Ulrich Kaiser
Martin Guddat

C/C++

Das umfassende Lehrbuch

- Von den Grundlagen zur professionellen Programmierung
- Von einfachen Beispielen zu anspruchsvollen Algorithmen
- Für Ausbildung und Studium geeignet

5., aktualisierte und überarbeitete Auflage

Galileo Computing

Liebe Leserin, lieber Leser,

wir freuen uns, Ihnen die fünfte Auflage dieses Lehrwerkes zu C und C++ vorzustellen. Entstanden aus einem Kurs von Prof. Dr. Kaiser über Grundlagen der Informatik und Programmiersprachen, ist dieses Buch seinem Anliegen immer treu geblieben: Es lehrt Programmieren auf professionellem Niveau, ohne konkrete Kenntnisse vorauszusetzen. Die Syntax der Sprachen ist dabei ein notwendiges Hilfsmittel, das Programmieren selbst darf vielleicht als Kunst verstanden werden; in jedem Fall aber als eine Praxis, für die Talent, Neugierde und ein Verständnis der Grundlagen der Informatik von Bedeutung sind. Letzteres erarbeiten Sie sich mit diesem Buch, das Theorie und Praxis lebendig verbindet.

Es gibt dabei keine Vorgriffe auf den Stoff späterer Kapitel, so dass sich Anfänger problemlos von den Grundbegriffen zu den fortgeschrittenen Themen vorarbeiten können. Sie können die nötige Theorie nicht nur leicht nachvollziehen, sondern lernen ihren Nutzen auch im großen Zusammenhang kennen.

Alles wird anhand anschaulicher Beispiele erläutert – wo es um die Genauigkeit einer mathematischen Abschätzung geht, denken Sie etwa an den prüfenden Blick ins Portemonnaie, ob Ihr Bargeld für ein Brötchen reicht. Im Laufe der Zeit konnten viele Leserwünsche und Lehrerfahrungen einfließen – so haben die Behandlung der Standardbibliotheken, der Abbau bestimmter Hürden bei mathematischen Inhalten und die ausführlichen, vollständigen Musterlösungen das Buch verbessert. Neu in dieser Auflage: Falls Sie einmal nicht weiterkommen, schauen Sie erst nach Lösungshinweisen, bevor Sie sich die vollständige Lösung ansehen. Die Codebeispiele und Lösungen finden Sie außerdem zum Download bei den Materialien zum Buch unter *http://www.galileo-press.de/3536*.

Dieses Buch wurde mit großer Sorgfalt geschrieben, geprüft und produziert. Sollten Sie dennoch etwas nicht so vorfinden, wie Sie es erwarten, so zögern Sie nicht, mit uns Kontakt aufzunehmen. Ihre Anmerkungen, Ihr Lob oder Ihre konstruktive Kritik sind mir herzlich willkommen!

Ihre Almut Poll
Lektorat Galileo Computing

almut.poll@galileo-press.de
www.galileocomputing.de
Galileo Press · Rheinwerkallee 4 · 53227 Bonn

Auf einen Blick

Impressum

Dieses E-Book ist ein Verlagsprodukt, an dem viele mitgewirkt haben, insbesondere:

Lektorat Almut Poll, Erik Lipperts
Korrektorat Friederike Daenecke
Herstellung E-Book Martin Pätzold
Covergestaltung Janina Conrady
Satz E-Book Typographie & Computer, Krefeld

Bibliografische Information der Deutschen Nationalbibliothek:
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

ISBN 978-3-8362-8912-2

5., aktualisierte und überarbeitete Auflage 2014

Inhalt

23 Zusammenfassung und Überblick 879

24 Die C++-Standardbibliothek und Ergänzung 953

Vorwort

Als die erste Auflage dieses Buches erschien, war Roman Herzog Präsident der Bundesrepublik Deutschland. Auf Herzog folgten Rau, Köhler, Wulff und Gauck und jeweils eine neue Auflage dieses Buches. Jetzt liegt die fünfte, vollständig überarbeitete Auflage vor. Der Leitgedanke des Buches ist aber über all die Jahre gleich geblieben. Dazu möchte ich aus dem Vorwort der ersten Auflage zitieren:

> Ziel des Buches ist es, Leser ohne Vorkenntnisse auf ein professionelles Niveau der C- und C++-Programmierung zu führen. Unter »Programmierung« wird dabei weitaus mehr verstanden als die Beherrschung einer Programmiersprache. So wie »Schreiben« mehr ist, als Wörter unter Beachtung der Regeln von Rechtschreibung, Zeichensetzung und Grammatik zu Sätzen zusammenzufügen, ist Programmieren mehr als die Erstellung formal korrekter Programme. Zum Programmieren gehört ein Überblick über die Grundlagen und die Anwendungen der Programmierung. Der Leitgedanke dieses Buches ist es, wichtige Grundlagen und Konzepte der Informatik darzustellen und unmittelbar mit der Programmierung zu verknüpfen. Die Grundlagen liefern dann die Ideen zur Programmierung, und die Programmierung liefert die Motivation für die Beschäftigung mit den Grundlagen.

Dem ist auch heute nichts hinzuzufügen.

Es freut mich, dass ich mit Martin Guddat einen Kollegen gefunden habe, der die Arbeit am Buch für die Amtsperioden der nächsten fünf Bundespräsidenten fortsetzen wird. Dazu wünsche ich ihm viel Erfolg.

Bocholt, im September 2014
Ulrich Kaiser

Ich verwende das Buch des Kollegen Kaiser seit mehreren Jahren in meinen eigenen Vorlesungen und empfehle es immer wieder gerne als umfassendes und konsistentes Werk, das eine breite Basis für die Programmierung legt. Umso mehr freut es mich, dass ich die Gelegenheit bekomme, das Buch in den kommenden Jahren weiterzuführen, zu pflegen und an neue Entwicklungen anzupassen.

Dafür möchte ich Ulrich Kaiser meinen besonderen Dank aussprechen und hoffe, dass ich seiner riesigen Vorarbeit gerecht werde.

Für die Durchsicht des gesamten Manuskripts, für seine Anmerkungen und seine Änderungsvorschläge danken wir beide besonders Herrn Daniel Hacirisoglu!

Bocholt, im September 2014
Martin Guddat

Kapitel 1
Einige Grundbegriffe

Computer Science is no more about computers than astronomy is about telescopes.
– Edsger W. Dijkstra

Womit beschäftigen wir uns in diesem Buch? Mit Informatik? Mit Programmierung? Mit der Programmiersprache C/C++? Mit Computern? Alles scheint miteinander verwoben. Und dann sagt auch noch einer der bedeutendsten Informatiker und Pioniere der Programmierung:

> Computerwissenschaft hat mit Computern genauso viel zu tun wie Astronomie mit Teleskopen.

Sie sind vielleicht über das Interesse an Technik zu Computern und über das Interesse an Computern zu Programmiersprachen gekommen. Wir möchten mit Ihnen diesen Weg weitergehen und Sie über das Interesse an Programmiersprachen zur Programmierung und über das Interesse an der Programmierung zur Informatik führen. Ein weiter Weg, der merkwürdigerweise mit einem Kochrezept beginnt.

Im Internet habe ich das folgende Rezept zur Herstellung eines Pfannkuchens gefunden:

Zutaten:

50 g	Butter oder Magarine
100 g	Zucker
1 Pck.	Vanillezucker
4	Eier
200 ml	Milch
200 g	Mehl
1 TL	Backpulver
etwas	Butter zum Ausbacken

Zubereitung:

Butter mit Zucker und Vanillezucker vermengen (dazu evtl. in der Mikrowelle weich werden lassen). Die Eigelbe hinzufügen und schaumig rühren, dann die Milch zugeben und unterrühren. Mehl mit Backpulver über die Masse sieben und glatt rühren. Eiweiße steif schlagen und zum Schluss unterheben.

Eine Pfanne bei mittlerer Hitze heiß werden lassen. Portionsweise aus dem Teig nun Pfannkuchen in wenig Butter von beiden Seiten braten, bis sie goldgelb sind.

Abbildung 1.1 Ein Pfannkuchenrezept

Das Rezept gliedert sich in zwei Teile. Im ersten Teil werden die erforderlichen *Zutaten* genannt, und im zweiten Teil wird die *Zubereitung* beschrieben. Die beiden Teile sind wesentlich verschieden und gehören doch untrennbar zusammen. Ohne Zutaten ist die Zubereitung nicht möglich, und ohne Zubereitung bleiben die Zutaten ungenießbar. Außerdem sehen Sie, dass sich der Autor bei der Formulierung des Rezepts einer bestimmten Fachsprache (schaumig rühren, steif schlagen, unterheben) bedient. Ohne diese Fachsprache wäre die Anleitung wahrscheinlich weitschweifiger, umständlicher und vielleicht sogar missverständlich. Die Verwendung einer Fachsprache setzt allerdings voraus, dass sich Autor und Leser des Rezepts zuvor (ausgesprochen oder unausgesprochen) auf eine gemeinsame Terminologie verständigt haben.

Wir übertragen dieses Beispiel in unsere Welt – die Welt der Datenverarbeitung:

- Die Zutaten für das Rezept sind die Daten bzw. *Datenstrukturen*, die wir verarbeiten wollen.
- Die Zubereitungsvorschrift ist ein *Algorithmus*[1], der festlegt, wie die Daten verarbeitet werden sollen.
- Das Rezept insgesamt ist ein *Programm*, das alle Datenstrukturen (Zutaten) und Algorithmen (Zubereitungsvorschriften) zum Lösen der gestellten Aufgabe enthält.
- Die gemeinsame Terminologie, in der sich Autor und Leser des Rezepts verständigen, ist die *Programmiersprache*, in der das Programm geschrieben ist. Die Programmiersprache muss dabei alle im Hinblick auf die Zutaten und die Zubereitung bedeutsamen Informationen zweifelsfrei zu übermitteln.
- Die Küche ist die technische Infrastruktur zur Umsetzung von Rezepten in schmackhafte Gerichte und ist vergleichbar mit einem *Computer*, seinem *Betriebssystem* und den benötigten *Entwicklungswerkzeugen*.
- Der Koch übersetzt das Rezept in einzelne Arbeitsschritte in der Küche. Üblicherweise geht ein Koch in zwei Schritten vor. Im ersten Schritt bereitet er die Zutaten einzeln und unabhängig voneinander vor (z. B. Kartoffeln kochen), um die Einzelteile dann in einem zweiten Schritt zusammenzufügen und abzuschmecken. In der Datenverarbeitung sprechen wir in diesem Zusammenhang von *Compiler* und *Linker*.
- Das fertige Gericht ist das *lauffähige Programm*, das vom Anwender (Esser) genutzt (verzehrt) werden kann.

Nur, welche Rolle spielen *wir* in diesem Szenario? Sollte für uns kein Platz vorgesehen sein? Nein, wir suchen uns die interessanteste Aufgabe aus:

1 Dieser Begriff geht zurück auf Abu Jafar Muhammad Ibn Musa Al-Khwarizmi, der als Bibliothekar des Kalifen von Bagdad um 825 ein Rechenbuch verfasste und dessen Name in der lateinischen Übersetzung von 1200 als »Algorithmus« angegeben wurde.

- Wir sind Autoren, die sich neue, schmackhafte Gerichte für unterschiedliche Anlässe ausdenken und Rezepte bzw. Kochbücher mit den besten Kreationen veröffentlichen.

Was müssen wir lernen, um unsere Rolle ausfüllen zu können?

- Wir müssen die Sprache beherrschen, in der Rezepte formuliert werden.
- Wir müssen einen Überblick über die üblicherweise verwendeten Zutaten, deren Eigenschaften und Zubereitungsmöglichkeiten haben.
- Wir müssen einen Vorrat an Zubereitungsverfahren bzw. kompletten Rezepten abrufbereit im Kopf haben.
- Wir müssen wissen, welche Zutaten oder Verfahren miteinander harmonieren und welche nicht.
- Wir müssen wissen, was in einer Küche üblicherweise an Hilfsmitteln vorhanden ist und wie bzw. wozu diese Hilfsmittel verwendet werden.
- Bei anspruchsvolleren Gerichten müssen wir wissen, in welcher Reihenfolge und mit welchem Timing die Einzelteile zuzubereiten sind und wie die einzelnen Aufgaben verteilt werden müssen, damit alles zeitgleich serviert werden kann.
- Wir müssen auch wissen, worauf ein potenzieller, späterer Esser Wert legt und worauf nicht. Dies ist besonders wichtig, wenn wir Rezepte für einen ganz besonderen Anlass erstellen.

Letztlich möchten wir komplette Festmenüs und deren Speisefolge komponieren und benötigen dazu eine Mischung aus Phantasie, Kreativität, logischer Strenge, Ausdauer und Fleiß, wie sie auch ein guter Koch, Komponist oder Architekt benötigt.

Zurück zu den Grundbegriffen der Informatik. Wir haben informell folgende Begriffe eingeführt:

- *Datenstruktur*
- *Algorithmus*
- *Programm*

Dabei haben Sie bereits erkannt, dass diese Begriffe untrennbar zusammengehören und eigentlich nur unterschiedliche Facetten ein und desselben Themenkomplexes sind.

- Algorithmen arbeiten auf Datenstrukturen. Algorithmen ohne Datenstrukturen sind leere Formalismen.
- Datenstrukturen benötigen Algorithmen, die auf ihnen operieren und sie damit zum »Leben« erwecken.
- Programme realisieren Datenstrukturen und Algorithmen. Datenstrukturen und Algorithmen sind zwar ohne Programme denkbar, aber viele Datenstrukturen

und Algorithmen wären ohne Programmierung allenfalls von akademischem Interesse.

In einem ersten Wurf versuchen wir, die Begriffe *Algorithmus*, *Datenstruktur* und *Programm* einigermaßen exakt zu erfassen.

1.1 Algorithmus

Um unsere noch sehr vage Vorstellung von einem Algorithmus zu präzisieren, starten wir mit einer Definition:

Was ist ein Algorithmus?

Ein *Algorithmus* ist eine endliche Menge genau beschriebener Anweisungen, die unter Verwendung vorgegebener Anfangsdaten in einer genau festgelegten Reihenfolge ausgeführt werden müssen, um die Lösung eines Problems in endlich vielen Schritten zu ermitteln.

Bei dem Begriff »Algorithmus« denkt man heute sofort an »Programmierung«. Das war nicht immer so. In der Tat gab es Algorithmen schon lange, bevor man auch nur entfernt an Programmierung dachte. Bereits im antiken Griechenland wurden Algorithmen zur Lösung mathematischer Probleme formuliert, so z. B. der euklidische Algorithmus zur Bestimmung des größten gemeinsamen Teilers zweier Zahlen oder das sogenannte Sieb des Eratosthenes zur Bestimmung aller Primzahlen unterhalb einer vorgegebenen Schranke.[2]

Sie kennen den Algorithmus zur schrittweisen Berechnung des Quotienten zweier Zahlen. Um etwa 84 durch 16 zu dividieren, gehen Sie nach einem Schema vor, das Sie bereits in der Schule gelernt haben:

```
84 ÷ 16 = 5.25
 80
 --
  40
  32
  --
   80
   80
   --
    0
```

Abbildung 1.2 Die schriftliche Division

2 Euklid von Alexandria (um 300 v. Chr.) und Eratosthenes von Kyrene (um 200 v. Chr.)

Dieses Schema ist aber keine ausreichend präzise Verfahrensbeschreibung. Das Verfahren sollte so beschrieben werden, dass es jemand quasi mechanisch ohne fremde Hilfe anwenden kann. Sie erinnern sich noch an die Definition von *Algorithmus*. Dort hatten wir im Zusammenhang mit einem Algorithmus die Begriffe *Problem*, *Anfangsdaten* und *Anweisungen* verwendet. Als Erstes müssen Sie das Problem und die Anfangsdaten identifizieren. Danach können Sie sich Gedanken über Anweisungen zur Lösung des Problems machen:

Problem:

Berechne den Quotienten zweier natürlicher Zahlen!

Anfangsdaten:

z = Zähler $(z \geq 0)$

n = Nenner $(n > 0)$

a = Anzahl der zu berechnenden Nachkommastellen[3]

Anweisungen:

1. Bestimme die größte ganze Zahl x mit $nx \leq z$! Dies ist der Vorkomma-Anteil der gesuchten Zahl.
2. Zur Bestimmung der Nachkommastellen fahre wie folgt fort:
   - 2.1 Sind noch Nachkommastellen zu berechnen (d. h. $a > 0$)? Wenn nein, dann beende das Verfahren!
   - 2.2 Setze $z = 10(z - nx)$!
   - 2.3 Ist $z = 0$, beende das Verfahren!
   - 2.4 Bestimme die größte ganze Zahl x mit $nx \leq z$! Dies ist die nächste Ziffer.
   - 2.5 Jetzt ist eine Ziffer weniger zu bestimmen. Vermindere also den Wert von a um 1, und fahre anschließend bei 2.1 fort!

Führen Sie diese Anweisungen an dem Beispiel $z = 84$, $n = 16$ und $a = 5$ Schritt für Schritt durch, und Sie werden sehen, dass sich das Ergebnis 5.25 ergibt.

Die einzelnen Anweisungen und ihre Abfolge können Sie sich durch ein sogenanntes *Flussdiagramm* veranschaulichen. In einem solchen Diagramm werden alle beim Ablauf des Algorithmus möglicherweise vorkommenden Wege unter Verwendung bestimmter Symbole grafisch beschrieben. Die dabei zulässigen Symbole sind in einer Norm (DIN 66001) festgelegt. Von den zahlreichen in dieser Norm festgelegten

3 Zunächst ist a die Anzahl der zu berechnenden Nachkommastellen. Im Verfahren verwenden wir a als die Anzahl der **noch** zu berechnenden Nachkommastellen. Wir werden den Wert von a in jedem Verfahrensschritt herunterzählen, bis $a = 0$ ist und keine Nachkommastellen mehr zu berechnen sind.

Symbolen möchten wir Ihnen an dieser Stelle nur einige wenige vorstellen und sie verwenden:

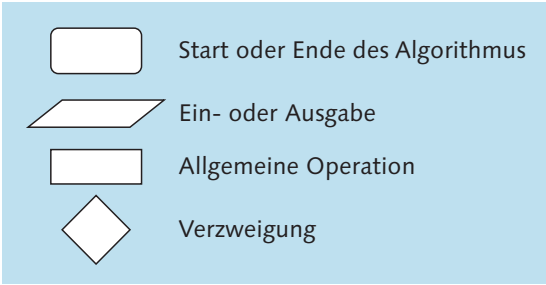


Abbildung 1.3 Symbole im Flussdiagramm

Mit diesen Symbolen können Sie den zuvor nur sprachlich beschriebenen Algorithmus grafisch darstellen, wenn Sie zusätzlich die Abfolge der einzelnen Operationen durch Richtungspfeile kennzeichnen:

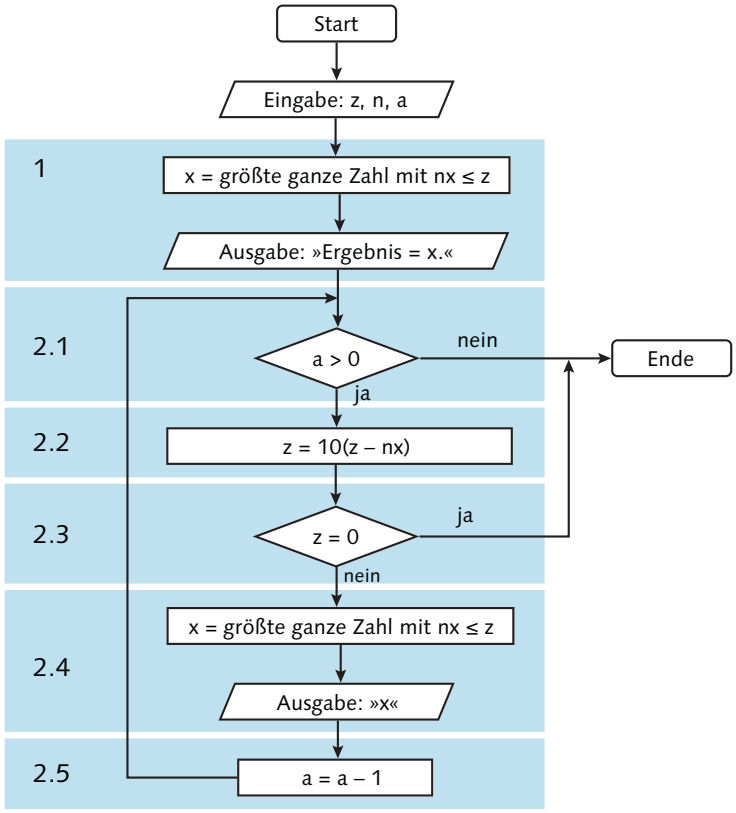


Abbildung 1.4 Flussdiagramm des Algorithmus

In Abbildung 1.4 können Sie den Ablauf des Algorithmus für konkrete Anfangswerte »mit dem Finger« nachfahren und erhalten so eine recht gute Vorstellung von der Dynamik des Verfahrens.

Wir möchten Ihnen den Divisionsalgorithmus anhand des Flussdiagramms für konkrete Daten (`z=84, n=16, a=4`) Schritt für Schritt erläutern. Mehrfach durchlaufene Teile zeichnen wir dabei entsprechend oft, nicht durchlaufene Pfade lassen wir weg:

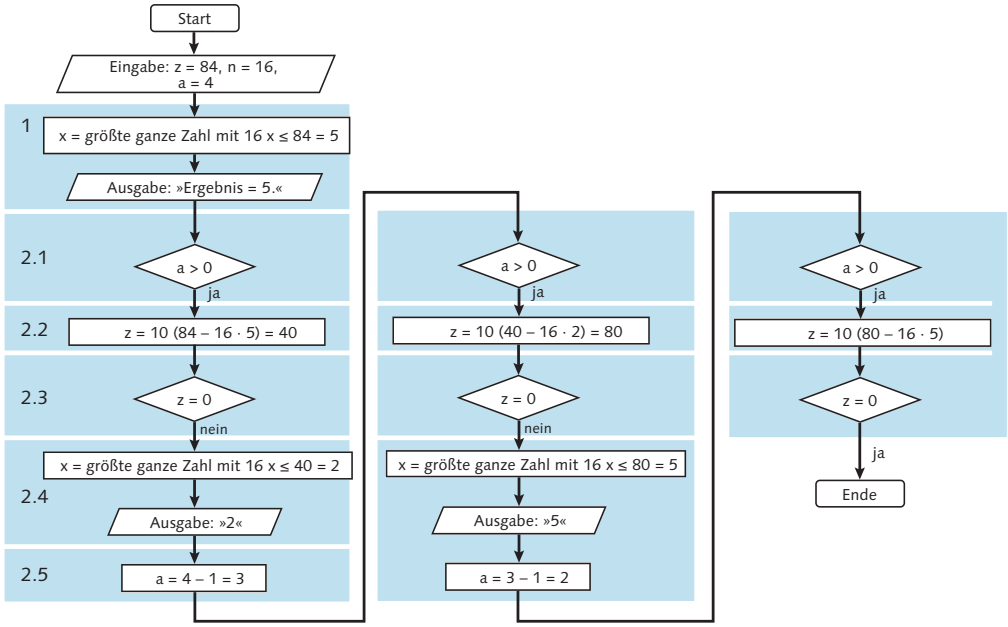


Abbildung 1.5 Das Flussdiagramm für einen konkreten Fall

Als Ergebnis erhalten wir die Ausgabe `"5.25"`. Sie sehen, dass der Algorithmus gewisse Verfahrensschritte (z. B. 2.1) mehrfach – allerdings mit unterschiedlichen Daten – durchläuft. Die Daten steuern letztlich den konkreten Ablauf des Algorithmus. Das Verfahren zeigt im Ablauf eine gewisse Regelmäßigkeit – um nicht zu sagen Monotonie. Gerade solche monotonen Aufgaben würde man sich gern von einer Maschine abnehmen lassen. Eine Maschine müsste natürlich jeden einzelnen Verfahrensschritt »verstehen«, um das Verfahren als Ganzes durchführen zu können. Einige unserer Schritte (z. B. 2.2) erscheinen unmittelbar verständlich, während andere (z. B. 2.4) ein gewisses mathematisches Vorverständnis voraussetzen. Je nachdem, welche Intelligenz man bei demjenigen (Mensch oder Maschine) voraussetzt, der den Algorithmus durchführen soll, wird man an manchen Stellen noch präziser formulieren und einen Verfahrensschritt gegebenenfalls in einfachere Teilschritte zerlegen müssen.

Festgehalten werden sollte noch, dass wir von einem Algorithmus gefordert haben, dass er nach endlich vielen Schritten zu einem Ergebnis kommt (terminiert). Dies ist bei unserem Divisionsalgorithmus durch die Vorgabe der Anzahl der zu berechnenden Nachkommastellen sichergestellt, auch wenn in unserem konkreten Beispiel ein vorzeitiger Abbruch eintritt. Würden wir das Abbruchkriterium fallenlassen, würde unser Verfahren unter Umständen (z. B. bei der Berechnung von `10:3`) nicht abbrechen, und eine mit der Berechnung beauftragte Maschine würde endlos rechnen. Es ist zu befürchten, dass die Eigenschaft des Terminierens für manche Verfahren schwer oder vielleicht auch gar nicht nachzuweisen ist.

1.2 Datenstruktur

Wir starten wieder mit einer Definition:

Was ist eine Datenstruktur?

Eine *Datenstruktur* ist ein Modell, das die zur Lösung eines Problems benötigten Informationen (Ausgangsdaten, Zwischenergebnisse, Endergebnisse) enthält und für alle Informationen genau festgelegte Zugriffswege bereitstellt.

Auch Datenstrukturen hat es bereits lange vor der Programmierung gegeben, obwohl man hier mit einigem Recht sagen kann, dass die Theorie der Datenstrukturen erst mit der maschinellen Datenverarbeitung zur Blüte gekommen ist.

Als Beispiel betrachten wir ein Versandhaus, das seine Geschäftsvorfälle durch drei Karteien organisiert: Eine Kundenkartei mit den personenbezogenen Daten aller Kunden, eine Artikelkartei für die Stammdaten und den Lagerbestand aller lieferbaren Artikel und eine Bestellkartei für alle eingehenden Bestellungen (siehe Abbildung 1.6).

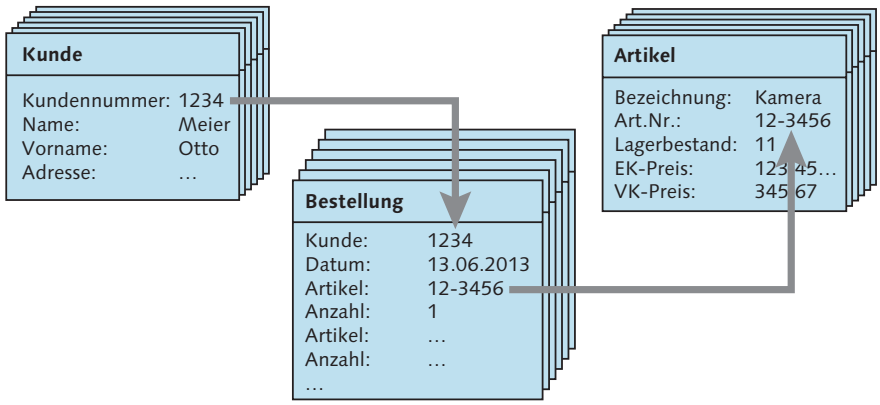


Abbildung 1.6 Verbundene Karteikästen

Ein einzelner Datensatz entspricht einer ausgefüllten Karteikarte. Auf jeder Karteikarte sind zwei Bereiche erkennbar. Links steht jeweils die Struktur der Daten, während rechts die konkreten Datenwerte stehen. Die Datensätze für Kunden, Artikel und Bestellungen sind dabei strukturell verschieden. Neben der Struktur der Karteikarten ist natürlich auch noch die Organisation der einzelnen Karteikästen von Bedeutung. Stellen Sie sich vor, dass die Kundendatei nach Kundennummern, die Artikeldatei nach Artikelnummern und die Bestelldatei nach Bestelldatum sortiert ist. Darüber hinaus gibt es noch Querverweise zwischen den Datensätzen der verschiedenen Karteikästen. In der Bestelldatei finden Sie auf jeder Karteikarte z. B. Artikelnummern und eine Kundennummer.

Die drei Karteikästen mit ihrer Sortierung, der Struktur ihrer Karteikarten und der Querverweisstruktur bilden insgesamt die Datenstruktur. Beachten Sie, dass die konkreten Daten – also das, was auf den ausgefüllten Karteikarten steht – nicht zur Datenstruktur gehören. Die Datenstruktur legt nur die Organisationsform der Daten fest, nicht jedoch die konkreten Datenwerte.

Auf der Datenstruktur arbeiten Algorithmen (z. B. Kundenadresse ändern, Rechnung stellen, Artikel nachbestellen, Lieferung zusammenstellen etc.). Die Effizienz dieser Algorithmen hängt dabei ganz entscheidend von der Organisation der Datenstruktur ab. Zum Beispiel ist die Frage: »Was hat der Kunde Müller dem Unternehmen bisher an Umsatz eingebracht?« ausgesprochen schwer zu beantworten. Dazu müssten Sie zunächst in der Kundendatei die Kundennummer des Kunden Müller finden. Als Nächstes müssten Sie alle Bestellungen durchsuchen, um festzustellen, ob die Kundennummer von Müller dort vorkommt, und schließlich müssten Sie dann auch noch die Preise der in den betroffenen Bestellungen vorkommenden Artikel in der Artikeldatei suchen und aufsummieren. Die Frage: »Welche Artikel in welcher Menge sind im letzten Monat bestellt worden?« lässt sich mit dieser Datenstruktur erheblich einfacher beantworten.

Das Problem, eine »bestmögliche« Organisationsform für Daten zu finden, ist im Allgemeinen unlösbar, weil Sie dazu in der Regel gegenläufige Optimierungsaspekte in Einklang bringen müssten. Sie könnten z. B. bei der oben dargestellten Datenstruktur den Verbesserungsvorschlag machen, alle Kundendaten mit auf der Bestellkartei zu vermerken, um die Rechnungsstellung zu erleichtern. Dadurch erhöht sich dann aber der Aufwand, den Sie bei der Adressänderung eines Kunden in Kauf zu nehmen hätten. Die Erstellung von Datenstrukturen, die alle Algorithmen eines bestimmten Problemfeldes wirkungsvoll unterstützen, ist eine ausgesprochen schwierige Aufgabe, zumal man häufig zum Zeitpunkt der Festlegung einer Datenstruktur noch gar nicht absehen kann, welche Algorithmen in Zukunft mit den Daten dieser Struktur arbeiten werden.

Bei der Fülle der in der Praxis vorkommenden Probleme können Sie natürlich nicht erwarten, dass Sie für alle Probleme passende Datenstrukturen bereitstellen können. Sie müssen lernen, typische, immer wiederkehrende Bausteine zu identifizieren und zu beherrschen. Aus diesen Bausteinen können Sie dann komplexere, jeweils an ein bestimmtes Problem angepasste Strukturen aufbauen.

1.3 Programm

Ein Programm ist, im Gegensatz zu einer Datenstruktur oder einem Algorithmus, etwas sehr Konkretes – zumindest dann, wenn Sie schon einmal ein Programm erstellt oder benutzt haben.

Was ist ein Programm?

Ein *Programm* ist eine eindeutige, formalisierte Beschreibung von Algorithmen und Datenstrukturen, die durch einen automatischen Übersetzungsprozess auf einem Computer ablauffähig ist.

Den zur Formulierung eines Programms verwendeten Beschreibungsformalismus bezeichnen wir als *Programmiersprache*.

Im Gegensatz zu einem Algorithmus fordern wir von einem Programm nicht explizit, dass es terminiert. Viele Programme (z. B. ein Betriebssystem oder Programme zur Überwachung und Steuerung technischer Anlagen) sind auch so konzipiert, dass sie im Prinzip endlos laufen könnten.

Eine Programmiersprache muss nach dieser Definition Elemente zur exakten Beschreibung von Datenstrukturen und Algorithmen enthalten. Programmiersprachen dienen daher nicht nur zur Erstellung lauffähiger Programme, sondern auch zur präzisen Festlegung von Datenstrukturen und Algorithmen. Dazu müssen Sie lernen, in einer Programmiersprache so selbstverständlich zu »reden« wie in einer natürlichen Sprache.

Eigentlich stellen wir gegensätzliche Forderungen an eine Programmiersprache. Sie sollte automatisch übersetzbar, d. h. *maschinenlesbar*, und möglichst verständlich und leicht erlernbar, d. h. *menschenlesbar*, sein, und sie sollte darüber hinaus die maschinellen Berechnungs- und Verarbeitungsmöglichkeiten eines Computers möglichst vollständig ausschöpfen. Maschinenlesbarkeit und Menschenlesbarkeit sind bei den heutigen Maschinenkonzepten unvereinbare Begriffe. Da die Maschinenlesbarkeit jedoch unverzichtbar ist, müssen zwangsläufig bei der Menschenlesbarkeit Kompromisse gemacht werden; Kompromisse, von denen Berufsgruppen wie Systemanalytiker oder Programmierer leben.

1.4 Programmiersprachen

Sie kennen das sicherlich aus dem einen oder anderen Internetforum zur Programmierung. Da fragt ein Newbie um Rat, und es entwickelt sich folgender Dialog:

Newbie: Hallo, ich bin neu hier und habe da eine Frage. Wie kann man in der Programmiersprache abc ...

Experte1: Hallo Newbie, ich kenne abc nicht. Ich programmiere aber schon seit Jahren in xyz. In xyz kann man dein Problem ganz einfach lösen ...

Experte2: Also Experte1, du lebst ja völlig hinter dem Mond. Kein Mensch programmiert heute mehr in xyz. So etwas macht man in uvw ...

Der Expertenstreit, ob nun xyz oder uvw die bessere Programmiersprache sei, wird dann mit wachsender Schärfe über mehrere Wochen ausgefochten, bis beide Kontrahenten ermüdet aufgeben, nicht ohne vorher noch einmal deutlich klarzustellen, dass der jeweils andere keine Ahnung habe und jedes weitere Wort Zeitverschwendung sei. Vielleicht kommt auch der Newbie noch mal zu Wort:

Newbie: Hallo, ich habe inzwischen eine Lösung gefunden. Es war eigentlich ganz einfach ...

Lassen Sie sich auf solche zwecklosen ideologischen Grabenkriege, die seit Jahren mit erstarrten Fronten geführt werden, nicht ein. Sicherlich gibt es Sprachen, die für den einen oder anderen Anwendungszweck besser geeignet sind als andere, aber aus Sicht der Informatik sind alle Sprachen gleich gut (oder eher schlecht). Wichtig ist, dass es verschiedene Programmiersprachen gibt, denn nur diese Vielfalt und der damit verbundene Wettbewerb sorgen für die stetige Weiterentwicklung aller Programmiersprachen.

Vielleicht hilft Ihnen ein bisschen Statistik weiter. Der Tiobe-Index (*tiobe.com*) listet 225 verschiedene Programmiersprachen, die in einer monatlichen Statistik auf ihre Relevanz untersucht werden. Aktuell ergibt sich dabei das folgende Ranking:

Rang	Name	Anteil
1	C	17,8 %
2	Java	16,6 %
3	Objective-C	10,3 %
4	C++	8,8 %
5	PHP	5,9 %

Tabelle 1.1 Ranking der Programmiersprachen

Betrachtet man innerhalb dieser Tabelle die Sprachen, die sich explizit auf C als »Muttersprache« berufen, machen diese einen Anteil von über 40 % aus. Auch Programmiersprachen wie Java oder PHP sind sprachlich eng mit C verwandt, auch wenn sie auf anderen Laufzeitkonzepten beruhen.

Der Tiobe-Index unterscheidet auch verschiedene Programmierparadigmen[4] und kommt hier zu folgendem Ergebnis:

Rang	Name	Anteil
1	Objektorientiertes Paradigma	58,5 %
2	Prozedurales Paradigma	36,6 %
3	Funktionales Paradigma	3,2 %
4	Logisches Paradigma	1,8 %

Tabelle 1.2 Ranking der Programmierparadigmen

Diese Unterscheidung ist eigentlich viel wichtiger als die Unterscheidung in einzelne Programmiersprachen, denn wer eine Sprache eines bestimmten Paradigmas beherrscht, dem fällt es in der Regel leicht, auf eine andere Sprache des gleichen Paradigmas zu wechseln. Sie lernen hier mit C das prozedurale und mit C++ das objektorientierte Paradigma und sind damit für über 90 % aller Fälle bestens gerüstet.

Wenn Sie Ihre Programmierkenntnisse beruflich nutzen wollen, können Sie in der Regel die Programmiersprache, die in einem Softwareprojekt verwendet wird, nicht frei wählen. Die Sprache ist meistens durch innere oder äußere Randbedingungen festgelegt. In dieser Situation ist es wichtig, dass Sie »programmieren« können, und darunter verstehe ich weitaus mehr als die Beherrschung einer Programmiersprache. Wenn ein Verlag einen Autor sucht, dann wird jemand gesucht, der »schreiben« kann. Dabei bedeutet »schreiben« mehr als die bloße Beherrschung von Rechtschreibung und Grammatik. In diesem Sinne versteht sich dieses Buch als ein Lehrbuch zum Programmieren, wobei programmieren weitaus mehr ist als die Beherrschung einer konkreten Programmiersprache. Eines der bedeutendsten Bücher der Informatik heißt:

> The Art of Computer Programming[5] (Die Kunst der Computerprogrammierung)

In diesem mehrbändigen Werk finden Sie nicht eine einzige Zeile Code in einer konkreten Programmiersprache.

4 Unter dem Paradigma einer Programmiersprache versteht man, locker gesprochen, die »Denke«, die hinter einer Programmiersprache steckt.

5 Donald E. Knuth, The Art of Computer Programming

Natürlich macht Programmieren erst richtig Spaß, wenn das Ergebnis (z. B. ein Computerspiel) am Ende über den Bildschirm eines Computers flimmert. Darum nehmen konkrete Programmierbeispiele in C und C++ in diesem Buch breiten Raum ein.

1.5 Aufgaben

A 1.1 Formulieren Sie Ihr morgendliches Aufsteh-Ritual vom Klingeln des Weckers bis zum Verlassen des Hauses als Algorithmus. Berücksichtigen Sie dabei auch verschiedene Wochentagsvarianten! Zeichnen Sie ein Flussdiagramm!

A 1.2 Verfeinern Sie den Algorithmus zur Division zweier Zahlen aus Abschnitt 1.1 so, dass er von jemandem, der nur Zahlen addieren, subtrahieren und der Größe nach vergleichen kann, durchgeführt werden kann! Zeichnen Sie ein Flussdiagramm!

A 1.3 In unserem Kalender sind zum Ausgleich der astronomischen und der kalendarischen Jahreslänge in regelmäßigen Abständen Schaltjahre eingebaut. Zur exakten Festlegung der Schaltjahre dienen die folgenden Regeln:

1. Ist die Jahreszahl durch 4 teilbar, ist das Jahr ein Schaltjahr.
Diese Regel hat allerdings eine Ausnahme:
2. Ist die Jahreszahl durch 100 teilbar, ist das Jahr kein Schaltjahr.
Diese Ausnahme hat wiederum eine Ausnahme:
3. Ist die Jahreszahl durch 400 teilbar, ist das Jahr doch ein Schaltjahr.

Formulieren Sie einen Algorithmus, mit dessen Hilfe man feststellen kann, ob ein bestimmtes Jahr ein Schaltjahr ist oder nicht!

A 1.4 Sie sollen eine unbekannte Zahl x ($a \leq x \leq b$) erraten und haben beliebig viele Versuche dazu. Bei jedem Versuch erhalten Sie die Rückmeldung, ob die gesuchte Zahl größer, kleiner oder gleich der von Ihnen geratenen Zahl ist. Entwickeln Sie einen Algorithmus, um die gesuchte Zahl möglichst schnell zu ermitteln! Wie viele Versuche benötigen Sie bei Ihrem Verfahren maximal?

A 1.5 Formulieren Sie einen Algorithmus, der prüft, ob eine gegebene Zahl eine Primzahl ist oder nicht!

A 1.6 Ihr CD-Ständer hat 100 Fächer, die fortlaufend von 1–100 nummeriert sind. In jedem Fach befindet sich eine CD. Formulieren Sie einen Algorithmus, mit dessen Hilfe Sie die CDs alphabetisch nach Interpreten sortieren können! Das Verfahren soll dabei auf den beiden folgenden Grundfunktionen basieren:

```
vergleiche(n,m)
```

Vergleichen Sie CDs in den Fächern `n` und `m`. Das Ergebnis ist »richtig« oder »falsch« – je nachdem, ob die beiden CDs in der richtigen oder falschen Reihenfolge im Ständer stehen.

```
tausche(n,m)
```

Tauschen Sie die CDs in den Fächern `n` und `m`.

A 1.7 Formulieren Sie einen Algorithmus, mit dessen Hilfe Sie die CDs in Ihrem CD-Ständer jeweils um ein Fach aufwärts verschieben können! Die dabei am Ende herausgeschobene CD kommt in das erste Fach. Das Verfahren soll nur auf der Grundfunktion `tausche` aus Aufgabe 1.6 beruhen.

A 1.8 Formulieren Sie einen Algorithmus, mit dessen Hilfe Sie die Reihenfolge der CDs in Ihrem CD-Ständer umkehren können! Das Verfahren soll nur auf der Grundfunktion `tausche` aus Aufgabe 1.6 beruhen.

A 1.9 In einem Hochhaus mit `20` Stockwerken gibt es einen Aufzug. Im Aufzug sind `20` Knöpfe, mit denen man sein Fahrziel wählen kann, und auf jeder Etage ist ein Knopf, mit dem man den Aufzug rufen kann. Entwickeln Sie einen Algorithmus, der den Aufzug so steuert, dass alle Aufzugbenutzer gerecht bedient werden!

A 1.10 Beim Schach gibt es ein einfaches Endspiel, wenn die eine Seite den König und einen Turm, die andere Seite dagegen nur noch den König auf dem Spielfeld hat:

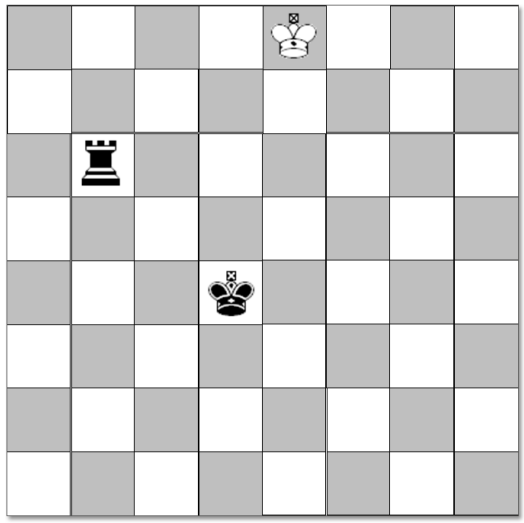


Abbildung 1.7 Darstellung des Endspiels

Versuchen Sie, den Algorithmus für das Endspiel so zu formulieren, dass auch ein Nicht-Schachspieler die Spielstrategie versteht!

Kapitel 2
Einführung in die Programmierung

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.
– Brian Kernighan

Bevor wir in den Mikrokosmos der C-Programmierung abtauchen, wollen wir Softwaresysteme und ihre Erstellung von einer höheren Warte aus betrachten. Dieser Abschnitt dient der Einordnung dessen, was Sie später im Detail kennenlernen werden, in einen Gesamtzusammenhang. Auch wenn Ihnen noch nicht alle Begriffe, die hier fallen werden, unmittelbar klar sind, ist es doch hilfreich, wenn Sie bei den vielen Details, die später wichtig werden, den Blick für das Ganze nicht verlieren.

2.1 Softwareentwicklung

Damit ein Problem durch ein Softwaresystem gelöst werden kann, muss es zunächst einmal erkannt, abgegrenzt und adäquat beschrieben werden. Der Softwareingenieur spricht in diesem Zusammenhang von *Systemanalyse*. In einem weiteren Schritt wird das Ergebnis der Systemanalyse in den *Systementwurf* überführt, der dann Grundlage für die nachfolgende *Realisierung* oder *Implementierung* ist. Der Softwareentwicklungszyklus beginnt also nicht mit der Programmierung, sondern es gibt wesentliche, der Programmierung vorgelagerte, aber auch nachgelagerte Aktivitäten.

Obwohl wir in diesem Buch nur die »Softwareentwicklung im Kleinen« und hier auch nur Realisierungsaspekte behandeln werden, möchten wir Sie doch zumindest auf einige Aktivitäten und Werkzeuge der »Softwareentwicklung im Großen« hinweisen.

Für die Realisierung großer Softwaresysteme muss zunächst einmal ein sogenanntes *Vorgehensmodell* zugrunde gelegt werden. Ausgangspunkt sind dabei Standardvorgehensmodelle wie etwa das V-Modell:

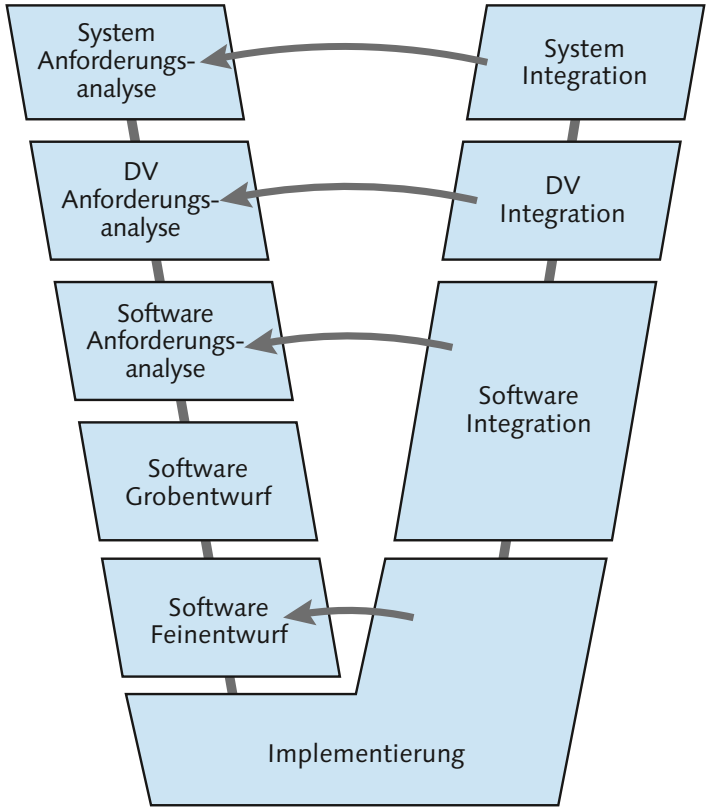


Abbildung 2.1 Das V-Modell

Große Unternehmen verfügen in der Regel über eigene Vorgehensmodelle zur Softwareentwicklung. Ein solches allgemeines Modell muss auf die Anforderungen eines konkreten Entwicklungsvorhabens zugeschnitten werden. Man spricht in diesem Zusammenhang von *Tailoring*. Das auf ein konkretes Projekt zugeschnittene Vorgehensmodell nennt dann alle prinzipiell anfallenden Projektaktivitäten mit den zugeordneten Eingangs- und Ausgangsprodukten (Dokumente, Code ...) sowie deren mögliche Zustände (geplant, in Bearbeitung, vorgelegt, akzeptiert) im Laufe der Entwicklung. Durch Erstellung einer Aktivitätenliste, Aufwandsschätzungen, Reihenfolgeplanung und Ressourcenzuordnung[1] entsteht ein *Projektplan*. Wesentliche Querschnittsaktivitäten eines Projektplans sind:

- Projektplanung und Projektmanagement
- Konfigurations- und Change Management
- Systemanalyse
- Systementwurf

1 Ressourcen sind Mitarbeiter, aber auch technisches Gerät oder Rechenzeit.

- Implementierung
- Test und Integration
- Qualitätssicherung

Diese übergeordneten Tätigkeiten werden dabei oft noch in viele (hundert) Einzelaktivitäten zerlegt. Der Projektplan wird durch regelmäßige Reviews überprüft (Soll-Ist-Vergleich) und dem wirklichen Projektstand angepasst. Ziel ist es, Entwicklungsengpässe, Entwicklungsverzögerungen und Konfliktsituationen rechtzeitig zu erkennen, um wirkungsvoll gegensteuern zu können.

Für alle genannten Aktivitäten gibt es Methoden und Werkzeuge, die den Softwareingenieur bei seiner Arbeit unterstützen. Einige davon seien im Folgenden aufgezählt:

Für die *Projektplanung* gibt es Werkzeuge, die Aktivitäten und deren Abhängigkeiten sowie Aufwände und Ressourcen erfassen und verwalten können. Solche Werkzeuge können dann konkrete Zeitplanungen auf Basis von Aufwandsschätzungen und Ressourcenverfügbarkeit erstellen. Mithilfe der Werkzeuge erstellt man dann Aktivitäten-Abhängigkeitsdiagramme (Pert-Charts) und Aktivitäten-Zeit-Diagramme (Gantt-Charts) sowie Berichte über den Projektfortschritt, aufgelaufene Projektkosten, Soll-Ist-Vergleiche, Auslastung der Mitarbeiter etc.

Das *Konfigurationsmanagement* wird von Werkzeugen, die alle Quellen (Programme und Dokumentation) eines Projekts in ein Archiv aufnehmen und jedem Mitarbeiter aktuelle Versionen mit Sperr- und Ausleihmechanismen zum Schutz vor konkurrierender Bearbeitung zur Verfügung stellen, unterstützt. Die Werkzeuge halten die Historie aller Quellen nach und können jederzeit frühere, konsistente Versionen der Software oder der Dokumentation restaurieren.

Bei der *Systemanalyse* werden objektorientierte Analysemethoden und Beschreibungsformalismen, insbesondere UML (Unified Modeling Language), eingesetzt. Für die Analyse der Datenstrukturen verwendet man häufig sogenannte *Entity-Relationship-Methoden*. Alle genannten Methoden werden durch Werkzeuge (sogenannte CASE[2]-Tools) unterstützt. In der Regel handelt es sich dabei um Werkzeuge zur interaktiven, grafischen Eingabe des jeweiligen Modells. Alle Eingaben werden über ein zentrales Data Dictionary (Datenwörterbuch oder Datenkatalog) abgeglichen und konsistent gehalten. Durch einen Transformationsschritt erfolgt bei vielen Werkzeugen der Übergang von der Analyse zum Design, d. h. zum *Systementwurf*. Auch hier stehen wieder computerunterstützte Verfahren vom Klassen-, Schnittstellen- und Datendesign bis hin zur Codegenerierung oder zur Generierung eines Datenbankschemas oder von Teilen der Benutzeroberfläche (Masken, Menüs) zur Verfügung. Je nach Entwicklungsumgebung gibt es eine Vielzahl von Werkzeugen, die den Programmierer bei der *Implementierung* unterstützen. Verwiesen sei hier besonders auf

2 Computer Aided Software Engineering

die heute sehr kompletten Datenbank-Entwicklungsumgebungen sowie die vielen interaktiven Werkzeuge zur Erstellung grafischer Benutzeroberflächen. Sogenannte *Make-Utilities* verwalten die Abhängigkeiten aller Systembausteine und automatisieren den Prozess der Systemgenerierung aus den aktuellen Quellen.

Werkzeuge zur Generierung bzw. Durchführung von Testfällen und zur Leistungsmessung runden den Softwareentwicklungsprozess in Richtung *Test* und *Qualitätssicherung* ab.

Von den oben angesprochenen Themen interessiert uns hier nur die konkrete Implementierung von Softwaresystemen. Betrachtet man komplexe, aber gut konzipierte Softwaresysteme, findet man häufig eine Aufteilung (Modularisierung) des Systems in verschiedene Ebenen oder Schichten. Die Aufteilung erfolgt so, dass jede Schicht die Dienstleistungen der darunterliegenden Schicht nutzt, ohne deren konkrete Implementierung zu kennen. Typische Schichten eines Grobdesigns sehen Sie in Abbildung 2.2.

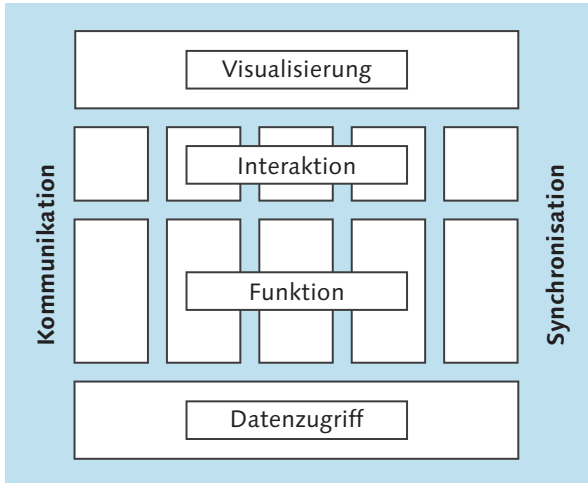


Abbildung 2.2 Schichten eines Softwaresystems

Jede Schicht hat ihre spezifischen Aufgaben.

Auf der Ebene der *Visualisierung* werden die Elemente der Benutzerschnittstelle (Masken, Dialoge, Menüs, Buttons ...), aber auch Grafikfunktionen bereitgestellt. Früher wurde auf dieser Ebene mit Maskengeneratoren gearbeitet. Heute findet man hier objektorientierte Klassenbibliotheken und Werkzeuge zur interaktiven Erstellung von Benutzeroberflächen. Angestrebt wird eine konsequente Trennung von Form und Inhalt. Das heißt, das Layout der Elemente der Benutzerschnittstelle wird getrennt von den Funktionen des Systems. Unter *Interaktion* sind die Funktionen zusammengefasst, die die anwendungsspezifische Steuerung der Benutzerschnittstelle ausmachen. Einfache, nicht anwendungsbezogene Steuerungen, wie z. B. das Aufklappen eines

Menüs, liegen bereits in der Visualisierungskomponente. In der Regel werden die Funktionen zur Interaktion über den Benutzer (Mausklick auf einen Button etc.) angestoßen und vermitteln dann zwischen den Benutzerwünschen und den eigentlichen Funktionen des Anwendungssystems, die hier unter dem Begriff *Funktion* zusammengefasst sind. Auf den Ebenen Interaktion und Funktion zerfällt ein System häufig in unabhängige, vielleicht sogar parallel laufende Module, die auf einem gemeinsamen Datenbestand arbeiten. Die Datenhaltung und der *Datenzugriff* werden häufig in einer übergreifenden Schicht vorgenommen, denn hier muss sichergestellt werden, dass unterschiedliche Funktionen trotz konkurrierenden Zugriffs einen konsistenten Blick auf die Daten haben. Bei großen Softwaresystemen kommen Datenbanken mit ihren Management-Systemen zum Einsatz. Diese verfügen über spezielle Sprachen zur Definition, Abfrage, Manipulation und Integritätssicherung von Daten. Unterschiedliche Teile eines Systems können auf einem Rechner, aber auch verteilt in einem lokalen oder weltweiten Netz laufen. Wir sprechen dann von einem »verteilten System«. Unter dem Begriff *Kommunikation* werden Funktionen zum Datenaustausch zwischen verschiedenen Komponenten eines verteilten Systems zusammengefasst. Über Funktionen zur *Synchronisation* schließlich werden parallel arbeitende Systemfunktionen, etwa bei konkurrierendem Zugriff auf Betriebsmittel, wieder koordiniert. Die Schichten Kommunikation und Synchronisation stützen sich stark auf die vom jeweiligen Betriebssystem bereitgestellten Funktionen und sind von daher häufig an ein bestimmtes Betriebssystem gebunden. In allen anderen Bereichen versucht man, nach Möglichkeit portable Funktionen, d. h. Funktionen, die nicht an ein bestimmtes System gebunden sind, zu erstellen. Man erreicht dies, indem man allgemein verbindliche Standards, wie z. B. die Programmiersprache C, verwendet.

Von den zuvor genannten Aspekten betrachten wir, wie durch eine Lupe, nur einen kleinen Ausschnitt, und zwar die Realisierung einzelner Anwendungsfunktionen:

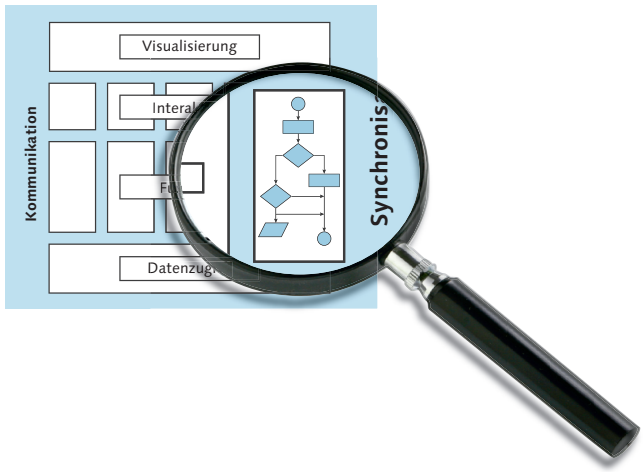


Abbildung 2.3 Realisierung von Anwendungsfunktionen

In den Schichten Visualisierung und Interaktion werden wir uns auf das absolute Minimum beschränken, das wir benötigen, um lauffähige Programme zu erhalten, die Benutzereingaben entgegennehmen und Ergebnisse auf dem Bildschirm ausgeben können. Auch den Datenzugriff werden wir nur an sehr spartanischen Dateikonzepten praktizieren. Kommunikation und Synchronisation behandeln wir hier gar nicht. Diese Themen werden in Büchern über Betriebssysteme oder verteilte Systeme thematisiert.

2.2 Die Programmierumgebung

Bei der Realisierung von Softwaresystemen ist die Programmierung natürlich eine der zentralen Aufgaben. Abbildung 2.4 zeigt die Programmierung als eine Abfolge von Arbeitsschritten:

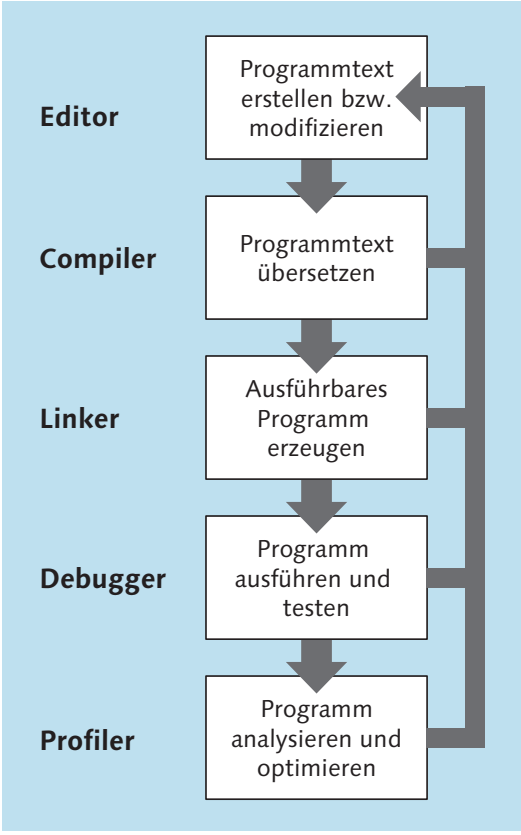


Abbildung 2.4 Arbeitsschritte bei der Programmierung

Der Programmierer wird bei jedem dieser Schritte von folgenden Werkzeugen unterstützt:

- Editor
- Compiler
- Linker
- Debugger
- Profiler

Sie werden diese Werkzeuge hier nur grundsätzlich kennenlernen. Es ist absolut notwendig, dass Sie, parallel zur Arbeit mit diesem Buch, eine Entwicklungsumgebung zur Verfügung haben, mit der Sie Ihre C/C++-Programme erstellen. Um welche Entwicklungsumgebung es sich dabei handelt, ist relativ unwichtig, da wir uns mit unseren Programmen nur in einem Bereich bewegen werden, der von allen Entwicklungsumgebungen unterstützt wird. Alle konkreten Details über Editor, Compiler, Linker, Debugger und Profiler entnehmen Sie bitte den Handbüchern Ihrer Entwicklungsumgebung!

2.2.1 Der Editor

Ein Programm wird wie ein Brief in einer Textdatei erstellt und abgespeichert. Der Programmtext (Quelltext) wird mit einem sogenannten *Editor*[3] erstellt. Es kann nicht Sinn und Zweck dieses Buches sein, Ihnen einen bestimmten Editor mit all seinen Möglichkeiten vorzustellen. Die Editoren der meisten Entwicklungsumgebungen orientieren sich an den Möglichkeiten moderner Textverarbeitungssysteme, sodass Sie, sofern Sie mit einem Textverarbeitungssystem vertraut sind, keine Schwierigkeiten mit der Bedienung des Editors Ihrer Entwicklungsumgebung haben sollten. Über die reinen Textverarbeitungsfunktionen hinaus hat der Editor in der Regel Funktionen, die Sie bei der Programmerstellung gezielt unterstützen. Art und Umfang dieser Funktionen sind allerdings auch von Entwicklungsumgebung zu Entwicklungsumgebung verschieden, sodass wir hier nicht darauf eingehen können.

Üben Sie gezielt den Umgang mit den Funktionen Ihres Editors, denn auch die »handwerklichen« Aspekte der Programmierung sind wichtig!

Mit dem Editor als Werkzeug erstellen wir unsere Programme, die wir in Dateien ablegen. Im Zusammenhang mit der C-Programmierung sind dies:

- Header-Dateien
- Quellcodedateien

3 engl. to edit = einen Text erstellen oder überarbeiten

Header-Dateien (engl. Headerfiles) sind Dateien, die Informationen zu Datentypen und -strukturen, Schnittstellen von Funktionen etc. enthalten. Es handelt sich dabei um allgemeine Vereinbarungen, die an verschiedenen Stellen (d. h. in verschiedenen Source- und Headerfiles) einheitlich und konsistent benötigt werden. Headerfiles stehen im Moment noch nicht im Mittelpunkt unseres Interesses. Spätestens mit der Einführung von Datenstrukturen werden wir Ihnen jedoch die große Bedeutung dieser Dateien erläutern.

Die *Quellcodedateien* (engl. Sourcefiles) enthalten den eigentlichen Programmtext und stehen für uns zunächst im Vordergrund.

Den Typ (Header oder Source) einer Datei können Sie bereits am Namen der Datei erkennen. Header-Dateien sind an der Dateinamenserweiterung *.h*, Quellcodedateien an der Erweiterung *.c* in C bzw. *.cpp* und *.cc* in C++ zu erkennen.

2.2.2 Der Compiler

Ein Programm in einer höheren Programmiersprache ist auf einem Rechner nicht unmittelbar ablauffähig. Es muss durch einen Compiler[4] in die Maschinensprache des Trägersystems übersetzt werden.

Der Compiler übersetzt den Quellcode (die C- oder CPP-Dateien) in den sogenannten *Objectcode* und nimmt dabei verschiedene Prüfungen zur Korrektheit des übergebenen Quellcodes vor. Alle Verstöße gegen die Regeln der Programmiersprache[5] werden durch gezielte Fehlermeldungen unter Angabe der Zeile angezeigt. Nur ein vollständig fehlerfreies Programm kann in Objectcode übersetzt werden. Viele Compiler mahnen auch formal zwar korrekte, aber möglicherweise problematische Anweisungen durch Warnungen an. Bei der Fehlerbeseitigung sollten Sie strikt in der Reihenfolge, in der der Compiler die Fehler gemeldet hat, vorgehen. Denn häufig findet der Compiler nach einem Fehler nicht den richtigen Wiederaufsetzpunkt und meldet Folgefehler in Ihrem Programmcode, die sich bei genauem Hinsehen als gar nicht vorhanden erweisen.

Der Compiler erzeugt zu jedem Sourcefile genau ein Objectfile, wobei nur die innere Korrektheit des Sourcefiles überprüft wird. Übergreifende Prüfungen können hier noch nicht durchgeführt werden. Der vom Compiler erzeugte Objectcode ist daher auch noch nicht lauffähig, denn ein Programm besteht in der Regel aus mehreren Sourcefiles, deren Objectfiles noch in geeigneter Weise kombiniert werden müssen.

4 engl. to compile = zusammenstellen
5 Man nennt so etwas einen *Syntaxfehler*.

2.2.3 Der Linker

Die noch fehlende Montage der einzelnen Objectfiles zu einem fertigen Programm übernimmt der Linker[6]. Der Linker nimmt dabei die noch ausstehenden übergreifenden Prüfungen vor. Auch dabei kann noch eine Reihe von Fehlern aufgedeckt werden. Zum Beispiel kann der Linker in der Zusammenschau aller Objectfiles feststellen, dass versucht wird, eine Funktion zu verwenden, die es nirgendwo gibt.

Letztlich erstellt der Linker das ausführbare Programm, zu dem auch weitere Funktions- oder Klassenbibliotheken hinzugebunden werden können. Bibliotheken enthalten kompilierte Funktionen, zu denen zumeist kein Quellcode verfügbar ist, und werden z. B. vom Betriebssystem oder dem C-Laufzeitsystem zur Verfügung gestellt. Im Internet finden Sie viele nützliche, freie oder kommerzielle Bibliotheken, die Ihnen die Programmierarbeit sehr erleichtern können.

2.2.4 Der Debugger

Der Debugger[7] dient zum Testen von Programmen. Mit dem Debugger können die erstellten Programme bei ihrer Ausführung beobachtet werden. Darüber hinaus können Sie in das laufende Programm durch manuelles Ändern von Variablenwerten etc. eingreifen. Ein Debugger ist nicht nur zur Lokalisierung von Programmierfehlern, sondern auch zur Analyse eines Programms durch Nachvollzug des Programmablaufs oder zum interaktiven Erlernen einer Programmiersprache ausgesprochen hilfreich. Arbeiten Sie sich daher frühzeitig in die Bedienung des Debuggers Ihrer Entwicklungsumgebung ein und nicht erst, wenn Sie ihn zur Fehlersuche benötigen.

Bei der Fehlersuche in Ihren Programmen bedenken Sie stets, was Brian Kernighan, neben Dennis Ritchie und Ken Thomson einer der Väter der Programmiersprache C, in dem eingangs bereits erwähnten Zitat sagt, das frei übersetzt lautet:

> Fehlersuche ist doppelt so schwer wie das Schreiben von Code. Wenn man also versucht, den Code so intelligent wie möglich zu schreiben, ist man prinzipiell nicht in der Lage, seine Fehler zu finden.

2.2.5 Der Profiler

Wenn Sie die Performance Ihrer Programme analysieren und optimieren wollen, sollten Sie einen Profiler verwenden. Ein Profiler überwacht Ihr Programm zur Laufzeit und erstellt sogenannte *Laufzeitprofile*, die Informationen über die verbrauchte Rechenzeit und den in Anspruch genommenen Speicher enthalten. Häufig können Sie ein Programm nicht gleichzeitig bezüglich seiner Laufzeit und seines Speicher-

6 engl. to link = verbinden
7 engl. to debug = entwanzen

verbrauchs optimieren. Ein besseres Zeitverhalten erkauft man oft mit einem höheren Speicherbedarf und einen geringeren Speicherbedarf mit einer längeren Laufzeit. Sie kennen das von der Kaufentscheidung für ein Auto. Wenn Sie mehr transportieren wollen, müssen Sie Einschränkungen bei der Höchstgeschwindigkeit hinnehmen. Wenn Sie umgekehrt ein schnelles Auto wollen, haben Sie in der Regel weniger Raum. Im Extremfall müssen Sie sich zwischen einem Lkw und einem Sportwagen entscheiden.

Die Analyse der Speicher- und Laufzeitkomplexität von Programmen gehört zur professionellen Softwareentwicklung wie die Analyse der Effizienz eines Motors zu einer professionellen Motorenentwicklung. Ein ineffizientes Programm ist wie ein Motor, der die zugeführte Energie überwiegend in Abwärme umsetzt.

Kapitel 3
Ausgewählte Sprachelemente von C

Hello, World
– Sprichwörtlich gewordene Ausgabe eines C-Programms von Brian Kernighan

Dieses Kapitel führt im Vorgriff auf spätere Kapitel einige grundlegende Programmkonstrukte sowie Funktionen zur Tastatureingabe bzw. Bildschirmausgabe ein. Ziel dieses Kapitels ist es, Ihnen das minimal notwendige Rüstzeug zur Erstellung kleiner, interaktiver Beispielprogramme bereitzustellen. Es geht in den Beispielen dieses Kapitels noch nicht darum, komplizierte Algorithmen zu entwickeln, sondern sich anhand einfacher, überschaubarer Beispiele mit Editor, Compiler und gegebenenfalls Debugger vertraut zu machen. Es ist daher wichtig, dass Sie die Beispiele – so banal sie Ihnen anfänglich auch erscheinen mögen – in Ihrer Entwicklungsumgebung editieren, kompilieren, linken und testen.

3.1 Programmrahmen

Der minimale Rahmen für unsere Beispielprogramme sieht wie folgt aus:

```
A  # include <stdio.h>
B  # include <stdlib.h>

   void main()
       {
C      ...
       ...
       ...

D      ...
       ...
       ...
       }
```

Listing 3.1 Ein minimaler Programmrahmen

Die beiden ersten mit # beginnenden Zeilen (mit A und B am Rand gekennzeichnet) übernehmen Sie einfach in Ihren Programmcode. Ich werde später etwas dazu sagen.

Das eigentliche Programm besteht aus einem *Hauptprogramm*, das in C mit `main` bezeichnet werden muss. Den Zusatz `void` und die hinter `main` stehenden runden Klammern werde ich ebenfalls später erklären.

Die auf `main` folgenden geschweiften Klammern umschließen den Inhalt des Hauptprogramms, der aus *Variablendefinitionen* (im mit C markierten Bereich) und *Programmcode* (im folgenden Bereich D) besteht. Geschweifte Klammern kommen in der Programmiersprache C immer vor, wenn etwas zusammengefasst werden soll. Geschweifte Klammern treten immer paarig auf. Sie sollten die Klammern so einrücken, dass man sofort erkennen kann, welche schließende Klammer zu welcher öffnenden Klammer gehört. Das erhöht die Lesbarkeit Ihres Codes.

Der hier gezeigte Rahmen stellt bereits ein vollständiges Programm dar, das Sie kompilieren, linken und starten können. Sie können natürlich nicht erwarten, dass dieses Programm irgendetwas macht. Damit das Programm etwas macht, müssen wir den Bereich zwischen den geschweiften Klammern mit Variablendefinitionen und Programmcode füllen.

3.2 Zahlen

Natürlich benötigen wir in unseren Programmen gelegentlich konkrete Zahlenwerte. Man unterscheidet dabei zwischen ganzen Zahlen, z. B.:

```
1234
-4711
```

und Gleitkommazahlen, z. B.:

```
1.234
-47.11
```

Diese Schreibweisen sind Ihnen bekannt. Wichtig ist, dass bei Gleit*komma*zahlen, den angelsächsischen Konventionen folgend, ein Dezimal*punkt* verwendet wird.

3.3 Variablen

Variablen bilden das »Gedächtnis« eines Computerprogramms. Sie dienen dazu, Datenwerte eines bestimmten Typs zu speichern, die wir für unser Programm benötigen. Bei den Typen denken wir vorerst nur an Zahlen, also ganze Zahlen oder Gleitkommazahlen. Später werden auch andere Datentypen hinzukommen.

Was ist eine Variable?

Unter einer *Variablen* verstehen wir einen mit einem Namen versehenen Speicherbereich, in dem Daten eines bestimmten Typs hinterlegt werden können.

Das im Speicherbereich der Variablen hinterlegte Datum bezeichnen wir als den *Wert* der Variablen.

Zu einer Variablen gehören also:

- ein Name
- ein Typ
- ein Speicherbereich
- ein Wert

Den Namen vergibt der Programmierer. Der Name dient dazu, die Variable im Programm eindeutig ansprechen zu können. Denkbare Typen sind derzeit »ganze Zahl« oder »Gleitkommazahl«. Der Speicherbereich, in dem eine Variable angelegt ist, wird durch den Compiler/Linker festgelegt und soll uns im Moment nicht interessieren. Zunächst möchten wir Ihnen erläutern, wie Sie Variablen in einem Programm anlegen und wie Sie sie dann mit Werten versehen.

Variablen müssen vor ihrer erstmaligen Verwendung angelegt (definiert) werden. Dazu wird im Programm der Typ der Variablen, gefolgt vom Variablennamen, angegeben (A). Die Variablendefinition wird durch ein Semikolon abgeschlossen. Mehrere solcher Definitionen können aufeinanderfolgen, und mehrere Variablen gleichen Typs können in einem Zug definiert werden (B):

```
   # include <stdio.h>
   # include <stdlib.h>

   void main()
       {
A      int summe;
       float hoehe;
B      int a, b, c;
       }
```

Listing 3.2 Unterschiedliche Variablendefinitionen

Sie sehen hier zwei verschiedene Typen: `int` und `float`. Der Typ `int`[1] steht für eine ganze Zahl, `float`[2] für eine Gleitkommazahl. Für numerische Berechnungen würde

1 engl. Integer = ganze Zahl
2 engl. Floatingpoint Number = Gleitkommazahl

eigentlich der Typ `float` ausreichen, da eine ganze Zahl immer als Gleitkommazahl dargestellt werden kann. Es ist aber sinnvoll, diese Unterscheidung zu treffen, da ein Computer mit ganzen Zahlen sehr viel effizienter umgehen kann als mit Gleitkommazahlen. Das Rechnen mit ganzen Zahlen ist darüber hinaus exakt, während das Rechnen mit Gleitkommazahlen immer mit Ungenauigkeiten verbunden ist. Auf der anderen Seite haben Gleitkommazahlen einen erheblich größeren Rechenbereich als ganze Zahlen und werden dringend benötigt, wenn man sehr kleine oder sehr große Zahlen verarbeiten will. Grundsätzlich sollten Sie aber, wann immer möglich, den Datentyp `int` gegenüber `float` bevorzugen.

Der Variablenname kann vom Programmierer relativ frei vergeben werden und besteht aus einer Folge von Buchstaben (keine Umlaute oder ß) und Ziffern. Zusätzlich erlaubt ist das Zeichen »_«. Das erste Zeichen eines Variablennamens muss ein Buchstabe (oder »_«) sein. Grundsätzlich sollten Sie sinnvolle Variablennamen vergeben. Darunter verstehe ich Namen, die auf die beabsichtigte Verwendung der Variablen hinweisen. Variablennamen wie `summe` oder `maximum` helfen unter Umständen, ein Programm besser zu verstehen. C unterscheidet im Gegensatz zu manchen anderen Programmiersprachen zwischen Buchstaben in Groß- bzw. Kleinschreibung. Das bedeutet, dass es sich bei `summe`, `Summe` und `SUMME` um drei verschiedene Variablen handelt. Vermeiden Sie mögliche Fehler oder Missverständnisse, indem Sie Variablennamen immer kleinschreiben.

3.4 Operatoren

Variablen und Zahlen an sich sind wertlos, wenn man nicht sinnvolle Operationen mit ihnen ausführen kann. Spontan denkt man dabei sofort an die folgenden Operationen:

- Variablen Zahlenwerte zuweisen
- mit Variablen und Zahlen rechnen
- Variablen und Zahlen miteinander vergleichen

Diese Möglichkeiten gibt es natürlich auch in der Programmiersprache C.

3.4.1 Zuweisungsoperator

Variablen können direkt bei ihrer Definition oder später im Programm Werte zugewiesen werden. Die Notation dafür ist naheliegend:

```
   # include <stdio.h>
   # include <stdlib.h>

   void main()
       {
A      int summe = 1;
B      float hoehe = 3.7;
C      int a, b = 0, c;

D      a = 1;
E      hoehe = a;
F      a = 2;
       }
```

Listing 3.3 Wertzuweisung an Variablen

Bei einer Zuweisung steht links vom Gleichheitszeichen der Name einer zuvor definierten Variablen (A–F). Dieser Variablen wird durch die Zuweisung ein Wert gegeben. Als Wert kommen dabei konkrete Zahlen, aber auch Variablenwerte oder allgemeinere Ausdrücke (Berechnungen, Formeln etc.) infrage. Variablen können auch direkt bei der Definition initialisiert werden (A–C). Die Wertzuweisungen erfolgen in der angegebenen Reihenfolge, sodass wir im oben genannten Beispiel davon ausgehen können, dass `a` bereits den Wert `1` hat, wenn die Zuweisung an `hoehe` erfolgt (E). Zuweisungen sind nicht endgültig. Sie können den Wert einer Variablen jederzeit durch eine erneute Zuweisung ändern. Nicht initialisierte Variablen wie `a` und `c` in der Zeile (C) haben einen »Zufallswert«.

Wichtig ist, dass der zugewiesene Wert zum Typ der Variablen passt. Das bedeutet, dass Sie einer Variablen vom Typ `int` nur einen `int`-Wert zuweisen können. Einer `float`-Variablen können Sie dagegen einen `int`- oder einen `float`-Wert zuweisen, da ja eine ganze Zahl problemlos auch als Gleitkommazahl aufgefasst werden kann.

Eine Zuweisungsoperation hat übrigens den zugewiesenen Wert wiederum als eigenen Wert, sodass Zuweisungen, wie im folgenden Beispiel gezeigt, kaskadiert werden können:

```
a = b = c = 1;
```

3.4.2 Arithmetische Operatoren

Mit Variablen und Zahlen können Sie rechnen, wie Sie es von der Schulmathematik her gewohnt sind:

```
   # include <stdio.h>
   # include <stdlib.h>

   void main()
       {
       int summe = 1;
       float hoehe;
       int a, b, c = 0;

A      hoehe = 1.2 + 2*c;
B      a = b + c;
C      summe = summe + 1;
       }
```

Listing 3.4 Verwendung arithmetischer Operatoren

Variablenwerte können durch Formeln berechnet werden, und in Formeln können dabei wieder Variablen vorkommen (A).

[!] Besondere Vorsicht ist bei der Verwendung nicht initialisierter Variablen geboten, da das Ergebnis einer Operation auf nicht initialisierten Variablen undefiniert ist (B).

Die gleiche Variable kann auch auf beiden Seiten einer Zuweisung vorkommen (C).

In den Formelausdrücken auf der rechten Seite der Zuweisung können dabei die folgenden Operatoren verwendet werden:

Operator	Verwendung	Bedeutung
+	x + y	Addition von x und y
-	x – y	Subtraktion von x und y
*	x * y	Multiplikation von x und y
/	x / y	Division von x durch y (y ≠ 0)
%	x % y	Rest bei ganzzahliger Division von x durch y (Modulo-Operator, y ≠ 0)

Tabelle 3.1 Grundlegende Operatoren in C

Sie können in Formelausdrücken Klammern setzen, um eine bestimmte Auswertungsreihenfolge zu erzwingen. In Fällen, die nicht durch Klammern eindeutig geregelt sind, greift dann die aus der Schule bekannte Regel:

Punktrechnung (*, /, %) geht vor Strichrechnung (+, -).

Im Zweifel sollten Sie Klammern setzen, denn Klammern machen Formeln besser lesbar und haben keinen Einfluss auf die Verarbeitungsgeschwindigkeit des Programms.

Einige Beispiele:

```
int a;
float b;
float c;

a = 1;
b = (a+1)*(a+2);
c = (3.14*a - 2.7)/5;
```

Ganze Zahlen und Gleitkommazahlen können in Formeln durchaus gemischt vorkommen. Es wird immer so lange wie möglich im Bereich der ganzen Zahlen gerechnet. Sobald aber die erste Gleitkommazahl ins Spiel kommt, wird die weitere Berechnung im Bereich der Gleitkommazahlen durchgeführt.

Die Variable auf der linken Seite einer Zuweisung kann auch auf der rechten Seite derselben Zuweisung vorkommen. Zuweisungen dieser Art sind nicht nur möglich, sie kommen sogar ausgesprochen häufig vor. Zunächst wird der rechts vom Zuweisungsoperator stehende Ausdruck vollständig ausgewertet, dann wird das Ergebnis der Variablen links vom Gleichheitszeichen zugewiesen. Die Anweisung

```
a = a+1;
```

enthält also keinen mathematischen Widerspruch, sondern erhöht den Wert der Variablen `a` um `1`. Treffender wäre daher eigentlich die Notation:

```
a ← a+1;
```

Anweisungen wie `a = a + 5` oder `b = b – a` werden in Programmen sogar recht häufig verwendet. Sie können dann vereinfachend `a += 5` oder `b -= a` schreiben. Insgesamt gibt es folgende Vereinfachungsmöglichkeiten:

Operator	Verwendung	Entsprechung
+=	x += y	x = x + y
-=	x -= y	x = x – y
*=	x *= y	x = x * y

Tabelle 3.2 Vereinfachende Operatoren

Operator	Verwendung	Entsprechung
/=	x /= y	x = x / y
%=	x %= y	x = x % y

Tabelle 3.2 Vereinfachende Operatoren (Forts.)

In dem noch häufiger vorkommenden Fall einer Addition oder Subtraktion von 1 kann man noch einfacher formulieren:

Operator	Verwendung	Entsprechung
++	x++ bzw. ++x	x = x + 1
--	x-- bzw. --x	x = x – 1

Tabelle 3.3 Operatoren für die Addition und Subtraktion von 1

Diese Operatoren gibt es in Präfix- und Postfixnotation. Das heißt, diese Operatoren können ihrem Operanden voran- oder nachgestellt werden. Im ersten Fall wird der Operator angewandt, bevor der Operand in einen Ausdruck eingeht, im zweiten Fall erst danach. Das kann ein kleiner, aber bedeutsamer Unterschied sein. Betrachten Sie dazu das folgende Beispiel:

```
   int i, k;

   i = 0;
A  k = i++;

   i = 0;
B  k = ++i;
```

In der Postfix-Notation (A) wird der Wert von `i` erst nach der Zuweisung an `k` erhöht. Also: `k = 0`. In der Präfix-Variante hingegen (B) wird der Wert von `i` vor der Zuweisung an `k` erhöht. Also: `k = 1`.

Die Variable `i` hat in beiden Fällen im Anschluss an die Zuweisung den Wert 1.

Auf eine Besonderheit möchten wir Sie an dieser Stelle unbedingt hinweisen:

[!] Das Ergebnis einer arithmetischen Operation, an der *nur* ganzzahlige Operanden beteiligt sind, ist *immer* eine ganze Zahl.

Im Falle einer Division wird in dieser Situation eine *Division ohne Rest* (Integer-Division) durchgeführt.

Betrachten Sie dazu das folgende Codefragment:

```
a = (100*10)/100;
b = 100*(10/100);
```

Rein mathematisch müsste eigentlich in beiden Fällen 10 als Ergebnis herauskommen. Im Programm ergibt sich aber `a = 10` und `b = 0`. Dabei handelt es sich nicht um einen Rechen- oder Designfehler, das ist ein ganz wichtiges und gewünschtes Verhalten. Die Integer-Division ist für die Programmierung mindestens genauso wichtig wie die »richtige« Division.

Wenn Sie sich bei einer Integer-Division für den unter den Tisch fallenden Rest interessieren, können Sie diesen mit dem Modulo-Operator (%) ermitteln. Der Ausdruck

```
a = 20%7;
```

berechnet den Rest, der bei einer Division von `20` durch `7` bleibt, und weist diesen der Variablen `a` zu. Die Variable `a` hat also anschließend den Wert 6. Im Gegensatz zu den anderen hier besprochenen Operatoren müssen bei einer Modulo-Operation beide Operanden ganzzahlig und sollten sogar positiv sein.

Die Integer-Division bildet zusammen mit dem Modulo-Operator ein in der Programmierung unverzichtbares Operatorengespann. Ich möchte Ihnen das an einem Beispiel erläutern. Stellen Sie sich vor, dass Sie im Rechner eine zweidimensionale Struktur (z. B. ein Foto) mit einer gewissen Höhe (`hoehe`) und Breite (`breite`) verwalten:

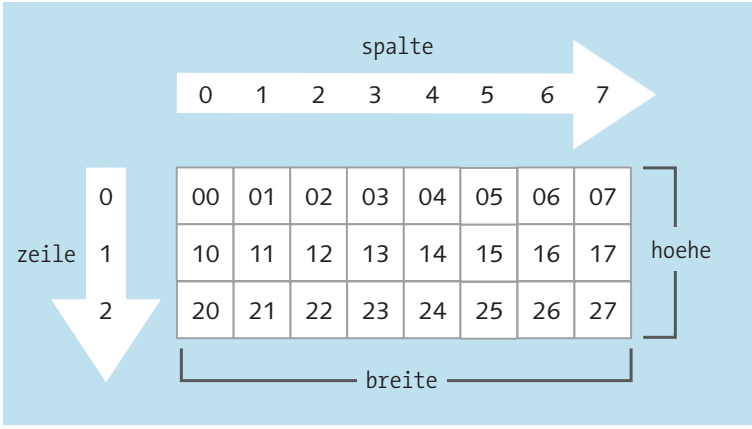


Abbildung 3.1 Beispiel einer zweidimensionalen Struktur

Dieses Bild werden Sie nun in eine eindimensionale Struktur (z. B. eine Datei) umspeichern:

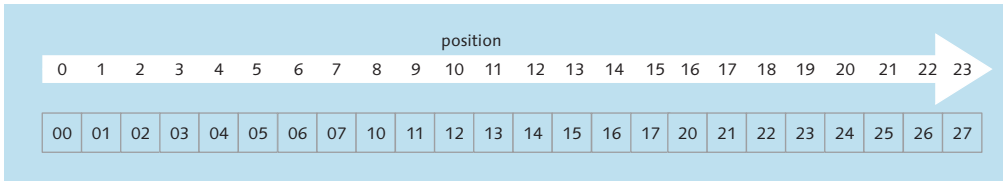


Abbildung 3.2 Beispiel einer eindimensionalen Struktur

Wenn Sie aus Zeile und Spalte in der zweidimensionalen Struktur die Position in der eindimensionalen Struktur berechnen möchten, geht das mit der Formel:

```
position = zeile*breite + spalte;
```

Um umgekehrt aus der Position die Zeile und die Spalte für einen Bildpunkt zu berechnen, benötigen Sie die Integer-Division und den Modulo-Operator. Es ist nämlich:

```
zeile = position/breite;
spalte = position%breite;
```

Beachten Sie dabei, dass alle Positionsangaben hier beginnend mit der Startposition 0 festgelegt sind. Das werden wir auch zukünftig immer so halten, da diese Festlegung zu einfacheren Positionsberechnungen führt, als wenn man mit der Position 1 beginnen würde. Also: Das 1. Element befindet sich an der Position 0, das 2. an der Position 1 etc.

Das Beispiel zeigt, dass in der Integer-Welt das Tandem aus Integer-Division und Modulo-Operation in gewisser Weise die Umkehrung der Multiplikation darstellt und somit an die Stelle der »richtigen« Division tritt. Auf dieses Tandem werden Sie immer wieder bei der Programmierung stoßen. Wenn Sie z. B. die drittletzte Ziffer einer bestimmten Zahl im Dezimalsystem bestimmen wollen, erhalten Sie diese mit der Formel:

```
ziffer = (zahl/100)%10;
```

Bedenken Sie aber immer, dass bei der Integer-Division eine Berechnung der Form `(a/b)*b` nicht den Wert `a` als Ergebnis haben muss. Das Ergebnis ist im Vergleich zur exakten Rechnung die nächstkleinere Zahl, die durch `b` teilbar ist.

3.4.3 Typkonvertierungen

Manchmal möchte man, obwohl man es nur mit Integer-Werten zu tun hat, eine »richtige« Division durchführen und das Ergebnis einer Gleitkommazahl zuweisen. Die bloße Zuweisung an eine Gleitkommazahl konvertiert das Ergebnis zwar automatisch in eine Gleitkommazahl, aber erst nachdem die Division durchgeführt wurde:

```
  void main()
      {
      int a = 1, b = 2;
      float x;
A     x = a/b;
      }
```

Listing 3.5 Beispiel der Integer-Division

Das Ergebnis der Division in der Zeile (A) ist 0.

Bevor Sie nun künstlich eine Gleitkommazahl in die Division einbringen, können Sie in der Formel eine Typkonvertierung durchführen. Sie ändern z. B. für die Berechnung (und nur für die Berechnung) den Datentyp von `a` in `float`, indem Sie der Variablen den gewünschten Datentyp in Klammern voranstellen:

```
  void main()
     {
     int a = 1, b = 2;
     float x;

A    x = ((float)a)/b;
     }
```

Listing 3.6 Typumwandlung vor der Division

Durch die explizite Typumwandlung (A) wird `a` vor der Division in `float` konvertiert. Das Ergebnis der Division ist dann 0.5.

Bei der Typumwandlung handelt es sich um einen einstelligen Operator – den sogenannten *Cast-Operator*. Eine Typumwandlung bezeichnet man auch als *Typecast*.

3.4.4 Vergleichsoperationen

Zahlen und Variablen können untereinander verglichen werden. Tabelle 3.4 zeigt die in C verwendeten Vergleichsoperatoren:

Operator	Verwendung	Entsprechung
<	x < y	kleiner
<=	x <= y	kleiner oder gleich
>	x > y	größer
>=	x >= y	größer oder gleich
==	x == y	gleich
!=	x != y	ungleich

Tabelle 3.4 Vergleichsoperatoren

Auf der linken bzw. rechten Seite eines Vergleichsausdrucks können beliebige Ausdrücke (üblicherweise handelt es sich um arithmetische Ausdrücke) mit Variablen oder Zahlen stehen:

```
a < 7
a <= 2*(b+1)
a+1 == a*a
```

Das Ergebnis eines Vergleichs ist ein logischer Wert (»wahr« oder »falsch«), der in C durch `1` (wahr) oder `0` (falsch) dargestellt wird. Mit diesem Wert können Sie dann, wie mit einem durch einen arithmetischen Ausdruck gewonnenen Wert, weiterarbeiten.

Vergleiche stellt man allerdings üblicherweise nicht an, um mit dem Vergleichsergebnis zu rechnen, sondern um anschließend im Programm zu verzweigen. Man möchte erreichen, dass das Programm in Abhängigkeit vom Ergebnis des Vergleichs unterschiedlich fortfährt. Wie Sie das erreichen, erfahren Sie im nächsten Abschnitt über den »Kontrollfluss«.

3.5 Kontrollfluss

Bei einem Programm kommt es ganz entscheidend darauf an, in welcher Reihenfolge die einzelnen Anweisungen ausgeführt werden. Üblicherweise werden Anweisungen in der Reihenfolge ihres Vorkommens im Programm ausgeführt. Sie haben aber im Eingangsbeispiel (Divisionsalgorithmus aus der Schule) bereits gesehen, dass es erforderlich ist, Fallunterscheidungen und gezielte Wiederholungen von Anweisungsfolgen zu ermöglichen.

3.5.1 Bedingte Befehlsausführung

Die bedingte Ausführung einer Anweisungsfolge realisieren wir in C durch eine `if`-Anweisung, die die folgende Struktur hat:

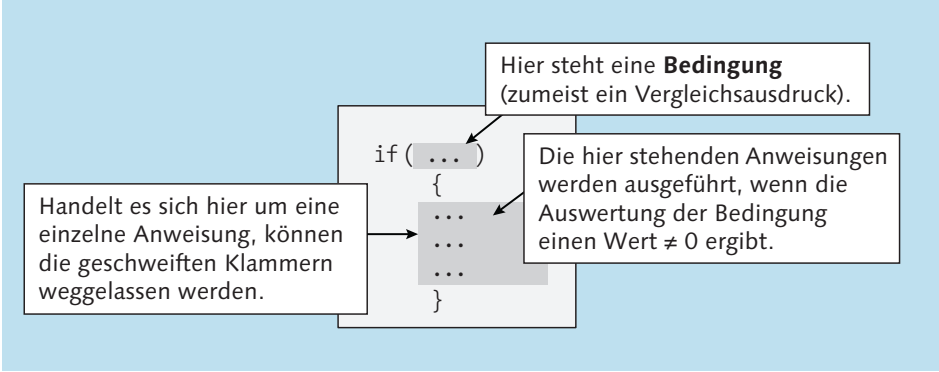


Abbildung 3.3 Bedingte Befehlsausführung

Zum besseren Verständnis betrachten wir einige einfache Beispiele.

Das folgende Codefragment berechnet den Absolutbetrag einer Variablen `a`:

```
if( a < 0)
   a = -a;
```

Wenn der Wert von `a` kleiner als der Wert von `b` ist, dann tausche die Werte von `a` und `b`:

```
if( a < b)
   {
   c = a;
   a = b;
   b = c;
   }
```

Weise der Variablen `max` den größeren der Werte von `a` und `b` zu:

```
max = a;
if( a < b)
   max = b;
```

Mit `else` können wir einem `if`-Ausdruck Anweisungen hinzufügen, die ausgeführt werden sollen, wenn die `if`-Bedingung *nicht* zutrifft. Von der Struktur her sieht das vollständige `if`-Statement dann wie folgt aus:

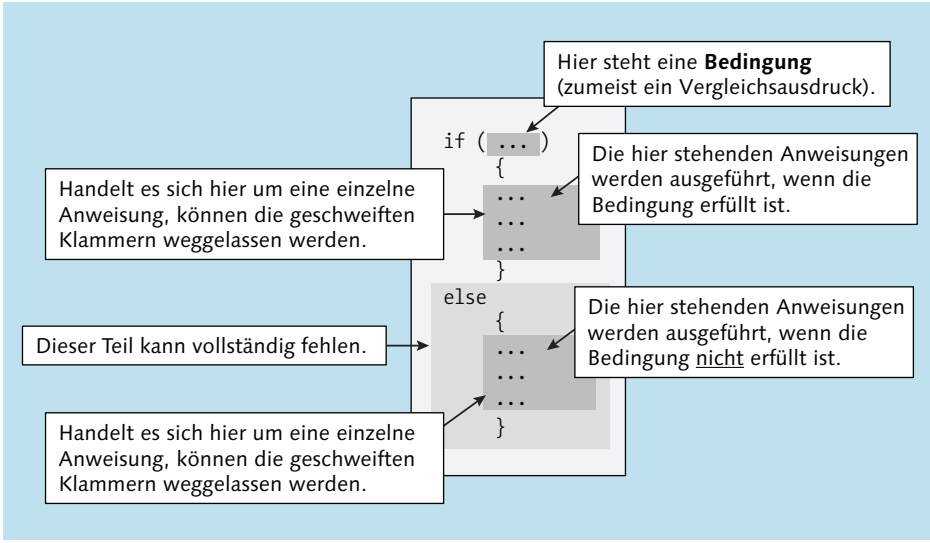


Abbildung 3.4 Die vollständige if-Anweisung

Auch dazu betrachten wir einige einfache Beispiele.

Berechne das Maximum zweier Zahlen `a` und `b`:

```
if( a < b)
   max = b;
else
   max = a;
```

Berechne den Abstand von `a` und `b`:

```
if( a < b)
   abst = b - a;
else
   abst = a - b;
```

Die Prüfung, ob eine Bedingung erfüllt ist, ist letztlich eine Prüfung auf gleich oder ungleich 0. Das heißt, wenn Sie eine Variable auf gleich oder ungleich 0 testen möchten, können Sie die Bedingung vereinfachen:

```
if ( a != 0)
    {
    ...
    }
```

kann auch so ausgedrückt werden:

```
if ( a)
    {
    ...
    }
```

Andersherum kann

```
if ( a == 0)
    {
    ...
    }
```

auch so dargestellt werden:

```
if ( !a)
    {
    ...
    }
```

Da der C-Programmierer mit Buchstaben im Quellcode geizt, favorisiert er zumeist die kürzere Formulierung, aber bleiben Sie zunächst ruhig bei der längeren, wenn Sie den Code dann besser lesen können.

Bei der Verwendung von `if` sollten Sie beachten, dass ein Vergleich auf Gleichheit mit dem doppelten Gleichheitszeichen durchgeführt wird. Das einfache Gleichheitszeichen bedeutet eine Zuweisung. Die Verwechslung des Vergleichs auf Gleichheit mit der Zuweisungsoperation ist einer der »beliebtesten« Anfängerfehler in C. Im folgenden Codefragment

```
if( a = 1)
   b = 5;
```

wird zunächst der Variablen `a` der Wert `1` zugewiesen. Das Ergebnis dieser Zuweisung ist 1, sodass die nachfolgende Zeile (`b = 5`) immer ausgeführt wird. Sagen Sie daher im Beispiel oben nicht »if a *gleich* 1«, sondern »if a *ist gleich* 1«, dann kann Ihnen der Fehler nicht so leicht unterlaufen.

3.5.2 Wiederholte Befehlsausführung

Am Beispiel der Division aus dem ersten Kapitel haben Sie gesehen, dass es erforderlich sein kann, in einem Algorithmus eine bestimmte Folge von Anweisungen wiederholt zu durchlaufen, bis eine bestimmte Situation eingetreten ist. Wir nennen dies eine *Programmschleife*. Versucht man, die Anatomie von Schleifen allgemein zu beschreiben, stößt man auf ein immer wiederkehrendes Muster:

- Es gibt eine Reihe von Dingen, die zu tun sind, bevor man mit der Durchführung der Schleife beginnen kann. Wir nennen dies die *Initialisierung* der Schleife.
- Zunächst muss eine Prüfung durchgeführt werden, ob die Bearbeitung der Schleife abgebrochen oder fortgesetzt werden soll. Wir nennen dies den *Test* auf Fortsetzung der Schleife.
- Bei jedem Schleifendurchlauf werden die eigentlichen Tätigkeiten durchgeführt. Wir nennen dies den *Schleifenkörper*.
- Nach dem Ende eines einzelnen Schleifendurchlaufs müssen gewisse Operationen durchgeführt werden, um den nächsten Schleifendurchlauf vorzubereiten. Wir nennen dies das *Inkrement* der Schleife.

Machen Sie sich dies am Beispiel einer Routineaufgabe klar:

Sie haben Ihre Post erledigt und wollen die Briefe frankieren, bevor Sie sie zum Briefkasten bringen. Dazu stellen Sie zunächst die erforderlichen Hilfsmittel bereit. Sie besorgen sich einen Bogen mit Briefmarken und legen den Stapel der unfrankierten Briefe vor sich auf den Schreibtisch. Das ist die Initialisierung. Bevor Sie nun fortfahren, prüfen Sie, ob der Stapel der Ausgangspost noch nicht abgearbeitet ist. Das ist der Test auf Fortsetzung. Liegen noch Briefe vor Ihnen, treten Sie in den eigentlichen Arbeitsprozess ein. Sie trennen eine Briefmarke ab, befeuchten sie auf der Rückseite und kleben sie auf den obersten Brief des Stapels. Das ist der Schleifenkörper. Nachdem Sie einen Brief frankiert haben, nehmen Sie ihn und legen ihn in den Postausgangskorb. Das ist das Inkrement, mit dem Sie den nächsten Frankiervorgang vorbereiten. Danach setzen Sie die Arbeit mit dem Test fort. Wir fassen Initialisierung, Test und Inkrement unter dem Begriff *Schleifenkopf* zusammen und zeichnen ein Flussdiagramm:

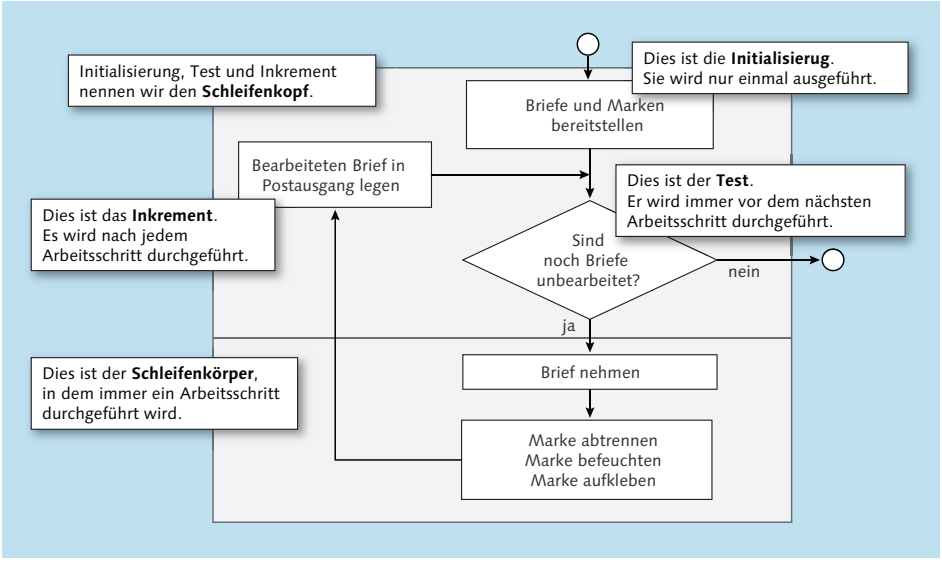


Abbildung 3.5 Flussdiagramm »Briefe frankieren«

In C gibt es ein Sprachelement, das das hier diskutierte Schleifenmuster exakt abbildet. Es handelt sich um die `for`-Anweisung, die sich aus Schleifenkopf und Schleifenkörper zusammensetzt. Der Schleifenkopf enthält drei durch Semikolon getrennte Ausdrücke, die die Abarbeitung der Schleife steuern:

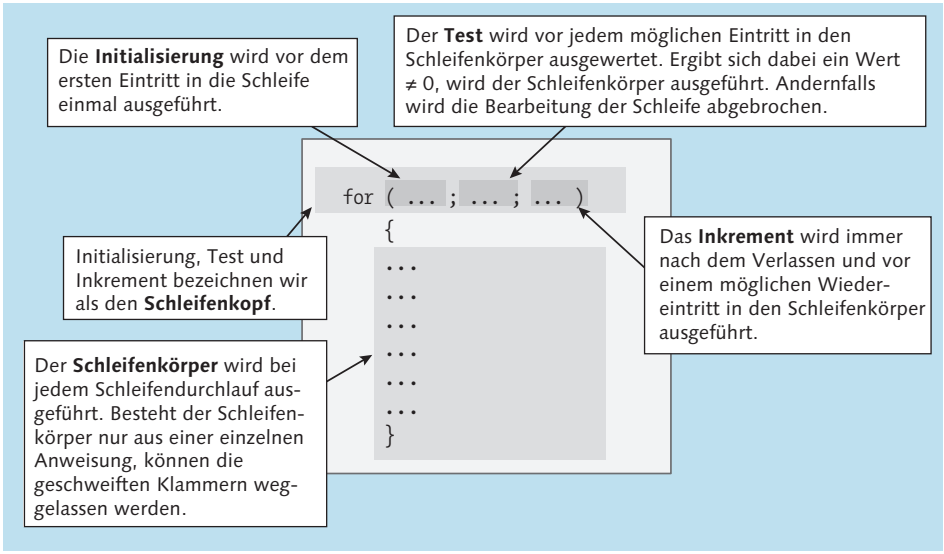


Abbildung 3.6 Die for-Anweisung

Der Test dient letztlich dazu, die Schleife abzubrechen. Deshalb spricht man oft etwas oberflächlich von einer »Abbruchbedingung«. Dies suggeriert, dass die Schleife abgebrochen wird, wenn der Test positiv ausfällt. Es ist aber genau umgekehrt: Die Schleife wird abgebrochen, wenn der Test negativ ausfällt, bzw. fortgesetzt, wenn der Test positiv ausfällt. In diesem Sinne handelt es sich also bei dem Test um eine »Weitermachbedingung«. Prägen Sie sich daher den irreführenden Begriff »Abbruchbedingung« und das damit verbundene Bild erst gar nicht ein.

Eine der häufigsten Anwendungen von Schleifen ist die sogenannte *Zählschleife*. Hier wird die Anzahl der Schleifendurchläufe über eine Zählvariable gesteuert. Konkret kann das etwa so aussehen:

```
int i, summe;

summe = 0;
for( i = 1; i <= 100; i = i + 1)
   summe = summe + i;
```

In dieser Schleife werden die Zahlen von 1 bis 100 aufsummiert. Das kann man natürlich auch rückwärts machen:

```
summe = 0;
for( i = 100; i > 0; i = i - 1)
   summe = summe + i;
```

Der Endwert in `summe` ist jeweils der gleiche.

Man kann auch mehrere Anweisungen durch Komma getrennt in die Initialisierung oder das Inkrement der Schleife aufnehmen. In unserem Beispiel nehmen wir die Initialisierung der Summe mit in den Schleifenkopf auf:

```
for(summe = 0, i = 1; i <= 100; i++)
   summe = summe + i;
```

Anstelle von `i = i + 1` habe ich hier die Kurzform `i++` verwendet.

In der folgenden Schleife wird eine Variable `a` von `1` ausgehend hoch- und eine Variable `b` von `100` ausgehend heruntergezählt, solange `a` dabei kleiner als `b` bleibt:

```
for(a = 1, b = 100; a < b; a++, b--)
    ...;
```

Einzelne Felder im Schleifenkopf können auch leer gelassen werden. Ist die Initialisierung oder das Inkrement leer, wird dort nichts gemacht. Ein leer gelassenes Feld für den Test führt dazu, dass der Test immer positiv ausfällt. Achtung, beim folgenden Beispiel handelt es sich um eine Endlosschleife:

```
for( ; ; )
    ;
```

Eine solche Endlosschleife zu programmieren ist natürlich Unsinn. Trotzdem gibt es Situationen, in denen man den Test weglässt, da man den Ablauf der Schleife auch aus dem Schleifenkörper heraus steuern kann. Um das zu verstehen, kehren wir noch einmal zu dem einführenden Beispiel zurück und diskutieren zwei Sonderfälle, die beim Frankieren der Briefe auftreten können:

1. Sie stellen fest, dass oben auf dem Stapel ein Brief liegt, der an jemanden in der Nachbarschaft gerichtet ist. Sie beschließen, das Porto zu sparen und den Brief selbst vorbeizubringen. Dazu überspringen Sie die weitere Bearbeitung dieses Briefes und legen den Brief unfrankiert in den Postausgang. Sie fahren dann mit der Abarbeitung des Stapels fort.
2. Sie stellen fest, dass Ihnen die Briefmarken ausgegangen sind. Es bleibt Ihnen nichts anderes übrig, als die Bearbeitung der Schleife vorzeitig abzubrechen.

Wir nehmen diese beiden Fälle in das Flussdiagramm auf:

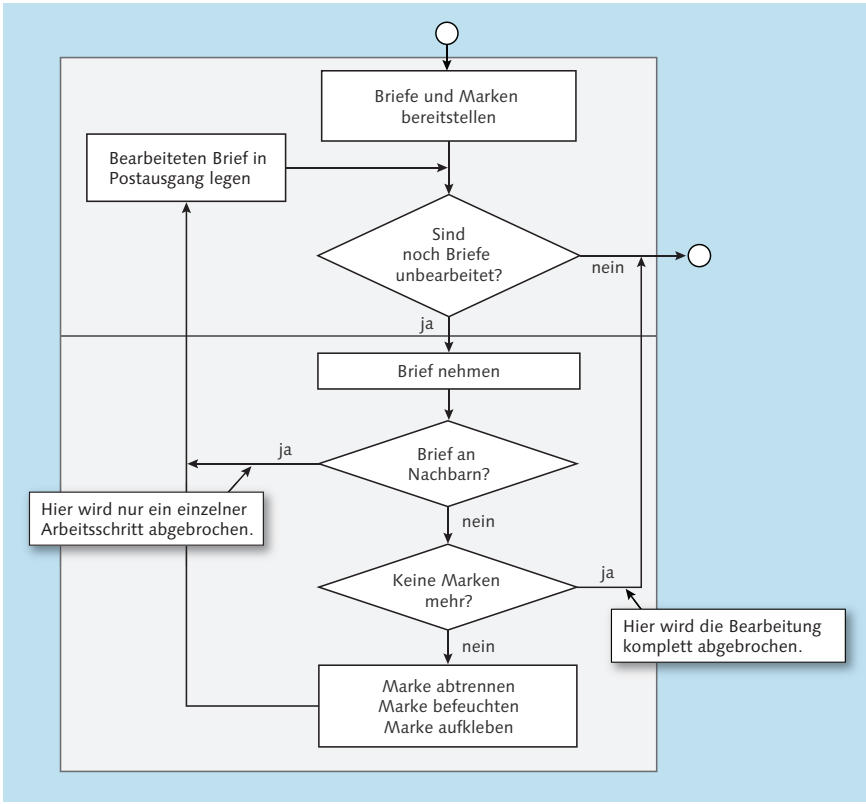


Abbildung 3.7 Das erweiterte Flussdiagramm

Um auf Sonderfälle sinnvoll zu reagieren, müssen wir aus dem Schleifenkörper heraus in die Schleifensteuerung eingreifen. Das ist in C durch eine `break`- oder eine `continue`-Anweisung möglich:

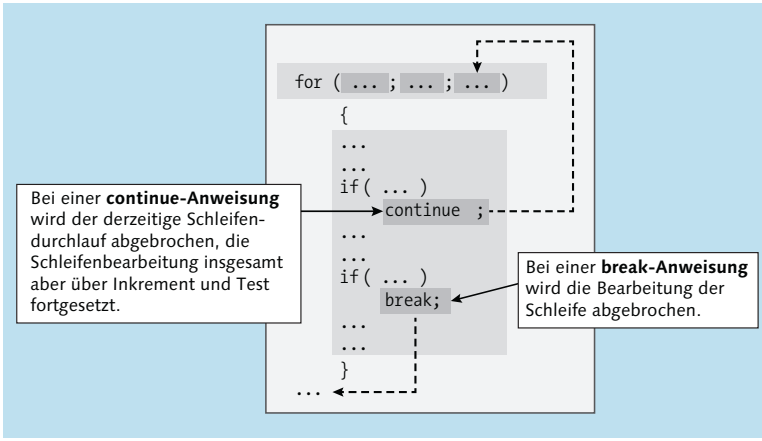


Abbildung 3.8 Schleifensteuerung innerhalb der for-Anweisung

Beachten Sie noch einmal den Unterschied! Durch `continue` wird nur der aktuelle Schleifendurchlauf abgebrochen, die Schleife insgesamt jedoch über Inkrement und Test fortgesetzt. Durch `break` wird dagegen die Schleife sofort abgebrochen. Natürlich kann es mehrere `break`- oder `continue`-Anweisungen in beliebiger Reihenfolge in einer Schleife geben. Solche Anweisungen werden aber immer unter einer Bedingung stehen. Ein unbedingtes `break` oder `continue` ist nicht sinnvoll, da der nachfolgende Code nie erreicht würde.

In die oben konstruierte Schleife zum Aufsummieren aller Zahlen zwischen 1 und 100 bauen wir jetzt zusätzlich eine `continue`-Anweisung ein, die dafür sorgt, dass alle durch 7 teilbaren Zahlen bei der Summation übersprungen werden:

```
for(summe = 0, i = 1; i <= 100; i = i + 1)
   {
   if( i%7 == 0)
      continue;
   summe = summe + i;
   }
```

Beachten Sie, dass wir jetzt die geschweiften Klammern benötigen, da wir mehr als eine Anweisung im Schleifenkörper haben. Was berechnet das Programm, wenn Sie die geschweiften Klammern weglassen? Wenn Sie nicht sicher sind, probieren Sie es aus.

Wir wollen zusätzlich noch einen harten Schleifenabbruch in unser Programm einbauen:

```
for(summe = 0, i = 1; i <= 100; i = i + 1)
    {
    if( i%7 == 0)
        continue;
    summe = summe + i;
    if( summe > 1000)
        break;
    }
```

Jetzt wird die Schleife sofort verlassen, wenn sich in `summe` ein Wert größer als 1000 ergibt. Wissen Sie, bei welchem Wert von `i` die Schleife jetzt abgebrochen wird und welchen Wert die Variable `summe` dann hat? Implementieren Sie das Programm, um es herauszufinden.

Ganz selbstverständlich haben wir in unserem Beispiel eine Bedingung in eine Schleife eingebaut. Das zeigt, dass man offensichtlich die verschiedenen Kontroll-

strukturen ineinander schachteln kann. Diese Beobachtung wollen wir im folgenden Abschnitt noch etwas vertiefen.

Für die Testbedingung im Schleifenkopf gilt das bereits zur `if`-Bedingung Gesagte. Bei einer Prüfung auf gleich oder ungleich 0 kann man vereinfachen:

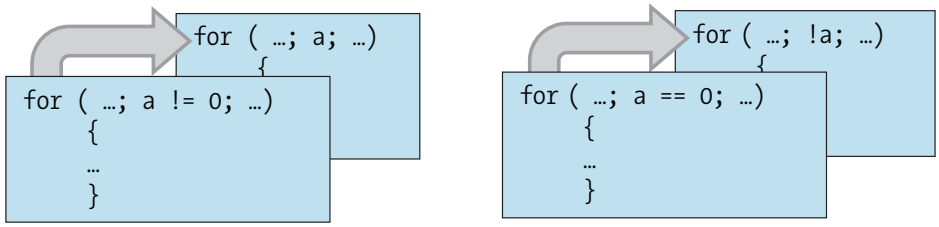


Abbildung 3.9 Vereinfachung der Testbedingung

Insbesondere muss auch hier wieder auf den Unterschied zwischen Test auf Gleichheit (`==`) und Zuweisung (`=`) hingewiesen werden.

Wenn eine Schleife keine Initialisierung und kein Inkrement benötigt, können Sie anstelle einer `for`- auch eine `while`-Anweisung verwenden:

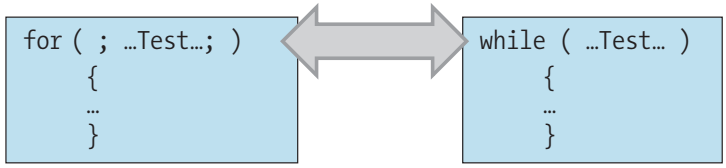


Abbildung 3.10 Ersatz der for- durch eine while-Anweisung

Die Anweisungen `break` und `continue` können bei `while` genauso wie bei `for` verwendet werden. Im Grunde genommen ist `while` entbehrlich, da es ein Spezialfall von `for` ist. Umgekehrt könnte auch `for` vollständig durch `while` nachgebildet werden. Das ist aber im Sinne einer guten Lesbarkeit der Programme nicht immer sinnvoll, da Initialisierung und Inkrement der Schleife nicht mehr explizit ausgewiesen und im restlichen Programm »versteckt« sind. Das kann bei Programmänderungen oder -erweiterungen zu Problemen führen.

3.5.3 Verschachtelung von Kontrollstrukturen

Kontrollstrukturen können beliebig ineinander eingesetzt werden. Möglich sind z. B.:

- ein `if` in einem `if`
- ein `if` in einem `for`
- ein `for` in einem `for`

- ein `for` in einem `if`
- ein `if` in einem `for` in einem `for`
- ein `for` in einem `if` in einem `for` in einem `if`

Als Beispiel betrachten wir ein Programm, das das »kleine Einmaleins« durch zwei ineinander geschachtelte Zählschleifen berechnet:

```
for( i = 1; i <= 10; i = i + 1)
   {
   for( k = 1; k <= 10; k = k + 1)
      produkt = i*k;
   }
```

Die Variable `i` durchläuft in der äußeren Schleife die Werte von 1 bis 10. Für jeden Wert von `i` durchläuft dann die Variable `k` in der inneren Schleife ebenfalls die Werte von 1 bis 10. Insgesamt wird damit die Berechnung in der inneren Schleife 100-mal für alle möglichen Kombinationen von `i` und `k` ausgeführt.

Das folgende Programm berechnet das Produkt nur, wenn beide Faktoren gerade sind:

```
for( i = 1; i <= 10; i = i + 1)
    {
    if( i%2 == 0)
        {
        for( k = 1; k <= 10; k = k + 1)
            {
            if( k%2 == 0)
                produkt = i*k;
            }
        }
    }
```

Nur wenn `i` gerade ist, wird jetzt in die innere Schleife über `k` eingetreten, und dort wird dann das Produkt nur dann berechnet, wenn `k` ebenfalls gerade ist.

Beachten Sie, dass in diesem Beispiel alle geschweiften Klammern und auch die Einrückungen eigentlich überflüssig sind. Zusätzlich gesetzte Klammern und einheitliche Einrückungen verbessern aber die Lesbarkeit des Programms.

Das oben dargestellte Programm berechnet zwar das kleine Einmaleins, aber die Ergebnisse verfliegen im luftleeren Raum. Sinnvoll wäre es, immer dann, wenn man ein neues Ergebnis ermittelt hat, dieses auf dem Bildschirm auszugeben. Damit werden wir uns im nächsten Abschnitt beschäftigen.

3.6 Elementare Ein- und Ausgabe

Um erste einfache Programme schreiben zu können, müssen Sie Werte von der Tastatur in Variablen einlesen und Werte von Variablen auf dem Bildschirm ausgeben können. Es geht dabei nicht darum, komplexe Interaktionen mit dem Benutzer abzuwickeln. Es reicht, wenn Sie einige wenige Benutzereingaben in Ihre Programme hinein- und einige wenige Ergebnisse aus Ihren Programmen herausbekommen. Dementsprechend spartanisch sind die Methoden, die ich Ihnen hier vorstellen werde. Da Computernutzer heutzutage von opulenten Bedienoberflächen verwöhnt sind, wird das vielleicht enttäuschend für Sie sein. Aber vielleicht lenkt gerade diese Genügsamkeit Ihren Blick auf das Wesentliche.

C hat keine Sprachelemente für Ein- oder Ausgabe. Ein- und Ausgabe werden nicht durch die Sprache selbst, sondern durch sogenannte *Funktionen* erledigt. Die hinter den Funktionen stehenden Konzepte werde ich Ihnen später vorstellen. Sie können Funktionen aber auch verwenden, ohne genau verstanden zu haben, was bei der Verwendung »unter der Haube« passiert. Sollten bei den folgenden Erklärungen noch Fragen offen bleiben, versichere ich Ihnen, dass ich diese Fragen später ausführlich beantworten werde.

3.6.1 Bildschirmausgabe

Um einen Text auf dem Bildschirm auszugeben, verwenden wir die Funktion `printf` und schreiben:

```
printf( "Dieser Text wird ausgegeben\n");
```

Der auszugebende Text wird in doppelte Hochkommata eingeschlossen. Die am Ende des Textes stehende Zeichenfolge `\n` erzeugt einen Zeilenvorschub. Vergessen Sie nicht das Semikolon am Ende der Zeile!

In den auszugebenden Text können wir Zahlenwerte einstreuen, indem wir als Platzhalter für die fehlenden Zahlenwerte eine sogenannte *Formatanweisung* in den Text einfügen. Eine solche Formatanweisung besteht aus einem Prozentzeichen, gefolgt von dem Buchstaben d (für Dezimalwert) oder f (für Gleitkommawert) – also `%d` oder `%f`. Die zugehörigen Werte werden dann als Konstanten oder Variablen durch Kommata getrennt hinter dem Text angefügt.

Das Programmfragment

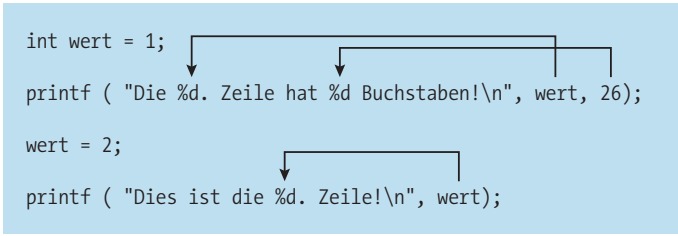


Abbildung 3.11 Programmfragment zur Ausgabe

führt zu der Ausgabe:

```
Die 1. Zeile hat 26 Buchstaben!
Dies ist die 2. Zeile!
```

Der auszugebende Wert kann auch ohne Verwendung einer Variablen direkt dort berechnet werden, wo er benötigt wird:

```
printf( "Ergebnis = %d\n", 3*a + b);
```

Der Ausdruck `3*a+b` wird zunächst vollständig ausgewertet, und das Ergebnis wird an der durch `%d` markierten Stelle in die Ausgabe eingefügt.

Zur Ausgabe von Gleitkommazahlen verwendet man die Formatanweisung `%f`. Im folgenden Beispiel

```
float preis;

preis = 10.99;
printf( "Die Ware kostet %f EURO\n", preis);
```

erhalten wir die Ausgabe:

```
Die Ware kostet 10.99 EURO
```

Wichtig ist, dass die Formatanweisung exakt zum Typ des auszugebenden Werts passt – also `%d` bei ganzen Zahlen und `%f` bei Gleitkommazahlen.

Wir wollen unser Beispiel zur Berechnung des kleinen Einmaleins jetzt mit einer Ausgabe ausstatten:

```
for( i = 1; i <= 10; i = i + 1)
   {
   for( k = 1; k <= 10; k = k + 1)
      {
```

```
      produkt = i*k;
      printf( "%d mal %d ist %d\n", i, k, produkt);
      }
   printf( "\n");
   }
```

Das Programm gibt jetzt das kleine Einmaleins auf dem Bildschirm aus und erzeugt nach jedem Zehnerpäckchen einen zusätzlichen Zeilenvorschub. Das sieht so aus:

```
2 mal 6 ist 12
2 mal 7 ist 14
2 mal 8 ist 16
2 mal 9 ist 18
2 mal 10 ist 20

3 mal 1 ist 3
3 mal 2 ist 6
3 mal 3 ist 9
3 mal 4 ist 12
3 mal 5 ist 15
3 mal 6 ist 18
3 mal 7 ist 21
3 mal 8 ist 24
3 mal 9 ist 27
3 mal 10 ist 30

4 mal 1 ist 4
4 mal 2 ist 8
4 mal 3 ist 12
```

3.6.2 Tastatureingabe

Eine oder mehrere ganze Zahlen lesen wir mit der Funktion `scanf` von der Tastatur ein.

```
int zahl1, zahl2;

printf ( "Bitte geben Sie zwei Zahlen ein: ");

scanf ( "%d %d", &zahl1, &zahl2);
printf ( "Sie haben %d und %d eingegeben\n", zahl1, zahl2);
```

Abbildung 3.12 Einlesen von Werten

Beim Einlesen müssen Variablen angegeben werden, denen die Werte zugewiesen werden sollen. Wir stellen dazu dem Variablennamen ein & voran. Die exakte Bedeutung des &-Zeichens können Sie im Moment noch nicht verstehen, sie wird später erklärt. Lassen Sie das & jedoch nicht weg, auch wenn es Ihnen an dieser Stelle unmotiviert erscheint.

Der zum oben dargestellten Programm gehörende Bildschirmdialog sieht bei entsprechenden Benutzereingaben wie folgt aus:

```
Bitte geben Sie zwei Zahlen ein: 123 456
Sie haben 123 und 456 eingegeben!
```

Für die Eingabe von Gleitkommazahlen verwenden Sie dann natürlich Gleitkommavariablen und die Formatanweisung %f.

Wir können das Programm zur Ausgabe des kleinen Einmaleins jetzt so erweitern, dass die Bereiche, in denen das kleine Einmaleins berechnet werden soll, durch den Benutzer festgelegt werden. Hier sehen Sie das vollständige Programm dazu:

```
void main()
   {
   int i, k;
   int maxi, maxk;
   int produkt;

   printf( "Bitte maxi eingeben: ");
   scanf( "%d", &maxi);
   printf( "Bitte maxk eingeben: ");
   scanf( "%d", &maxk);

   for( i = 1; i <= maxi; i = i + 1)
      {
      for( k = 1; k <= maxk; k = k + 1)
         {
         produkt = i*k;
         printf( "%d mal %d ist %d\n", i, k, produkt);
         }
      printf( "\n");
      }
   }
```

Listing 3.7 Das kleine Einmaleins

Der Benutzer wird aufgefordert, Maximalwerte für `i` und `k` einzugeben. Die eingegebenen Werte werden dann in den Schleifen verwendet, um die zulässigen Werte für `i` und `k` nach oben zu begrenzen. Das folgende Bild zeigt einen möglichen Programmlauf:

```
Bitte maxi eingeben: 3
Bitte maxk eingeben: 5
1 mal 1 ist 1
1 mal 2 ist 2
1 mal 3 ist 3
1 mal 4 ist 4
1 mal 5 ist 5

2 mal 1 ist 2
2 mal 2 ist 4
2 mal 3 ist 6
2 mal 4 ist 8
2 mal 5 ist 10

3 mal 1 ist 3
3 mal 2 ist 6
3 mal 3 ist 9
3 mal 4 ist 12
3 mal 5 ist 15
```

Die Formatanweisung in `scanf` kann neben den %-Anweisungen auch zusätzliche Zeichen enthalten. Zum Beispiel

```
int zahl;
scanf( "ABC%dxyz", &zahl);
```

In diesem Fall erwartet `scanf` genau das in der Formatanweisung angegebene Muster in der Eingabe, also `ABC`, gefolgt von einer Zahl, die der Variablen `zahl` zugewiesen wird, und dann wiederum gefolgt von `xyz`. Auf diese Weise können Sie Ihre Eingaben aus einem komplexeren Kontext »herauspicken« oder den Eingabetext in seine Einzelbestandteile zerlegen. Diese strenge Auslegung der Eingabe verlangt allerdings vom Benutzer, dass er die Zeichen genauso eingibt, wie im Formatstring vorgegeben. Eine Abweichung führt zu Fehleingaben oder zu scheinbar unmotiviertem Warten auf weitere Eingaben. Wir wollen das hier nicht weiter diskutieren, da diese Art der Eingabe bei modernen Softwaresystemen mit grafischer Benutzeroberfläche nicht verwendet wird.

3.6.3 Kommentare und Layout

C-Programme können durch Kommentare verständlicher gestaltet werden. Es gibt zwei Arten, ein Programm in C zu kommentieren:

- Einzeilige Kommentare beginnen mit // und erstrecken sich dann bis zum Ende der Zeile.
- Mehrzeilige Kommentare beginnen mit /* und enden mit */.

Kommentare werden beim Übersetzen des Programms einfach ignoriert.

```
/*
** Variablendefinitionen
*/
int zahl1; // Dies ist eine Zahl

/*
** Programmcode
*/
zahl1 = 123; // Der Wert ist jetzt 123
```

Setzen Sie Kommentare nur dort ein, wo sie wirklich etwas zum Programmverständnis beitragen! Vermeiden Sie Plattitüden wie im Beispiel oben!

Die in diesem Buch als Beispiele vorgestellten Programme enthalten in der Regel keine Kommentare. Das liegt daran, dass alle Beispielprogramme im umgebenden Text ausführlich besprochen werden. Lassen Sie sich durch das Fehlen von Kommentaren nicht zu der irrigen Annahme verleiten, dass Kommentare in C-Programmen überflüssig sind.

Das Layout des Programmtextes können Sie, von den #-Anweisungen, die immer am Anfang einer Zeile stehen müssen, einmal abgesehen, mit Leerzeichen, Zeilenumbrüchen, Seitenvorschüben und Tabulatorzeichen relativ frei gestalten. Ein einheitliches, klar gegliedertes Layout erhöht die Lesbarkeit und damit auch die Pflegbarkeit eines Programms. Die Frage nach einer einheitlichen und verbindlichen Gestaltung des Programmcodes gewinnt insbesondere dann an Bedeutung, wenn Software von mehreren Programmierern im Team erstellt wird und die Notwendigkeit besteht, dass ein und derselbe Code von verschiedenen Entwicklern bearbeitet wird. Viele Unternehmen haben daher Codier-Richtlinien aufgestellt, und die Entwickler sind gehalten, sich an diesen Vorgaben zu orientieren. Ich werde Ihnen an einigen Stellen Empfehlungen über einen »guten« Gebrauch der durch C bzw. C++ zur Verfügung gestellten Sprachmittel geben. Eine vollständige Bereitstellung von Codier-Richtlinien finden Sie in diesem Buch jedoch nicht.

So sollten Sie es jedenfalls nicht machen:

```
void main() { int i; int k; int maxi; int maxk; int produkt; printf(
"Bitte maxi eingeben: "); scanf( "%d", &maxi); printf(
"Bitte maxk eingeben: "); scanf( "%d", &maxk);  for( i = 1; i <= maxi; i =
 i + 1) { for( k = 1; k <= maxk; k = k + 1) { produkt = i*k; printf(
"%d mal %d ist %d\n", i, k, i*k); } printf( "\n"); } }
```

Es gibt übrigens einen hochinteressanten Wettbewerb (International Obfuscated C Code Contest) im Internet, bei dem es darum geht, möglichst kreativ C-Programme zu erstellen, die ihre wahre Funktion verschleiern. Das ist sozusagen das genaue Gegenteil dessen, was von einem seriösen Programmierer erwartet wird. Im Rahmen dieses Wettbewerbs sind im Laufe der Jahre kleine Kunstwerke entstanden, die nur mit perfekten C-Kenntnissen analysiert und verstanden werden können. Im Moment ist es vielleicht noch etwas zu früh für Sie, sich an solchen Programmen zu versuchen, aber wenn Sie später einmal testen wollen, ob Sie C wirklich verstanden haben, finden Sie dort echte Herausforderungen.

3.7 Beispiele

Es wird Sie vielleicht überraschen, aber mit dem, was Sie bisher gelernt haben, können Sie bereits alles programmieren, was man überhaupt nur programmieren kann. Im Grunde genommen könnten Sie dieses Buch jetzt zuklappen und den Rest vergessen. Ich hoffe natürlich, dass Sie weiterlesen, denn wenn Sie jetzt aufhören, wäre das so, als würde ich Sie mit einem Teelöffel vor ein riesiges Schwimmbecken stellen und sagen, dass Sie jetzt alles haben, was Sie benötigen, um das Becken zu leeren.

Es ist noch ein weiter Weg zum Ziel, das ja professionelle Programmierung heißt. Aber ein wichtiges Etappenziel haben Sie erreicht. Um zu sehen, was Sie bereits können, finden Sie hier einige Beispiele.

3.7.1 Das erste Programm

Was liegt näher, als mit Ihren frisch erworbenen Programmierkenntnissen zu versuchen, den Algorithmus zur Division aus dem ersten Kapitel zu realisieren? Erinnern Sie sich an das zugehörige Flussdiagramm, das Sie jetzt in ein C-Programm umsetzen können.

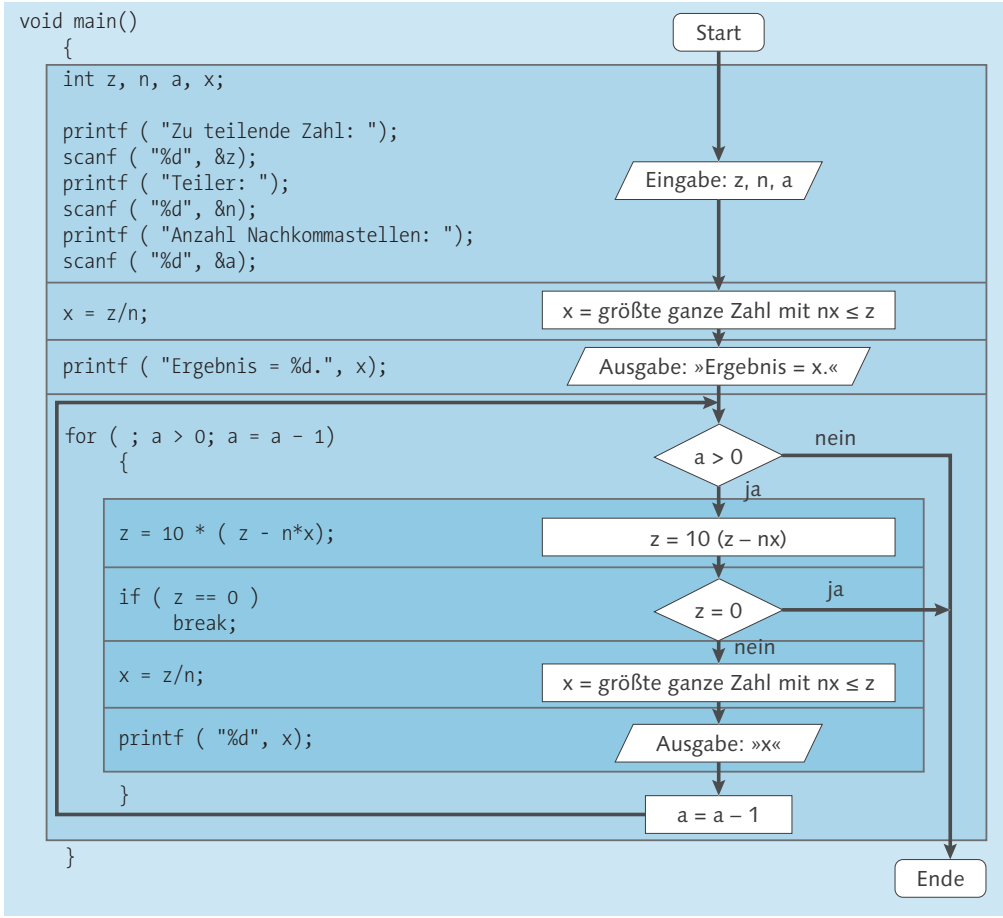


Abbildung 3.13 Flussdiagramm der schriftlichen Division

Die Schleife wird so lange ausgeführt, wie Ziffern zu berechnen sind – es sei denn, dass der Divisionsrest `0` wird. Dann wird die Schleife vorzeitig durch die `break`-Anweisung beendet.

Wir testen das Programm mit unserem Standardfall (`84:16`)

```
Zu teilende Zahl: 84
Teiler: 16
Anzahl Nachkommastellen: 4
Ergebnis = 5.25
```

und mit einem Testfall, bei dem das Abbruchkriterium über die Stellenzahl zum Zuge kommt (`100:7`):

```
Zu teilende Zahl: 100
Teiler: 7
Anzahl Nachkommastellen: 6
Ergebnis = 14.285714
```

Das Programm arbeitet einwandfrei.

3.7.2 Das zweite Programm

Wir betrachten ein einfaches Spiel, bei dem eine Kugel durch eine Reihe von Weichen (`weiche1` bis `weiche4`) zu einem von zwei möglichen Ausgängen fällt:

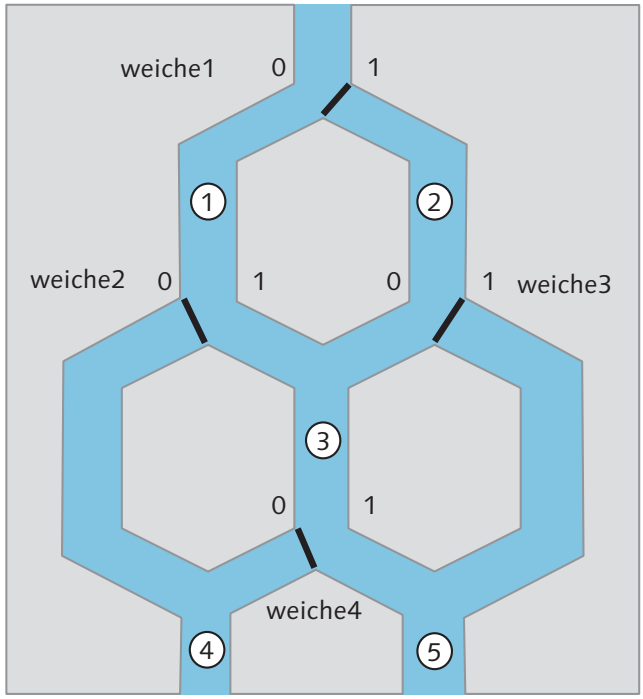


Abbildung 3.14 Das Kugelspiel

Die möglichen Positionen der Kugel auf dem Weg zu einem der Ausgänge sind in Abbildung 3.14 fortlaufend von 1 bis 5 nummeriert. Die Weichen sind so konstruiert, dass sie beim Passieren einer Kugel umschlagen und auf diese Weise die nächste Kugel in die entgegengesetzte Richtung lenken. Die Frage, an welchem Ausgang die Kugel das System verlässt, können wir über eine Reihe geschachtelter Verzweigungen beantworten, wenn wir den Weg einer Kugel durch das System nachvollziehen.

Wir modellieren die Problemlösung zunächst durch ein Flussdiagramm, in dessen Struktur man das Spiel direkt wiedererkennt:

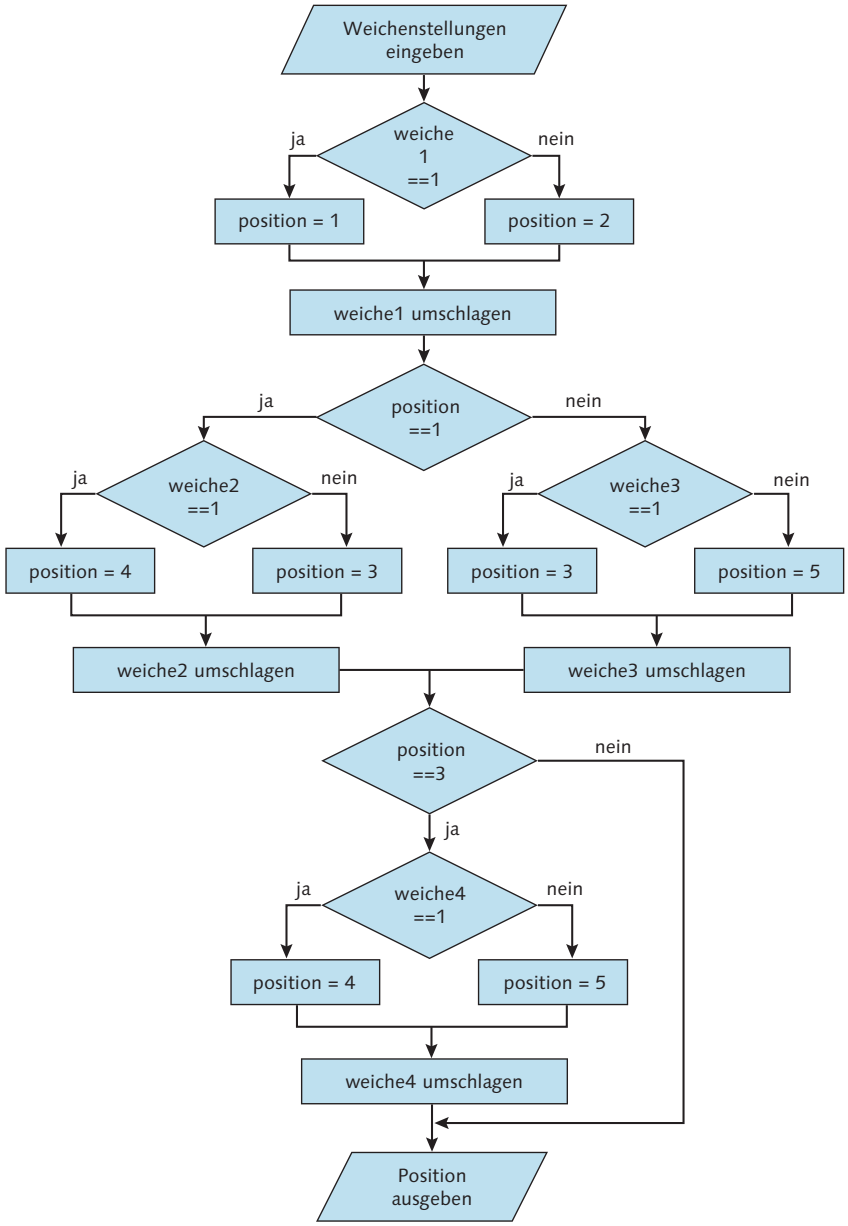


Abbildung 3.15 Flussdiagramm des Kugelspiels

Dieses Flussdiagramm setzen wir dann in C-Code um. Zum Programmstart lassen wir den Benutzer die Anfangsstellung der vier Weichen eingeben, dann läuft der Algorithmus so ab, wie im Flussdiagramm vorgegeben:

```
void main()
   {
   int weiche1, weiche2, weiche3, weiche4;
   int position;

   printf( "Bitte geben Sie die Weichenstellungen ein: ");
   scanf( "%d %d %d %d", &weiche1, &weiche2, &weiche3, &weiche4);

   if( weiche1 == 1)
      position = 1;
   else
      position = 2;
   weiche1 = 1 - weiche1;
   if( position == 1)
      {
      if( weiche2 == 1)
         position = 4;
      else
         position = 3;
      weiche2 = 1 - weiche2;
      }
   else
      {
      if( weiche3 == 1)
         position = 3;
      else
         position = 5;
      weiche3 = 1 - weiche3;
      }
   if( position == 3)
      {
      if( weiche4 == 1)
         position = 4;
      else
         position = 5;
      weiche4 = 1 - weiche4;
      }
   printf( "Auslauf: %d, ", position);
   printf( "neue Weichenstellung %d %d %d %d\n",
                           weiche1, weiche2, weiche3, weiche4);
    }
```

Listing 3.8 Implementierung des Kugelspiels

Das Umschlagen der Weichen realisieren wir durch `weiche = 1 – weiche`. Diese Anweisung bewirkt, dass der Wert von `weiche` immer zwischen 0 und 1 hin- und herschaltet.

Und so läuft das Programm aus Benutzersicht ab:

```
Bitte geben Sie die Weichenstellungen ein: 1 0 1 0
Auslauf: 5, neue Weichenstellung 0 1 1 1
```

Um mehrere Kugeln durch das System laufen zu lassen, müssen wir die Anzahl der gewünschten Kugeln erfragen und den einzelnen Durchlauf in eine Schleife einpacken. Dazu dienen die folgenden Erweiterungen:

```
void main()
    {
    int weiche1, weiche2, weiche3, weiche4;
    int position;
    int kugeln;

    printf( "Bitte geben Sie die Weichenstellungen ein: ");
    scanf( "%d %d %d %d", &weiche1, &weiche2, &weiche3, &weiche4);
    printf( "Bitte geben Sie die Anzahl der Kugeln ein: ");
    scanf( "%d", &kugeln);

    for( ; kugeln > 0; kugeln = kugeln - 1)
        {
        ... wie bisher ...
        }
    }
```

Listing 3.9 Erweiterung des Kugelspiels

Und so läuft das erweiterte Programm ab:

```
Bitte geben Sie die Weichenstellungen ein: 0 1 0 1
Bitte geben Sie die Anzahl der Kugeln ein: 5
Auslauf: 5, neue Weichenstellung 1 1 1 1
Auslauf: 4, neue Weichenstellung 0 0 1 1
Auslauf: 4, neue Weichenstellung 1 0 0 0
Auslauf: 5, neue Weichenstellung 0 1 0 1
Auslauf: 5, neue Weichenstellung 1 1 1 1
```

3.7.3 Das dritte Programm

Für unser drittes Programm stellen wir uns die folgende Programmieraufgabe:

Der Benutzer soll eine von ihm vorab festgelegte Anzahl von Zahlen eingeben. Das Programm summiert unabhängig voneinander die positiven und die negativen Eingaben und gibt am Ende die Summe der negativen Eingaben, die Summe der positiven Eingaben und die Gesamtsumme aus.

In einem konkreten Beispiel soll das Programm so ablaufen, dass zunächst im Dialog mit dem Benutzer alle erforderlichen Eingaben erfragt werden:

```
Wie viele Zahlen sollen eingegeben werden: 8
1. Zahl: 1
2. Zahl: 2
3. Zahl: -5
4. Zahl: 4
5. Zahl: 5
6. Zahl: -8
7. Zahl: 3
8. Zahl: -7
```

Anschließend werden die gewünschten Berechnungsergebnisse ausgegeben:

```
Die Summe aller positiven Eingaben ist: 15
Die Summe aller negtiven Eingaben ist: -20
Die Gesamtsumme ist: -5
```

Zur Realisierung nehmen wir unseren Standardprogrammrahmen und ergänzen die gewünschte Funktionalität:

```
    # include <stdio.h>
    # include <stdlib.h>

    void main()
        {
A       int anzahl;
        int z;
        int summand;
        int psum;
        int nsum;
B       printf( "Wie viele Zahlen sollen eingegeben werden: ");
        scanf( "%d", &anzahl);
        fflush( stdin);
```

```
C       psum = 0;
        nsum = 0;

D       for( z = 1; z <= anzahl; z = z + 1)
           {
E           printf( "%d. Zahl: ", z);
            scanf( "%d", &summand);

F           if( summand > 0)
                psum = psum + summand;
            else
                nsum = nsum + summand;
           }
G       printf( "Summe aller positiven Eingaben: %d\n", psum);
        printf( "Summe aller negativen Eingaben: %d\n", nsum);
        printf( "Gesamtsumme: %d\n", psum + nsum);
        }
```

Listing 3.10 Das dritte Programm

Weil dies eines unserer ersten Programme ist, wollen wir alle Teile noch einmal intensiv betrachten und diskutieren:

Bereich A: Hier werden die benötigten Variablen definiert. Alle Variablen sind ganzzahlig und werden in der folgenden Bedeutung verwendet:

- `anzahl` ist die vom Benutzer gewählte Zahl der Eingaben.
- `z` ist die Kontrollvariable für die Zählschleife.
- `summand` ist die vom Benutzer aktuell eingegebene Zahl.
- `psum` ist die jeweils aufgelaufene Summe der positiven Eingaben.
- `nsum` ist die jeweils aufgelaufene Summe der negativen Eingaben.

Bereich B: Hier wird der Benutzer zunächst aufgefordert, die gewünschte Anzahl einzugeben. Dann wird die Benutzereingabe in die Variable `anzahl` übertragen. Vergessen Sie nicht das `&`-Zeichen vor der einzulesenden Variablen!

Bereich C: Die zur Summenbildung verwendeten Variablen (`psum`, `nsum`) werden mit `0` initialisiert.

Zeile D: In einer Schleife werden für `z = 1, 2,...,anzahl` jeweils die Unterpunkte E–G ausgeführt.

Bereich E: Der Benutzer wird aufgefordert, die nächste Zahl einzugeben, und diese Zahl wird der Variablen `summand` zugewiesen.

Bereich F: Wenn die vom Benutzer eingegebene Zahl (`summand`) größer als 0 ist, wird `psum` entsprechend erhöht, andernfalls wird `nsum` entsprechend verkleinert.

Bereich G: Die gewünschten Ergebnisse `psum`, `nsum` und `psum+nsum` werden ausgegeben.

3.8 Aufgaben

A 3.1 Machen Sie sich mit Editor, Compiler und Linker Ihrer Entwicklungsumgebung vertraut, indem Sie die Programme dieses Kapitels eingeben und zum Laufen bringen!

A 3.2 Schreiben Sie ein Programm, das zwei ganze Zahlen von der Tastatur einliest und anschließend deren Summe, Differenz, Produkt, den Quotienten und den Divisionsrest auf dem Bildschirm ausgibt!

```
1. Zahl: 10
2. Zahl: 4

Summe     10 + 4 = 14
Differenz 10 - 4 = 6
Produkt   10*4 = 40
Quotient  10/4 = 2 Rest 2
```

Was passiert, wenn man versucht, durch 0 zu dividieren?

A 3.3 Erstellen Sie ein Programm, das unter Verwendung der in Aufgabe 1.3 formulierten Regeln berechnet, ob eine vom Benutzer eingegebene Jahreszahl ein Schaltjahr bezeichnet oder nicht!

A 3.4 Erstellen Sie ein Programm, das zu einem eingegebenen Datum (Tag, Monat und Jahr) berechnet, um den wievielten Tag des Jahres es sich handelt! Berücksichtigen Sie dabei die Schaltjahrregel!

A 3.5 Schreiben Sie ein Programm, das alle durch 7 teilbaren Zahlen zwischen zwei zuvor eingegebenen Grenzen ausgibt!

A 3.6 Schreiben Sie ein Programm, das berechnet, wie viele Legosteine zum Bau der folgenden Treppe mit der zuvor eingegebenen Höhe `h` erforderlich sind:

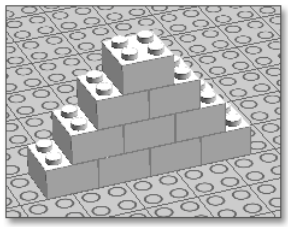


Abbildung 3.16 Treppe aus Legosteinen

A 3.7 Schreiben Sie ein Programm, das eine vom Benutzer festgelegte Anzahl von Zahlen einliest und anschließend die größte und die kleinste der eingegebenen Zahlen auf dem Bildschirm ausgibt!

A 3.8 Implementieren Sie das Ratespiel aus Aufgabe 1.4 entsprechend dem von Ihnen gewählten Algorithmus!

A 3.9 Implementieren Sie Ihren Algorithmus aus Aufgabe 1.5 zur Feststellung, ob eine Zahl eine Primzahl ist!

A 3.10 Schreiben Sie ein Programm, das das kleine Einmaleins berechnet und in Tabellenform auf dem Bildschirm ausgibt! Die Darstellung auf dem Bildschirm sollte wie folgt sein:

```
        1   2   3   4   5   6   7   8   9  10
-------------------------------------------
  1 |   1   2   3   4   5   6   7   8   9  10
  2 |   2   4   6   8  10  12  14  16  18  20
  3 |   3   6   9  12  15  18  21  24  27  30
  4 |   4   8  12  16  20  24  28  32  36  40
  5 |   5  10  15  20  25  30  35  40  45  50
  6 |   6  12  18  24  30  36  42  48  54  60
  7 |   7  14  21  28  35  42  49  56  63  70
  8 |   8  16  24  32  40  48  56  64  72  80
  9 |   9  18  27  36  45  54  63  72  81  90
 10 |  10  20  30  40  50  60  70  80  90 100
-------------------------------------------
```

Die Ausgabe einer ganzen Zahl in einer bestimmten Feldbreite erreichen Sie übrigens dadurch, dass Sie in der Formatanweisung zwischen dem Prozentzeichen und dem Buchstaben für den Datentyp die gewünschte Feldbreite, z. B. in der Form `"%3d"`, angeben.

Kapitel 4
Arithmetik

Der Mangel an mathematischer Bildung gibt sich durch nichts so auffallend zu erkennen wie durch maßlose Schärfe im Zahlenrechnen.
– Carl Friedrich Gauß

Computer bedeutet im Wortsinn Rechner. Einen Computer für eine einmalig vorkommende Berechnung zu verwenden ist nicht besonders sinnvoll. In einer solchen Situation nimmt man besser einen Taschenrechner. Eine besondere Hilfe sind Computerprogramme aber bei sich stereotyp wiederholenden Rechenoperationen. Solche Operationen werden Sie in diesem Abschnitt kennenlernen.

Es gibt einige fundamentale Unterschiede zwischen Berechnungen in der Mathematik und in der Programmierung. Ein wichtiger Unterschied ist, dass die Mathematik mit unendlich vielen Zahlen arbeitet, während ein Computer nur endlich viele Zahlen kennt. In der Mathematik ist es so, dass es zwischen zwei verschiedenen Zahlen immer eine weitere Zahl gibt. Auf einem Computer ist das nicht immer so. Ein Computer muss das mathematische Modell von unendlich vielen Zahlen in ein endliches Zahlenmodell pressen, wobei es dann immer eine größte und eine kleinste Zahl und auch einen Mindestabstand zwischen Zahlen gibt. Bei dieser »Diskretisierung« ergeben sich zwangsläufig Probleme (z. B. Rechenungenauigkeit), mit denen man umzugehen lernen muss.

Es gibt aber noch einen weiteren Unterschied zwischen der Arithmetik der Mathematik und der Arithmetik der Informatik, der mir hier viel wichtiger ist. In der Mathematik versucht man, arithmetische Zusammenhänge durch möglichst einfache und elegante Formeln auszudrücken. In der Programmierung schaut man aus einem anderen Blickwinkel auf diese Formeln, da man sich fragt, wie man einen Formelausdruck möglichst effizient berechnen kann. Naiv würde man vielleicht vermuten, dass eine elegante mathematische Formulierung auch eine effiziente Berechnung nach sich zieht. Das ist aber nicht so. Wir betrachten den folgenden mathematischen Ausdruck:

$$a \cdot x^5 + b \cdot x^4 + c \cdot x^3 + d \cdot x^2 + e \cdot x + f$$

Mit den arithmetischen Grundoperationen können wir den Ausdruck wie folgt berechnen:

$a \cdot x \cdot x \cdot x \cdot x \cdot x + b \cdot x \cdot x \cdot x \cdot x + c \cdot x \cdot x \cdot x + d \cdot x \cdot x + e \cdot x + f$

Aus Sicht der Mathematik ist hier nichts einzuwenden. Der erfahrene Programmierer aber formt den Ausdruck durch systematisches Ausklammern um:

$((((a \cdot x + b) \cdot x + c) \cdot x + d) \cdot x + e) \cdot x + f$

Das ist jetzt nicht mehr so gut lesbar, aber es gibt einen frappierenden Unterschied zur ersten Formulierung. Während die erste Formel 15 Multiplikationen enthält, kommt die zweite mit fünf Multiplikationen aus. Additionen gibt es in beiden Formeln gleich viele. Die zusätzlichen Klammern in der zweiten Formel steigern den Berechnungsaufwand nicht. Sie legen ja nur fest, in welcher Reihenfolge die einzelnen Operationen durchzuführen sind. Das bedeutet, dass man bei der Programmierung die zweite Formulierung bevorzugen wird, zumal diese Formulierung ein sehr einfaches, leicht zu programmierendes Bildungsgesetz aufweist. In dieser Formel klingt der Rhythmus der Informatik: »a mal x + b mal x + c mal x + d ...«

Wir haben die Ausgangsformel in eine einfache Abfolge gleichartiger Rechenschritte zerlegt und wollen diesen Prozess im Folgenden noch präziser beschreiben:

Zunächst indizieren wir die Koeffizienten. Anstelle von a, b, c, d, e und f schreiben wir a_0, a_1, a_2, a_3, a_4 und a_5. Wir erhalten eine Folge von Zwischenergebnissen $z_1 \ldots z_6$, wobei z_6 zugleich das Endergebnis ist:

$z_1 = a_0$
$z_2 = z_1 \cdot x + a_1$
$z_3 = z_2 \cdot x + a_2$
$z_4 = z_3 \cdot x + a_3$
$z_5 = z_4 \cdot x + a_4$
$z_6 = z_5 \cdot x + a_5$

Jetzt erkennen Sie ein wiederkehrendes Muster, das sich auch wie folgt beschreiben lässt:

$z_1 = a_0$
$z_{n+1} = z_n \cdot x + a_n$ für $n = 1,2, \ldots 5$

Damit haben wir eine ganz präzise Vorschrift gefunden, die sich leicht in ein Programm umsetzen lässt[1]. Diesen Ansatz werden wir jetzt weiterverfolgen und mit der Programmierung verbinden.

1 Sie wissen noch nicht, wie Sie »indizierte« Variablen erzeugen können. Dazu später mehr.

4.1 Folgen

In konkreten Problemstellungen stoßen Sie häufig auf Folgen von Zahlen, die einem bestimmten Bildungsgesetz unterliegen. Zum Beispiel:

$$1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \ldots$$

Das allgemeine Bildungsgesetz ist in dieser Schreibweise zwar zu erkennen, aber nicht exakt festgelegt. Sie präzisieren dies, indem Sie das Bildungsgesetz für die k-te Zahl exakt aufschreiben:

$$a_k = \frac{1}{2^k} \quad k = 0, 1, \ldots$$

Jetzt können Sie genau sagen, welchen Wert eine bestimmte Zahl in der Folge hat, indem Sie den entsprechenden Wert für k einsetzen und ausrechnen.

$$a_0 = \frac{1}{2^0} = 1$$

$$a_1 = \frac{1}{2^1} = \frac{1}{2}$$

$$a_2 = \frac{1}{2^2} = \frac{1}{4}$$

$$a_3 = \frac{1}{2^3} = \frac{1}{8}$$

Wir sprechen in diesem Zusammenhang von einer *expliziten Definition* der Folge a_k.

Sie können die Folge a_k aber auch unter einem anderen Blickwinkel betrachten:

> Das erste Glied der Folge hat den Wert 1, alle weiteren Glieder erhalten Sie jeweils durch Halbieren des vorangegangenen Werts.

In einer etwas formaleren Notation schreiben wir das wie folgt:

$$a_k = \begin{cases} 1 & \text{falls } k = 0 \\ \frac{a_{k-1}}{2} & \text{falls } k = 1, 2, 3, \ldots \end{cases}$$

Dies bezeichnen wir als eine *induktive Definition* der Folge a_k. Sie erkennen intuitiv, dass durch die induktive und die explizite Definition die gleiche Zahlenfolge beschrieben ist. An dieser Stelle sollten Sie sich klarmachen, dass induktiv definierte Folgen in der Programmierpraxis häufig vorkommen und sich in besonderer Weise für eine Berechnung durch Computerprogramme eignen.

Wir betrachten dazu ein konkretes Problem. Dieses Problem wollen wir in drei Schritten lösen.

1. Analyse
2. Modellierung
3. Programmierung

Wir beginnen mit der *Analyse*. Dazu müssen wir das Problem zunächst einmal formulieren:

> Ein Student möchte bei seiner Bank ein Darlehen in einer bestimmten Höhe aufnehmen. Er vereinbart eine feste monatliche Ratenzahlung. Diese Rate dient dazu, die monatlich anfallenden Zinsen zu bezahlen, und enthält darüber hinaus einen Tilgungsbetrag, mit dem das Darlehen abbezahlt wird. In dem Maße, in dem die Restschuld abgetragen wird, sinkt der Anteil der Zinsen an der monatlichen Ratenzahlung, und der Tilgungsbetrag wächst entsprechend. Daraus ergibt sich ein ganz bestimmter Tilgungsplan, den wir im Folgenden aufstellen wollen. Darüber hinaus werden wir noch einige durchaus bankenübliche Zusatzregelungen wie etwa Zinsbindung und Sondertilgungen in die Berechnung einfließen lassen.

Wir stellen noch einmal alle relevanten Begriffe zusammen und präzisieren die Aufgabenstellung:

> Ausgangspunkt für den Tilgungsplan ist die anfängliche Darlehenssumme bzw. die *Restschuld*, die jeweils noch zu Buche steht. Mit der Bank wird ein sogenannter *Nominalzins* vereinbart. Die Restschuld wird monatlich mit 1/12 dieses Nominalzinses verzinst. Die monatlich zu zahlende *Rate* wird ebenfalls festgelegt und muss natürlich größer als die anfallenden Zinsen sein, damit noch ein Tilgungsbetrag übrig bleibt. Der *Tilgungsbetrag* ergibt sich dann aus der Monatsrate nach Abzug der monatlichen Zinsen. Wegen des Risikos von Zinsschwankungen garantiert die Bank den obigen Nominalzins allerdings nur über einen gewissen Zeitraum. In diesem Zeitraum besteht dann eine *Zinsbindung*. Nach Ablauf dieser Zeit gelten die dann marktüblichen Zinsen, die im Vorhinein natürlich nur geschätzt werden können und ein gewisses Risiko im Tilgungsplan darstellen. Letztlich wird mit der Bank noch vereinbart, dass jährliche *Sondertilgungen* in einer bestimmten Höhe vorgenommen werden können.

Damit ist das Problem noch nicht gelöst, sondern nur abgegrenzt. Der wesentliche Schritt zur Lösung ist die jetzt folgende *Modellierung*:

Den Kreditnehmer interessiert, wie hoch nach einer gewissen Anzahl von Monaten seine Restschuld ist. Wir bezeichnen die Restschuld nach Ablauf von Monaten mit $rest_n$. In diesem Sinne ist $rest_0$ der volle Darlehensbetrag, aber über die weitere Ent-

wicklung der Folge $rest_n$ wissen Sie noch nicht sehr viel. Sie wissen aber, dass die Zinsen einen großen Einfluss auf die Entwicklung dieser Folge haben. Nun ist der Zinssatz ebenfalls abhängig von der Zeit, da Sie ja einen Zinssatz (*zins1*) für den Zeitraum innerhalb der Zinsbindung und einen weiteren Zinssatz (*zins2*) außerhalb der Zinsbindung zu betrachten haben. Wenn Sie die Anzahl der Jahre, für die die Zinsbindung besteht, mit *bindung* bezeichnen, erhalten Sie die folgende Formel für den gültigen Zinssatz (*zins*) im n-ten Monat:

$$zins_n = \begin{cases} zins1 & \text{falls } n \le bindung \cdot 12 \\ zins2 & \text{falls } n > bindung \cdot 12 \end{cases}$$

Mit diesem Zinssatz können Sie dann die monatliche Zinslast (*zinsen*) auf der Restschuld berechnen:

$$zinsen_n = \frac{rest_n \cdot zins_n}{1200}$$

Was von der monatlichen Rate nach Abzug der Zinsen ($rate - zinsen_n$) noch übrig bleibt, dient zur Tilgung des Darlehens. Ist dieser mögliche Tilgungsbetrag größer als die Restschuld, wird nur in Höhe der Restschuld getilgt, denn der Kreditnehmer will natürlich nicht mehr Geld zurückzahlen, als er bekommen hat. Damit ergibt sich für die Tilgung im n-ten Monat:

$$tilgung_n = \begin{cases} rate - zinsen_n & \text{falls } rate - zinsen_n \le rest_n \\ rest_n & \text{falls } rate - zinsen_n > rest_n \end{cases}$$

Die Restschuld mindert sich dann um diesen Tilgungsbetrag. Sie haben aber noch die jährlich vereinbarten Sonderzahlungen zu berücksichtigen. Diese dürfen natürlich ebenfalls nicht den nach Abzug der Tilgung verbleibenden Darlehensrest übersteigen, und es gilt:

$$sonderz_n = \begin{cases} sondertilgung & \text{falls } n \text{ durch } 12 \text{ teilbar} \\ & \text{und } sondertilgung < rest_n - tilgung_n \\ rest_n - tilgung_n & \text{falls } n \text{ durch } 12 \text{ teilbar} \\ & \text{und } sondertilgung \ge rest_n - tilgung_n \\ 0 & \text{falls } n \text{ nicht durch } 12 \text{ teilbar} \end{cases}$$

Insgesamt ergibt sich dann nach Abzug aller Zahlungen der neue Darlehensrest:

$$rest_{n+1} = rest_n - tilgung_n - sonderz_n$$

Sie haben damit alle für unser Problem relevanten Formeln hergeleitet, und unser Modell ist fertig. Jetzt können Sie mit der *Programmierung* beginnen.

Schritt für Schritt erstellen Sie das Programm. Zunächst legen Sie die erforderlichen Variablen an. Verwenden Sie dabei die oben eingeführten Namen, sodass der Verwendungszweck der Variablen klar sein sollte:

```
void main()
   {
   float rest, rate, zins1, zins2, sondertilgung;
   int bindung;

   int monat;
   float zins, zinsen, tilgung, sonderz;
   }
```

Listing 4.1 Deklaration der verwendeten Variablen

Für die ersten 6 Variablen muss der Benutzer Werte eingeben, während die restlichen nur zur internen Verarbeitung dienen. Den Dialog mit dem Benutzer führen Sie in der folgenden Weise aus:

```
void main()
    {
    float rest, rate, zins1, zins2, sondertilgung;
    int bindung;

    int monat;
    float zins, zinsen, tilgung, sonderz;

    printf( "Darlehen:                 ");
    scanf( "%f", &rest);
    printf( "Nominalzins:              ");
    scanf( "%f", &zins1);
    printf( "Monatsrate:               ");
    scanf( "%f", &rate);
    printf( "Zinsbindung (Jahre):      ");
    scanf( "%d", &bindung);
    printf( "Zinssatz nach Bindung:    ");
    scanf( "%f", &zins2);
    printf( "Jaehrliche Sondertilgung: ");
    scanf( "%f", &sondertilgung);
    }
```

Listing 4.2 Dialog mit dem Benutzer

Jetzt sind alle Daten zur Erstellung des Tilgungsplans eingegeben, und Sie können mit der Berechnung des Plans beginnen. Zunächst wird eine Überschrift ausgegeben. Dann gehen Sie in einer Schleife Monat für Monat vor. Die Schleife endet, wenn das Darlehen vollständig abgetragen ist, also kein Rest mehr bleibt.

```
   void main()
       {
       ... Variablendefinition und Eingaben wie oben ...
       printf( "\nTilgungsplan:\n\n");
       printf( "Monat Zinssatz Zinsen Tilgung Sondertilg Rest\n");

A      for( monat = 1; rest > 0; monat = monat + 1)
           {
B          printf( "%5d", monat);
C          if( monat <= bindung * 12)
               zins = zins1;
           else
               zins = zins2;
           printf( " %10.2f", zins);
D          zinsen = rest * zins / 1200;
           printf( " %10.2f", zinsen);

E          tilgung = rate - zinsen;
           if( tilgung > rest)
               tilgung = rest;
           printf( " %10.2f", tilgung);
F          rest = rest - tilgung;

G          sonderz = 0;
           if( (monat % 12) == 0)
               {
               sonderz = sondertilgung;
               if( sonderz > rest)
                   sonderz = rest;
               }
           printf( " %10.2f", sonderz);
H          rest = rest - sonderz;
           printf( " %10.2f", rest);
I          printf( "\n");
           }
       }
```

Listing 4.3 Vollständige Berechnung des Tilgungsplans

Dazu einige Erklärungen:

(A) In einer Schleife wird Monat für Monat bearbeitet. Für jeden Monat werden die Anweisungen (B–I) ausgeführt. Die Schleife endet, wenn keine Restschuld mehr besteht, das Darlehen also vollständig getilgt ist.

(B) Zunächst wird die laufende Nummer des Monats ausgegeben. Die Feldbreite für die Ausgabe wird durch die Zahl 5 in der Formatanweisung festgelegt. Es erfolgt kein Zeilenvorschub. Alle Ausgaben für einen Monat erscheinen in der gleichen Zeile.

(C) Jetzt wird der zur Anwendung kommende `zins` ermittelt. Vor Ablauf der Zinsbindung ist dies `zins1`, danach `zins2`. Bei der Ausgabe des Zinssatzes wird eine spezielle Formatanweisung für Gleitkommazahlen verwendet, die die Feldbreite (10) und die Anzahl der Nachkommastellen (2) festlegt.

(D) Hier werden die auf die Restschuld fälligen Zinsen berechnet.

(E) Dann wird die Tilgung nach der oben hergeleiteten Formel berechnet und ausgegeben.

(F) Der Darlehensrest wird um die Tilgung gemindert.

(G) Hier wird festgestellt, ob eine Sondertilgung fällig ist. Eine Sondertilgung ist fällig, wenn die Monatszahl ohne Rest durch `12` teilbar ist. Wir verwenden hier den Operator `%`, der den Rest einer Division ermittelt. Das Ergebnis von `monat % 12` ist `0`, wenn ein komplettes Jahr abgelaufen ist und eine Sonderzahlung geleistet wird. Vor der Ausgabe wird noch dafür gesorgt, dass die Sondertilgung nicht höher als der Darlehensrest ausfällt. Zur formatierten Ausgabe der Sondertilgung siehe Punkt C.

(H) Jetzt wird auch noch die Sondertilgung vom Darlehensrest abgezogen. Der jetzt noch verbleibende Betrag wird entsprechend formatiert ausgegeben.

(I) Ein Zeilenvorschub schließt die Ausgabezeile für einen Monat ab.

In einem konkreten Lauf erfragt das Programm zunächst alle für das Darlehen relevanten Daten.

```
Darlehen:                 100000
Nominalzins:              6.5
Monatsrate:               3000
Zinsbindung (Jahre):      1
Zinssatz nach Bindung:    8.0
Jaehrliche Sondertilgung: 10000
```

Im Anschluss wird dann der zugehörige Tilgungsplan erzeugt:

```
Tilgungsplan:

Monat   Zinssatz     Zinsen    Tilgung Sondertilg Rest
    1       6.50     541.67    2458.33       0.00   97541.66
    2       6.50     528.35    2471.65       0.00   95070.02
    3       6.50     514.96    2485.04       0.00   92584.98
    4       6.50     501.50    2498.50       0.00   90086.48
    5       6.50     487.97    2512.03       0.00   87574.45
    6       6.50     474.36    2525.64       0.00   85048.80
    7       6.50     460.68    2539.32       0.00   82509.48
    8       6.50     446.93    2553.07       0.00   79956.41
    9       6.50     433.10    2566.90       0.00   77389.51
   10       6.50     419.19    2580.81       0.00   74808.70
   11       6.50     405.21    2594.79       0.00   72213.91
   12       6.50     391.16    2608.84   10000.00   59605.07
   13       8.00     397.37    2602.63       0.00   57002.44
   14       8.00     380.02    2619.98       0.00   54382.45
   15       8.00     362.55    2637.45       0.00   51745.00
   16       8.00     344.97    2655.03       0.00   49089.97
   17       8.00     327.27    2672.73       0.00   46417.23
   18       8.00     309.45    2690.55       0.00   43726.68
   19       8.00     291.51    2708.49       0.00   41018.20
   20       8.00     273.45    2726.55       0.00   38291.65
   21       8.00     255.28    2744.72       0.00   35546.93
   22       8.00     236.98    2763.02       0.00   32783.91
   23       8.00     218.56    2781.44       0.00   30002.46
   24       8.00     200.02    2799.98   10000.00   17202.48
   25       8.00     114.68    2885.32       0.00   14317.16
   26       8.00      95.45    2904.55       0.00   11412.61
   27       8.00      76.08    2923.92       0.00    8488.70
   28       8.00      56.59    2943.41       0.00    5545.29
   29       8.00      36.97    2963.03       0.00    2582.26
   30       8.00      17.22    2582.26       0.00       0.00
```

Das Beispiel zeigt, wie einfach Sie in einer Programmschleife eine iterativ definierte Folge berechnen können, ohne sich Gedanken über eine explizite Darstellung der Folge machen zu müssen. Das Beispiel zeigt auch, dass Sie eine Aufgabenstellung zunächst mit Papier und Bleistift analysieren sollten, bevor Sie mit der Programmierung beginnen.

Im Prinzip handelt es sich bei dem oben dargestellten Programm um die Simulation eines endlichen Prozesses. Sie wissen ja, dass die Schuld irgendwann vollständig getilgt ist, wenn jeden Monat ein gewisser Mindestbetrag getilgt wird. Manchmal

haben Sie es aber auch mit Prozessen zu tun, bei denen es nicht von vornherein klar ist, dass sie enden oder dass sich ein stabiles Ergebnis einstellt. Einem solchen Prozess wollen wir uns jetzt zuwenden.

Wenn Sie bei einem einfachen Problem auf eine Gleichung wie $x \cdot x = 10$ stoßen, können Sie diese Gleichung mit den arithmetischen Grundoperationen nicht lösen, da Sie zur Lösung ja die Wurzel ziehen müssen. Bei einem Taschenrechner drücken Sie einfach auf die »$\sqrt{}$«-Taste und erhalten:

$$\sqrt{10} = 3.162...$$

Das ist natürlich nur ein Näherungswert, und Sie müssen davon ausgehen, dass der exakte Wert im endlichen Zahlenmodell des Computers nicht vorkommt. Um eine Näherungslösung zu finden, folgen Sie einer uralten Idee des griechischen Mathematikers Heron[2]. Für die alten Griechen war Mathematik im Wesentlichen Geometrie[3], und auch das »Ziehen der Wurzel« war für sie ein geometrisches Problem:

> Gesucht ist die Kantenlänge eines Quadrats, das eine vorgegebene Fläche a (z. B. a = 10) hat.

Wir nennen die gesuchte Lösung w und starten mit einer mehr oder weniger willkürlichen ersten Näherung:

$$w_0 = a$$

Wenn wir w_0 als eine Seitenlänge eines Rechtecks auffassen, das die Fläche a haben soll, müssen wir $\frac{a}{w_0}$ als Länge der anderen Seite wählen. Das ist sicher noch eine ungenügende Annäherung an ein Quadrat, aber wenn wir im nächsten Schritt den Mittelwert aus den beiden Kantenlängen wählen, wird unser Rechteck schon deutlich quadratischer:

$$w_1 = \frac{1}{2}\left(w_0 + \frac{a}{w_0}\right)$$

Das setzen wir jetzt einfach so fort:

$$w_2 = \frac{1}{2}\left(w_1 + \frac{a}{w_1}\right)$$

Abbildung 4.1 zeigt die Entwicklung unserer Folge, die sich offensichtlich längs des Funktionsgraphen $f(x) = \frac{10}{x}$ an das Ziel $\sqrt{10}$ herantastet:

2 Heron von Alexandria lebte und lehrte vermutlich im 1. Jahrhundert n. Chr. in Alexandria.

3 Die Algebra stammt zwar auch aus Griechenland, wurde aber erst ca. 300 Jahre nach Heron entdeckt.

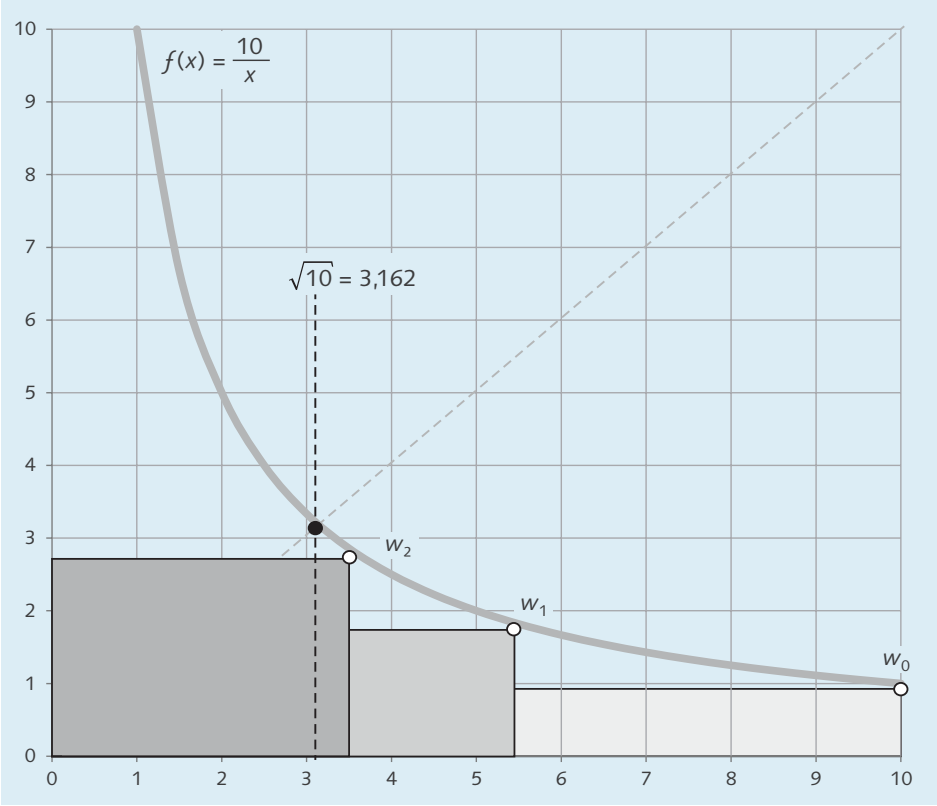


Abbildung 4.1 Entwicklung der Folge

Die Folge

$$w_n = \begin{cases} a & \text{falls } n = 0 \\ \frac{1}{2}\left(w_{n-1} + \frac{a}{w_{n-1}}\right) & \text{falls } n = 1, 2, \ldots \end{cases}$$

scheint eine gute Annäherung an den Zielwert $w = \sqrt{a}$ zu liefern. Sie probieren das mit einem Programm aus, wobei Sie den Wert a für die zu berechnende Wurzel vom Benutzer eingeben lassen. Sie wissen allerdings noch nicht, wie oft Sie die Iteration durchführen müssen, bis das Ergebnis genau genug ist. Versuchen Sie es zunächst mit zehn Durchläufen:

```
void main()
    {
    float a, w;
    int i;

    printf( "Bitte Zahl eingeben: ");
    scanf( "%f", &a);

    w = a;
    for( i = 0; i < 10; i++)
        {
        w = (w + a/w)/2;
        printf( "%f\n", w);
        }
    }
```

Listing 4.4 Berechnung der Wurzel durch Iteration

Dieses Programm arbeitet dann wie folgt:

```
Bitte Zahl eingeben: 10
5.500000
3.659091
3.196005
3.162456
3.162278
3.162278
3.162278
3.162278
3.162278
3.162278
```

Das ist eine sehr gute Näherung, offensichtlich hätten sogar weniger Schleifendurchläufe ausgereicht. Aber das Programm selbst kann Ihnen nicht sagen, ob es allgemein (d. h. nicht nur für 10) funktioniert. Selbst weitere Tests könnten Sie nicht zufriedenstellen, da immer ein Restzweifel bestehen bleibt. Eine befriedigende Antwort kann Ihnen nur die Mathematik geben. Sie muss Ihnen zwei Fragen beantworten, bevor Sie diesem Programm trauen können:

Konvergiert dieses Verfahren allgemein – und wenn ja, gegen welchen Wert?

Die Mathematik sagt[4]:

$$w_n \geq w_{n+1} \geq \sqrt{a} \quad (\textit{für } n = 1, 2, \ldots)$$

Die Mathematik sagt auch, dass solche Folgen, die monoton fallen und nach unten beschränkt sind, konvergieren. Wenn Sie sicher sind, dass das Verfahren konvergiert, können Sie sehr einfach den Grenzwert ermitteln. Wir nennen den Grenzwert *w* und machen in der Formel

$$w_n = \frac{1}{2}\left(w_{n-1} + \frac{a}{w_{n-1}}\right)$$

auf beiden Seiten den »Grenzübergang ins Unendliche« und erhalten für *w* die folgende Gleichung:

$$w = \frac{1}{2}\left(w + \frac{a}{w}\right)$$

Aus dieser Gleichung folgt unmittelbar:

$w^2 = a$

Also:

$w = \sqrt{a}$

Nach diesen Überlegungen sind Sie sicher, dass Sie das Verfahren nach einem Schritt mit der Bedingung

$w_n \cdot w_n - a < fehlerschranke$

abbrechen können. Der Schleifenzähler wird nicht mehr benötigt:

```
void main()
   {
   float a, w;

   printf( "  Bitte Zahl eingeben: ");
   scanf( "%f", &a);

   w = a;
   for( ; ; )
      {
      w = (w + a/w)/2;
      printf( "  %f\n", w);
```

4 Wenn Sie der mathematische Beweis interessiert, schauen Sie im Internet unter dem Stichwort »Heron-Verfahren« nach.

```
        if( w*w - a < 0.001)
            break;
        }
    }
```

Listing 4.5 Iteration mit Abbruchkriterium

Jetzt bricht das Programm ab, sobald die geforderte Genauigkeit erreicht ist:

```
Bitte Zahl eingeben: 10
5.500000
3.659091
3.196005
3.162456
```

Das Programm zur Berechnung der Wurzel ist ein einfaches Beispiel für ein sogenanntes *numerisches Verfahren*. Numerische Verfahren sind sehr eng mit der Mathematik verknüpft und werden eingesetzt, wenn Probleme aus der »analogen« Welt in der diskreten Welt eines Computers simuliert und gelöst werden sollen. Denken Sie dabei an Wetter- oder Klimasimulationen, an die Simulation eines Tsunamis oder des dynamischen Fahrverhaltens eines Autos. Allein das Gebiet der numerischen Verfahren ist so umfangreich, dass die Literatur dazu ganze Bibliotheken füllt.

4.2 Summen und Produkte

Am Ende des vorangegangenen Kapitels hatte ich Ihnen die Aufgabe gestellt zu berechnen, aus wie vielen Steinen die folgende Legotreppe besteht, wenn Sie von einer beliebigen Höhe h ausgehen:

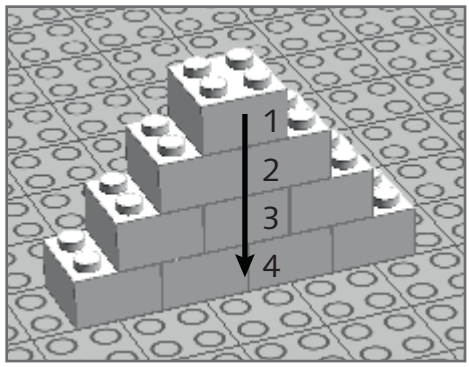


Abbildung 4.2 Legotreppe

Das iterative Bildungsgesetz für die Anzahl der Steine ist schnell gefunden:

$$s_h = \begin{cases} 0 & \text{für } h = 0 \\ s_{h-1} + h & \text{für } h > 0 \end{cases}$$

Das bedeutet:

$s_0 = 0$
$s_1 = 1$
$s_2 = 1 + 2 = 3$
$s_3 = 1 + 2 + 3 = 6$
$s_4 = 1 + 2 + 3 + 4 = 10$
$s_5 = 1 + 2 + 3 + 4 + 5 = 15$
...
$s_h = 1 + 2 + 3 + 4 + 5 + \cdots + h = ?$

Sie können den gesuchten Wert iterativ durch ein C-Programm berechnen:

```
void main()
   {
   int max =9;
   int steine = 0;
   int h;

   for ( h=1; h<= max; h= h + 1)
      {
      steine = steine + h;
      printf("  Hoehe: %d, Steine = %d\n", h, steine);
      }
   }
```

Listing 4.6 Berechnung der Treppenelemente

Mit dem Programm erhalten Sie das folgende Ergebnis:

```
  Hoehe: 1, Steine = 1
  Hoehe: 2, Steine = 3
  Hoehe: 3, Steine = 6
  Hoehe: 4, Steine = 10
  Hoehe: 5, Steine = 15
  Hoehe: 6, Steine = 21
  Hoehe: 7, Steine = 28
  Hoehe: 8, Steine = 36
  Hoehe: 9, Steine = 45
```

Aber Sie sind in diesem Fall auch in der Lage, eine explizite Formel anzugeben. Dazu bauen Sie die gleiche Treppe noch einmal – auf dem Kopf stehend – neben die zu untersuchende Treppe:

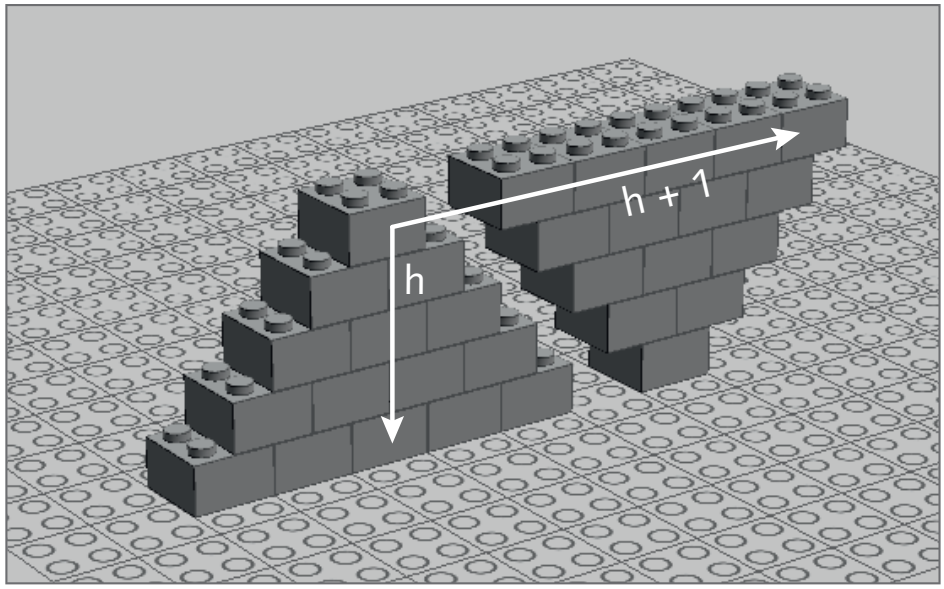


Abbildung 4.3 Zwei angeordnete Legotreppen

Sie sehen, dass jeweils *h+1* Steine in *h* Schichten übereinander vorhanden sind und dass das doppelt so viele Steine sind, wie in einer Treppe benötigt werden. Es gilt also:

$$s_h = 0 + 1 + 2 + 3 + 4 + 5 + \cdots + h = \frac{(h+1)h}{2}$$

Dieser Formel, der sogenannten *gaußschen Summenformel*, werden Sie häufiger begegnen, da sie eine wichtige Rolle bei der Beurteilung von Algorithmen spielt.

Wir können die Addition in der gaußschen Summenformel durch eine Multiplikation ersetzen und uns das folgende Bildungsgesetz anschauen:

$$f_n = \begin{cases} 1 & \text{für } n = 0 \\ f_{n-1} \cdot n & \text{für } n > 0 \end{cases}$$

Auch diese Folge

$f_0 = 1$
$f_1 = 1$
$f_2 = 1 \cdot 2 = 2$
$f_3 = 1 \cdot 2 \cdot 3 = 6$

$f_4 = 1 \cdot 2 \cdot 3 \cdot 4 = 24$

$f_5 = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$

...

$f_n = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot \ldots \cdot n$

können wir durch ein C-Programm berechnen:

```
void main()
   {
   int max = 9;
   int f = 1;
   int n;

   for ( n=1; n<= max; n= n + 1)
      {
      f = f * n;
      printf("  %d! = %d\n", n,f);
      }
   }
```

Listing 4.7 Berechnung der Fakultäten

Das Programm erzeugt diese Ausgabe:

```
  1! = 1
  2! = 2
  3! = 6
  4! = 24
  5! = 120
  6! = 720
  7! = 5040
  8! = 40320
  9! = 362880
```

Diese Folge ist so wichtig, dass man ihr einen eigenen Namen gegeben hat. Es ist die Folge der *Fakultäten*. Das einzelne Folgenglied zum Index n nennen wir *n-Fakultät* und schreiben dafür »n!«. Also:

0! = 1	*0-Fakultät*
1! = 1	*1-Fakultät*
2! = 1 · 2 = 2	*2-Fakultät*
3! = 1 · 2 · 3 = 6	*3-Fakultät*
4! = 1 · 2 · 3 · 4 = 24	*4-Fakultät*

$5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$ *5-Fakultät*

...

$n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot \ldots \cdot n$ *n-Fakultät*

Auch diese Folge wird Ihnen bei der Programmierung häufig begegnen.

4.3 Aufgaben

A 4.1 Schreiben Sie ein Programm, das zu einem gegebenen Anfangskapital und einem jährlichen Zinssatz berechnet, wie viele Jahre benötigt werden, damit das Kapital eine bestimmte Zielsumme überschreitet!

A 4.2 Den größten gemeinsamen Teiler (ggT) von zwei natürlichen Zahlen können Sie berechnen, indem Sie so lange die kleinere Zahl von der größeren Zahl abziehen, bis beide Zahlen gleich sind. Sie möchten z. B. den ggT von 152 und 56 berechnen. Dann gehen Sie wie folgt vor:

152 – 56 = 96
96 – 56 = 40
56 – 40 = 16
40 – 16 = 22
22 – 16 = 8
16 – 8 = 8 = ggT

Erstellen Sie ein Programm, das mit diesem Algorithmus den ggT berechnet!

A 4.3 Sie haben zwei ausreichend große Eimer. Im ersten befinden sich x, im zweiten y Liter Wasser. Sie füllen nun immer a Prozent des Wassers aus dem ersten in den zweiten und anschließend b Prozent des Wassers aus dem zweiten in den ersten Eimer. Diesen Umfüllprozess führen Sie n-mal durch. Erstellen Sie ein Programm, das nach Eingabe der Startwerte (x, y, a, b und n) die Füllstände der Eimer nach jedem Umfüllen ermittelt und auf dem Bildschirm ausgibt! Welche Aufteilung des Wassers ergibt sich auf lange Sicht für unterschiedliche Startwerte?

A 4.4 In einem Schulbezirk gibt es 1200 Planstellen für Lehrer. Diese unterteilen sich derzeit in 40 Studiendirektoren, 160 Oberstudienräte und 1000 Studienräte. Alle drei Jahre ist eine Beförderung möglich, dabei steigen jeweils 10 % der Oberstudienräte und 20 % der Studienräte in die nächsthöhere Gruppe auf. Darüber hinaus gehen 20 % einer jeden Gruppe innerhalb von drei Jahren in den Ruhestand. Die dadurch frei werdenden Planstellen werden mit Studienräten besetzt. Schreiben Sie ein Programm, das die bestehende Situation in Dreijahreszyklen fortschreibt! Welche Verteilung von Direktoren, Oberräten und Räten ergibt sich auf lange Sicht? Drehen Sie an der »Beförderungs-

schraube« für Oberstudienräte und Studienräte, um andere Verteilungen zu erreichen!

A 4.5 Epidemien (z. B. Grippewellen) breiten sich in der Bevölkerung nach gewissen Gesetzmäßigkeiten aus. Die Bevölkerung zerfällt im Verlauf einer Epidemie in drei Gruppen. Als *Gesunde* bezeichnen wir Menschen, die mit dem Krankheitserreger noch nicht in Berührung gekommen sind und deshalb ansteckungsgefährdet sind. *Kranke* sind Menschen, die akut infiziert und ansteckend sind. *Immunisierte* schließlich sind Menschen, die die Krankheit überstanden haben und weder ansteckend noch ansteckungsgefährdet sind.

Als Ausgangssituation betrachten wir eine feste Population von x Menschen, unter denen sich bereits eine gewisse Anzahl y von Kranken befindet:

$gesund_0 = x - y$

$krank_0 = y$

$immun_0 = 0$

Ausgehend von diesen Daten, wollen wir die Ausbreitung der Krankheit in Zeitsprüngen von einem Tag berechnen. Wir überlegen uns dazu, welche Veränderungen von Tag zu Tag auftreten. Es gibt zwei Arten von Übergängen zwischen den Gruppen. Aus Gesunden werden Kranke (Infektion), und aus Kranken werden Immune (Immunisierung).

Die Zahl der Infektionen ist proportional zur Zahl der Gesunden und proportional zum Anteil der Kranken in der Gesamtbevölkerung. Denn je mehr Gesunde es gibt, desto mehr Menschen können sich anstecken, und je mehr Ansteckende es gibt, desto mehr Menschen können angesteckt werden. Mit einem geeigneten Proportionalitätsfaktor (Infektionsrate) nimmt daher die Zahl der Gesunden ständig ab:

$$gesund_{n+1} = gesund_n - infektionsrate \cdot \frac{gesund_n krank_n}{x}$$

Die Zahl der Immunisierungen ist proportional zur Zahl der Kranken, denn je mehr Menschen erkrankt sind, desto mehr Menschen erlangen Immunität. Mit einem geeigneten Proportionalitätsfaktor (Immunisierungsrate) gilt daher:

$immun_{n+1} = immun_n + immunisierungsrate \cdot krank_n$

Der Rest der Population ist krank.

$krank_{n+1} = x - gesund_{n+1} - immun_{n+1}$

Die Proportionalitätsfaktoren (Infektionsrate und Immunisierungsrate) hängen dabei von medizinisch-sozialen Faktoren wie Art der Krankheit, hygienische Bedingungen, Bevölkerungsdichte, medizinische Versorgung etc. ab und

können daher nur empirisch ermittelt werden. Sind Ihnen diese Faktoren aber aus der Kenntnis früherer Epidemien her bekannt, können Sie mit einem einfachen Programm den Verlauf der Krankheitswelle vorausberechnen. Erstellen Sie das Programm, und ermitteln Sie den Verlauf einer Epidemie mit den folgenden Basisdaten:

```
Infektionsrate:      0.6
Immunisierungsrate: 0.06
Gesamtpopulation:    2000
Akut Kranke:         10
Anzahl Tage:         25
```

Abbildung 4.4 zeigt für die oben genannten Basisdaten das epidemische Anwachsen des Krankenstandes, bis dem Virus der Nährboden entzogen wird und der Krankenstand langsam wieder abfällt:

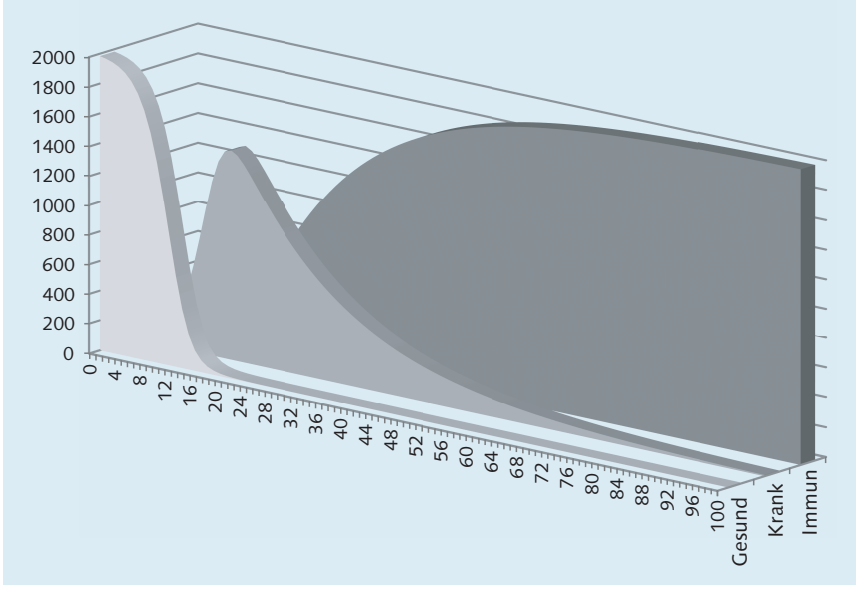


Abbildung 4.4 Epidemieverlauf

A 4.6 Der belgische Mathematiker Viktor d'Hondt entwickelte 1882 ein Verfahren, um zu einem Wahlergebnis die zugehörige Sitzverteilung für ein Parlament zu berechnen. Dieses Verfahren (d'Hondtsches Höchstzahlverfahren) wurde bis 1983 verwendet, um die Sitzverteilung für den Deutschen Bundestag festzulegen.

Zur Durchführung des Verfahrens werden die Stimmergebnisse der Parteien fortlaufend durch die Zahlen 1, 2, 3, 4, ... dividiert. Sind n Sitze im Parlament zu

vergeben, werden die n größten Divisionsergebnisse ausgewählt, und die zugehörigen Parteien erhalten für jede ausgewählte Zahl einen Sitz. Das folgende Beispiel zeigt das Ergebnis einer Wahl mit drei Parteien und 200000 abgegebenen Stimmen, bei der zehn Sitze zu vergeben waren:

	Partei A	**Partei B**	**Partei C**
Stimmen	100000	80000	20000
1	100000	80000	20000
2	50000	40000	10000
3	33333	26666	6666
4	25000	20000	5000
5	20000	16000	4000
6	16666	13333	3333
7	14285	11429	2857
8	12500	10000	2500
Sitze	5	4	1

Tabelle 4.1 Stimmen und Sitzverteilung nach d'Hondt

Schreiben Sie ein Programm, das für eine beliebige Wahl mit drei Parteien die Sitzverteilung berechnet! Die Anzahl der zu vergebenden Sitze und die Stimmen für die drei Parteien sollen dabei vom Benutzer eingegeben werden.

A 4.7 Im folgenden Zahlenkreis stehen die Buchstaben jeweils für eine Ziffer.

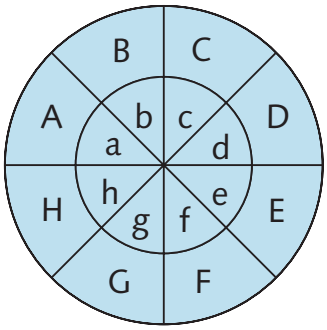


Abbildung 4.5 Zahlenkreis

Bestimmen Sie diese Ziffern (1 bis 9) so, dass folgende Bedingungen erfüllt werden:

- `Aa`, `Bb`, `Cc`, `Dd`, `Ee`, `Ff`, `Gg` und `Hh` sind Primzahlen.
- `ABC` ist ein Vielfaches von `Aa`.
- `abc` ist gleich `cba`.
- `CDE` ist Produkt von `Cc` mit der Quersumme von `CDE`.
- `Bb` ist gleich der Quersumme von `cde`.
- `EFG` ist ein Vielfaches von `Aa`.
- `efg` ist Produkt von `Aa` mit der Quersumme von `efg`.
- `GHA` ist Produkt von `eE` mit der Quersumme von `ABC`.
- Die Quersumme von `gha` ist `Cc`.

Zeigen Sie durch ein Programm, dass es genau eine mögliche Ziffernzuordnung gibt, und bestimmen Sie diese!

A 4.8 Erstellen Sie ein Programm, das zu einer vom Benutzer eingegebenen Zahl die Primzahlzerlegung ermittelt

```
Zahl: 13230
13230 = 2*3*3*3*5*7*7
```

und auf dem Bildschirm ausgibt!

A 4.9 Wichtige mathematische Funktionen können näherungsweise durch Summen (man nennt dies *Potenzreihenentwicklung*) berechnet werden.

Zum Beispiel:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots$$

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

Die im Nenner der Brüche vorkommenden Fakultäten kennen Sie ja aus Abschnitt 4.2, »Summen und Produkte«.

Erstellen Sie auf diesen Formeln basierende Berechnungsprogramme für Sinus, Cosinus und e-Funktion! Überprüfen Sie die Ergebnisse Ihrer Programme mit einem Taschenrechner!

A 4.10 Erstellen Sie Programme, um den Steinverbrauch für die in Abbildung 4.6 abgebildete Treppe und die beiden Pyramiden zu berechnen. Die Pyramiden sind dabei innen nicht hohl.

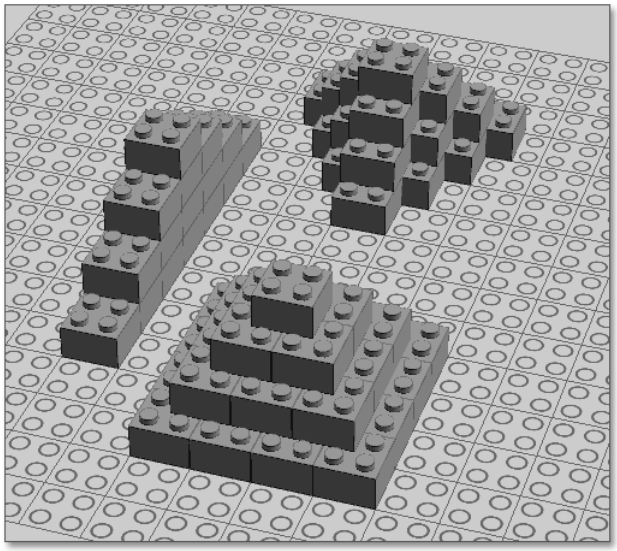


Abbildung 4.6 Pyramiden

Versuchen Sie, auch explizite Formeln für den Steinverbrauch herzuleiten. Vergleichen Sie die iterativ berechneten Ergebnisse mit den durch die expliziten Formeln gegebenen Zahlen.

Kapitel 5
Aussagenlogik

Logiker: Alle Katzen sind sterblich. Sokrates ist gestorben. Also ist Sokrates eine Katze.
Älterer Herr: Ich habe eine Katze, die heißt Sokrates.
Logiker: Sehen Sie ...
Älterer Herr: Sokrates war also eine Katze!
Logiker: Die Logik hat es uns eben bewiesen.
– Aus »Die Nashörner« von Eugène Ionesco

Eine ganze Nacht habe ich mich mit der Frage gequält, wie ich in die Thematik dieses Abschnitts einsteigen soll. Als ich heute beim Frühstück saß, war plötzlich alles ganz einfach, denn in meiner Morgenlektüre fand ich den folgenden Artikel:

Anzeige wegen 40 Cent

QUIZ. Ein Mann durfte die Scherzfrage einer TV-Show nicht beantworten, aber zahlen. Betrug?

MÜNSTER. Ein kurioser Fall für die Justiz: Polizei und Staatsanwaltschaft in Münster müssen sich mit einer Betrugsanzeige um 40 Cent und einer Scherzfrage im Fernsehen befassen. Ein Mann habe in der Nacht zu gestern bei einer TV-Quizshow angerufen, um die Frage zu beantworten, „wie viel Milch geben zehn Ochsen in zehn Minuten, wenn fünf Ochsen in fünf Minuten fünf Liter Milch geben?", berichtete die Polizei. Jetzt fühlte er sich betrogen, weil ihm am Telefon mitgeteilt wurde, dass es sich nur um eine Scherzfrage handele, da Ochsen keine Milch geben. Der Mann hatte hingegen das mathematische Problem mit dem Dreisatz gelöst. Zudem musste er statt der in der Sendung angegebenen 50 Cent pro Anruf 90 Cent berappen.

So sehr beide Betrugsvorwürfe für Schmunzeln in den Amtsstuben sorgten, müssen nun sowohl Polizei als auch Staatsanwaltschaft beiden Vorwürfen nachgehen. (dpa)

Abbildung 5.1 Anzeige wegen 40 Cent (aus der NRZ)

Natürlich handelt es sich bei dieser Frage um eine Scherzfrage, aber wie geht der Logiker mit solchen Sätzen um? Etwa mit dem folgenden Satz:

> Wenn fünf Ochsen in fünf Minuten fünf Liter Milch geben, dann gibt es den Osterhasen.

Ist dieser Satz falsch, weil Ochsen keine Milch geben? Oder ist er falsch, weil es den Osterhasen nicht gibt? Oder ist er vielleicht sogar richtig? Und wenn er richtig ist, ist dann die Existenz des Osterhasen bewiesen? Mit so wichtigen Fragen werden wir uns in diesem Abschnitt beschäftigen, und Sie werden auch wieder einiges an Programmierung lernen.

5.1 Aussagen

Die Aussagenlogik beschäftigt sich, wie nicht anders zu erwarten, mit Aussagen. Unter einer Aussage verstehen wir einen Satz, der entweder wahr oder falsch ist. Wir müssen nicht wissen, ob der Satz wahr oder falsch ist, wir müssen ihm nur prinzipiell zugestehen, dass er wahr oder falsch ist. Genau genommen, interessieren wir uns nicht einmal dafür, ob der Satz wahr oder falsch ist. Und ganz genau genommen, interessieren wir uns nicht einmal dafür, was »wahr« und »falsch« inhaltlich bedeutet. Wir können jederzeit 0 oder 1 anstelle von »falsch« oder »wahr« sagen. Insofern betreiben wir Logik als ein rein formales System ohne Bezug zur Realität.

Konkrete Aussagen sind z. B.:

»Köln liegt in Deutschland.«
»Köln hat mehr als 1 Mio. Einwohner.«

Keine Aussagen im Sinne unserer Begriffsbildung sind dagegen:

»Guten Tag, meine Damen und Herren!«
»Wie spät ist es?«

Bei der Programmierung haben wir es nicht mit umgangssprachlichen Aussagen, sondern mit präzise formulierten Aussagen in einer Programmiersprache zu tun wie »$wert < 10$« oder »$a + b < c$«. Wir wollen uns deshalb auch nicht zu weit auf das glatte Eis umgangssprachlicher Aussagen hinausbegeben.

5.2 Aussagenlogische Operatoren

Durch die Aussagenlogik möchten wir nicht ergründen, ob eine Aussage wirklich wahr oder falsch ist. Im Falle der Aussage über die Einwohnerzahl Kölns wäre dafür auch eher das Einwohnermeldeamt als die Logik zuständig. Wir möchten aus elementaren Aussagen, deren Wahrheitswert wir als gegeben annehmen, komplexere Aussagen zusammensetzen und uns Gedanken über den Wahrheitswert dieser neuen Aussagen machen.

Eine zusammengesetzte Aussage ist etwa:

»Köln liegt in Deutschland *und* Köln hat mehr als 1 Mio. Einwohner.«

Der Wahrheitswert dieser zusammengesetzten Aussage hängt von den Wahrheitswerten der Einzelaussagen ab. Unser Sprachgefühl sagt uns, dass die Gesamtaussage richtig ist, wenn *beide* Teilaussagen richtig sind. Kennen wir also den Wahrheitswert der Teilaussagen, kennen wir auch den Wahrheitswert der Gesamtaussage.

Das Wort »und«, mit dem wir die Teilaussagen verbunden haben, ist ein sogenannter *logischer Operator*. Unsere Sprache kennt viele weitere solcher Operatoren, die wir täglich benutzen, ohne uns vielleicht jemals deren genaue Bedeutung klargemacht zu haben. Als ein Beispiel betrachten wir den Operator »während«. Diesen Operator benutzen wir in verschiedenen Bedeutungen, zum einen, um einen schwachen Gegensatz, zum anderen, um einen gleichzeitigen Verlauf auszudrücken. Die Aussage

»Mein Auto ist rot, *während* dein Auto grün ist.«

heißt eigentlich nichts anderes als: »Mein Auto ist rot *und* dein Auto ist grün«. Dies allerdings mit dem deutlichen Zusatz: »Man beachte den feinen Unterschied«.

Die Aussage

»Es regnete, *während* ich im Kino war.«

dagegen beschreibt den zeitlichen Verlauf zweier Ereignisse. Im ersten Fall macht der Operator »während« einen subtilen Zusatz, der oft nur aus dem Zusammenhang und für eingeweihte Zuhörer zu verstehen ist. Im zweiten Fall kann der Wahrheitswert der Gesamtaussage nicht aus den Wahrheitswerten der einzelnen Aussagen abgeleitet werden, weil der Operator eine Zusatzaussage über die zeitliche Parallelität der Einzelaussagen macht. Beide Varianten des Operators »während« sind für unsere Zwecke ungeeignet, denn wir wollen hier nur Operatoren behandeln, bei denen sich der resultierende Wahrheitswert zweifelsfrei aus den Wahrheitswerten der beteiligten Einzelaussagen herleiten lässt.

Weitere Beispiele für umgangssprachliche Operatoren sind:

- nicht ...
- ... oder ...
- weder ... noch ...
- wenn ... dann ...
- zwar ... aber ...
- entweder ... oder ...
- sowohl ... als auch ...

Mit umgangssprachlichen Formulierungen sind wir wegen der häufig auftretenden Fehlinterpretationen nicht zufrieden. Wir werden daher im Folgenden einige Präzisierungen vornehmen müssen.

Zunächst benutzen wir für die Wahrheitswerte »wahr« bzw. »falsch« die Symbole *1* bzw. *0*. Für Aussagen setzen wir Großbuchstaben `A`, `B`, `C` etc. oder `A1`, `A2`. »Die Aussage

A ist wahr« heißt dann in Formelschreibweise $A = 1$. Umgekehrt heißt $A = 0$: »Die Aussage *A* ist falsch«.

Die drei wichtigsten Operatoren sind sicher: *nicht*, *und* und *oder*. Da die Operanden eines logischen Operators nur die Werte 0 oder 1 annehmen, können wir einen logischen Operator durch eine Tabelle vollständig beschreiben: Eine solche Tabelle nennen wir *Wahrheitstafel*. Über solche Wahrheitstafeln werden wir jetzt die drei wichtigsten Operatoren einführen. Dabei handelt es sich um: *nicht*, *und* und *oder*.

Die Aussage »nicht A« ist genau dann wahr, wenn die ursprüngliche Aussage *A* falsch ist. Damit ergibt sich für den Nicht-Operator folgende Wahrheitstafel:

A	nicht A
0	1
1	0

Tabelle 5.1 Wahrheitstafel für den Nicht-Operator

Anstelle von »nicht *A*« schreiben wir in Formeln auch $\overline{A}$ oder *!A*.

Wir hatten bereits festgestellt, dass eine Und-Aussage genau dann wahr ist, wenn *beide* Teilaussagen wahr sind. Den Operator »und« definieren wir also über die folgende Wahrheitstafel:

A	B	A und B
0	0	0
0	1	0
1	0	0
1	1	1

Tabelle 5.2 Wahrheitstafel für den Und-Operator

Auch für diesen Operator verwenden wir spezielle Formelsymbole. Anstelle von *A und B* schreiben wir auch $A \wedge B$ oder *A && B*.

Bleibt noch das »oder«, für das wir die Notationen *A oder B*, $A \vee B$ und $A \,||\, B$ verwenden:

A	B	A oder B
0	0	0
0	1	1
1	0	1
1	1	1

Tabelle 5.3 Wahrheitstafel für den Oder-Operator

Eine Aussage, die aus zwei mit »oder« verbundenen Teilaussagen besteht, ist also genau dann wahr, wenn *mindestens eine* der beiden Teilaussagen wahr ist.

An dieser Definition erhitzen sich gelegentlich die Gemüter. Vielfach wird gefordert, dass die Aussage »*A* oder *B*« falsch zu sein habe, wenn *A* und *B* beide wahr sind. Dies entspricht dem Operator »entweder ... oder ...«. Die deutsche Sprache[1] trennt leider nicht sauber zwischen »oder« und »entweder ... oder«. Vielfach wird dort, wo eigentlich »entweder ... oder ...« gemeint ist, einfach nur »oder« verwendet. In aller Regel ist das unproblematisch, weil zumeist aus dem Zusammenhang klar ist, welcher der beiden Operatoren gemeint ist, oder weil sich die Alternativen sowieso gegenseitig ausschließen.

So bedeutet die Frage

»Sollen wir um 8 Uhr *oder* um 10 Uhr ins Kino gehen?«

in aller Regel:

»Sollen wir *entweder* um 8 Uhr *oder* um 10 Uhr ins Kino gehen?«

Der Fall, dass beide Alternativen gewählt werden, wird dabei von vornherein ausgeschlossen. Im strengen Sinne unseres Gebrauchs des Operators »oder« schließt die erste Formulierung der Frage aber nicht aus, sowohl um 8 als auch um 10 ins Kino zu gehen. Seien Sie also immer vorsichtig! Wenn Sie ein Logiker auf der Straße mit den Worten »Geld oder Leben!« überfällt, und Sie geben Ihm das Geld, kann er immer noch Ihr Leben nehmen, ohne wortbrüchig zu werden. Bestehen Sie also in dieser Situation auf der Formulierung »Entweder Geld oder Leben«, und geben Sie erst dann das Geld. Zum Schluss aber noch ein Beispiel dazu, dass wir auch in unserer Umgangssprache das nicht ausschließende »oder« ganz selbstverständlich benutzen. Wenn Sie etwa an der Grenze gefragt werden

»Haben Sie Ihren Pass *oder* Ihren Personalausweis dabei?«,

würden Sie dann mit Nein antworten, wenn Sie zufällig *beide* Dokumente eingesteckt haben?

Mit den Bausteinen »nicht«, »und« und »oder« können wir jetzt beliebig komplexe logische Ausdrücke zusammensetzen. Aber noch immer lässt die Umgangssprache zu viel Interpretationsspielraum. Wenn etwa die Zollvorschriften besagen, dass man

entweder 1 Liter Spirituosen *oder* 5 Liter Bier *und* eine Stange Zigaretten

importieren darf, ist die Frage, ob

1 Die lateinische Sprache kennt z. B. »vel« für das nicht ausschließende und »aut« für das ausschließende »oder«.

(*entweder* 1 Liter Spirituosen *oder* 5 Liter Bier) *und* eine Stange Zigaretten

oder

entweder 1 Liter Spirituosen *oder* (5 Liter Bier *und* eine Stange Zigaretten)

gemeint ist. Vermutlich das Erstere. Diese Vermutung leitet sich aber nicht aus dem logischen Gerüst der Aussage, sondern aus der Tatsache ab, dass es sich bei Spirituosen und Bier um ähnliche und daher vielleicht austauschbare Dinge handelt. Soll ein Logiker es aufgrund dieser vagen Annahme riskieren, mit 1 Liter Spirituosen *und* einer Stange Zigaretten die Grenze zu überqueren? Dies wäre zwar bei der ersten Interpretation erlaubt, bei der zweiten aber verboten.

Wir müssen präziser sein und Operatoren so definieren, dass immer eine eindeutige Auswertungsreihenfolge gegeben ist. Dazu geben wir in gemischten Ausdrücken »nicht« eine höhere Priorität als »und« und »und« eine höhere Priorität als »oder«. Wollen wir eine andere Auswertungsreihenfolge erzwingen, setzen wir Klammern. Das Ergebnis zusammengesetzter Ausdrücke kann mit diesen Zusatzregeln einfach ermittelt werden, indem zunächst die Wahrheitswerte der Teilausdrücke und dann sukzessive die Wahrheitswerte zusammengesetzter Ausdrücke ermittelt werden. Als Beispiel wählen wir den Ausdruck $(\overline{A} \vee B) \wedge (C \vee A)$:

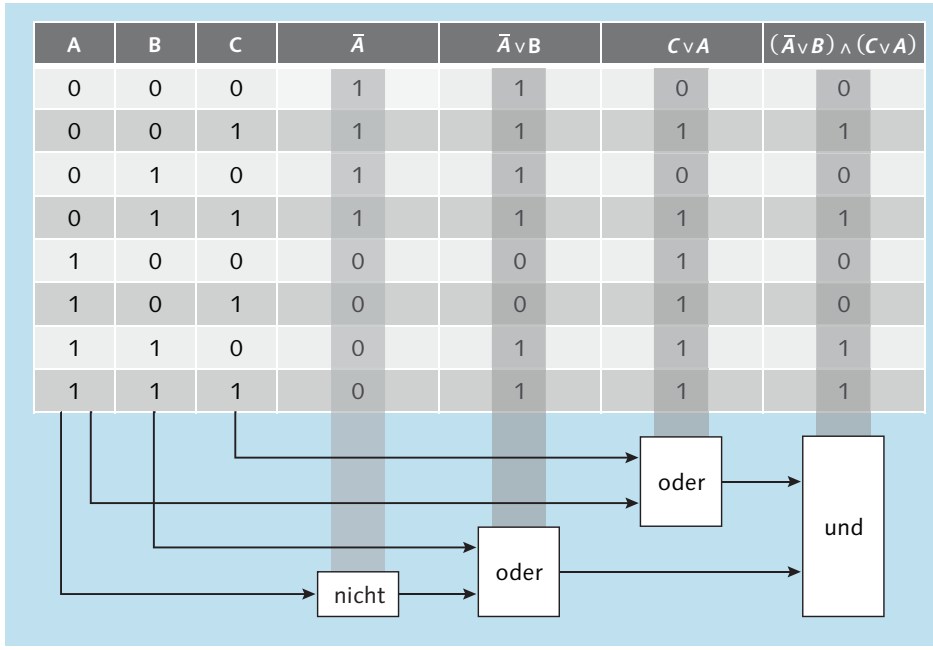


Abbildung 5.2 Wahrheitstafel eines zusammengesetzten Ausdrucks

Wenn Sie die Skizze unter der Tabelle an eine elektrische Schaltung erinnert, ist dieser Eindruck durchaus gewollt. Große Teile des Innenlebens eines Computers setzen sich aus Schaltungen zusammen, die nach den Prinzipien der Aussagenlogik arbeiten.

Von besonderem Interesse sind für uns verschiedene Ausdrücke, die die gleichen Werte in ihrer Wahrheitstabelle haben, denn solche Ausdrücke können wir in einer Formel austauschen, ohne den logischen Gehalt der Formel zu ändern. Als Beispiel betrachten wir die Ausdrücke $\overline{A \wedge B}$ bzw. $\overline{A} \vee \overline{B}$.

A	B	$\overline{A \wedge B}$	$\overline{A} \vee \overline{B}$
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

Tabelle 5.4 Wahrheitstafel gleichwertiger Ausdrücke

Beide Ausdrücke beschreiben also die gleiche logische Funktion. Das war auch zu erwarten, denn der Satz »Nicht beide Autos sind rot« ist logisch gleichwertig mit »Eines der beiden Autos ist nicht rot«. Die Sätze sind nicht gleich, aber gleichwertig. Das kennen Sie ja auch schon aus der Arithmetik: Die Formeln $(a + b)^2$ und $a^2 + 2ab + b^2$ sind auch nicht gleich (im Sinne von identisch), aber in jeder algebraischen Formel können Sie den einen Ausdruck durch den anderen ersetzen. Ebenso können Sie jetzt in jeder logischen Formel den Ausdruck $\overline{A \wedge B}$ durch $\overline{A} \vee \overline{B}$ ersetzen. Wir sprechen in diesem Zusammenhang auch von *logischer Äquivalenz* oder *Gleichheit*. Um dies in Formeln ausdrücken zu können, führen wir einen neuen Operator – den Äquivalenzoperator – ein:

A	B	A $\Leftrightarrow$ B
0	0	1
0	1	0
1	0	0
1	1	1

Tabelle 5.5 Wahrheitstafel für den Äquivalenzoperator

Um Klammern zu sparen, wollen wir vereinbaren, dass $\Leftrightarrow$ schwächer bindet als die zuvor eingeführten Operatoren. Dann können wir die Äquivalenz von $\overline{A \wedge B}$ und $\overline{A} \vee \overline{B}$ auch durch eine Formel ausdrücken:

$$\overline{A \wedge B} \Leftrightarrow \overline{A} \vee \overline{B}$$

Dieser Ausdruck ist unabhängig von den Wahrheitswerten von A und B immer wahr. Formeln, die unabhängig vom Wahrheitsgehalt der Elementaraussagen immer wahr sind, bezeichnen wir als *Tautologien*. Tautologien haben in der Aussagenlogik die gleiche Bedeutung wie wichtige Identitäten (z. B. binomische Formeln) in der Algebra. Einige wichtige Tautologien sind im Folgenden zusammengestellt:

Logische Äquivalenzen		
$A \wedge (B \wedge C) \Leftrightarrow (A \wedge B) \wedge C$	$A \vee (B \vee C) \Leftrightarrow (A \vee B) \vee C$	Assoziativgesetz
$A \wedge B \Leftrightarrow B \wedge A$	$A \vee B \Leftrightarrow B \vee A$	Kommutativgesetz
$(A \vee B) \wedge A \Leftrightarrow A$	$(A \wedge B) \vee A \Leftrightarrow A$	Verschmelzungsgesetz
$A \wedge (B \vee C) \Leftrightarrow (A \wedge B) \vee (A \wedge C)$	$A \vee (B \wedge C) \Leftrightarrow (A \vee B) \wedge (A \vee C)$	Distributivgesetz
$A \wedge (B \vee \overline{B}) \Leftrightarrow A$	$A \vee (B \wedge \overline{B}) \Leftrightarrow A$	Komplementgesetz
$A \wedge A \Leftrightarrow A$	$A \vee A \Leftrightarrow A$	Idempotenzgesetz
$\overline{A \wedge B} \Leftrightarrow \overline{A} \vee \overline{B}$	$\overline{A \vee B} \Leftrightarrow \overline{A} \wedge \overline{B}$	De Morgansches Gesetz
$A \wedge \overline{A} \Leftrightarrow 0$	$A \vee \overline{A} \Leftrightarrow 1$	
$\overline{\overline{A}} \Leftrightarrow A$		

Abbildung 5.3 Wichtige Tautologien

Diese Formeln eröffnen Ihnen die Möglichkeit, mit logischen Ausdrücken wie mit algebraischen Formeln zu rechnen. Teilweise ähneln diese Formeln sehr stark Formeln, die Sie aus der Algebra kennen.

Einen letzten Operator, den Implikationsoperator, möchten wir Ihnen noch vorstellen und dafür das Symbol $\Rightarrow$ verwenden:

A	B	A $\Rightarrow$ B
0	0	1
0	1	1
1	0	0
1	1	1

Tabelle 5.6 Wahrheitstafel des Implikationsoperators

Dieser Operator ist sehr eng mit unserer logischen Schlussfolgerungsweise (wenn A gilt, dann gilt auch B) verwandt. Trotzdem sollten Sie die Implikation nicht mit einem logischen Schluss verwechseln. Wenn Sie sagen:

Wenn Köln 1 Mio. Einwohner hat, *dann* liegt Köln in Deutschland,

ist diese Aussage im Sinne zweier mit dem Implikationsoperator verbundener Teilaussagen gemäß obiger Wahrheitstafel wahr. Keinesfalls ist damit aber gemeint, dass es eine Kausalität gibt, die besagt, dass Städte mit mehr als 1 Mio. Einwohnern immer in Deutschland liegen. Wenn Köln mehr als 1 Mio. Einwohner hätte, könnte man der Formulierung

Köln hat mehr als 1 Mio. Einwohner, *also* liegt Köln in Deutschland

sicherlich nicht zustimmen, da eine derartige Kausalität nicht besteht. Besteht allerdings eine kausale Beziehung, wie etwa in

wenn eine Zahl kleiner als 5 ist, *dann* ist sie auch kleiner als 10,

dann gilt auch die Implikation für alle konkret eingesetzten Zahlen, da der Fall einer wahren Aussage links und einer falschen Aussage rechts vom Implikationspfeil durch den Kausalzusammenhang ausgeschlossen ist.

Beachten Sie, dass wir eine Implikation als wahr definiert haben, wenn die Prämisse – das ist die Aussage links vom Implikationspfeil – falsch ist; und zwar völlig unabhängig davon, was auf der rechten Seite folgt. Auch hier gibt es oft Widerspruch, weil die Implikation unausgesprochen als Äquivalenz verstanden wird. Ein in diesem Zusammenhang häufig zu beobachtender Fehler ist es, dass die Aussagen $A \Rightarrow B$ und $\overline{A} \Rightarrow \overline{B}$ als gleichwertig angesehen werden. Eine Betrachtung der Wahrheitstafeln zeigt aber, dass eine solche Gleichsetzung falsch ist. Wenn ich z. B. sage:

Wenn morgen die Sonne scheint, dann gehe ich ins Schwimmbad,

dann heißt das nicht, dass ich bei Regen nicht ins Schwimmbad gehe. Ich habe mich für diesen Fall nicht festgelegt. Hätte ich mich auch in diesem Fall festlegen wollen, hätte ich eine Äquivalenzaussage formulieren müssen:

Ich gehe morgen genau dann ins Schwimmbad, wenn die Sonne scheint.

Aber wer formuliert schon so gestelzt?

Auch für die Implikation gilt eine Reihe von Rechenregeln. Die drei vielleicht wichtigsten zeigt die folgende Tabelle:

$(A \Longrightarrow B) \Leftrightarrow (\overline{A} \vee B)$
$(A \Longrightarrow B) \Leftrightarrow (\overline{B} \Longrightarrow \overline{A})$
$(A \Leftrightarrow B) \Leftrightarrow (A \Longrightarrow B) \wedge (B \Longrightarrow A)$

Abbildung 5.4 Rechenregeln für die Implikation

Die erste Regel zeigt, wie wir eine Implikation durch »nicht« und »oder« ausdrücken können. Die zweite ermöglicht, eine Implikation »rückwärts« zu lesen. Die dritte formuliert einen naheliegenden Zusammenhang zwischen Implikation und Äquivalenz.

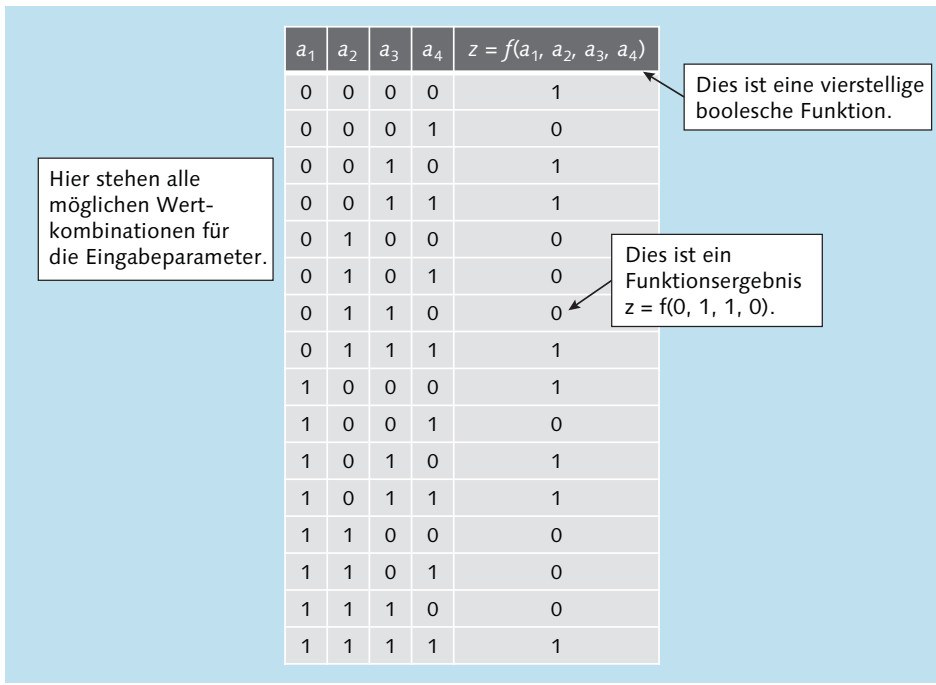
Jetzt können Sie übrigens die Eingangsfrage dieses Kapitels beantworten: »Wenn fünf Ochsen in fünf Minuten fünf Liter Milch geben, dann gibt es den Osterhasen«. Dieser Satz ist aussagenlogisch wahr, was aber keine Auswirkungen auf die Milchproduktion von Ochsen oder die Existenz des Osterhasen hat.

5.3 Boolesche Funktionen

Bevor wir uns wieder der Programmierung zuwenden, wollen wir uns Gedanken darüber machen, wie weit die logischen Operatoren uns denn tragen. Ein logischer Operator realisiert eine Funktion, und die Wahrheitstafel ist eigentlich nur eine vollständige Wertetabelle dieser Funktion. Wenn Sie sich z. B. die Funktion

$z = f(x,y) = x \wedge y$

anschauen, dann handelt es sich um eine Funktion, die zwei logische Werte (x und y) übergeben bekommt und daraus einen logischen Wert (z) berechnet. So eine Funktion nennen wir eine zweistellige boolesche Funktion.



a_1	a_2	a_3	a_4	$z = f(a_1, a_2, a_3, a_4)$
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Abbildung 5.5 Eine vierstellige boolesche Funktion

Allgemein können wir *n*-stellige boolesche Funktionen betrachten. Letztlich ist eine *n*-stellige boolesche Funktion durch eine Tabelle mit *n* Eingabespalten und einer Ausgabespalte gegeben. In der Tabelle stehen nur 0 und 1 (siehe Abbildung 5.5).

Die Anzahl der Zeilen einer solchen Tabelle ist abhängig von der Anzahl der Eingabespalten. Bei vier Eingabespalten haben wir 16 Zeilen. Die Zahl der Zeilen verdoppelt sich mit jeder hinzukommenden Spalte, sodass wir bei *n* Spalten 2^n Zeilen in der Tabelle haben. In jeder Zeile können wir dann einen Funktionswert angeben, sodass wir insgesamt $2^{(2^n)}$ *n*-stellige boolesche Funktionen aufstellen können. Wenn Sie noch kein Gefühl für das Wachstum dieser Zahl haben, dann betrachten Sie die Tabelle in Abbildung 5.6.

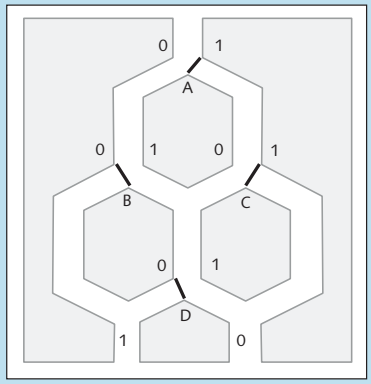
n	$2^{(2^n)}$
0	2
1	4
2	16
3	256
4	65536
5	4294967296
6	1,84467E+19
7	3,40282E+38
8	1,15792E+77
9	1,3408E+154

So viele fünfstellige boolesche Funktionen gibt es.

Abbildung 5.6 Anzahl n-stelliger boolescher Funktionen

Trotz dieser schieren Menge sind wir in der Lage, alle booleschen Funktionen mit unseren drei logischen Grundoperatoren »nicht«, »und« und »oder« zu berechnen. Wie das geht, zeige ich Ihnen an einem Beispiel. Dabei sollten Sie darauf achten, dass sich das Vorgehen problemlos auf jedes beliebige andere Beispiel übertragen lässt.

Wir betrachten das Kugelspiel aus dem letzten Kapitel.



A	B	C	D	z = f(A,B,C,D)	
0	0	0	0	0	
0	0	0	1	0	
0	0	1	0	0	
0	0	1	1	1	$\bar{A} \wedge \bar{B} \wedge C \wedge D$
0	1	0	0	0	
0	1	0	1	0	
0	1	1	0	0	
0	1	1	1	1	$\bar{A} \wedge B \wedge C \wedge D$
1	0	0	0	0	
1	0	0	1	1	$A \wedge \bar{B} \wedge \bar{C} \wedge D$
1	0	1	0	0	
1	0	1	1	1	$A \wedge \bar{B} \wedge C \wedge D$
1	1	0	0	1	$A \wedge B \wedge \bar{C} \wedge \bar{D}$
1	1	0	1	1	$A \wedge B \wedge \bar{C} \wedge D$
1	1	1	0	1	$A \wedge B \wedge C \wedge \bar{D}$
1	1	1	1	1	$A \wedge B \wedge C \wedge D$

$$z = \bar{A} \wedge \bar{B} \wedge C \wedge D \vee \bar{A} \wedge B \wedge C \wedge D \vee A \wedge \bar{B} \wedge \bar{C} \wedge D \vee A \wedge \bar{B} \wedge C \wedge D \vee A \wedge B \wedge \bar{C} \wedge \bar{D} \vee A \wedge B \wedge \bar{C} \wedge D \vee A \wedge B \wedge C \wedge \bar{D} \vee A \wedge B \wedge C \wedge D$$

Abbildung 5.7 Darstellung des Kugelspiels

Für das Spiel erstellen Sie eine Wahrheitstafel, indem Sie alle 16 möglichen Weichenstelllungen in Gedanken durchspielen und den Auslauf notieren. Dann betrachten Sie in der Tabelle die Zeilen, in denen Sie eine 1 als Ergebnis erhalten haben. Für diese Zeilen gibt es eine einfache Darstellung ausschließlich mit »und« und »nicht«. Am Ende sammeln Sie diese Terme durch eine Oder-Verknüpfung ein. Jede 1 in der Wertespalte der Funktionstabelle triggert damit genau einen Term, der eine 1 erzeugt. Diese 1 sorgt dann dafür, dass sich in dieser Situation insgesamt eine 1 als Funktionsergebnis ergibt.

Um die Lesbarkeit unserer Formeln zu verbessern, lassen wir das $\wedge$-Zeichen in den Formeln einfach weg und erhalten:

$$z = \bar{A}\bar{B}CD \vee \bar{A}BCD \vee A\bar{B}\bar{C}D \vee A\bar{B}CD \vee AB\bar{C}\bar{D} \vee AB\bar{C}D \vee ABC\bar{D} \vee ABCD$$

Diese Formel ist relativ komplex, und Sie können versuchen, sie zu vereinfachen. Dazu gibt es Techniken, wie z. B. die sogenannten *Karnaugh-Diagramme*, die wir hier aber nicht behandeln werden. Letztlich werden boolesche Funktionen mit zunehmender Stellenzahl so komplex, dass sie sich nur noch mit Computerunterstützung optimieren lassen. Die Optimierung komplexer boolescher Funktionen ist ein ganz

wichtiger Aspekt beim Entwurf digitaler Schaltungen – ohne Computerunterstützung könnte man solche Schaltungen heute nicht mehr entwickeln. Die wichtige Erkenntnis für uns ist ja, dass wir jede boolesche Funktion – und sei sie noch so komplex – mit unseren drei Grundoperatoren[2] realisieren können.

Vielleicht finden wir ja eine Ad-hoc-Vereinfachung, wenn wir noch einmal einen Blick auf das Spiel werfen:

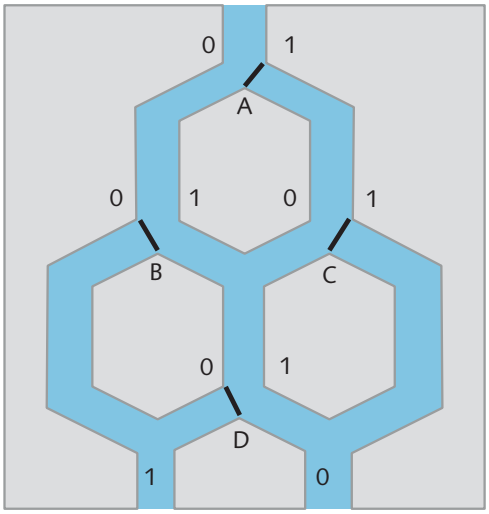


Abbildung 5.8 Suche nach Vereinfachungen

Wir müssen untersuchen, wann die Kugel den Ausgang 1 nimmt. Sie sehen, dass, wenn A = 1 und B = 1 ist, alle Kugeln zum Ausgang 1 gelenkt werden, egal, wie die beiden anderen Weichen stehen. Wenn A = 1 und B = 0 ist oder wenn A = 0 und C = 1 ist, geht die Kugel durch die Mitte, und Weiche D entscheidet, wo sie letztlich hingeht. In allen anderen Fällen ist der Ausgang 0.

$$z = A\,B \vee (A\,\overline{B} \vee \overline{A}\,C)\,D$$

Im nächsten Abschnitt erfahren Sie, wie Sie in C boolesche Funktionen erstellen, danach werden wir dieses Spiel programmieren.

5.4 Logische Operatoren in C

Als Variable für boolesche Werte können Sie in C einfach `int` verwenden. Als logische Operatoren gibt es nur das »nicht«, das »und« und das »oder«, aber Sie wissen ja

2 Wenn Sie sich schon einmal mit digitalen Schaltungen beschäftigt haben, wissen Sie, dass es sogar einen einzigen Operator gibt, mit dem man alle Schaltungen aufbauen kann; wenn nicht, dann versuchen Sie, diesen Operator zu finden.

schon, dass Sie mehr nicht brauchen. C verwendet die folgenden Zeichen für die logischen Operatoren:

Operator	Darstellung in C
nicht	!
und	&&
oder	||

Tabelle 5.7 Logische Operatoren in C

In der Auswertung boolescher Ausdrücke folgt C den Regeln, die wir oben bereits aufgestellt haben: ! vor && vor ||. Im Zweifel setzen Sie Klammern.

Das ist eigentlich schon alles, was Sie wissen müssen, um mit logischen Ausdrücken zu programmieren.

5.5 Beispiele

Unsere Kenntnisse über die boolesche Algebra und die logischen Operatoren in C fassen wir jetzt zusammen, um zwei kleine Programmieraufgaben zu lösen.

5.5.1 Kugelspiel

Für das Kugelspiel hatten wir zwei Lösungsformeln hergeleitet:

$$z = \overline{A}\,\overline{B}CD \vee \overline{A}B\overline{C}D \vee A\overline{B}\,\overline{C}D \vee A\overline{B}CD \vee AB\overline{C}\,\overline{D} \vee AB\overline{C}D \vee ABC\overline{D} \vee ABCD$$

Und

$$z = AB \vee (A\overline{B} \vee \overline{A}C)D$$

Beide lassen sich einfach in C-Code umsetzen. Wir wollen beide Lösungen vergleichen und geben dazu die gesamte Funktionstabelle mit den beiden berechneten Werten aus:

```
void main()
    {
    int A, B, C, D;
    int z1, z2;
```

```
for( A = 0; A <= 1 ; A++)
    {
    for( B = 0; B <= 1 ; B++)
        {
        for( C = 0; C <= 1 ; C++)
            {
            for( D = 0; D <= 1 ; D++)
                {
                z1 = A&&B || (A&&!B || !A&&C) && D;
                z2 =  !A&&!B&&C&&D || !A&&B&&C&&D  || A&&!B&&!C&&D ||
                       A&&!B&&C&&D || A&&B&&!C&&!D || A&&B&&!C&&D  ||
                       A&&B&&C&&!D || A&&B&&C&&D;
                printf( "%d %d %d %d | %d %d\n", A, B, C, D, z1, z2);
                }
            }
        }
    }
```

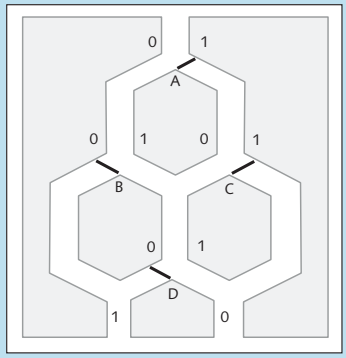
Listing 5.1 Erstellung aller Kombinationen

Neu ist für Sie vielleicht die Technik, mit der hier durch vier ineinander geschachtelte Schleifen die Tabelle erzeugt wird. Aber das ist ganz einfach:

- `A` durchläuft die Werte 0 und 1.
- Für jeden Wert von `A` durchläuft dann `B` die Werte 0 und 1. Damit ergeben sich alle Wertekombinationen von `A` und `B`.
- Für jede Wertekombination von `A` und `B` durchläuft dann `C` die Werte 0 und 1. Damit ergeben sich alle Dreierkombinationen.
- Für jede Dreierkombination durchläuft dann `D` die Werte 0 und 1. Damit ergeben sich alle Viererkombinationen.

Bei diesem Prozess bewegt sich `A` am trägsten und `D` am hektischsten. Auf diese Weise entstehen die Kombinationen genau in der Reihenfolge, in der wir sie bisher auch immer notiert haben, und wir sehen, dass die in der innersten Schleife berechneten logischen Werte exakt den Erwartungen entsprechen (siehe Abbildung 5.9).

Das Umschlagen der Weichen können wir hier übrigens nicht mehr simulieren, da dieses Verhalten in diesem booleschen Modell nicht mehr abgebildet ist.



A	B	C	D	z = f(A,B,C,D)
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

```
0 0 0 0 : 0 0
0 0 0 1 : 0 0
0 0 1 0 : 0 0
0 0 1 1 : 1 1
0 1 0 0 : 0 0
0 1 0 1 : 0 0
0 1 1 0 : 0 0
0 1 1 1 : 1 1
1 0 0 0 : 0 0
1 0 0 1 : 1 1
1 0 1 0 : 0 0
1 0 1 1 : 1 1
1 1 0 0 : 1 1
1 1 0 1 : 1 1
1 1 1 0 : 1 1
1 1 1 1 : 1 1
```

Abbildung 5.9 Berechnete Werte für das Kugelspiel

5.5.2 Schaltung

Als eine weitere Anwendung der Aussagenlogik wollen wir ein Programm schreiben, das alle Schalterstellungen, bei denen in der folgenden Schaltung die Lampe leuchtet, tabellarisch ausgibt.

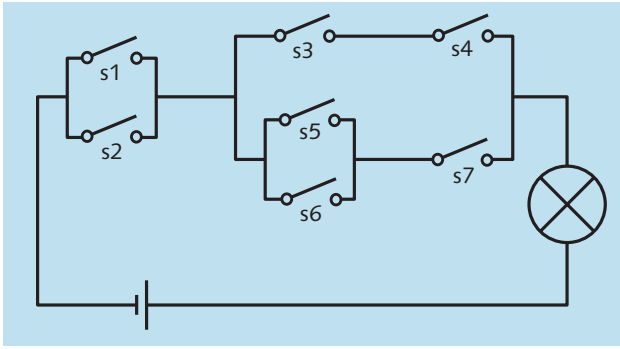


Abbildung 5.10 Beispiel für eine Lampenschaltung

Wir verwenden für jeden der Schalter S1–7 eine Variable, die jeweils die Werte 0 oder 1 annehmen kann. Dabei bedeutet:

1 – Der Schalter ist geschlossen.
0 – Der Schalter ist geöffnet.

Zusätzlich wissen wir:

- Hintereinanderliegende Schalter realisieren eine *Und*-Verbindung.
- Parallel liegende Schalter realisieren eine *Oder*-Verbindung.

Für unsere Schaltung bedeutet dies:

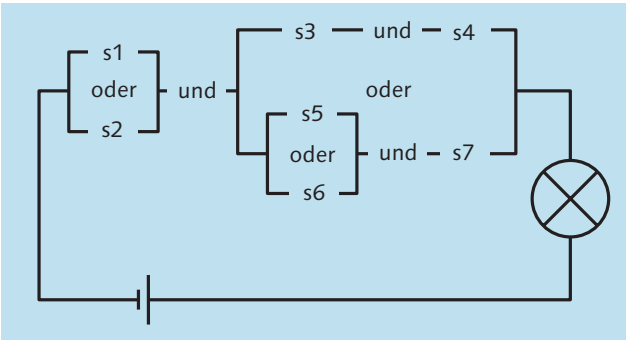


Abbildung 5.11 Umsetzung der Schaltung

Damit können wir den Zustand der Lampe (1 = an, 0 = aus) als eine boolesche Funktion der Schalterstellungen darstellen. In C-Notation erhalten wir also:

```
lampe = (s1 || s2) && ((s3 && s4) || ((s5 || s6) && s7))
```

Jetzt müssen wir alle möglichen Schalterstellungen generieren und dann jeweils prüfen, ob die Lampe brennt. Alle 128 möglichen Schalterstellungen erzeugen wir, indem wir in sieben ineinander geschachtelten Zählschleifen alle Schalter jeweils auf 0 bzw. 1 setzen. Diese Methode kennen Sie bereits aus der letzten Aufgabe.

```
void main()
    {
    int s1, s2, s3, s4, s5, s6, s7;
    int lampe;
    printf( "s1 s2 s3 s4 s5 s6 s7\n");
    for( s1 = 0; s1 <= 1; s1 = s1 + 1)
      {
      for( s2 = 0; s2 <= 1; s2 = s2 + 1)
        {
        for( s3 = 0; s3 <= 1; s3 = s3 + 1)
          {
          for( s4 = 0; s4 <= 1; s4 = s4 + 1)
            {
            for( s5 = 0; s5 <= 1; s5 = s5 + 1)
              {
```