes Objekt in das neu erzeugte Objekt kopiert wird. In den Folgezeilen wird daher kein Kopierkonstruktor aufgerufen.

```
Ort o1, o2;          // allg. Konstruktoren mit Vorgabewerten (0,0)
o1 = Ort(8,7);       // allgemeiner Konstruktor + Zuweisung
o2 = o1;             // Zuweisung
Ort o3 = Ort(1, 17); // Ein temporäres Objekt wird in das neu erzeugte Objekt kopiert.
```

Merke:
Die Übergabe von Objekten an eine Funktion per Wert und die Rückgabe eines Ergebnisobjekts wird ebenfalls als Initialisierung betrachtet, ruft also den Kopierkonstruktor implizit auf. Ausnahme: Optimierung durch den Compiler (s.u.).

Dies lässt sich am folgenden Beispiel zeigen, wobei angenommen wird, dass der Kopierkonstruktor wie oben mit einer Ausgabeanweisung versehen ist. Es gebe eine Funktion `ortsverschiebung()`, die auf einen gegebenen Ort eine bestimmte Entfernung in x- bzw. y-Richtung hinzuaddieren soll. Die Funktion greift nicht auf private Attribute eines Ortobjekts zu und braucht daher keine Elementfunktion zu sein. Diese Funktion soll innerhalb `main()` benutzt werden, zum Beispiel:

Listing 4.5: Beispiel zum Kopierkonstruktor

```
// cppbuch/k4/ortmain.cpp
#include"../include/ort.h"
using namespace std;

// Funktion zum Verschieben des Orts um dx und dy
Ort ortsverschiebung(Ort derOrt, int dx, int dy) {
    derOrt.aendern(derOrt.getX() + dx, derOrt.getY() + dy);
    return derOrt;      // Rückgabe des veränderten Orts
}

int main() {
   Ort einOrt(10, 300);
   Ort verschobenerOrt = ortsverschiebung(einOrt, 10, -90);
   cout << " alter Ort: ";
   anzeigen(einOrt);
   cout << "\n neuer Ort: ";
   anzeigen(verschobenerOrt);
}
```

Der oben definierte Kopierkonstruktor wird zweimal aufgerufen (falls der Aufruf nicht wegoptimiert wird, siehe unten): das erste Mal bei der Übergabe des Objektes `einOrt` an die Funktion und das zweite Mal während der Ausführung der `return`-Anweisung. Hier wird besonders deutlich, dass die Übergabe per Referenz auf `const` einiges an Geschwindigkeitsgewinn bringen kann, falls man keine Objektkopie in der Funktion benötigt. Im obigen Beispiel ist allerdings die Übergabe per Wert erforderlich, weil eine Objektkopie zum Aufruf von `aendern()` benötigt wird, ohne dass `einOrt` verändert wird.

Optimierung durch den Compiler

Im obigen Beispiel wird ein temporäres Objekt erzeugt, mit dem die Variable `verschobenerOrt` initialisiert wird. Dies kostet Zeit und Speicherplatz. Die Initialisierung selbst

benötigt ebenfalls Zeit, ebenso die anschließende »Entsorgung« des temporären Objekts, das nicht weiter benötigt wird. Deshalb ist es Compilern unter bestimmten Bedingungen erlaubt, die Erzeugung temporärer Objekte zu vermeiden. Bezogen auf das obige Beispiel könnte der Compiler das zurückzugebende Objekt `derOrt` der Funktion `ortsverschiebung()` *direkt* an die Stelle von `verschobenerOrt` schreiben, statt den Kopierkonstruktor aufzurufen. Dies gilt auch, wenn es Seiteneffekte gibt (Abschnitt 12.8 in [ISOC++]). Wenn etwa der Kopierkonstruktor seine Aufrufe protokolliert, kann es ein, dass die Protokollausgaben im Fall der Optimierung fehlen, weil der Kopierkonstruktor nicht aufgerufen wird.

Mehr dazu lesen Sie in Abschnitt 22.1.

4.3.4 Typumwandlungskonstruktor

Der Typumwandlungskonstruktor dient zur Umwandlung anderer Datentypen in die gewünschte Klasse. Das erste Argument des Typumwandlungskonstruktors ist verschieden vom Typ der Klasse. Falls weitere Argumente folgen, was im Allgemeinen nicht der Fall sein wird, müssen sie Initialisierungswerte haben. Im Grunde ist der Typumwandlungskonstruktor nichts anderes als der Spezialfall eines allgemeinen Konstruktors, der etwas anders eingesetzt wird.

Hier wird ein Typumwandlungskonstruktor gezeigt, der als String vorliegende Ortsangaben in ein Ort-Objekt verwandelt. Das Format ist so gewählt, dass zwei getrennte Folgen von Ziffern als x- und y-Koordinaten interpretiert werden. Andere Zeichen werden ignoriert. Damit sind Schreibweisen wie `"100 200"` möglich, aber auch `"(100,200)"`.

```
// zusätzlich benötigte Include-Anweisungen in ort.h:
#include<cctype>   // isdigit()
#include<string>
#include<cassert>

// im public-Bereich von ort.h (Seite 157) einfügen:
      // Typumwandlungskonstruktor. Format: 2 Folgen von Ziffern
      Ort(const std::string& str) {
         unsigned int pos = 0;   // Position einer Ziffer im String str
         for(int j = 0; j < 2; ++j) { // für jede Koordinate
            while(pos < str.size() && !isdigit(str.at(pos))) // erste Ziffer suchen
                 ++pos;
            assert(pos < str.size()); // Ziffer gefunden? Abbruch, falls nicht
            // Zahl bilden
            int koordinate = 0;
            while(pos < str.size() && isdigit(str.at(pos))) {
                 // implizite Typumwandlung char → int
              koordinate = 10*koordinate + str.at(pos) - '0';
              ++pos;
            }
            switch(j) {
              case 0: xKoordinate = koordinate; break;
              case 1: yKoordinate = koordinate;
            }
         }
      }
```

Ein Beispiel zeigt die Anwendung:

```
Ort nochEinOrt(string("21 99")); // mögliches Format
anzeigen(nochEinOrt);
cout << endl;
Ort einWeitererOrt("(55, 8)");  // weiteres mögliches Format
anzeigen(einWeitererOrt);
```

Im zweiten Fall findet der Compiler eine in " " eingeschlossene Folge von Zeichen. Die Umwandlung in ein String-Objekt nimmt der Compiler automatisch vor, die Typprüfung des Compilers wird eingeschränkt. Die Einschränkung wird hier besonders deutlich:

```
string wo("20,400");
Ort hier = ortsverschiebung(wo, 10, -90);
// besser: explizite Angabe der Typumwandlung (s.u.)
Ort dort = ortsverschiebung(Ort(wo), 10, -90);
```

Die Umwandlung für das Objekt `hier` funktioniert, obwohl es keine überladene Funktion `ortsverschiebung()` mit einem String-Objekt als erstem Parameter gibt. Man kann sich vorstellen, dass der Compiler intern, das heißt, ohne dass wir etwas davon mitbekommen, eine temporäre Variable erzeugt, die die Umwandlung vornimmt:

```
// Erzeugen eines temporären Ort-Objekts __temp:
Ort __temp(wo);            // Typumwandlungskonstruktor
Ort hier = ortsverschiebung(__temp, 10, -90);
// Ab hier kann das temporäre Objekt zerstört werden.
```

Die Variable `__temp` ist temporär und existiert nur als Hilfsvariable des Compilers, sodass sie später nicht weiter stört. An die Funktion wird also ein temporäres Objekt mit dem richtigen Datentyp übergeben.

Tipp

Im Allgemeinen möchte man sowohl die Möglichkeit der Typumwandlung haben als auch die Typprüfung durch den Compiler, damit keine Fehler durch implizite Typumwandlungen entstehen. Das Schlüsselwort `explicit` erlaubt es, explizite Typumwandlungen durchzuführen, aber andere, vielleicht unbeabsichtigte, zu verbieten.

```
class Ort {
   public:
     // Typumwandlungskonstruktor mit explicit
     explicit Ort(const std::string& str);
    // ... Rest wie vorher
};
```

```
Ort o1;
string wo("10, 200");
o1 = wo;              // jetzt ein Fehler!
o1 = Ort(wo);         // erlaubte explizite Typumwandlung
```

Ein weiteres Beispiel für den Einsatz eines Typumwandlungskonstruktors wird im nächsten Abschnitt gezeigt.

4.4 Beispiel: Rationale Zahlen

4.4.1 Aufgabenstellung

Es folgt ein vollständiges Beispiel für eine Klasse, mit deren Objekten gerechnet werden kann. Es soll eine Bibliothek zum Rechnen mit rationalen Zahlen programmiert werden. Rationale Zahlen sind Zahlen, die durch einen Bruch darstellbar sind, wobei Zähler und Nenner ganzzahlig sein müssen. Dabei soll ein Anwender rationale Zahlen mit dem Datentyp `Rational` definieren können. Mit diesen Zahlen sollen alle Grundrechenarten ohne Genauigkeitsverlust durchführbar sein. Alle Ergebnisse von Rechnungen mit rationalen Zahlen sollen in bereits gekürzter Form vorliegen. Negative Zahlen sind durch einen negativen Zähler zu repräsentieren.

Benutzungsschnittstelle

Ein die Klasse benutzendes Programm muss die Datei *rational.h* per `#include`-Anweisung einlesen. Die Objektdatei *rational.o* muss eingebunden (»gelinkt«) werden. Die Definition einer rationalen Zahl `r` soll wie eine übliche Variablendeklaration geschrieben werden, zum Beispiel: `Rational r;`.

Die folgenden Funktionen sollen von der Bibliothek bereitgestellt werden, wobei die Operanden `a` und `b` vom Typ `Rational`, `int` oder `long int` sein können. Im Kapitel 9 wird gezeigt, wie man diese Funktionen direkt durch die üblichen mathematischen Operatoren ersetzen kann.

`r.eingabe();`	Dialogeingabe der rationalen Zahl `r`
`r.ausgabe();`	Dialogausgabe der rationalen Zahl im Format Zähler/Nenner
`r.getZaehler();`	Zähler zurückgeben
`r.getNenner();`	Nenner zurückgeben
`r.set( z, n);`	Setzen der Werte für Zähler und Nenner
`r.kehrwert();`	`r` enthält danach den Kehrwert

Rechenfunktionen:

```
r = add(a, b);      r = a + b
r = sub(a, b);      r = a - b
r = mult(a, b);     r = a * b
r = div(a, b);      r = a / b
```

Kurzformoperationen:

```
r.add(a);           r += a
r.sub(a);           r -= a
r.mult(a);          r *= a
r.div(a);           r /= a
```

Beschränkungen und Hinweise

Nach jeder Operation sollen die Zahlen gekürzt werden. Eine Bereichsüberprüfung ist der Einfachheit halber nicht vorgesehen. Anwender sind demnach selbst dafür verantwortlich, dass der für den Datentyp `long` zutreffende Bereich ihres Rechners in Zwischenrechnungen nicht überschritten wird. In einer kommerziellen Version sollte dieser Fall we-

nigstens zu einer Fehlermeldung führen. Falls der Nenner null wird, soll das Programm mit einer Fehlermeldung abbrechen.

4.4.2 Entwurf

In diesem Abschnitt werden die Algorithmen, die die mathematische Grundlage des Programms bilden, sowie einige Überlegungen zur Implementierung dargestellt. Für die folgenden arithmetischen Ausdrücke gilt :

$x.z$ ist der Zähler der rationalen Zahl x
$x.n$ ist der Nenner der rationalen Zahl x

Addition	$r = a + b$:	$r = \frac{a.z * b.n + b.z * a.n}{a.n * b.n}$
Subtraktion	$r = a - b$:	$r = \frac{a.z * b.n - b.z * a.n}{a.n * b.n}$
Multiplikation	$r = a * b$:	$r = \frac{a.z * b.z}{a.n * b.n}$
Division	$r = a/b$:	$r = \frac{a.z * b.n}{a.n * b.z}$
Kehrwert bilden	$1/r$:	Vertauschen von Zähler und Nenner

Die Rechenregeln für die Kurzformoperatoren sind entsprechend. Im Programm können Vereinfachungen vorgenommen werden etwa der Art, dass die Subtraktion auf die Addition einer negativen Zahl zurückgeführt wird.

Für den Datentyp der rationalen Zahl wird die Klasse `Rational` deklariert, die als private Attribute nur den Zähler und den Nenner hat. Die in der Anforderungsdefinition vorgeschriebenen Methoden werden in die Klasse übernommen. Hinzu kommt noch die Methode `kuerzen()`, die intern benötigt wird. Beim Betrachten der Operationen ist festzustellen:

- Eine Operation wie `r.add(a)` ändert das Objekt `r`, aber nicht das Objekt `a`. Weil Objekte nur über Methoden geändert werden können (Abstrakter Datentyp!), ist `add()` eine Methode oder Elementfunktion. Der Parameter `a` kann per Wert oder per Referenz auf `const` übergeben werden. Letztes spart das Erzeugen einer lokalen Kopie, lohnenswert bei großen Objekten. Ein Objekt des Typs `Rational` ist jedoch relativ klein, weswegen dieser Punkt hier keine große Rolle spielt. Die Funktion `add()` gibt nichts zurück, der Rückgabetyp ist `void`. Eine andere Möglichkeit für den Rückgabetyp wird in Punkt 6 auf Seite 559 diskutiert.
- Eine Operation wie `r = add(a,b)` ändert nicht die Parameter `a` und `b`. Für die Parameterübergabe gelten daher die Argumente des vorhergehenden Punktes. Die Funktion `add(a,b)` erzeugt eine rationale Zahl als Ergebnis, die der Variablen `r` zugewiesen wird. Der Rückgabetyp ist also `Rational`. Da die Funktion wegen der Funktionen `getZaehler()` und `getNenner()` nicht auf private Attribute zugreifen muss, kann sie global sein.

Weil Operationen mit gemischten Datentypen möglich sein sollen, muss ein Mechanismus dafür bereitgestellt werden. Zwei Möglichkeiten sind üblich:

1. Die arithmetischen Methoden und Funktionen können überladen werden, sodass für die beiden Operanden folgende Kombinationen erlaubt sind:
   Methode, zum Beispiel
   `void Rational::add(const Rational&)`
   `void Rational::add(long)`
   und globale Funktion, zum Beispiel
   `Rational add(const Rational&, const Rational&)`
   `Rational add(const Rational&, long)`
   `Rational add(long, const Rational&)`
   `int`-Werte würden implizit nach `long` konvertiert werden. Pro globaler Rechenoperation würden drei Methoden anstatt einer benötigt, die allerdings teilweise aufeinander zugreifen könnten.
2. Wenn die Anzahl der Methoden und Funktionen klein gehalten werden soll, muss dem Compiler eine Möglichkeit zur Typumwandlung zur Verfügung gestellt werden, wenn er die Daten an die Rechenfunktionen übergibt. Dies geschieht am günstigsten durch einen Typumwandlungskonstruktor, wie er im letzten Abschnitt behandelt wurde. Er darf hier natürlich *nicht* `explicit` sein!

In diesem Fall wurde die Entscheidung für die zweite Variante getroffen. Die Klassendeklaration in der Datei *rational.h* lautet:

Listing 4.6: Klasse Rational

```
// cppbuch/k4/ratio/rational.h Klasse für rationale Zahlen
#ifndef RATIONAL_H
#define RATIONAL_H

class Rational {
  public:
    Rational();
    Rational(long z, long n); // allgemeiner Konstruktor
    Rational(long);           // Typumwandlungskonstruktor

    // Abfragen
    long getZaehler() const;
    long getNenner() const;

    // arithmetische Methoden für +=, -=, *=, /=,
    // (werden später durch überladene Operatoren ergänzt)
    void add(const Rational& r);
    void sub(const Rational& r);
    void mult(const Rational& r);
    void div(const Rational& r);

    // weitere Methoden
    void set(long zaehler, long nenner);
    void eingabe();
    void ausgabe() const; // siehe dazu Seite 170
    void kehrwert();
```

```
    void kuerzen();
  private:
    long zaehler, nenner;
};

// inline-Methoden
inline Rational::Rational() : zaehler(0), nenner(1) {}

inline Rational::Rational(long z, long n)
: zaehler(z), nenner(n) {}

inline Rational::Rational(long ganzeZahl)
: zaehler(ganzeZahl), nenner(1) {}

inline long Rational::getZaehler() const {return zaehler;}
inline long Rational::getNenner() const {return nenner;}

// globale Funktionsprototypen
const Rational add(const Rational& a, const Rational& b);
const Rational sub(const Rational& a, const Rational& b);
const Rational mult(const Rational& a, const Rational& b);
const Rational div(const Rational& z, const Rational& n);
#endif
```

Die Methode `kuerzen()` wird implizit von den anderen Methoden aufgerufen und könnte daher privat sein. Die `public`-Eigenschaft schadet aber nicht.

Hinweis

Warum haben die globalen Funktionsprototypen einen `const`-Spezifizierer? Wenn `const` fehlte, könnte das zurückgegebene Objekt als L-Wert (englisch *lvalue*) benutzt werden, weil es veränderbar ist. Mit anderen Worten, es könnte auf der linken Seite einer Zuweisung stehen (vgl. Seite 62), etwa so: `add(a, b) = c;`. Natürlich ist die Zuweisung von `c` zu dem temporären Ergebnis unsinnig, der Compiler würde die Anweisung aber akzeptieren. Mit `const` gibt der Compiler an dieser Stelle eine Fehlermeldung aus.

Fehlerbetrachtung

Ein Überlauf des Zahlenbereichs soll nicht geprüft werden, wohl aber der Fall, dass ein Nenner 0 wird. Dies kann in den Methoden `eingabe()`, `kehrwert()`, `set()` und `div()` geschehen, wobei Letztere nicht betrachtet werden muss, wenn man die Division durch die Multiplikation mit dem Kehrwert implementiert. Mit Hilfe von `assert()` wird geprüft, ob der Nenner 0 ist. Bei der Eingabe ist dieses Vorgehen sicher nicht sehr benutzerfreundlich und könnte daher modifiziert werden.

4.4.3 Implementation

Die Implementation der Methoden, globalen Funktionen einschließlich der Hilfsfunktion `ggt()` zum Finden des größten gemeinsamen Teilers[1], um einen Bruch zu kürzen, sind in der Datei *rational.cpp* abgelegt.

Listing 4.7: Implementation der Klasse Rational

```
// cppbuch/k4/ratio/rational.cpp (Definition der Methoden und globalen Funktionen)
#include"rational.h"
#include<iostream>
#include<cassert>
using namespace std;

void Rational::set(long z, long n) {
    zaehler = z;
    nenner = n;
    assert(nenner != 0);
    kuerzen();
}

void Rational::eingabe() {
    // Bildschirmausgabe nur zu Demonstrationszwecken.
    // cerr wird gewählt, damit die Abfragen auch dann auf dem Bildschirm erscheinen,
    // wenn die Standardausgabe in eine Datei zur Dokumentation geleitet wird.
    cerr << "Zähler :";
    cin >> zaehler;
    cerr << "Nenner :";
    cin >> nenner;
    assert(nenner != 0);
    kuerzen();
}

void Rational::ausgabe() const {
    cout << zaehler << '/' << nenner << endl;
}

void Rational::kehrwert() {
    long temp = zaehler;
    zaehler = nenner;
    nenner = temp;
    assert(nenner != 0);
}

void Rational::add(const Rational& r) {
    zaehler = zaehler*r.nenner + r.zaehler*nenner;
    nenner = nenner*r.nenner;
    kuerzen();
}

void Rational::sub(const Rational& s) {
    Rational r = s;
```

[1] Die erste, langsamere Fassung des Algorithmus wird auf Seite 70 gezeigt.

```
    // Das temporäre Objekt r wird benötigt, weil s wegen der const-Eigenschaft nicht
    // verändert werden kann (und darf). Die eigentlich bessere Alternative wäre die Übergabe
    // per Wert – dann könnte mit der lokalen Kopie gearbeitet werden (Begründung siehe
    // nachfolgender Text). Die Subtraktion kann durch Addition des negativen Arguments
    // erreicht werden.
    r.zaehler *=-1;
    add(r);
}

void Rational::mult(const Rational& r) {
    zaehler = zaehler*r.zaehler;
    nenner = nenner *r.nenner;
    kuerzen();
}

void Rational::div(const Rational& n) {
    Rational r = n; // siehe Diskussion bei sub()
    r.kehrwert();
    mult(r);        // Division = Multiplikation mit dem Kehrwert
}

// Die globale Funktion ggt() wird zum Kürzen benötigt. Sie berechnet den größten
// gemeinsamen Teiler. Verwendet wird ein modifizierter Euklid-Algorithmus, in dem
// Subtraktionen durch die schnellere Restbildung ersetzt werden.
long ggt(long x, long y) {
   long rest;
   while(y > 0) {
      rest = x % y;
      x = y;
      y = rest;
   }
   return x;
}

void Rational::kuerzen() {
   // Vorzeichen merken und Betrag bilden
   int sign = 1;
   if(zaehler < 0) { sign = -sign; zaehler = -zaehler;}
   if(nenner < 0) { sign = -sign; nenner = -nenner;}
   long teiler = ggt(zaehler, nenner); // siehe oben
   zaehler = sign*zaehler/teiler; // Vorzeichen restaurieren
   nenner = nenner/teiler;
}

// Es folgen die globalen arithmetische Funktionen für die
// Operationen mit 2 Argumenten (binäre Operationen)

const Rational add(const Rational& a, const Rational& b) {
   // Die eigentliche Berechnung muss hier nicht wiederholt werden, sondern die bereits
   // vorhandenen Funktionen für die Kurzformen der Addition usw. können vorteilhaft
   // wiederverwendet werden. Dazu wird ein mit a initialisiertes temporäres Objekt
   // erzeugt, auf das das Argument b addiert und das dann als Ergebnis
   // zurückgegeben wird. Zum temporären Objekt r siehe auch die Diskussion bei der
```

```
   // Elementfunktion sub() oben.
   Rational r = a;
   r.add(b);
   return r;
}

const Rational sub(const Rational& a, const Rational& b) {
   Rational r = a;
   r.sub(b);
   return r;
}

const Rational mult(const Rational& a, const Rational& b) {
   Rational r = a;
   r.mult(b);
   return r;
}

const Rational div(const Rational& z, const Rational& n) {
   Rational r = z;
   r.div(n);
   return r;
}
```

Warum ist die Übergabe per Wert bei der Funktion `sub()` besser? Antwort: Falls der Parameter ein temporäres Objekt ist, hat der Compiler die Chance, den Aufruf des Kopierkonstruktors zu eliminieren (siehe Seite 159 unten). So kann sich gelegentlich erst bei dem Entwurf der Implementierung herausstellen, dass eine andere Schnittstelle günstiger ist, hier `sub(Rational)` statt `sub(const Rational&)`. `Rational`-Objekte sind klein; insofern macht der Unterschied hier nicht viel aus und die Schnittstelle wird beibehalten.

Der Einsatz des Typumwandlungskonstruktors vereinfacht das Programm durch die Reduktion der Methodenanzahl, führt aber zu einem kleinen Effizienzverlust, weil zur Laufzeit einige Schritte mehr ausgeführt werden müssen. Solange ein Programm nicht zeitkritisch ist, ist es immer sinnvoll, der Einfachheit den Vorzug zu geben und nicht der Geschwindigkeit.

Übungen

4.1 Schreiben Sie die Funktionen `add(long a, Rational b)` und `add(Rational a, long b)`, die bei *Abwesenheit* des Typumwandlungskonstruktors erforderlich wären.

4.2 Schreiben Sie eine Funktion, die ein Objekt vom Typ `Rational` übergeben bekommt und dasselbe tut wie die Funktion `Rational::ausgabe()`, ohne auf private Daten zuzugreifen.

Testdokumentation

Um die Klasse `Rational` zu testen, wurde die Datei *cppbuch/k4/ratio/main.cpp* geschrieben, die *rational.h* einbindet. Die aus *rational.cpp* durch Compilation entstandene Datei *rational.o* muss dazu gelinkt werden. Das ausführbare Programm *a.out* bringt die Test-

ausgaben auf den Bildschirm. Mit der Anweisung *a.out > test.erg* werden alle Ausgaben in die Datei *test.erg* geschrieben.

Ergebnisse des Testprogramms

Die Testergebnisse werden hier aus Platzgründen und weil sie überaus langweilig zu lesen sind, nicht abgedruckt. Probieren Sie das Testprogramm aus und erweitern Sie es um zusätzliche Prüfungen, um Fehlern auf die Spur zu kommen!

Listing 4.8: Testprogramm

```
// cppbuch/k4/ratio/main.cpp    Testprogramm für Klasse Rational (Auszug)
#include"rational.h"
#include<iostream>
using namespace std;

// alle 4 Operationen für a und b
void druckeTestfall(const Rational& a, const Rational& b) {
    Rational erg;
    cout << "a = "; a.ausgabe();
    cout << "b = "; b.ausgabe();
    // Die Elementfunktionen werden implizit mitgetestet.
    erg = add(a,b);
    cout << "+ : "; erg.ausgabe();
    erg = sub(a,b);
    cout << "- : "; erg.ausgabe();
    erg = mult(a,b);
    cout << "* : "; erg.ausgabe();
    erg = div(a,b);
    cout << "/ : "; erg.ausgabe();
    cout << endl;
}

int main() {
    Rational a,b;
    cout << "Test der Eingabe\n";
    a.eingabe();
    b.eingabe();
    druckeTestfall(a,b);
    cout <<"\n Test mit verschiedenen Vorzeichen\n";
    a.set(3,7);
    b.set(6,13);
    druckeTestfall(a,b);
    a.set(3,-7);
    druckeTestfall(a,b);
    //...und so weiter
    cout <<"\n Test mit gemischten Datentypen\n";
    a.set(2301,77777);
    druckeTestfall(a,17);
    druckeTestfall(17, a);
    //...und so weiter
    cout <<"\n Test mit Nullwerten\n";  // ...und noch mehr
}
```

4.5 const-Objekte und Methoden

Objekte können wie einfache Variablen als *konstant* deklariert werden. Um jegliche Änderungen zu vermeiden, dürfen Methoden von konstanten Objekten nicht aufgerufen werden (ausgenommen Konstruktoren und Destruktoren), es sei denn, sie sind als konstante Elementfunktionen deklariert und definiert. Nehmen wir an, wir würden Bezug nehmend auf das Beispiel auf den Seiten 162 ff. eine konstante rationale Zahl definieren: `const Rational CR;`. Nehmen wir ferner an, wir hätten nach der Deklaration der Methode `ausgabe()` das Wort `const` vergessen. Der Aufruf `CR.ausgabe();` riefe dann eine Warnung oder Fehlermeldung des Compilers hervor. Wenn aber in der Deklaration und Definition das Schlüsselwort `const` angegeben wird, erhält die damit ausgezeichnete Methode das Privileg, für konstante (*und* nichtkonstante) Objekte aufgerufen werden zu können:

```
void ausgabe() const;                  // Deklaration
```

```
void Rational::ausgabe() const {   // Definition
    //.... wie vorher
}
```

Ein konstantes Objekt kann nicht durch `const`- oder andere Funktionen geändert werden, selbst dann nicht, wenn es per Referenz übergeben wird. Von dieser Regel gibt es zwei Ausnahmen:

1. Die `const`-Eigenschaft kann durch eine explizite Typumwandlung umgangen werden (englisch *casting the const away*) (siehe dazu Abschnitt 7.9).
2. In einer Klasse können Variablen mit dem Schlüsselwort `mutable` versehen werden. Es ist erlaubt, diese Attribute durch eine Methode zu ändern, auch wenn das Objekt konstant ist, zu dem das Attribut gehört. Der Sinn des Schlüsselworts liegt darin, dass eine Implementation sicherstellen will, dass einerseits Objekte nicht geändert werden, andererseits ein schneller Zugriff möglich sein soll, was die Änderung interner Verwaltungsinformationen erfordert. Beispielsweise könnte man sich in einer konstanten Liste die zuletzt benutzte Position für einen Zugriff merken (Stichwort »cache«), um beim nächsten Zugriff schnell zu sein.

Ein `const`-Qualifizierer wird auch beim Überladen von Methoden ausgewertet. Man kann *zwei* Methoden mit gleicher Parameterliste, aber verschiedener Wirkung, schreiben, die sich nur durch `const` unterscheiden:

```
void ausgabe();              // Methode 1
void ausgabe() const;        // Methode 2
```

In der Anwendung ruft der Compiler je nach Eigenschaft des Objekts die eine oder die andere Methode auf:

```
Rational r(6, 7);
const Rational CR(8, 9);
r.ausgabe();          // ruft Methode 1
CR.ausgabe();         // ruft Methode 2
```

Übung

4.3 Schreiben Sie eine Klasse `IntMenge`, bestehend aus den zwei Dateien *IntMenge.h* und *IntMenge.cpp*, sowie ein Testprogramm *main.cpp* entsprechend den Regeln dieses Kapitels. Die Klasse soll als mathematische Menge für ganze Zahlen nachbilden. Es sollen nur die folgenden einfachen Funktionen möglich sein, auf Operationen mit zwei Mengen wie Vereinigung und Durchschnitt werde verzichtet:

- `void hinzufuegen(int el)`: Element `el` hinzufügen, falls es noch nicht existiert, andernfalls nichts tun.
- `void entfernen(int el)`: Element `el` entfernen, falls es vorhanden ist, andernfalls nichts tun.
- `bool istMitglied(int el)`: Gibt an, ob `el` in der Menge enthalten ist.
- `size_t size()`: Gibt die Anzahl der gespeicherten Elemente zurück.
- `void anzeigen()`: Gibt alle Elemente auf der Standardausgabe aus.
- `void loeschen()`: Alle Elemente löschen.
- `int getMax()` und `int getMin()`: Geben das größte bzw. kleinste Element zurück.

Benutzen Sie intern zum Speichern der Werte ein `vector<int>`-Objekt (vgl. Abschnitt 1.9.2). Ein Auszug einer Anwendung könnte etwa wie folgt aussehen:

```
IntMenge menge;
menge.hinzufuegen(2); // ok
menge.hinzufuegen(-9); // ok
menge.hinzufuegen(2); // keine Wirkung, 2 gibt es schon
menge.entfernen(99); // keine Wirkung, nicht vorhanden
menge.entfernen(-9); // ok
menge.anzeigen();
menge.loeschen();
for(int i=17; i < 33; ++i) {
  menge.hinzufuegen(i*i);
}
cout << "Anzahl=" << menge.size() << " Minimum=" << menge.getMin();
if(menge.istMitglied(-11)) {
// ... usw.
```

Diese Aufgabe ist eine Vorübung für das Thema einer Klasse »Menge«. Die C++-Bibliothek stellt für Mengen die Klasse `set` zur Verfügung, die in Abschnitt 28.3.3 beschrieben wird.

4.6 Destruktoren

Destruktoren dienen dazu, Aufräumarbeiten für nicht mehr benötigte Objekte zu leisten. Wenn Destruktoren nicht vorgegeben werden, werden sie vom System automatisch erzeugt (implizite Deklaration). Der häufigste Zweck ist die Speicherfreigabe, wenn der Gültigkeitsbereich eines Objekts verlassen wird. Konstruktoren haben die Aufgabe, Res-

sourcen zu beschaffen, Destruktoren obliegt es, sie wieder freizugeben. Die Reihenfolge des Aufrufs der Destruktoren ist *umgekehrt* wie die der Konstruktoren.

Destruktoren haben keine Argumente und keinen Rückgabetyp. In der Deklaration wird eine Tilde ~ vorangestellt. Im Beispiel werden nummerierte Testobjekte erzeugt. Um den Ablauf verfolgen zu können, sind Konstruktor und Destruktor mit Ausgabeanweisungen versehen. Die Gültigkeit oder Lebensdauer eines Objekts endet, wie schon aus Abschnitt 1.7 bekannt, an der durch eine schließende geschweifte Klammer markierten Grenze des Blocks, in dem das Objekt definiert wurde. Genau dann wird das Objekt zerstört, das heißt, dass der von diesem Objekt belegte Speicherplatz freigegeben wird.

Daraus folgt, dass durchaus außerhalb von `main()` einige Aktivitäten stattfinden können:

- Falls es globale Objekte gibt, wird ihr Konstruktor *vor* der ersten Anweisung von `main()` aufgerufen.
- Innerhalb des äußersten Blocks von `main()` definierte Objekte werden erst beim Verlassen von `main()` freigegeben.
- Wegen der umgekehrten Reihenfolge der Destruktoraufrufe werden globale Objekte zuletzt freigegeben.

Die Ausgabe des Programms belegt, dass die Objekte *nach* der letzten Anweisung ihres Blocks zerstört werden.

Listing 4.9: Wirkung des Destruktors

```
// cppbuch/k4/destrukt.cpp
#include<iostream>
using namespace std;

class Beispiel {
    int zahl;                          // zur Identifizierung

  public:
    Beispiel(int i = 0);               // Konstruktor
    ~Beispiel();                       // Destruktor
};

Beispiel::Beispiel(int i)              // Konstruktor
: zahl(i) {
    cout << "Objekt " << zahl << " wird erzeugt.\n";
}

Beispiel::~Beispiel() {                // Destruktor
    cout << "Objekt " << zahl << " wird zerstört.\n";
}

// globale Variable, durch Vorgabewert mit 0 initialisiert
Beispiel ein_globales_Beispiel;

int main() {
    cout << "main wird begonnen\n";
    Beispiel einBeispiel(1);
    {
        cout << "   neuer Block\n    ";
```

```
        Beispiel einBeispiel(2);
        cout << "   Block wird verlassen\n    ";
    }
    cout << "main wird verlassen\n";
}
```

Die Ausgabe des Programms ist:

Objekt 0 wird erzeugt.
main wird begonnen
Objekt 1 wird erzeugt.
neuer Block
Objekt 2 wird erzeugt.
Block wird verlassen
Objekt 2 wird zerstört.
main wird verlassen
Objekt 1 wird zerstört.
Objekt 0 wird zerstört.

Der Destruktor statischer Objekte (static oder globale Objekte) wird nicht nur beim Verlassen eines Programms mit `return`, sondern auch bei Verlassen mit `exit()` aufgerufen. Im Gegensatz zum normalen Verlassen eines Blocks wird der Speicherplatz bei `exit()` jedoch nicht freigegeben.

4.7 Wie kommt man zu Klassen und Objekten? Ein Beispiel

Es kann hier keine allgemeine Methode gezeigt werden, wie man von einer Aufgabe zu Klassen und Objekten kommen kann. Es wird jedoch anhand eines Beispiels ein erster Eindruck vermittelt, wie man von einer Problemstellung zum objektorientierten Programm kommen kann.

Simulation der Benutzung eines einfachen Getränkeautomaten

In der Kantine stehen Getränkeautomaten, unter anderem einer mit *ObjektCola*. Der Automat sei so eingestellt, dass eine Dose dieses Getränks 2 € kostet. Der Automat nimmt einen beliebigen Geldbetrag an. Wenn zu wenig eingeworfen wird, wird das eingeworfene Geld wieder herausgegeben. Wenn zu viel eingeworfen wird, wird eine Dose *ObjektCola* sowie der Restbetrag herausgegeben. Nach Geldeinwurf löst ein Knopfdruck die Prüfung des Geldbetrags und gegebenenfalls die Ausgabe einer Dose aus. Wenn keine Dose mehr vorrätig ist, ist der Automat gesperrt, das heißt, dass jeder eingeworfene Geldbetrag vollständig zurückgegeben wird. Das folgende, bewusst einfach gehaltene Szenario soll mit einem Programm simuliert werden:

Szenario

Der Automat wird anfangs mit 50 Dosen befüllt. Zum Automaten gehen Personen, die eine Dose aus dem Automaten ziehen wollen. Sie tragen unterschiedliche Geldbeträge bei sich. Wenn der Betrag ausreicht, werfen sie Münzen in den Automaten, wobei manche nicht genau zählen und zufällig zu viele oder zu wenige Münzen einwerfen. Anschließend drücken sie auf den Ausgabeknopf mit der Wirkung, dass gegebenenfalls eine Dose und Rückgeld ausgegeben werden. Das Szenario endet, wenn der Automat gesperrt wird, weil er leer ist.

Das Programm soll zur Kontrolle die einzelnen Schritte auf dem Bildschirm dokumentieren. Der Einfachheit halber genügt es, einheitliche Münzen anzunehmen, zum Beispiel 1-€-Stücke.

4.7.1 Einige Analyse-Überlegungen

Um die Problemstellung zu verdeutlichen, wird sie aus verschiedenen Blickwinkeln betrachtet. Es handelt sich dabei nur um *Möglichkeiten*, nicht um den einzig wahren Lösungsansatz (den es nicht gibt).

1. In der Analyse geht es zunächst einmal darum, Prozesse *in der Sprache des (späteren Programm-) Anwenders* zu beschreiben. Dabei sind typische Abläufe, Szenarien genannt, ein gutes Hilfsmittel. Die Aufgabenstellung ist als Szenario formuliert.
2. Im zweiten Schritt wird versucht, beteiligte Objekte, ihr Verhalten und ihr Zusammenwirken zu identifizieren. Dies ist nicht unbedingt einfach, weil spontan identifizierte Beziehungen zwischen Objekten im Programm nicht immer die wesentliche Rolle spielen. Auf einem geeigneten Abstraktionsgrad muss entschieden werden, was zum »System« und was zur »Außenwelt« gehören soll.

Strukturierung des Ablaufs

Das Szenario beschreibt einen *Vorgang*, dessen einzelne Schritte als Pseudocode strukturiert dargestellt werden können.

Anfang des Szenarios

```
01 Automat mit 50 Dosen füllen.
02 Solange der Automat nicht gesperrt, das heißt nicht leer ist, wiederhole:
03    Eine durstige Person (z.B. eine Studentin) kommt vorbei, sie hat X € dabei.
04    X steht für eine zufällige Zahl.
05    Falls sie genug Geld hat (d.h. mindestens den Preis pro Dose).
06       steckt sie eine Anzahl Y Münzen in den Münzschlitz
07       und drückt auf den Knopf.
08       Falls Rückgeld ausgegeben wurde,
09          nimmt sie es.
10       Falls eine Dose ausgegeben wurde,
11          nimmt sie sie und
12          trinkt sie aus.
```

13 Andernfalls stellt sie fest: »Zu wenig Geld!«.

14 Die Person verlässt die Szene
(gegebenenfalls nicht mehr durstig und mit weniger Geld).

Ende des Szenarios

Objekte und Operationen identifizieren

Im nächsten Schritt wird versucht, die beteiligten Objekte und damit ihre Klassen zu identifizieren und eine Beschreibung ihres Verhaltens zu finden. Dazu wird auf die einzelnen nummerierten Zeilen des Szenarios Bezug genommen, die *kursiv* dargestellt sind. Bezeichnungen in `dieser Schrift` deuten auf *mögliche* Klassen- und Methodennamen. Es ist nicht nur das aktive Verhalten wichtig, sondern manchmal benötigt man außerhalb des Objekts Informationen über seinen inneren Zustand.

01 Automat mit 50 Dosen füllen.
Der Automat ist ein `GetraenkeAutomat`. Er enthält `Dosen`, deren `Anzahl` uns interessiert. Die Anzahl ist eine Eigenschaft des Automaten, nicht der Dose. Andere Automaten können gleichartige Dosen in anderer Stückzahl enthalten. Unklar ist hier, *wer* den Automaten füllt. Da wir uns in diesem Szenario nicht dafür interessieren, liegt die Antwort außerhalb unseres »Systems«. Wir müssen nur dafür sorgen, dass der Automat am Anfang des Szenarios ein paar Dosen enthält.

02 Solange der Automat nicht gesperrt, das heißt leer ist, wiederhole:
Der Automat hat ein Attribut `gesperrt`, das anzeigt, dass keine Dosen mehr da sind.

03 Eine Person (z.B eine Studentin) kommt vorbei, sie hat X €.
Anstelle einer Studentin könnte natürlich eine andere `Person` mit etwas `Geld` kommen. Die Aktivität hier ist `kommen`.

05 Falls sie genug Geld hat (d.h. mindestens den Preis pro Dose),
`Hat sie genug Geld`, verglichen mit dem `Preis pro Dose`, der in dem Automaten eingestellt ist? Die Menge des Geldes ist ein Attribut der Person, der Dosenpreis ein Attribut des Automaten.

06 steckt sie eine Anzahl Y Münzen in den Münzschlitz
Das `GeldEinwerfen` ist eine Aktivität der Studentin. Dabei ist wichtig, wie viele `Muenzen` in welchen `Getraenkeautomat` (hier gibt es nur einen) gesteckt werden. Der Automat muss die `Muenzen akzeptieren` und die `Anzahl` der `eingeworfenen Muenzen` kennen.

07 und drückt auf den Knopf.
`Knopf druecken` ist eine Aktivität der Studentin, die sich auf einen bestimmten Automaten bezieht. Gleichzeitig ist dies eine Aufforderung an den Automaten, das `Geld zu pruefen und eine Dose herauszugeben`.

08 Falls Rückgeld ausgegeben wurde,
09 nimmt sie es.
Nur wenn `Rueckgeld vorhanden` ist, kann sie `Geld entnehmen`. Rückgeld kann nur vorhanden sein, wenn es vorher eine `Geldrueckgabe` gab.

10 Falls eine Dose ausgegeben wurde,
11 nimmt sie sie und
12 trinkt sie aus.
Falls der Automat eine `Dose herausgegeben` hat, kann die Studentin die `Dose entnehmen`. Dann `trinkt` sie.

13 Andernfalls stellt sie fest: »Zu wenig Geld!«.
Die Studentin `sagt`....

14 Die Person verlässt die Szene (hoffentlich nicht mehr durstig und mit weniger Geld).
Die Aktivität ist hier `verlassen`.

Ende des Szenarios

Analyse einiger Aktivitäten

In der objektorientierten Programmierung kommunizieren Objekte durch Botschaften (englisch *message*). Das Wort »Botschaft« ist eine häufig benutzte, aber ungenaue Übersetzung, weil der Aufforderungscharakter unter den Tisch fällt. Besser ist es, nur von Aufforderungen zu sprechen, die an ein Objekt gerichtet sind. Die Notation ist *Objekt.Aufforderung(Daten)*. In den oben beschriebenen Aktivitäten stecken Aufforderungen:

zu 06:
Die Daten, die zu der Aktivität `Geld einwerfen` gehören, sind der `Automat`, in den die Münzen gesteckt werden, sowie die Anzahl der Münzen. In der obigen Notation geschrieben:
dieStudentin.GeldEinwerfen(derAutomat, Münzenanzahl)

Die Aufforderung an den Automaten, die *innerhalb* der Aktivität `Geld einwerfen` steckt, ist es, die `Muenzen zu akzeptieren`. Das Akzeptieren der Münzen beinhaltet die Kenntnis, wie viele Münzen eingeworfen wurden. In der obigen Notation geschrieben:
derAutomat.akzeptieren(Münzenanzahl)

zu 07:
Ähnliche Überlegungen sind für die Aktivität `Knopf druecken` anzustellen, weil diese Aktivität den Automaten dazu veranlasst, das eingeworfene Geld zu prüfen und eine Dose herauszugeben.

Erster Klassenentwurf

In der Analyse können Objekte identifiziert und damit schon erste Klassen gebildet werden. Die Aktivitäten oder das Verhalten der Objekte sind nach außen sichtbar und erscheinen deshalb im Programm als öffentliche Schnittstelle. Im Design werden die Klassen näher untersucht und genauer formuliert. Insbesondere ergeben sich aus den Aktivitäten, welche Informationen über ein Objekt benötigt werden bzw. welche *Attribute* es hat.

Die Klassen ergeben sich aus den handelnden Objekten mit ihren Attributen und Aktivitäten. Der Ansatz, zunächst die Substantive mit Objekten gleichzusetzen und die Verben mit Methoden, kann irreführend sein, weil beides austauschbar ist. Beispiel: »Die Entnahme der Dose wird von der Studentin durchgeführt.« Entnahme als Objekt? Durchführen als Aktivität? – etwas zweifelhaft. »Die Studentin entnimmt die Dose« ist erheblich klarer. Passivkonstruktionen sollten also vor der Analyse der Substantive und Verben stets in Aktivkonstruktionen verwandelt werden. Ebenso ist das Subjekt genau zu identifizieren.

»Die Messung erfolgt um 13 Uhr.« ist in diesem Sinne ein unbrauchbarer Satz, weil die Frage nach dem »wer?« nicht beantwortet werden kann.

Objekte sind aus dem vorherigen Abschnitt identifizierbar. Es gibt Objekte, die jedoch passiv sind: Dosen und Münzen. Zu diesen Objekten gibt es keine Attribute, es interessiert hier nur die Anzahl. Wenn zu einem Objekt keine Attribute und keine Aktivitäten (passiv!) angebbar sind, ist es im Allgemeinen nicht notwendig, es als Klasse zu formulieren. Die *aktiven*, handelnden Objekte sind die Studentin und der Automat. Da anstelle der Studentin (Spezialfall) auch andere Personen kommen können, ist es sinnvoll, dafür eine Klasse `Person` zu erfinden. Die Klasse für den Automaten nennen wir `GetraenkeAutomat`. Die *vorläufig* gefundenen Attribute und Aktivitäten sind in Tabelle 4.1 zusammengefasst.

Tabelle 4.1: Vorläufige Kandidaten für Klassen, Attribute und Aktivitäten

Klasse	Attribute	Aktivitäten/Zustandsabfragen
Person	Geldmenge	kommen hat genug Geld? Geld einwerfen Knopf drücken Geld entnehmen Dose entnehmen trinken gehen / Szene verlassen
GetränkeAutomat	Anzahl Dosen gesperrt Preis pro Dose Rückgeld eingeworfene Münzen	befüllt werden (von wem?) Geld prüfen Dose herausgegeben? Rückgeld vorhanden? Münzen akzeptieren Dose herausgeben ist gesperrt? Geldrückgabe

4.7.2 Formulierung des Szenarios in C++

Bei den identifizierten Objekten mit ihren Methoden handelt es sich *zunächst um eine erste Näherung*, die weiter verfeinert werden muss. Weil nur ein erster Eindruck vermittelt werden soll, wird auf eine vollständige objektorientierte Analyse (OOA) und ein entsprechendes Design (OOD) verzichtet und auf die Literatur verwiesen, die die OOA/D-Thematik ausführlich behandelt, zum Beispiel [Oe] oder [Bal]. Hier wird *als Starthilfe* das oben strukturiert beschriebene Szenario in einer möglichen C++-Darstellung formuliert. Es gibt nur einige Besonderheiten:

- `main()` beschreibt im Ablauf Aktivitäten einer Person, die ihrerseits Aktivitäten beim Getränkeautomaten auslösen.
- Der Automat wird mit Anfangswerten initialisiert (Konstruktor). Damit ist er am Anfang mit Dosen gefüllt, ohne dass wir uns Gedanken machen müssen, wer ihn gefüllt hat.
- Die Methoden `kommen` und `gehen` werden in C++ geeignet durch Konstruktor und Destruktor beschrieben.

Listing 4.10: Simulation des Getränkeautomaten

```
// /cppbuch/loesungen/k4/4/main.cpp
#include"person.h"
#include"automat.h"
using namespace std;

int zufall(int x) { // gibt eine Pseudo-Zufallszahl zwischen 0 und x zurück
   static long r = 1;
   r = (125 * r) % 8192;
   return (x+1)*r/8192;
}

int main() {
    const int DOSENANZAHL = 50,
              DOSENPREIS = 2,  // in €
              MAX_GELD   = 20; // in €
    // Szenario:
    // Der Konstruktor initialisiert den Automaten mit der
    // gewünschten Anzahl von Dosen und dem Dosenpreis.
    GetraenkeAutomat objektColaAutomat(DOSENANZAHL, DOSENPREIS);
    while(!objektColaAutomat.istGesperrt()) {
       // eine Person betritt die Szene:
       // Der Konstruktor erzeugt ein Objekt der Klasse Person (siehe Text) mit Namen
       // einePerson, wobei das Objekt (das heißt sein Geldvorrat) mit einer zufälligen
       // Zahl zwischen 0 und MAX_GELD initialisiert wird.
       Person einePerson(zufall(MAX_GELD));
       if(einePerson.genugGeld( objektColaAutomat.getraenkePreis())) {
          // eine zufällig gewählte Anzahl Münzen wird eingeworfen,
          // aber nicht mehr als vorhanden:
          int wieviel = zufall(einePerson.wievielGeld());
          einePerson.geldEinwerfen(objektColaAutomat, wieviel);
          einePerson.knopfDruecken(objektColaAutomat);
          if(objektColaAutomat.rueckgeldVorhanden())
             einePerson.geldEntnehmen( objektColaAutomat.geldRueckGabe());
          if(objektColaAutomat.doseHerausgegeben()) {
              einePerson.doseEntnehmen(objektColaAutomat);
              einePerson.trinkt();
          }
          if(objektColaAutomat.istGesperrt())
             cout << "Automat gesperrt! (leer)" << endl;
       }
       else einePerson.sagt(" Leider zu wenig Geld.");
    } // Blockende: die Studentin verlässt die Szene:
      // hier automatisch realisiert durch den Destruktor
}
```


Übungen

4.4 Versuchen Sie, die einzelnen Schritte nachzuvollziehen. Stellen Sie fest, wo es Unzulänglichkeiten und Unvollständigkeiten gibt. Entwerfen Sie die nötigen Klassen in C++, vervollständigen Sie das Programm und bringen Sie es zum Laufen. Das Programm sollte verschiedene Fälle abdecken (verschiedene, zufällig gewählte anfängliche Geldmengen),

die in der zu erzeugenden Testdokumentation aufgeführt werden. Die Testdokumentation erhält man am einfachsten durch Umleiten der Bildschirmausgaben in eine Datei.

Falls der Automat einer Methode als Parameter übergeben wird, die ihrerseits eine Methode des Automaten aufruft, welche den Zustand des Automaten ändert, ist nur die Übergabe per Referenz sinnvoll. Änderungen des Zustands des Automaten müssen im Original wirksam werden, nicht in einer Kopie, die am Ende der Methode weggeworfen wird. Dies kann erzwungen werden, wenn man den Kopierkonstruktor privat deklariert. Eine Definition ist nicht erforderlich, weil kein Aufruf erfolgt.

4.5 Wie können Sie mit C++ erreichen, dass ein Attribut *direkt*, also ohne Einsatz einer Methode, zwar gelesen, aber nicht verändert werden kann? Beispiel:

```
int main() {
    MeineKlasse objekt;
    objekt.readonlyAttribut = 999; // Fehler! nicht erlaubte Aktion
    // erlaubter direkter lesender Zugriff:
    cout << "objekt.readonlyAttribut="
         << objekt.readonlyAttribut << endl;    // ok
}
```

Wie sieht die Realisierung in der Klasse `MeineKlasse` aus?

4.6 Schreiben Sie auf Basis der Lösung der Taschenrechner-Aufgabe von Seite 134 eine Klasse `Taschenrechner`, die einen eingegebenen String verarbeitet. Die Anwendung könnte wie folgt aussehen:

```
int main() {
  while(true) {                  // Abbruch mit break
    cout << "Bitte einen mathematischen Ausdruck eingeben, z.B. 4*(12+3)"
         << "\n(Abbruch durch Eingabe einer Leerzeile) : ";
    string anfrage;
    getline(cin, anfrage);
    if(anfrage.length() > 0) {
       Taschenrechner tr(anfrage);
       cout << "Das Ergebnis der Anfrage '"
            << tr.getAnfrage() << "' ist " << tr.getErgebnis() << endl;
    }
    else break;
  }
}
```

Der auf den Seiten 119-120 verwendete Funktionsaufruf `cin.get(c);` muss dabei durch den Aufruf einer Funktion ersetzt werden, die das jeweils nächste Zeichen des übergebenen Anfrage-Strings holt.

4.8 Gegenseitige Abhängigkeit von Klassen

In der Lösung der Aufgabe des vorhergehenden Abschnitts werden in den inline-Funktionen der Klasse `Person` teilweise Methoden der Klasse `GetraenkeAutomat` benutzt, weswegen *automat.h* von *person.h* eingeschlossen wird. Hingegen benutzt die Klasse `GetraenkeAutomat` keinerlei Eigenschaften oder Methoden der Klasse `Person`. Was ist aber, wenn jede der beiden Klassen Methoden der jeweils anderen Klasse benutzt? Es nutzt nichts, gegenseitig die Header-Datei der jeweils anderen Klasse einzuschließen, weil der Compiler die nötigen Informationen nicht bekommt. Betrachten wir zwei Header-Dateien, die sich aufeinander beziehen:

```
// Datei A.h
#ifndef A_h
#define A_h
#include"B.h"
.... usw.
#endif
```

```
// Datei B.h
#ifndef B_h
#define B_h
#include"A.h"
.... usw.
#endif
```

Wenn *A.h* zuerst gelesen wird, wird bei Ausführung der dritten Zeile *B.h* eingelesen. Die Ausführung der dritten Zeile von *B.h* scheitert jedoch, weil `A_h` nun definiert ist und der Rest von *A.h* nicht zur Kenntnis genommen wird. Die Lösung des Problems besteht in der *Vorwärtsdeklaration*:

```
// Datei A.h
#ifndef A_h
#define A_h

class B; //Vorwärtsdeklaration

class A {
  public:
   void benutzeB(const B&);
   void eineAMethode();
   // ... usw.
};
#endif
```

```
// Datei B.h
#ifndef B_h
#define B_h

class A; //Vorwärtsdeklaration

class B {
  public:
   void machWasMitA(A*);
   void eineBMethode() const;
   // ... usw.
};
#endif
```

Die Notation `A*` bedeutet »Zeiger[2] auf Objekt der Klasse A«. In den Header-Dateien werden gegenseitig nur die Klassennamen bekannt gemacht, und die Kenntnisnahme der Methoden wird auf die Implementierungsdateien verschoben. Dies funktioniert dann, wenn die Header-Dateien ausschließlich Zeiger oder Referenzen der jeweils anderen Klasse enthalten, aber keine Methodenaufrufe. Dies kann leicht erreicht werden, wenn auf inline-Methoden verzichtet wird, die Methoden der anderen Klasse benutzen. Die dazu notwendige Struktur wird für zwei Klassen gezeigt, eine Erweiterung auf die gegenseitige

[2] Zeiger werden in Kapitel 5 besprochen, aber hier der Vollständigkeit halber mit erwähnt.

Abhängigkeit mehrerer Klassen ist nach diesem Muster leicht möglich. Die Implementierungsdateien schließen die Header-Dateien auch der anderen Klasse ein, wobei die Reihenfolge der Include-Anweisungen keine Rolle spielt. Nun können Methoden der jeweils anderen Klasse problemlos in den Implementierungsdateien aufgerufen werden.

```
// Datei A.cpp
#include"A.h"
#include"B.h"

void A::benutzeB(const B& b) {
  b.eineBMethode();
}

void A::eineAMethode() {
.... usw.
```

```
// Datei B.cpp
#include"B.h"
#include"A.h"

void B::machWasMitA(A* pA) {
  pA->eineAMethode();
}

void B::eineBMethode() const {
.... usw.
```

Es gibt natürlich auch Fälle, wo einer Klasse die Größe eines Objekts einer anderen Klasse bekannt sein *muss*, wenn zum Beispiel die Klasse `A` ein Objekt der Klasse `B` aggregiert. In diesem Fall hilft die Vorwärtsdeklaration, unsymmetrisch angewendet, ebenfalls weiter, wie das folgende Listing zeigt. Damit kann natürlich nicht der unwahrscheinliche Fall gelöst werden, dass die Klasse `A` ein Objekt der Klasse `B` aggregieren *und* Klasse `B` ein Objekt der Klasse `A` einschließen soll. Wer das unbedingt möchte, sollte seinen Entwurf noch einmal überdenken. Wenn es denn sein muss, kann dieser Fall mit als Zeiger auf `A` bzw. `B` realisierten Attributen gelöst werden, an die der Konstruktor zur Laufzeit dynamisch erzeugte Objekte hängt.

```
// Datei A.h
#ifndef A_h
#define A_h
// Klasse B einschließen:
#include"B.h"

class A {
  public:
   void benutzeB(const B&);
   void eineAMethode();
  private:
   B einB; // aggregiertes Objekt
   // .... usw.
};
#endif
```

```
// Datei B.h
#ifndef B_h
#define B_h
// Vorwärtsdeklaration:
class A;

class B {
  public:
   void machWasMitA(A*);
   void eineBMethode() const;
   // .... usw.
};
#endif
```

4.9 Konstruktor und mehr vorgeben oder verbieten[3]

Der Compiler erzeugt automatisch einen Konstruktor ohne Argumente, einen Kopierkonstruktor, einen Destruktor und einen Zuweisungsoperator, falls eine Klasse diese nicht zur Verfügung stellt. Um dem Programmierer mehr Kontrolle darüber zu erlauben, sind die syntaktischen Konstruktionen `= default` und `= delete` eingeführt worden. Beispiel:

```
class X {
public:
  X() = default; // Compiler-generierten Konstruktor erlauben, obwohl es schon
                 // X(int) gibt.
  X(int i);
  X(const X&) = delete;                  // Kopieren und
  X& operator=(const X&) = delete;       // Zuweisung verbieten
  void* operator new(std::size_t) = delete; // Erzeugung mit new verbieten
  void f(int);
  void f(double) = delete; // Aufruf von f(double) verbieten
};
```

Ohne die `f(double)`-Deklaration wäre wegen der automatischen Typumwandlung ein Aufruf von `f(int)` mit einem `double`-Argument möglich. Alternativ kann man die zu verbietenden Methoden in den `private`-Bereich legen. Dabei genügen die Prototypen; eine Implementation wird nicht gebraucht.

4.10 Delegierender Konstruktor[4]

Auf Seite 156 wird die Initialisierung eines Objekts mit Listen in der Konstruktordefinition gezeigt. Eine nach [ISOC++] neue Möglichkeit ist der Aufruf eines anderen Konstruktors derselben Klasse in der Initialisierungsliste. Der Konstruktor *delegiert* so seine Aufgabe an einen anderen Konstruktor. Der Sinn besteht darin, die Initialisierung von Attributen ohne Code-Duplikation zu erreichen. Ein Beispiel noch ohne delegierenden Konstruktor:

```
// cppbuch/k4/delegierenderKonstruktor/klasse1.h
class Klasse {
  public:
    Klasse(int a, int b)      // Konstruktor 1
    : attr1(a), attr2(b) {
      weitereInitialisierungen();
    }
```

[3] Dieser Abschnitt sollte beim ersten Lesen übersprungen werden. Zu seinem Verständnis wird die Kenntnis der Kapitel 5 und 9 vorausgesetzt.

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

```
    Klasse()                    // Konstruktor 2
    : attr1(1), attr2(42) {  // vorgegebene Werte
      weitereInitialisierungen();
    }
  private:
    void weitereInitialisierungen() {
      // Code für weitere Initialisierungen
    }
    int attr1;
    int attr2;
};
```

In diesem Beispiel ist ein Teil der Initialisierung in eine Funktion ausgelagert worden, die Initialisierung der Attribute `attr1` und `attr2` jedoch nicht. Im Fall von Anpassungen sind ggf. beide Konstruktoren zu ändern, was fehleranfällig sein kann. Eine Delegation der Konstruktoraufgaben würde Letzteres vermeiden helfen. Der Programmcode der Funktion `weitereInitialisierungen()` wird dabei in den Konstruktor verlegt, den der andere aufruft:

```
// cppbuch/k4/delegierenderKonstruktor/klasse2.h
class Klasse {
  public:
    Klasse(int a, int b)   // Konstruktor 1
    : attr1(a), attr2(b) {
      // Code für weitere Initialisierungen
    }
    Klasse()               // Konstruktor 2
    : Klasse(1, 42) {      // Delegation an Konstruktor 1
    }
  private:
    int attr1;
    int attr2;
};
```

Die Klasse ist kürzer und lesbarer geworden, weil der zweite Konstruktor nunmehr fast leer ist. Das folgende Beispiel zeigt, welcher Konstruktor aufgerufen wird:

```
Klasse k1(9);    // Konstruktor 1: Daten: 9, 17
Klasse k2(5, 20); // Konstruktor 1: Daten: 5, 20
Klasse k3;       // Konstruktor 2, Konstruktor 1 rufend: Daten: 1, 42
```