Kapitel 1
Eine erste Einführung

Die Sprache C++ ist weit verbreitet, wird vielfältig genutzt und zählt schon lange zu den wichtigsten Programmiersprachen.

Nachdem wir als Erstes geklärt haben, wie wir mit C++ arbeiten und welcher C++-Standard für uns wichtig ist, schreiben wir bereits das erste Programm.

1.1 Was machen wir mit C++?

Objektorientierung

C++ ermöglicht uns, rein objektorientiert zu programmieren, also mit *Klassen*, *Objekten*, *Eigenschaften* und *Methoden*.

Grundlagen

Als Vorbereitung dazu werden wir mit C++ zunächst klassisch prozedural arbeiten, also mit *Datentypen*, *Operatoren*, *Kontrollstrukturen* und *Funktionen* starten. Auf diese Weise erhalten Sie eine solide Grundlage, die Ihnen anschließend den Einstieg in die Objektorientierung erleichtert. Alle genannten Begriffe lernen Sie ausführlich kennen.

Bibliotheken

Es gibt zahlreiche Bibliotheken, die die Möglichkeiten von C++ erweitern. Die Sprache C++ ist ursprünglich als Weiterentwicklung zu der Sprache C entstanden. Aufgrund der Verwandtschaft ist es auch in C++ möglich, hardware-orientierte schnelle Programme zu entwickeln.

Hinweis

Im Buch wird nur mit vollständigen lauffähigen Beispielen gearbeitet. Sie finden sie auch im Downloadbereich zum Buch unter der Adresse *www.rheinwerk-verlag.de/4361*.

Beispiele

1.2 Was benötige ich zum Programmieren?

IDE

Für die Betriebssysteme Windows, Ubuntu Linux und macOS gibt es eine Reihe von frei verfügbaren IDEs (kurz für: *Integrated Development Environment*, dt.: Integrierte Entwicklungsumgebung), die Ihnen das bieten, was Sie als angehender C++-Programmierer benötigen:

Editor

- einen *Editor*, mit dessen Hilfe Sie den Programmcode schreiben

Compiler

- einen *Compiler*, mit dessen Hilfe Sie den Programmcode in die Sprache übersetzen, die der jeweilige Rechner versteht

Debugger

- einen *Debugger*, der Ihnen bei der Fehlersuche hilft

Viele Editoren bieten zur Erleichterung ein sogenanntes *Syntax-Highlighting*, also ein farbliches Hervorheben der Elemente von C++.

Installation

Die Installation und vor allem die Nutzung verschiedener Entwicklungsumgebungen (unter anderem *Eclipse*, *NetBeans*, *Xcode* und *Orwell Dev C++*) für Windows, Ubuntu Linux und macOS werden in Anhang A, »Installationen«, beschrieben.

1.3 Die Entwicklung von C++

Standards

C++ wurde im Jahre 1979 von Bjarne Stroustrup entwickelt. Über die Jahre hinweg hat die Sprache viele Neuerungen erfahren. Im Jahre 1998 erschien die standardisierte Version C++98. Weitere Verbesserungen sind mit den Standards C++03, C++11 und C++14 in den Jahren 2003, 2011 und 2014 eingeflossen. Aktuell ist die Entwicklung der Version C++17 bereits weit fortgeschritten.

C++11

Sie sollten mit einer Umgebung arbeiten, die den Standard C++11 berücksichtigt. In Anhang A, »Installationen«, wird die Installation der verschiedenen Entwicklungsumgebungen beschrieben inklusive der Einstellungen für diesen Standard. Einige Gründe für C++11:

Umsetzung

- Die verfügbaren C++-Compiler setzen zwar meist vollständig die Standards bis einschließlich C++11 um, aber nicht immer alle Bestandteile der nachfolgenden Versionen.

Einsteiger

- Die Standards bis einschließlich C++11 enthalten im Gegensatz zu den nachfolgenden Versionen sehr viele Neuerungen, die auch für den Einsteiger interessant sind.

Qt

- Für die Entwicklung von Programmen mit Benutzeroberflächen (siehe Kapitel 16, »Grafische Benutzeroberflächen mit der Qt-Bibliothek«) und für den Zugriff auf Datenbanken (siehe Kapitel 17, »Datenbanken mit SQLite verwalten«) wird die betriebssystemunabhängige Entwicklungsumgebung *Qt* genutzt. Seit der Version 5.7 von Juni 2016 baut Qt vollständig auf C++11 auf.

C++-Referenz

Falls Sie weitere Informationen zur Sprache C++ und der Zuordnung ihrer Elemente zu den verschiedenen Versionen benötigen, empfehle ich die C++-Referenz unter der folgenden Adresse: *http://cppreference.com*.

1.4 So sieht das erste Programm aus

Aufbau

In diesem Abschnitt lernen Sie das erste C++-Programm kennen. Es geht zunächst nur um das Verständnis für den Aufbau des Programms. Es wird eine Reihe von Begriffen eingeführt. Ihre Bedeutung wird erläutert, zudem werden sie Ihnen im Verlauf des Buchs immer vertrauter.

Hallo Welt

Das nachfolgende Programm gibt den Text `Hallo Welt` auf dem Bildschirm aus, gefolgt von einem Zeilenumbruch:

```
#include <iostream>
using namespace std;

int main()
{
   cout << "Hallo Welt" << endl;
}
```

Listing 1.1 Datei »hallo.cpp«

cout

Die entscheidende Zeile des Programms beginnt mit `cout`. Das Objekt `cout` dient zur Ausgabe von Daten auf dem Bildschirm. Der Operator `<<` versorgt das Objekt `cout` mit den Informationen, die angezeigt werden.

endl

Zunächst wird der Text `Hallo Welt` ausgegeben. Texte werden in doppelten Anführungsstrichen notiert. Nach dem Text folgt der *Manipulator* `endl`, der einen Zeilenumbruch erzeugt. Manipulatoren dienen zur Gestaltung der Ausgabe. Mehr dazu in Abschnitt 2.5, »Zahlen formatieren mit Manipulatoren«. Der Begriff `endl` steht abkürzend für *End of Line* (dt.: Ende der Zeile).

main() Ein einfaches Programm besteht aus einer oder mehreren Anweisungen innerhalb der Funktion `main()`, die der Reihe nach ausgeführt werden. Jede Anweisung wird durch ein Semikolon am Ende begrenzt. Die Zeile mit `cout` beinhaltet eine Anweisung.

{ ... } Der Aufbau der Funktion `main()` beginnt mit `int main()`. Die Bedeutung von `int` und den runden Klammern werden Sie noch kennenlernen. Nach den runden Klammern folgen die Anweisungen der Funktion `main()` innerhalb von geschweiften Klammern. Sie erreichen sie über die Tastenkombinationen Alt Gr + 7 beziehungsweise Alt Gr + 0. Es empfiehlt sich, die Anweisungen wie im vorliegenden Fall einzurücken. Sie erhöhen damit die Lesbarkeit Ihrer Programme.

iostream, std Die Bibliothek `iostream` ermöglicht die Ausgabe auf dem Bildschirm und die Eingabe von der Tastatur. Sie muss mithilfe von `#include <iostream>` eingebunden werden. Das Objekt `cout` und der Manipulator `endl` stammen aus dem Namensraum (engl.: *namespace*) `std`. Seine Benutzung muss mithilfe der Anweisung `using namespace std;` bekannt gemacht werden.

Hinweis

In Anhang A, »Installationen«, wird erläutert, wie Sie Ihre Entwicklungsumgebung unter Ihrem Betriebssystem nutzen können. Sie geben darin das Programm ein, speichern es in der Datei *hallo.cpp*, übersetzen es und führen es aus.

1.5 Kommentieren Sie Ihre Programme

Erläuterung Es kann sein, dass Sie sich selbst nach längerer Zeit wieder mit einem Ihrer Programme beschäftigen oder eines Ihrer Programme an einen anderen Entwickler weitergeben. In diesen Fällen sind Kommentare innerhalb Ihrer Programme sehr nützlich. Sie werden nicht übersetzt und dienen nur zur Erläuterung des Ablaufs.

Im nachfolgenden Programm sehen Sie einige Kommentare:

```
#include <iostream>
using namespace std;

int main()
{
```



```
   /* Ein Kommentar
      über mehrere Zeilen */
   cout << "Hallo Welt" << endl;

   // Ein einzeiliger Kommentar
   cout << "Das ist die zweite Zeile" << endl;

   cout << "Ende" << endl;   // Ende des Programms
}
```

Listing 1.2 Datei »kommentar.cpp«

/* */ und // Sie können einen längeren Kommentar über mehrere Zeilen notieren. Er muss innerhalb der Zeichenkombinationen /* und */ stehen. Ein kurzer Kommentar nach der Zeichenkombination // erstreckt sich nur bis zum Ende der Zeile. Ein Kommentar kann auch nach einer Anweisung in derselben Zeile beginnen.

Das Programm gibt drei Zeilen aus:

```
Hallo Welt
Das ist die zweite Zeile
Ende
```

Kapitel 2
Arbeiten mit Zahlen und Operatoren

Sie speichern Zahlenwerte, geben sie auf dem Bildschirm aus, lesen sie von Tastatur ein und rechnen mit ihnen.

Variablen

Innerhalb eines Programms können sowohl Zahlen als auch Texte in sogenannten *Variablen* gespeichert werden. Die Zahlen und Texte können aus Eingaben des Benutzers stammen. Sie können aber auch Ergebnisse und andere Informationen beinhalten, die das Programm ermittelt hat.

2.1 Wie speichere ich Zahlen?

int, double

Zunächst geht es nur um die Arbeit mit Zahlen. Es gibt *ganze Zahlen* ohne Stellen nach dem Komma und *Fließkommazahlen* mit Stellen nach dem Komma. Variablen zur Speicherung besitzen einen *Datentyp*, zum Beispiel `int` für ganze Zahlen und `double` für Fließkommazahlen. Zudem besitzen Variablen individuelle Namen, mit denen sie eindeutig voneinander unterschieden werden.

Anzahl, Preis

Nehmen wir an, wir möchten die Daten eines Einkaufs festhalten. Es werden zwei Exemplare eines Artikels erworben, der einen bestimmten Preis hat. Mit dem nachfolgenden Programm speichern wir diese Informationen und geben sie auf dem Bildschirm aus:

```
#include <iostream>
using namespace std;
int main()
{
   int anzahl;
   double preis;

   anzahl = 2;
   preis = 1.45;
```

```
   cout << "Anzahl: " << anzahl << endl;
   cout << "Preis: " << preis << " Euro" << endl;
}
```

Listing 2.1 Datei »datentyp_einfach.cpp«

Deklaration

Zunächst erfolgt eine *Deklaration*. Sie dient dazu, dem Programm die genutzten Variablen bekannt zu machen. Im vorliegenden Programm werden die Variable `anzahl` vom Datentyp `int` und die Variable `preis` vom Datentyp `double` deklariert.

Zuweisung

Es folgen zwei Zuweisungen: `anzahl` erhält den ganzzahligen Wert 2 und `preis` den Fließkommawert 1.45. Beachten Sie, dass ein Punkt als Dezimaltrennzeichen verwendet werden muss. Bei einer Zuweisung erhält die Variable auf der linken Seite des Zeichens `=` den Wert, der sich auf der rechten Seite des Gleichheitszeichens ergibt.

Ausgabe

Die Ausgabe auf dem Bildschirm erfolgt mithilfe von `cout`. Dabei werden nacheinander mehrere Texte und die Werte der genannten Variablen ausgegeben, jeweils getrennt durch den Operator `<<`.

Die Ausgabe des Programms:

```
Anzahl: 2
Preis: 1.45 Euro
```

Dezimalpunkt

Wie Sie sehen, wird sowohl im Programmcode als auch in der Ausgabe ein Punkt als Dezimaltrennzeichen verwendet. Zum besseren Verständnis arbeite ich daher auch im erläuternden Text mit dem Dezimalpunkt.

Namensregeln

Einige Hinweise

Der Name einer Variablen muss gemäß den Namensregeln von C++ gebildet werden:

- Er muss mit einem Buchstaben oder einem Unterstrich beginnen. Anschließend dürfen Buchstaben, Zahlen oder Unterstriche folgen.
- Zwischen Groß- und Kleinschreibung wird unterschieden.
- Nicht vorkommen dürfen zum Beispiel: Leerzeichen, Sonderzeichen, die deutschen Umlaute (ä, ö, ü) oder das scharfe ß.
- Der Name sollte keinem Schlüsselwort der Sprache C++ entsprechen, siehe Abschnitt B.2.4, »Schlüsselwörter der Sprache C++«.

camelCase

Er sollte etwas über den Inhalt einer Variablen aussagen, damit die Lesbarkeit eines Programms verbessert wird. Bei längeren Namen sollten Sie die

camelCase-Schreibweise verwenden. Dabei besteht ein Name aus mehreren Wörtern, die ohne Leerzeichen nacheinander notiert werden. Der erste Buchstabe des zweiten und aller folgenden Wörter wird großgeschrieben. Ein Beispiel: `anzahlApfelRot`.

Zufälliger Wert

Falls Sie einer Variablen keinen Wert zuweisen, besitzt sie einen zufälligen Wert. Bei einer Berechnung oder einer Ausgabe führt das zu nicht vorhersagbaren Ergebnissen.

2.2 Rechnen mit Operatoren

Operatoren + - * /

Zur Durchführung von Berechnungen stehen Ihnen die Operatoren `+` für die Addition, `-` für die Subtraktion, `*` für die Multiplikation und `/` für die Division zur Verfügung.

Punkt- vor Strichrechnung

Rechenausdrücke mit mehreren Operatoren werden von links nach rechts bearbeitet. Wie in der Mathematik werden Ausdrücke, die innerhalb von runden Klammern stehen, vorrangig berechnet. Zudem gilt: Punktrechnung (*, /) vor Strichrechnung (+, -).

Im nachfolgenden Programm werden einige Berechnungen mit den Daten eines Einkaufs durchgeführt. Dabei werden 2 Äpfel à 1.45 € und 4 Birnen à 0.85 € erworben.

```
#include <iostream>
using namespace std;
int main()
{
   int anzahlApfel = 2, anzahlBirne = 4;
   double preisApfel = 1.45, preisBirne = 0.85;
   double summe, mittel, anteil, differenz;

   summe =
      anzahlApfel * preisApfel + anzahlBirne * preisBirne;
   mittel = summe / (anzahlApfel + anzahlBirne);
   anteil = 100.0 * anzahlBirne / (anzahlApfel + anzahlBirne);
   differenz = preisApfel - preisBirne;

   cout << "Summe: " << summe << " Euro" << endl;
   cout << "Mittlerer Preis: " << mittel << " Euro" << endl;
```

```
    cout << "Anteil: " << anteil << " %" << endl;
    cout << "Differenz: " << differenz << " Euro" << endl;
}
```

Listing 2.2 Datei »operator_rechnen.cpp«

Initialisierung

Die `double`-Variablen `summe`, `mittel`, `anteil` und `differenz` werden zur Speicherung der Rechenergebnisse benötigt. Mehrere Variablen desselben Datentyps können innerhalb einer Anweisung deklariert werden. Falls der Wert einer Variablen zu Beginn des Programms bekannt ist, kann er bereits bei der Deklaration zugewiesen werden. Dieser Vorgang wird auch *Initialisierung* genannt.

Punkt- vor Strichrechnung

Zunächst wird der Gesamtbetrag für diesen Einkauf berechnet und in der Variablen `summe` gespeichert. Die Reihenfolge: Erst werden die beiden Multiplikationen durchgeführt, dann die Addition, dann die Zuweisung an die Variable `summe`.

Runde Klammern

Der mittlere Preis eines Artikels bei diesem Einkauf ergibt sich aus dem Gesamtbetrag, geteilt durch die Anzahl aller Artikel. Die Addition muss innerhalb von runden Klammern stehen, ansonsten würde die Division als Erstes ausgeführt werden.

Ganzzahl-Division

Falls man die Anzahl der Birnen durch die Anzahl aller Artikel teilt, ergibt sich der zahlenmäßige Anteil der Birnen am Einkauf. Das Problem dabei: Es handelt sich um eine *Division* von zwei ganzen Zahlen. Dabei werden in C++ die Stellen nach dem Komma abgeschnitten. Im vorliegenden Fall ergäbe sich der Wert 0.

Wir müssen also dafür sorgen, dass an der Division mindestens ein Fließkommawert beteiligt ist, zum Beispiel durch die Multiplikation des ersten Werts mit dem Fließkommawert 1.0. Da wir das Ergebnis ohnehin mit 100 multiplizieren müssen, um den Anteil in Prozent zu erhalten, schlagen wir zwei Fliegen mit einer Klappe: Wir multiplizieren den ersten Wert mit 100.0.

Als Letztes wird die Preisdifferenz zwischen Äpfeln und Birnen berechnet. Es folgt die Ausgabe der Ergebnisse:

```
Summe: 6.3 Euro
Mittlerer Preis: 1.05 Euro
Anteil: 66.6667 %
Differenz: 0.6 Euro
```


In Abschnitt 2.5, »Zahlen formatieren mit Manipulatoren«, wird erläutert, wie Sie die Anzahl der Stellen nach dem Komma einstellen. In obigem Programm wird die Anzahl aller Artikel zweimal berechnet. Normalerweise berechnet man ein solches Zwischenergebnis nur einmal und speichert es dann in einer eigenen Variablen. Hier wird aber die Bedeutung der runden Klammern gezeigt.

Hinweis

Längere Anweisung

Eine längere Anweisung kann sich über mehrere Zeilen erstrecken, wie Sie bei der Berechnung des Gesamtbetrags sehen. Der notwendige Zeilenumbruch kann an vielen Stellen erfolgen, allerdings zum Beispiel nicht innerhalb des Namens einer Variablen oder innerhalb eines Textes, der in Anführungsstrichen steht.

2.3 Fehler suchen

Debugging

Sollte eines Ihrer Programme unerwartete Ergebnisse haben, so sollten Sie sich auf die Suche nach dem Fehler machen und diesen beseitigen. Das sogenannte *Debugging* kann Ihnen dabei helfen. Ich beschreibe es anhand der Entwicklungsumgebung Eclipse. In den anderen Umgebungen läuft es in ähnlicher Art und Weise.

Werte kontrollieren

Ich stelle es Ihnen bereits jetzt in einfacher Version vor, damit Sie sich an die Handhabung gewöhnen. Sie können Ihr Programm schrittweise ablaufen lassen, um die Werte bestimmter Variablen zu unterschiedlichen Zeitpunkten des Programms zu kontrollieren. In Abschnitt 5.9, »Fehler suchen«, kommen weitere Möglichkeiten im Zusammenhang mit Funktionen hinzu.

Haltepunkt setzen

Laden Sie das Projekt mit dem Programm *operator_rechnen.cpp* aus Abschnitt 2.2, »Rechnen mit Operatoren«, in Eclipse. Setzen Sie den Cursor zum Beispiel in die Zeile, in der die Variable `mittel` ihren Wert erhält. Nach Aufruf des Menüpunkts Run • Toggle Breakpoint (Tastenkombination Strg + ⇧ + B) erscheint vor der Zeile ein kleiner blauer Punkt. Die betreffende Zeile dient nun als Haltepunkt. Sie können innerhalb eines Programms mehrere Haltepunkte setzen. In Abbildung 2.1 sehen Sie das genannte Programm mit zwei Haltepunkten.

Haltepunkt entfernen

Falls Sie den Menüpunkt in derselben Zeile noch einmal aufrufen, wird der Haltepunkt wieder entfernt. Der Menüpunkt RUN • REMOVE ALL BREAKPOINTS entfernt alle gesetzten Haltepunkte auf einmal.

Bei der normalen Ausführung des Programms mithilfe des Menüpunkts RUN • RUN (Tastenkombination Strg + F11) ändert sich durch die Haltepunkte noch nichts.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5     int anzahlApfel = 2, anzahlBirne = 4;
6     double preisApfel = 1.45, preisBirne = 0.85;
7     double summe, mittel, anteil, differenz;
8
9     summe =
10       anzahlApfel * preisApfel + anzahlBirne * preisBirne;
11    mittel = summe / (anzahlApfel + anzahlBirne);
12    anteil = 100.0 * anzahlBirne / (anzahlApfel + anzahlBirne);
13    differenz = preisApfel - preisBirne;
14
15    cout << "Summe: " << summe << " Euro" << endl;
16    cout << "Mittlerer Preis: " << mittel << " Euro" << endl;
17    cout << "Anteil: " << anteil << " %" << endl;
18    cout << "Differenz: " << differenz << " Euro" << endl;
19 }
20
```

Abbildung 2.1 Zwei Haltepunkte

Debug-Modus

Falls Sie den Menüpunkt RUN • DEBUG wählen (Funktionstaste F11), wechselt das Programm in den Debug-Modus. Sie werden gefragt, ob Sie in eine andere Ansicht wechseln möchten. Ich bin aus Gründen der Übersicht in der gewohnten Ansicht geblieben.

Schritt für Schritt

Die erste Zeile des Programms ist nun farbig unterlegt. Das ist die aktuelle Zeile, in der das Programm zurzeit steht. Durch Aufruf des Menüpunkts RUN • STEP INTO (Funktionstaste F5) können Sie sich schrittweise durch das Programm bewegen. Die farbige Unterlegung wandert dabei mit der aktuellen Zeile. Falls Ausgaben getätigt werden, sind diese bereits im Ausgabefenster sichtbar.

Werte kontrollieren

Sie können sich jederzeit die Werte ansehen, die die verschiedenen Variablen vor der aktuellen farbig unterlegten Zeile haben. Lassen Sie dazu die Maus über dem Namen der betreffenden Variablen »schweben«. In Abbildung 2.2 ist die Zeile 12 aktuell. Die Variable `mittel` hat zuvor bereits ihren Wert von 1.05 erhalten. Die Maus schwebt (im Buch nicht sichtbar) über den Namen der Variablen `mittel`.

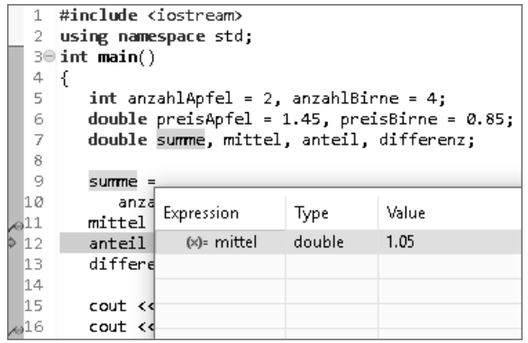

Abbildung 2.2 Kontrolle des Werts

Sprung zum Haltepunkt

Falls Sie bereits Haltepunkte gesetzt haben, können Sie sich durch Aufruf des Menüpunkts RUN • RUN TO LINE (Tastenkombination Strg + R) von Haltepunkt zu Haltepunkt bewegen. Das ermöglicht Ihnen, in schnellen Schritten die wichtigen Zeilen des Programms anzusteuern.

Debugging beenden

Sie können das Debugging jederzeit abbrechen mithilfe des Menüpunkts RUN • TERMINATE (Tastenkombination Strg + F2).

2.4 Wie können Daten eingegeben werden?

cin

Zur Eingabe von Daten durch den Benutzer des Programms werden das Objekt `cin` und der Operator `>>` benötigt. Vor einer Eingabe sollten Sie den Benutzer mithilfe einer Eingabeaufforderung darüber informieren, welche Daten er über die Tastatur eingeben soll.

Im nachfolgenden Programm wird der Benutzer dazu aufgefordert, erst eine Anzahl und anschließend einen Preis in Euro einzugeben:

```
#include <iostream>
using namespace std;
int main()
{
   int anzahl;
   double preis;

   cout << "Anzahl eingeben: ";
   cin >> anzahl;
```

```
   cout << "Preis in Euro eingeben: ";
   cin >> preis;

   cout << "Anzahl: " << anzahl << endl;
   cout << "Preis: " << preis << " Euro" << endl;
}
```

Listing 2.3 Datei »zahl_eingabe.cpp«

Die Ausgabe des Programms mit möglichen Eingaben des Benutzers:

```
Anzahl eingeben: 12
Preis in Euro eingeben: 1.65
Anzahl: 12
Preis: 1.65 Euro
```

Die Eingabeaufforderung endet häufig nicht mit einem Zeilenumbruch, sondern mit einem Leerzeichen. Der Benutzer kann dadurch seine Eingabe unmittelbar hinter der zugehörigen Aufforderung tätigen.

Programm hält an

Das Programm hält aufgrund der Anweisung mit `cin` an. Es erwartet eine Eingabe des Benutzers mit anschließender Betätigung der Taste ↵. Erst dann läuft das Programm weiter. Auch bei der Eingabe einer Fließkommazahl muss der Dezimalpunkt gesetzt werden.

Gültige Eingaben

Zur Vereinfachung gehen wir in den ersten Programmen davon aus, dass der Benutzer gültige Eingaben tätigt, also zum Beispiel keine Buchstaben eingibt. In Abschnitt 7.5, »Wie verarbeite ich Eingaben?«, wird erläutert, wie Sie ungültige Eingaben abfangen und verarbeiten können.

Datenstrom

Hinweis

Die Operatoren << und >> deuten die Richtung des Datenstroms an. Beim Operator << *strömen* die Daten zum `cout`, also zur Ausgabe. Beim Operator << *strömen* die Daten von der Eingabe mit `cin` in die Variable.

2.5 Zahlen formatieren mit Manipulatoren

Formatierte Ausgabe

Manipulatoren bieten Ihnen die Möglichkeit, Zahlen formatiert auszugeben. Ein Beispiel: Fließkommazahlen werden standardmäßig mit der vorhandenen Anzahl an Nachkommastellen ausgegeben, dabei jedoch maximal mit sechs Stellen. Bei einem Währungsbetrag bietet sich eine einheitliche Ausgabe mit zwei Nachkommastellen an.

In nachfolgendem Programm sehen Sie einige Möglichkeiten der Formatierung mithilfe von Manipulatoren:

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
   double preis = 1.4;
   cout << preis << " Euro" << endl;

   cout << setw(8) << preis << " Euro" << endl;

   cout << setfill('.');
   cout << setw(8) << preis << " Euro" << endl;

   cout << fixed << setprecision(2);
   cout << setw(8) << preis << " Euro" << endl;

   cout << left;
   cout << setw(8) << preis << " Euro" << endl;
}
```

Listing 2.4 Datei »manipulator.cpp«

iomanip

Es muss der Header `iomanip` für die Nutzung der Manipulatoren eingebunden werden. Zunächst die Ausgabe des Programms:

```
1.4 Euro
     1.4 Euro
.....1.4 Euro
....1.40 Euro
1.40.... Euro
```

Zunächst erfolgt eine unformatierte Ausgabe.

setw()

Die Funktion `setw()` führt zu einer Einstellung der Mindestbreite, allerdings nur für die unmittelbar nachfolgende Ausgabe. Hier wird der Wert der Variablen `preis` auf einer Breite von acht Stellen ausgegeben, standard-

mäßig rechtsbündig. Die verbleibenden fünf Stellen vor den drei Zeichen `1.4` werden standardmäßig mit Leerzeichen aufgefüllt.

setfill() Mithilfe der Funktion `setfill()` wählen Sie ein Füllzeichen für die Auffüllung auf die Mindestbreite. Die Funktion wirkt bis zum nächsten Aufruf von `setfill()`, maximal für den restlichen Verlauf des Programms.

fixed Die Angabe `fixed` dient zur Einstellung der *dezimalen Notation*, also der gewohnten Ausgabe von Fließkommazahlen. Die Angabe wirkt maximal für den restlichen Verlauf des Programms. Die Einstellung kann zum Beispiel durch die Angabe `scientific` geändert werden, zur Einstellung der *wissenschaftlichen Notation*. Diese nutzt zum Beispiel Exponenten bei sehr kleinen oder sehr großen Zahlen. Dazu mehr in Abschnitt 2.7.2, »Datentypen für Fließkommazahlen«.

setprecision() Nach der Angabe `fixed` können Sie mithilfe der Funktion `setprecision()` die Anzahl der Nachkommastellen einstellen. Die Zahl wird nur für die Ausgabe auf diese Stellenzahl gerundet. Ihr gespeicherter Wert bleibt unverändert. Die Funktion wirkt bis zum nächsten Aufruf von `setprecision()`, maximal für den restlichen Verlauf des Programms.

left Die Angabe `left` dient zur Einstellung der linksbündigen Ausgabe im Zusammenhang mit einer Mindestbreite. Die Angabe wirkt maximal für den restlichen Verlauf des Programms. Die Einstellung kann zum Beispiel durch die Angabe `right` geändert werden, zur Einstellung der rechtsbündigen Ausgabe.

2.6 Zuweisungen kürzer schreiben

Kombinierte Operatoren Sie haben die Möglichkeit, Zuweisungen mithilfe von kombinierten Operatoren zu verkürzen wie in nachfolgendem Programm:

```
#include <iostream>
using namespace std;
int main()
{
   int anzahl = 2;   cout << "Wert " << anzahl << endl;

   anzahl += 5;      cout << "+= 5 ergibt " << anzahl << endl;
   anzahl -= 3;      cout << "-= 3 ergibt " << anzahl << endl;
   anzahl *= 4;      cout << "*= 4 ergibt " << anzahl << endl;
   anzahl /= 2;      cout << "/= 2 ergibt " << anzahl << endl;

   anzahl++;         cout << "++ ergibt " << anzahl << endl;
   anzahl--;         cout << "-- ergibt " << anzahl << endl;
   ++anzahl;         cout << "++ ergibt " << anzahl << endl;
   --anzahl;         cout << "-- ergibt " << anzahl << endl;
}
```

Listing 2.5 Datei »operator_zuweisung.cpp«

Operator += Die Anweisung `anzahl+=5;` nutzt den kombinierten Zuweisungsoperator `+=`. Sie stellt eine verkürzte Form der Anweisung `anzahl=anzahl+5;` dar, also der Erhöhung des Werts der Variablen `anzahl` um 5. Auf diese Weise können Sie besonders bei langen Variablennamen die Lesbarkeit eines Programms verbessern. Entsprechendes gilt für die weiteren kombinierten Zuweisungsoperatoren `-=`, `*=` und `/=`.

Operatoren ++ -- Der *Inkrement*-Operator `++` führt zu einer Erhöhung des Werts einer Variablen um 1. `anzahl=anzahl+1` wird also verkürzt. `anzahl=anzahl-1` wird durch den *Dekrement*-Operator `--` verkürzt.

Es folgt die Ausgabe des Programms:

```
Wert 2
+= 5 ergibt 7
-= 3 ergibt 4
*= 4 ergibt 16
/= 2 ergibt 8
++ ergibt 9
-- ergibt 8
++ ergibt 9
-- ergibt 8
```

Lesbarkeit

Einige Hinweise

Die Anweisungen `b=++a;` und `b=a++;` haben unterschiedliche Auswirkungen, da die Zuweisung an `b` einmal vor und einmal nach der Erhöhung von `a` stattfindet. Erfahrene Programmierer schreiben gerne solche oder noch längere Anweisungen mit den Operatoren `++` und `--`. Für Einsteiger vermindert das allerdings die Lesbarkeit von Programmen. Daher empfehle ich Ihnen zunächst, die Operatoren `++` und `--` nur so einzusetzen, wie Sie es in meinem Programm sehen, und nicht in längeren Ausdrücken und Zuweisungen.

Semikolon

Zur besseren Übersicht stehen in diesem Programm jeweils zwei Anweisungen in einer Zeile. Das ist kein Problem, denn das Semikolon kennzeichnet das Ende der Anweisung.

2.7 Mehr über die Speicherung von Zahlen

Sie können an dieser Stelle bereits mit dem nächsten Kapitel fortfahren und die weiteren Abschnitte dieses Kapitels bei Bedarf später bearbeiten. Sie dienen der Ergänzung Ihres Wissens.

Speicherbedarf, Wertebereich

Für ganze Zahlen werden häufig Variablen des Datentyps `int` genutzt. Es gibt aber noch weitere ganzzahlige Datentypen. Ihre Gemeinsamkeit: Ganze Zahlen werden in den Variablen mathematisch genau gespeichert. Die verschiedenen Datentypen unterscheiden sich im Speicherbedarf und im gültigen Wertebereich.

Genauigkeit

Für Fließkommazahlen werden häufig Variablen des Datentyps `double` genutzt. Auch hier gibt es weitere Datentypen. Sie unterscheiden sich ebenfalls im Speicherbedarf und im gültigen Wertebereich. Zudem bieten sie eine unterschiedliche Genauigkeit. Sie sind allerdings niemals mathematisch genau.

2.7.1 Ganzzahlige Datentypen

Zunächst ein Programm mit den ganzzahligen Datentypen:

```
#include <iostream>
#include <climits>
using namespace std;
int main()
{
   cout << "char: " << sizeof(char) << " Byte, ";
   cout << "von " << CHAR_MIN << " bis " << CHAR_MAX << endl;

   cout << "unsigned char: " << sizeof(unsigned char) << " Byte, ";
   cout << "von 0 bis " << UCHAR_MAX << endl;

   cout << "short: " << sizeof(short) << " Byte, ";
   cout << "von " << SHRT_MIN << " bis " << SHRT_MAX << endl;
```



```
   cout << "unsigned short: "
      << sizeof(unsigned short) << " Byte, ";
   cout << "von 0 bis " << USHRT_MAX << endl;

   cout << "int: " << sizeof(int) << " Byte, ";
   cout << "von " << INT_MIN << " bis " << INT_MAX << endl;

   cout << "unsigned int: " << sizeof(unsigned int) << " Byte, ";
   cout << "von 0 bis " << UINT_MAX << endl;

   cout << "long: " << sizeof(long) << " Byte, ";
   cout << "von " << LONG_MIN << " bis " << LONG_MAX << endl;

   cout << "unsigned long: " << sizeof(unsigned long) << " Byte, ";
   cout << "von 0 bis " << ULONG_MAX << endl;
}
```

Listing 2.6 Datei »datentyp_ganz.cpp«

char, short, int, long

Es werden die ganzzahligen Datentypen `char`, `short`, `int` und `long` zusammen mit ihren vorzeichenlosen Varianten vorgestellt.

unsigned

Die Bezeichnung `unsigned` bedeutet vorzeichenlos, also ohne Vorzeichen. Das Gegenteil von `unsigned` ist `signed` und bedeutet vorzeichenbehaftet, also mit Vorzeichen. Variablen eines vorzeichenlosen Datentyps können nur den Wert 0 oder einen positiven Wert annehmen. Allerdings ist ihr Wertebereich doppelt so groß wie der Wertebereich des zugehörigen vorzeichenbehafteten Datentyps.

sizeof

Der Operator `sizeof` liefert den Speicherbedarf des genannten Datentyps oder der genannten Variablen in Byte.

Grenzen der Wertebereiche

Konstanten dienen zur Speicherung von unveränderlichen Werten. C++ stellt eine Reihe von *Systemkonstanten* zur Verfügung, unter anderem zur Speicherung der Grenzen der verschiedenen Wertebereiche. Sie werden mithilfe des Headers `climits` zur Verfügung gestellt. In Abschnitt 2.8, »Feste Werte in Konstanten speichern«, sehen Sie, wie Sie eigene Konstanten definieren können.

char

Der Datentyp `char` dient zur Speicherung von Zeichen. Intern wird dabei allerdings der Zeichencode, also eine ganze Zahl, gespeichert.

Die Angaben für den Speicherbedarf und den Wertebereich unterscheiden sich auch in Abhängigkeit von der genutzten Entwicklungsumgebung. Unter einem 32-Bit-System sehen Sie die folgende Ausgabe:

```
char: 1 Byte, von -128 bis 127
unsigned char: 1 Byte, von 0 bis 255
short: 2 Byte, von -32768 bis 32767
unsigned short: 2 Byte, von 0 bis 65535
int: 4 Byte, von -2147483648 bis 2147483647
unsigned int: 4 Byte, von 0 bis 4294967295
long: 4 Byte, von -2147483648 bis 2147483647
unsigned long: 4 Byte, von 0 bis 4294967295
```

In Abhängigkeit von dem genutzten Compiler kann es für den Datentyp `long` auch zu folgender Ausgabe kommen:

```
long: 8 Byte, von -9223372036854775808 bis 9223372036854775807
unsigned long: 8 Byte, von 0 bis 18446744073709551615
```

2.7.2 Datentypen für Fließkommazahlen

Es folgt ein Programm mit den Datentypen für Fließkommazahlen:

```
#include <iostream>
#include <iomanip>
#include <cfloat>
using namespace std;
int main()
{
   cout << "float: " << sizeof(float) << " Byte, ";
   cout << "von " << FLT_MIN << " bis " << FLT_MAX << endl;

   cout << "double: " << sizeof(double) << " Byte, ";
   cout << "von " << DBL_MIN << " bis " << DBL_MAX << endl;

   cout << "long double: " << sizeof(long double) << " Byte, ";
   cout << "von " << LDBL_MIN
      << " bis " << LDBL_MAX << endl;

   cout << fixed << setprecision(30);
   cout << "1/7 in float:       " << 1.0f / 7 << endl;
   cout << "1/7 in double:      " << 1.0 / 7 << endl;
   cout << "1/7 in long double: " << 1.0L / 7 << endl;
}
```

Listing 2.7 Datei »datentyp_fliesskomma.cpp«

float, double, long double

Sie sehen die Angaben für die Datentypen `float`, `double` und `long double`. Die Konstanten für die Grenzen der Wertebereiche werden mithilfe des Headers `cfloat` zur Verfügung gestellt.

Exponentialschreibweise

Bei der Ausgabe und bei der Eingabe sehr großer oder sehr kleiner Zahlenwerte empfiehlt sich die Exponentialschreibweise, also mit kleinen `e` oder auch großem `E`. Sie sehen sie bei der Ausgabe der Grenzen der Wertebereiche. Zwei weitere Beispiele:

$5.2e3 = 5.2 * 10^3 = 5.2 * 1000 = 5200$
$5.2e\text{-}3 = 5.2 * 10^{-3} = 5.2 * 0.001 = 0.0052$

Genauigkeit

Anhand der Division von 1 durch 7 wird die unterschiedliche Genauigkeit verdeutlicht. Das mathematisch genaue Ergebnis enthält unendlich oft die Zahlenfolge 142857. Beim Datentyp `float` wird ab der 9. Stelle nach dem Komma davon abgewichen, beim Datentyp `double` ab der 17. Stelle, beim Datentyp `long double` ab der 21. Stelle.

Die Fließkommazahl `1.0` stellt einen `double`-Wert dar, die Fließkommazahl `1.0f` einen `float`-Wert und die Fließkommazahl `1.0L` einen `long double`-Wert. Die Zahl `1` würde einen `int`-Wert darstellen.

Speicherbedarf, Wertebereich

Die Angaben für den Speicherbedarf und den Wertebereich unterscheiden sich auch hier in Abhängigkeit von dem genutzten Compiler. Besonders beim Datentyp `long double`, seinem Literal `L` und seinen Konstanten für den Wertebereich zeigen sich Unterschiede bei den Compilern.

Mit Dev C++ unter einem 64-Bit-Windows ergibt sich die folgende Ausgabe:

```
float: 4 Byte, von 1.17549e-038 bis 3.40282e+038
double: 8 Byte, von 2.22507e-308 bis 1.79769e+308
long double: 16 Byte, von 3.3621e-4932 bis 1.18973e+4932
1/7 in float:       0.142857149243354797363281250000
1/7 in double:      0.142857142857142849212692681249
1/7 in long double: 0.142857142857142857140921067549
```

2.8 Feste Werte in Konstanten speichern

Unveränderlich

Wie bereits in Abschnitt 2.7.1, »Ganzzahlige Datentypen«, erwähnt, dienen *Konstanten* zur Speicherung von unveränderlichen Werten. Sie werden unter anderem deshalb genutzt, weil man sich ihren Namen besser merken kann als den zugehörigen Wert. Im nachfolgenden Programm werden eine ganze Zahl und eine Fließkommazahl jeweils in einer Konstante gespeichert:

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
   const int FINGER = 5;
   // FINGER = 6;
   cout << "Finger: " << FINGER << endl;

   const double PI = 3.1415926;
   double radius = 3.8;
   double kreisUmfang = 2 * PI * radius;

   cout << fixed << setprecision(3);
   cout << "Kreisumfang: " << kreisUmfang << endl;
}
```

Listing 2.8 Datei »konstante_const.cpp«

const

Eine Konstante wird durch das Schlüsselwort `const` gekennzeichnet, das vor oder nach der Angabe des Datentyps notiert wird. Sie muss bei ihrer Deklaration einen Wert erhalten. Der Name einer Konstanten besteht häufig nur aus Großbuchstaben, damit sie besser zu erkennen ist. Die spätere Zuweisung eines Werts an eine Konstante ist nicht möglich.

Es folgt die Ausgabe des Programms:

```
Finger: 5
Kreisumfang: 23.876
```

Die zweite Konstante wird zur Berechnung eines Kreisumfangs genutzt.

2.9 Konstanten in Enumerationen zusammenfassen

Ganzzahliger Datentyp

Innerhalb einer *Enumeration* können Sie mehrere ganzzahlige Konstanten, die thematisch zusammengehören, zusammenfassen. Eine Enumeration entspricht einem vereinfachten ganzzahligen Datentyp, der nur einige ausgewählte Werte umfasst. Wie bei einzelnen Konstanten gilt: Die Namen der Elemente einer Enumeration kann man sich leichter merken als den zugehörigen Wert.

Nachfolgend wird eine Enumeration definiert und genutzt:

```
#include <iostream>
using namespace std;
int main()
{
   enum farbe { SCHWARZ, GELB, BLAU=8, ORANGE };
   farbe fa;
   fa = GELB;
   cout << "Farbe: " << fa << endl;
   fa = ORANGE;
   cout << "Farbe: " << fa << endl;
}
```

Listing 2.9 Datei »konstante_enum.cpp«

enum

Nach dem Schlüsselwort `enum` folgt der Name der Enumeration, anschließend die Namen ihrer Elemente in geschweiften Klammern.

Werte zuweisen

Falls Sie keine eigenen Werte zuweisen, erhalten die Elemente nacheinander die aufsteigenden Werte 0, 1, 2 und so weiter. Nach der Zuweisung eines eigenen Werts (hier: 8) basieren die Werte der jeweils nachfolgenden Elemente auf diesem Wert. Es ergibt sich also 9, 10, 11 und so weiter.

Im Programm wird die Variable `fa` des Enumerations-Datentyps `farbe` deklariert.

Die Ausgabe sieht wie folgt aus:

```
Farbe: 1
Farbe: 9
```

2.10 Übung

Eingabe, Berechnung, Ausgabe

Übung 2A: Entwickeln Sie ein Programm in der Datei *u_zahlen.cpp*. Im Programm wird der Benutzer dazu aufgefordert, nacheinander drei Zahlen einzugeben. Die Zahlen können Nachkommastellen haben. Das Programm gibt die eingegebenen Zahlen zur Kontrolle wieder aus. Anschließend werden die Summe und der Mittelwert der drei Zahlen berechnet und auf drei Nachkommastellen gerundet ausgegeben. Zu jeder Übung dieses Buchs finden Sie eine mögliche Lösung in Anhang B.

Schritt für Schritt

Gehen Sie bei der Entwicklung in den nachfolgend beschriebenen Schritten vor. Testen Sie Ihr Programm nach jedem Schritt. Sollte sich ein Fehler bei der Übersetzung und Ausführung ergeben, verbessern Sie ihn sofort, bevor Sie mit dem nächsten Schritt beginnen. Auf diese Weise wissen Sie genau, bei welcher Änderung sich der Fehler ergeben hat und an welcher Stelle des Programms er liegt. Vergessen Sie nicht, die Variablen zu deklarieren, die beim jeweiligen Schritt benötigt werden.

Die einzelnen Schritte:

- Lassen Sie den Benutzer die erste Zahl eingeben. Geben Sie die Zahl zur Kontrolle wieder aus.
- Lassen Sie den Benutzer die zweite und die dritte Zahl eingeben. Geben Sie alle drei Zahlen zur Kontrolle wieder aus.
- Berechnen Sie die Summe, und geben Sie diese ohne Rundung aus.
- Berechnen Sie den Mittelwert, und geben Sie ihn aus.
- Ändern Sie die Ausgaben, diesmal mit Rundung.

Ein Aufruf des fertigen Programms könnte wie folgt aussehen:

```
Erste Zahl eingeben: 3.2
Ihre erste Zahl: 3.2
Zweite Zahl eingeben: 5
Ihre zweite Zahl: 5
Dritte Zahl eingeben: 4.9
Ihre dritte Zahl: 4.9
Summe: 13.100
Mittelwert: 4.367
```

Kapitel 11
Vererbung und Polymorphie

Wenden Sie den Mechanismus der Vererbung an, um bereits vorhandene Eigenschaften und Methoden einfach zu übernehmen.

Verwandte Klassen

Eine Klasse kann ihre Elemente an eine andere Klasse vererben. Diese Klasse kann wiederum ihre Elemente an eine weitere Klasse vererben und so weiter. Durch diese aufeinanderfolgenden Vererbungsvorgänge erzeugen Sie nach und nach eine Hierarchie von miteinander verwandten Klassen, welche die Darstellung von Objekten mit teils übereinstimmenden, teils unterschiedlichen Merkmalen ermöglichen.

11.1 Basisklasse und abgeleitete Klassen

Allgemein/ spezialisiert

Eine Klasse, die Elemente einer anderen Klasse durch Vererben übernimmt, nennt man eine spezialisierte Klasse oder auch *abgeleitete Klasse*. Die Klasse, von der geerbt wurde, nennt man eine allgemeine Klasse oder auch *Basisklasse*.

Grafische Figuren

Im nachfolgenden Beispiel werden mehrere Klassen zur Speicherung und Bearbeitung der Daten von grafischen Figuren definiert, die sich innerhalb einer Zeichnung befinden. Es gibt die Basisklasse `figur` und die beiden von ihr abgeleiteten Klassen `rechteck` und `kreis`.

Basisklasse

Zunächst folgt der Beginn der Datei mit der Definition der Basisklasse `figur` und der Definition ihrer Methoden:

```
#include <iostream>
#include <string>
#include <cmath>
using namespace std;

class figur
{
```

```
   private:
      double x,y;
      string farbe;
   public:
      void werteZuweisen(const double&,
         const double&, const string&);
      void verschieben(const double &, const double&);
      void faerben(const string&);
      void ausgeben();
};

void figur::werteZuweisen(const double &xPos,
   const double &yPos, const string &f)
{
   x = xPos;
   y = yPos;
   farbe = f;
}

void figur::verschieben(const double &xDelta, const double &yDelta)
{
   x += xDelta;
   y += yDelta;
}

void figur::faerben(const string &f)
{
   farbe = f;
}

void figur::ausgeben()
{
   cout << "Figur: " << x << " / " << y << " " << farbe;
}
```

Listing 11.1 Datei »ableitung.cpp«, Teil 1 von 5

Drei Eigenschaften

Eine allgemeine grafische Figur ist durch ihren Ort innerhalb der Zeichnung gekennzeichnet, also ihre beiden `double`-Werte für die Koordinaten `x` und `y` entlang der entsprechenden Achsen. Zudem ist sie durch den `string`-Wert `farbe` für ihre Farbe gekennzeichnet. Von dieser Basisklasse `figur` können verschiedene Klassen für unterschiedliche geformte grafische Figuren abgeleitet werden, wie zum Beispiel Rechtecke oder Kreise.

Die Klasse `figur` umfasst vier Methoden:

Vier Methoden

- die Methode `werteZuweisen()`: zur Zuweisung von Werten für die drei Eigenschaften
- die Methode `verschieben()`: zur Verschiebung um bestimmte Werte (`xDelta` und `yDelta`) in x- und y-Richtung auf der Zeichnung. Dabei werden die Werte ihrer Koordinaten x und y verändert.
- die Methode `faerben()`: zur Änderung des `string`-Werts der Eigenschaft `farbe`
- die Methode `ausgeben()`: zur Ausgabe der Werte der drei Eigenschaften auf dem Bildschirm

Erste abgeleitete Klasse

Es folgt die Definition der Klasse `rechteck`, die von der Basisklasse `figur` abgeleitet wird, sowie die Definition ihrer Methoden:

```
class rechteck:public figur
{
   private:
      double hoehe, breite;
   public:
      void werteZuweisen(const double&, const double&,
         const string&, const double&, const double&);
      void skalieren(const double&, const double&);
      double flaeche();
      void ausgeben();
};

void rechteck::werteZuweisen(
   const double &xPos, const double &yPos,
   const string &f, const double &h, double const &b)
{
   figur::werteZuweisen(xPos, yPos, f);
   hoehe = h;
   breite = b;
}

void rechteck::skalieren(
   const double &hoeheFaktor, const double &breiteFaktor)
{
```

```
   hoehe *= hoeheFaktor;
   breite *= breiteFaktor;
}

double rechteck::flaeche()
{
   return hoehe * breite;
}

void rechteck::ausgeben()
{
   figur::ausgeben();
   cout << " Rechteck: " << hoehe << " / " << breite << endl;
}
```

Listing 11.2 Datei »ableitung.cpp«, Teil 2 von 5

Erbt von ... Nach dem Namen der Klasse `rechteck` folgt ein Doppelpunkt, das Schlüsselwort `public` und anschließend der Name der Klasse `figur`, von der die Klasse `rechteck` abgeleitet wird.

public Durch das Schlüsselwort `public` wird gekennzeichnet, dass die Vererbung vollständig und uneingeschränkt ist. Es gibt hier noch Varianten, die aber nicht Thema dieses Einsteigerbuchs sind.

Zwei weitere Eigenschaften Ein Rechteck innerhalb einer Zeichnung ist durch seinen Ort, seine Größe und seine Farbe gekennzeichnet. Die Eigenschaften `x` und `y` für den Ort und `farbe` für die Farbe werden geerbt, die Eigenschaften `hoehe` und `breite` stehen für seine Größe in x- und y-Richtung.

Nur indirekter Zugriff Allerdings haben Sie gemäß dem Prinzip der Datenkapselung innerhalb der Klasse `rechteck` keinen direkten Zugriff auf die geerbten Eigenschaften aus dem `private`-Bereich der Klasse `figur`, nur indirekt über die öffentlich zugänglichen Methoden aus dem `public`-Bereich der Klasse `figur`.

Vier weitere Methoden Die Klasse `rechteck` umfasst zusätzlich zu den vier geerbten Methoden vier eigene Methoden:

- die Methode `werteZuweisen()`: zur Zuweisung von Werten für die insgesamt fünf Eigenschaften. Unter anderem wird darin mithilfe des Scope-Operators die Methode `werteZuweisen()` der Basisklasse aufgerufen, um die Werte der ersten drei Parameter an die geerbten Eigenschaften weiterzureichen.

- die Methode `skalieren()`: zur Änderung der Größe des Rechtecks um bestimmte Werte (`hoeheFaktor` und `breiteFaktor`)
- die Methode `flaeche()`: zur Berechnung und Rückgabe der Fläche des Rechtecks
- die Methode `ausgeben()`: zur Ausgabe der Werte der insgesamt fünf Eigenschaften auf dem Bildschirm. Auch hier wird mithilfe des Scope-Operators die Methode `ausgeben()` der Basisklasse aufgerufen. Damit wird für die Ausgabe der Werte der geerbten Eigenschaften gesorgt.

Überschreiben An dieser Stelle sehen Sie den Vorgang des *Überschreibens*: Die beiden Methoden `werteZuweisen()` und `ausgeben()` der Basisklasse `figur` werden jeweils durch gleichnamige Methoden der Klasse `rechteck` überschrieben.

Zweite abgeleitete Klasse Es folgt die Definition der Klasse `kreis`, die ebenfalls von der Basisklasse `figur` abgeleitet wird, sowie die Definition ihrer Methoden:


```
class kreis:public figur
{
   private:
      double radius;
   public:
      void werteZuweisen(const double&,
         const double&, const string&, const double&);
      void skalieren(const double&);
      double flaeche();
      void ausgeben();
};

void kreis::werteZuweisen(const double &xPos,
   const double &yPos, const string &f, const double &r)
{
   figur::werteZuweisen(xPos, yPos, f);
   radius = r;
}

void kreis::skalieren(const double &radiusFaktor)
{
   radius *= radiusFaktor;
}

double kreis::flaeche()
{
```

```
   return 4 * atan(1.0) * radius * radius;
}

void kreis::ausgeben()
{
   figur::ausgeben();
   cout << " Kreis: " << radius << endl;
}
```

Listing 11.3 Datei »ableitung.cpp«, Teil 3 von 5

Eine weitere Eigenschaft

Ein Kreis innerhalb einer Zeichnung ist ebenso durch seinen Ort, seine Größe und seine Farbe gekennzeichnet. Die Eigenschaften `x` und `y` für den Ort und `farbe` für die Farbe werden geerbt, die Eigenschaft `radius` steht für seine Größe.

Vier weitere Methoden

Die Klasse `kreis` umfasst zusätzlich zu den vier geerbten Methoden ebenso vier eigene Methoden. Sie besitzen dieselben Aufgaben und einen ähnlichen Aufbau wie die gleichnamigen Methoden der Klasse `rechteck`. Aufgrund der unterschiedlichen Geometrie ergeben sich natürlich einige Abweichungen, zum Beispiel bei der Berechnung der Fläche.

Hauptprogramm, Teil 1

Es folgt der erste Teil des Hauptprogramms. Darin wird ein Rechteck erzeugt, verändert und ausgegeben:

```
int main()
{
   // Rechteck
   rechteck ra;
   ra.werteZuweisen(7.4, 2.3, "Blau", 4.0, 2.6);
   ra.ausgeben();

   ra.skalieren(0.5, 3.0);
   ra.verschieben(2.8, 4.2);
   ra.faerben("Schwarz");
   ra.ausgeben();

   cout << "Rechteck, Flaeche: " << ra.flaeche() << endl << endl;
```

Listing 11.4 Datei »ableitung.cpp«, Teil 4 von 5

rechteck-Objekt

Nach der Erzeugung des Rechtecks `ra` werden ihm Werte zugewiesen. Es erfolgt die erste Ausgabe. Nach dem Skalieren, Verschieben und Färben des Rechtecks erfolgt die zweite Ausgabe.

Methode suchen

Die Methoden werden zunächst in der Klasse des Objekts gesucht, also in der Klasse `rechteck`. Falls sie dort nicht gefunden werden, werden sie in der Klasse gesucht, von der die Klasse des Objekts abgeleitet wurde, also in der Klasse `figur`.

Die Ausgabe dieses Teils des Hauptprogramms:

```
Figur: 7.4 / 2.3 Blau Rechteck: 4 / 2.6
Figur: 10.2 / 6.5 Schwarz Rechteck: 2 / 7.8
Rechteck, Flaeche: 15.6
```

Hauptprogramm, Teil 2

Es folgt der zweite Teil des Hauptprogramms. Darin wird ein Kreis erzeugt, verändert und ausgegeben:

```
   // Kreis
   kreis ka;
   ka.werteZuweisen(4.6, 1.7, "Blau", 2.3);
   ka.ausgeben();

   ka.skalieren(2.0);
   ka.verschieben(3.3, 2.4);
   ka.faerben("Weiss");
   ka.ausgeben();

   cout << "Kreis, Flaeche: " << ka.flaeche() << endl;
}
```

Listing 11.5 Datei »ableitung.cpp«, Teil 5 von 5

kreis-Objekt

Nach der Erzeugung des Kreises `ka` werden auch ihm Werte zugewiesen. Es erfolgt die erste Ausgabe. Nach dem Skalieren, Verschieben und Färben des Rechtecks erfolgt ebenso die zweite Ausgabe.

Die Ausgabe dieses Teils des Hauptprogramms:

```
Figur: 4.6 / 1.7 Blau Kreis: 2.3
Figur: 7.9 / 4.1 Weiss Kreis: 4.6
Kreis, Flaeche: 66.4761
```

11.2 Welche Elemente sind an welcher Stelle erreichbar?

Kapselung

Gemäß dem Prinzip der Datenkapselung haben Sie innerhalb der Klasse `rechteck` keinen direkten Zugriff auf die geerbten Eigenschaften aus dem `private`-Bereich der Klasse `figur`, nur indirekt über die öffentlich zugänglichen Methoden aus dem `public`-Bereich der Klasse `figur`.

protected

Zwischen den Bereichen `private` und `public` kann es noch den Bereich `protected` geben. Elemente einer Klasse aus diesem Bereich sind nur innerhalb derselben Klasse oder innerhalb der von ihr abgeleiteten Klassen erreichbar. Sie sind nicht von außen, also etwa von einer nicht verwandten Klasse aus oder vom Hauptprogramm aus, erreichbar.

Basisklasse

Dies wird an einer Variante des Programms aus Abschnitt 11.1, »Basisklasse und abgeleitete Klassen«, gezeigt. Zunächst die geänderte Definition der Basisklasse:

```
class figur
{
   private:
      double x,y;
   protected:
      string farbe;
   public:
      void werteZuweisen(const double&,
         const double&, const string&);
      void ausgeben();
};
```

Listing 11.6 Datei »zugriffsbereich.cpp«, erster Ausschnitt

Eigenschaft verschieben

Die Eigenschaft `farbe` der Basisklasse wird aus dem Bereich `private` in den Bereich `protected` verschoben. Die Definitionen der Methoden ändern sich nicht. Es folgt die Definition der abgeleiteten Klasse:

```
class rechteck:public figur
{
   private:
      double hoehe, breite;
   public:
      void werteZuweisen(const double&, const double&,
         const string&, const double&, const double&);
      void ausgeben();
      void ausgebenFarbe();
};
...
void rechteck::ausgebenFarbe()
{
   cout << "Farbe: " << farbe << endl;
}
```

Listing 11.7 Datei »zugriffsbereich.cpp«, zweiter Ausschnitt

Weitere Methode

Es wird eine zusätzliche Methode `ausgebenFarbe()` definiert. Darin wird direkt auf die Eigenschaft `farbe` der Basisklasse zugegriffen. Die Definitionen der restlichen Methoden ändern sich nicht.

Es folgt das Hauptprogramm:

```
int main()
{
   rechteck ra;
   ra.werteZuweisen(7.4, 2.3, "Blau", 4.0, 2.6);
   ra.ausgeben();
   ra.ausgebenFarbe();
   // cout << ra.farbe << endl;
}
```

Listing 11.8 Datei »zugriffsbereich.cpp«, dritter Ausschnitt

Die Ausgabe des Programms sieht wie folgt aus:

```
Figur: 7.4 / 2.3 Blau Rechteck: 4 / 2.6
Farbe: Blau
```

Anderer Zugriff

Die Eigenschaft `farbe` kann direkt von der abgeleiteten Klasse aus erreicht werden. Der Zugriff aus dem Hauptprogramm würde nach wie vor zu einem Fehler führen.

11.3 Konstruktoren in abgeleiteten Klassen

Passende Konstruktoren

Sie können im Zusammenhang mit der Vererbung auch Konstruktoren und Destruktoren verwenden. Besonders bei den Konstruktoren müssen Sie darauf achten, dass sie aufeinander abgestimmt sind.

Reihenfolge

Ein Objekt einer abgeleiteten Klasse wird konstruiert, indem als Erstes der Konstruktor der Basisklasse durchlaufen wird. Anschließend wird der Konstruktor der abgeleiteten Klasse durchlaufen. Zum Ende der Existenz eines Objekts verläuft es umgekehrt: Zunächst wird der Destruktor der abgeleiteten Klasse durchlaufen, anschließend der Destruktor der Basisklasse.

Das wird anhand der beiden Klassen `figur` und `rechteck` verdeutlicht. Statt der Methode `werteZuweisen()` werden Konstruktoren verwendet. Zudem gibt es jeweils einen Destruktor.

Basisklasse

Es folgt der Beginn der Datei mit der Definition der Klasse `figur` und ihrer Methoden:

```
#include <iostream>
#include <string>
using namespace std;

class figur
{
   private:
      double x,y;
      string farbe;
   public:
      figur();
      figur(const double&, const double&, const string&);
      void ausgeben();
      ~figur();
};

figur::figur()
{
   cout << "figur-Konstruktor ohne Parameter" << endl;
}

figur::figur(const double &xPos,
   const double &yPos, const string &f)
{
   cout << "figur-Konstruktor mit Parametern" << endl;
   x = xPos;
   y = yPos;
   farbe = f;
}
```



```
void figur::ausgeben()
{
   cout << "Figur: " << x << " / " << y << " " << farbe;
}

figur::~figur()
{
   cout << "figur-Destruktor" << endl << endl;
}
```

Listing 11.9 Datei »konstruktor_ableitung.cpp«, Teil 1 von 3

Zwei Konstruktoren

Es gibt einen Konstruktor ohne Parameter und einen Konstruktor mit drei Parametern für die drei Eigenschaften der Klasse `figur`. Außerdem gibt es einen Destruktor. Zur Verdeutlichung der Reihenfolge geben alle Methoden eine kurze Information aus, dass sie durchlaufen werden.

Manche Compiler geben eine Warnung aus, falls eine Eigenschaft, wie im vorliegenden Fall, nicht im Konstruktor initialisiert wird. Das soll normalerweise nicht vorkommen. Hier dient der Konstruktor aber nur zu Ihrer Information über die Reihenfolge des Ablaufs.

Abgeleitete Klasse

Es folgt die Definition der Klasse `rechteck` und ihrer Methoden:

```
class rechteck:public figur
{
   private:
      double hoehe, breite;
   public:
      rechteck();
      rechteck(const double&, const double&,
         const string&, const double&, const double&);
      void ausgeben();
      ~rechteck();
};

rechteck::rechteck()
{
   cout << "rechteck-Konstruktor ohne Parameter" << endl;
}
```

```
rechteck::rechteck(
   const double &xPos, const double &yPos, const string &f,
   const double &h, const double &b):figur(xPos, yPos, f)
{
   cout << "rechteck-Konstruktor mit Parametern" << endl;
   hoehe = h;
   breite = b;
}

void rechteck::ausgeben()
{
   figur::ausgeben();
   cout << " Rechteck: " << hoehe << " / " << breite << endl;
}

rechteck::~rechteck()
{
   cout << "rechteck-Destruktor" << endl;
}
```

Listing 11.10 Datei »konstruktor_ableitung.cpp«, Teil 2 von 3

Für alle Eigenschaften

Es gibt einen Konstruktor ohne Parameter und einen Konstruktor mit allen fünf Parametern für die fünf Eigenschaften der Klasse `rechteck`. Außerdem gibt es einen Destruktor.

Parameter weitergeben

Der Konstruktor mit Parametern ruft den Konstruktor der Basisklasse `figur` ähnlich wie eine Methode auf. Das wird erreicht, indem nach der Liste der Parameter im Kopf der Methode ein Doppelpunkt folgt, anschließend der Name der Basisklasse `figur`, gefolgt von den drei Parametern, die weitergegeben werden. Es muss ein passender Konstruktor der Basisklasse zur Verfügung stehen.

Konstruktor ruft Konstruktor auf

Wie Sie an der Ausgabe weiter unten sehen, ruft auch der parameterlose Konstruktor den parameterlosen Konstruktor der Basisklasse auf. Auch dieser muss zur Verfügung stehen.

Es folgt das Hauptprogramm:

```
int main()
{
   rechteck ra(7.4, 2.3, "Blau", 4.0, 2.6);
   ra.ausgeben();
```



```
   cout << endl;

   rechteck rb;
   cout << endl;
}
```

Listing 11.11 Datei »konstruktor_ableitung.cpp«, Teil 3 von 3

Reihenfolge

Es wird ein Objekt der Klasse `rechteck` mit Werten und eines ohne Werte erzeugt. Anhand der nachfolgenden Ausgabe sehen Sie die Reihenfolge des Ablaufs bei der Erzeugung der Objekte und beim Ende der Existenz der Objekte.

```
figur-Konstruktor mit Parametern
rechteck-Konstruktor mit Parametern
Figur: 7.4 / 2.3 Blau Rechteck: 4 / 2.6

figur-Konstruktor ohne Parameter
rechteck-Konstruktor ohne Parameter

rechteck-Destruktor
figur-Destruktor

rechteck-Destruktor
figur-Destruktor
```


Hinweis

Das Objekt einer abgeleiteten Klasse wird erbaut wie ein Haus: zunächst das Erdgeschoss (die Basisklasse), anschließend weitere Etagen (die abgeleiteten Klassen). Zum Ende der Existenz wird das Objekt nacheinander von der obersten Etage bis zum Erdgeschoss abgebaut.

11.4 Was bedeutet Polymorphie?

Vielgestaltig

Polymorphie bedeutet Vielgestaltigkeit. Innerhalb der objektorientierten Programmierung bedeutet dieser Begriff, dass einem Objekt abhängig von seiner Nutzung unterschiedliche Klassen zugeordnet werden können. Ein Objekt der Klasse `rechteck` ist auch ein Objekt der Klasse `figur`. Dadurch kann auf die jeweils zugehörigen Objektelemente zugegriffen werden. Das

vergrößert die Flexibilität bei der Programmierung mit Objekten verwandter Klassen.

Feld von Zeigern

Im nachfolgenden Beispiel werden zunächst einige Objekte der miteinander verwandten Klassen `figur`, `rechteck` und `kreis` erzeugt. Anschließend wird ein `vector`-Feld von einfachen Zeigern auf Objekte der gemeinsamen Basisklasse `figur` erstellt. Die Adressen der soeben erstellten Objekte werden den Elementen des Zeigerfelds zugewiesen.

Gemeinsamer Zugriff

Auf diese Weise sind alle miteinander verwandten Objekte gemeinsam erreichbar. Innerhalb einer Schleife werden sie gemeinsam verschoben und ausgegeben. Zudem wird die Gesamtfläche aller Objekte ermittelt.

Basisklasse

Zunächst die Basisklasse `figur`. Es werden nur die Teile des Programms erläutert, die für die Thematik wichtig sind:

```
class figur
{
   private:
      double x,y;
      string farbe;
   public:
      figur(const double&, const double&, const string&);
      void verschieben(const double&, const double&);
      void faerben(const string&);
      virtual double flaeche(){return 0.0;};
      virtual void ausgeben();
};
```

Listing 11.12 Datei »polymorphie.cpp«, erster Ausschnitt

Fläche einer Figur

Objekte der Basisklasse `figur` haben keine Fläche. Dennoch wird für den gemeinsamen Zugriff eine Methode mit dem Namen `flaeche()` benötigt. Sie muss zum Erstellen einer korrekten Definition einen `double`-Wert zurückliefern wie die gleichnamigen bereits vorhandenen Methoden der beiden abgeleiteten Klassen `rechteck` und `kreis`.

Virtuelle Methode

Die Methode `flaeche()` dient in der Basisklasse allerdings nur als Umleitung, damit die Methode der Klasse des Objekts gefunden wird, auf die der Zeiger zeigt. Das wird erreicht, indem sie mithilfe des Schlüsselworts `virtual` als *virtuelle Methode* gekennzeichnet wird. Aus demselben Grund wird die bereits vorhandene Methode `ausgeben()` als virtuelle Methode gekennzeichnet.

Manche Compiler geben eine Warnung aus, dass die Klasse eine virtuelle Methode `ausgeben()` besitzt, aber keinen virtuellen Destruktor. Das hat keine Auswirkung auf den Ablauf.

Die beiden abgeleiteten Klassen `rechteck` und `kreis` bleiben mit Ausnahme eines Zeilenumbruchs unverändert.

Es folgt das Hauptprogramm:

Hauptprogramm

```
int main()
{
   figur fa(2.9, 1.2, "Schwarz");
   rechteck ra(7.4, 2.3, "Blau", 4.0, 2.6);
   kreis ka(4.6, 1.7, "Rot", 2.3);
   kreis kb(3.8, 0.5, "Gelb", 3.5);

   vector<figur *> figurZeigerFeld;
   figurZeigerFeld.push_back(&fa);
   figurZeigerFeld.push_back(&ra);
   figurZeigerFeld.push_back(&ka);
   figurZeigerFeld.push_back(&kb);

   double summeFlaeche = 0;
   for(figur *fz:figurZeigerFeld)
   {
      (*fz).verschieben(1.0, 1.0);
      (*fz).ausgeben();
      summeFlaeche += (*fz).flaeche();
   }
   cout << "Summe: " << summeFlaeche << endl;
}
```

Listing 11.13 Datei »polymorphie.cpp«, zweiter Ausschnitt

Vier Objekte

Zunächst werden die vier Objekte `fa`, `ra`, `ka` und `kb` der Klassen `figur`, `rechteck` beziehungsweise `kreis` erzeugt.

Zeiger auf Objekte

Anschließend wird das Feld `figurZeigerFeld` erstellt. Es handelt sich um ein `vector`-Feld von einfachen Zeigern auf Objekte der gemeinsamen Basisklasse `figur`. Mithilfe der Methode `push_back()` erhält das Feld einzelne Elemente. Jedes der Elemente beinhaltet die Adresse eines der zuvor erzeugten Objekte. Drei der vier Zeiger vom Typ *Einfacher Zeiger auf Objekt der Basisklasse* zeigen auf Objekte von abgeleiteten Klassen.

Methode suchen

Alle Methoden, die innerhalb der bereichsbasierten `for`-Schleife für diese Zeiger aufgerufen werden, werden zunächst in der Basisklasse gesucht. Entweder sind sie dort vorhanden, wie die Methode `verschieben()`, oder es existiert eine Umleitung zu der jeweils gleichnamigen Methode der tatsächlichen Klasse des Objekts, wie bei den Methoden `flaeche()` und `ausgeben()`.

Dieser theoretische Zusammenhang wird an zwei Beispielen erläutert:

Objekt der abgeleiteten Klasse

- Der zweite einfache Zeiger zeigt auf das Objekt `ra` der abgeleiteten Klasse `rechteck`. Die Methode `verschieben()` wird unmittelbar in der Klasse `figur` gefunden und genutzt. Die beiden Methoden `ausgeben()` und `flaeche()` sind innerhalb der Basisklasse virtuell. Daher wird zu den gleichnamigen Methoden der tatsächlichen Klasse des Objekts umgeleitet, und das ist die Klasse `rechteck`.

Objekt der Basisklasse

- Der erste einfache Zeiger zeigt auf das Objekt `fa` der Basisklasse `figur`. Auch hier wird für die beiden Methoden `ausgeben()` und `flaeche()` zur tatsächlichen Klasse des Objekts umgeleitet, und das ist wiederum die Klasse `figur`.

Es folgt die Ausgabe des Programms:

```
Figur: 3.9 / 2.2 Schwarz
Figur: 8.4 / 3.3 Blau
   Rechteck: 4 / 2.6
Figur: 5.6 / 2.7 Rot
   Kreis: 2.3
Figur: 4.8 / 1.5 Gelb
   Kreis: 3.5
Summe: 65.5035
```

11.5 Erben von mehreren Klassen

In C++ können Klassen ihre Eigenschaften und Methoden auch gleichzeitig von mehreren Klassen erben.

Sowohl ... als auch

So kann es zum Beispiel die Klassen `flugzeug` und `schiff` geben, die jeweils Elemente für ihre fliegenden beziehungsweise schwimmenden Objekte bieten. Die Klasse `wasserflugzeug` erbt von beiden Klassen, da ein Wasserflugzeug sowohl fliegen als auch schwimmen kann.

Mehrfachvererbung

Bei solchen *Mehrfachvererbungen* muss man allerdings darauf achten, dass manche Begriffe auf unterschiedliche Art zu deuten sind. Sollte es in beiden Klassen `flugzeug` und `schiff` die Eigenschaft `geschwindigkeit` geben, so kann es sich dabei für ein Objekt der Klasse `wasserfahrzeug` um die Geschwindigkeit des fliegenden Objekts in der Luft oder um die Geschwindigkeit des schwimmenden Objekts auf dem Wasser handeln.

Rechteck mit Textzeile

Im nachfolgenden Beispiel gibt es neben den beiden bekannten Klassen `figur` und `rechteck` die Klasse `textzeile`. Eine Textzeile, also ein Objekt dieser Klasse, hat einen textlichen Inhalt, der in einer bestimmten Schriftart, Schriftfarbe und Schriftgröße erscheint. Es kommt eine weitere Klasse `rechteckTextzeile` hinzu. Eine Rechteck-Textzeile, also ein Objekt dieser Klasse, ist ein Rechteck mit einer Textzeile.

Klasse »textzeile«

Zunächst die Definition der Klasse `textzeile` und ihrer Methoden:


```
class textzeile
{
   private:
      string inhalt;
      string art;
      string farbe;
      double groesse;
   public:
      textzeile(const string&, const string&,
         const string &, const double&);
      void ausgeben();
};

textzeile::textzeile(const string &i, const string &a,
   const string &f, const double &g)
{
   inhalt = i;
   art = a;
   farbe = f;
   groesse = g;
}

void textzeile::ausgeben()
{
```

```
   cout << "Textzeile: " << inhalt << " " << art
      << " " << farbe << " " << groesse << endl;
}
```

Listing 11.14 Datei »mehrfachvererbung.cpp«, erster Ausschnitt

Die Klasse umfasst vier Eigenschaften, einen Konstruktor mit vier Parametern und eine Methode zur Ausgabe.

Klasse »rechteck-Textzeile«

Es folgt die Definition der Klasse `rechteckTextzeile` und ihrer Methoden. Sie erbt von zwei Klassen. Besonders beim Aufbau des Konstruktors muss man den Überblick bewahren:

```
class rechteckTextzeile:public rechteck, public textzeile
{
   private:
      string ausrichtung;
   public:
      rechteckTextzeile(
         const double&, const double&, const string&,
            const double&, const double&,
         const string&, const string&, const string&, const double&,
         const string&);
      void ausgeben();
};

rechteckTextzeile::rechteckTextzeile(
      const double &xPos, const double &yPos, const string &fr,
         const double &h, const double &b,
      const string &tx, const string &art,
         const string &ft, const double &groesse,
      const string &aus):
   rechteck(xPos, yPos, fr, h, b),
   textzeile(tx, art, ft, groesse)
{
   ausrichtung = aus;
}

void rechteckTextzeile::ausgeben()
{
   rechteck::ausgeben();
```



```
   textzeile::ausgeben();
   cout << "Rechteck-Textzeile: " << ausrichtung << endl;
}
```

Listing 11.15 Datei »mehrfachvererbung.cpp«, zweiter Ausschnitt

Erbt von zwei Klassen

Nach dem Namen der Klasse folgen ein Doppelpunkt und anschließend die beiden Klassen, von denen diese Klasse erbt. Sie werden durch ein Komma voneinander getrennt.

Zehn Eigenschaften

Die Klasse umfasst insgesamt zehn Eigenschaften. Fünf Eigenschaften erbt sie von der Klasse `rechteck`, vier Eigenschaften stammen von der Klasse `textzeile`, und die eigene Eigenschaft `ausrichtung` kommt hinzu. Mit der letztgenannten Eigenschaft wird die Ausrichtung der Textzeile innerhalb des Rechtecks bezeichnet.

Weitergabe der Parameter

Der Konstruktor erwartet alle zehn Parameter. Er gibt fünf Parameter an die Klasse `rechteck` weiter. Nach einem Komma folgt die Weitergabe von vier Parametern an die Klasse `textzeile`. Beachten Sie die unterschiedlichen Namen der Parameter `fr` und `ft` für die beiden Farben. Innerhalb der Konstruktormethode wird nur noch der Wert für die eigene Eigenschaft `ausrichtung` zugewiesen.

Die Ausgabemethode ruft zunächst nacheinander die beiden gleichnamigen Methoden der Klassen auf, von denen geerbt wird.

Hauptprogramm

Es folgt das Hauptprogramm:

```
int main()
{
   textzeile ta("Guten Morgen", "Tahoma", "Schwarz", 12);
   ta.ausgeben();
   cout << endl;

   rechteckTextzeile rta(
      5.2, 1.8, "Blau", 1.2, 3.5,
      "Hallo Welt", "Arial", "Rot", 8.5,
      "Zentriert");
   rta.ausgeben();
}
```

Listing 11.16 Datei »mehrfachvererbung.cpp«, dritter Ausschnitt

Es wird zunächst ein Objekt der Klasse `textzeile` erstellt. Es folgt die Erzeugung eines Objekts der Klasse `rechteckTextzeile`. Zur besseren Zuordnung habe ich die Parameter des Konstruktors auf mehrere Zeilen verteilt. Die Ausgabe sieht wie folgt aus:

```
Textzeile: Guten Morgen Tahoma Schwarz 12

Figur: 5.2 / 1.8 Blau Rechteck: 1.2 / 3.5
Textzeile: Hallo Welt Arial Rot 8.5
Rechteck-Textzeile: Zentriert
```

Hinweis

Das Problem der Mehrdeutigkeit wurde bereits erwähnt. Wie ist die Zuordnung zu sehen? Ist eine Rechteck-Textzeile ein Rechteck und eine Textzeile, oder handelt es sich um ein Rechteck, das eine Textzeile beinhaltet? Der Aufbau der objektorientierten Struktur könnte auf unterschiedliche Art vorgenommen werden.