Kapitel 1
Einstieg in die Programmierung

Inhalt

- Wie entsteht ein Programm?
- Algorithmus ist eine Ablaufbeschreibung
- Die Eigenschaften von C++
- Programmgerüst, Variablen, Konstanten und Typen
- Zuweisungen, Ein- und Ausgabe

Sie wollen programmieren lernen. Wahrscheinlich haben Sie schon eine klare Vorstellung davon, was ein Programm ist. Ihre Textverarbeitung beispielsweise ist ein Programm. Sie können einen Text eintippen. Das Programm richtet ihn nach Ihren Wünschen aus, und anschließend können Sie das Ergebnis auf Ihrer Festplatte speichern oder auf Papier ausdrucken. Damit ist der typische Ablauf eines Programms bereits umschrieben.

1.1 Programmieren

Ein Programm nimmt Eingaben des Anwenders entgegen, verarbeitet sie nach Verfahren, die vom Programmierer vorgegeben wurden, und gibt die Ergebnisse aus. Damit wird deutlich, dass sich der Anwender in den Grenzen bewegt, die der Programmierer vorgibt. Wenn Sie selbst programmieren lernen, übernehmen Sie die Kontrolle über Ihren Computer. Während Sie bisher nur das mit dem Computer

machen konnten, was die gekaufte Software zuließ, können Sie als Programmierer frei gestalten, was der Computer tun soll.

1.1.1 Start eines Programms

Ein Programm ist zunächst eine Datei, die Befehle enthält, die der Computer ausführen kann. Diese Programmdatei befindet sich typischerweise auf der Festplatte oder auf einem sonstigen Datenträger. Wenn ein Programm gestartet wird, wird es zunächst von der Festplatte in den Hauptspeicher geladen. Nur dort kann es gestartet werden. Ein gestartetes Programm nennt man Prozess. Es wird vom Prozessor, der auch CPU (Central Processing Unit) genannt wird, abgearbeitet. Der Prozessor besitzt einen Programmzeiger, der auf die Stelle zeigt, die als Nächstes bearbeitet wird. Beim Starten des Prozesses wird dieser Zeiger auf den ersten Befehl des Programms gesetzt. Befehle weisen den Prozessor beispielsweise an, Werte aus dem Speicher zu lesen, zu schreiben oder zu berechnen. Der Prozessor kann Werte in Speicherstellen vergleichen und in Abhängigkeit davon mit der Abarbeitung an einer anderen Stelle fortfahren.

Maschinensprache

Die Befehle, die vom Prozessor derart interpretiert werden, sind der Befehlssatz der Maschinensprache dieses Prozessors. Der Hauptspeicher enthält also sowohl die Daten als auch die Programme. Der Hauptspeicher ist der Speicher, der unter Gedächtnisschwund leidet, wenn es ihm an Strom fehlt. Er kann eigentlich nur ganze Zahlen aufnehmen. Entsprechend bestehen die Befehle, die der Prozessor direkt interpretieren kann, nur aus Zahlen. Während Prozessoren solche Zahlenkolonnen lieben, ist es nicht jedermanns Sache, Programme als Zahlenfolgen zu schreiben. Die Programme laufen nur auf dem Prozessor, für den sie geschrieben sind, da die verschiedenen Prozessoren unterschiedliche Befehlssätze haben.

Assembler

Wenn tatsächlich Software in Maschinensprache entwickelt werden soll, dann verwendet man als Hilfsmittel eine Sprache namens Assembler. Diese Sprache lässt sich 1:1 in Maschinensprache übersetzen, ist aber für den Menschen leichter lesbar. So entspricht der Sprungbefehl einer 6502-CPU der Zahl 76. Der passende Assembler-Befehl lautet `JMP`. `JMP` steht für »jump«, zu Deutsch »springe«. Ein Über-

setzungsprogramm, das ebenfalls Assembler genannt wird, erzeugt aus den für Menschen lesbaren Befehlen Maschinensprache.

Heutzutage werden Maschinensprache und Assembler nur noch sehr selten eingesetzt, da die Entwicklungszeit hoch ist und die Programme nur auf dem Computertyp laufen, für den sie geschrieben sind. Der Vorteil von Assembler-Programmen ist, dass sie extrem schnell laufen. Da die Computer aber sowieso immer schneller werden, spart man sich lieber die hohen Entwicklungskosten. So wird Assembler heutzutage fast nur noch eingesetzt, wenn es darum geht, Betriebssystembestandteile zu schreiben, die durch Hochsprachen nicht abzudecken sind.

1.1.2 Eintippen, übersetzen, ausführen

Wenn ein Programmierer ein Programm schreibt, erstellt er eine Textdatei, in der sich Befehle befinden, die sich an die Regeln der von ihm verwendeten Programmiersprache halten. Die heutigen Programmiersprachen sind in erster Linie aus englischen Wörtern und einigen Sonderzeichen aufgebaut. Der Programmtext wird mithilfe eines Editors eingetippt. Ein Editor ist eine Textverarbeitung, die nicht mit dem Ziel entwickelt wurde, Text zu gestalten, sondern um effizient Programme schreiben zu können. Der Programmtext, den man als Source oder Quelltext bezeichnet, darf nur reinen Text enthalten, damit er einwandfrei weiterverarbeitet werden kann. Sämtliche Formatierungsinformationen stören dabei nur. Darum können Sie auch kein Word-Dokument als Quelltext verwenden. Bei den integrierten Entwicklungsumgebungen ist ein solch spezialisierter Editor bereits enthalten. Die Mindestanforderung an einen Programm-Editor ist, dass er Zeilennummern anzeigt, da die Compiler die Position des Fehlers in Form von Zeilennummern angeben.

Die von den Programmierern erstellten Quelltexte werden vom Computer nicht direkt verstanden. Wie schon erwähnt, versteht er nur die Maschinensprache. Aus diesem Grund müssen die Quelltexte übersetzt werden. Dazu wird ein Übersetzungsprogramm gestartet, das man Compiler nennt. Der Compiler erzeugt aus den Befehlen im Quelltext die Maschinensprache des Prozessors. Aus jeder Quelltextdatei erzeugt er eine sogenannte Objektdatei.

Im Falle von C und C++ besitzt der Compiler noch eine Vorstufe, die Präprozessor genannt wird. Der Präprozessor bereitet den Quelltext auf, bevor der eigentliche Compiler ihn in Maschinensprache übersetzt. Er kann textuelle Ersetzungen durchführen, Dateien einbinden und nach bestimmten Bedingungen Quelltextpassagen

von der Übersetzung ausschließen. Sie erkennen Befehle, die an den Präprozessor gerichtet sind, an einem vorangestellten #.

In der Praxis besteht ein Programmierprojekt normalerweise aus mehreren Quelltextdateien. Diese werden durch den Compiler in Objektdateien übersetzt. Anschließend werden sie vom Linker zusammengebunden. Hinzu kommt, dass ein Programm nicht nur aus dem Code besteht, den der Programmierer selbst schreibt, sondern auch Standardroutinen wie Bildschirmausgaben enthält, die immer wieder gebraucht werden und nicht immer neu geschrieben werden müssen. Diese Teile liegen dem Compiler-Paket als vorübersetzte Objektdateien bei und sind zu Bibliotheken zusammengefasst. Eine solche Bibliothek wird auch Library genannt. Der Linker entnimmt den Bibliotheken die vom Programm benötigten Routinen und bindet sie mit den neuen Objektdateien zusammen. Das Ergebnis ist ein vom Betriebssystem ausführbares Programm.

Der typische Alltag eines Programmierers besteht darin, Programme einzutippen und dann den Compiler zu starten, der den Text in Prozessor-Code übersetzt. Nach dem Binden wird das Programm gestartet und getestet, ob es die Anforderungen erfüllt. Danach wendet sich der Programmierer wieder den Quelltexten zu, um die gefundenen Fehler zu korrigieren.

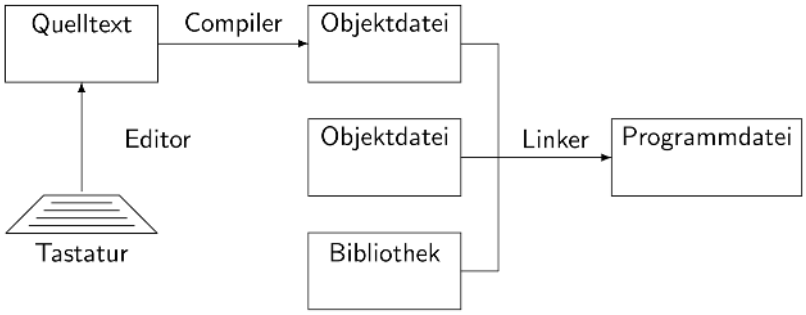

Abbildung 1.1 Entwicklungsweg eines Programms

Es sollte nicht unerwähnt bleiben, dass einige Programmierer dazu neigen, vor dem Eintippen zu denken. Ja, manche Entwickler machen richtige Entwürfe dessen, was sie programmieren wollen. Sie behaupten, dass sie dann letztlich schneller zum Ziel kommen. Und im Allgemeinen haben sie sogar Recht.

1.1.3 Der Algorithmus

Nachdem der Weg beschrieben ist, wie Sie vom Programmquelltext zu einem ausführbaren Programm kommen, sollte die Frage beantwortet werden, wie Programme inhaltlich erstellt werden.

Ein Programm ist eine sehr formale Beschreibung eines Ablaufs. Der Computer hat keine Fantasie und keine Erfahrungen. Darum kann er Ihren Anweisungen nur dann korrekt folgen, wenn Sie ihm zwingend vorschreiben, was er tun soll, und dabei keine Missverständnisse zulassen. Alle Voraussetzungen müssen ausformuliert werden und gehören zum Ablauf dazu. Eine solche Verfahrensbeschreibung nennt man einen Algorithmus. Algorithmen werden gern mit Kochrezepten verglichen. Die Analogie passt auch ganz gut, sofern es sich um ein Kochbuch für Informatiker handelt, das davon ausgeht, dass der Leser des Rezepts eventuell nicht weiß, dass man zum Kochen oft Wasser in den Topf füllt und dass zum Braten gern Öl oder anderes Fett verwendet wird.

Wer das erste Mal einen Algorithmus entwirft, stellt dabei fest, wie schwierig es ist, ein Verfahren so zu beschreiben, dass der Ausführende zwingend zu einem bestimmten Ergebnis kommt. Als kleine Übung sollten Sie jetzt kurz die Lektüre unterbrechen und beschreiben, wie Sie zählen. Der besseren Kontrolle halber sollen die Zahlen von 1 bis 10 auf einen Zettel geschrieben werden.

Fertig? Prima! Dann sollte auf Ihrem Zettel etwa Folgendes stehen:

1. Schreibe die Zahl 1 auf den Zettel.
2. Lies die Zahl, die zuunterst auf dem Zettel steht.
3. Wenn diese Zahl 10 ist, höre auf.
4. Zähle zu dieser Zahl 1 hinzu.
5. Schreibe diese Zahl auf den Zettel unter die letzte Zahl.
6. Fahre mit Schritt 2 fort.

Sie können sich darin üben, indem Sie alltägliche Abläufe formalisiert beschreiben. Notieren Sie beispielsweise, was Sie tun, um sich anzukleiden oder um mit dem Auto zur Arbeit zu fahren. Stellen Sie sich dann vor, jemand führt die Anweisungen so aus, dass er jeden Freiraum zur Interpretation nutzt, um den Algorithmus zum Scheitern zu bringen. Testen Sie es mit Verwandten oder wirklich guten Freunden aus: Die kennen in dieser Hinsicht erfahrungsgemäß die geringsten Hemmungen.

1.1.4 Die Sprache C++

Offensichtlich ist Ihre Entscheidung für C++ bereits gefallen. Sonst würden Sie dieses Buch ja nicht lesen. Je nachdem, ob Sie C++ aus eigenem Antrieb lernen oder aufgrund eines zarten Hinweises Ihres Arbeitgebers, werden Sie mehr oder weniger über C++ als Sprache wissen. Darum erlaube ich mir hier ein paar Bemerkungen.

Compiler-Sprache

C++ ist eine Compiler-Sprache. Das bedeutet, dass die vom Programmierer erstellten Programmtexte vor dem Start des Programms durch den Compiler in die Maschinensprache des Prozessors übersetzt werden, wie das in Abbildung 1.1 dargestellt ist.

Einige andere Programmiersprachen wie beispielsweise BASIC oder die Skriptsprachen werden während des Programmablaufs interpretiert. Das bedeutet, dass ein Interpreter den Programmtext lädt und Schritt für Schritt übersetzt und ausführt. Wird eine Zeile mehrfach ausgeführt, muss sie auch mehrfach übersetzt werden. Da die Übersetzung während der Laufzeit stattfindet, sind Programme, die in Interpreter-Sprachen geschrieben wurden, deutlich langsamer.

Daneben gibt es noch Mischformen. Der bekannteste Vertreter dürfte Java sein. Hier wird der Quelltext zwar auch vor dem Ablauf übersetzt. Allerdings ist das Ergebnis der Übersetzung nicht die Maschinensprache des Prozessors, sondern eine Zwischensprache, die sehr viel schneller interpretiert werden kann als die ursprünglichen Quelltexte.

Jede dieser Vorgehensweisen hat Vor- und Nachteile. Der Vorteil einer Compiler-Sprache liegt zum einen in der Ausführungsgeschwindigkeit der erzeugten Programme. Der Programmtext wird genau einmal übersetzt, ganz gleich, wie oft er durchlaufen wird, und die Übersetzung wird nicht zur Laufzeit des Programms durchgeführt. Ein weiterer Vorteil besteht zum anderen darin, dass viele Fehler bereits beim Kompilieren entdeckt werden. Interpreter haben den Vorteil, dass ein Programmierfehler nicht sofort das Programm abstürzen lässt. Der Interpreter kann stoppen, den Programmierer auf den Ernst der Lage hinweisen und um Vorschläge bitten, wie die Situation noch zu retten ist. Gerade Anfänger schätzen diese Vorgehensweise sehr, weil sie Schritt für Schritt ihr Programm ausführen lassen und sehen können, was der Computer tut. In Compiler-Sprachen können Sie das auch, benötigen dazu allerdings die Hilfe eines Debuggers (siehe Abschnitt 6.5). Die Sprachen, die mit virtuellen Maschinen arbeiten, bieten die Möglichkeit, ein

fertig übersetztes Programm auf verschiedensten Computern laufen zu lassen. Da die Programme vorübersetzt sind, sind sie deutlich schneller als reine Interpreter-Sprachen.

Wurzeln in C

Bjarne Stroustrup hat C++ aus der Programmiersprache C weiterentwickelt. C wurde in den 70er-Jahren von Kernighan und Ritchie im Zusammenhang mit der Entwicklung von UNIX entwickelt. Bis dahin waren Betriebssysteme in Assembler geschrieben worden. Dadurch waren die Betriebssysteme auf das Engste mit den Prozessoren und den anderen Hardware-Gegebenheiten ihrer Entwicklungsmaschinen verbunden. Für die Entwicklung von UNIX wurde eine Sprache geschaffen, die in erster Linie ein »portabler Assembler« sein sollte. Der erzeugte Code musste schnell und kompakt sein. Es war erforderlich, dass auch maschinennahe Aufgaben gelöst werden konnten. Auf diese Weise mussten nur kleine Teile von UNIX in Assembler realisiert werden, und das Betriebssystem konnte leicht auf andere Hardware-Architekturen portiert werden.

Bei dem Entwurf der Sprache C wurden die Aspekte der seinerzeit modernen Programmiersprachen berücksichtigt. So konnten in C die damals entwickelten Grundsätze der strukturierten und prozeduralen Programmierung umgesetzt werden. Die Sprache C entwickelte sich im Laufe der 80er- und 90er-Jahre zum Standard. Seit dieser Zeit wurden immer mehr Betriebssysteme in C geschrieben. Die meisten APIs (API ist die Abkürzung für *Application Programming Interface* und bezeichnet die Programmierschnittstelle vom Anwendungsprogramm zur Systemumgebung) sind in C formuliert und in dieser Sprache auf dem direktesten Wege zu erreichen. Aber auch in der Anwendungsprogrammierung wurde vermehrt C eingesetzt. Immerhin hatte C in den Betriebssystemen bewiesen, dass es sich als Entwicklungssprache für große Projekte eignete.

Strukturierte, modulare und objektorientierte Programmierung

Große Projekte erfordern, dass das Schreiben des Programms auf mehrere Teams verteilt wird. Damit ergeben sich automatisch Abgrenzungsschwierigkeiten. Projekte, an denen viele Programmierer arbeiten, müssen so aufgeteilt werden können, dass jede kleine Gruppe einen klaren eigenen Auftrag bekommt. Zum Schluss wäre es ideal, wenn man die fertigen Teillösungen einfach nur noch zusammenstecken müsste. Die Notwendigkeit von Absprachen sollte möglichst reduziert werden, da einer alten Regel zufolge die Dauer einer Konferenz mit dem Quadrat ihrer Teilnehmer ansteigt.

Um die Probleme großer Projekte in den Griff zu bekommen, haben die Entwickler von Programmiersprachen immer wieder neue Lösungsmöglichkeiten entworfen. Was sich bewährte, wurde in neuere Programmiersprachen übernommen und erweitert.

Strukturierte Programmierung

Die strukturierte Programmierung sagte dem wilden Springen zwischen den Programmteilen den Kampf an. Sie fasste Schleifen und Bedingungen zu Blöcken zusammen, sodass Sprungbefehle nicht nur überflüssig wurden, sondern sogar als Zeichen schlampiger Programmierung gebrandmarkt wurden. Abläufe, die immer wieder auftauchten, wurden zu Funktionen zusammengefasst. Klassische strukturierte Programmiersprachen sind PASCAL und C.

Modulare Programmierung

Die modulare Programmierung fasste Funktionen themenorientiert zu Modulen zusammen. Dabei besteht ein Modul aus einer Schnittstelle und einem Implementierungsblock. Die Schnittstelle beschreibt, welche Funktionen das Modul nach außen zur Verfügung stellt und wie die Programmierer anderer Module sie verwenden. Die Implementierungsblöcke sind so unabhängig vom Gesamtprojekt und können von verschiedenen Programmiererteams entwickelt werden, ohne dass sie sich gegenseitig stören. MODULA-2 ist eine typische Sprache für modulare Programmierung.

Objektorientierte Programmierung

Darauf aufbauend werden auch in der objektorientierten Programmierung Module gebildet. Dazu kommt allerdings der Grundgedanke, Datenobjekte in den Mittelpunkt des Denkens zu stellen und damit auch als Grundlage für die Zergliederung von Projekten zu verwenden. Lange Zeit waren Programmierer auf die Algorithmen fixiert, also auf die Verfahren, mit denen Probleme gelöst werden können. Mit der objektorientierten Programmierung ändert sich der Blickwinkel. Statt des Verfahrens werden die Daten als Dreh- und Angelpunkt eines Programms betrachtet. So werden in der objektorientierten Programmierung die Funktionalitäten an die Datenstrukturen gebunden. Durch das Konzept der Vererbung ist es möglich, dass ähnliche Objektklassen auf existierenden aufbauen, ihre Eigenschaften übernehmen und nur das implementiert wird, was neu ist.

Die Unterstützung der objektorientierten Programmierung durch die Einführung der Klassen ist der wichtigste Teilaspekt, der C++ von C abhebt. Gleichzeitig erlebte die objektorientierte Programmierung mit der Einführung von C++ ihre große Popularität. Heutzutage darf sich kaum noch eine Programmiersprache auf die Straße trauen, wenn sie nicht objektorientiert ist.

C++11

Auch C++ wurde inzwischen mehrfach standardisiert. Da jede programmierte Zeile investiertes Geld bedeutet, achtet man darauf, dass ältere Programme mit den aktuellen Compilern wenn möglich übersetzt werden können. Derzeit gilt die Überarbeitung von 2011, die als C++11 bezeichnet wird. Allerdings dauert es eine Weile, bis sich die Änderungen in den Compilern niederschlagen. Im Einzelfall müssen die Neuerungen am Compiler erst aktiviert werden.

C++ und die anderen Sprachen

Keine Frage, objektorientierte Programmierung ist in. C++ hat die objektorientierte Programmierung zwar nicht eingeführt, aber zumindest populär gemacht. Puristen der objektorientierten Programmierung werfen C++ vor, dass es eine Hybrid-Sprache ist. Das bedeutet, dass C++ nicht nur objektorientierte Programmierung zulässt, sondern es auch erlaubt, in klassischem C-Stil zu programmieren. Das ist aber durchaus so gewollt. C++ unterstützt die objektorientierte Programmierung, erzwingt sie aber nicht. Bjarne Stroustrup sah dies wohl eher als Vorteil. Er wollte eine Programmiersprache schaffen, die den Programmierer darin unterstützt, seiner Strategie zu folgen. Dieser sehr pragmatische Ansatz ist vielleicht auch ein Grund, warum C++ trotz Java und C# in der professionellen Entwicklung noch immer eine wesentliche Rolle spielt. Mit C++ ist es einem Anfänger möglich, kleine Programme zu schreiben, ohne zuerst die objektorientierte Programmierung verstehen zu müssen. Und gerade damit hat C++ den Erfolg der objektorientierten Programmierung vermutlich erst möglich gemacht. Denn so können Programmierer in kleinen Schritten die Vorteile der objektorientierten Programmierung kennenlernen und dann nutzen.

Effizienz

Ein anderer Vorteil von C++ ist seine Geschwindigkeit. Stroustrup legte großen Wert darauf, dass die Schnelligkeit und Kompaktheit von C erhalten bleibt. Damit bleibt C++ die Sprache erster Wahl, wenn es um zeitkritische Programme geht.

Portabilität

Natürlich können Sie in C++ portabel programmieren, aber es ist kein Dogma, wie etwa in Java. Java schränkt Sie in allen Bereichen ein, die nicht in der virtuellen Maschine implementiert sind. Da Ihr Programm aber vermutlich nicht unverändert auf einem Handy und einem Großrechner laufen soll, erlaubt C++ den direkten Zugriff auf das umgebende Betriebssystem und sogar auf die Hardware. Ordentliche Programmierer kapseln so etwas in eigenen Klassen, die bei der Portierung auf eine andere Plattform angepasst werden müssen.

Zuletzt hat die nach wie vor stabile Position von C++ auch etwas mit Marktstrategien zu tun. Java wurde entwickelt, um einmal geschriebenen Code auf allen Plattformen ohne Neukompilierung laufen lassen zu können. Dies konnte Microsoft nicht recht sein. So versucht der Konzern seit Jahren, diese Sprache zu unterlaufen. Der letzte Schlag ist die Entwicklung einer eigenen Konkurrenzsprache namens C#. So haben Projektleiter die Entscheidung, ob sie die Sprache Java verwenden, die auf allen Plattformen läuft, aber von Windows nur lieblos unterstützt wird. Oder sie verwenden die Sprache C#, die von Windows intensiv unterstützt wird, aber von allen anderen Plattformen nicht. Der lachende Dritte ist C++. Jedenfalls blieben die Forderungen nach C++-Kenntnissen bei Freiberuflern in den letzten Jahren stabil. In Bereichen, die mit wenig Ressourcen auskommen müssen wie Embedded Systems, und in Bereichen, in denen es auf extreme Performance ankommt wie bei Computerspielen, geht kaum ein Weg an C++ vorbei.

1.2 Compiler beschaffen und einrichten

Damit Sie die Beispiele aus dem Buch nachvollziehen können, benötigen Sie einen Compiler und am besten auch noch eine Entwicklungsumgebung (kurz IDE für *Integrated Development Environment*), die Sie bei der Entwicklung unterstützt. Eine sehr brauchbare Entwicklungsumgebung nennt sich Code::Blocks und ist sogar kostenlos erhältlich. Besonders angenehm ist, dass Code::Blocks nicht nur für Windows, sondern auch für Linux und Macintosh zur Verfügung steht.

Unter der Haube von Code::Blocks arbeitet der GNU-Compiler, der auf allen namhaften Plattformen erhältlich ist. Ohne die IDE kann er von der Kommandozeile aufgerufen werden. Eine nähere Beschreibung des Umgangs mit Entwicklungswerkzeugen finden Sie in Kapitel 6, in den folgenden Abschnitten steht aber schon mal das Wichtigste in Kürze, damit Sie Ihre ersten Programme schreiben und ausführen können.

1.2.1 Installation

Code::Blocks ist wie erwähnt eine kostenlose Open-Source-IDE. Sie finden es unter der URL *http://www.codeblocks.org/downloads*. Dort finden Sie einen Link, der mit *Download the binary release* bezeichnet ist. Er verweist auf eine Seite, auf der Sie für Ihr Betriebssystem die passende Version zum Download finden.

Linux-Benutzer können sich Code::Blocks auch über ihren Standard-Paket-Dienst installieren, den die Distribution mitbringt. Das vereinfacht die Installation erheblich und sorgt automatisch dafür, dass die Version zum Betriebssystem passt und Updates immer passend zum System erfolgen.

Windows-Benutzer können zwischen zwei unterschiedlichen Varianten wählen. Die eine enthält nur die IDE ohne einen Compiler. Das wird für Sie vermutlich nicht interessant sein. Darum wählen Sie die größere Datei, die »mingw« im Namen enthält.

Nach dem erfolgreichen Download starten Sie die Datei und Sie sehen sich einem normalen Installationsprogramm gegenüber, wie Sie es von vielen anderen Programmen her kennen.

Wenn Sie der GNU-Lizenz zustimmen und ansonsten NEXT anklicken, gelangen Sie irgendwann an den Dialog, den Sie mit INSTALL bestätigen müssen. Es entsteht ein Symbol auf Ihrem Desktop und ein Eintrag im Programm-Menü. Dann fragt Sie das Installationsprogramm, ob Code::Block gleich gestartet werden soll. Bei seinem ersten Start sucht es nach Compilern. Sie wählen aus der Liste den GNU-Compiler aus und werden dann noch gefragt, ob Code::Blocks automatisch die Verantwortung für alle CPP-Dateien übernehmen soll. Danach ist Ihre IDE fertig eingerichtet (siehe Abbildung 1.2).

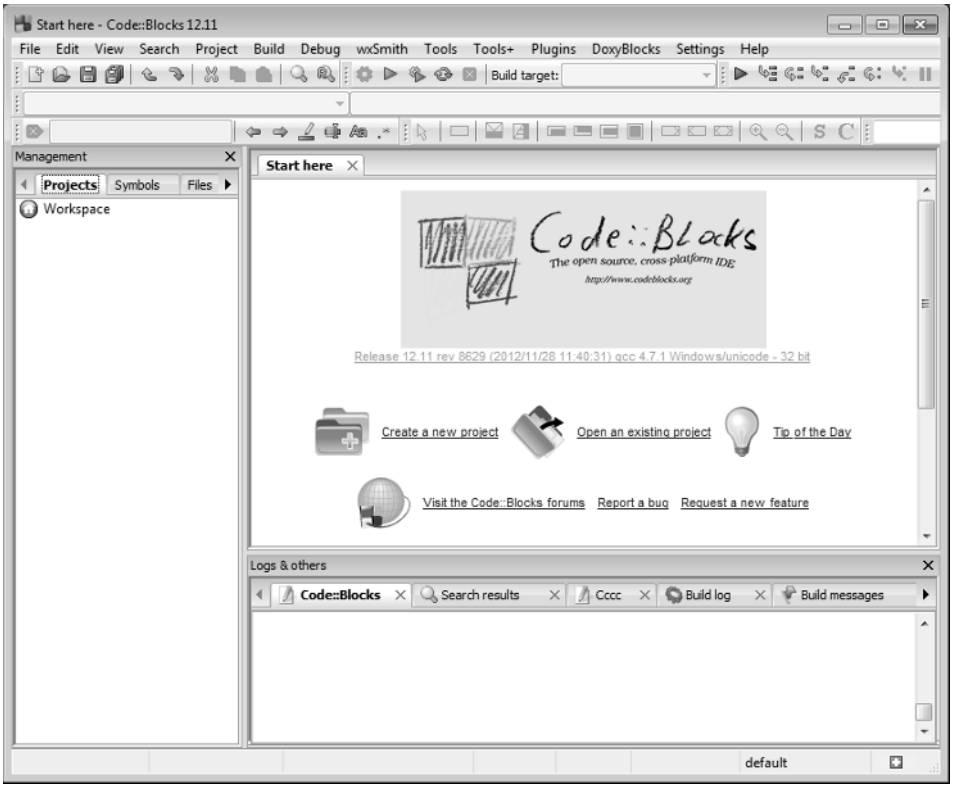

Abbildung 1.2 Das Programm Code::Blocks

1.2.2 IDE-Aufbau

Wie bei vielen IDEs findet sich im Zentrum ein Editor, in dem die Quelltexte erfasst werden können. Über die Tabs am oberen Rand kann zwischen den Dateien umgeschaltet werden, wenn Sie mit mehr als einer Quelltextdatei arbeiten.

Auf der linken Seite gibt es eine Projektliste. Jedes Projekt lässt sich aufklappen, um seine Elemente zu bearbeiten. Unten ist das Anzeigefeld. Hier erscheinen Feldermeldungen und Warnungen, aber auch die Ausgaben des fertigen Programms.

1.2.3 Ein Projekt anlegen

Bevor Sie den Quelltext Ihres Programms erfassen, müssen Sie ein Projekt anlegen. Ein Projekt enthält alle Informationen, die für das Erstellen eines Programms er-

forderlich sind. Über die Menüpunkte FILE | NEW | PROJECT... gelangen Sie zu einem Dialog, der Ihnen verschiedene Projektarten anbietet. Für die Beispiele im Buch wählen Sie eine Konsolenapplikation (Console application).

Nun gelangen Sie zu einer Folge von Dialogen, die genauere Einstellungen Ihres neuen Projektes abfragt. Sie werden gefragt, ob Sie in C oder C++ programmieren möchten. Hier wählen Sie natürlich C++. Im Folgedialog sollen Sie das Projekt benennen und Code::Blocks wird aus diesem Namen auch gleich die Pfade generieren, wo Ihre Dateien stehen werden.

Nun sollen Sie festlegen, welchen Compiler Sie verwenden wollen. Relevant ist das nur, wenn auf dem Computer mehrere Compiler installiert sind. Hier ist der GNU-Compiler eine gute Wahl. Sie sehen, dass die IDE sowohl eine Debug- als auch eine Release-Version erstellen will. Die Debug-Version ist hilfreich, wenn Sie in größeren Projekten nach Fehlern suchen. Eine Release-Version ist schlanker und für die Auslieferung an den Kunden gedacht.

Anschließend steht in der Projektliste links das neue Projekt unter dem von Ihnen gewählten Namen. Sie können das Projekt aufblättern. Sie finden dann einen Unterordner namens SOURCES und darunter ist bereits je nach Projektart und -optionen der Rahmen für ein kleines Hello-Programm unter dem Namen *main.cpp* erzeugt worden.

1.2.4 Übersetzen und starten

Damit das Programm in ausführbaren Code übersetzt wird, müssen Sie die Menüpunkte BUILD | BUILD oder alternativ Strg-F9 drücken. Unterhalb des Editierfensters sehen Sie die Compilermeldungen. Sollten sich Fehler eingeschlichen haben, können Sie auf den Fehler klicken und landen an der Stelle in Ihrem Quelltext, wo sich der Fehler befindet.

Sobald Ihr Programm fehlerfrei ist, können Sie es direkt aus der IDE heraus starten. Dazu wählen Sie die Menüpunkte BUILD | RUN. Da Sie eine Konsolenanwendung geschrieben haben, startet zunächst ein Konsolenfenster, das schwarz mit weißer Schrift ist. Darin sehen Sie die Ausgabe »Hello World«. Es folgt eine kurze Statistik, ob Ihr Programm fehlerfrei durchlaufen wurde und wie lange es lief. Wenn Sie die ↵-Taste drücken, wird das Fenster wieder geschlossen und Sie befinden sich wieder in der IDE.

1.3 Grundgerüst eines Programms

Ein Programm ist eine Aneinanderreihung von Befehlen, die dem Computer zur Abarbeitung zugeführt werden können. Wie schon erwähnt, wird ein C++-Programm in einem Texteditor geschrieben und dann von einem Compiler in ausführbaren Code übersetzt. Danach kann es gestartet werden. In diesem Fall ist mit dem Compiler übrigens der komplette Übersetzer gemeint, der aus Compiler und Linker besteht. Diese Ungenauigkeit ist im allgemeinen Sprachgebrauch durchaus üblich.

main()

Ein C++-Programm besteht mindestens aus der Funktion `main()`. Die Silbe »main« ist englisch und bedeutet so viel wie die deutsche Vorsilbe »Haupt«. Was in C++ genau unter einer Funktion zu verstehen ist, wird später noch sehr viel ausführlicher behandelt und soll Sie hier nicht verwirren. Wichtig ist, dass Sie in jedem C++-Programm irgendwo den Namen `main` mit einem Klammernpaar finden werden, dem die Silbe `int` voransteht. In manchen Fällen stehen zwischen den Klammern ein paar kryptisch anmutende Zeichen. Aber auch das sollte Sie zu Anfang nicht stören. Ignorieren Sie es von ganzem Herzen, bis Sie wissen, wozu Sie es brauchen.

Die Hauptfunktion beginnt direkt nach einer öffnenden geschweiften Klammer und endet mit der schließenden Klammer. Zwischen den geschweiften Klammern stehen die Befehle der Hauptfunktion. Sie werden nacheinander ausgeführt. Das folgende kurze Listing enthält nichts und ist damit das kleinste denkbare C++-Programm. Als Listing bezeichnet man den Quelltext eines Programms.

```
int main()
{
}
```

Listing 1.1 Ein Minimalprogramm, das nichts tut

Dieses Programm tut gar nichts. Wenn Sie es später um ein paar Tätigkeiten erweitern wollten, würden Sie die Befehle zwischen die geschweiften Klammern schreiben.

1.3.1 Kommentare

Wenn wir schon einmal ein Programm haben, das nichts tut, wollen wir es auch um Befehle erweitern, die nichts tun: Kommentare.

Sie können in das Programm Kommentare einfügen, die vom Compiler völlig ignoriert werden. Die Kommentare sollen dokumentieren, warum das Programm so und nicht anders geschrieben wurde, und richten sich an Programmierer. Dies können Kollegen sein, die Ihr Programm korrigieren oder erweitern sollen. Aber noch öfter helfen Sie sich selbst damit. In der Praxis ist es so, dass Sie vermutlich auch herangezogen werden, wenn eines Ihrer Programme nicht korrekt läuft oder ergänzt werden soll. Da Sie zwischendurch andere Programme geschrieben haben werden, vielleicht geheiratet haben, umgezogen sind und noch drei weitere Programmiersprachen gelernt haben, werden Sie sich nicht mehr an jedes Detail erinnern. Sie werden dankbar sein, wenn Sie in den Programmen ausführliche Hinweise finden, wie und warum es so funktioniert oder zumindest funktionieren soll.

Zeilenweise

Es gibt zwei Arten, einen Text davor zu schützen, dass der Compiler versucht, ihn als Programm zu interpretieren. Die einfachere Art der Kommentierung ist es, zwei Schrägstriche direkt hintereinanderzusetzen. Damit gilt der Rest der Zeile als Kommentar.

```
int main()
{
    // Hier beginnt der Kommentar.

    // Die nächste Zeile braucht
    // ihr eigenes Kommentarzeichen.
}
```

Listing 1.2 Zeilenweises Kommentieren

Kommentarblock

Daneben gibt es die Möglichkeit, einen größeren Text in Kommentarklammern einzuschließen und damit dem Compiler zu entziehen. Der Anfang des Kommentars wird durch den Schrägstrich, gefolgt von einem Stern, festgelegt. Der Kommentar endet mit der umgekehrten Zeichenfolge, also mit einem Stern und einem darauf folgenden Schrägstrich.

```
int main()
{
  /* Hier beginnt der Kommentar.
     Die nächste Zeile braucht kein
```

```
    eigenes Kommentarzeichen.
  */
}
```

Listing 1.3 Blockweises Kommentieren

Sie können diese Kommentarklammern nicht verschachteln. Mit dem ersten Auftreten der Kombination von Schrägstrich und Stern beginnt der Kommentar. Er endet mit dem ersten Auftreten der Kombination aus Stern und Schrägstrich. Dazwischen liegende Kommentaranfangzeichen gehören zum Kommentar. Der Compiler wird den folgenden Text als Programm ansehen und versuchen, ihn zu übersetzen.

```
int main()
{
  /* Hier beginnt der Kommentar
     /*
       Die nächste Zeile braucht kein
       eigenes Kommentarzeichen
     */
     Dies wird der Compiler wieder übersetzen wollen.
  */
}
```

Listing 1.4 Das geht schief!

Mit `/*` und `*/` können nicht nur große Blöcke, sondern auch kurze Teile innerhalb einer Zeile kommentiert werden. Diese Möglichkeit haben Sie mit dem doppelten Schrägstrich nicht, weil dieser Kommentar ja immer bis zum Ende der Zeile geht. Im folgenden Beispiel ist die schließende Klammer von `main()` außerhalb des Kommentars.

```
int main( /* hier könnte auch noch etwas stehen */ )
{
}
```

Inhalt der Kommentare

In vielen Programmierkursen und auch in einigen Firmen finden Sie umfangreiche Beschreibungen, die sich darüber auslassen, wo Kommentare erscheinen und was sie beinhalten sollen. Der Zweck dieser Vorschriften ist, dass die Investition in die Programmierung nicht verloren geht. Wenn ein unverständlicher Programmcode später kontrolliert wird, kann es sein, dass dann Fehler eingebaut werden, nur weil

der Programmierer nicht verstanden hat, was der Zweck des Programmteils war. Die einfachste Regel in der Kommentierung lautet darum:

Tipp

Goldene Regel der Kommentierung: Kommentieren Sie nicht, *was* Sie gemacht haben, sondern *warum* Sie es so gemacht haben!

Ein Kommentar »Zwei Werte werden addiert« ist nutzlos. Denn jeder Programmierer, der die Sprache C++ auch nur halbwegs kennt, kann das direkt im Quelltext ablesen. Dagegen ist die Aussage »Ermittle den Bruttobetrag aus Nettobetrag und MwSt« hilfreich, da dadurch jeder Leser weiß, warum diese Werte addiert werden.

1.3.2 Anweisungen

Ein Programm besteht nicht nur aus dem Rahmen und aus Kommentaren, sondern auch aus Anweisungen. Eine Anweisung kann dazu dienen, etwas zu lesen, zu speichern, zu berechnen, zu definieren oder auf dem Bildschirm auszugeben. Anweisungen sind die Grundeinheiten, aus denen Programme bestehen.

Anweisungen können in C++ beliebig formatiert werden. Es besteht keine Verpflichtung, sie in einer bestimmten Position anzuordnen. Auch können Anweisungen über beliebig viele Zeilen gehen. Allerdings besteht der Compiler extrem kleinlich darauf, dass Anweisungen immer mit einem Semikolon abgeschlossen werden.

1.3.3 Blöcke

Mehrere Anweisungen können zu einem Block zusammengefasst werden. Ein Block wird durch geschweifte Klammern eingefasst. Der Compiler wird einen Block wie eine einzige Anweisung behandeln.

Sicher ist Ihnen schon aufgefallen, dass die Hauptfunktion `main()` ebenfalls solche geschweiften Klammern hat. In der Tat ist dies der Block, in dem die Befehle des Programms zusammengefasst werden.

Um die Übersicht zu erhöhen, pflegt man alle Befehle innerhalb eines Blocks einzurücken. Achten Sie vor allem am Anfang darauf, dass die zusammengehörigen geschweiften Klammern auf einer Ebene stehen. Das folgende Beispiel mit Pseudobefehlen zeigt korrektes Einrücken.

```
int main()
{
    Hier sind Pseudo-Anweisungen;
    Diese gehört dazu;
    {
        Neuer Block, also einrücken;
        Wir bleiben auf diesem Niveau;
        Das kann ewig weitergehen;
    }
    Nun ist die Klammer geschlossen;
    Und die Einrückung ist auch zurück;
}
```

Listing 1.5 Eingerückte Blöcke

Wie weit Sie einrücken, ist Geschmackssache. Eine Einrückung von einem Zeichen ist kaum erkennbar. Zwei Leerschritte sind das absolute Minimum. Drei oder vier Schritte sind weit verbreitet. Auch wenn viele Editoren acht Schritte für den Tabulator verwenden, wird der Code damit oft unübersichtlich, weil er zu weit nach rechts herausragt.

1.4 Variablen

Der Abschnitt zum Thema Variablen enthält viele Details, die an dieser Stelle ausgeführt werden müssen. Wenn Sie als Anfänger diesen Abschnitt zum ersten Mal lesen, brauchen Sie sich nicht alle Informationen zu merken. Versuchen Sie, ein grundlegendes Verständnis für Variablen, Typen und Konstanten zu entwickeln. Sie sind jederzeit eingeladen, wieder hierher zurückzukommen, wenn Sie später einmal Wissenslücken auffüllen möchten.

In jedem Programm werden Informationen verarbeitet. Diese Informationen liegen im Hauptspeicher, der auch RAM (Random Access Memory) genannt wird. Höhere Programmiersprachen greifen nicht direkt auf den Speicher zu, sondern verwenden Variablen. Variablen sind also die Behälter, in denen das Programm Zahlen und Texte ablegt. Eine Variable hat drei wichtige Eigenschaften:

- **Speicher** – Die Variable benötigt zum Ablegen ihrer Informationen immer Speicher. Über Lage und Größe des Speichers braucht sich der Programmierer normalerweise nicht zu kümmern. Der Compiler ermittelt die benötigte Größe aus dem Typ der Variablen.

- **Name** – Die Variable wird im Programm über einen weitgehend frei wählbaren Namen angesprochen. Dieser Name identifiziert die Variable eindeutig. Verschiedene Namen bezeichnen verschiedene Variablen. Die Namensregeln finden Sie ausführlich ab Abschnitt 1.4.3.
- **Typ** – Der Typ einer Variablen bestimmt, welche Informationen abgelegt werden können. So kann eine Variable je nach ihrem Typ beispielsweise einen Buchstaben oder eine Zahl speichern. Der Typ bestimmt natürlich auch die Speichergröße, die benötigt wird. Der Typ bestimmt aber auch, welche Operationen auf die Variable angewendet werden können. Zwei Variablen, die Zahlen enthalten, können beispielsweise miteinander multipliziert werden. Enthalten die Variablen dagegen Texte, ist eine Multiplikation nicht besonders sinnvoll.

1.4.1 Variablendefinition

Das folgende Beispiel zeigt eine Variablendefinition innerhalb der Hauptfunktion `main()`:

```
int main()
{
    int einkommen;
}
```

Listing 1.6 Variablendefinition

Typ

Eine Variablendefinition beginnt immer mit dem Typ der Variablen. Hier heißt der Typ `int`. Dieser Typ steht für eine ganze Zahl mit Vorzeichen, aber ohne Nachkommastellen (Integer).

Durch ein Leerzeichen abgesetzt, beginnt der Name der Variablen. Den Namen sollten Sie immer so wählen, dass Sie auf den Inhalt schließen können. Hier lässt der Name `einkommen` bereits auf die Verwendung der Variablen im Programm schließen. Verwenden Sie ruhig lange Namen. Abkürzungen ersparen zwar Tipparbeit, Sie werden aber auf lange Sicht mehr Zeit verschwenden, wenn Sie darüber nachdenken müssen, was in welcher Variablen abgelegt ist. Das gilt auch dann, wenn Sie extrem langsam tippen. Noch mehr Zeit werden Sie brauchen, wenn Sie einen Fehler suchen müssen, der entstand, weil Sie zwei Variablen verwechselt haben, nur weil der Name unklar war. Zu guter Letzt folgt das Semikolon. Damit wird jede Anweisung abgeschlossen, auch eine Variablendefinition.

Initialisierung

Sie können Variablen gleich bei ihrer Definition mit einem Wert vorbelegen. Dazu setzen Sie hinter den Variablennamen ein Gleichheitszeichen. Das initialisiert die Variable mit dem nachfolgenden Wert.

```
int einkommen=0;
```

Hier wird die Variable `einkommen` gleich bei der Definition auf 0 gesetzt. Die Initialisierung von Variablen muss nicht zwingend durchgeführt werden. Allerdings ergeben sich viele Programmfehler daraus, dass Variablen nicht korrekt initialisiert waren. Es ist keineswegs gesichert, dass eine neu angelegte Variable den Wert 0 hat, solange Sie das nicht explizit festlegen. Anstatt eine Initialisierung mit dem Gleichheitszeichen durchzuführen, kann in C++ auch eine Klammer verwendet werden. Das obige Beispiel sähe in dieser Syntax so aus:

```
int einkommen(0);
```

Initialisierung seit C++11

Seit C++11 können Sie statt der runden Klammern auch geschweifte Klammern verwenden und dann auch noch das Gleichheitszeichen hinzufügen. Diese Vielfalt mag jetzt irritierend sein, hat aber seinen Grund darin, dass auch komplexere Datenstrukturen über geschweifte Klammern initialisiert werden können.

```
int einkommen{0};
int ausgaben={0};
```

Mehrfache Definition

Es können mehrere Variablen gleichen Typs direkt hintereinander definiert werden, indem sie durch Kommata getrennt werden.

```
int i, j=0, k;
```

Hier werden die Variablen i, j und k definiert. j wird mit 0 initialisiert. Diese Schreibweise ist mit der folgenden gleichwertig:

```
int i;
int j=0;
int k;
```

Den Syntaxgraph zur Variablendefinition finden Sie in Abbildung 1.5.

1.4.2 Geltungsbereich

Sie können in C++ eine Variable erst verwenden, nachdem Sie sie definiert haben. (Um genau zu sein, reicht es, sie deklariert zu haben. Das bedeutet, dass Sie dem Compiler mitteilen, dass er diese Variable verwenden kann und welchen Typ sie hat. Sie kann dann an anderer Stelle definiert werden. Siehe Abschnitt 4.7.1) Der Compiler prüft, ob die Variable ihrem Typ gemäß verwendet wird. Es gibt Programmiersprachen, die eine Variable automatisch anlegen, sobald sie das erste Mal verwendet wird. Dieser Komfort wird tückisch, wenn Sie eine Variable namens `abteilungsNr` versehentlich an anderer Stelle ohne s, also als `abteilungNr` bezeichnen. Sie werden vom Compiler nicht auf Ihren Tippfehler hingewiesen. Stattdessen arbeiten Sie mit zwei unterschiedlichen Variablen, ohne es zu ahnen. Das kann Ihnen in C++ nicht passieren. Der Compiler wird sofort nörgeln, dass er die Variable `abteilungNr` überhaupt nicht kennt, und er wird sich weigern, mit einer solchen Variablen irgendetwas zu tun. Sie werden also sofort auf Ihren Fehler hingewiesen und können ihn korrigieren.

Blockgrenzen

Während in C Variablendefinitionen nur am Blockanfang vor der ersten Anweisung erlaubt sind, kann in C++ eine Variable überall definiert werden. Sie gilt dann für den gesamten restlichen Bereich des aktuellen Blocks und aller darin verschachtelten Blöcke.

Es können sogar in verschachtelten Blöcken Variablen mit dem gleichen Namen verwendet werden. Dabei überdeckt die innen liegende Definition diejenige, die außerhalb des Blocks liegt. Das folgende Beispiel macht das deutlich:

```
{
    int a = 5;
    {
        // Hier hat a den Inhalt 5.
        int a = 3;
```

```
        // Hier hat a den Inhalt 3.
        {
            // a ist immer noch 3.
        }
    }
    // Hier ist a wieder 5.
}
```

Listing 1.7 Zwei Variablen

Lokale Variable

Jede Variable, die innerhalb eines Blocks definiert wird, wird als lokale Variable bezeichnet. Sie ist lokal für den Bereich des Blocks definiert. Für ihren Geltungsbereich überdeckt die innen definierte Variable `a` die außen liegende Variable `a`. Die außen liegende Variable existiert durchaus noch, aber aufgrund der Namensgleichheit mit der lokalen Variablen kann man bis zu deren Auflösung nicht auf sie zugreifen. Sobald das Programm den Block verlässt, in dem die lokale Variable definiert ist, wird die äußere Variable wieder sichtbar. Alle Operationen auf der lokalen Variablen berühren die äußere Variable nicht.

Variablen, die außerhalb jedes Blocks definiert werden, nennt man globale Variablen. Sie gelten für alle Blöcke, die nach der Definition im Quelltext auftreten, sofern darin nicht eine Variable gleichen Namens lokal definiert wird.

```
int einkommen=0; // globale Variable
int main()
{
    int ausgaben=0; // lokale Variable
}
```

Listing 1.8 Variablendefinition

Globale Variablen werden bei C++ grundsätzlich vom Compiler mit 0 initialisiert, sofern das Programm nicht eine eigene Initialisierung vornimmt. Dagegen ist der Inhalt lokaler Variablen bei ihrer Definition unbestimmt und meist nicht 0.

Das Thema der globalen und lokalen Variablen spielt im Bereich der Funktionen (siehe Kapitel 4) eine besondere Rolle. Daher wird dieses Thema dort noch einmal aufgenommen.

1.4.3 Namensregeln und Syntaxgraph

In C++ unterliegen alle Namen, die vom Programmierer gewählt werden können, den gleichen Regeln. Diese Regeln gelten nicht nur für Variablen, sondern auch für Funktionen oder Klassen. In der englischen Literatur werden diese Namen als »identifier« bezeichnet. Das wird meist mit »Bezeichner« übersetzt. Ein Name muss mit einem Buchstaben oder einem Unterstrich beginnen und darf anschließend beliebig viele Buchstaben, Unterstriche und Ziffern enthalten. Das erste Zeichen darf keine Ziffer sein, damit der Compiler einen Namen leichter von einer Zahl unterscheiden kann. Die Buchstaben können klein oder groß sein. In C und C++ wird zwischen Groß- und Kleinschreibung unterschieden. Die Variable `Anton` ist eine andere Variable als die Variable `ANTON` oder `anton`.

In Abbildung 1.3 wird die Regel zur Bildung eines Namens grafisch dargestellt. Zur Bildung eines Namens dürfen Sie dem Graphen in Pfeilrichtung immer entlang der Kurven folgen. Wenn Sie den Graphen auf der rechten Seite verlassen haben, haben Sie einen zulässigen Namen gebildet. Sie werden feststellen, dass alle Namen mindestens ein Zeichen haben müssen, das entweder ein Buchstabe oder ein Unterstrich ist, und dass dann beliebig viele Zeichen in beliebiger Reihenfolge verwendet werden dürfen. Dann dürfen zu den Buchstaben und dem Unterstrich auch Ziffern hinzukommen.

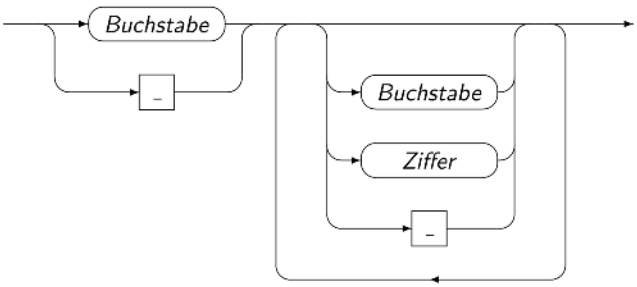

Abbildung 1.3 Syntaxgraph eines Namens

In rechteckigen Feldern finden Sie in Fettschrift sogenannte *Terminale*. Das sind Zeichen oder Zeichenketten, die nicht weiter aufgelöst werden, sondern so eingesetzt werden, wie sie sind. Im obigen Beispiel ist das der Unterstrich. In ovalen Feldern finden Sie in Kursivschrift Nonterminale, also Elemente, die einer näheren Beschreibung bedürfen. Diese folgt dann entweder im Text oder in weiteren Syntaxgraphen. Hier sind das die Buchstaben und Ziffern. Der entsprechende Graph ist in Abbildung 1.4 zu sehen.

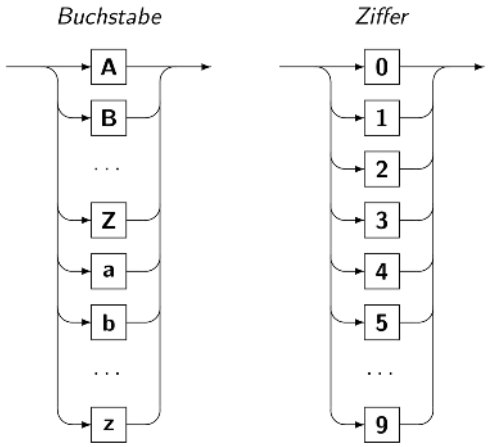

Abbildung 1.4 Syntaxgraph für *Buchstabe* und *Ziffer*

Wie Sie sehen, gehen die Buchstaben von A bis Z und von a bis z. Grundsätzlich ist die Verwendung nationaler Sonderzeichen in Namen nicht erlaubt. Umlaute oder ein ß im Variablennamen führen also immer zu einer Fehlermeldung.

Zu guter Letzt dürfen keine Befehle der Sprache C++ als Namen verwendet werden. Auch wenn Sie noch nicht alle Befehle von C++ kennen, können Sie dies ganz einfach umgehen, indem Sie Variablennamen wählen, die mindestens einen Großbuchstaben enthalten. Die Schlüsselwörter in C++ bestehen nur aus Kleinbuchstaben.

Schlüsselwörter

Die folgende Liste enthält die unter C++ verwendeten Schlüsselwörter. Sie dürfen nicht als Namen verwendet werden.

```
alignas         alignof         and             and_eq
asm             auto            bitand          bitor
bool            break           case            catch
char            char16_t        char32_t        class
compl           const           constexpr       const_cast
continue        decltype        default         delete
do              double          dynamic_cast    else
enum            explicit        export          extern
false           float           for             friend
goto            if              inline          int
```

```
long            mutable         namespace       new
noexcept        not             not_eq          nullptr
operator        or              or_eq           private
protected       public          register        reinterpret_cast
return          short           signed          sizeof
static          static_assert   static_cast     struct
switch          template        this            thread_local
throw           true            try             typedef
typeid          typename        union           unsigned
using           virtual         void            volatile
wchar_t         while           xor             xor_eq
```

Listing 1.9 Schlüsselwörter von C++

1.4.4 Typen

Variablen haben immer einen Typ. Der Typ besagt, welche Informationen eine Variable aufnehmen kann. Der Typ bestimmt, wie viel Speicher für eine Variable benötigt wird. Ohne Typ können Sie keine Variable definieren. Der Compiler will zum Zeitpunkt der Übersetzung wissen, wie groß der erforderliche Speicher ist. Er überprüft auch, ob der Variablentyp im Kontext überhaupt passt. Diese Nörgeleien des Compilers sind nur zum Besten des Programmierers. Der Compiler erkennt so Flüchtigkeitsfehler, bevor sie Schaden anrichten können. In gewissen Grenzen ist C++ allerdings großzügig. Ähnliche Typen werden direkt ineinander überführt, sofern kein Informationsverlust auftreten kann.

Informationsspeicherung

Die Informationen, die in den Variablen stehen, werden im Hauptspeicher des Computers abgelegt. Wie funktioniert der Hauptspeicher eines Computers? Ein Computer arbeitet digital. Er verarbeitet also zwei Zustände: An und Aus. Der kleinste Speicher im Computer kann sich genau diese Information merken. Man kann »An« als 1 und »Aus« als 0 interpretieren. Die Informationseinheit, die eine Information 1 und 0 unterscheiden kann, nennt man Bit.

Aus praktischen Gründen stellt man mehrere Bits zusammen. Dadurch ist es möglich, Zahlen darzustellen, die größer als 1 sind. Das Verfahren ist analog zu dem, das in unserem Zehnerzahlensystem angewandt wird, um Zahlen darzustellen, die größer als 9 sind: Man verwendet eine weitere Stelle und multipliziert diese mit der Anzahl der Ziffern, die zur Verfügung stehen. Im Zehnersystem sind das zehn,

im Binärsystem sind das zwei. Mit einem Bit lassen sich 0 und 1 darstellen. Mit zwei Bits gibt es die Kombinationen 00, 01, 10 und 11. Damit sind also mit zwei Bits die Zahlen 0, 1, 2 und 3 darstellbar. Ein weiteres Bit verdoppelt wiederum die Anzahl der Kombinationen. Die möglichen Kombinationen sind 000, 001, 010, 011, 100, 101, 110 und 111 und entsprechen den Zahlen von 0 bis 7. Aus technischen Gründen ist das Byte eine übliche Größe für die kleinste Zusammenfassung von Bits. Es besteht aus acht Bits und kann damit $2^8 = 256$ Zustände annehmen, also beispielsweise die Zahlen von 0 bis 255 oder von –128 bis 127.

Ganze Zahlen

Ein häufig benötigter Typ ist eine ganze Zahl, also eine ungebrochene Zahl ohne Nachkommastellen. Im Englischen und insbesondere im Computer-Umfeld spricht man von einem Integer. Der Typ einer Integer-Variablen heißt `int`. Um eine Variable vom Typ `int` zu definieren, wird zuerst der Typ `int` genannt, dann folgt ein Leerraum und dann der Name der Variablen. Abgeschlossen wird die Definition durch ein Semikolon.

```
int zaehler;
```

Die Zahl, die die Variable `zaehler` aufnimmt, kann positiv oder negativ sein. Sie können aber auch Variablen definieren, die nur positive Zahlen einschließlich der 0 zulassen. Dazu stellen Sie das Schlüsselwort `unsigned` vor den Typ `int`.

```
unsigned int zaehler;
```

Auf diese Weise wird nicht nur verhindert, dass die Variable `zaehler` negativ wird. Der Zahlenbereich ins Positive wird verdoppelt.

Es gibt zwei besondere Fälle von ganzen Zahlen: `short` und `long`. Beide Modifizierer können dem Typ `int` vorangestellt werden. Die Namen der Modifizierer beziehen sich auf die Größe der maximal darstellbaren Zahl und damit auf den Speicher, der einer solchen Variablen zur Verfügung gestellt wird. Seit C++11 kann sogar `long long` vorangestellt werden, wenn besonders große Zahlen benötigt werden. Variablen vom Typ `short` oder `long` können vorzeichenlos definiert werden. Dann wird auch ihnen das Schlüsselwort `unsigned` vorangestellt.

```
unsigned short int zaehler;
```

Wenn ein Modifizierer wie `short` oder `unsigned` verwendet wird, kann das Schlüsselwort `int` auch weggelassen werden. Damit sind auch folgende Definitionen zulässig und werden vom Compiler als attributierte Integer-Variablen verstanden:

```
unsigned short zaehler;
short Wenig;
unsigned Positiv;
```

Listing 1.10 Integer-Variablen ohne int

Eine Variable vom Typ `short` belegt immer mindestens zwei Bytes. Eine Variable vom Typ `int` ist mindestens so groß wie ein `short`. Eine Variable vom Typ `long` umfasst mindestens vier Bytes und ist mindestens so groß wie ein `int`. Wie viele Bytes die verschiedenen Typen annehmen, ist nicht festgelegt, sondern liegt im Ermessen des Compiler-Herstellers.

Zahlenkodierung

Ganzzahlige Werte werden binär kodiert. Belegt eine Integer-Variable zwei Bytes, so stehen 16 Bits zur Verfügung. Zur Darstellung der Zahl 0 werden alle Bits auf 0 gesetzt. Die Zahl 1 wird durch 15 Nullen und eine 1 kodiert. Die in Tabelle 1.1 dargestellten Zahlen zeigen die Binärkodierung.

Dual	Dezimal
0000000000000000	0
0000000000000001	1
0000000000000010	2
0000000000000011	3
0000000000000100	4
0000000000000101	5
0111111111111111	32.767

Tabelle 1.1 Binärkodierung positiver Werte

Die binären Zahlen kann ein Computer leicht addieren und subtrahieren. Das Addieren funktioniert genau so, wie unsere Schulkinder das Addieren im Zehnersystem lernen. Allerdings gibt es nur zwei Ziffern. So lassen sich die einstelligen Additionen leicht aufzählen: 0 + 0 ergibt 0. 1 + 0 ergibt 1. Das ist auch bei 0 + 1 so. 1 + 1 ergibt allerdings nicht 2, denn die gibt es ja nicht, sondern 0 mit

einem Übertrag, also 10. Wenn eine weitere binäre Stelle berechnet wird, fließt der Übertrag in die Berechnung ein. Sind alle Bits durchgerechnet und es bleibt immer noch ein Übertrag, verfällt er.

Analog funktioniert das Subtrahieren. 0 – 0 ergibt wie 1 – 1 naheliegenderweise 0. 1 – 0 ergibt 1. 0 – 1 ergibt 1 und einen Übertrag, der auf die nächste Stelle übernommen wird, wenn es eine gibt.

Nachdem die Kodierung der positiven Zahlen klar ist, stellt sich die Frage, wie negative Zahlen auf Bit-Ebene aussehen. Die Kodierung muss mit den üblichen Berechnungen möglichst kompatibel sein. Wenn Sie –1 und 1 addieren, soll möglichst 0 herauskommen. Dazu errechnen wir die –1, indem wir von 0 die Zahl 1 abziehen. Wenn wir mit der Stelle ganz rechts beginnen, ist 0 – 1 = 1. Dabei entsteht ein Übertrag, der auf die nächste Stelle umgelegt wird. Dadurch entsteht auch in der nächsten Stelle die Aufgabe 0 – 1, was wiederum 1 und einen Übertrag ergibt.

```
          0000
-         0001
Übertrag: 111
          ----
Ergebnis: 1111
```

Listing 1.11 Binäre Berechnung von 0 – 1

Der Übertrag setzt sich also durch alle Stellen fort. Daraus ergibt sich, dass die Zahl –1 binär dargestellt wird, indem alle Bits auf 1 gesetzt werden. Sollte Ihnen das unlogisch erscheinen, überlegen Sie, welche Zahl 0 ergibt, wenn Sie eine 1 hinzuaddieren. Hier sehen Sie die Kontrollrechnung:

```
          1111
+         0001
Übertrag: 111
          ----
Ergebnis: 0000
```

Listing 1.12 Binäre Berechnung von –1 + 1

Die Zahlen –2, –3 und so weiter ergeben sich durch jeweiliges Dekrementieren, wie Tabelle 1.2 zeigt.

Dabei lässt sich leicht erkennen, dass das Vorzeichen der Zahl am ersten Bit abzulesen ist. Steht hier eine 1, ist die Zahl negativ. Anhand der Kodierung lässt sich auch ermitteln, wie groß die größten und kleinsten Werte sind. Bei zwei Bytes

Dual	Dezimal
0000000000000000	0
1111111111111111	–1
1111111111111110	–2
1111111111111101	–3
1111111111111100	–4
1111111111111011	–5
1000000000000000	–32.768

Tabelle 1.2 Binärkodierung negativer Werte

geht der Wert von –32.768 bis +32.767. Werden vier Bytes eingesetzt, ergeben sich –2.147.483.648 bis +2.147.483.647.

Hinweis

Ist das erste Bit einer binär kodierten Zahl 1, so ist die Zahl negativ.

Zur Kontrolle berechnen wir noch einmal –3 + 5. Das Ergebnis sollte 2 sein.

```
          1101
+         0101
Übertrag: 1 1
          ----
Ergebnis: 0010
```

Listing 1.13 Binäre Berechnung von –3 + 5

Werden die Variablen ohne Vorzeichen verwendet, ist die 0 die kleinste Zahl. Das erste Bit wird dann nicht zur Vorzeichenkodierung verwendet, sondern erhöht den maximalen Wert. Dadurch hat eine `short`-Variable einen Wertebereich von 0 bis 65.535. Wird die obere Grenze um 1 erhöht, ergibt sich wieder die 0.

Warnung

Etwas tückisch ist die Tatsache, dass Sie vor einem Überlauf der Grenzen von Integer-Werten nicht gewarnt werden. Enthält eine Integer-Variable den maximal darstellbaren Wert und wird diese dann um 1 erhöht, so enthält sie anschließend den kleinstmöglichen Wert. Damit kann das Programm wunderbar weiterarbeiten. Es kann aber sein, dass Sie das nicht beabsichtigt haben. Es ist Ihre Aufgabe, dafür zu sorgen, dass ein solcher Überlauf nicht versehentlich auftritt.

Dual	Dezimal
0000000000000000	0
0000000000000001	1
0000000000000010	2
0000000000000011	3
0000000000000100	4
0000000000000101	5
1111111111111110	65.534
1111111111111111	65.535

Tabelle 1.3 Binärkodierung vorzeichenloser Werte

Zeichen

C++ verfügt über den Datentyp `char`, der Buchstaben und andere Zeichen aufnehmen kann. Dieser Datentyp ist genau ein Byte groß. Ein Byte kann 256 Zustände annehmen. Das heißt, es gibt 256 unterschiedliche Kombinationen von Nullen und Einsen. Damit können beispielsweise die Zahlen von 0 bis 256 kodiert werden. Es gibt 26 Buchstaben, jeweils klein und groß, also werden mindestens 52 Zeichen benötigt. Dazu kommen die zehn Ziffern und einige Sonderzeichen. Die ersten 128 Zeichen sind international genormt und bei allen ASCII-Zeichensätzen gleich. Die verbleibenden 128 Zustände werden für nationale Sonderzeichen wie die deutschen Umlaute oder die französischen Sonderzeichen verwendet.

Solange die nationalen Sonderzeichen nur diejenigen von Westeuropa umfassen, reicht ein Byte pro Buchstabe. Dieser Zeichensatz ist als ISO 8859-1 genormt. Für russische oder türkische Zeichen gibt es wiederum einen anderen Zeichensatz. Aber mit welchem Recht kann man die arabischen, japanischen und hebräischen Zeichen ausschließen? Was passiert, wenn ein Programm russische, deutsche und hebräische Zeichen gleichzeitig benötigt? Für diese vielen Zeichen ist in einem einzelnen Byte kein Platz mehr.

Mit zunehmender Internationalisierung entsteht der Bedarf nach einheitlicher Kodierung der jeweils nationalen Sonderzeichen. Seit Anfang der 1990er Jahre wurde UNICODE vorangetrieben, um möglichst alle Schriftzeichen der Welt in einem System zu kodieren. Schnell war klar, dass die Anzahl der Zeichen nicht in einem Byte Platz findet. Also legte man zwei Bytes fest. In die 16 Bits passen bis zu 65.536 Zeichen. Für solche Zeichensätze besitzt C++ einen speziellen Datentyp namens `wchar_t`. Wie viele Bytes er belegt, ist implementierungsabhängig. Mit dem Stan-

dard C++11 wurde für 16-Bit UNICODE der Typ `char16_t` eingeführt. Die Annahme, zwei Bytes könnten alle Zeichen der Welt fassen, hat sich allerdings als Illusion herausgestellt, da vor allem die asiatischen Sprachen sehr viele Schriftzeichen besitzen. Dazu gibt es einen 32-Bit-UNICODE, der in C++11 mit dem Typ `char32_t` unterstützt wird. Aber warum sollte auf einem amerikanischen Computer die vierfache Speichermenge reserviert werden, nur für den Fall, dass dieser eines Tages von chinesischen Sonderzeichen heimgesucht wird, die dessen Besitzer vermutlich nicht einmal lesen kann? Darum wird meist UTF-8 eingesetzt. Hier wird der UNICODE in einzelnen Bytes, also im Typ `char`, abgelegt. Der Vorteil liegt auf der Hand: Auch Programme, die sich nicht um internationale Zeichen kümmern, können noch eingesetzt werden.

Zahl oder Ziffer?

Besonders verwirrend ist für den Anfänger oft die Unterscheidung zwischen Zahl und Ziffer. Eine Ziffer ist ein Zeichen (also quasi ein Buchstabe), das zur Darstellung von Zahlen (also Werten) dient. Im Zusammenhang mit der Programmierung ist eine Ziffer im Allgemeinen vom Typ `char`, also das Zeichen, das die Ziffer beispielsweise auf dem Bildschirm darstellt. Eine Zahl ist dagegen ein Wert, mit dem gerechnet werden kann.

Um die Verwirrung komplett zu machen, sind Buchstaben intern eigentlich auch Zahlen. Jeder Buchstabe wird durch eine Zahl repräsentiert, die in ein Byte passt, also durch eine Zahl zwischen 0 und 255. Die Ziffern als Buchstaben sind ebenfalls intern als Zahl kodiert, aber nicht etwa gleich ihrem Zahlenwert. Sie finden im ASCII-Zeichensatz ihre Position ab der Nummer 48. Die Ziffer '0' wird also als 48 kodiert, die Ziffer '1' als 49 und so fort. Dies muss besonders berücksichtigt werden, wenn Zahleneingaben von der Tastatur verarbeitet werden sollen. Wenn die einzelnen Ziffern als Buchstaben eingelesen werden, muss 48, also der Gegenwert der Ziffer '0' abgezogen werden, um ihren Zahlenwert zu erlangen. Übrigens ist ASCII nicht der einzige Zeichensatz. Beispielsweise ist auf den IBM-Großrechnern der Zeichensatz EBCDIC gebräuchlich. Das Beste ist, man trifft als Programmierer keine Annahmen darüber, wie die Ziffern kodiert sind.

Mit char rechnen

Besonders verwirrend scheint es, dass C++ durchaus erlaubt, mit Variablen vom Typ `char` zu rechnen. Tatsächlich stört es C++ auch nicht, wenn Sie einer Integer-Variablen einen Buchstaben zuweisen. Der Typ `char` besagt in erster Linie, dass ein

Byte Speicher zur Verfügung steht. Der Inhalt kann sowohl als Zeichen als auch als kleine Zahl interpretiert werden.

Da eine Variable vom Typ `char` eigentlich nur eine kleinere `int`-Variable ist, gibt es auch ein Vorzeichen. Wie bei `int` ist auch die `char`-Variable zunächst vorzeichenbehaftet. Das hat Konsequenzen bei Umlauten. Wie oben erwähnt, liegen die nationalen Sonderzeichen in den hinteren 128 Positionen. Also steht bei einem nationalen Sonderzeichen das erste Bit auf 1. Das wird aber bei einer normalen `char`-Variablen als Zeichen für eine negative Zahl interpretiert. Um Missinterpretationen dieser Art zu vermeiden, sollten Sie eine `char`-Variable im Zweifelsfall mit dem Modifizierer `unsigned` versehen. Ansonsten kann es sein, dass ein `'ß'` kleiner ist als ein `'a'`, da es durch das gesetzte erste Bit als negativ interpretiert wird. Eine Sortierung würde dann alle nationalen Sonderzeichen vor den eigentlichen Buchstaben erscheinen lassen. (Bei Verwendung von `unsigned` werden die nationalen Sonderzeichen hinter dem `'z'` einsortiert, was auch etwas gewöhnungsbedürftig ist.)

Fließkommazahlen

In der realen Welt sind ganzzahlige Werte oft nicht ausreichend. Bei Gewichten, Geschwindigkeiten und anderen Werten aus der Physik ist immer mit Nachkommastellen zu rechnen. Und auch bei Preisen werden Nachkommastellen benötigt. Dieser Tatsache kann sich auch eine Computersprache nicht entziehen. Um eine Fließkommazahl darzustellen, gibt es eine Normalform, in der das Komma vor die erste Ziffer geschoben wird. Damit die Zahl gleich bleibt, wird sie mit Zehnerpotenzen multipliziert. Die folgenden Zahlen sind identisch:

$823{,}25 = 823{,}25 * 1 = 823{,}25 * 10^0$
$823{,}25 = 82{,}325 * 10 = 82{,}325 * 10^1$
$823{,}25 = 8{,}2325 * 100 = 8{,}2325 * 10^2$
$823{,}25 = 0{,}82325 * 1000 = 0{,}82325 * 10^3$

Mantisse und Exponent

Die unterste Zahl, bei der die erste signifikante Ziffer hinter dem Komma steht, ist die Standarddarstellung. Von dieser Darstellung ausgehend, hat eine Fließkommazahl zwei Komponenten. Die eine ist die Mantisse, hier 82325. Eine Mantisse ist der Zahlenanteil einer Fließkommakonstanten ohne Exponenten. Die andere Komponente ist der Exponent zur Basis 10, hier 3. Der Exponent wird oft durch ein kleines oder großes E abgetrennt. So würde unsere Zahl als 0.82325E3 dargestellt. Das Komma ist aus Sicht des Computers ein Punkt.

Der einfachste Typ mit Nachkommastellen heißt `float`. Auch die Zahlen dieses Typs sind binär kodiert, damit sie effizient im Computer verarbeitet werden können. Die Zahl lässt ganzzahlige, aber auch negative Exponenten zu. Dadurch sind nicht nur sehr große Zahlen darstellbar, sondern auch sehr kleine Brüche. Die Speicheranforderung einer `float`-Variablen ist nicht sehr hoch, typischerweise liegt sie bei vier Bytes. Dennoch können Zahlen in der Größenordnung von etwa 10^{38} dargestellt werden. Dafür geht eine solche Variable Kompromisse in der Genauigkeit der Mantisse ein.

Reicht die Genauigkeit nicht aus oder werden Größenordnungen benötigt, die über die Kapazität einer `float`-Variablen hinausgehen, steht der Typ `double` zur Verfügung. Der Name bedeutet »doppelt« und bezieht sich auf die Genauigkeit. Die Genauigkeit geht zulasten des Speichers. Auf vielen Compilern belegt eine `double`-Variable auch doppelt so viel Speicher wie eine `float`-Variable. Und natürlich erhöht die Berechnung doppelt genauer Werte auch die Laufzeit eines Programms.

Werden besonders genaue Werte benötigt, verfügt ein ANSI-C++-Compiler über einen Datentyp, der noch genauer ist als der Typ `double`. Das ist der Typ `long double`. Dieser belegt je nach Compiler 10 bis 16 Bytes.

Beim Umgang mit Fließkommazahlen ergeben sich leicht Genauigkeitsprobleme. Das beruht zum einen auf der begrenzten Zahl von Stellen der Mantisse. Bei dezimalen Nachkommastellen kann aber zum anderen auch die interne binäre Kodierung eine der Ursachen sein. Sie können dies feststellen, wenn Sie eine Variable auf –1,0 setzen und schrittweise um 0,1 erhöhen. Sie werden auf diese Weise auf den meisten Systemen den Wert 0,0 nicht exakt treffen. Der Grund ist, dass 0,1 in binärer Darstellung ebenso eine Periode darstellt wie ein Drittel in Dezimaldarstellung.

Das 0,1-Problem

Um 0,1 dezimal im binären Zahlensystem darzustellen, müssen binäre Nachkommastellen verwendet werden. Binäre Nachkommastellen müssen Sie sich so vorstellen, dass 0,1 ein Halb, 0,01 ein Viertel und 0,001 ein Achtel ist. Zur Darstellung eines Zehntels ist ein Achtel zu viel. Ein Sechszehntel ist 0,0625. Es verbleiben 0,0375 zu einem Zehntel. Ein Zweiunddreißigstel ist 0,03125. Es bleiben 0,00625. Ein 256stel ist 0,00390625. Durch Weiterberechnen kommen Sie auf eine binäre Darstellung von 0,00011001100110011 und noch immer bleibt etwas übrig. Wenn Sie das Ganze zu Ende rechnen, werden Sie feststellen müssen, dass es niemals aufgeht. Und so wie einige Taschenrechner ein Problem damit haben, bei drei Dritteln

auf ein Ganzes zu kommen, haben Computer mit ihren binär kodierten Fließkommazahlen ein Problem bei der Berechnung von zehn Zehnteln.

In einigen Programmiersprachen und auch in Datenbanken gibt es explizite Typen mit einer dezimalen Anzahl von Nachkommastellen. Solche Werte sind vor allem bei Währungen sehr exakt. Allerdings hat die Genauigkeit bereits an der nächsten Tankstelle ihr Ende, wo der Literpreis auch 0,9 Cent enthält. C++ kennt keine Nachkommavariablen, sondern nur Fließkommatypen. Das heißt, dass so viele Nachkommastellen gebildet werden, wie benötigt werden und darstellbar sind.

Größen und Limits

Die Sprache C++ legt die Speicheranforderung der meisten Typen nicht fest. Solche Implementierungsdetails werden den Compilern überlassen. Lediglich die Qualitätsunterschiede zwischen den Typen werden gesichert. Sie können sich also darauf verlassen, dass ein `short` nicht größer ist als ein `long`. Tabelle 1.4 gibt eine Übersicht, welche Größenordnungen in der Praxis derzeit üblich sind.

Typ	Typische Größe	Typische Verwendung
`char`	1 Byte	Buchstaben, Zeichen und Ziffern
`wchar_t`	2 oder 4 Bytes	Internationale Zeichencodierung
`char16_t`	2 Bytes	UNICODE 16 Bit
`char32_t`	4 Bytes	UNICODE 32 Bit
`short int`	2 Bytes	Zahlen für Nummerierungen oder Positionen
`int`	2 oder 4 Bytes	Standardgröße für ganze Zahlen
`long int`	4 oder 8 Bytes	Große Werte ohne Nachkommastellen
`long long int`	8 Bytes	Sehr große Werte ohne Nachkommastellen
`float`	4 Bytes	Analog ermittelte Werte mit Nachkommastellen
`double`	8 Bytes	Berechnungen und höhere Preise
`long double`	12 Bytes	Berechnungen höherer Genauigkeit

Tabelle 1.4 Übliche Speichergröße der Typen

Welche Größe welcher Typ letztlich wirklich hat, hängt nicht von der Laune eines Compiler-Herstellers ab, sondern beispielsweise auch von der Hardware- oder Betriebssystemarchitektur. Die gängigen Maschinen hatten bislang meist eine Wortbreite von 32 Bits, also vier Bytes. Entsprechend können vier Bytes sehr effizient verarbeitet werden. Die heutigen Modelle werden als 64-Bit-Systeme ausgeliefert.

Auf solchen Systemen wird der Typ `long` typischerweise mit acht Bytes implementiert, da ein kürzerer Typ ineffizient wäre. Zu den Zeiten, als C++ entstand, waren 16-Bit-Maschinen die Regel. Hätte man damals den Typ `int` auf zwei Bytes festgelegt, würden die zukünftigen Compiler einen höheren Aufwand betreiben müssen, nur um einen 64-Bit-Speicherplatz auf 16 Bits zu begrenzen.

sizeof()

Wenn Sie konkret die Größe eines Typs wissen müssen, können Sie die Funktion `sizeof()` verwenden. Diese liefert die Größe eines Typs oder einer Variablen. Die folgende Beispielzeile gibt aus, wie viele Bytes der Typ `double` auf Ihrem System beansprucht.

```
cout << sizeof(double) << endl;
```

Genauso kann zwischen die Klammern von `sizeof()` eine Variable oder ein selbstdefinierter Typ gestellt werden. Das Ergebnis ist immer der Speicherbedarf in Bytes. (In dieser Flexibilität hat `sizeof()` eine Sonderstellung. Ansonsten können Funktionen nur Werte oder Variablen übergeben werden. Deren Typ wird zudem vom Compiler geprüft.)

In der Datei *limits.h* werden die Größen der Grundtypen festgelegt. So findet sich hier die Definition von `INT_MAX`. Dies ist der höchste Wert, den eine Integer-Variable annehmen kann. Mit `CHAR_MAX` erfahren Sie die gleiche Information zum Typ `char`. Es wird Sie nicht verwundern, dass der Wert `LONG_MAX` den maximalen Wert liefert, den eine Variable vom Typ `long` annehmen kann. Dafür könnte ich Sie wahrscheinlich mit `SHRT_MAX` als dem größten Wert für eine `short`-Variable überraschen. Denn Sie würden nach den Vorgaben vermuten, dass noch ein O darin vorkommt.

Zu all diesen Definitionen gibt es also auch eine passende Variation mit der Nachsilbe `MIN`, mit der Sie erfahren, welches der kleinstmögliche Wert des entsprechenden Typs ist. Schließlich können Sie den jeweiligen Definitionen ein großes U voranstellen, um zu erfahren, welches der größte Wert ist, wenn der Typ den Modifizierer `unsigned` trägt. Wenn Sie Ihre Kollegen einem kurzen Intelligenztest unterziehen wollen, fragen Sie doch mal, warum `ULONG_MIN` nicht definiert wurde.

Konstante	Bedeutung
INT_MAX	Höchster Wert einer int-Variablen
INT_MIN	Niedrigster Wert einer int-Variablen
UINT_MAX	Höchster Wert einer unsigned int-Variablen
CHAR_MAX	Höchster Wert einer char-Variablen
CHAR_MIN	Niedrigster Wert einer char-Variablen
WCHAR_MAX	Höchster Wert einer wchar_t-Variablen
WCHAR_MIN	Niedrigster Wert einer wchar_t-Variablen
UCHAR_MAX	Höchster Wert einer unsigned char-Variablen
SHRT_MAX	Höchster Wert einer short-Variablen
SHRT_MIN	Niedrigster Wert einer short-Variablen
USHRT_MAX	Höchster Wert einer unsigned short-Variablen
LONG_MAX	Höchster Wert einer long-Variablen
LONG_MIN	Niedrigster Wert einer long-Variablen
ULONG_MAX	Höchster Wert einer unsigned long-Variablen

Tabelle 1.5 Limit-Konstanten

1.4.5 Syntax der Variablendefinition

Der Syntaxgraph in Abbildung 1.5 zeigt die Variablendefinitionen der grundlegenden Typen. Zuerst wird der Typ der Variablen genannt. Es folgt der Name der Variablen, der bereits als Syntaxgraph in Abbildung 1.3 dargestellt wurde. Durch Anhängen eines Gleichheitszeichens und einer Konstanten kann eine Initialisierung erfolgen. Die Variablendefinition wird durch ein Semikolon abgeschlossen. Es können mehrere Variablen in einer Anweisung definiert werden, indem man sie durch ein Komma voneinander trennt.

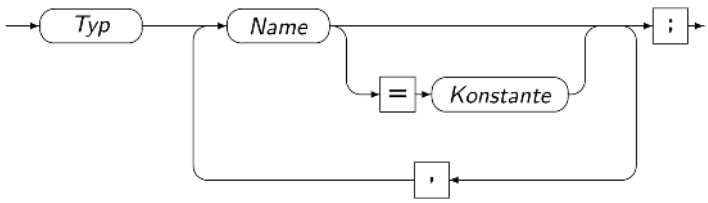

Abbildung 1.5 Syntaxgraph einer Variablendefinition

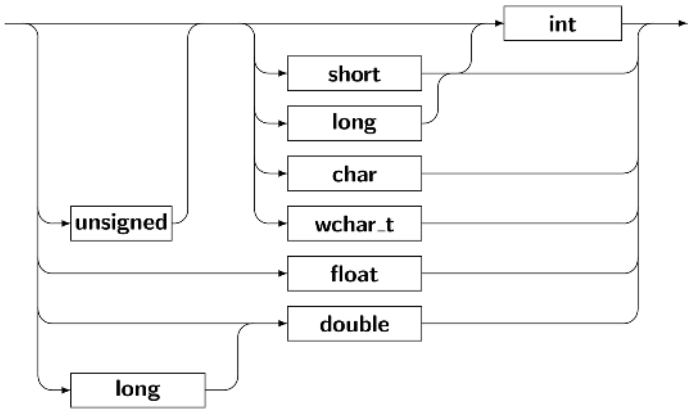

Abbildung 1.6 Syntaxgraph einer Typbeschreibung

Die gängigsten Typen werden in dem Syntaxgraphen in Abbildung 1.6 dargestellt. Es ist zu erkennen, dass der Typ `char`, `wchar_t`, `int`, `float` oder `double` sein kann. Vor `char` und den Integer-Typen darf `unsigned` gestellt werden. Vor `int` können die Modifizierer `long` und `short` verwendet werden. Wird einer dieser beiden Modifizierer verwendet, muss `int` nicht mehr explizit genannt werden.

1.4.6 Konstanten

Eine Konstante ist ein unveränderlicher Wert. Alle Zahlen in einem Programm sind Konstanten, da ihr Wert, im Gegensatz zu Variablen, nicht geändert werden kann. In C++ muss auch Konstanten ein Typ zugewiesen werden.

Ganze Zahlen

Am einfachsten sind ganzzahlige Konstanten zu verstehen. Es sind Zahlen wie 5, –3, 17 oder 5498. Dabei ist nur darauf zu achten, dass eine Zahlkonstante nur aus Ziffern besteht und keine Leerzeichen oder andere Sonderzeichen enthält. Einer negativen Zahl wird das Minuszeichen vorangestellt. Bei positiven Zahlen ist das Pluszeichen erlaubt, aber nicht erforderlich. Im Syntaxgraphen in Abbildung 1.7 wird das Symbol *Ziffern* verwendet. Dieses steht für eine beliebige Folge von dezimalen Ziffern.

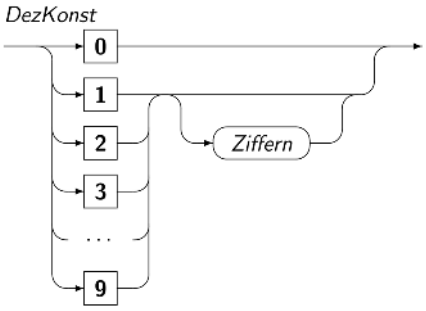

Abbildung 1.7 Syntaxgraph einer dezimalen Zahlkonstanten (DezKonst)

Hinweis

Eine dezimale, ganzzahlige Konstante ist 0 oder beginnt mit einer Ziffer ungleich 0. Ihr folgen beliebig viele weitere Ziffern einschließlich der 0.

Der Syntaxgraph in Abbildung 1.8 beschreibt diese dezimale Ziffernfolge. Das »Auslagern« dieses Teils des Syntaxgraphen hat seinen Grund darin, dass er später auch zur Definition der Nachkommakonstanten noch benötigt wird.

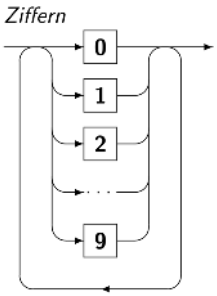

Abbildung 1.8 Syntaxgraph einer dezimalen Ziffernfolge (Ziffern)

Wie schon erwähnt, darf einer Zahlkonstanten ein Plus- oder Minuszeichen vorangestellt werden. Im Syntaxgraphen in Abbildung 1.9 wird *DezKonst* noch um die Vorzeichen ergänzt.

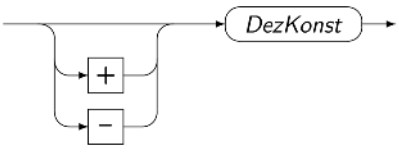

Abbildung 1.9 Syntaxgraph einer dezimalen Zahlkonstanten mit Vorzeichen

Tabelle 1.6 zeigt einige Beispiele unzulässiger Zahlkonstanten.

Konstante	Grund
`1 234`	Es befindet sich ein Leerzeichen zwischen 1 und 2.
`1,234`	Das Komma darf nie in einer Konstanten stehen.
`1-`	Das Vorzeichen muss vor der Zahl stehen.
`12kg`	Einheiten sind nicht erlaubt.

Tabelle 1.6 Unzulässige ganzzahlige Konstanten

Führende Null

Neben den Dezimalzahlen gibt es in C++ auch die Möglichkeit, das oktale und das hexadezimale Zahlensystem zur Darstellung von Konstanten zu verwenden. Diese Möglichkeit ist in erster Linie für Programmierer interessant, die auf Controller-Ebene programmieren. Als Programmieranfänger wird Sie das vielleicht nicht besonders interessieren, und Sie können mit den Fließkommakonstanten im Abschnitt 1.4.6 fortfahren. Aber auch für den Anfänger ist folgende Regel wichtig:

Warnung ×

Jede ganzzahlige Konstante, die mit einer 0 beginnt, wird als nicht dezimale Konstante interpretiert. Vermeiden Sie also, einer Konstanten eine überflüssige 0 voranzustellen, sofern Sie nicht genau wissen, was Sie tun.

Beginnt eine Zahlkonstante mit 0 und folgt der 0 ein x, ist es eine hexadezimale Konstante. Das ist eine Zahl zur Basis 16, die die Ziffern 0 bis 9 und die Buchstaben A bis F für die Ziffern 10 bis 15 verwendet. In diesem Fall dürfen auch die Kleinbuchstaben a bis f verwendet werden. Folgt der 0 eine Ziffer, ist es eine Oktalzahl, also eine Zahl zur Basis 8. Erlaubte Ziffern sind die Ziffern von 0 bis 7. Tabelle 1.7 zeigt den unterschiedlichen Wert der Ziffernfolge 11.

Konstante	Zahlensystem	Dezimaler Wert
`11`	dezimal (Basis 10)	$1*10^1 + 1*10^0 = 1*10 + 1*1 = 11$
`011`	oktal (Basis 8)	$1*8^1 + 1*8^0 = 1*8 + 1*1 = 9$
`0x11`	hexadezimal (Basis 16)	$1*16^1 + 1*16^0 = 1*16 + 1*1 = 17$

Tabelle 1.7 Zahlensysteme

Die Berechnung der Werte anderer Zahlensysteme erfolgt genau so, wie Sie es intuitiv mit dem normal üblichen, dezimalen Zahlensystem auch tun. Im Dezimalsystem multiplizieren Sie jede Stelle mit der zugehörigen Zehnerpotenz. Der Exponent beginnt rechts mit der 0. Damit ist der Faktor, mit der die äußerste rechte Ziffer multipliziert wird, 10^0 oder 1. Die zweite Stelle von rechts wird mit 10^1, also 10, multipliziert. Es folgt 10^2 gleich 100, 10^3 gleich 1.000 und so weiter.

Hexadezimale Zahlen

Die hexadezimalen[1] Zahlen haben die Basis 16. Damit reichen die dezimalen Ziffern 0 bis 9 nicht für eine hexadezimale Ziffer aus. Darum werden die Buchstaben A bis F zu Hilfe genommen. A hat den Wert 10, B ist 11 und so weiter. Schließlich hat F als höchste Ziffer den Wert 15. Das Vorgehen zur Ermittlung des Wertes einer hexadezimalen Konstanten entspricht dem im dezimalen System. Die äußerste rechte Stelle bleibt, weil auch $16^0 = 1$ ist. Die nächste Stelle wird mit 16 multipliziert, die darauf folgende Stelle mit $16^2 = 256$ und so fort. Damit wäre 0x168 gleich 1 * 256 + 6 * 16 + 8, also dezimal 360.

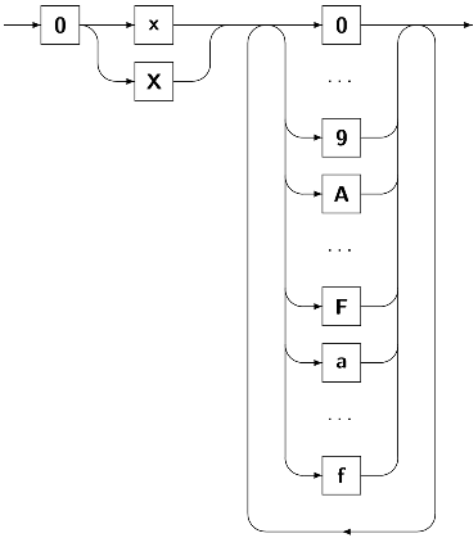

Abbildung 1.10 Syntaxgraph einer hexadezimalen Zahlkonstante

1 Es gibt einige Programmierer mit humanistischer Bildung, die darauf bestehen, dass es eigentlich sedezimal heißen müsste, weil »hexa« aus dem Griechischen und »decem« aus dem Lateinischen stamme. Sie werden sich nicht viele Freunde machen, wenn Sie diese Erkenntnis verbreiten und Ihre Kollegen damit auf ihre Bildungslücken in Latein und Griechisch stoßen. Also sollten Sie diese Anmerkung schnell wieder vergessen.

Hinweis

Eine hexadezimale, ganzzahlige Konstante beginnt mit der Zeichenfolge 0x oder 0X. Ihr folgen beliebig viele weitere Ziffern zwischen 0 und 9 beziehungsweise A bis F. Die Ziffern A bis F dürfen auch als Kleinbuchstaben dargestellt werden.

Oktale Zahlen

Oktale[2] Zahlen bestehen aus den Ziffern 0 bis 7. Hier werden die Stellen mit Achterpotenzen berechnet. Sie haben von rechts nach links also die Faktoren 1, 8, 64 und so fort. Die Konstante 0167 wäre umgerechnet also 1 * 64 + 6 * 8 + 7. Damit entspricht die Konstante 0167 dem dezimalen Wert 119.

Hinweis

Eine oktale, ganzzahlige Konstante beginnt immer mit der 0. Ihr folgen beliebig viele weitere Ziffern zwischen 0 und 7.

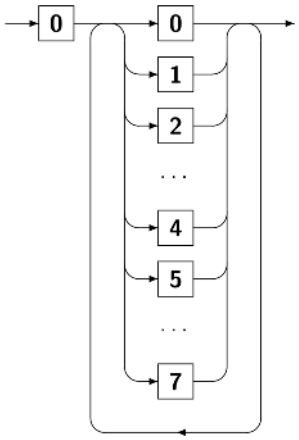

Abbildung 1.11 Syntaxgraph einer oktalen Zahlkonstanten

2 Der Begriff »oktal« leitet sich von Oktopus (Krake) her. Da Kraken nur acht Arme haben, können sie nicht bis zehn zählen und haben darum dieses Zahlensystem erfunden, damit es die Krakenkinder leichter in der Schule haben.

Fließkommakonstanten

Fließkommazahlen sind Zahlen mit Nachkommastellen. Leider ist das Komma als Dezimaltrennzeichen international nicht üblich. Vor allem im englischsprachigen Raum wird stattdessen der Punkt verwendet. Dementsprechend versteht C++ die Schreibweise 1,2 nicht, sondern bevorzugt 1.2.

Eine Fließkommakonstante beginnt mit der Mantisse. Die Mantisse ist der Anteil einer Fließkommakonstanten ohne Exponenten. Sie hat optional ein Vorzeichen, auf das eine dezimale Ganzzahl folgt. Dann können der Dezimalpunkt und die Nachkommastellen als dezimale Ganzzahl erscheinen. Steht vor dem Dezimalpunkt nur eine 0, kann sie weggelassen werden.

Wie bei einem Taschenrechner lassen sich auch Exponenten verwenden. Dabei wird als Basis stillschweigend 10 vereinbart. Um den Exponenten von der Mantisse zu trennen, wird ein großes oder kleines E verwendet. Der Exponent selbst ist eine ganzzahlige Dezimalkonstante, kann also auch negativ sein. Tabelle 1.8 zeigt ein Beispiel.

Exponentialdarstellung	Wert	Darstellung ohne Exponenten
`1E3`	10^3	`1000.0`
`1E0`	10^0	`1.0`
`1E-3`	10^{-3}	`0.001`

Tabelle 1.8 Exponentialdarstellung

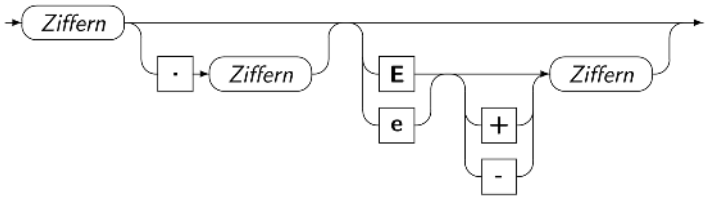

Abbildung 1.12 Syntaxgraph einer Fließkommakonstanten

Im Syntaxgraphen in Abbildung 1.12 wird dreimal auf *Ziffern* verwiesen. Dabei handelt es sich um eine dezimale Ziffernfolge, wie sie im Syntaxgraphen in Abbildung 1.8 beschrieben ist.

Eine ganzzahlige Konstante wird durch Anhängen von `.0` zu einer Fließkommakonstanten vom Typ `double`. Wird ein f angehängt, erhält sie den Typ `float`.

Zeichenkonstanten

Einzelne Zeichen, also Konstanten vom Typ `char`, werden in Hochkommata gesetzt. Diese Form der Darstellung verbessert die Lesbarkeit für den Menschen. Intern verwendet der Computer für Buchstaben Zahlen.

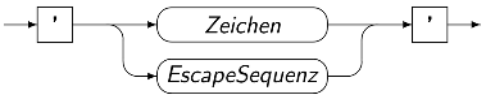

Abbildung 1.13 Syntaxgraph einer Zeichenkonstanten

Soll die Konstante explizit für den Typ `wchar_t` gesetzt werden, wird vor das erste Hochkomma ein großes L gesetzt. Bei Konstanten für den Typ `char16_t` wird ein kleines u und für `char32_t` ein großes U vor das erste Hochkomma gestellt.

ASCII

Welche Zeichen auf welche Zahlen abgebildet werden, hängt vom Zeichensatz ab. Auf den meisten Systemen wird heute ASCII (American Standard Code for Information Interchange) verwendet. Die folgende Tabelle stellt die ersten 127 Zeichen dar. Zur Ermittlung der Kodierung steht vor jeder Zeile die Nummer des ersten Zeichens in dezimaler und hexadezimaler Schreibweise. Über den Zeichen steht die Zahl, die zu der Spaltennummer addiert werden muss. So hat der Buchstabe `'Z'` die dezimale Nummer 80 + 10, also 90. Hexadezimal ist dies 0x50 + 0x0A, also 0x5A.

```
dez          0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
     hex     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F

 00   0        ^A ^B ^C ^D ^E ^F ^G ^H ^I ^J ^K ^L ^M ^N ^O
 16  10     ^P ^Q ^R ^S ^T ^U ^V ^W ^X ^Y ^Z
 32  20        !  "  #  $  %  &  '  (  )  *  +  ,  -  .  /
 48  30     0  1  2  3  4  5  6  7  8  9  :  ;  <  =  >  ?
 64  40     @  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O
 80  50     P  Q  R  S  T  U  V  W  X  Y  Z  [  \  ]  ^  _
 96  60     `  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
112  70     p  q  r  s  t  u  v  w  x  y  z  {  |  }  ~
```

Im ASCII-Zeichensatz werden die ersten Positionen für Kontrollzeichen verwendet. In der Tabelle werden sie durch die Tasten angedeutet, die gedrückt werden

müssen, um sie zu erzeugen. Dabei wird ^ als Zeichen für die Strg- oder Ctrl-Taste verwendet. Werden diese Zeichen ausgegeben, steuern sie das Ausgabegerät. Darin befindet sich beispielsweise das Zeilenendezeichen mit der Nummer 10 oder das Piepzeichen mit der Nummer 7.

Ab der Nummer 32 beginnen die druckbaren Zeichen. Das sind die Zeichen, die einfach ausgegeben werden und vom Drucker nicht als Steuerungszeichen ausgewertet werden. Kurioserweise ist gerade das erste dieser Zeichen unsichtbar: das Leerzeichen mit der Nummer 32 oder hexadezimal 0x20. Es geht mit dem Ausrufezeichen und anderen Sonderzeichen weiter.

Ab der Nummer 48 beginnen im ASCII-Zeichensatz die Ziffern. Die `'0'` ist als 48 kodiert, die `'1'` als 49. So geht es fort bis zur `'9'` als 57. Als Programmierer sollten Sie sehr genau zwischen einer `'0'` und einer 0 unterscheiden. Wenn der Anwender Zahlen eintippt, so sind diese eine Folge von Buchstaben. Um aus diesen Buchstaben Zahlen zu berechnen, müssen Sie den Wert der Ziffer `'0'` abziehen. Aus der `'3'` wird eine 3, indem Sie den Wert `'0'` subtrahieren. Sie finden dazu ein Programmbeispiel im Abschnitt 3.1.3.

Die Buchstaben beginnen mit dem Großbuchstaben `'A'` ab 65. Danach folgen die Kleinbuchstaben mit `'a'` ab Nummer 97. Die Buchstaben sind durchgehend aufsteigend sortiert von A bis Z. Internationale Sonderzeichen wie die deutschen Umlaute befinden sich jenseits der 128. Wo sich beispielsweise das `'ö'` befindet und ob es überhaupt vorhanden ist, hängt vom Zeichensatz ab. In der westlichen Welt finden Sie unter UNIX, Linux und MS-Windows meist den Zeichensatz ISO 8859-1 (Latin-1). Osteuropäische Sprachen verwenden Latin-2, und ISO 8859-9 beispielsweise ist ein türkisch angepasster Latin-1-Zeichensatz.

Verlassen Sie sich nicht blind auf die Position der einzelnen Zeichen im Zeichensatz. Zwar ist der ASCII-Zeichensatz auf allen PCs und allen gängigen UNIX-Maschinen zu Hause, aber beispielsweise IBM-Großrechner arbeiten üblicherweise mit dem EBCDIC-Zeichensatz, der völlig anders aufgebaut ist. Sie können sich bestenfalls darauf verlassen, dass Buchstaben und Ziffern in sich aufsteigend sortiert sind und dass die Ziffern lückenlos aufeinanderfolgen. Im folgenden Beispiel liegt ein Zeichen in der Variablen `ZiffernZeichen` vor. Daraus soll der Ziffernwert bestimmt werden.

```
ZiffernWert = ZiffernZeichen - '0';  // richtig
ZiffernWert = ZiffernZeichen - 48;   // nur bei ASCII!
```

In der ersten Zeile wird das Zeichen `'0'` abgezogen. Daraus bildet der Compiler auf einer Maschine mit ASCII-Zeichensatz den Wert 48. Übersetzen Sie diese Zeile auf einer Maschine mit einem anderen Zeichensatz, wird der dort verwendete Code

für die Ziffer `'0'` eingesetzt. Da C++ garantiert, dass alle Ziffern direkt aufeinander folgen, ist das Ergebnis für alle Ziffern korrekt. Wenn stattdessen, wie in der zweiten Zeile, direkt 48 kodiert wird, funktioniert dies nur auf einer Maschine mit ASCII-Zeichensatz. Hinzu kommt, dass in der ersten Zeile die Idee des Programmierers wesentlich besser zum Ausdruck kommt.

Um vom Zeichensatz der Maschine unabhängig zu sein, definiert C++ einige wichtige Sonderzeichen durch Voranstellen eines Backslashs (\). Der Backslash ähnelt dem Schrägstrich (/), den man auf einer deutschen Tastatur über der 7 findet, steht aber in der umgekehrten Richtung. Auf deutschen Tastaturen erzeugt man ihn durch die Kombination der Tasten AltGr und ß. Zum Backslash gehört immer mindestens ein Zeichen, das die Bedeutung der Sequenz kodiert. Die Tabelle 1.9 zeigt die wichtigsten Sonderzeichen.

Sequenz	Bedeutung
`\n`	Neue Zeile (line feed)
`\r`	Zeilenrücklauf (carriage return)
`\t`	Tabulatorzeichen
`\b`	Backspace, also Zeichen rückwärts
`\f`	Seitenvorschub
`\0`	Echte 0, also nicht die Ziffer `'0'`
`\\`	Ausgabe eines Backslashs
`\"`	Ausgabe eines Anführungszeichens
`\'`	Ausgabe eines Hochkommas
`\0nnn`	Oktalzahl nnn bestimmt das Zeichen.
`\0xnn`	Hexadezimalzahl nn bestimmt das Zeichen.
`\unnnn`	Hexadezimalzahl nnnn bestimmt das Unicode-Zeichen.

Tabelle 1.9 Kontrollzeichen

Zeichenketten

Zeichenketten bestehen aus mehreren, aneinandergehängten Buchstaben. Alles, was zur Zeichenkettenkonstanten gehören soll, wird in Anführungszeichen eingeschlossen. In einer Zeichenkette können alle Zeichen verwendet werden, die auch in einer Buchstabenkonstanten verwendet werden, also auch die Kontrollzeichen. Beispielsweise würde die folgende Zeichenkette zwei Zeilen darstellen:

```
"Dies ist eine Zeichenkette\n aus zwei Zeilen"
```

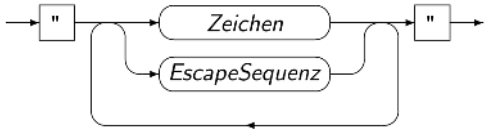

Abbildung 1.14 Syntaxgraph einer Zeichenkettenkonstante

Manchmal sind Texte etwas länger und passen nicht in eine Zeile. Darum ist es möglich, mehrere Zeichenkettenkonstanten hintereinander auf mehrere Zeilen zu verteilen. Der Compiler ist so clever und setzt sie zu einer Zeichenkettenkonstanten zusammen.

```
"Dies ist eine Zeichenkette, die so lang ist, dass sie "
"nur schlecht in eine Zeile passt. Aber das ist kein "
"Problem. Fahren Sie einfach in der Folgezeile fort!"
```

Da Zeichenketten zusammengesetzte Elemente sind, können sie nicht so einfach behandelt werden wie Zahlen oder einzelne Buchstaben. Zunächst werden sie hier im Buch darum nur als Konstanten für die Ausgabe der Programme verwendet. Wie man Variablen für Zeichenketten definiert und wie man sie zuweist und verarbeitet, wird im Abschnitt 3.1.2 beschrieben, wenn es um zusammengesetzte Typen geht.

Symbolische Konstanten

Zahlkonstanten, die ohne weiteren Kontext im Programm stehen, haben keinen Dokumentationswert und führen zu Verwechslungen. Verwenden Sie an einer Stelle im Programm eine 8, so wird sich sofort jeder Leser fragen, was diese 8 eigentlich bedeuten soll. Nehmen wir an, es handele sich um die Regelarbeitszeit pro Tag, so würde es die Lesbarkeit des Programms wesentlich verbessern, wenn Sie statt der 8 das Wort »RegelArbeitsZeit« verwenden. Damit wird sofort deutlich, was die Zahl zu bedeuten hat.

Um das Wort mit der Zahl zu verbinden, deklarieren Sie eine Konstante. Eine Konstante entspricht in C++ einer unveränderbaren Variablen. Sie hat einen Typ und einen Namen wie eine Variable, kann aber nicht geändert werden. Da die Konstante nicht änderbar ist, muss sie bei ihrer Deklaration initialisiert werden. Eine Konstantendeklaration sieht also aus wie eine Variablendefinition mit Initialisierung, der man das Schlüsselwort `const` vorangestellt hat. (Zum Unterschied zwischen den Begriffen »Deklaration« und »Definition« finden Sie Hinweise im Glossar)

```
const int RegelArbeitsZeit = 8;
```

Listing 1.14 Deklaration einer Konstanten

Konstanten benötigen im Gegensatz zu Variablen keinen Speicher, da sie nur einen Namen für einen konstanten Wert einführen, der nach der Kompilierung restlos verschwindet.

Es gibt zwei wichtige Gründe dafür, Konstanten zu deklarieren. Erstens führt die Verwendung von Konstanten zu einer besseren Lesbarkeit des Programms. Zweitens ist bei Änderung des Wertes einer Konstanten nur eine Stelle im Programm zu korrigieren. Würde im obigen Beispiel die Regelarbeitszeit eines Tages nicht mehr acht, sondern sieben Stunden betragen, müssten Sie das Programm nur an einer Stelle ändern. Hätten Sie aber im Programm überall die Zahl stehen, müssten Sie nach jedem Vorkommen der 8 suchen. Dabei müssten Sie zusätzlich aufpassen, ob diese 8 auch wirklich die Regelarbeitszeit ist und nicht vielleicht die maximale Größe einer Arbeiterkolonne oder den Mindesttariflohn der Reinigungskräfte darstellt. Es gibt eine Faustregel, die besagt, dass alle Zahlen außer 0 und 1 mit Namen deklariert werden sollten.

Alte Form per #define

Diese Form der Deklaration von Konstanten gibt es in C erst seit Einführung des ANSI-Standards. Vorher wurden Konstanten über den Präprozessor des Compilers gesetzt, der mit dem Befehl `#define` eine textuelle Ersetzung herbeiführen kann. So würde der folgende Befehl die Regelarbeitszeit von oben definieren.

```
#define RegelArbeitsZeit  8
```

Listing 1.15 Deklaration einer Konstanten

Das führt dazu, dass der Präprozessor im Quelltext, kurz bevor der Compiler ihn übersetzt, überall eine 8 einsetzt, wo der Programmierer das Wort »RegelArbeitsZeit« geschrieben hat. Auf den ersten Blick ist der Effekt derselbe. In zwei wichtigen Aspekten unterscheiden sich die Mechanismen: Erstens legt die C++-Schreibweise auch den Typ der Konstanten fest, und zweitens können die C++-Konstanten auch mit Werten initialisiert werden, die vorher im Programm errechnet wurden.

1.5 Verarbeitung

Es gibt also Variablen und es gibt Konstanten. Im folgenden Abschnitt wird beschrieben, wie die Konstanten in die Variablen kommen, wie Variablen kopiert und miteinander verrechnet werden können. Es kommt also Leben in den Computer.

1.5.1 Zuweisung

Bei der Initialisierung wurde bereits das Gleichheitszeichen verwendet, um eine Variable mit einem Wert vorzubelegen. Das Gleichheitszeichen wird auch dann verwendet, wenn eine Variable einen neuen Wert bekommen soll.

Links vom Gleichheitszeichen steht immer eine Variable als Ziel der Zuweisung. Auf der rechten Seite des Gleichheitszeichens steht die Datenquelle. Das kann eine andere Variable, ein Zahlenwert oder eine Berechnung sein. Man bezeichnet die Datenquelle auf der rechten Seite einer Zuweisung allgemein als Ausdruck. Die englische Bezeichnung dafür ist »expression«. Das folgende Beispiel zeigt mehrere Zuweisungen. Dabei wird auch schon den Rechenoperationen ein wenig vorgegriffen.

```
MwStSatz = 19.0;
Netto = 200.0;
MwStBetrag = Netto * MwStSatz / 100;
Brutto = Netto + MwStBetrag;
```

Listing 1.16 Zuweisungen

L-Value

Steht auf der linken Seite des Gleichheitszeichens etwas anderes als eine Variable, werden die meisten Compiler eine Fehlermeldung bringen, die `L-Value expected` oder ähnlich lautet. Direkt übersetzt heißt die Meldung, dass ein »Linkswert« erwartet wird, also etwas, was auf der linken Seite einer Zuweisung stehen kann. Der L-Value muss etwas sein, dem etwas anderes zugewiesen werden kann. Wie Sie später noch sehen werden, werden nicht alle Variablenkonstrukte als L-Value akzeptiert.

C++ hat die Besonderheit, dass Sie in einer Anweisung mehreren Variablen den gleichen Wert zuweisen können. Sie müssen sich das so vorstellen, dass eine Zuweisung ihren Wert nach links durchreicht. Diese Fähigkeit ermöglicht eine Zeile wie die folgende:

```
a = b = c = d = 5 + 2;
```

Listing 1.17 Kaskadierende Zuweisung

Die Anweisung wird von der Datenquelle her abgearbeitet. 5 + 2 ergibt 7. Diese 7 wird der Variablen `d` zugewiesen. Das Ergebnis der Zuweisung ist eben dieser Wert 7, der dann `c` zugewiesen wird. Von dort geht es weiter zur Variablen `b` und schließlich zur Variablen `a`. Im Ergebnis enthalten alle aufgeführten Variablen den Wert 7.

1.5.2 Rechenkünstler

In Listing 1.16 haben Sie bereits gesehen, wie Sie in einem Programm rechnen können. Letztlich sieht es nicht sehr viel anders aus, als würden Sie eine Rechenaufgabe auf einen Zettel schreiben. Etwas ungewohnt ist lediglich, dass auf der linken Seite das Zuweisungsziel und ein Gleichheitszeichen stehen. Das Multiplikationszeichen ist der Stern und das Divisionszeichen der Schrägstrich. Plus- und Minuszeichen sehen so aus, wie man es erwartet. Sie können sogar das Minuszeichen wie gewohnt als Vorzeichen verwenden.

Eine besondere Rechenart ist die Modulo-Rechnung. Sie liefert den Rest einer ganzzahligen Division. Wenn Sie sich daran erinnern, wie Sie in den ersten Schulklassen dividiert haben, dann fallen Ihnen vielleicht noch Sätze ein wie: »25 geteilt durch 7 sind 3, Rest 4«. Diese Restberechnung gibt es auch unter C++. Man bezeichnet sie als die Modulo-Rechnung. Als Operatorzeichen wird das Prozentzeichen verwendet.

```
Rest = 25 % 7; // Rest ist also 4
```

Auch in C++ werden die Gesetze der Mathematik beachtet. So wird die alte Regel »Punktrechnung geht vor Strichrechnung« auch hier eingehalten. Diese Regel sagt aus, dass die Multiplikation und die Division vor einer Addition oder Subtraktion ausgeführt werden, wenn beide gleichwertig nebeneinanderstehen.

Binäre Operatoren, also Rechensymbole, die zwei Ausdrücke miteinander verknüpfen, sind im Allgemeinen linksbindend. Das bedeutet, dass sie von links nach rechts ausgeführt werden, wenn die Priorität gleich ist. Das heißt, dass `a*b/c` als `(a*b)/c` ausgewertet wird. Eine Ausnahme ist die Zuweisung. Hier wird `a=b=c` als `a=(b=c)` ausgewertet. Der Zuweisungsoperator ist also rechtsbindend. Auch einstellige Operatoren sind rechtsbindend. Dazu gehört auch der Operator `++`, den Sie im Laufe des Abschnitts noch kennenlernen werden.

Unklare Reihenfolge

In ganz besonderen Spezialsituationen ist nicht eindeutig zu klären, in welcher Reihenfolge die einzelnen Operatoren ausgeführt werden. Das folgende Beispiel kann auf verschiedene Weisen interpretiert werden.

```
a = 1;
b = (a*2) + (a=2);
```

Warnung ×

Die Zuweisung `a=2` in der rechten Klammer ergibt als Ergebnis 2, wie zuvor bei den kaskadierenden Zuweisungen schon erwähnt wurde. Aber ob die Zuweisung vor oder nach der linken Klammer ausgeführt wird, bleibt unklar. Die Variable `b` könnte nach dieser Zeile also sowohl 4 als auch 6 enthalten.

Programme entwickeln ohnehin eine erhebliche Komplexität. Darum sollten Sie Mehrdeutigkeiten vermeiden, wo es geht. Wenn Ausdrücke unübersichtlich werden, sollten Sie Klammern benutzen oder die Berechnung sogar in mehrere Zwischenschritte aufteilen. Sie sollten das Ziel verfolgen, Ihr Programm so einfach und übersichtlich wie möglich zu halten. Im englischen Sprachraum gibt es dafür die KISS-Regel: »Keep It Small and Simple« (»Halte es klein und einfach«). Ein Programm, das aus einer Sammlung der raffiniertesten Kniffe besteht, wird unlesbar und ist damit unwartbar und darum unprofessionell.

1.5.3 Abkürzungen

Der Mensch neigt zur Bequemlichkeit. Nicht anders geht es Programmierern. Sie handeln nach dem Grundgedanken, niemals etwas zu tun, was ein Computer für sie tun kann, und so ist es naheliegend, dass es Mittel und Wege gibt, wiederkehrende Aufgaben möglichst kurz zu formulieren. Um den Wert einer Variablen um 1 zu erhöhen, können Sie die folgende Zeile schreiben:

```
zaehler = zaehler + 1;
```

Listing 1.18 Der Inhalt der Variablen `zaehler` erhöht sich um 1.

Das bedeutet, dass sich der neue Wert der Variablen `zaehler` aus dem alten Wert der Variablen `zaehler` plus 1 bildet. Es wird zuerst die rechte Seite des Gleichheitszeichens ausgewertet, bevor sie der Variablen auf der linken Seite zugewiesen wird. Insgesamt bewirkt die Zeile, dass sich der Inhalt der Variablen `zaehler` um 1 erhöht.

Es kommt häufiger vor, dass sich der neue Wert einer Variablen aus ihrem bisherigen Wert ergibt, der mit einem anderen Wert verrechnet wird. Immer wenn ein Wert erhöht, vermindert, verdoppelt oder halbiert wird, kann eine kürzere

Variante verwendet werden. In der folgenden Zeile wird wiederum der Wert der Variablen `zaehler` um 1 erhöht:

```
zaehler += 1;
```

Listing 1.19 Schreibfaules Inkrementieren

Hier wird das Pluszeichen mit dem Gleichheitszeichen kombiniert. Damit das Plus nicht fälschlicherweise als Vorzeichen der 1 interpretiert wird, muss es vor dem Gleichheitszeichen stehen. Zwischen Plus- und Gleichheitszeichen darf kein Leerzeichen stehen. Die Bedeutung der Zeile ist also: »Addiere der Variablen `zaehler` den Wert 1 hinzu.«

Der Gedanke liegt nahe, dass dies nicht nur für das Addieren funktioniert. Sie können es beim Subtrahieren, beim Multiplizieren, bei der Division und der Modulo-Rechnung verwenden.

Kurze Schreibweise	Lange Schreibweise
`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a *= b`	`a = a * b`
`a /= b`	`a = a / b`
`a %= b`	`a = a % b`

Tabelle 1.10 Kurzschreibweisen

In dem besonderen Fall, dass der Wert um 1 erhöht wird, kann der Ausdruck noch weiter verkürzt werden. Dazu werden an die Variable einfach zwei Pluszeichen angehängt.

```
zaehler++;
```

Listing 1.20 Sehr schreibfaules Inkrementieren

Sie werden schon ahnen, dass dieses Doppelplus der Sprache C++ den Namen gegeben hat. C++ entspricht der Sprache C, die um 1 erhöht wurde.

Wie fast nicht anders zu erwarten, gibt es auch ein Doppelminus. Es tut genau das, was Sie schon vermuten: Es zieht von der Variablen den Wert 1 ab. Warum es weder einen Doppelstern noch einen Doppelschrägstrich gibt, werde ich Ihnen nicht verraten. Betrachten Sie es als eines der letzten Geheimnisse unserer Erde.

Sie können das doppelte Plus oder Minus auch auf der rechten Seite einer Zuweisung verwenden. Dann wird nach der Variablenauswertung ihr Wert erhöht respektive herabgesetzt. Betrachten Sie das folgende Beispiel:

```
zaehler = 5;
summe = 2 + zaehler++; // summe enthält 7!
```

Listing 1.21 Inkrementieren auf der rechten Seite

Klar ist, dass die Variable `zaehler` nach diesen Befehlen den Wert 6 enthält. Etwas unklarer ist dagegen, welchen Wert die Variable `summe` hat. Wenn Sie auf 7 tippen, liegen Sie richtig. Wie oben erwähnt, wird das Inkrementieren der Variablen `zaehler` erst nach der Auswertung durchgeführt. Sie können allerdings auch das Doppelplus vor die Variable stellen. Dann wird die Variable erst ausgewertet, nachdem sie inkrementiert worden ist.

```
zaehler = 5;
summe = 2 + ++zaehler; // summe enthält 8!
```

Listing 1.22 Ein anderes Ergebnis

In diesem Fall wird die Variable `summe` den Wert 8 haben. Falls Sie das Ganze etwas unübersichtlich finden, spricht das für Ihren Geschmack. Ich würde eine solche Konstruktion im Programm vermeiden. Schreiben Sie lieber ein paar Zeichen mehr. Dann wird ein Kollege später schneller verstehen, was Sie geschrieben haben. Die folgenden Zeilen bewirken das Gleiche und sind viel einfacher zu lesen:

```
zaehler = 5;
++zaehler;
summe = 2 + zaehler;
```

Listing 1.23 Das Gleiche, leichter lesbar

Das Voranstellen des Inkrementoperators nennt man Präfix. Es bewirkt, dass die Variable zuerst inkrementiert und dann ausgewertet wird. Das Nachstellen des Operators heißt Postfix. Die Variable wird zuerst ausgewertet und dann erst inkrementiert. Übrigens ist die Präfixvariante ein klein wenig effizienter zu implementieren. In Tabelle 1.11 finden Sie eine Übersicht über die mathematischen Operatoren in aufsteigender Priorität.

Operator	Bedeutung	Beispiel
+	Addition	`a = 11 + 5; (16)`
-	Subtraktion	`a = 11 - 5; (6)`
*	Multiplikation	`a = 11 * 5; (55)`
/	Division	`a = 11 / 5; (2)`
%	Modulo	`a = 11 % 5 (1)`
++	Inkrementieren	`++a; oder a++;`
--	Dekrementieren	`--a; oder a--;`

Tabelle 1.11 Mathematische Operatoren

1.5.4 Funktionen am Beispiel der Zufallsfunktion

Neben den Grundoperationen bietet Ihnen C++ auch eine Reihe von Funktionen an, auf die später in den entsprechenden Abschnitten detailliert eingegangen wird. Eine Funktion unterscheidet sich rein optisch von einer Variablen durch ein Klammernpaar, das dem Funktionsnamen angehängt ist. Ein Funktionsaufruf führt dazu, dass das Programm den aktuellen Ablauf unterbricht und zunächst den Code der Funktion durchläuft und dann zurückkehrt. In den meisten Fällen liefert die Funktion dabei einen Ergebniswert zurück. Bei einigen Funktionen können Sie in den Klammern Parameter an die Funktion übergeben. So würde der Aufruf der Sinusfunktion beispielsweise so aussehen:

```
f = sin(45);
```

Der Name der Funktion lautet `sin`, als Parameter wird 45 übergeben, und das Ergebnis wird in der Variablen `f` abgelegt.

Als erstes Beispiel für mathematische Funktionen betrachten wir die Zufallsfunktion. Für einige Beispielprogramme werden sich Zufallszahlen als nützlich erweisen. Auch in der Praxis leisten sie gute Dienste. Sie können damit Daten erzeugen, um Programmteile zu testen.

Eine vom Computer erzeugte Zufallszahl ist nicht wirklich zufällig, sondern wird durch eine Funktion generiert. Gute Zufallszahlen haben zwei Eigenschaften: Sie sind möglichst schwer vorhersehbar und gleichmäßig verteilt. Immerhin werden sie nicht nur zum Würfeln oder für Kartenspiele gebraucht, sondern auch für Simulationen, die Millionen Mal durchlaufen werden.

Damit Versuchsreihen wiederholbar sind, gibt es eine Startfunktion. Sie erhält einen Startwert als Parameter. Wenn der gleiche Startwert verwendet wird, wird anschließend immer die gleiche Folge von Zufallszahlen generiert.

Mit dem Aufruf der Funktion `srand()` wird der Zufallszahlengenerator initialisiert. Die Funktion hat als Parameter eine ganze Zahl, die als Startwert dient.

Nachdem mit der Funktion `srand()` der Zufallszahlengenerator einmal initialisiert wurde, kann durch den Aufruf von `rand()` beliebig oft ein quasi zufälliger Rückgabewert vom Typ `long` abgerufen werden. Bei jedem Neuaufruf liefert die Funktion einen neuen Zufallswert. Beide Funktionen stammen aus der Standardbibliothek `cstdlib`.

Das folgende kleine Programm startet einmal die Zufallszahlen mit irgendeinem x-beliebigen Wert. Dann wird zweimal die Funktion `rand()` aufgerufen und der Rückgabewert der Variablen `zufall` zugewiesen.

```
// Programm zur Demonstration der Zufallsfunktion
#include <cstdlib>
using namespace std;

int main()
{
    const int IrgendEinStartWert=9;
    long zufall;     // Hier wird das Ergebnis stehen.
    srand(IrgendEinStartWert);  // Würfel schütteln
    zufall = rand();            // Einmal werfen
    zufall = rand();            // Noch mal werfen
}
```

Listing 1.24 Zufallszahlen

Der Rückgabewert der Funktion `rand()` ist, wie bereits erwähnt, eine positive, ganze Zahl, die nach jedem Aufruf anders lauten kann. Das Ergebnis ist minimal 0. Der größtmögliche Rückgabewert ist die durch den Compiler festgelegte Konstante RAND_MAX. In der Praxis ist RAND_MAX gleich LONG_MAX, also meist etwa 2 Millionen.

Damit können die meisten Programme aber wenig anfangen. Typischerweise wollen die Programme einen Würfel, eine Lottozahl oder eine Spielkarte simulieren. Um die großen Zahlen auf die Werte 6, 49 oder 52 herunterzubrechen, gibt es eine einfache Methode: Sie verwenden die Modulo-Rechnung. Wollen Sie einen Würfel simulieren, so berechnen Sie Modulo 6 und erhalten einen Wert zwischen 0

und 5. Nun müssen Sie nur noch eine 1 addieren, und Sie erhalten die gewohnten Augenzahlen zwischen 1 und 6.

```
augen = rand() % 6 + 1;
```

Falls Sie für ein Spiel einen wirklich nicht vorhersehbaren Startwert brauchen, empfehle ich Ihnen die Zeitfunktionen (siehe Abschnitt 9.2). Die Sekunden seit dem 1.1.1970 sind prima als Startwert geeignet. Und falls Ihnen das als noch zu kalkulierbar erscheint, verwenden Sie doch die Millisekunden modulo 1000, die in dem Zeitraum vergangen sind, die der Benutzer für seine Eingabe benötigte.

1.5.5 Typumwandlung

Hin und wieder ist es notwendig, Werte eines Typs in eine Variable eines anderen Typs zu speichern. So ergibt das Dividieren zweier ganzer Zahlen eine ganze Zahl. Wird 3 durch 4 geteilt, ergibt sich also 0 (mit dem Rest 3, der hier verloren geht). Wollen Sie stattdessen aber als Ergebnis 0,75 haben, müssen Sie einen der beiden Operanden zum Fließkommawert wandeln, damit C++ auch die Fließkommadivision verwendet. Ein solches Umwandeln des Typs nennt man *Casting*[3].

Um einen Ausdruck eines bestimmten Typs in einen anderen umzuwandeln, gibt es zwei Schreibweisen. Die eine Schreibweise ist ein Erbstück der Sprache C. Dort wurde dem Ausdruck der Zieltyp in Klammern vorangestellt. In C++ wurde die Schreibweise eingeführt, dass auf den Namen des Typs eine Klammer folgt, in der der zu konvertierende Ausdruck steht.

```
int Wert;
Wert = (int)IrgendWas; // klassisches C-Casting
Wert = int(IrgendWas); // C++
```

Einige Umwandlungen führt C++ direkt durch, ohne darüber zu reden. Das geschieht immer dann, wenn die Umwandlung ohne jeden Informationsverlust des Inhalts gewährleistet ist. So wird eine `short`-Variable oder -Konstante direkt einer `long`-Variablen zugewiesen. Hier gibt es keine Interpretationsprobleme, und es kann jeder beliebige `short`-Wert in einer `long`-Variablen abgelegt werden. Der umgekehrte Weg ist schwieriger. Die Zahl 200.000 passt nicht in eine `short`-Variable, wenn diese nur aus zwei Bytes besteht. Hier wird der Compiler im Allgemeinen eine Warnung absetzen, dass relevante Informationen verloren gehen könnten.

3 Der Name leitet sich von den gleichnamigen Shows her, in denen Menschen in andere Typen umgebrochen werden.

Besonders tückisch kann es sein, wenn der Compiler statt Fließkommazahlen ganzzahlige Werte verwendet. Als Beispiel soll ein klassischer Dreisatz verwendet werden. Drei Tomaten kosten 4 Euro. Wie viel kosten fünf Tomaten? Im Programm würde das wie folgt umgesetzt:

```
float SollPreis = (4/3)*5;
```

Der Inhalt der Variablen `SollPreis` dürfte überraschen: Er ist 5. Der Grund ist, dass der Compiler den Ausdruck `4/3` als Integer-Berechnung ausführt und die Nachkommastellen abschneidet. Also ist das Ergebnis der Division 1. Multipliziert mit 5 ergibt sich das oben genannte Ergebnis. Dennoch würden Sie eher erwarten, dass Sie an der Kasse 6,67 Euro zahlen müssen, und der Kaufmann wird sich Ihrer Ansicht gewiss anschließen. In solchen Fällen können Sie mit einer Typumwandlung eingreifen. Es muss mindestens ein Operand der Division zum `float`-Wert konvertiert werden, um eine `float`-Berechnung zu erzwingen.

```
float SollPreis = (float(4)/3)*5;
```

Nach dieser Anpassung ergibt die Berechnung die erwarteten 6.66667.

Am Rande sei erwähnt, dass die Klammern um `4/3` aus Sicht von C++ zwar überflüssig sind, weil der Ausdruck nur Punktrechnung enthält und darum von links nach rechts ausgeführt wird. Dafür haben diese Klammern aber einen nicht unerheblichen Dokumentationswert. Jemand, der später einen Fehler in dem Programm sucht, sieht sofort, wie der Autor die Prioritäten setzen wollte. Wenn also irgendwo Zweifel über die Prioritäten bei Ausdrücken entstehen könnten, ist es besser, ein paar Klammern zu viel als zu wenig zu setzen. Das Programm wird dadurch nicht langsamer. Der Compiler wird die Klammern sowieso wegoptimieren.

1.6 Ein- und Ausgabe

Es wird Sie gewiss mit Freude erfüllen, dass der Computer so wunderbar rechnen kann. Aber selbst wenn Neugierde nicht zu Ihren Untugenden gehört, werden Sie irgendwann einmal wissen wollen, was der Computer denn herausgefunden hat. Eine Ausgabefunktion muss her!

1.6.1 Ausgabestrom nach cout

C++ verwendet für die Ein- und Ausgabe das Datenstrommodell. Die Variablen werden quasi auf ein Ausgabeobjekt umgeleitet. Dieses Ausgabeobjekt heißt `cout`.

Der Datenstrom zur Ein- und Ausgabe wird in einer eigenen Bibliothek behandelt. Vor der ersten Verwendung müssen die folgenden Zeilen im Programm stehen:

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

In der ersten Zeile wird mit dem Kommando `#include` die Datei `iostream` in das Programm eingebunden. Der Dateiname ist hier in spitzen Klammern eingeschlossen. Das bedeutet für den Compiler, dass die Datei nicht im gleichen Verzeichnis liegt wie das zu übersetzende Programm, sondern in den voreingestellten Pfaden des Compilers.

Die zweite Zeile besagt, dass der Namensraum `std` verwendet wird. Namensräume ermöglichen es, gleiche Namen gegeneinander abzugrenzen. Damit die Namen der Standardbibliotheken auch in den eigenen Programmen oder in anderen Bibliotheken verwendet werden können, wurden sie in den Namensraum `std` gelegt. Sie können so bei jedem einzelnen Namen festlegen, ob Sie ihn aus `std` oder einem anderen Namensraum verwenden wollen. Da eine Kollision aber eher selten auftritt, kann durch den `using namespace`-Befehl erreicht werden, dass auf alle Elemente des Namensraums `std` direkt zugegriffen werden kann. Namensräume werden im Abschnitt 7.2 behandelt.

Alte Programme binden manchmal noch die Datei *iostream.h* ein. Diese verwendet nicht den Namensraum `std`. Sie sollten solche Programme umstellen, weil die aktuelleren Compiler dies nicht mehr unterstützen.

Um Daten anzuzeigen, werden sie auf die Datenausgabe `cout` geleitet. Das »out« in `cout` ist englisch und bedeutet »aus«. Es ist also das Ausgabeobjekt, auf das umgeleitet wird. Das »c« steht davor, weil es der Lieblingsbuchstabe von Bjarne Stroustrup ist. Darum hat er seine Sprache ja auch C++ genannt. Zunächst wird das Datenziel genannt. Dann werden zwei Kleiner-Zeichen verwendet, um die Daten auf die Ausgabe zu lenken.

```
cout << meinZahlenWert;
cout << endl;
```

Listing 1.25 Ausgabe einer Variablen

Im Beispiel wird der Inhalt der Variablen `meinZahlenWert` ausgegeben. In der Zeile darunter wird `endl` auf den Datenstrom gesendet. Dies ist keine selbst definierte Variable, sondern eine vordefinierte Konstante für das Zeilenende. Die Verwendung von `endl` sorgt dafür, dass eine neue Zeile angefangen wird und dass alle anzuzeigenden Daten sofort auf dem Bildschirm erscheinen.

Der Übersicht halber oder um sich Tipparbeit zu sparen, können mehrere Ausgabeobjekte direkt hintereinander, durch die doppelten Kleiner-Zeichen getrennt, aufgeführt werden.

```
cout << meinZahlenWert << endl;
```

Listing 1.26 Ausgabekette

Sie können nicht nur Variablen, sondern auch Konstanten auf diesem Weg ausgeben. Das ist besonders interessant bei Zeichenkettenkonstanten, mit denen Sie Ihren Programmausgaben ein paar erläuternde Texte beifügen können.

```
cout << "Ergebnis: " << meinZahlenWert << endl;
```

Listing 1.27 Ausgabekette

Dieses gekonnte Beispiel trockener Informatiker-Poesie sagt dem Anwender, dass der Wert, der jetzt auf dem Bildschirm erscheint, das Ergebnis des Programms ist. Solche Informationen sind es, die ein benutzerfreundliches Programm ausmachen.

Für Fehlerausgaben ist das Objekt `cerr` vorgesehen. Normalerweise gelangt die Ausgabe genau wie bei `cout` auf den Bildschirm. Wenn der Benutzer aber die Standardausgabe eines Programms umleitet, gehen die Fehlermeldungen nicht in den Ausgabestrom, sondern erscheinen weiterhin auf dem Bildschirm. Entsprechend sollten Sie die weiterverwertbaren Ausgaben Ihres Programms an `cout` leiten und Fehlermeldungen des Programs an `cerr`.

1.6.2 Formatierte Ausgabe

Wenn Sie mehrere Zahlenwerte nacheinander an `cout` umleiten, so werden Sie feststellen, dass sie ohne Trennzeichen direkt nebeneinander dargestellt werden. Die Ausgabe eines Wertes 8 und die darauf folgende Ausgabe von 16 erschiene also als 816. Entsprechend sollten Sie zumindest ein paar Leerzeichen oder ein paar erläuternde Worte dazwischen setzen:

```
cout << "Eingabe war: " << Input
     << " Das Doppelte ist: " << Output << endl;
```

Aufwendiger wird es, wenn Sie Tabellen anzeigen wollen. Um zu wissen, wie viele Leerzeichen Abstand zwischen die Zahlen gesetzt werden muss, müssten Sie zunächst ermitteln, wie viele Stellen jede Zahl hat. Um dies zu vereinfachen, gibt es einen sogenannten Manipulator namens `setw()`. (`setw` steht für »set width«, übersetzt »setze Breite«. Das w hat also nichts mit meinem Lieblingsbuchstaben zu tun.) Dieser wird vor der eigentlichen Ausgabe an `cout` gesendet und bereitet

die Formatierung der folgenden Ausgabe vor. Zwischen die Klammern von `setw()` schreiben Sie die Anzahl der Stellen, die für die nachfolgende Ausgabe reserviert werden sollen. Alle Stellen, die nicht von der Zahl selbst belegt werden, werden mit Leerzeichen so aufgefüllt, dass die Zahl rechtsbündig erscheint.

Bevor Sie allerdings mit Manipulatoren arbeiten können, müssen Sie die Datei *iomanip* durch eine `include`-Anweisung einbinden. Die folgenden Zeilen sollten also gleich zu Anfang Ihres Programms stehen:

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

Das folgende Beispiel sollte demnach dafür sorgen, dass die Zahlen der beiden Ausgabezeilen hübsch rechtsbündig nebeneinander stehen.

```
cout << setw(7) << 3233 << setw(6) << 128 << endl;
cout << setw(7) << 3 << setw(6) << 1 << endl;
```

Als Ergebnis dieser Anweisungen wird auf dem Bildschirm Folgendes erscheinen:

```
   3233   128
      3     1
```

Weitere Informationen zu Manipulatoren finden Sie im Abschnitt 8.2.2.

1.6.3 Eingabestrom aus cin

Um Eingaben von der Tastatur zu lesen, wird die Datenquelle `cin` auf die Variable umgeleitet. (Das »in« in `cin` steht für »ein«. Es ist also das Eingabeobjekt. Die Herkunft des c wurde ja bereits bei `cout` erläutert.) Der Eingabeoperator besteht genau spiegelverkehrt zum Ausgabeoperator aus zwei Größer-Zeichen. Sie weisen quasi von `cin` auf die Variable, in der die Eingabe abgelegt werden soll.

```
// Demonstration von Ein- und Ausgabe
#include <iostream>
using namespace std;

int main()
{
    int Zahleingabe;
    int Doppel;

    cout << "Bitte geben Sie eine Zahl ein!" << endl;
```

```
    cin >> Zahleingabe;
    Doppel = Zahleingabe * 2;
    cout << "Das Doppelte dieser Zahl ist "
        << Doppel << "." << endl;
}
```

Listing 1.28 Ein einfacher Benutzerdialog (*reinraus.cpp*)

1.7 Übungen

1 Schreiben Sie einen Algorithmus für das Kochen von Kaffee. Bitten Sie einen Freund, diesen Anweisungen zu folgen, die Sie durch die geschlossene Tür geben. Welche Anweisungen führten zu Fehlern? Welche Anweisungen waren fehlinterpretierbar? Würden Sie den Algorithmus anders schreiben, wenn der Freund nicht wüsste, was Kaffee ist und wofür man ihn braucht (den Kaffee, nicht den Freund)?

2 Warum ist ein Programm, das mit einem Interpreter übersetzt wird, typischerweise langsamer als eines, das vom Compiler übersetzt wurde?

3 Schreiben Sie Listing 1.28 ab. Übersetzen und starten Sie es. Im Abschnitt 1.2 ist beschrieben, wie Sie Ihren Compiler einrichten. In Kapitel 6 finden Sie weitere Hilfe für Ihre ersten Schritte mit Compilern. Geben Sie unterschiedliche Werte ein.

4 Ändern Sie das Programm, indem Sie Teile löschen. Überlegen Sie, was Ihre Änderung bewirken wird. Falls Sie eine Fehlermeldung des Compilers erhalten, lesen Sie die Meldung. Versuchen Sie, den Zusammenhang zwischen Ihrer Änderung und der Compiler-Meldung herzustellen.

5 Schreiben Sie ein Programm, das die beiden Wörter »Hallo Benutzer« auf dem Bildschirm ausgibt.

6 Schreiben Sie ein Programm *mwst.cpp*, das nach der Eingabe eines Nettopreises die Mehrwertsteuer berechnet und ausgibt. Seien Sie einmal Finanzminister, und legen Sie den Satz der Mehrwertsteuer für Ihr Programm selbst fest.

7 Ergänzen Sie das Programm *mwst.cpp* dahingehend, dass auch der Bruttopreis ausgegeben wird.

Lösungen zu nicht-selbsterklärenden Aufgaben finden Sie in Anhang B.