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.
– 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:

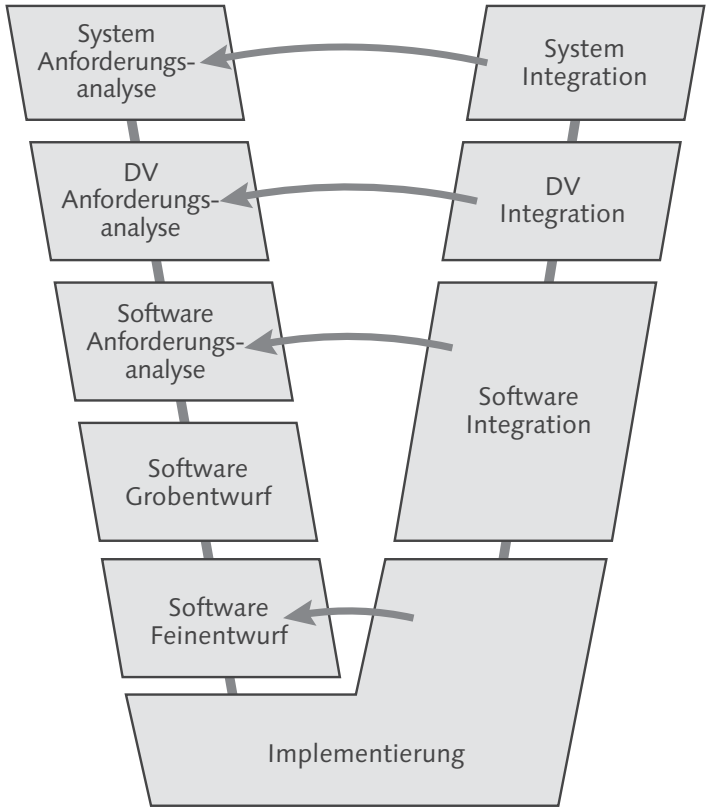


Abbildung 2.1 Das V-Modell

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.

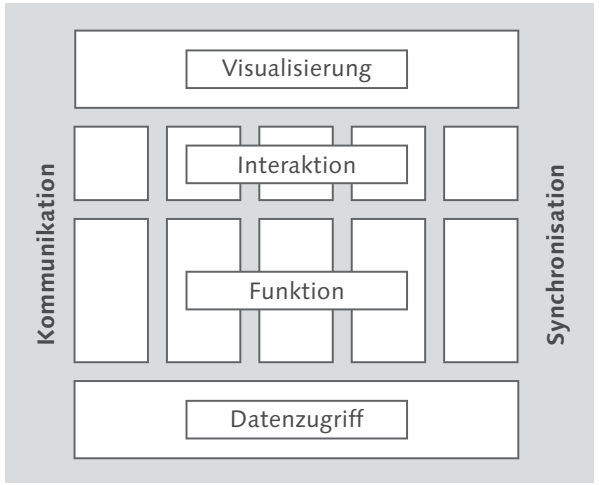


Abbildung 2.2 Schichten eines Softwaresystems

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:

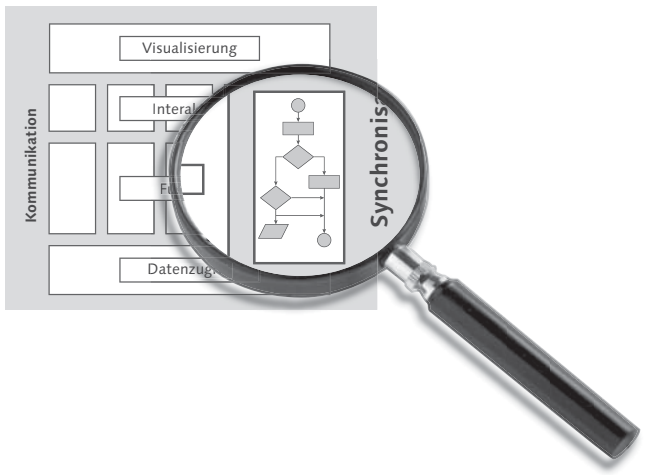


Abbildung 2.3 Realisierung von Anwendungsfunktionen

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:

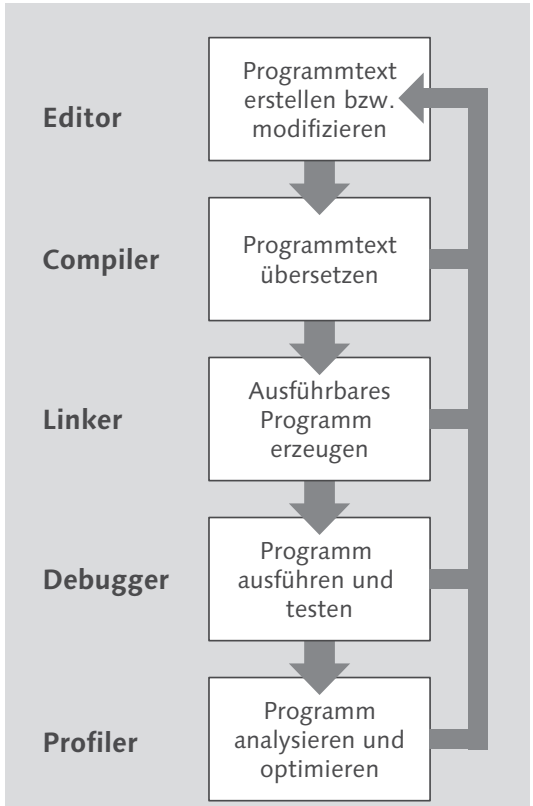


Abbildung 2.4 Arbeitsschritte bei der Programmierung

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:

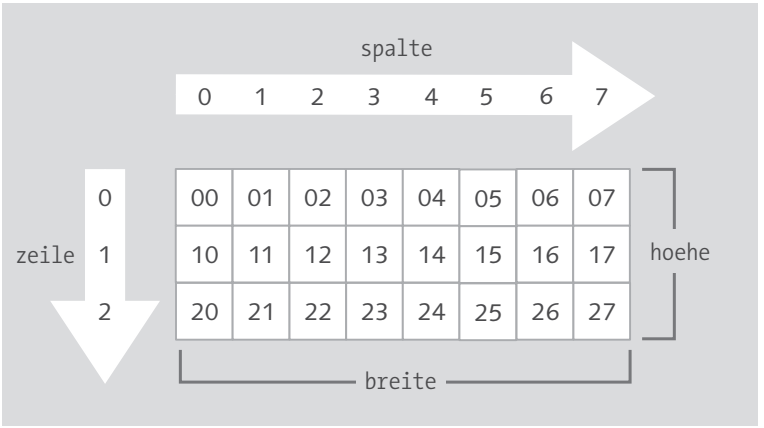


Abbildung 3.1 Beispiel einer zweidimensionalen Struktur

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

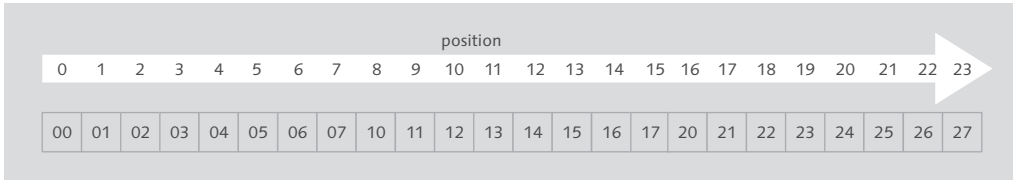


Abbildung 3.2 Beispiel einer eindimensionalen Struktur

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:

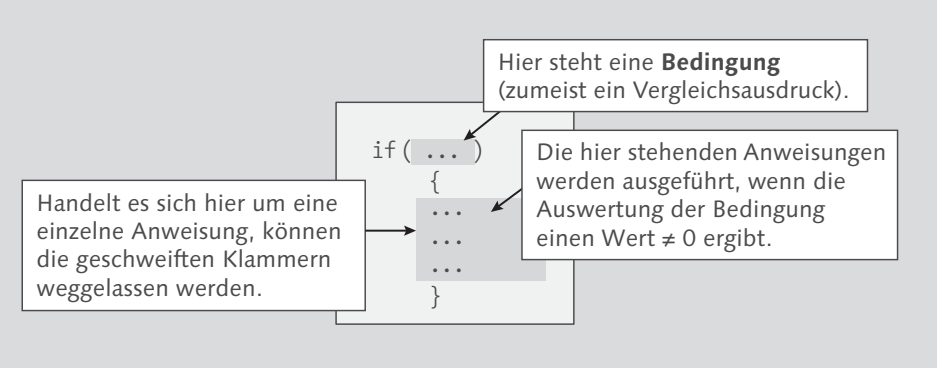


Abbildung 3.3 Bedingte Befehlsausführung

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:

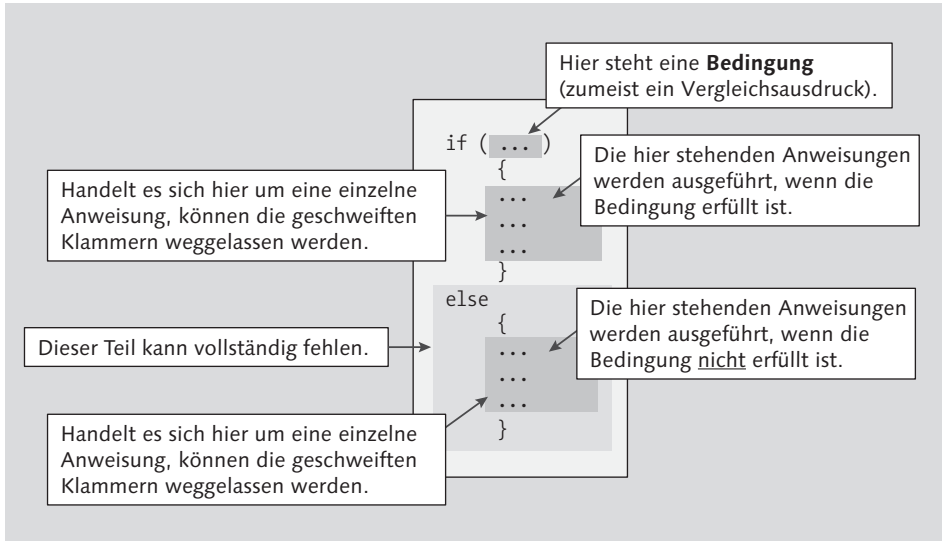


Abbildung 3.4 Die vollständige if-Anweisung

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:

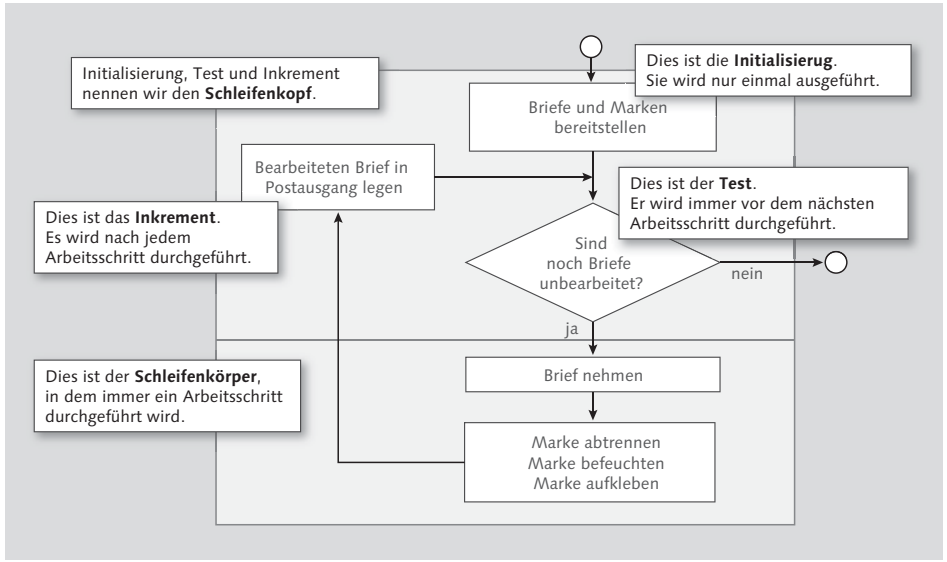


Abbildung 3.5 Flussdiagramm »Briefe frankieren«

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:

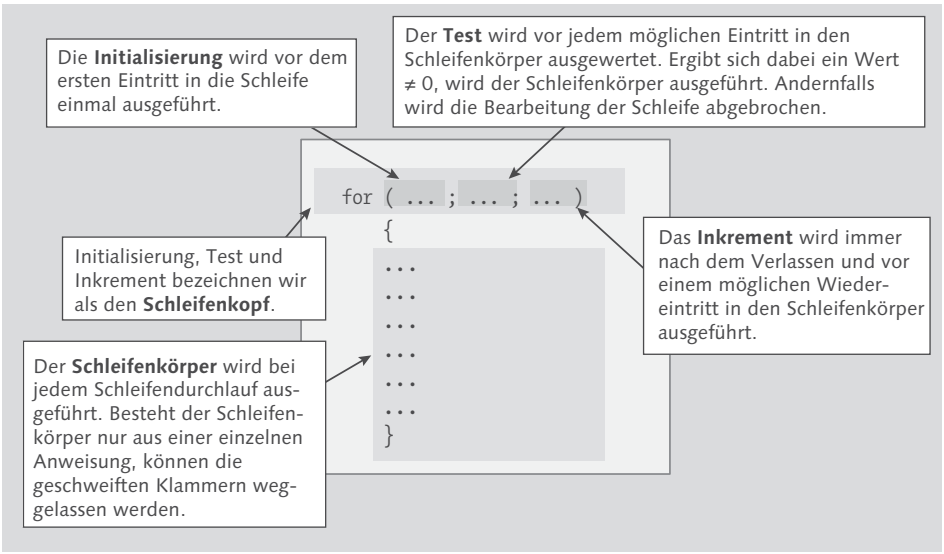


Abbildung 3.6 Die for-Anweisung

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:

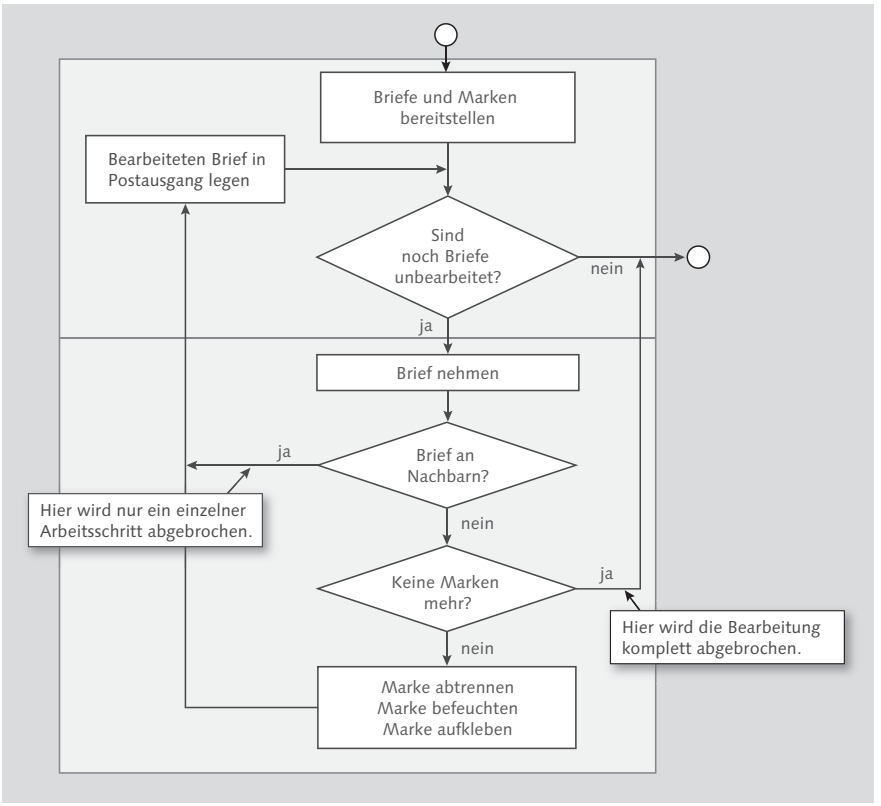


Abbildung 3.7 Das erweiterte Flussdiagramm

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:

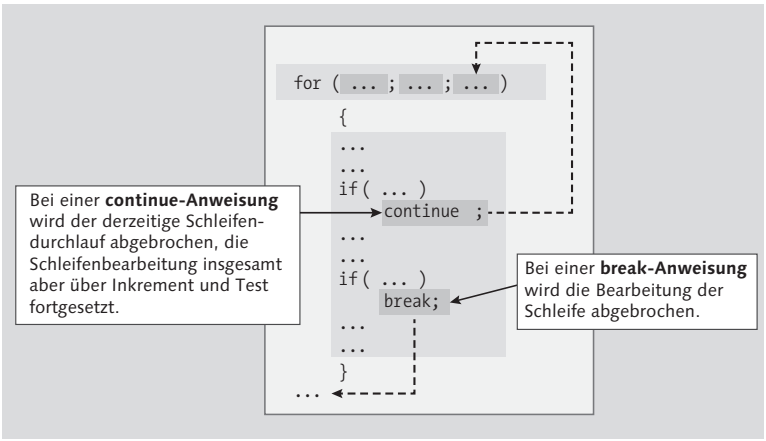


Abbildung 3.8 Schleifensteuerung innerhalb der for-Anweisung

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 Kontrollstrukturen 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:

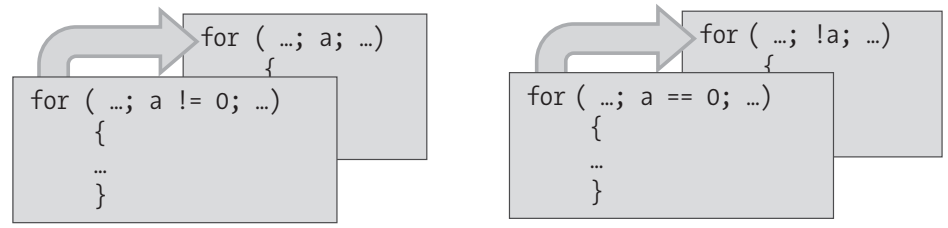


Abbildung 3.9 Vereinfachung der Testbedingung

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:

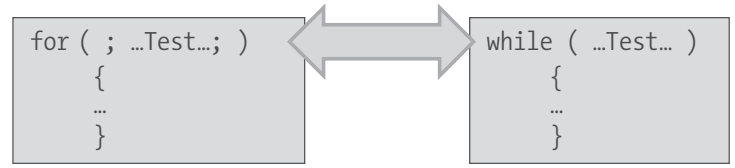


Abbildung 3.10 Ersatz der for- durch eine while-Anweisung

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

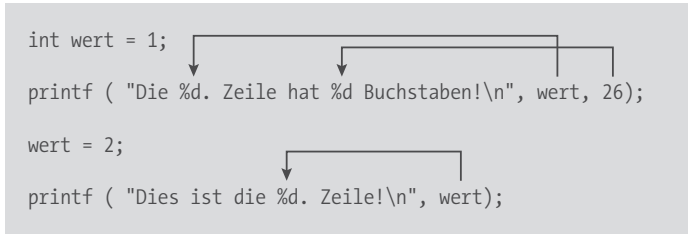


Abbildung 3.11 Programmfragment zur Ausgabe

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.

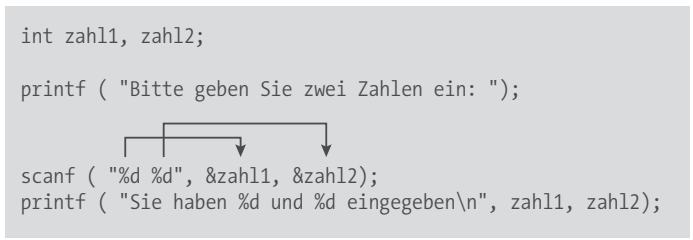


Abbildung 3.12 Einlesen von Werten

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.

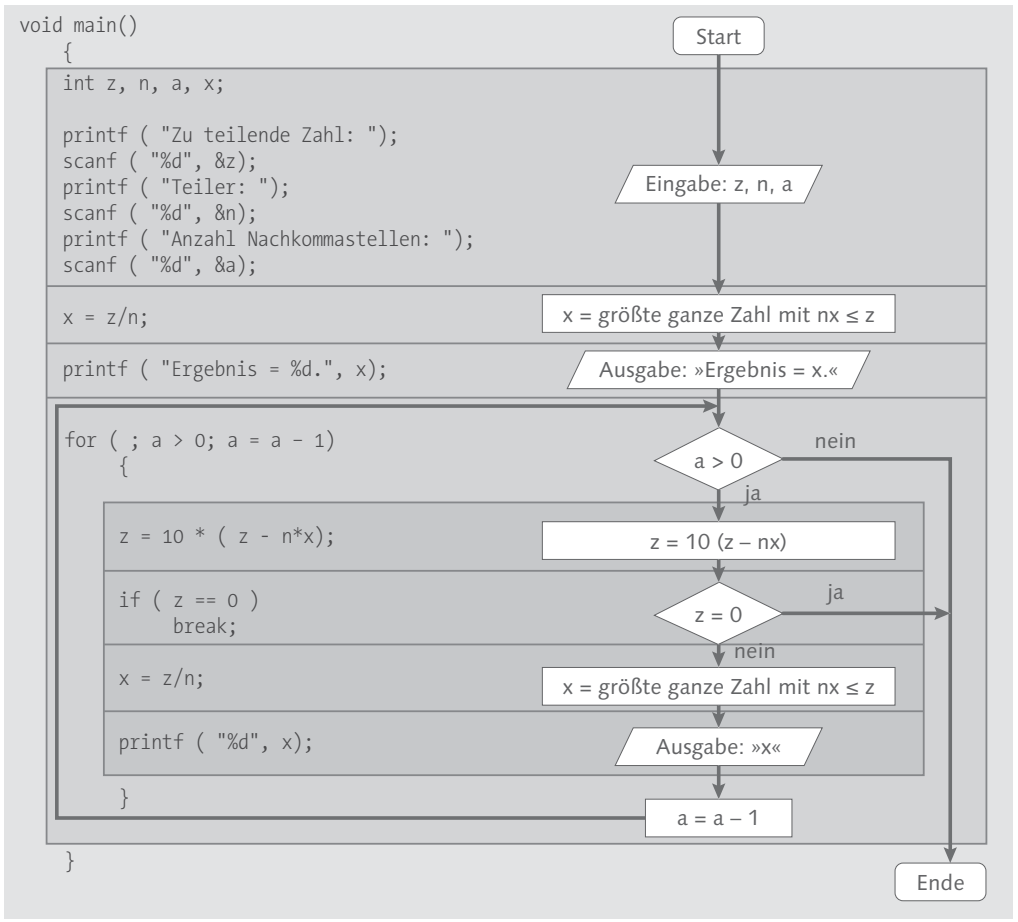


Abbildung 3.13 Flussdiagramm der schriftlichen Division

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:

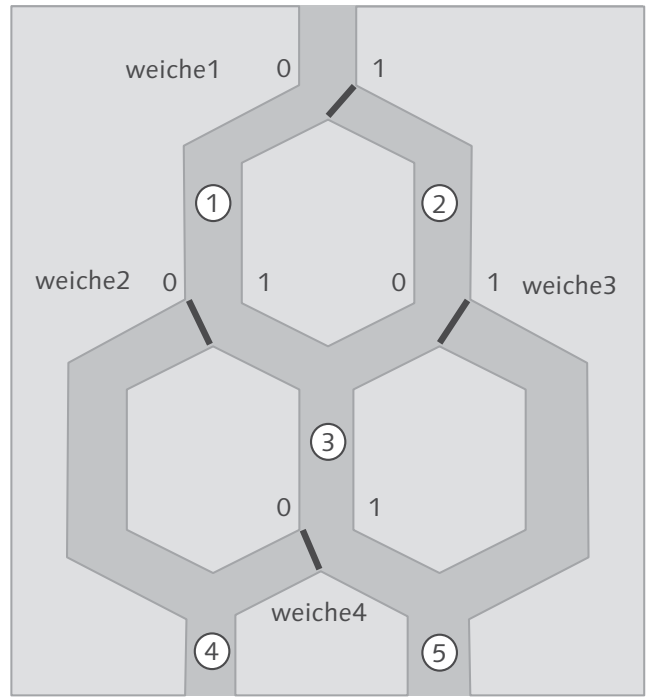


Abbildung 3.14 Das Kugelspiel

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:

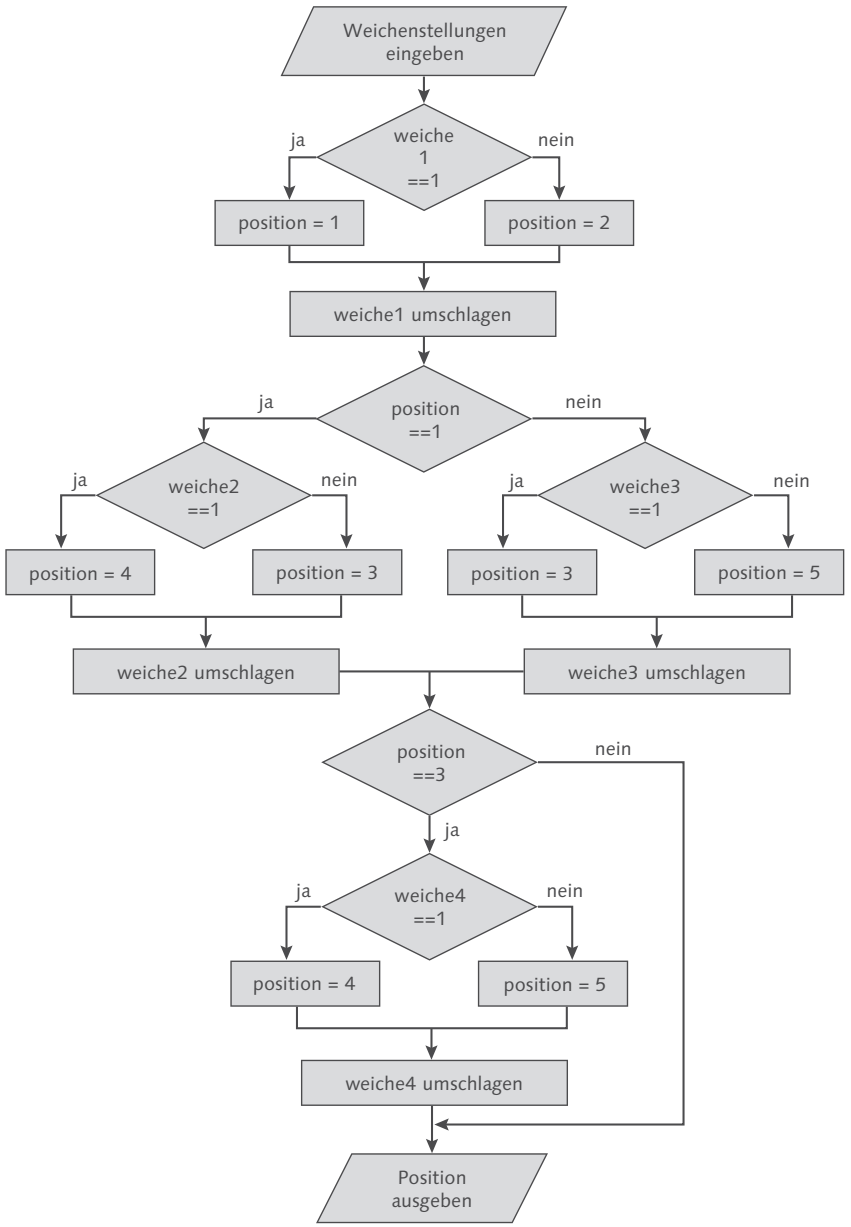


Abbildung 3.15 Flussdiagramm des Kugelspiels

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:

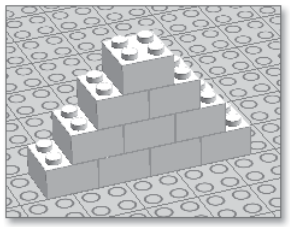


Abbildung 3.16 Treppe aus Legosteinen

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 6

Elementare Datentypen und ihre Darstellung

Das Buch der Natur ist mit mathematischen Symbolen geschrieben.
– Galileo Galilei

Jemand bittet Sie, sich eine geheime Zahl zwischen 0 und 31 zu denken, und legt Ihnen dann nacheinander die folgenden fünf Karten vor:

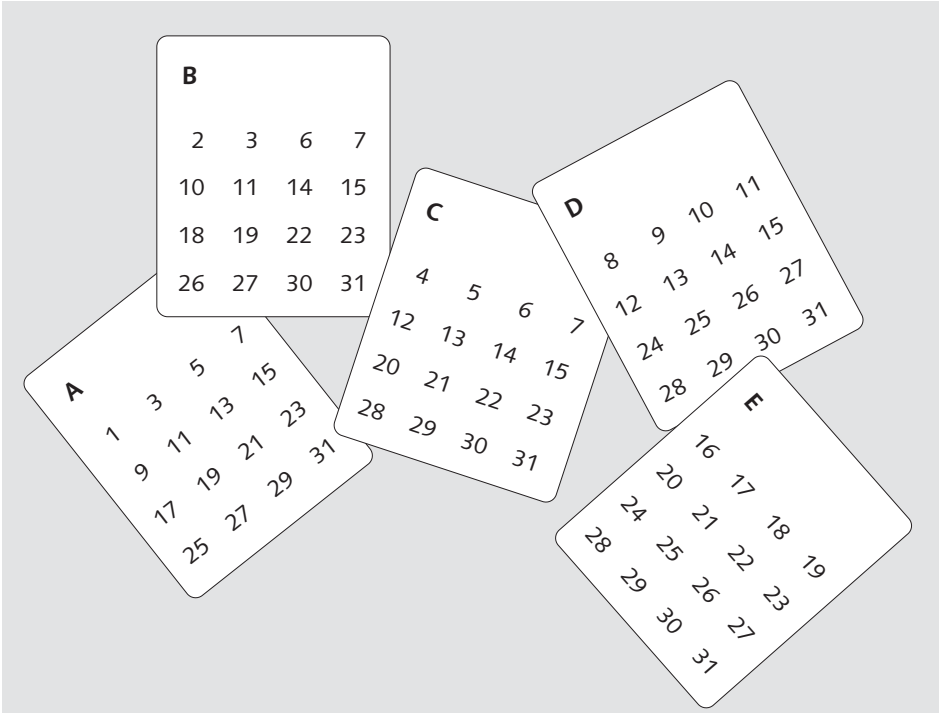


Abbildung 6.1 Zahlenraten

Er fordert Sie auf, jeweils zu sagen, ob die gedachte Zahl auf der Karte steht oder nicht. Nachdem Sie die Fragen beantwortet haben, nennt er Ihnen, ohne lange zu zögern, Ihre Geheimzahl.

Versuchen Sie, hinter diesen Trick zu kommen. Wenn es Ihnen nicht gelingt, dann lesen Sie aufmerksam das folgende Kapitel, denn Sie erfahren dort mehr über den Hintergrund dieses Tricks. Im Laufe dieses Kapitels werden wir den Trick auflösen und Ihnen zeigen, wie Sie ihn programmieren.

6.1 Zahlendarstellungen

Zahlen sind abstrakte mathematische Objekte. Damit man sie konkret benutzen kann, brauchen sie eine »Benutzerschnittstelle«, mittels derer man sie addieren, multiplizieren oder vergleichen kann. Die denkbar einfachste Benutzerschnittstelle erhält man, wenn man für die Eins einen Strich und für jede folgende Zahl jeweils einen weiteren Strich macht. Solche Darstellungen werden allerdings sehr schnell unübersichtlich, sodass man zur besseren Lesbarkeit Gruppierungen einführen muss:

Abbildung 6.2 Zahlendarstellung und Gruppierung

Dieses Strichsystem kennt im Prinzip nur eine Operation (Addition von 1) und hat in dieser Beschränkung durchaus seine Vorteile, sodass wir es heute noch – z. B. auf Bierdeckeln – verwenden. Wirklich große Zahlen lassen sich dadurch allerdings nicht darstellen, sodass man gezwungen ist, zur Abkürzung zusätzliche Symbole einzuführen. So ist es z. B. im römischen Zahlensystem, aber rechnen Sie bitte mal CCCLXXVII + DCXXIII. Das römische Zahlensystem verwendet man heute nur noch aus nostalgischen Gründen – etwa auf den Zifferblättern von Uhren oder für Jahreszahlen in Kalendern.

Heute werden zur Darstellung von Zahlen sogenannte *Stellenwertsysteme* verwendet. Im Alltag nutzen wir das Dezimalsystem:

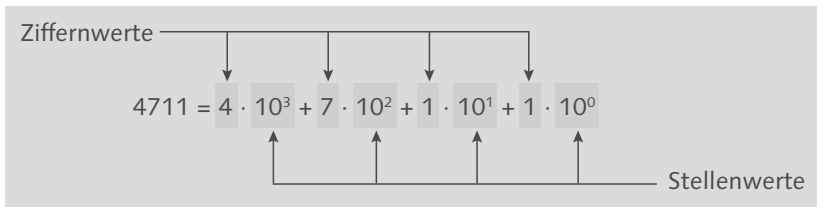


Abbildung 6.3 Darstellung im Dezimalsystem

Diese Darstellung beruht auf der Zahl 10 als *Basis*. Im Grunde genommen müssten Sie die Basiszahl an der Ziffernfolge vermerken (z. B. 4711_{10}), denn nur mithilfe der Basis können Sie aus der Ziffernfolge den Zahlenwert rekonstruieren.

Sie können jede andere natürliche Zahl größer als 1 als Basis verwenden – z. B. die Zahl 7. Sie erhalten dann nur eine andere Ziffernfolge:

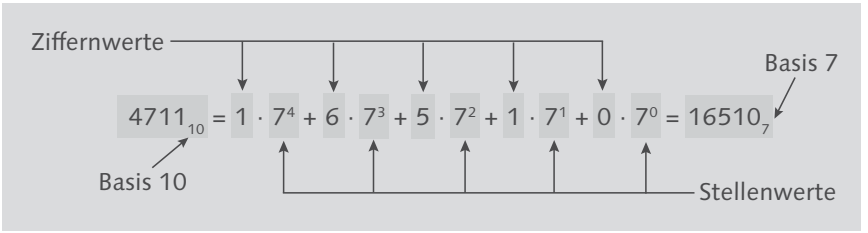


Abbildung 6.4 Darstellung im 7er-System

Wie kommen Sie zu dieser neuen Ziffernfolge? Wir formulieren den oben genannten Ausdruck durch Ausklammern um:

$$4711_{10} = \Big(\big((1 \cdot 7 + 6) \cdot 7 + 5\big) \cdot 7 + 1\Big) \cdot 7 + 0$$

Jetzt sehen Sie, dass Sie die Ziffernwerte erhalten, indem Sie die Reste bei Division durch 7 betrachten. Also dividieren Sie 4711 fortlaufend durch die Basiszahl 7 und notieren sich die Reste. Das ergibt die gesuchte Ziffernfolge – allerdings in umgekehrter Reihenfolge, da die niederwertigste Ziffer zuerst berechnet wird:

```
4711 =  673 · 7 +       0
 673 =   96 · 7 +     1
  96 =   13 · 7 +   5
  13 =    1 · 7 + 6
   7 =    0 · 7 + 1
                1 6 5 1 0
```

Abbildung 6.5 Umrechnung auf eine andere Basis

Eigentlich ist im 7er-System alles genauso wie im 10er-System, außer dass es nur die Ziffern 0–6[1] gibt und dass beim Zählen immer bei 6 ein Übertrag erfolgt.

10er-System	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
7er-System	0	1	2	3	4	5	6	10	11	12	13	14	15	16	20	21	22

Abbildung 6.6 Vergleich des Dezimalsystems und des 7er-Systems

Auch rechnen können Sie im 7er-System genauso gut wie im 10er-System. Die vermeintliche Überlegenheit des Dezimalsystems ergibt sich daraus, dass wir durch jahrelange Übung eine große Vertrautheit mit diesem System erworben haben und alle

1 Divisionsreste größer als 6 können bei einer Division durch 7 ja nicht vorkommen.

konkreten Rechenprozesse des Alltags (z. B. Münzen und Geldscheine im Zahlungsverkehr) auf dieses System ausgerichtet sind. Sie müssen in den verschiedenen Stellenwertsystemen nicht perfekt rechnen können, sollten aber zumindest von einem Stellenwertsystem in ein anderes umrechnen können.

Um uns die Arbeit des Umrechnens zu erleichtern, schreiben wir ein kleines Programm, das die oben beschriebene fortlaufende Division durch die Basis vornimmt:

```
   void main()
       {
A      int basis = 7;
       int zahl = 4711;
       int z;

       printf( "Basis: %d\n", basis);
       printf( "------------------\n");

B      for( z = zahl ; z != 0; z = z/basis)
C          printf( "%4d = %4d*%d + %2d\n", z, z/basis, basis, z%basis);

       printf( "  ------------------\n\n");
       }
```

Listing 6.1 Programm zur Umrechnung auf andere Basen

Die Basis, auf die umgerechnet werden soll, ist 7, kann aber geändert werden (A). Im Verlauf des Programms wird die umzurechnende Zahl so lange durch die Basis geteilt, bis nichts mehr übrig bleibt (B), und in der Schleife wird jeweils eine Zeile ausgegeben (C), sodass wir folgendes Ergebnis erhalten:

```
Basis: 7
------------------
4711 =   673*7 + 0
 673 =    96*7 + 1
  96 =    13*7 + 5
  13 =     1*7 + 6
   1 =     0*7 + 1
```

Eigentlich besteht dieses Programm fast nur aus Ausgaben. Die wesentlichen Berechnungen verstecken sich in den beiden Ausdrücken `z/basis` und `z%basis`, in denen der ganzzahlige Quotient und der Rest ermittelt werden. Wir spielen alle Basen von 2 bis 9 durch und erhalten die folgenden Ergebnisse:


```
Basis: 2
----------------
4711 = 2355*2 + 1
2355 = 1177*2 + 1
1177 =  588*2 + 1
 588 =  294*2 + 1
 294 =  147*2 + 0
 147 =   73*2 + 1
  73 =   36*2 + 1
  36 =   18*2 + 0
  18 =    9*2 + 0
   9 =    4*2 + 1
   4 =    2*2 + 0
   2 =    1*2 + 0
   1 =    0*2 + 1
```

```
Basis: 3
----------------
4711 = 1570*3 + 1
1570 =  523*3 + 1
 523 =  174*3 + 1
 174 =   58*3 + 0
  58 =   19*3 + 1
  19 =    6*3 + 1
   6 =    2*3 + 0
   2 =    0*3 + 2
```

```
Basis: 4
----------------
4711 = 1177*4 + 3
1177 =  294*4 + 1
 294 =   73*4 + 2
  73 =   18*4 + 1
  18 =    4*4 + 2
   4 =    1*4 + 0
   1 =    0*4 + 1
```

```
Basis: 5
----------------
4711 =  942*5 + 1
 942 =  188*5 + 2
 188 =   37*5 + 3
  37 =    7*5 + 2
   7 =    1*5 + 2
   1 =    0*5 + 1
```

```
Basis: 6
----------------
4711 =  785*6 + 1
 785 =  130*6 + 5
 130 =   21*6 + 4
  21 =    3*6 + 3
   3 =    0*6 + 3
```

```
Basis: 7
----------------
4711 =  673*7 + 0
 673 =   96*7 + 1
  96 =   13*7 + 5
  13 =    1*7 + 6
   1 =    0*7 + 1
```

```
Basis: 8
----------------
4711 =  588*8 + 7
 588 =   73*8 + 4
  73 =    9*8 + 1
   9 =    1*8 + 1
   1 =    0*8 + 1
```

```
Basis: 9
----------------
4711 =  523*9 + 4
 523 =   58*9 + 1
  58 =    6*9 + 4
   6 =    0*9 + 6
```

Abbildung 6.7 Ausgabe für die Basen 2 bis 9

Grundsätzlich ist es kein Problem, eine Basis größer als 10 zu verwenden. Es werden dann aber unter Umständen Ziffernwerte größer als 10 auftauchen. Wir testen dies mit der Basis 13:

```
Basis: 13
-----------------
4711 = 362*13 +  5
 362 =  27*13 + 11
  27 =   2*13 +  1
   2 =   0*13 +  2
-----------------
```

Abbildung 6.8 Ausgabe für die Basis 13

Es ergibt sich die Ziffernfolge 2, 1, 11, 5. Diese Ziffernfolge können Sie jedoch nicht nahtlos aneinanderreihen (21115), da dann die eindeutige Zuordnung der Ziffern zu den Stellenwerten verloren gehen würde. Wir behelfen uns dadurch, dass wir die zusätzlichen Ziffernsymbole a, b und c (oder A, B und C) für die Ziffernwerte 10, 11 und 12 einführen. Damit lautet die Darstellung der Zahl 4711 im 13er-System 21b5 oder 21B5.

Eine zentrale Frage steht aber noch im Raum. Warum machen wir das eigentlich? Sind wir nicht mit dem Dezimalsystem glücklich und zufrieden? Die Antwort auf diese Frage erhalten Sie im nächsten Abschnitt.

6.1.1 Dualdarstellung

Sie wissen, dass ein Digitalrechner intern mit zwei Zuständen – nennen wir sie 0 und 1 – arbeitet. Alle Daten und auch Programme im Rechner bestehen in diesem Sinne aus Folgen von 0 und 1. Der Rechner braucht eine Organisation, über die er effizient auf die Daten und Programme zugreifen kann. Dazu wird der Speicher des Rechners in kleine Speicherzellen unterteilt, und die Speicherzellen werden fortlaufend nummeriert. Da alles nur mit 0 und 1 dargestellt wird, können Sie sich das wie folgt vorstellen:

Speicherzelle					Wert							
0	0	0	0	0	0	1	1	1	0	0	1	0
1	0	0	0	1	1	0	0	1	0	1	0	0
2	0	0	1	0								
3	0	0	1	1								
4	0	1	0	0								
5	0	1	0	1								
6	0	1	1	0								
7	0	1	1	1								
8	1	0	0	0			weitere					
9	1	0	0	1			Werte					
10	1	0	1	0								
11	1	0	1	1								
12	1	1	0	0								
13	1	1	0	1								
14	1	1	1	0								
15	1	1	1	1								

Abbildung 6.9 Organisation der Speicherzellen

Wenn Sie dem Rechner die Anweisung geben, Ihnen den Wert aus der Speicherzelle 13 zu geben, wird der Rechner intern die Zellennummer im 2er-System erwarten und Ihnen auch den Wert der Speicherzelle im 2er-System zurückgeben. Wenn Sie sich also mit den Interna des Rechners beschäftigen wollen, kommen Sie um das 2er-System – auch *Dualsystem* genannt – nicht herum. Für uns Menschen hat dieses System aber erhebliche Nachteile. Wegen der kleinen Basis sind die Zahlen viel zu lang und nur umständlich zu handhaben. Außerdem fehlt uns jegliche Größenvorstellung für

Dualzahlen. Wenn etwa in der Zeitung ein Auto zu einem Kaufpreis von 23456 Euro annonciert wäre, würde bei Verwendung des Dualsystems dort ein Kaufpreis von 101101110100000 Euro stehen. Bei einer 0 mehr am Ende der Ziffernfolge wäre es der doppelte Kaufpreis. Das könnte man nur schwer erkennen.

Als Ergänzung zum Dualsystem benötigen wir dringend Zahlensysteme, die einfache Umrechnungen ins Dualsystem erlauben, dabei aber »menschenfreundlicher« sind als das Dualsystem.

6.1.2 Oktaldarstellung

Wir betrachten noch einmal unsere Lieblingszahl 4711, für die wir bereits die Dualdarstellung 1001001100111 kennen. Ich setze zwei führende Nullen hinzu und gruppiere die Ziffern in Dreierpäckchen: 001 001 001 100 111

$$\begin{aligned}001\,001\,001\,100\,111_2 = {} & 0 \cdot 2^{14} + 0 \cdot 2^{13} + 1 \cdot 2^{12} + \\ & 0 \cdot 2^{11} + 0 \cdot 2^{10} + 1 \cdot 2^{9} + \\ & 0 \cdot 2^{8} + 0 \cdot 2^{7} + 1 \cdot 2^{6} + \\ & 1 \cdot 2^{5} + 0 \cdot 2^{4} + 0 \cdot 2^{3} + \\ & 1 \cdot 2^{2} + 1 \cdot 2^{1} + 1 \cdot 2^{0}\end{aligned}$$

Jetzt klammere ich in jeder Zeile die höchste vorkommende Zweierpotenz aus:

$$\begin{aligned}001\,001\,001\,100\,111_2 = {} & (0 \cdot 2^{2} + 0 \cdot 2^{1} + 1) \cdot 2^{12} + \\ & (0 \cdot 2^{2} + 0 \cdot 2^{1} + 1) \cdot 2^{9} + \\ & (0 \cdot 2^{2} + 0 \cdot 2^{1} + 1) \cdot 2^{6} + \\ & (1 \cdot 2^{2} + 0 \cdot 2^{1} + 0) \cdot 2^{3} + \\ & (1 \cdot 2^{2} + 1 \cdot 2^{1} + 1) \cdot 2^{0}\end{aligned}$$

In den Klammern stehen jetzt Ziffernwerte zwischen 0 und 7. Die rechnen wir aus:

$$\begin{aligned}001\,001\,001\,100\,111_2 = {} & 1 \cdot 2^{12} + \\ & 1 \cdot 2^{9} + \\ & 1 \cdot 2^{6} + \\ & 4 \cdot 2^{3} + \\ & 7 \cdot 2^{0}\end{aligned}$$

Da aber $2^3 = 8$ ist, können wir auch schreiben:

$$\begin{aligned}001\,001\,001\,100\,111_2 = {} & 1 \cdot 8^{4} + \\ & 1 \cdot 8^{3} + \\ & 1 \cdot 8^{2} + \\ & 4 \cdot 8^{1} + \\ & 7 \cdot 8^{0} \qquad = 11147_8\end{aligned}$$

Damit haben wir eine einfache Umrechnung zwischen dem 8er-System (*Oktalsystem*) und dem 2er-System gefunden. Sie müssen einfach nur vom Ende der Zahl her Dreiergruppen bilden und diese Dreiergruppen mit folgender Tabelle ziffernweise in das Oktalsystem übersetzen:

Oktal	0	1	2	3	4	5	6	7
Dual	000	001	010	011	100	101	110	111

Abbildung 6.10 Darstellung im Oktalsystem

6.1.3 Hexadezimaldarstellung

Die *Hexadezimaldarstellung* verwendet die Basis 16 und die zusätzlichen Ziffernsymbole a, b, c, d, e und f (oder A, B, C, D, E und F), sodass wir die folgende Übersetzungstabelle für die Ziffern des Hexadezimalsystems nutzen können.

Hexadezimal	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Dual	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Abbildung 6.11 Darstellung im Hexadezimalsystem

Da $16 = 2^4$ genauso eine Potenz von 2 ist wie $8 = 2^3$, gilt das zur Umrechnung zwischen der Dual- und Oktaldarstellung Gesagte auch für die Umrechnung zwischen Dual- und Hexadezimaldarstellung. Der einzige Unterschied ist, dass anstelle der Dreierpäckchen jetzt Viererpäckchen gebildet werden müssen. Abbildung 6.12 zeigt Ihnen die Umrechnung an einem Beispiel:

Oktal	5			6			4			3		
Dual	1	0	1	1	1	0	1	0	0	0	1	1
Hexadezimal	b			a			3					

Abbildung 6.12 Oktal-, Dual- und Hexadezimalsystem in der Gegenüberstellung

Mit dem Hexadezimalsystem und dem Oktalsystem haben wir Zahlendarstellungen gefunden, die einerseits sehr nah an der internen Zahlendarstellung des Computers sind und andererseits der menschlichen Auffassung von Zahlen näher kommen als das Dualsystem. Das liegt daran, dass 8 und 16 die beiden der 10 am nächsten liegenden Zweierpotenzen sind.

6.2 Bits und Bytes

Die kleinste Informationseinheit auf einem Digitalrechner bezeichnen wir als *Bit*[2]. Ein Bit kann die logischen Werte 0 (Bit gelöscht) und 1 (Bit gesetzt) annehmen. Alle Informationen auf einem Rechner, seien es nun Programme oder Daten, sind als Folgen von Bits gespeichert. Den Speicherinhalt eines Rechners können wir zu einem Zeitpunkt als eine (sehr) lange Folge von Bits betrachten. Um Teile dieser Informationen gezielt ansprechen und manipulieren zu können, muss der Bitfolge eine Struktur gegeben werden. Zunächst fassen wir jeweils acht Bits zusammen und nennen diese Informationsgröße ein *Byte*.

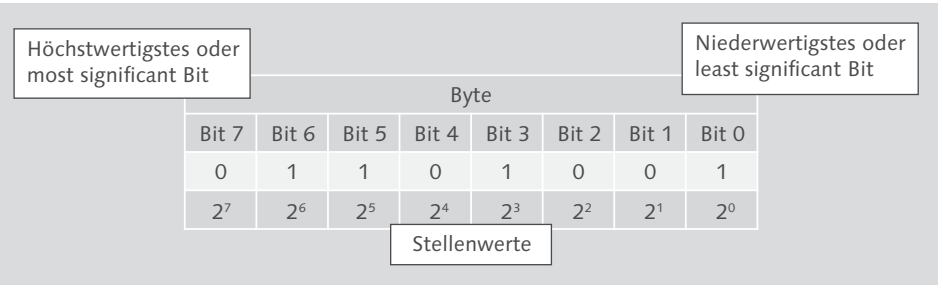


Abbildung 6.13 Darstellung eines Bytes

Als Dualzahl interpretiert, kann ein Byte also Zahlen von 0–255 (hex. 00–ff) darstellen. Bit 7 bezeichnen wir dabei als das höchstwertige (most significant), Bit 0 als das niederwertigste (least significant) Bit. Im Sinne der Interpretation als Dualzahl hat das höchstwertige Bit den Stellenwert $2^7 = 128$, das niederwertigste den Stellenwert $2^0 = 1$.

Jedes Byte im Speicher bekommt eine fortlaufende Nummer, seine Adresse. Über diese Adresse kann es vom Prozessor angesprochen (adressiert) werden (siehe Abbildung 6.14).

Der Prozessor wählt über den Adressbus eine Speicherzelle an und kann dann über den Datenbus den Inhalt der Speicherzelle laden, um die Daten zu verarbeiten. Auf dem gleichen Weg kann er dann Daten in den Speicher zurückschreiben.

Wir können die Informationen auf dem Adress- und Datenbus als Dualzahlen auffassen. In unserem Beispiel ist die Speicherstelle mit der Adresse $1011_2 = b_{16}$ angewählt, und in dieser Speicherstelle steht der Wert $11011001_2 = d9_{16} = 217$.

2 Binary Digit, gleichzeitig engl. bit = ein bisschen, kein Bier aus der Eifel

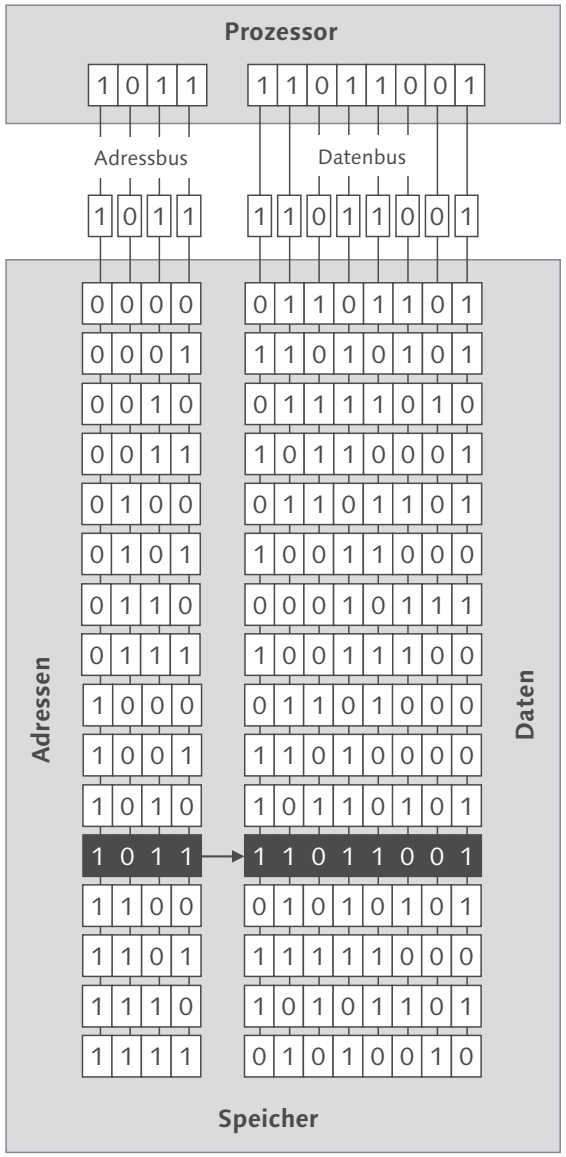


Abbildung 6.14 Adress- und Datenbus

Aus der »Breite« von Adress- und Datenbus ergeben sich grundlegende Leistungsdaten für einen Computer. Der hier schematisch gezeichnete Rechner kann über den vierstelligen Adressbus insgesamt 16 Speicherzellen anwählen, in denen er jeweils Werte im Bereich von 0–255 findet. Das ist natürlich nichts im Vergleich zu den Kapazitäten heutiger Rechner, die in der Regel über ein 64-Bit-Bussystem verfügen. Da sich die Kapazität mit jedem zusätzlichen Bit verdoppelt, ergeben sich Werte in ganz anderen Größenordnungen:

Breite	Adressierbare Zellen	
4 Bit	2^4	16
8 Bit	2^8	256
16 Bit	2^{16}	65536
32 Bit	2^{32}	4294967296
64 Bit	2^{64}	$1{,}84467 \cdot 10^{19}$

Tabelle 6.1 Busbreite und adressierbare Zellen

Wenn man in diese Größenordnungen vorstößt, braucht man Begriffe für große Datenmengen. Man orientiert sich dabei an den Maßeinheitenpräfixen (der Physik, Kilo, Mega, ...). Anders als in der Physik, in der diese Präfixe immer eine Vervielfachung um den Faktor $10^3 = 1000$ bedeuten, verwendet man in der Informatik den Faktor $2^{10} = 1024$.

Physik (SI-Präfixe)		Informatik (Binärpräfixe)		rel. Abweichung
Kilo	10^3	2^{10}	Kibi	2,40 %
Mega	10^6	2^{20}	Mebi	4,86 %
Giga	10^9	2^{30}	Gibi	7,37 %
Tera	10^{12}	2^{40}	Tebi	9,95 %
Peta	10^{15}	2^{50}	Pebi	12,59 %
Exa	10^{18}	2^{60}	Exbi	15,29 %

Tabelle 6.2 Vergleich von SI- und Binärpräfixen

Wegen der zunehmenden Abweichungen für große Werte sollte man in der Informatik eigentlich immer die Binärpräfixe verwenden. In der Praxis hat sich das jedoch bisher nicht durchgesetzt. In der Regel verwendet man die SI-Präfixe und meint damit die Werte der Binärpräfixe.

6.3 Skalare Datentypen in C

Sie wissen bereits, dass jede Variable in C einen Typ haben muss. Bisher haben Sie die Typen `int` und `float` kennengelernt. In diesem Abschnitt kommen weitere soge-

nannte *skalare Datentypen* hinzu. Unter einem skalaren Datentyp verstehen wir einen Datentyp zur Darstellung eindimensionaler numerischer Werte.

6.3.1 Ganze Zahlen

Es gibt eine Reihe von Datentypen für ganze Zahlen. Betrachten Sie dazu das folgende Diagramm[3]:

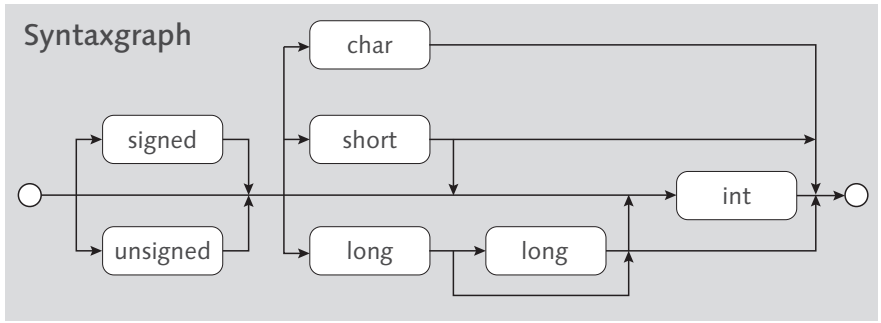


Abbildung 6.15 Syntaxgraph für Datentypen

Egal, wie Sie das Diagramm durchlaufen, Sie erhalten immer eine gültige Typvereinbarung, auf die dann ein Variablenname folgen muss. Einige Beispiele:

```
int a;
signed char b;
unsigned short int c;
long d;
unsigned long long int e;
```

Zu Beginn der Typvereinbarung können Sie festlegen, ob es sich um einen vorzeichenbehafteten (`signed`) oder vorzeichenlosen (`unsigned`) Typ handelt. Wenn Sie an dieser Stelle nichts angeben, wird ein vorzeichenbehafteter Typ angelegt. Aus diesem Grund ist die explizite Angabe von `signed` überflüssig und kommt in C-Programmen nur sehr selten vor. Danach folgt der eigentliche Datentyp, wobei die verschiedenen Varianten (`char`, `short`, `int`, `long`, `long long`) sich dadurch unterscheiden, wie viele Bytes der Datentyp im Speicher belegt. Das gegebenenfalls noch hinter `short` oder `long` zusätzlich vorkommende `int` ist ohne Bedeutung und wird daher auch nur sehr selten verwendet. Viele der theoretisch möglichen Kombinationen werden Sie nie in einem C-Programm finden. Die wichtigsten Typen sind sicherlich `char` und `int` zusammen mit ihren `unsigned`-Varianten. Das liegt daran, dass `char` der Typ ist, der den wenigsten Platz belegt, und `int` der Typ ist, mit dem der Rechner am schnellsten

3 So etwas nennt man einen *Syntaxgraphen*.

rechnen kann. Speicherverbrauch und Rechengeschwindigkeit sind eben die wichtigsten Kriterien bei der Programmoptimierung.

C legt sich bezüglich einer Anzahl der Bytes bei den Datentypen nur auf eine Mindestgröße fest:

Datentyp	Mindestgröße	typische Größe
`char`	1 Byte	1 Byte
`short`	2 Bytes	2 Bytes
`int`	2 Bytes	4 Bytes
`long`	4 Bytes	4 Bytes
`long long`	8 Bytes	8 Bytes

Tabelle 6.3 Datentypen und deren Größen

Zusätzlich ist festgelegt, dass die Datentypen in der oben genannten Reihenfolge ineinander enthalten sind. Auf unterschiedlichen Zielsystemen können Größen der Datentypen durchaus unterschiedlich sein. Sie können die Werte auf Ihrem Rechner mit einem kleinen Programm ermitteln:

```
printf( "char:      %d\n", sizeof( char));
printf( "short:     %d\n", sizeof( short));
printf( "int:       %d\n", sizeof( int));
printf( "long:      %d\n", sizeof( long));
printf( "long long: %d\n", sizeof( long long));
```

Listing 6.2 Größe unterschiedlicher ganzzahliger Datentypen

```
char:      1
short:     2
int:       4
long:      4
long long: 8
```

Wir verwenden hier den C-Operator `sizeof`, der uns die Größe eines Datentyps in Bytes liefert. Abhängig von der Anzahl der Bytes ergibt sich dann ein unterschiedlich großer Rechenbereich[4]:

4 Über die interne Darstellung vorzeichenbehafteter Zahlen haben wir nicht gesprochen, aber anschaulich sollte klar sein, dass bei gleicher Bytezahl die größte vorzeichenbehaftete Zahl etwa halb so groß ist wie die größte vorzeichenlose Zahl.

	signed		unsigned	
Größe	**min**	**max**	**min**	**max**
1 Byte	–128	127	0	255
2 Bytes	–32768	32767	0	65535
4 Bytes	–2147483648	2147483647	0	4294967295
8 Bytes	–9,2234E+18	9,2234E+18	0	1,84467E+19
Allgemein				
n Bit	-2^{n-1}	$2^{n-1}-1$	0	2^n-1

Tabelle 6.4 Rechenbereiche der Datentypen

Das Rechnen mit ganzen Zahlen ist exakt, solange Sie den vorgegebenen Rechenbereich nicht verlassen.

C unterstützt das Dezimal-, das Oktal- und das Hexadezimalsystem. Das Dualsystem ist wegen seiner Nähe zu Oktal- bzw. Hexadezimalsystem dadurch mit abgedeckt. Wenn wir mit Zahlen in verschiedenen Zahlensystemen in einem Programm arbeiten möchten, stellen sich drei Fragen:

- Wie schreiben wir eine Zahlkonstante in einem bestimmten Format im Quellcode?
- Wie lesen wir eine Zahl in einem bestimmten Format von der Tastatur ein?
- Wie schreiben wir eine Zahl in einem bestimmten Format auf dem Bildschirm?

Im Quellcode setzen wir einer Zahl ein Präfix voran, an dem man erkennt, ob es sich um eine Zahl im Oktalsystem (Präfix 0) oder Hexadezimalsystem (Präfix 0x) handelt. Bei der Eingabe mit `scanf` bzw. der Ausgabe mit `printf` verwenden wir spezielle Formatanweisungen (`"%..."`):

Zahlenformat	Eingabe	Ausgabe	Präfix
Dezimalsystem	`"%d"`	`"%d"`	(kein Präfix)
Oktalsystem	`"%o"`	`"%o"`	0
Hexadezimalsystem	`"%x"`	`"%x"`	0x

Tabelle 6.5 Zahlenformate bei Ein- und Ausgabe

Beachten Sie, dass 1234 und 01234 in einem C-Programm verschiedene Werte darstellen, da es sich im ersten Fall um eine Dezimal- und im zweiten Fall um eine Oktaldarstellung handelt.

Die folgenden Beispiele zeigen, wie Sie mit den verschiedenen Zahlendarstellungen in einem Programm arbeiten können. Dabei haben wir uns hier auf den in diesem Zusammenhang wichtigsten Datentyp (`unsigned int`) beschränkt. Im ersten Beispiel werden Zahlkonstanten in verschiedenen Systemen einer Variablen zugewiesen.

```
unsigned int zahl;

zahl = 123456;
printf( "Dezimalausgabe:     %d\n", zahl);
printf( "Oktalausgabe:       %o\n", zahl);
printf( "Hexadezimalausgabe: %x\n", zahl);

zahl = 0123456;
printf( "Dezimalausgabe:     %d\n", zahl);
printf( "Oktalausgabe:       %o\n", zahl);
printf( "Hexadezimalausgabe: %x\n", zahl);

zahl = 0x123abc;
printf( "Dezimalausgabe:     %d\n", zahl);
printf( "Oktalausgabe:       %o\n", zahl);
printf( "Hexadezimalausgabe: %x\n", zahl);
```

Listing 6.3 Ausgabe von Zahlen in unterschiedlicher Darstellung

Diese wird dann in verschiedenen Darstellungen ausgegeben:

```
Dezimalausgabe:     123456
Oktalausgabe:       361100
Hexadezimalausgabe: 1e240

Dezimalausgabe:     42798
Oktalausgabe:       123456
Hexadezimalausgabe: a72e

Dezimalausgabe:     1194684
Oktalausgabe:       4435274
Hexadezimalausgabe: 123abc
```

Im zweiten Beispiel wird der Zahlenwert in unterschiedlichen Formaten von der Tastatur eingelesen:

```
unsigned int zahl;

printf( "Dezimaleingabe:     ", &zahl);
```

```
scanf( "%d", &zahl);
printf( "Dezimalausgabe:     %d\n", zahl);
printf( "Oktalausgabe:       %o\n", zahl);
printf( "Hexadezimalausgabe: %x\n\n\n", zahl);

printf( "Oktaleingabe:       ", &zahl);
scanf( "%o", &zahl);
printf( "Dezimalausgabe:     %d\n", zahl);
printf( "Oktalausgabe:       %o\n", zahl);
printf( "Hexadezimalausgabe: %x\n\n\n", zahl);

printf( "Hexadezimaleingabe: ", &zahl);
scanf( "%x", &zahl);
printf( "Dezimalausgabe:     %d\n", zahl);
printf( "Oktalausgabe:       %o\n", zahl);
printf( "Hexadezimalausgabe: %x\n\n\n", zahl);
```

Listing 6.4 Einlesen von Werten in unterschiedlichen Formaten

```
Dezimaleingabe:     123456
Dezimalausgabe:     123456
Oktalausgabe:       361100
Hexadezimalausgabe: 1e240

Oktaleingabe:       123456
Dezimalausgabe:     42798
Oktalausgabe:       123456
Hexadezimalausgabe: a72e

Hexadezimaleingabe: 123abc
Dezimalausgabe:     1194684
Oktalausgabe:       4435274
Hexadezimalausgabe: 123abc
```

6.3.2 Gleitkommazahlen

Für Gleitkommazahlen gibt es nicht so viele Typvarianten wie für ganze Zahlen:

Datentyp	typische Größe	
float	1 Byte	einfache Genauigkeit

Tabelle 6.6 Typvarianten der Gleitkommazahlen

Datentyp	typische Größe	
double	2 Bytes	doppelte Genauigkeit
long double	4 Bytes	besonders hohe Genauigkeit

Tabelle 6.6 Typvarianten der Gleitkommazahlen (Forts.)

Wertebereich und Genauigkeit der verschiedenen Gleitkommatypen sind im Standard nicht festgelegt. Der Speicherplatzbedarf auf einem konkreten Zielsystem lässt sich wieder über ein kleines Programm ermitteln:

```
printf( "float:     %d\n", sizeof( float));
printf( "double:    %d\n", sizeof( double));
printf( "long double %d\n", sizeof( long double));
```

Listing 6.5 Größe unterschiedlicher Gleitkommadatentypen

Hier erhalten wir die folgende Ausgabe:

```
float       4
double      8
long double 8
```

Die Ergebnisse dieses Programms sind aber, wie schon bei den Ganzzahltypen, maschinenabhängig.

Zur Eingabe konkreter Gleitkommazahlenwerte verwenden Sie die vom Taschenrechner her bekannte technisch-wissenschaftliche Notation mit Vorzeichen, Mantisse und Exponent:

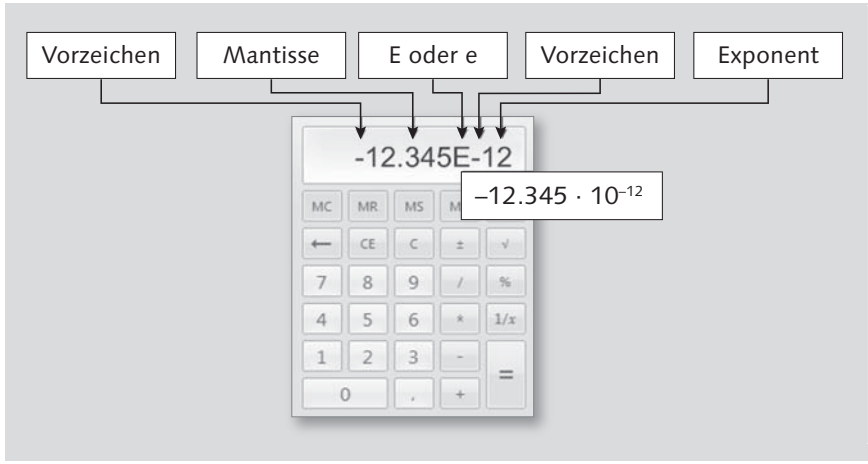


Abbildung 6.16 Bestandteile von Gleitkommazahlen

Beispiele:

```
float a = -1;
float b = 1E2;
double c = 1.234;
long double d = -123.456E-345;
```

Gleitkommazahlen gibt es nur in Dezimalschreibweise, aber es gibt wieder verschiedene Formatanweisungen für die Ein- bzw. Ausgabe.

Typ	Eingabe	Ausgabe
`float`	`"%f"`	`"%f"`
`double`	`"%lf"`	`"%lf"`
`long double`	`"%Lf"`	`"%LF"`

Tabelle 6.7 Formatanweisungen für Gleitkommazahlen

Achtung, während das Rechnen mit ganzen Zahlen exakt ist, solange man im zulässigen Rechenbereich bleibt, ist das Rechnen mit Gleitkommazahlen fehleranfällig. Es gibt einen Mindestabstand zwischen zwei Zahlen, unterhalb dessen der Rechner nicht genauer auflösen kann. Durch häufige Rechenoperationen können sich die Rechenfehler dann aufschaukeln. Dies bei numerischen Berechnungen zu vermeiden ist es eine anspruchsvolle Aufgabe, mit der wir uns hier aber nicht beschäftigen werden.

6.4 Bitoperationen

Bisher haben wir Zahlen immer als Material für arithmetische Operationen gesehen, aber man kann Zahlen auch viel elementarer als ein Bitmuster betrachten – also einfach als eine Folge von 0 und 1. Es ist hilfreich, wenn Sie im Folgenden gar nicht daran denken, dass Zahlen einen Wert haben. Wir wollen uns überlegen, wie wir im Bitmuster einer ganzen Zahl gezielt Manipulationen durchführen können. Solche Manipulationen sind z. B.:

- Setze gezielt ein bestimmtes Bit.
- Lösche gezielt ein bestimmtes Bit.
- Invertiere gezielt ein bestimmtes Bit.

Natürlich haben solche Operationen auch eine arithmetische Bedeutung, aber uns interessiert hier in erster Linie das Bitmuster. In C gibt es sechs Operationen auf Bit-

mustern, die Sie im Folgenden kennenlernen werden. Diese Operationen werden vorrangig auf vorzeichenlosen Zahlen (`unsigned char`, `unsigned short`, `unsigned int`, `unsigned long` und `unsigned long long`) durchgeführt, und in diesem Kontext werden wir diese Operationen auch ausschließlich betrachten.

Das *bitweise Komplement* (~) invertiert das Bitmuster einer Zahl. Aus einer 0 wird eine 1 und aus einer 1 eine 0:

Bitweises Komplement								
x	1	0	0	1	1	0	1	1
~x	0	1	1	0	0	1	0	0

Abbildung 6.17 Bitweises Komplement

Das *bitweise Und* (&) benötigt zwei Operanden und verknüpft deren Muster Bit für Bit mit einer logischen Und-Operation:

Bitweises Und								
x	1	1	0	0	0	0	1	0
y	1	0	0	1	1	0	1	1
x&y	1	0	0	0	0	0	1	0

Abbildung 6.18 Bitweises Und

Ganz analog arbeitet das *bitweise Oder* (|) mit einer logischen Oder-Operation:

Bitweises Oder								
x	1	1	0	0	0	0	1	0
y	1	0	0	1	1	0	1	1
x|y	1	1	0	1	1	0	1	1

Abbildung 6.19 Bitweises Oder

Zusätzlich gibt es das *bitweise Entweder-Oder*, das eine ausschließende Oder-Operation durchführt:

Bitweises Entweder-Oder								
x	1	1	0	0	0	0	1	0
y	1	0	0	1	1	0	1	1
x^y	0	1	0	1	1	0	0	1

Abbildung 6.20 Bitweises Entweder-Oder

Schließlich gibt es noch zwei Schiebeoperationen. Bei einem *Bitshift nach rechts* wird das Bitmuster um eine gewisse Anzahl von Stellen nach rechts geschoben, und die frei werdenden Stellen werden mit Nullen aufgefüllt:

Bitshift rechts								
x	1	0	0	1	1	0	1	1
x>>2	0	0	1	0	0	1	1	0

Abbildung 6.21 Bitshift rechts

Der *Bitshift nach links* schiebt in die andere Richtung. Auch hier werden Nullen nachgeschoben:

Bitshift links								
x	1	0	0	1	1	0	1	1
x<<2	0	1	1	0	1	1	0	0

Abbildung 6.22 Bitshift links

Ein Schieben um eine Stelle nach links entspricht übrigens einer Multiplikation mit 2, ein Schieben um eine Stelle nach rechts entspricht einer Division durch 2 ohne Rest. Dementsprechend ist $1 << n = 2^n$.

Mit den jetzt bereitgestellten Grundoperationen können Sie die oben beschriebenen Bitmanipulationen durchführen. Starten Sie mit der Aufgabe, in einer Zahl x ein bestimmtes Bit – etwa das dritte von rechts[5] – zu setzen. Dazu gehen Sie wie folgt vor:

Eine 1 wird um drei Positionen nach links geschoben und über ein bitweises Oder mit x verknüpft. Dadurch wird das dritte Bit in x gesetzt.

Setzen des dritten Bits in x								
1	0	0	0	0	0	0	0	1
1<<3	0	0	0	0	1	0	0	0
x	1	0	1	1	0	0	1	1
x|(1<<3)	1	0	1	1	1	0	1	1

Abbildung 6.23 Setzen des dritten Bits in x

5 Wenn ich vom »dritten Bit von rechts« spreche, meine ich das Bit mit dem Stellenwert 2^3. Ich fange also, wie so oft in der Programmierung, bei 0 an zu zählen.

Zum Löschen eines Bits verwenden Sie das bitweise Und:

Eine 1 wird um vier Positionen nach links geschoben, komplementiert und dann über ein bitweises Und mit x verknüpft. Dadurch wird das vierte Bit in x gelöscht.

Löschen des vierten Bits in x								
1	0	0	0	0	0	0	0	1
1<<4	0	0	0	1	0	0	0	0
~(1<<4)	1	1	1	0	1	1	1	1
x	1	0	1	1	0	0	1	1
x&~(1<<4)	1	0	1	0	0	0	1	1

Abbildung 6.24 Löschen des vierten Bits in x

Um ein Bit zu invertieren, verwenden Sie das bitweise Entweder-Oder:

Eine 1 wird um fünf Positionen nach links geschoben und über ein bitweises Entweder-Oder mit x verknüpft. Dadurch wird das fünfte Bit in x invertiert.

Invertieren des fünften Bits in x								
1	0	0	0	0	0	0	0	1
1<<5	0	0	1	0	0	0	0	0
x	1	0	1	1	0	0	1	1
x^(1<<5)	1	0	0	1	0	0	1	1

Abbildung 6.25 Invertieren des fünften Bits in x

Wir fassen das noch einmal zusammen:

```
int n = 3;
unsigned int x = 0xaffe;

x = x | (1<<n);  // Setzen des n-ten Bits in x
x = x & ~(1<<n); // Loeschen des n-ten Bits in x
x = x ^ (1<<n);  // Invertieren des n-ten Bits in x
```

Das Bitmuster `1<<n`, in dem ja genau ein Bit gesetzt ist, bezeichnet man auch als *Maske*, weil über dieses Muster genau ein Bit aus der Zahl `x` herausgefiltert (maskiert) wird.

Häufig will man ein Bitmuster nicht verändern, sondern einfach nur testen, ob ein bestimmtes Bit in dem Bitmuster gesetzt ist. Das geht so:

```
int n = 3;
unsigned int x = 0xaffe;

if( x & (1<<n))  // Test, ob das n-te Bit in x gesetzt ist
    {
    ...
    }
```

Sie werden sich vielleicht fragen, was solche »Bitfummeleien« sollen. Darauf will ich Ihnen zwei Antworten geben. Erstens, wenn Sie einmal maschinennah, etwa auf einem Microcontroller, programmieren, werden Ihre Programme zu einem großen Teil aus solchen Bitoperationen bestehen, und zweitens können Sie mit diesen Techniken den zu Beginn des Kapitels gezeigten Kartentrick programmieren. Das machen wir als erstes Beispiel im nächsten Abschnitt.

6.5 Programmierbeispiele

Unsere Kenntnisse über die Darstellung von Zahlen in verschiedenen Zahlensystemen setzen wir jetzt in einigen Beispielen praktisch ein. Dabei kommen wir auch noch einmal auf den Kartentrick vom Anfang des Kapitels zurück, den Sie mittlerweile schon durchschaut haben dürften.

6.5.1 Kartentrick

Ich weiß nicht, ob es Ihnen gelungen ist, hinter den am Anfang des Kapitels beschriebenen Kartentrick zu kommen. Wenn nicht, dann versuchen Sie es, bevor Sie weiterlesen, vielleicht noch einmal mit den inzwischen erworbenen Kenntnissen über Dualzahlen und Bitmuster.

Wir schauen uns an, welche Dualdarstellung die Zahlen auf den Karten haben, und kommen zu folgendem Ergebnis (siehe Abbildung 6.26).

Auf der Karte A stehen alle Zahlen, die das nullte Bit gesetzt haben. Auf der Karte B stehen alle Zahlen, die das erste Bit gesetzt haben, und so geht das weiter. Das heißt, jedes Mal, wenn der Besitzer der Geheimzahl sagt, dass die Zahl auf einer Karte steht oder nicht, gibt er ein Bit seiner Zahl preis. Diese Bits müssen Sie nur noch zu einer Zahl kombinieren. Dazu betrachten Sie die erste Zahl auf jeder Karte. In dieser Zahl ist immer nur genau das für diese Karte charakteristische Bit gesetzt.

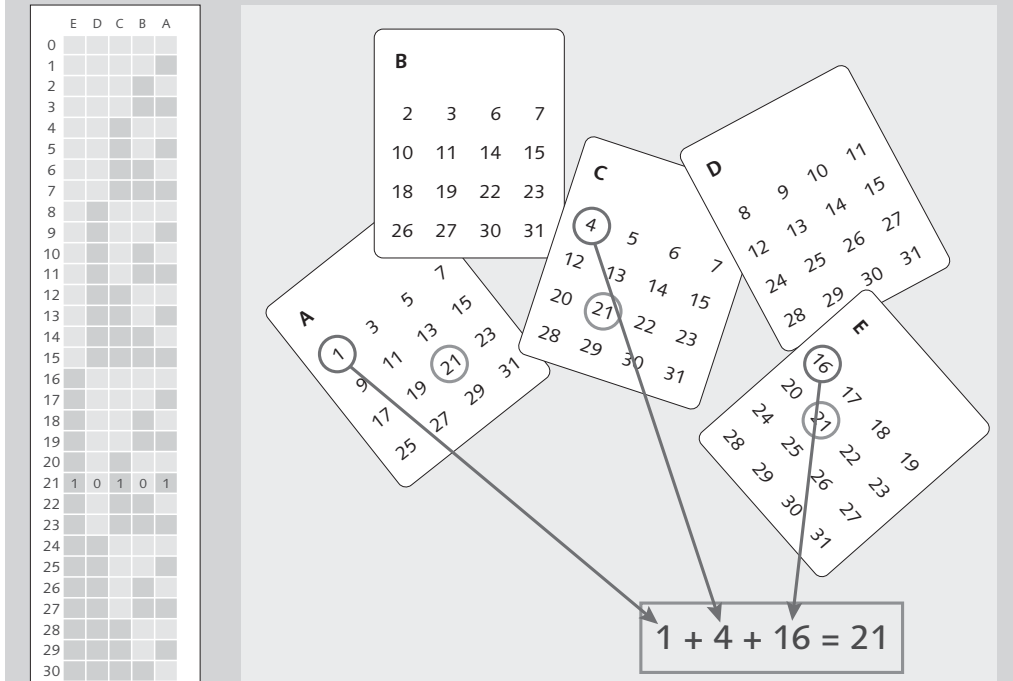


Abbildung 6.26 Die Auflösung des Kartentricks

Sie müssen also nur noch ein logisches Oder zwischen den ersten Zahlen aller ausgewählten Karten bilden, um die Geheimzahl zu erhalten. Da die Bitmuster dieser Zahlen sich aber nicht überschneiden, entspricht diese Oder-Verknüpfung einer Addition. Wenn Sie also die jeweils ersten Zahlen der Karten, auf denen die Geheimzahl stehen, addieren, erhalten Sie die Geheimzahl.

Das können wir in einem Programm umsetzen:

```
  void main()
      {
      int bit, z, zahl, antwort;

      printf( "Denk dir eine Zahl zwischen 0 und 31\n");

A     for( bit = 1, zahl = 0; bit < 32; bit = bit << 1)
          {
          printf( "Ist die Zahl in dieser Liste");
B         for( z = 0; z < 32; z++)
              {
```

```
C               if( z & bit)
                    printf( " %d", z);
                }
            printf( ":");
            scanf( "%d", &antwort);
            if( antwort == 1)
D               zahl = zahl |  bit;
            }

        printf( "Die Zahl ist %d\n", zahl);
        }
```

Listing 6.6 Das Kartenraten als Programm

(A) Die Variable `bit` durchläuft in der Schleife die Werte:

$1 = 00001_2$
$2 = 00010_2$
$4 = 00100_2$
$8 = 01000_2$
$16 = 10000_2$

In der folgenden Schleife (B) werden alle 32 Zahlen durchlaufen, aber ausgegeben werden nur die, die das `bit` gesetzt haben (C). Wenn die gesuchte Zahl auf der ausgegebenen Karte steht, wird das `bit` gesetzt (D).

Am Beispiel der Zahl 21 ergibt sich dann folgendes Ablaufprotokoll:

```
Denk dir eine Zahl zwischen 0 und 31
Ist die Zahl in dieser Liste 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31: 1
Ist die Zahl in dieser Liste 2 3 6 7 10 11 14 15 18 19 22 23 26 27 30 31: 0
Ist die Zahl in dieser Liste 4 5 6 7 12 13 14 15 20 21 22 23 28 29 30 31: 1
Ist die Zahl in dieser Liste 8 9 10 11 12 13 14 15 24 25 26 27 28 29 30 31: 0
Ist die Zahl in dieser Liste 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31:
 1

Die Zahl ist 21
```

6.5.2 Zahlenraten

Wenn Sie das letzte Beispiel abwandeln, kommen Sie zu einer anderen Form des Zahlenratens. Sie können das höchste Bit setzen und fragen, ob die Geheimzahl größer oder kleiner ist. Wenn sie größer ist, bleibt das Bit gesetzt, ansonsten löschen wir es wieder. Dann setzen wir das zweithöchste Bit und fragen wieder. Auf diese Weise

können wir mit jeder Frage die Anzahl der noch möglichen Zahlen halbieren, bis am Ende nur noch eine Zahl übrig bleibt. Hier ist das Programm dazu:

```
    void main()
        {
        int n, antwort;
        unsigned int zahl, bit;


A       printf( "Anzahl Stellen: ");
        scanf( "%d", &n);
        printf( "Denk dir eine Zahl zwischen 0 und %d\n", (1<<n)-1);

B       for( zahl = 0, bit = (1<<n-1); bit > 0; bit = (bit>>1))
            {
C           zahl = zahl | bit;
            printf( "Ist die Zahl kleiner als %d: ", zahl);
            scanf( "%d", &antwort);
            if( antwort == 1)
D               zahl = zahl & ~bit;
            }

        printf( "Die Zahl ist %d\n", zahl);
        }
```

Listing 6.7 Ein abgewandeltes Zahlenraten

In dem Programm werden `n`-stellige Dualzahlen betrachtet (A). Die größte `n`-stellige Dualzahl ist:

$$(1 << n) - 1 = 2^n - 1 = \underbrace{111 \ldots 1}_{n\text{-}mal}$$

In der Schleife (B) wird das Bit zunächst *gesetzt*. Wenn die Zahl dann zu groß ist, wird das Bit wieder gelöscht (D).

Innerhalb der Schleife durchläuft die Variable `bit` Potenzen von 2 (C):

$2^{n-1} = 1000 \ldots 000_2$
$2^{n-2} = 0100 \ldots 000_2$
$2^{n-3} = 0010 \ldots 000_2$
...
$2^1 = 0000 \ldots 010_2$
$2^0 = 0000 \ldots 001_2$

Als Geheimzahl wählen wir natürlich 4711 und lassen den Computer raten:

```
Anzahl Stellen: 16
Denk dir eine Zahl zwischen 0 und 65535
Ist die Zahl kleiner als 32768: 1
Ist die Zahl kleiner als 16384: 1
Ist die Zahl kleiner als 8192: 1
Ist die Zahl kleiner als 4096: 0
Ist die Zahl kleiner als 6144: 1
Ist die Zahl kleiner als 5120: 1
Ist die Zahl kleiner als 4608: 0
Ist die Zahl kleiner als 4864: 1
Ist die Zahl kleiner als 4736: 1
Ist die Zahl kleiner als 4672: 0
Ist die Zahl kleiner als 4704: 0
Ist die Zahl kleiner als 4720: 1
Ist die Zahl kleiner als 4712: 1
Ist die Zahl kleiner als 4708: 0
Ist die Zahl kleiner als 4710: 0
Ist die Zahl kleiner als 4711: 0
Die Zahl ist 4711
```

6.5.3 Addierwerk

Im nächsten Beispiel werden Sie ein Programm schreiben, das zwei Zahlen addieren soll. Nichts leichter als das, werden Sie sagen, aber wir wollen es so machen, wie der Rechner es intern macht. Das heißt, wir stellen uns vor, dass es noch gar keine Addition gibt und dass unser Programm nur elementare Operationen auf Bitmustern durchführen kann. Sie werden also ein Programm schreiben, das zwei Zahlen addiert, ohne dass irgendwo im Programm ein +-Zeichen auftaucht. Versuchen Sie es zunächst einmal allein, bevor Sie sich meine Lösung ansehen.

Eine Addition läuft im Dualsystem genauso ab, wie Sie es in der Schule im 10er-System gelernt haben. Man schreibt beide Zahlen untereinander und addiert ziffernweise von rechts nach links, wobei gegebenenfalls ein Übertrag entsteht. Den Übertrag verarbeitet man immer im nächsten Rechenschritt. Im Dualsystem müssen Sie nur darauf achten, dass ein Übertrag schon bei 1 (1+1 = 10)[6] und nicht erst bei 9 eintritt. Außerdem müssen Sie im Hinterkopf behalten, dass Sie einen endlichen Rechenbereich haben und irgendwann ein Überlauf erfolgt. Die Berechnung der Summe wird nach folgendem Schema durchgeführt:

6 Vielleicht kennen Sie den Spruch: »There are 10 types of people in the world: Those who understand binary and those who don't.« Ich hoffe, Sie gehören inzwischen zur ersten Art.

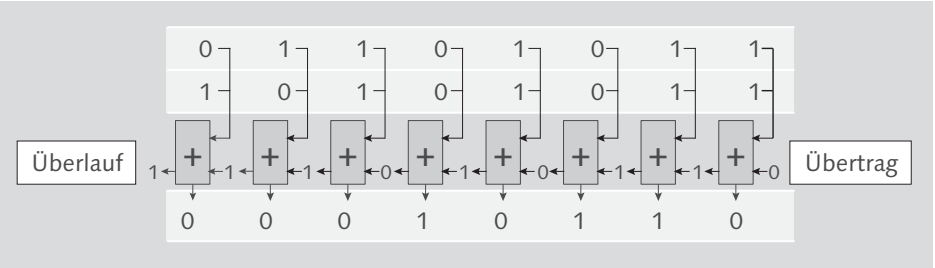


Abbildung 6.27 Schema des Addierwerkes

Bezeichnen Sie die eingehenden Bits mit `s1` und `s2` und den Übertrag mit `c`. Wenn Sie jetzt beachten, dass der Entweder-Oder-Operator einer Addition von Bits ohne Übertrag entspricht, ergibt sich die Summe (ohne Übertrag) der drei Werte durch diesen C-Ausdruck:

```
summe = s1 ^ s2 ^c;
```

Einen Übertrag erhalten Sie, wenn mindestens zwei der drei Werte 1 sind:

```
c = (s1 & s2) | (s1 & c) | (s2 & c)
```

Jetzt können Sie das Addierwerk programmieren:

```
   void main()
       {
A      unsigned int z1, z2;
B      unsigned int s, s1, s2, sum, c;

       printf( "Gib bitte zwei Zahlen ein: ");
       scanf( "%d %d", &z1, &z2);

C      for( sum = 0, s = 1, c = 0; s != 0; s = s << 1, c = c << 1)
           {
D          s1 = z1 & s;
           s2 = z2 & s;
E          sum = sum | (s1 ^ s2 ^ c);
           c = (s1 & s2) | (s1 & c) | (s2 & c);
           }

       printf( "Summe: %d\n", sum);
       }
```

Listing 6.8 Implementierung des Addierwerkes

`z1` und `z2` sind die zu addierenden Zahlen (A), `s` ist die Maske, die über die Zahlen geschoben wird, `s1` und `s2` sind die aus den Zahlen `z1` und `z2` maskierten Bits, `sum` ist die zu berechnende Summe, und `c` ist der Übertrag (Carry) (B).

Im Programm werden Maske und Carry über die Zahlen geschoben (C) und in der Schleife die Bits aus den Zahlen gefiltert (D), danach werden Summe und Übertrag für die betrachtete Stelle berechnet (E).

Zum Abschluss können Sie das Programm testen:

```
Gib bitte zwei Zahlen ein: 12345 67890
Summe: 80235
```

Das Programm kann addieren, obwohl nirgendwo im Programm, außer in der Zählschleife, ein +-Zeichen steht.

6.6 Zeichen

Ein Computer soll nicht nur Zahlen, sondern auch Buchstaben und Text verarbeiten können. Da der Computer intern aber nur Dualzahlen – besser gesagt: Bitmuster – kennt, muss es eine Zuordnung von Buchstaben zu Bitmustern geben. Eine solche Zuordnung nennt man einen *Code*.

Ein klassischer Code für die gebräuchlichsten Zeichen ist der ASCII-Code. Die meisten Rechner benutzen diesen Code, haben jedoch oft individuelle Erweiterungen (nationale Zeichensätze, grafische Symbole etc.).

ASCII-Zeichentabelle, hexadezimale Nummerierung

Code	…0	…1	…2	…3	…4	…5	…6	…7	…8	…9	…A	…B	…C	…D	…E	…F
0…	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1…	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2…	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3…	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4…	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5…	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6…	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7…	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Quelle: Wikipedia

Das Zeichen Z hat den ASCII-Code $5a_{16} = 90_{10}$.
Der numerische Wert ist dabei relativ unwichtig.
Wichtig ist das Bitmuster:

5a = 0101 1010

Abbildung 6.28 Der ASCII-Zeichensatz

Grundsätzlich unterscheiden wir im ASCII-Code druckbare und nicht druckbare Zeichen. Die druckbaren Zeichen (A, B, C, ...) sprechen für sich. Von den nicht druckbaren Zeichen interessiert uns hier nur das Linefeed-Zeichen (LF), das, wenn man es auf dem Bildschirm ausgibt, einen Zeilenvorschub erzeugt.

Zeichenkonstanten werden in einfache Hochkommata gesetzt (`'a'`, `'Z'`). Für das nicht druckbare Linefeed-Zeichen verwenden wir die Ersatzdarstellung `'\n'`. Als Typ für Zeichenvariablen wird `char` (oder `unsigned char`) verwendet. Bei der Ein- bzw. Ausgabe einzelner Zeichen wird `"%c"` als Formatanweisung verwendet:

```
   char a, b, c;

A  a = 'x';
   b = '\n';

   printf( "Bitte gib einen Buchstaben ein: ");
   scanf( "%c", &c);

C  printf( "Ausgabe: %c%c%c\n", a, b, c);
```

Listing 6.9 Verwendung von char

In dem Programm werden zuerst Zeichenkonstanten in Variablen gespeichert (A), bevor in einem weiteren Schritt ein Zeichen von der Tastatur eingelesen wird. Zeichen werden mit der Formatanweisung `%c` eingelesen (B) bzw. ausgegeben (C). Dabei erzeugt die Ausgabe von `b` einen Zeilenvorschub:

```
Bitte gib einen Buchstaben ein: Q
Ausgabe: x
Q
```

Beachten Sie, dass beim Lesen mit `%c` nur das eine Zeichen eingelesen wird und dass zusätzlich eigegebene Zeichen, wie das unvermeidliche Linefeed zum Abschluss der Eingabe, im Eingabepuffer verbleiben und dann gegebenenfalls bei der nächsten Leseoperation gelesen werden.

Dass Zeichen (`char`) zugleich als Zahlen interpretiert werden können, hat den positiven Seiteneffekt, dass mit Zeichen wie mit Zahlen gerechnet werden kann. Für das folgende Programm

```
char x;

for( x = 'A'; x <= 'K'; x = x + 1)
    printf( "%d %c\n", x, x);
```

Listing 6.10 Interpretation von char als Zahlen

erhalten wir diese Ausgabe:

```
65: A
66: B
67: C
68: D
69: E
70: F
71: G
72: H
73: I
74: J
75: K
```

Da die Nummerierung der Zeichen der alphabetischen Reihenfolge entspricht, kann dies z. B. genutzt werden, um Zeichen alphabetisch zu sortieren.

Zum Abschluss dieses Abschnitts wollen wir uns vergewissern, dass unser Rechner den ASCII-Zeichencode verwendet. Dazu erstellen wir folgendes Programm:

```
   void main()
      {
A     unsigned char zeile, spalte, z;
      printf( "    ");
B     for( spalte = 0; spalte < 0x10; spalte ++)
C         printf( " .%x", spalte);
      printf( "\n");
D     for( zeile = 0; zeile < 0x08; zeile++)
          {
E         printf( "  %x.", zeile);
F         for( spalte = 0; spalte < 0x10; spalte ++)
              {
G             z = (zeile << 4)|spalte;
H             if( (z > 0x20) && ( z < 0x7f))
I                 printf( "  %c", z);
              else
                  printf( "  .");
             }
          printf( "\n");
          }
      }
```

Listing 6.11 Ausgabe des ASCII-Zeichensatzes

Der hier verwendete Datentyp für Zeichen ist `unsigned char` (A). Das Programm erzeugt in einer Schleife die Überschrift (B) mit insgesamt `0x10 = 16` Spalten. Dabei werden die Spaltenüberschriften mit der Formatanweisung `%x` hexadezimal ausgegeben. Im Anschluss an die Überschrift folgt eine Doppelschleife zur Erzeugung der Tabelle (D) und (F), auch hier wieder mit einer hexadezimalen Ausgabe (E). Der Index des aktuellen Zeichens wird arithmetisch ermittelt, $z = zeile \cdot 2^4 + spalte$ (G).

Damit nur die druckbaren Zeichen in der Tabelle ausgegeben werden, erfolgt eine Unterscheidung in druckbare bzw. nicht druckbare Zeichen (H). Die druckbaren Zeichen liegen dabei in dem Bereich $20_{16} < z < 7f_{16}$ und werden mit `%c` ausgegeben (I).

Mit unserem Programm erhalten wir eine eigene Tabelle des ASCII-Zeichensatzes:

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

Den Zeichencode für die auszugebenden Zeichen setzen wir im Programm als Bitmuster aus dem Zeilen- und dem Spaltenindex zusammen.

```
z = (zeile << 4) | spalte;
```

Der Zeilenindex wird um 4 Bit nach links geschoben. In die 4 unteren Bytes wird dann der Spaltenindex montiert, damit ergibt sich in `z` der Zeichencode in der betrachteten Zeile und Spalte.

6.7 Arrays

Stellen Sie sich vor, dass Sie ein Programm erstellen sollen, das 100 Zahlen einliest und die Zahlen in umgekehrter Reihenfolge wieder ausgibt. Mit Ihren derzeitigen Programmierkenntnissen wären Sie tatsächlich gezwungen, 100 Variablen anzulegen, einzeln einzulesen und anschließend einzeln wieder auszugeben. Sie könnten für die erforderlichen Ein- und Ausgaben nicht einmal eine Schleife verwenden, da Sie keinen Datentyp kennen, der 100 Zahlen aufnehmen kann und dessen Inhalt flexibel über eine Schleife bearbeitet werden kann. Zum Glück handelt es sich bei dem angesprochenen Problem nicht um einen Mangel der Programmiersprache C, son-

dern um einen Mangel an Programmierkenntnissen in C, den wir umgehend beseitigen werden.

6.7.1 Eindimensionale Arrays

Ein *Array* ist eine Aneinanderreihung von Datenelementen gleichen Typs. Über einen Index kann auf jedes Datenelement unmittelbar zugegriffen werden.

Wenn wir ein Array benötigen, müssen wir uns drei Fragen stellen:

1. Wie soll das Array heißen?
2. Wie viele Datenelemente soll das Array haben?
3. Welchen Datentyp sollen die Elemente des Arrays haben?

Wenn wir ein Array für fünf ganze Zahlen (`int`) benötigen, das `meinezahlen` heißen soll, schreiben wir im Programm:

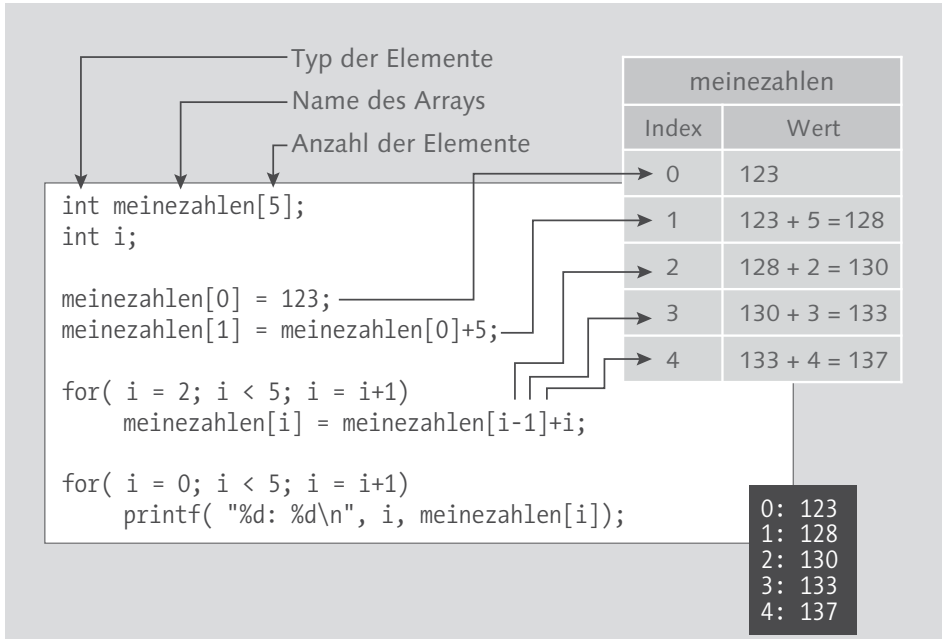


Abbildung 6.29 Verwendung eines eindimensionalen Arrays

Natürlich können Sie einem Array auch jeden anderen Datentyp (`char`, `unsigned short`, `float`, ...) zugrunde legen.

Auf die einzelnen Elemente des Arrays greifen wir mit einem sogenannten *Index* zu. Verwendet werden die indizierten Elemente eines Arrays wie eine Variable des entsprechenden Datentyps. Der Index selbst ist eine ganze Zahl und kann über eine Konstante, eine Variable oder einen Formelausdruck gegeben sein. Das erste Element des Arrays hat den Index 0, das zweite den Index 1 etc.:

Wenn das Array n Elemente hat, sind die Elemente von 0 bis n-1 nummeriert.

Achten Sie darauf, dass Sie nur gültige Indizes aus diesem Bereich verwenden. Der Compiler überprüft das nicht. In der Regel stürzt Ihr Programm ab, wenn Sie einen ungültigen Index verwenden.

Arrays können direkt bei ihrer Definition mit Werten gefüllt werden. Man gibt dazu die gewünschten Werte in geschweiften Klammern und durch Kommata getrennt an:

```
int meinezahlen[5] = { 1, 2, 3, 4, 5};
```

Dass dabei u. U. nicht alle Felder besetzt werden, ist unproblematisch. Der Compiler füllt das Array von vorn beginnend. Nicht angesprochene Felder bleiben uninitialisiert.

In Verbindung mit Zählschleifen ergeben sich vielfältige Verarbeitungsmöglichkeiten für Arrays. Insbesondere können Sie das in der Einleitung gestellte Problem (100 Zahlen einlesen und in umgekehrter Reihenfolge wieder ausgeben) elegant lösen:

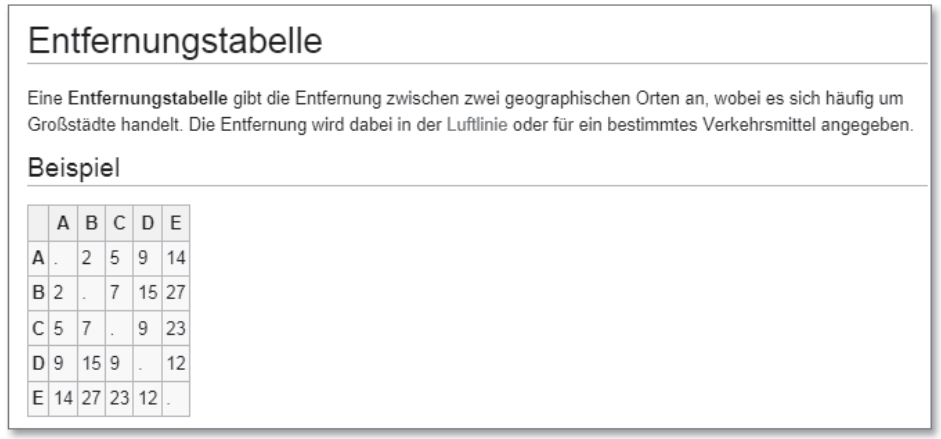
```
   void main()
       {
A      int daten[100];
       int i;

B      for( i = 0; i < 100; i = i+1)
           {
           printf( "Gib die %d-te Zahl ein: ", i);
           scanf( "%d", &daten[i]);
           }

C      for( i = 99; i >= 0; i = i-1)
           printf( "Die %d-te Zahl ist: %d\n", i, daten[i]);
       }
```

Listing 6.12 Einlesen und Ausgeben von 100 Zahlen

Zuerst wird ein Array für 100 Zahlen erstellt (A). Dieses Array wird zunächst in aufsteigender Richtung durchlaufen, um Zahlen einzugeben (B). Nach erfolgter Eingabe wird das Array in absteigender Richtung durchlaufen, um die Zahlen auszugeben (C).

6.7.2 Mehrdimensionale Arrays

Der Wikipedia habe ich den folgenden Auszug entnommen:



Entfernungstabelle

Eine **Entfernungstabelle** gibt die Entfernung zwischen zwei geographischen Orten an, wobei es sich häufig um Großstädte handelt. Die Entfernung wird dabei in der Luftlinie oder für ein bestimmtes Verkehrsmittel angegeben.

Beispiel

	A	B	C	D	E
A	.	2	5	9	14
B	2	.	7	15	27
C	5	7	.	9	23
D	9	15	9	.	12
E	14	27	23	12	.

Abbildung 6.30 Entfernungstabelle

Eine Entfernungstabelle ist eine zweidimensionale Reihung von Daten gleichen Typs – also ein zweidimensionales Array:

```
   void main()
       {
       int start, ziel, distanz;
A      int entfernung[5][5] = {
                                 { 0, 2, 5, 9,14},
                                 { 2, 0, 7,15,27},
                                 { 5, 7, 0, 9,23},
                                 { 9,15, 9, 0,12},
                                 {14,27,23,12, 0}
                               };

       printf( "Gib zwei Orte (0-4) ein: ");
B      scanf( "%d %d", &start, &ziel);

C      distanz = entfernung[start][ziel];

D      printf( "Entfernung zwischen %d und %d: %d km\n", start,
                                                        ziel, distanz);
       }
```

Listing 6.13 Implementierung eines mehrdimensionalen Arrays

In dem Programm wird in (A) ein zweidimensionales Array für 5 × 5 `int`-Werte angelegt und initialisiert. Anschließend werden Zeilen- und Spaltenindex eingelesen (B). Mit den vorgegebenen Orten wird dann die Entfernung aus der Tabelle gelesen (C) und ausgegeben (D).

```
Gib zwei Orte (0-4) ein: 0 3
Entfernung zwischen 0 und 3: 9 km
```

Beim Anlegen und beim Zugriff werden jetzt zwei Indizes (Zeilenindex und Spaltenindex) verwendet. Natürlich können Zeilen- und Spaltenzahl in einem Array unterschiedlich sein:

```
double tabelle[100][200];
```

Wichtig ist auch hier wieder, dass die Indizierung der Elemente beim Index 0 beginnt. Im oben dargestellten Beispiel sind also die Zeilen von 0 bis 99 und die Spalten von 0 bis 199 nummeriert.

»Zeilen« und »Spalten« sind anschauliche, aber im Grunde genommen irreführende Begriffe, denn ein zweidimensionales Array ist im Rechner nicht »zweidimensional« organisiert. Insbesondere werden diese Begriffe dann unbrauchbar, wenn man Arrays noch höherer Dimension betrachtet, was problemlos möglich ist. Wenn Sie z. B. über einen Zeitraum von 100 Tagen sekündlich die Temperatur aufzeichnen wollten, könnten Sie ein vierdimensionales Array verwenden:

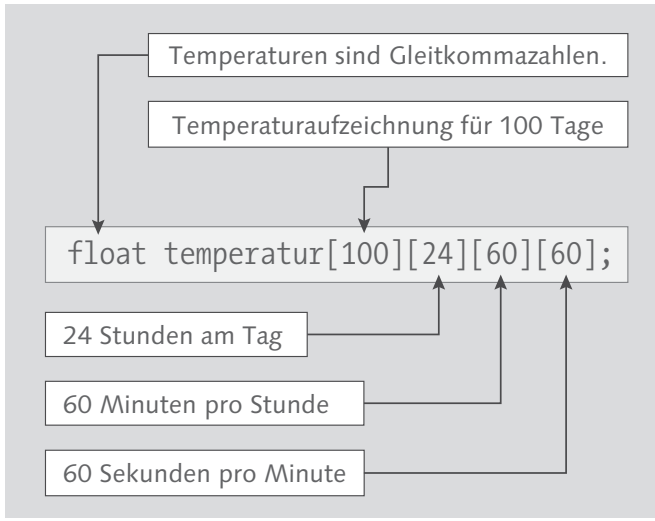


Abbildung 6.31 Beispielhaftes Array zur Aufzeichnung von Temperaturen

Die Temperatur am 5. Tag der Aufzeichnungen um 10 Sekunden nach 14:00 Uhr erhalten Sie dann mit dem Zugriff:

```
t = temperatur[4][14][0][10]
```

Beachten Sie auch hier wieder, dass die Zählung der Tage, Stunden, Minuten und Sekunden im Array mit 0 beginnt. Am Beispiel der Uhrzeit sehen Sie auch, dass das eine ganz natürliche Zählweise ist, da der Tag um 0 Uhr 0 beginnt.

6.8 Zeichenketten

Ein Wort der deutschen Sprache hat es bis in die englischen Zeitungen gebracht:


The Telegraph

HOME » NEWS » WORLD NEWS » EUROPE » GERMANY

Germany drops its longest word: Rindfleischeti...

Germany's longest word - Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz - a 63-letter long title of a law regulating the testing of beef, has officially ceased to exist.

Abbildung 6.32 Zeitungsmeldung zur deutschen Sprache

In einem Computer wollen wir auch Worte, Sätze und Texte variabler Länge verarbeiten können. Wir sprechen in diesem Zusammenhang allgemein von *Zeichenketten* oder *Strings*. Bisher kennen Sie nur Stringkonstanten wie `"Hallo Welt\n"` und einzelne Zeichen wie `'A'` oder `'\n'`. Beachten Sie hier den Unterschied:

- `'A'` ist der Buchstabe A.
- `"A"` ist eine Zeichenkette, die nur den Buchstaben A enthält.

Diese Unterscheidung ist keine Spitzfindigkeit, da es sich um grundsätzlich verschiedene Datentypen handelt. Den Datentyp für Zeichenketten werden Sie jetzt kennenlernen.

Da Zeichenketten Reihungen von Zeichen (`char`) sind, ist es naheliegend, zur Speicherung von Zeichenketten ein Array zu verwenden. Da Sie Zeichenketten im Rechner verändern möchten, ist es sinnvoll, eine Zeichenkette in einem ausreichend großen Array abzulegen, das noch Platz, z. B. für das Einfügen von Zeichen, lässt. Im Array stehen dann die Zeichencodes der einzelnen Zeichen:

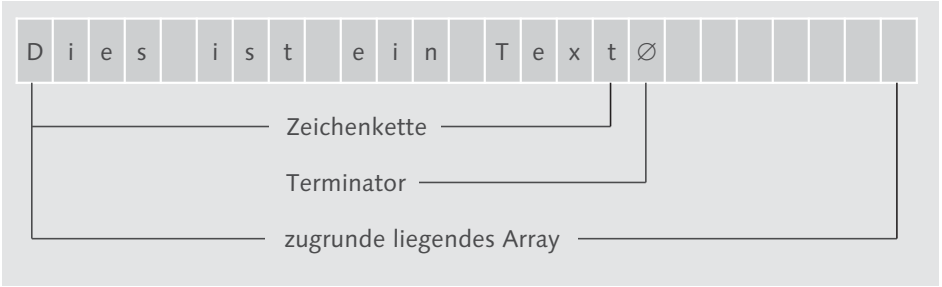


Abbildung 6.33 Aufbau einer Zeichenkette

Da die Zeichenkette unter Umständen nicht das ganze Array ausfüllt, benötigen Sie einen Code, der das Ende der Zeichenkette markiert (*Terminator*). Dieser Code darf natürlich nicht innerhalb der Zeichenkette als »normales« Zeichen vorkommen. Deshalb wählen Sie die 0 als Terminator. Bitte beachten Sie, dass es sich hier nicht um das Zeichen '0' (ASCII-Code 30_{16}), sondern um eine »richtige« 0 (00_{16}) handelt. Dieser Code (ASCII-Zeichen NUL) steht ja nicht für einen sinnvollen Buchstaben.

Bevor Sie mit einer Zeichenkette arbeiten können, müssen Sie ein Array ausreichender Größe bereitstellen. Zum Beispiel:

```
char wort[100];
```

In ein solches Array können Sie eine Zeichenkette mit `scanf` einlesen. Die Eingabe von Zeichenketten erfolgt mit der Formatanweisung `"%s"`:

```
scanf( "%s", wort);
```

Achtung

- Beim Einlesen von Zeichenketten in ein Array wird dem Variablennamen kein & vorangestellt7. Wenn Sie hier ein & setzen, wird Ihr Programm abstürzen.
- Bei der Eingabe wird nicht geprüft, ob das Array groß genug ist, um den String aufzunehmen. Werden mehr Zeichen eingegeben, als das Array aufnehmen kann, stürzt das Programm ab.

Mit `scanf` können Sie nur einzelne Wörter jeweils bis zum nächsten Trennzeichen (Leerzeichen, Tabulator oder Zeilenumbruch) einlesen. Möchten Sie eine komplette Eingabezeile, gegebenenfalls mit Leerzeichen und einschließlich des abschließenden Zeilenvorschubs, in ein Array einlesen, verwenden Sie die folgende Anweisung:

7 Diese scheinbare Abweichung von der Norm werde ich Ihnen später erklären. Nehmen Sie das für den Moment bitte zunächst ohne weitere Erklärung hin.

```
char zeile[100];

fgets( zeile, 100, stdin);
```

Dabei übergeben Sie die Länge des Eingabe-Arrays (im Beispiel 100), um zu verhindern, dass die Grenzen des Arrays überschritten werden.

Da die Zeichenkette nach dem Einlesen in einem Array zur Verfügung steht, können Sie über den Index auf jedes Zeichen zugreifen und es bei Bedarf verändern.

```
wort[0] = 'A';     // Erster Buchstabe A
wort[1] = wort[0]; // Zweiter Buchstabe auch A
wort[2] = 0;       // Terminator, Zeichenkette ist "AA"
```

Achtung: Bei der Veränderung von Zeichenketten müssen Sie Folgendes unbedingt beachten:

- Die Nummerierung der Zeichen beginnt beim Index 0. Wenn die Zeichenkette n Zeichen hat, sind diese von 0 bis n-1 nummeriert. Der Terminator hat den Index n.
- Die Zeichenkette befindet sich in einem Array fester Länge. Sie müssen darauf achten, dass bei Veränderungen (z. B. durch Anfügen von Buchstaben) die Grenzen des zugrunde liegenden Arrays nicht überschritten werden.
- Wegen des Terminators muss das Array, das den String aufnimmt, mindestens ein Element mehr haben, als der String Zeichen enthält.
- Die Zeichenkette muss nach eventuellen Manipulationen immer konsistent sein. Insbesondere bedeutet das, dass das Terminator-Zeichen korrekt positioniert werden muss.

Auf die Rahmenbedingungen muss der Programmierer achten. Verletzt er eine dieser Bedingungen, stürzt das Programm in der Regel ab.

Die Ausgabe eines Strings, auch mit Leerzeichen oder Zeilenumbrüchen, kennen Sie schon von Stringkonstanten. Man verwendet `printf` mit der Formatanweisung `"%s"`:

```
printf( "%s", wort);
```

Jetzt können Sie ein erstes zusammenhängendes Beispiel programmieren. Sie werden ein Wort einlesen, um dann seine Länge festzustellen:

```
   void main()
       {
A      char wort[100];
       int i;
```



```
       printf( "Wort: ");
B      scanf( "%s", wort);

C      for( i = 0; wort[i] != 0; i++)
D          ;

E      printf( "%s hat %d Zeichen\n", wort, i);
       }
```

Listing 6.14 Verwendung einer Zeichenkette

Um eine Zeichenkette zu verwenden, wird das Array für die Zeichenkette bereitgestellt (A). In das Array kann dann das Wort eingelesen werden (B).

Beachten Sie, dass kein & vor dem Variablennamen steht!

In einer Schleife werden die Zeichen im Array gezählt, solange nicht der Terminator auftaucht (C). Die Zählung findet komplett im Schleifenkopf statt, beachten Sie, dass der Schleifenkörper leer ist (D).

Abschließend werden die Zeichenkette und ihre Länge ausgegeben (E):

```
Wort: Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz
Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz hat 63 Zeichen
```

Im nächsten Beispiel werden Sie ein Wort einlesen und prüfen, ob es sich bei diesem Wort um ein Palindrom handelt. Unter einem *Palindrom* verstehen wir ein Wort oder eine Wortfolge, die vorwärts und rückwärts gelesen gleich ist, wobei es auf Leerzeichen, Satzzeichen oder Groß- bzw. Kleinschreibung nicht ankommt. Palindrome sind z. B. »otto«, »lagerregal« oder »rentner«. Schreiben Sie bei der Eingabe alles klein und zusammen, damit Sie keinen Zusatzaufwand bei der Überprüfung haben:

```
   void main()
       {
       char wort[100];
       int vorn, hinten;

       printf( "Wort: ");
A      scanf( "%s", wort);

B      for( hinten = 0; wort[hinten] != 0; hinten++)
           ;
```

```
    for( vorn = 0, hinten--; vorn < hinten; vorn++, hinten--)
C       {
        if( wort[vorn] != wort[hinten])
D           break;
        }

    if( vorn < hinten)
E       printf( "Kein Palindrom\n");
    else
F       printf( "Palindrom\n");
    }
```

Listing 6.15 Palindromerkennung

Zuerst müssen Sie das Wort, das Sie prüfen wollen, einlesen (A). In diesem Wort wird dann das Wortende gesucht (B). Nach Ablauf der Schleife ist in `hinten` der Index des Terminators. Dieser wird im folgenden Schleifenkopf (C) daher um 1 zurückgesetzt. In dieser Schleife wird das Wort von vorn vorwärts und von hinten rückwärts durchlaufen, solange `vorn` noch vor `hinten` liegt. Innerhalb der Schleife erfolgt die Prüfung. Wenn vorn ein anderes Zeichen steht als hinten, wird die Schleife abgebrochen (D).

Außerhalb der Schleife erfolgt die Prüfung des Ergebnisses, wenn die Schleife vorzeitig abgebrochen wurde (E), handelt es sich um kein Palindrom (F).

```
Wort: einnegermitgazellezagtimregennie
Palindrom
```

Bis jetzt haben Sie noch keine Zeichenketten verändert oder sogar neue Zeichenketten erstellt. Das machen Sie im nächsten Beispiel. Sie kennen sicherlich das Spiel »Galgenmännchen«, bei dem man versucht, durch Raten von Buchstaben in möglichst wenig Versuchen ein unbekanntes Wort zu ermitteln. Im Folgenden sehen Sie meine Lösung, die insofern etwas merkwürdig ist, als der Rater selbst das Geheimwort eingibt, es angezeigt bekommt und dann aufgefordert wird, das Wort zu raten. Aber im Vordergrund steht ja nicht das Spiel, sondern das Programm:

```
    void main()
        {
A       char wort[100], anzeige[100];
        char versuch;
        int nochzuraten, i, anzahl;
```



```
        printf( "Wort: ");
B       scanf( "%s", wort);

C       for( nochzuraten = 0; wort[nochzuraten] != 0; nochzuraten++)
            anzeige[nochzuraten] = '-';
D       anzeige[nochzuraten] = 0;

E       for( anzahl = 1; nochzuraten != 0; anzahl++)
            {
            printf( "%s\n", anzeige);
            printf( "%d-ter Versuch: ", anzahl);

F           scanf( "\n%c", &versuch);


G           for( i = 0; wort[i] != 0; i++)
                {
H               if( (wort[i] == versuch) && (anzeige[i] == '-'))
                    {
                    anzeige[i] = versuch;
I                   nochzuraten--;
                    }
                }
            }
        printf( "%s\n", anzeige);
        printf( "Du hast %d Versuche benoetigt\n", anzahl-1);
        }
```

Listing 6.16 Galgenmännchen

Das Programm startet mit der Erstellung von Puffern für das Ratewort und den Anzeigestring (A), das zu ratende Wort wird in (B) eingelesen.

In (C) wird der Anzeigestring aufbereitet, indem für alle Zeichen ein `'-'` gesetzt wird. Gleichzeitig wird gezählt, wie viele Zeichen zu raten sind.

Nach Ablauf dieser Schleife wird der Anzeigestring terminiert (D). Jetzt beginnt das eigentliche Spiel mit einer Schleife über alle Rateversuche (E).

In jedem Schleifendurchlauf erfolgt die Eingabe eines neuen Zeichens (F). Vor dem Lesen wird mit `\n` der noch in der Eingabe stehende Zeilenvorschub aus der letzten Eingabe konsumiert. Nach der Eingabe des zu ratenden Zeichens läuft eine Schleife über das zu ratende Wort (G).

Wenn der geratene Buchstabe mit dem Zeichen im Wort übereinstimmt und in der Anzeige noch ein '-' steht, wird das Zeichen in die Anzeige übernommen, und es ist nur noch ein Buchstabe weniger zu erraten (H) bis (I). Im folgenden Bildschirmdialog habe ich versucht, das Wort »mississippi« zu erraten:

```
Wort: mississippi
-----------
1-ter Versuch: i
-i--i--i--i
2-ter Versuch: a
-i--i--i--i
3-ter Versuch: s
-ississi--i
4-ter Versuch: p
-ississippi
5-ter Versuch: m
mississippi
Du hast 5 Versuche benoetigt
```

An dieser Stelle sind einige ergänzende Informationen zur Eingabe mit `scanf` angebracht. Alle Eingaben des Benutzers landen in einem Zwischenpuffer, aus dem `scanf` nach und nach die geforderten Eingaben abruft. Mit `%c` wird nur ein einzelnes Zeichen abgerufen. Alles, was der Benutzer zusätzlich eingegeben hat (z. B. Leerzeichen oder Zeilenumbrüche), bleibt in der Eingabe stehen und muss gegebenenfalls durch gezielte Leseoperationen beseitigt werden. Im oben dargestellten Beispiel wird durch die Anweisung `\n%c` vor dem Lesen des Eingabezeichens (`%c`) der von der letzten Eingabe noch anstehende Zeilenumbruch (`\n`) beseitigt.

Häufig will man Zeichenketten kopieren oder miteinander vergleichen. Da ist es verlockend, es genauso wie bei Zahlen zu machen:

```
int a = 1, b;

b = a;

if( a == b)
   ...
```

Das geht bei Zeichenketten so nicht:

> Zeichenketten können nicht mit `=` kopiert und nicht mit `==` oder `!=` miteinander verglichen werden. Bei einer Kopie müssen die Zeichenketten Zeichen für Zeichen kopiert und bei einem Vergleich Zeichen für Zeichen verglichen werden.

Nach Ihrem derzeitigen Kenntnisstand bedeutet das, dass Sie das Kopieren und das Vergleichen von Strings selbst implementieren müssen. Fangen Sie mit dem Kopieren an:

```
   char original[100], kopie[100];
   int i;

   printf( "Eingabe:  ");
   scanf( "%s", original);

A  for( i = 0; original[i] != 0; i++)
       kopie[i] = original[i];
B  kopie[i] = 0;

   printf( "\nOriginal: %s", original);
   printf( "\nKopie:    %s", kopie);
```

Listing 6.17 Kopieren von Zeichenketten

In dem Programm wird Zeichen für Zeichen von `original` nach `kopie` kopiert (A), abschließend wird die Kopie in (B) terminiert.

```
Eingabe:  Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz

Original: Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz
Kopie:    Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz
```

Im nächsten Programmfragment werden zwei Strings eingelesen und miteinander verglichen. Das Vergleichsergebnis wird auf dem Bildschirm ausgegeben:

```
   char wort1[100], wort2[100];
   int i;

   printf( "Wort1:  ");
   scanf( "%s", wort1);
   printf( "Wort2:  ");
   scanf( "%s", wort2);

A  for( i = 0; (wort1[i] != 0) && (wort1[i] == wort2[i]); i++)
       ;
B  if( wort1[i] == wort2[i])
       printf( "Die Worte sind gleich\n");
   else
       printf( "Die Worte sind verschieden");
```

Listing 6.18 Vergleich von Zeichenketten

In dem Programm werden die eingegebenen Zeichenketten Zeichen für Zeichen geprüft (A). Solange das erste Wort noch nicht beendet ist und die Zeichen im ersten und zweiten Wort gleich sind, wird weitergeprüft.

Am zuletzt geprüften Zeichen können Sie erkennen, ob die beiden Worte gleich waren (B).

```
Wort1:
Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz
Wort2:
Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesezz
Die Worte sind verschieden
```

Die Steuerung dieser Schleife ist durchaus trickreich, sodass wir uns die Testbedingung noch einmal genauer anschauen wollen:

Wort1 beendet	Wort2 beendet	Zeichen gleich	Aktion	Ergebnis
nein	nein	nein	abbrechen	verschieden
nein	nein	ja	weitermachen	offen
nein	ja	nein	abbrechen	verschieden
nein	ja	ja	kann nicht vorkommen	
ja	nein	nein	abbrechen	verschieden
ja	nein	ja	kann nicht vorkommen	
ja	ja	nein	kann nicht vorkommen	
ja	ja	ja	abbrechen	gleich
nein	nein	nein	abbrechen	verschieden

Tabelle 6.8 Die Testbedingungen im Detail

Sie sehen, dass in jedem vorkommenden Fall die Schleife korrekt fortgesetzt oder abgebrochen wird.

Sie werden an dieser Stelle mit Recht einwenden, dass es nicht sein darf, dass man für Grundaufgaben wie Kopieren oder Vergleichen immer und immer wieder den gleichen Code schreiben muss. Es muss in einer Programmiersprache Möglichkeiten geben, solche Aufgaben einmal und dann auch endgültig zu lösen. Mit den dazu erforderlichen Sprachmitteln werden wir uns im nächsten Kapitel beschäftigen. Zunächst aber schließen wir dieses Kapitel mit einigen Programmierbeispielen ab.

6.9 Programmierbeispiele

Beispielprogramme mit Verwendung von Zeichenketten und Arrays schließen dieses Kapitel ab.

6.9.1 Buchstabenstatistik

Sie werden ein Programm erstellen, das eine komplette Textzeile einliest und dann eine Statistik über die vorkommenden Buchstaben (a–z) erzeugt:

```
    void main()
        {
        char text[100];
A       int statistik[26];
        int i;

B       for( i = 0; i < 26; i++)
            statistik[i] = 0;

        printf( "Text: ");
C       scanf( "%s", text);

D       for( i = 0; text[i] != 0; i++)
            {
E           if( (text[i] >= 'a') && ( text[i] <= 'z'))
                statistik[text[i]-'a']++;
            }

        printf( "\nAuswertung:\n");
F       for( i = 0; i < 26; i++)
            printf( "%c: %d\n", 'a' + i, statistik[i]);
        }
```

Listing 6.19 Erstellen einer Buchstabenstatistik

Auf das Zählen der Buchstaben möchte ich noch etwas genauer eingehen. Im Array `statistik` haben wir 26 Zähler für das Vorkommen der Buchstaben (A). Das Vorkommen von `a` wollen wir in `statistik[0]`, das Vorkommen von `b` in `statistik[1]` etc. zählen. Wie kommen Sie nun von einem Zeichen zum Index seines Zählers? Ganz einfach, Sie ziehen den Code von `a` als Zahlenwert vom Code des Zeichens ab. So ergibt sich die Formel

```
statistik[zeichen[i]-'a']
```

Und wie gelangen Sie umgekehrt von einem Index zu dem Zeichen, das zu diesem Index gehört? Ganz einfach: Sie addieren den Zeichencode von a zum Index hinzu. So erhalten Sie die Formel

```
'a' + i
```

in der Ausgabe der Statistik.

Mit dieser Erläuterung ist der Rest des Programmes schnell erklärt. In (B) werden alle 26 Buchstabenzähler auf 0 gesetzt. Dann wird in (C) der Text eingelesen und mit (D) über die gesamte Eingabe iteriert. Dabei werden im Text nur Zeichen betrachtet, die zwischen a und z liegen (E). Abschließend erfolgt die Ausgabe der Statistik (F).

Sie werden nun das Programm testen und dazu Eingaben verwenden, die möglichst viele verschiedene Buchstaben enthalten. In der Wikipedia finden Sie eine Liste von Pangrammen[8], die als Testfälle geeignet sind:

Vogel Quax zwickt Johnys Pferd Bim. (29 Buchstaben)
Sylvia wagt quick den Jux bei Pforzheim. (33 Buchstaben)
Prall vom Whisky flog Quax den Jet zu Bruch. (35 Buchstaben)
Jeder wackere Bayer vertilgt bequem zwo Pfund Kalbshaxen. (49 Buchstaben)
Franz jagt im komplett verwahrlosten Taxi quer durch Bayern. (51 Buchstaben)
Stanleys Expeditionszug quer durch Afrika wird von jedermann bewundert. (61 Buchstaben)

Abbildung 6.34 Beispiele von Pangrammen

Eines der bekanntesten Pangramme der englischen Sprache ist übrigens »The quick brown fox jumps over the lazy dog«. Dieses Pangramm wurde in der Steinzeit der Datenverarbeitung benutzt, um Fernschreibverbindungen mit einem vollständigen Satz an Zeichen zu testen. Noch viele Modems können heute diesen Satz auf Knopfdruck automatisch erzeugen. Sie testen Ihr Programm allerdings mit einem anderen Pangramm (ohne Leerzeichen):

```
Text: franzjagtimkomplettverwahrlostentaxiquerdurchbayern

Auswertung:
a: 5
b: 1
c: 1
d: 1
e: 5
f: 1
```

8 Das sind Texte, die alle Zeichen des Alphabets enthalten.


```
g: 1
h: 2
i: 2
j: 1
k: 1
l: 2
m: 2
n: 3
o: 2
p: 1
q: 1
r: 6
s: 1
t: 5
u: 2
v: 1
w: 1
x: 1
y: 1
z: 1
```

6.9.2 Sudoku

Sicher haben Sie sich schon einmal an einem Sudoku-Rätsel versucht. Man muss ein 9 × 9-Zahlenschema, in dem gewisse Zahlen vorgegeben sind, so ausfüllen, dass in jeder Zeile und in jeder Spalte und in jedem der neun Teilquadrate immer alle Zahlen von 1 bis 9 stehen. Sie werden ein kleines Programm schreiben, das Sie bei der Lösung unterstützt. Das Programm enthält aber nur die Ein- und Ausgabe. Weitergehende Funktionen sind zunächst einmal nicht vorgesehen.

Das 9 × 9-Zahlenfeld bilden Sie natürlich auf einem zweidimensionalen Array ab:

```
int sudoku[9][9];
```

Der Einfachheit halber nummerieren Sie die Zeilen und Spalten des Sudokus von 0 bis 9. Dann kann es auch schon losgehen:

```
    void main()
        {
A       int sudoku[9][9] = {
                           {5,0,9,7,0,2,0,4,0},
                           {7,6,0,3,4,8,9,1,5},
                           {1,3,4,5,0,9,0,7,0},
```

```
                    {6,7,1,0,0,0,0,0,0},
                    {8,0,5,6,2,1,7,3,4},
                    {0,0,3,0,5,0,1,6,9},
                    {0,5,8,1,0,0,3,2,7},
                    {3,0,0,2,0,0,0,9,6},
                    {9,0,0,0,7,3,8,5,0},
                    };
    int zeile, spalte, zahl;

B   for( zahl = 1; zahl !=0; )
        {
        printf( "\n");
C       for( zeile = 0; zeile < 9; zeile++)
            {
D           if( zeile %3 == 0)
                printf( "+-------+-------+-------+\n");
E           for( spalte = 0; spalte < 9; spalte++)
                {
F               if( spalte % 3 == 0)
                    printf( "| ");
                if(sudoku[zeile][spalte] > 0)
G                   printf( "%d ", sudoku[zeile][spalte]);
                else
                    printf( "  ");
                }
H           printf( "|\n");
            }
I       printf( "+-------+-------+-------+\n");
        printf( "Zeile Spalte Zahl: ");
J       scanf( "%d %d %d", &zeile, &spalte, &zahl);
K       sudoku[zeile][spalte] = zahl;
        }
    }
```

Listing 6.20 Sudoku

In dem Programm wird in (A) eine Sudoku-Aufgabe festgelegt. Der Wert 0 steht für ein nicht ausgefülltes Feld.

Innerhalb einer Schleife wird so lange fortgefahren, wie keine 0 eingegeben wurde (B).

Das Sudoku wird zeilen- und spaltenweise ausgegeben (C) und (E), jedes dritte Mal kommt dabei eine Trennlinie (D) bzw. ein Trennstrich (F). Zahlen größer als 0 werden ausgegeben (G), ansonsten werden Leerzeichen dargestellt. Die Ausgaben werden

mit einem Zeilenabschluss und einer Abschlusszeile beendet (H) und (I). Am Ende eines jeden Durchlaufs erfolgen dann die Eingabe (J) und der Eintrag in das Sudoku-Feld (K).

Der folgende Screenshot zeigt das Programm bei der Arbeit:

```
+-------+-------+-------+
| 5   9 | 7   2 |   4   |
| 7 6   | 3 4 8 | 9 1 5 |
| 1 3 4 | 5   9 |   7   |
+-------+-------+-------+
| 6 7 1 |       |       |
| 8   5 | 6 2 1 | 7 3 4 |
|     3 |   5   | 1 6 9 |
+-------+-------+-------+
|   5 8 | 1     | 3 2 7 |
| 3     | 2     |   9 6 |
| 9     |   7 3 | 8 5   |
+-------+-------+-------+
Zeile Spalte Zahl:  0 1 8

+-------+-------+-------+
| 5 8 9 | 7   2 |   4   |
| 7 6   | 3 4 8 | 9 1 5 |
| 1 3 4 | 5   9 |   7   |
+-------+-------+-------+
| 6 7 1 |       |       |
| 8   5 | 6 2 1 | 7 3 4 |
|     3 |   5   | 1 6 9 |
+-------+-------+-------+
|   5 8 | 1     | 3 2 7 |
| 3     | 2     |   9 6 |
| 9     |   7 3 | 8 5   |
+-------+-------+-------+
Zeile Spalte Zahl:
```

Wenn Sie es sich zutrauen, können Sie zu diesem Programm noch Eingabeprüfungen, Lösungshinweise und das Erzeugen von Aufgaben hinzufügen. Das sind jedoch anspruchsvolle Aufgaben, die Ihre derzeitigen Programmierkenntnisse noch übersteigen.

6.10 Aufgaben

A 6.1 Erstellen Sie ein Programm, das einen String und einen Buchstaben übergeben bekommt und dann ausgibt, wie oft der Buchstabe in dem String vorkommt.

A 6.2 Schreiben Sie ein Programm, das die Reihenfolge der Zeichen in einem String umkehrt.

A 6.3 Schreiben Sie ein Programm, das alle `'e'` aus einem String entfernt.

A 6.4 Erweitern Sie das Programm zur Palindromerkennung so, dass nicht zwischen Groß- und Kleinbuchstaben unterschieden wird. Es sollen also Worte wie »Retsinakanister« korrekt als Palindrom erkannt werden.

A 6.5 Schreiben Sie ein Programm, das eine nur aus Ziffern bestehende Zeichenkette einliest und aus dem Eingabestring eine `int`-Zahl berechnet.

A 6.6 Schreiben Sie ein Programm, das zehn Zahlen einliest und anschließend auf Wunsch bestimmte Zahlen wieder ausgibt. Das Programm soll wie folgt arbeiten:

```
Gib die 1. Zahl ein: 23
Gib die 2. Zahl ein: 17
Gib die 3. Zahl ein: 234
Gib die 4. Zahl ein: 875
Gib die 5. Zahl ein: 328
Gib die 6. Zahl ein: 0
Gib die 7. Zahl ein: 519
Gib die 8. Zahl ein: 712
Gib die 9. Zahl ein: 1000
Gib die 10. Zahl ein: 14

Welche Zahl soll ich ausgeben: 3
Die 3. Zahl ist 234

Welche Zahl soll ich ausgeben: 9
Die 9. Zahl ist 1000

Welche Zahl soll ich ausgeben: 2
Die 2. Zahl ist 17
```

A 6.7 Schreiben Sie ein Programm, das zehn Zahlen einliest und anschließend der Größe nach sortiert wieder ausgibt.

A 6.8 Unter einem magischen Quadrat der Kantenlänge 5 verstehen wir eine Anordnung der Zahlen 1 bis 25 in einem quadratischen Schema auf eine Weise, dass die Summen in allen Zeilen, Spalten und den beiden Hauptdiagonalen gleich sind. Das folgende Beispiel zeigt ein solches Quadrat:

19	3	12	21	10
11	25	9	18	2
8	17	1	15	24
5	14	23	7	16
22	6	20	4	13

Erstellen Sie ein Programm, das überprüft, ob es sich bei einem 5 × 5-Quadrat um ein magisches Quadrat handelt.

A 6.9 Magische Quadrate ungerader Kantenlänge lassen sich nach folgendem Verfahren konstruieren:

1. Positioniere die 1 in dem Feld unmittelbar unter der Mitte des Quadrats!
2. Wenn die Zahl `x` in der Zeile `i` und der Spalte `k` positioniert wurde, dann versuche, die Zahl `x+1` in der Zeile `i+1` und der Spalte `k+1` abzulegen! Handelt es sich bei diesen Angaben um ungültige Zeilen- oder Spaltennummern, wende Regel 4 an! Ist das Zielfeld bereits besetzt, wende Regel 3 an!
3. Wird versucht, eine Zahl in einem bereits besetzten Feld in der Zeile `i` und der Spalte `k` zu positionieren, versuche stattdessen die Zeile `i+1` und die Spalte `k-1`. Handelt es sich bei diesen Angaben um ungültige Zeilen- oder Spaltennummern, wende Regel 4 an. Ist das Zielfeld bereits besetzt, wende Regel 3 erneut an!
4. Die Zeilen- und Spaltennummern laufen von `0` bis `n-1`. Ergibt sich im Laufe des Verfahrens eine zu kleine Zeilen- oder Spaltennummer, setze die Nummer auf den Maximalwert `n-1`! Ergibt sich eine zu große Spalten- oder Zeilennummer, setze die Nummer auf den Minimalwert `0`!

Erstellen Sie nach diesen Angaben ein Programm, das für ungerade Kantenlängen von 3 bis 9 ein magisches Quadrat erzeugen kann.

A 6.10 Informieren Sie sich im Internet, was unter einem Vigenère-Schlüssel zu verstehen ist. Erstellen Sie dann ein Programm, das einen eingegebenen String mit einem Passwort verschlüsselt und wieder entschlüsselt.

Kapitel 19

Einführung in C++

Der oft zitierte »Paradigmenwechsel« ist meist leeres Gerede, in C++ gibt es ihn jedoch wirklich. Dieses Kapitel bereitet Sie darauf vor.

Wenn Sie das Buch bis zu diesem Punkt durchgearbeitet haben, beherrschen Sie die Grundlagen der Programmiersprache C; Sie kennen verschiedene Algorithmen, können neue Algorithmen umsetzen und eigene Programme schreiben.

Bereits im ersten Kapitel dieses Buches haben Sie erfahren, dass Programmiersprachen bestimmte Paradigmen unterstützen. Die Programmiersprache C basiert auf dem prozeduralen Paradigma. Das prozedurale Programmieren haben Sie mittlerweile kennengelernt. C++ unterstützt dazu auch die Objektorientierung. Im verbleibenden Teil des Buches werde ich Ihnen das objektorientierten Paradigma und die damit verbundene »Denke« vorstellen.

C++ bietet aber auch wichtige Erweiterungen gegenüber C, die gar nichts mit objektorientierter Programmierung zu tun haben. Dennoch bringen auch diese Erweiterungen eine deutliche Erleichterung bei der prozeduralen Programmierung. Als Einstieg in C++ zeige ich Ihnen zuerst diese Veränderungen, bevor ich im folgenden Kapitel die eigentliche objektorientierte Entwicklung erläutere.

Bei der Entwicklung von C++ ist viel Wert auf die Kompatibilität gelegt worden, sodass sich die meisten C-Programme auch mit einem C++-Compiler übersetzen lassen. Lassen Sie sich aber von diesem Abschnitt nicht dazu verleiten, C++ auf ein leicht erweitertes C zu reduzieren. Sie werden in den folgenden Kapiteln noch völlig neue Möglichkeiten der Modellierung entdecken.

19.1 Schlüsselwörter

In C++ bleiben die bereits in C eingeführten Schlüsselwörter in ihrer Bedeutung erhalten:

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`

Tabelle 19.1 Schlüsselwörter in C

case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Tabelle 19.1 Schlüsselwörter in C (Forts.)

Zusätzlich wurde in C++ eine Reihe weiterer Schlüsselwörter eingeführt. Die meisten dieser zusätzlichen Schlüsselwörter und deren Verwendung lernen Sie im Laufe des Buches kennen:

asm	dynamic_cast	namespace	reinterpret_cast	try
bool	explicit	new	static_cast	typeid
catch	false	operator	template	typename
class	friend	private	this	using
const_cast	inline	public	throw	virtual
delete	mutable	protected	true	wchar_t

Tabelle 19.2 Neue Schlüsselwörter in C++

Auch die neuen Schlüsselwörter sind reservierte Wörter, die z. B. nicht zur Benennung von Variablen verwendet werden können. Wenn C-Programme diese Schlüsselwörter verwendet haben, lassen sie sich mit einem C++-Compiler nicht mehr übersetzen und müssen vorher angepasst werden.

19.2 Kommentare

C++ bietet zusätzliche Kommentierungsmöglichkeiten, die die Dokumentation von Code deutlich erleichtern. In C musste ein Kommentar ähnlich der Klammersetzung immer gestartet /* und beendet */ werden. Mit dieser Methode können leicht mehrere Zeilen Kommentar ergänzt oder »Codebereiche« auskommentiert werden. Die Syntax macht es aber umständlich, eine einzelne Zeile zu kommentieren. In C++ können mit // Kommentare erstellt werden, die bis zum Ende der Zeile reichen:


```
y = 2; /* C-Style-Kommentar */
// C++-Kommentar ab dem Start der Zeile
x = 1; // Hier steht ein C++-Kommentar für den Rest der Zeile
```

Solche Kommentare können überall in der Zeile starten. Prinzipiell ist es sogar möglich, solche einzeiligen Kommentare über das Zeilenende hinaus zu verlängern. Das geht mit einem *Backslash* \ am Ende der Zeile:

```
A  z = 3; // Ein C++-Kommentar kann (unüblich!) auch so -> \
B  z = x * y; fortgesetzt werden
```

Von dieser Möglichkeit (A) sollten Sie aber keinen Gebrauch machen. Die Verwendung dieser Variante ist sehr unüblich, da die Weiterführung des Kommentars extrem leicht übersehen wird, wie Sie hier vielleicht schon selbst erleben. In Zeile (B) startet kein neuer Code, stattdessen handelt es sich auch hier immer noch um Kommentartext.

19.3 Datentypen, Datenstrukturen und Variablen

Auch beim Thema Datentypen, -strukturen und Variablen gibt es einige Neuigkeiten. Die meisten Änderungen entfalten erst mit der Objektorientierung ihre volle Wirkung, einige Punkte möchte ich Ihnen aber vorab vorstellen.

19.3.1 Automatische Typisierung von Aufzählungstypen

Die Aufzählungstypen, die Sie aus C kennen, gibt es auch in C++:

```
enum wochentag {Mo, Di, Mi, Do, Fr, Sa, So};
```

Wenn Sie eine Variable des neuen Typs in C anlegen, müssen Sie so vorgehen:

```
enum wochentag wt_c; /* C-Stil */
```

Das Schlüsselwort `enum` muss hier vor dem Namen des Datentyps erneut angegeben werden. In C++ kann die explizite Verwendung von `enum` entfallen. Mit der oben erfolgten Anlage des `enum` ist durch die *automatische Typisierung* implizit ein neuer Datentyp eingeführt worden, der im Weiteren direkt verwendet werden kann:

```
wochentag wt_cpp; // C++-Stil
```

19.3.2 Automatische Typisierung von Strukturen

So wie für Aufzählungstypen `enum` wird in C++ auch für Datenstrukturen `struct` implizit ein Datentyp erstellt. Die Deklaration erfolgt weiter, wie Sie es bereits kennen:

```
struct punkt
    {
    int x;
    int y;
    };
```

In C erfolgt dagegen die Verwendung bekanntlich so:

```
struct punkt p_c; /* C-Stil */
```

In C++ kann auch für eine `struct` wie für ein `enum` das zusätzliche Schlüsselwort an dieser Stelle entfallen:

```
punkt p_cpp; // C++-Stil
```

Damit sind die in C oft verwendeten Konstruktionen

```
typedef struct punkt PUNKT
```

oder

```
# define PUNKT struct punkt
```

hier nicht mehr notwendig und auch nicht mehr üblich. Aufgrund der Kompatibilität mit C können die Konstruktionen aber weiterverwendet werden.

19.3.3 Vorwärtsverweise auf Strukturen

C++ erlaubt *Vorwärtsverweise* innerhalb von Strukturen. Damit kann später auf Strukturen verwiesen werden, die an dieser Stelle noch nicht definiert sind. Dies ist insbesondere notwendig, wenn es sich um Zirkelverweise handelt.

```
A   struct student;          // Vorwaertsverweis auf student
    struct bachelorarbeit;   // Vorwaertsverweis auf bachelorarbeit

    struct student
        {
        char name[50];
        bachelorarbeit* ba; // Verweis auf bachelorarbeit
        };

    struct bachelorarbeit
        {
        char thema[200];
        float note;
        student* stud; // Verweis auf den studenten
        };
```

Listing 19.1 Verwendung des Vorwärtsverweises

In dem Beispiel finden sich zuerst die Vorwärtsverweise auf die Strukturen `student` und `bachelorarbeit` (A). Diese Angaben sagen nur, dass entsprechende Strukturen noch bereitgestellt werden. In der Deklaration der Struktur `student` erfolgt dann ein Verweis auf die Struktur `bachelorarbeit` (B) (hier noch nicht vollständig deklariert).

Beachten Sie dabei, dass in den Datenstrukturen nur Zeiger auf noch nicht deklarierte Strukturen vorkommen dürfen. Durch den Vorwärtsverweis wird es nicht möglich, die Struktur selbst zu verwenden, da die einzelnen Felder und die Größe der einzubindenden Struktur an dieser Stelle noch nicht bekannt sind. Das folgende Beispiel führt daher zu einem Fehler bei der Übersetzung:

```
struct student;          // Vorwaertsverweis auf student
struct bachelorarbeit;   // Vorwaertsverweis auf bachelorarbeit

struct student
    {
    char name[50];
    bachelorarbeit ba; // Fehler, falls bachelorarbeit
                       // noch nicht vollständig deklariert wurde
    };
```

Listing 19.2 Fehlerhafte Verwendung des Vorwärtsverweises

19.3.4 Der Datentyp bool

Sie haben im Kapitel über Logik erfahren, wie C den Datentyp `int` verwendet, um die logischen Aussagewerte *wahr* und *falsch* mit den Zahlenwerten 1 und 0 abzubilden. Genau genommen, ist das Ergebnis einer logischen Operation aber keine Zahl, sondern ein Wahrheitswert – eben *wahr* oder *falsch*. Viele andere Programmiersprachen haben daher einen eigenen Datentyp für Wahrheitswerte.

Die in C verwendete Sichtweise hat dabei durchaus Vorteile, weil man mit logischen Ergebnissen wie mit Zahlen rechnen kann und z. B. die Ergebnisse einer logischen Operation zur Gesamtzahl der Ergebnisse mit dem Wert `wahr` aufaddieren kann.

In C++ hat man den Mittelweg beschritten und einen Datentyp `bool` eingeführt, der die beschriebenen Eigenschaften vereinigt und mit `int` kompatibel ist.

Der Datentyp `bool` kann die Werte `true` und `false` annehmen . Gleichzeitig gilt aber auch `true` = `1` und `false` = `0`. Damit können Sie logische Ausdrücke, die Sie bisher mit `int` gebildet haben, jetzt auch mit `bool` bilden, z. B. so:

```
bool b1, b2, b3, b4;

b1 = true;
b2 = !b1;
b3 = 3 > 2;

if( b2 != false)
    {
    b4 = b1 || b2;
    }
```

Listing 19.3 Verwendung des Datentyps bool

Der Datentyp `bool` ist zwar kompatibel mit dem Datentyp `int`, aber nicht identisch. Er verhält sich wie ein `int`, der nur die Werte 0 und 1 aufnehmen kann. Eine Wertzuweisung ist damit in beide Richtungen fehlerfrei möglich. Das Codefragment demonstriert dies, und

```
bool b = 7;
int ausgabe = b;
printf( "Der Wert von ausgabe ist '%d'\n", ausgabe);
```

erzeugt die folgende Ausgabe:

```
Der Wert von ausgabe ist '1'
```

Viele Entwickler setzen den Datentyp `bool` mit `true` und `false` kaum ein und verwenden weiterhin die Ihnen bereits bekannten Mechanismen aus C.

19.3.5 Verwendung von Konstanten

Sie haben die Verwendung *symbolischer Konstanten* in C kennengelernt, etwa um Arrays zu definieren:


```
#define ANZAHL 10

int array[ANZAHL]; /* Vorgehen in ANSI-C */
```

Dies hatte den Grund, dass die folgende Definition in C nicht möglich war:

```
const int anzahl = 10;
int array[anzahl];  /* In ANSI-C nicht moeglich */
```

Genau dieses Konstrukt ist in C++ möglich und erwünscht.

```
const int anzahl = 10;
int array[anzahl]; // Typische Vorgehensweise in C++
```

Der definierte konstante Wert `anzahl` kann bereits zur Übersetzungszeit verwendet werden.

Die Verwendung von *Konstanten* anstelle *symbolischer Konstanten* sollte in C++ bevorzugt werden. Symbolische Konstanten resultieren nur in einer Textersetzung durch den Präprozessor. Echte Konstanten ermöglichen dem Compiler eine Typprüfung und erhöhen damit die Typsicherheit.

Der Präprozessor, der in C noch von entscheidender Bedeutung ist, verliert in C++ durch den Ersatz symbolischer Konstanten an Relevanz. Auch die Makros, die Sie auch schon kennen, werden ersetzt – und zwar durch die Inline-Funktionen, die wir noch in diesem Kapitel behandeln.

Die Include-Anweisungen `#include` und die Compile-Schalter wie `#ifdef` behalten allerdings auch in C++ ihre Bedeutung.

19.3.6 Definition von Variablen

Während in C Variablen am Anfang eines Blocks definiert werden müssen, können sie in C++ frei eingeführt werden. Voraussetzung ist nur, dass sie vor der erstmaligen Verwendung definiert werden:

```
   int x = 0;
   x = 99;

A  int a = 0;

B  for( int i = 0; i<100; i++)
      {
```

```
    a += i;
C   }
```

Listing 19.4 Definition von Variablen in C++

In dem Beispiel wird die Variable `a` nach der Verwendung von `x` definiert (A), dies ist in C nicht möglich. Ebenso können in C++ die Schleifenvariablen im Schleifenkopf definiert werden, wie hier die Variable `i` (B). Die Schleifenvariable verliert ihre Gültigkeit mit dem Ende des zugehörigen Blocks in (C) und kann danach nicht mehr verwendet werden.

Im Moment mag Ihnen diese Erweiterung der Variablendefinition wie eine (nützliche) Spielerei erscheinen. Sie werden aber im folgenden Kapitel sehen, wie in C++ bei der Definition von Variablen automatisch spezieller Code ablaufen kann. Die Stelle der Definition bestimmt den Zeitpunkt der Codeausführung und gewinnt damit extrem an Bedeutung.

19.3.7 Verwendung von Referenzen

In C erfolgt die Übergabe von Parametern an eine Funktion immer als Kopie. Eine Änderung dieser Kopie innerhalb der Funktion ist für den Aufrufer ohne Wirkung. Sollen Werte einer Variablen des Hauptprogramms innerhalb einer Funktion geändert werden, muss deren Adresse übergeben werden. Die Funktion erhält dann einen Zeiger auf den entsprechenden Wert. Über den Zeiger wird der Wert dann aus der Funktion heraus manipuliert.

Ich zeige Ihnen die Vorgehensweise in C noch einmal anhand einer `swap`-Funktion zum Vertauschen zweier Variablenwerte:

```
A   void swap( int* a, int* b)
       {
       int tmp;

B      tmp = *a;
C      *a = *b;
D      *b = tmp;
       }

    int main()
       {
       int x, y;
       ...
```



```
E      swap ( &x, &y);
       return 0;
       }
```

Listing 19.5 Tauschen von Variablen in C

In der Schnittstelle der Funktion erfolgt die Übergabe der Parameter als Zeiger auf `int` (A). Der eigentliche Tausch der Werte erfolgt mithilfe der Hilfsvariablen `tmp` (B) und der dereferenzierten Zeiger (B–D). Bei Aufruf der Funktion werden die Adressen der zu tauschenden Variablen übergeben (E).

Innerhalb der Funktion werden die Adressen nur dereferenziert verwendet, um an die dahinterliegenden Werte zu gelangen. An den eigentlichen Adressen sind wir ja gar nicht interessiert.

C++ bietet Ihnen hier eine elegante und effiziente Alternative: die sogenannten *Referenzen*. Über Referenzen können Funktionen übergebene Variablen ändern!

Referenzen verhalten sich wie ein konstanter Zeiger, der bei jeder Verwendung automatisch referenziert wird. Eine Referenz ist ein L-Value. Sie kann also auf der rechten und auf der linken Seite einer Zuweisung verwendet werden.

Um an der Schnittstelle einer Funktion eine Referenz zu übergeben, wird bei der Vereinbarung der Parameter dem Datentyp ein `&` hintenangestellt. Gelesen wird dies als: »x« vom Typ Referenz Datentyp.

Die Schnittstelle der Funktion zum Tauschen der Werte sieht mit Referenzen so aus:

```
void swap( int& a, int& b)
```

Die Funktion `swap` liefert weiter keinen Rückgabewert und erhält jetzt aber zwei Parameter, `a` und `b`, vom Typ Referenz auf `int`.

Innerhalb der Funktion werden die Variablen dann wie »normale« `int`-Werte verwendet. Die gesamte Funktion zum Tauschen der Werte und ihr Aufruf sehen damit so aus:

```
A   void swap( int& a, int& b)
       {
       int tmp;

B      tmp = a;
C      a = b;
D      b = tmp;
       }
```

```
int main()
   {
   int x = 1, y = 2;
   //   ...
   printf( "Vorher; %d %d\n", x, y);

E  swap( x, y);

   printf( "Nachher; %d %d\n", x, y);
   }
```

Listing 19.6 Tauschen von Variablen in C++ mit Referenzen

In (A) wird die Funktion, wie bereits beschrieben, deklariert. Der Tausch der Werte über direkte Zuweisung erfolgt in (B–D). Hier sind durch die Verwendung der Referenzen keine Indirektionen mehr notwendig. Im Grunde handelt es sich bei Referenzen um Zeiger, die bei jeder Übergabe implizit dereferenziert werden. Der Aufruf der Funktion erfolgt in (E) direkt mit den Variablen ohne einen Adress-Operator. Es kommt zur erwarteten Ausgabe:

```
Vorher: 1 2
Nachher: 2 1
```

Referenzen sind nicht nur nützlich, wenn Sie übergebene Werte ändern wollen. Referenzen sind bei der Übergabe auch sehr effizient, da nur der Verweis anstelle des ganzen Elements über den Stack übergeben wird. Bei einer großen Struktur kann dies ein deutlicher Vorteil sein.

Achtung!

Bei der Übergabe eines Wertes per Zeiger sieht man an der Schnittstelle sofort, dass vom Zeiger referenzierte Werte in der Funktion geändert werden könnten.

Bei der Übergabe per Referenz ist dies für den Aufrufer nicht mehr so offensichtlich! Das ist auch der Grund, warum viele C-Programmierer die Verwendung von Referenzen mit gemischten Gefühlen betrachten. Bei der Verwendung von Zeigern muss der Aufrufer einer Funktion explizit die Adresse eines Wertes angeben und ist sich damit dessen bewusst, dass die Funktion die Werte möglicherweise ändert. Dies ist bei Referenzen nicht der Fall. Ich werde Ihnen in diesem Abschnitt aber noch eine Möglichkeit zeigen, das Problem abzumildern.

Zuerst werden wir aber noch weitere Details der Übergabe per Referenz betrachten. Dazu erstellen wir zuerst eine einfache Maximumfunktion und danach verschiedene Varianten. Die erste Variante ist wenig überraschend:


```
int max1( int a, int b )
   {
   if( a >= b )
       return a;
   else
       return b;
   }
```

Listing 19.7 Eine bereits bekannte max-Funktion

Die Funktion gibt das Maximum der beiden als Kopie übergebenen Werte als Ergebnis zurück. Auch das Ergebnis wird als Kopie an den Aufrufer übergeben.

In der zweiten Variante werden Referenzen als Parameter übergeben:

```
int max2( int& a, int& b )
   {
   if( a >= b )
       return a;
   else
       return b;
   }
```

Listing 19.8 Alternative max-Funktion mit Referenzen

In der üblichen Verwendung

```
z = max1( x, y );
z = max2( x, y );
```

zeigt sich kein Unterschied.

Wenn Sie nun aber konstante Werte an die Funktion übergeben:

```
const int u =99, v= 100;
z = max2( 99, 100 ); // Compiler-Fehler
z = max2( u, v );    // Compiler-Fehler
```

erhalten Sie für den Aufruf von `max2` eine Fehlermeldung. Der Datentyp der Konstanten kann nicht in eine Referenz auf ein `int` verwandelt werden.

In der Schnittstelle deklarierte Referenzen können und dürfen in einer Funktion verändert werden. Daher können solchen Referenzen keine konstanten Werte übergeben werden, da hier die Veränderung unmöglich wäre. Wenn Sie konstante Werte übergeben wollen, müssen Sie die Referenzen in der Schnittstelle ebenfalls als konstant deklarieren.

Wenn Sie dem Compiler damit angeben, dass die Referenzen innerhalb der Funktion nicht verändert werden, dann können Sie eine entsprechende Funktion auch mit Konstanten aufrufen:

```
int max3( const int& a, const int& b )
    {
    if( a >= b )
        return a;
    else
        return b;
    }
```

Listing 19.9 Maximumfunktion mit konstanten Referenzen

Durch die Angabe von `const int&` in der Schnittstelle werden in der Funktion konstante Referenzen verwendet. Der Compiler erstellt hier bei Bedarf Zwischenvariablen, die für den Zugriff verwendet werden, und der folgende Aufruf wird möglich:

```
z = max3( 99, 100 ); // ok
z = max3( u, v );    // ok
```

Die Verwendung einer konstanten Referenz in der Schnittstelle sorgt nicht nur dafür, dass die Funktion mit konstanten Werten aufgerufen werden kann. Sie zeigt dem Benutzer einer Funktion auch an, dass die Funktion die übergebenen Referenzen nicht verändern wird.

Funktionen, die die Übergabe per Referenz aus Performance-Gründen nutzen und die übergebenen Werte nicht verändern, sollten die Referenzen als konstant deklarieren. Dadurch können sie den Performance-Vorteil der Übergabe per Referenz gegenüber der Kopie nutzen. Anhand der Schnittstelle sieht der Benutzer aber, dass die Funktion die Werte nicht ändert.

19.3.8 Referenzen als Rückgabewerte

Sie können Referenzen auch als Rückgabewerte verwenden. Ein Beispiel für eine entsprechende Nutzung ist die folgende Variante der `max`-Funktion:

```
int& max4( int& a, int& b )
    {
    if( a >= b )
        return a;
    else
        return b;
    }
```

Listing 19.10 Maximumfunktion mit Rückgabe einer Referenz auf int

Die Funktion `max4` erhält Referenzen auf `int` und gibt eine Referenz auf `int` zurück. Damit wird die die folgende Verwendung möglich:

```
void main()
{
    int x = 1, y = 2;

A   max4( x, y ) = 4711;

    printf( "y: %d, x: %d \n", y, x );
}
```

Listing 19.11 Verwendung der Referenz als Rückgabewert (L-Value)

Hier ist auch der Rückgabewert der Funktion eine Referenz und damit ein L-Value, der auch auf der linken Seite einer Zuweisung stehen darf (A).

Aus der Funktion wird die Variable, die den größeren Wert enthält, als Referenz zurückgegeben. In diesem Fall ist das `y`. Dieser Variablen wird dann als L-Value der Wert 4711 zugewiesen.

Die Ausgabe ist damit:

```
y: 4711, x: 1
```

19.3.9 Referenzen außerhalb von Schnittstellen

Die bisherigen Beispiele legen zu Recht nahe, dass Referenzen hauptsächlich bei Funktionsschnittstellen zum Einsatz kommen. Referenzen können aber auch als Variablen in einem Programm verwendet werden. Hier werden sie oft auch als *Aliasnamen* für andere Variablen bezeichnet und müssen bei der Deklaration initialisiert werden:

```
int i;

int& ref = i;

i = 1;
printf(" %d %d\n", i, ref);
```

```
i++;
printf(" %d %d\n", i, ref);

ref++;
printf(" %d %d\n", i, ref);
```

Listing 19.12 Verwendung einer Referenz als Aliasname

Im Beispiel ist `ref` eine Referenz auf `i`, und die beiden Variablen können völlig synonym verwendet werden. Das Programm liefert die folgende Ausgabe:

```
 1 1
 2 2
 3 3
```

Dabei haben `ref` und `i` nicht nur immer den gleichen Wert, sie bezeichnen auch den gleichen Speicherbereich. Der Adress-Operator auf `ref` und `i` angewandt, liefert das gleiche Ergebnis. Damit produziert diese Zeile

```
printf(" %d \n", (&ref - &i));
```

die erwartete Ausgabe 0.

Generell müssen Sie beachten, dass die Referenz initialisiert werden muss:

```
int& ref = i;
```

Dabei handelt es sich um einen einmaligen Vorgang. Wie Sie schon gesehen haben, steht `ref` danach synonym für `i`. Spätere Wertzuweisungen

```
ref = x;
```

ändern nichts an der Initialisierung und der erfolgten Zuordnung, sondern setzen nur neue Werte.

19.4 Funktionen

Gerade im Bereich der Funktionen gibt es einige wichtige Erweiterungen in C++, die ich Ihnen noch zeigen werde, bevor wir in die objektorientierte Programmierung einsteigen.

19.4.1 Funktionsdeklarationen und Prototypen

Sie haben es im Verlauf des Buches bereits als guten Programmierstil kennengelernt, für alle Funktionen auch Funktionsprototypen zur Verfügung zu stellen. In C++ ist es erforderlich, für jede Funktion, die gerufen wird, bevor sie definiert worden ist, einen Funktionsprototyp bereitzustellen. Dies gibt dem Compiler die Möglichkeit, auch über Modulgrenzen hinweg eine konsequente Typüberprüfung vorzunehmen. Wenn Sie sich bereits an diesen guten Programmierstil gehalten haben, gibt es für Sie keine Änderung, ansonsten ist hier die richtige Gelegenheit, damit zu beginnen.

Wie andere Warnungen und Fehlermeldungen des Compilers sollten Sie diese Anforderungen nicht als Behinderung bei der Arbeit begreifen. Stattdessen sollten Sie die Warnung des Compilers als Hilfe auffassen, guten Code zu generieren. Der Compiler gibt Ihnen eine Unterstützung bei der Fehlervermeidung, indem er Sie zwingt, Ihre Absichten möglichst präzise zu formulieren. Dann kann er Sie auch frühzeitig darauf hinweisen, wenn es Abweichungen gibt.

Für das Hauptprogramm `main` sind im C++-Standard zwei Varianten vorgesehen. Für ein Hauptprogramm, das eine unbekannte Anzahl von Parametern erhält, typischerweise von der Kommandozeile:

```
int main( int argc, char** argv )
  {
  //...
  return 0;
  }
```

Alternativ der parameterlose Aufruf:

```
int main()
  {
  ...
  return 0;
  }
```

Die `return`-Anweisung in der `main`-Funktion darf in C++ weggelassen werden. Der Compiler ergänzt dann automatisch ein:

```
return 0;
```

Die allgemeine Regel, dass Funktionen mit einem Rückgabetyp auch einen Wert zurückgeben müssen, gilt weiter[1].

1 In den folgenden Kapiteln haben einige `main`-Funktionen nur eine Handvoll Zeilen. Dort habe ich die `return`-Anweisung weggelassen, um das Beispiel in den Vordergrund zu rücken. Generell bin ich aber dafür, sie zu verwenden.

19.4.2 Vorgegebene Werte in der Funktionsschnittstelle (Default-Werte)

Häufig haben Sie es in der Programmierung mit Funktionen zu tun, bei denen Sie nicht immer alle Parameter benötigen. Wir sehen uns das am Beispiel einer Funktion zum Hochzählen (und Ausgeben) von Werten an:

```
void hochzaehlen( int start, int ende, int inkrement)
    {
    int zaehler = start;
    while ( zaehler <= ende)
        {
        printf("%d\n", zaehler);
        zaehler += inkrement;
        }
    printf("\n");
    }
```

Die Funktion wird z. B. mit den folgenden Parametern aufgerufen:

```
hochzaehlen ( 0, 10, 2);
```

und liefert das erwartete Ergebnis:

```
0
2
4
6
8
10
```

Im Laufe der weiteren Programmierung und Verwendung der Funktion stellen Sie vielleicht fest, dass die Funktion überwiegend mit einem Inkrement von 1 eingesetzt wird. Bei jedem Aufruf müssen aber alle Parameter mit angegeben werden. Sie würden Ihre Funktion hier gerne um eine sinnvolle Vorgabe ergänzen.

C++ bietet mit den sogenannten *Default-Werten* eine solche Möglichkeit. Dies sind vorgegebene Argumentwerte, die an einer Funktionsschnittstelle verwendet werden, wenn vom rufenden Programm keine Werte übergeben wurden. Das rufende Programm kann also bestimmte Werte im Aufruf einfach auslassen, die fehlenden Werte werden in der Funktion ersetzt. Die Default-Werte werden auch *Standardwerte* genannt.

In C++ erhalten wir diese Vorgaben, indem die gewünschten Default-Werte wie eine Zuweisung oder Initialisierung an den entsprechenden Parameter angefügt werden (hier die 1 für `inkrement`):

```
void hochzaehlen( int start, int ende , int inkrement = 1 )
```

Jetzt können wir die Funktion mit zwei oder drei Parametern verwenden:

```
hochzaehlen( 0, 6, 2);
hochzaehlen( 0, 3);
```

und erhalten folgendes Ergebnis:

```
0
2
4
6

0
1
2
3
```

Bei Aufruf ohne den dritten Parameter wird automatisch der Standardwert verwendet.

Wenn auch der Endwert unseres Inkrements `ende` einen typischen Wert hat, kann natürlich auch hier ein entsprechender Default-Wert festgelegt werden:

```
void hochzaehlen( int start, int ende = 5 , int inkrement = 1 )
```

Jetzt könnte die Funktion auch mit einem Parameter aufgerufen werden. Sogar für den Startwert `start` können wir einen Default-Wert festlegen, sodass die Funktion dann so aussieht

```
void hochzaehlen( int start = 0, int ende = 5 , int inkrement = 1 )
```

und ganz ohne Parameter aufgerufen werden kann:

```
hochzaehlen();
```

Für die Default-Werte gibt es einige naheliegende Einschränkungen:

- Default-Werte können immer nur für die »letzten« Argumente einer Funktion (d. h. ab einer bestimmten Position, dann aber für alle folgenden Argumente) angegeben werden.

- Beim Aufruf einer Funktion mit Default-Argumenten können immer nur Argumente vom Ende der Parameterliste weggelassen werden. Benötigen Sie einen bestimmten Parameter, müssen alle davorliegenden Parameter mit angegeben werden.

Hat eine Funktion mit Default-Argumenten einen Funktionsprototyp, gehört die Festlegung der Standardwerte dorthinein:

```
void hochzaehlen( int start = 0, int ende = 5 , int inkrement = 1);
```

So werden alle Nutzer der Funktion über die Schnittstelle informiert. Für Bibliotheken kennt der Benutzer meist nur die Header-Dateien und nicht den Quellcode. Durch die Default-Werte im Prototyp ist auch von außen sichtbar, welche Standardwerte angewandt werden. Naheliegenderweise dürfen die Defaults dann bei der Implementierung der Funktion nicht erneut definiert werden, da sonst Standardwerte an zwei Stellen festgelegt würden.

```
void  hochzaehlen( int start, int ende, int inkrement)
    {
    int zaehler = start;
    //...
    }
```

19.4.3 Inline-Funktionen

Insbesondere bei der objektorientierten Programmierung entstehen häufig sehr kleine Funktionen, die einen Aufruf als Funktion eigentlich »nicht lohnen«. In C werden solche Funktionen vielfach als Präprozessor-Makros realisiert, um zu verhindern, dass Parameter aufwendig über den Stack übergeben werden müssen. Die Risiken, die dabei auch existieren, haben Sie in Kapitel 9 zur Programmgrobstruktur schon kennengelernt.

Für solche Aufgaben bietet C++ als Lösung *Inline-Funktionen* an. Inline-Funktionen haben die Effizienz von Makros, verbunden mit der Konsistenz und Schnittstellensicherheit von Funktionen.

Um eine Funktion zu einer Inline-Funktion zu machen, stellen Sie der Funktionsdefinition das Schlüsselwort `inline` voran:

```
A  inline int max( int a, int b)
      {
      return a > b ? a : b;
      }
```



```
void main()
   {
   int x = 10, y = 100

   int m = max( x, y);
   }
```

Listing 19.13 Verwendung von Inline-Funktionen

Überall dort, wo die Inline-Funktion verwendet wird, ersetzt der Compiler den Funktionsaufruf durch den entsprechenden Code der Funktion. Die Übergabe der Parameter über den Stack entfällt. Aus dem oben dargestellten Beispiel wird praktisch der folgende Quelltext:

```
void main()
   {
   int x = 10, y = 100

   int m = x > y ? x : y;
   }
```

Listing 19.14 Praktisch entstandener Code nach Ersetzung der Inline-Funktion

Die Inline-Anweisung kann vom Compiler nicht immer ausgeführt werden, etwa dann, wenn es sich um eine rekursive Funktion handelt. Sie ist daher als eine Empfehlung an den Compiler zu verstehen.

Sinnvoll ist das Inlining besonders für kleine Funktionen, die sehr oft gerufen werden und bei denen der Aufwand der Parameterübergabe über den Stack im Verhältnis zum Ausführungsaufwand hoch ist. Bei kleinen und einfachen Funktionen bedeutet Inlining oft sowohl kompakteren Programmcode als auch schnellere Ausführung und ist auf jeden Fall zu empfehlen[2].

Bei größeren Funktionen kann Inlining zu umfangreicherem Code führen, ergibt aber trotzdem oft eine bessere Performance.

Da der Compiler den Code der Inlining-Funktion an der Stelle ihres Aufrufs einsetzt, muss die Definition einer solchen Funktion bereits vorliegen, wenn sie verwendet wird. Der Prototyp reicht hier nicht aus. Falls eine Inline-Funktion in mehreren Modulen benutzt werden soll, muss ihre Definition daher in einer Header-Datei

2 Inlining kann zu erheblichen Performance-Gewinnen führen, z. B. bei einer Funktion, die in einer Schleife oft gerufen wird. Sie können z. B. Funktionen in den Sortieralgorithmen als Inline-Funktionen deklarieren und damit erhebliche Performance-Gewinne erzielen – etwa bei den Funktionen `insertion_h_sort` oder `adjustheap` in Kapitel 13 zum Sortieren.

abgelegt sein, die dann von den Modulen inkludiert wird, die die Funktion verwenden[3].

Es gibt noch eine weitere Methode, um Inline-Funktionen zu erstellen. Diese stelle ich Ihnen im folgenden Kapitel im Rahmen der Objektorientierung vor.

19.4.4 Überladen von Funktionen

Die wichtigste nicht-objektorientierte Funktion von C++ ist das *Überladen* von Funktionen. Eine typische Situation *ohne* Überladung zeige ich Ihnen hier noch einmal. Ich werde dabei von folgender Datenstruktur ausgehen:

```
struct punkt
   {
   int x;
   int y;
   };

struct vektor
   {
   int x;
   int y;
   int z;
  };
```

Typische Funktionen für deren Ausgabe sind in der Programmiersprache C dann die folgenden:

```
A  void print_punkt( punkt p)
     {
      printf( "Punkt: (%d,  %d)\n", p.x, p.y);
     }
B  void print_vektor( vektor v)
     {
      printf( "Vektor: (%d, %d, %d)\n", v.x, v.y, v.z);
     }

   void main ()
      {
      struct punkt p = {5, 4};
      struct vektor v = {1, 2, 3};
C     print_punkt( p);
D     print_vektor( v);
      }
```

Listing 19.15 Ausgabe unterschiedlicher Strukturen

In (A) und (B) erfolgt die Definition einer Ausgabefunktion je Datentyp mit unterschiedlichen Namen, da gleichnamige Funktionen zum Fehler führen würden. Die Ausgabe erfolgt dann unter Verwendung der passenden Funktionen zum jeweiligen Datentyp in (C) und (D).

In C++ wird die Verwendung der Datentypen durch das Überladen von Funktionen deutlich vereinfacht. Überladen bedeutet dabei, dass Funktionen nicht nur anhand ihres Namens, sondern auch anhand ihrer Parametersignatur unterschieden werden. Damit kann es in C++ verschiedene Funktionen gleichen Namens geben, sofern sich die Art und/oder Anzahl der Parameter unterscheiden.

Mit diesem Wissen erstellen wir nun zwei gleichnamige Ausgabefunktionen mit unterschiedlicher Schnittstelle, die wir gleich verwenden:

```
A  void print( punkt p)
     {
      printf( "Punkt: (%d, %d)\n", p.x, p.y);
     }
B  void print( vektor v)
     {
      printf( "Vektor: (%d, %d, %d)\n", v.x, v.y, v.z);
     }

   void main ()
      {
      punkt p = {5, 4};
      vektor v = {1, 2, 3};
C     print( p);
D     print( v);
      }
```

Listing 19.16 Ausgabe mit überladenen Funktionen

In (A) definieren wir die Ausgabe für den Datentyp `punkt`, in (B) für den `vektor`. Beide Funktionen haben den Namen `print`.

3 Dies widerspricht nicht der Anforderung, dass in einer Header-Datei kein Code stehen soll. Wie bei einer Datenstruktur steht in einer Header-Datei nur eine formale Definition, die erst dann Code wird, wenn sie in einem Programm verwendet wird.

In der Programmiersprache C würde der Compiler diesen Namenskonflikt nicht akzeptieren. In C++ werden die Funktionen jedoch anhand ihrer verschiedenen Aufrufparameter differenziert.

In (C) wird die Funktion `print` für einen `punkt` gerufen, in (D) die Variante für einen `vektor`.

Die passenden Funktionen werden vom Compiler anhand der Signatur ausgewählt, und wir erhalten die erwartete Ausgabe:

```
Punkt (5, 4)
Vektor (1, 2, 3)
```

[!] Verschiedene Funktionen gleichen Namens stellen in C++ kein Problem dar, sofern sie anhand ihrer *Parametersignatur* unterschieden werden können.

19.4.5 Parametersignatur von Funktionen

Zur Unterscheidung und Auswahl (überladener) Funktionen dient neben dem Funktionsnamen die Parametersignatur. Zu der Signatur gehören Anzahl, Reihenfolge und Typ der übergebenen Parameter.

[!] Der Typ des Rückgabewertes geht nicht mit in die Parametersignatur ein. Ebenso werden Default-Parameter nicht berücksichtigt.

Bei den folgenden Beispielen unterscheiden sich weder der Name noch die Parametersignatur der Funktionen:

```
int fkt1( int a) {return 0;}
void fkt1( int a) {return;}

int  main()
   {
   fkt1( 1);
   }
```

Das Programm kann nicht übersetzt werden, da die Funktionen `fkt1` für den Compiler nicht unterscheidbar sind. Eine Funktion kann generell ohne Information zum erwarteten Rückgabetyp aufgerufen werden, daher kann der Rückgabetyp kein Unterscheidungskriterium sein. Ebenso wie der Rückgabetyp gehen Default-Werte nicht mit in die Signatur ein:


```
int fkt2( int a) {return a;}
int fkt2( int a, int b=10) {return a + b;}

int  main()
   {
   fkt2( 1);
   }
```

Auch dieses Beispiel kann nicht übersetzt werden. Aus der Sicht des Compilers sind die beiden Varianten von `fkt2` nicht unterscheidbar.

19.4.6 Zuordnung der Parametersignaturen und der passenden Funktion

Um die Zuordnung der überladenen Funktionen zu ermöglichen, benutzt der Compiler ein einfaches Verfahren. Er verändert die Funktionsnamen intern so, dass Typ und Anzahl der Parameter mit in den Namen der Funktion eingehen. Diese Modifikation wird auch als das Dekorieren von Namen oder auch *Function Name Encoding* bezeichnet. Am folgenden Beispiel können Sie sehen, wie die Namensdekoration konkret abläuft. Wir nehmen die Funktion mit dem Namen `xxx` als Ausgangspunkt:

```
void xxx( int a, char b, float c);
```

Der vom Compiler generierte Funktionsname im erzeugten Objektcode lautet:

```
xxx_Ficf
```

und beinhaltet die Parameter `int`, `char` und `float`. Über diesen Namen greift der Linker dann auf die Funktion im Objektcode zu. Sie müssen den Prozess nicht im Detail verstehen. Sie müssen nur wissen, dass es ihn gibt und dass er als Teil des Sprachstandards normiert ist. Die Normierung ist notwendig, denn nur so kann die Interoperabilität der verschiedenen C++-Compiler sichergestellt werden. So ist es möglich, dass Sie nicht nur die Bibliotheken verwenden können, die Ihr Compiler mitbringt, sondern auch bereits übersetzte Bibliotheken aus anderen Quellen nutzen können.

Im gesamten Prozess der Funktionsauswahl und Namensgenerierung ist insbesondere die Auswahl der Funktionen nicht immer leicht nachzuvollziehen. Zuerst wird natürlich nach Funktionen gesucht, die eine exakt passende Signatur besitzen. Aber durch die verschiedenen Möglichkeiten der Konvertierung wird die Auswahl dann trickreich. So kann eine Funktion, die einen Parameter vom Typ `float` erwartetet, auch durchaus mit einem `int` aufgerufen werden, eine Konvertierung von `int` nach `float` ist ja verlustfrei möglich. Wir werden das Thema daher nicht näher betrachten und lassen es bei dieser kurzen Übersicht bewenden.

19.4.7 Verwendung von C-Funktionen in C++-Programmen

Wenn Sie C-Funktionen in ein C++-Programm einbinden möchten, wird der gerade besprochene Weg zur Namensermittlung für überladene Funktionen zum Problem.

Wie Sie gesehen haben, führt der Aufruf der Funktion

```
void xxx( int a, char b, float c);
```

dazu, dass der Linker eine Funktion mit dem Namen `xxx_Ficf` sucht.

Wenn diese Funktion mit einem C-Compiler übersetzt worden ist, wird es diese Funktion nicht geben, da der C-Compiler die Regeln der C++-Namensgenerierung natürlich nicht kennt und nicht anwendet.

Es muss also eine Möglichkeit geben, innerhalb von C++ für eine bestimmte Funktion, wie hier die Funktion `xxx`, das *Function Name Encoding* abzuschalten. Dazu deklarieren wir die Funktionen als `extern "C"`. Das kann für eine einzelne Funktion passieren:

```
extern "C"  void abc(int, char b, float c);
```

Es kann aber auch ein ganzer Block als `extern "C"` ausgezeichnet werden:

```
extern "C"
   {
   void abc(int, char b, float c);
   int aaa();
   }
```

Mit dieser zusätzlichen Auszeichnung gibt es allerdings ein neues Problem, wenn sie in einer Header-Datei verwendet wird, die parallel in C- und C++-Programme inkludiert werden soll. Innerhalb eines C-Compilers ist die Anweisung `extern "C"` unbekannt und soll es auch sein. Ein C-Compiler soll unabhängig von den C++-Standards sein. Genau genommen, soll er nicht einmal wissen müssen, dass C++ existiert. Binden wir daher einen solchen Header in ein C-Programm ein, bedeutet dies dort einen Fehler.

Wir müssen daher verhindern, dass die entsprechende Zeile für den C-Compiler sichtbar wird. Es gibt innerhalb der C++-Compiler die vorbelegte symbolische Konstante `__cplusplus`. Wenn diese Konstante definiert ist, kann davon ausgegangen werden, dass gerade ein C++-Compiler am Werk ist und die zusätzlichen `extern "C"`-Anweisungen benötigt werden. Mit diesem Wissen können wir eine entsprechende Header-Datei nun für die Bearbeitung durch den Präprozessor folgendermaßen gestalten:


```
#ifdef __cplusplus
extern "C"
   {
#endif

   void abc(int, char b, float c);
   int aaa();

#ifdef __cplusplus
   }
#endif
```

Dabei verlassen wir uns darauf, dass die symbolische Konstante in C nicht existiert. Aufgrund des nicht definierten `__cplusplus` sieht der C-Compiler nach dem Durchlauf des Präprozessors nur noch diesen Code:

```
void abc(int, char b, float c);
int aaa();
```

Für den C++-Compiler sind die `extern "C"`-Anweisungen aber weiterhin sichtbar:

```
extern "C"
   {
   void abc(int, char b, float c);
   int aaa();
   }
```

Entsprechend ausgestattet, können wir Header-Dateien erstellen, die sowohl von C als auch von C++ verwendet werden können. Dies geht zwar auf Kosten der Lesbarkeit, hat aber dennoch große Vorteile. Wenn Sie sich die Header-Dateien von Bibliotheken – auch die Ihres Compilers – ansehen, werden Sie entsprechende Konstrukte (und weitere dieser Art) finden.

19.5 Operatoren

Auch bei den Operatoren gibt es Erweiterungen und Ergänzungen. Insbesondere werden in C++ einige ganz neue Operatoren eingeführt:

Operator	Bezeichnung
: :	Globalzugriff

Tabelle 19.3 Neue Operatoren in C++

Operator	Bezeichnung
::	Class-Member-Zugriff
.*	Pointer-to-Member-Zugriff (direkt)
->*	Pointer-to-Member-Zugriff (indirekt)
new	Objekt Allokator
delete	Objekt Deallokator

Tabelle 19.3 Neue Operatoren in C++ (Forts.)

Den Operator für den *Globalzugriff* werde ich Ihnen im Folgenden direkt vorstellen, die anderen Operatoren kommen erst mit der objektorientierten Programmierung zum Einsatz.

19.5.1 Der Globalzugriff

Anders als in C können in C++ auch »verdeckte« globale Elemente sichtbar gemacht werden. Dazu wird der Operator :: für den Globalzugriff verwendet. Ich zeige Ihnen im folgendem Beispiel direkt seine Funktion:

```
A   int a = 1;     // globale Variable a

    void main()
       {
B      int a = 2;   // lokale Variable a

C      printf( "lokal: %d\n", a);     // ohne Scope Resolution

D      printf( "global: %d\n", ::a);  // mit Scope Resolution
       }
```

Listing 19.17 Der Globalzugriff

Im Programm wird die globale Variable a definiert (A). In der main-Funktion überdeckt die lokale Variable a (B) die globale Variable gleichen Namens. In der Programmiersprache C wäre der globale Wert damit nicht mehr adressierbar, und es wäre nur ein Zugriff auf den lokalen Wert möglich (C). In C++ können Sie über den Globalzugriff das globale a mit der Notation ::a ansprechen (D), und es ergibt sich die folgende Ausgabe:


```
lokal: 2
global: 1
```

Über den Globalzugriff lässt sich der Konflikt an dieser Stelle entschärfen. Generell sollten Sie Namensüberschneidungen natürlich dennoch vermeiden – im Allgemeinen indem Sie die Namen der lokalen Funktion ändern, da hier die Auswirkungen leichter zu überblicken sind. Dass Sie Überschneidungen vermeiden sollten, liegt auch daran, dass Sie mit der hier gezeigten Methode nur auf globale Variablen zugreifen können. Nicht-globale Überlagerungen können auch mit dieser Methode nicht aufgelöst werden.

Der Operator :: für den Globalzugriff wird auch als *Scope-Resolution-Operator* bezeichnet und kommt auch als sogenannter *Class-Member-Operator* zum Einsatz. Diese Funktion zeige ich Ihnen bei der objektorientierten Entwicklung und der Nutzung von Klassen.

19.5.2 Alle Operatoren in C++

Die neuen Operatoren in C++ kommen zu den bereits bekannten Operatoren aus C hinzu und ordnen sich dort in das Gesamtsystem ein. Dadurch erweitert sich die Liste der Operatoren. Die vollständige Liste für C++ sieht damit folgendermaßen aus:

Zeichen	Verwendung	Bezeichnung	Klassifizierung	Ass	Prio
::	::var	Globalzugriff	Zugriffsoperator		17
::	cl::mem	Class-Member-Zugriff			
( )	f(x, y)	Funktionsaufruf	Auswertungsoperator	L	16
[ ]	a[i]	Array-Zugriff	Zugriffsoperator		
->	p->x	Indirektzugriff			
.	a.x	Strukturzugriff			
++	x++	Post-Inkrement	Zuweisungsoperator		
--	x--	Post-Dekrement			

Tabelle 19.4 Liste aller Operatoren in C++

Zeichen	Verwendung	Bezeichnung	Klassifizierung	Ass	Prio
!	!x	logische Verneinung	logischer Operator	R	15
~	~x	bitweises Komplement	Bitoperator		
++	++x	Pre-Inkrement	Zuweisungsoperator		
- -	--x	Pre-Dekrement			
+	+x	plus x	arithmetischer Operator		
-	-x	minus x			
*	*p	Dereferenzierung	Zugriffsoperator		
&	&x	Adress-Operator			
( )	(type)	Typkonvertierung	Datentyp-Operator		
sizeof	sizeof (x)	Typspeichergröße			
new	new class	Objekt allokieren			
delete	delete a	Objekt deallokieren			
.*		Pointer-to-Member-Zugriff	Zugriffsoperator	L	14
->*		Pointer-to-Member-Zugriff			
*	x*y	Multiplikation	arithmetischer Operator	L	13
/	x/y	Division			
%	x%y	Rest bei Division			
+	x+y	Addition	arithmetischer Operator	L	12
-	x-y	Subtraktion			
<<	x<<y	Bitshift links	Bitoperator	L	11
>>	x>>y	Bitshift rechts			

Tabelle 19.4 Liste aller Operatoren in C++ (Forts.)

Zeichen	Verwendung	Bezeichnung	Klassifizierung	Ass	Prio
<	x<y	kleiner als	Vergleichsoperator	L	10
<=	x<=y	kleiner oder gleich			
>	x>y	größer als			
>=	x>=y	größer oder gleich			
= =	x==y	gleich	Vergleichsoperator	L	9
!=	x!=y	ungleich			
&	x & y	bitweises Und	Bitoperator	L	8
^	x ^ y	bitweises Entweder-Oder	Bitoperator	L	7
|	x | y	bitweises Oder	Bitoperator	L	6
&&	x && y	logisches Und	logischer Operator	L	5
||	x || y	logisches Oder	logischer Operator	L	4
? :	y ? y : z	bedingte Auswertung	Auswertungsoperator	L	3
=	x=y	Wertzuweisung	Zuweisungsoperator	R	2
+=	x+=y	Operation mit anschließender Zuweisung			
-=	x-=y				
*=	x*=y				
/=	x/=y				
%=	x%=y				
&=	x&=y				
^=	x^=y				
|=	x|=y				
<<=	x<<=y				
>>=	x>>=y				
,	x , y	sequenzielle Auswertung	Auswertungsoperator	L	1

Tabelle 19.4 Liste aller Operatoren in C++ (Forts.)

Eine Leseanleitung zu dieser Tabelle (noch in der Version für die Programmiersprache C) finden Sie in Kapitel 18, »Zusammenfassung und Ergänzung«.

Einige der Operatoren und deren Operatorzeichen liegen außerhalb des Minimalzeichensatzes, der weltweit gleich interpretiert wird und überall vollständig zur Verfügung steht. Daher wurden für einige Operatoren, die Sie bereits kennen, neue Schreibweisen hinzugefügt, die ohne diese entsprechenden Sonderzeichen auskommen.

Operator	Alternative
!	not
&&	and
&	bitand
||	or
|	bitor
^	xor
~	compl
!=	not_eq
|=	or_eq
^=	xor_eq
&=	and_eq

Tabelle 19.5 Neue Schlüsselwörter für bekannte Operatoren

So kann eine Prüfung jetzt z. B. ihr Aussehen folgendermaßen ändern:

```
if ( not ( a and b))
    {
    ...
    }

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


Auch wenn es sich um einen Standard handelt, benötigen manche Compiler zusätzliche Einstellungen oder Inkludierungen, um entsprechenden Code zu übersetzen. Sie sollten diese Varianten daher nicht aktiv verwenden, sie können Ihnen aber in fremden Programmen begegnen.

In C++ gibt es darüber hinaus noch eine weitere Änderung, die das gesamte System der Operatoren betrifft. Genau genommen, haben die meisten Operatoren gar keine feste Bedeutung mehr. Es handelt sich jetzt stattdessen eher um ein System, das die Stelligkeit (unär oder binär), die Assoziativität (links oder rechts) und die Priorität der Operatoren vorgibt. Die Funktionalität der Operatoren steht nur noch für die integralen Datentypen des Compilers fest. Für selbst definierte Datentypen hängt die konkrete Bedeutung eines Operators in C++ von den Datentypen ab, auf die er angewandt wird. Der Programmierer kann die Funktionsweise von Operatoren verändern. Was das bedeutet und wie das geht, erfahren Sie jetzt.

19.5.3 Überladen von Operatoren

In C++ können Operatoren auch als Funktion dargestellt werden. Anstelle der üblichen Schreibweise

```
a = b + c;
```

kann prinzipiell auch folgende Notation verwendet werden[4]:

```
a = operator+( b, c);
```

Damit ist es naheliegend, dass wir diese Funktion wie andere Funktionen überladen können. Das bedeutet, dass Sie die Operationen, die ein Operator ausführt, abhängig von den verwendeten Datentypen, selbst bestimmen können. Damit stehen Ihnen ganz neue Möglichkeiten offen.

Das gilt natürlich nicht nur für den +-Operator, sondern auch für fast alle anderen Operatoren wie etwa:

```
    ++ - * / % []
```

Überladen werden können alle Operatoren, außer:

- Strukturzugriff .
- Pointer-to-Member .*
- Class-Member-Zugriff : :

4 Beachten Sie, dass die explizite Verwendung dieser Darstellung für nicht selbst definierte Datentypen untersagt ist. Das soll uns aber nicht weiter stören, da der besondere Wert dieser Technik gerade bei eigenen Datentypen zum Tragen kommt, wie ich Ihnen gleich zeigen werde.

- bedingte Auswertung ? :
- Typspeichergröße sizeof

Ich werde Ihnen die Möglichkeiten dieser Überladung wieder mit der Datenstruktur punkt demonstrieren[5].

```
struct punkt
   {
   int x;
   int y;
   };
```

Für diese Struktur wollen wir eine Addition einführen. Aus der Mathematik kennen Sie die Addition von Vektoren. Die Summe zweier Punkte ist dort definiert als

(x, y) + (u, v) = (x + u, y + v)

Der neue einzuführende Operator + für Punkte ist eine Funktion, die zwei Punkte p1 und p2 als Übergabeparameter erhält und einen Punkt als Ergebnis zurückliefert. Die Schnittstelle der zugehörigen Operatorfunktion ergibt sich damit zu:

```
punkt operator+( punkt p1, punkt p2)
```

Die Implementierung der Funktion können Sie leicht vornehmen:

```
A  punkt operator+( punkt p1, punkt p2)
      {
B     punkt ergebnis;
C     ergebnis.x = p1.x + p2.x;
D     ergebnis.y = p1.y + p2.y;
E     return ergebnis;
      }
```

Listing 19.18 Überladener operator+ für zwei Punkte

Die Funktion erhält die beiden Punkte p1 und p2 als Argumente (A). Innerhalb der Funktion wird eine zusätzlich Variable vom Typ punkt für das Ergebnis erzeugt (B), und die eigentliche Addition wird durchgeführt (C und D). Als Resultat wird der neue Punkt zurückgegeben (E).

Die Addition von Punkten in einem Programm wird damit sehr einfach und gut lesbar dargestellt:

5 Das Überladen der eingebauten Operatoren (z. B. zur Addition von int-Werten) ist nicht möglich.


```
   void main()
      {
      punkt x = {1, 2};
      punkt y = {3, 4};
      punkt u,v ;

A     u = x + y;
      print( u);
B     v = operator+( y, y);
      print( v);
      }
```

Listing 19.19 Verwendung des überladenen Operators

In dem Programm werden beide möglichen Schreibweisen verwendet. In (A) wird die Addition über die Operatorschreibweise aufgerufen, in (B) wird die Funktionsschreibweise verwendet.

Die Ausgabe der Ergebnisse erfolgt über die passende print-Funktion aus dem vorangegangenen Abschnitt, die für den Datentyp punkt überladen worden ist:

```
Punkt: (4, 6)
Punkt: (6, 8)
```

In diesem Sinne können wir auch weitere Operatoren anpassen und z. B. einen Operator für die skalare Multiplikation erstellen. Hier soll der Operator einen int-Wert als Faktor sowie eine Struktur punkt erwarten und einen punkt zurückliefern. Die Berechnungsvorschrift dafür lautet:

$a \cdot (x, y) = (a \cdot x, a \cdot y)$

```
punkt operator*( int faktor, punkt p)
   {
   punkt ergebnis;
   ergebnis.x = faktor * p.x;
   ergebnis.y = faktor * p.y;
   return ergebnis;
   }
```

Listing 19.20 Definition eines weiteren überladenen Operators

Zur Nutzung dieses Operators erweitern wir unser Hauptprogramm leicht

```
   void main()
      {
      punkt x = { 1, 2 };
      punkt y = { 3, 4 };
      punkt u, v;

      u = x + y;
      print ( u );
      v = operator+( y, y );
      print( v );
A     print( operator*( 3, v ));
      }
```

Listing 19.21 Verwendung des zusätzlichen neuen Operators

und geben in (A) das Ergebnis der Skalarmuliplikation ohne Erstellung einer Zwischenvariablen mit der überladenen `print`-Funktion aus:

```
Punkt: (4, 6)
Punkt: (6, 8)
Punkt: (18, 24)
```

Mit überladenen Operatoren lassen sich viele Operationen auf Datenstrukturen gut erfassbar darstellen. Die umfassende Implementierung eines kompletten und konsistenten Operatorenmodells ist allerdings oft aufwendig.

Im Beispiel oben sollte etwa auch die folgende Skalarmultiplikation explizit implementiert werden:

```
punkt operator*( punkt p, int faktor)
   {
   return faktor * p;
   }
```

Wie Sie sehen, kann hier aber schon der bestehende Operator zur Implementierung verwendet werden. Bei der Verwendung überladener Operatoren muss darauf geachtet werden, dass die bereitgestellten Operatoren stimmig und intuitiv sind. Zum Beispiel sollte beim Überladen des `operator==` typischerweise auch `operator!=` überladen werden etc. Sind die Operatoren nicht entsprechend angelegt, kann besser eine normale Funktion verwendet werden, die den Benutzer nicht in die Irre führt.

Durch die leichtere Verwendung und Lesbarkeit des entstehenden Codes lohnt sich der Aufwand zur Implementierung der Operatoren aber durchaus. In den Beispielen des nächsten Abschnitts werde ich Ihnen die Umsetzung eines entsprechenden Operatormodells demonstrieren.

19.6 Auflösung von Namenskonflikten

Gerade bei großen Projekten kann es leicht vorkommen, dass Funktionen in unterschiedlichen Teilen des Projekts die gleichen Namen tragen.

Dies kommt umso öfter vor, je größer die Programme werden, und je mehr Bibliotheken genutzt werden – seien es fremde oder selbst erstellte. So ist z. B. der Funktionsname `print` zur Ausgabe wenig originell, taucht aber gerade daher häufig auf. Das Gleiche gilt für bestimmte Variablennamen.

In eigenen Programmen lassen sich solche Konflikte durch die Einführung von Namenkonventionen meist vermeiden. Dazu erhalten z. B. alle eigenen Funktionen ein bestimmtes Präfix.

Wenn verschiedene Teile eines Programms unabhängig voneinander entwickelt werden, ist es oft aber gar nicht möglich, entsprechende Konventionen zu vereinbaren oder durchzusetzen.

Das hier diskutierte Problem und eine möglich Lösung gibt es auch im realen Leben – und zwar mit Straßennamen. Innerhalb einer Stadt kann die Verwaltung dafür sorgen, dass alle Straßennamen innerhalb der Stadt einmalig sind. Werden aber zwei Städte zusammengelegt, kann es natürlich vorkommen, dass es plötzlich zwei Bahnhofstraßen oder Marktstraßen gibt. Wenn nun keine der Straßen umbenannt werden soll, muss für die Straßen eine anderweitige Unterscheidung festgelegt werden, ein sogenannter *Namensraum*, z. B. in Form von Stadtteilen. Wenn wir in der Stadt einen Stadtteil A und einen Stadtteil B mit jeweils einer Bahnhofstraße haben, wird man für die genaue Bezeichnung den Stadtteilnamen mit angeben, also »Bahnhofstraße in Stadtteil B«. Alternativ kann die Verwaltung einen neuen Stadtteil benennen. Einen ähnlichen Mechanismus gibt es mit den sogenannten *Namensräumen* auch in C++. Die Namensräume übernehmen hier die Funktion von Stadtteilen. In C++ bevorzugt man dazu allerdings eine etwas kompaktere Notation in der Art von

```
Stadtteil_B::Bahnhofstrasse
```

um die Bahnhofstraße in Stadtteil B anzusprechen. Auch hier kommt wieder der Scope-Resolution-Operator zum Einsatz.

Ich werde jetzt einen Fall von Namensüberschneidungen in C++ künstlich erzeugen und einige Dateien erstellen, die uns sicher einen Namenskonflikt einbringen werden. An diesem Beispiel werde ich Ihnen dann zeigen, wie Sie den Konflikt mit Namensräumen auflösen können.

Unsere Dateien für diesen Fall sehen folgendermaßen aus:

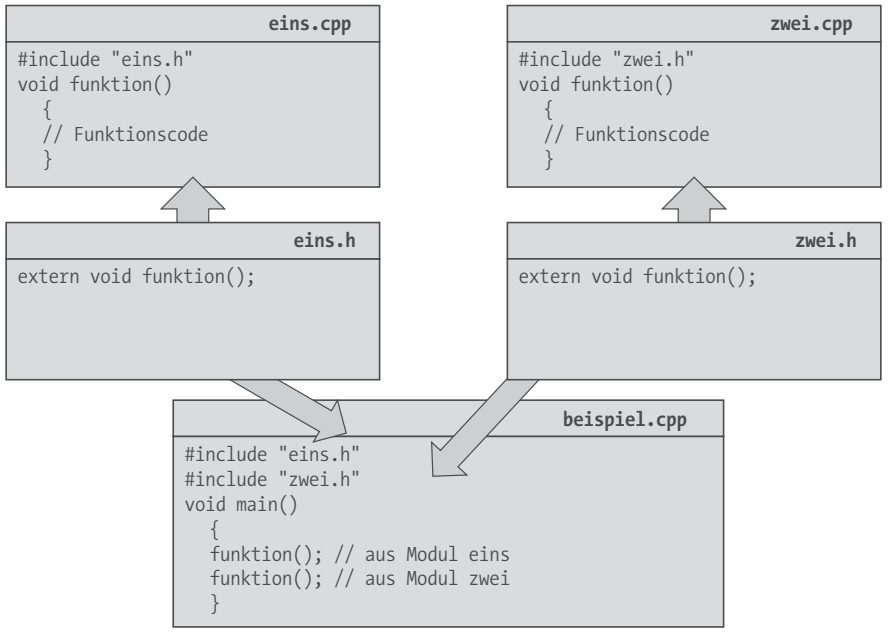


Abbildung 19.1 Namenskonflikte durch gleiche Funktionsnamen

Wegen des Namenskonflikts der mehrfach vorkommenden Funktion mit dem Namen `funktion` kann das Beispiel erwartungsgemäß nicht übersetzt werden. Im Folgenden werde ich diesen Konflikt durch die Verwendung von Namensräumen auflösen.

Erstellen von Namensräumen

Dazu werde ich für jedes Modul einen eigenen Namensraum schaffen. Ein Namensraum wird mit dem Schlüsselwort `namespace` neu eingerichtet:

```
namespace eins
    {
    }
```

Der Name des Namensraums ist im Rahmen der üblichen Namensregeln frei wählbar. Insbesondere muss er nicht dem Namen der Header-Datei entsprechen. Die geschweiften Klammern, die den Block nach dem Schlüsselwort markieren, enthalten all das, was in den erzeugten Namensraum gehören soll. In unserem einfachen Beispiel ist das jeweils die einzelne Funktion:


```
namespace eins
    {
    extern void funktion();
    }
```

Damit ist `funktion` aus der Datei `eins.cpp` im Namensraum `eins` platziert. Das bedeutet, dass sie zukünftig mit ihrem »vollen« Namen angesprochen werden muss. Die Notation dafür lautet:

```
eins::funktion()
```

Der Funktionsname in dieser Notation wird auch als *voll qualifiziert* bezeichnet.

Konsequenterweise werde ich jetzt auch das zweite Modul mit einem entsprechenden Namespace versehen. Das Gesamtergebnis sieht dann so aus:

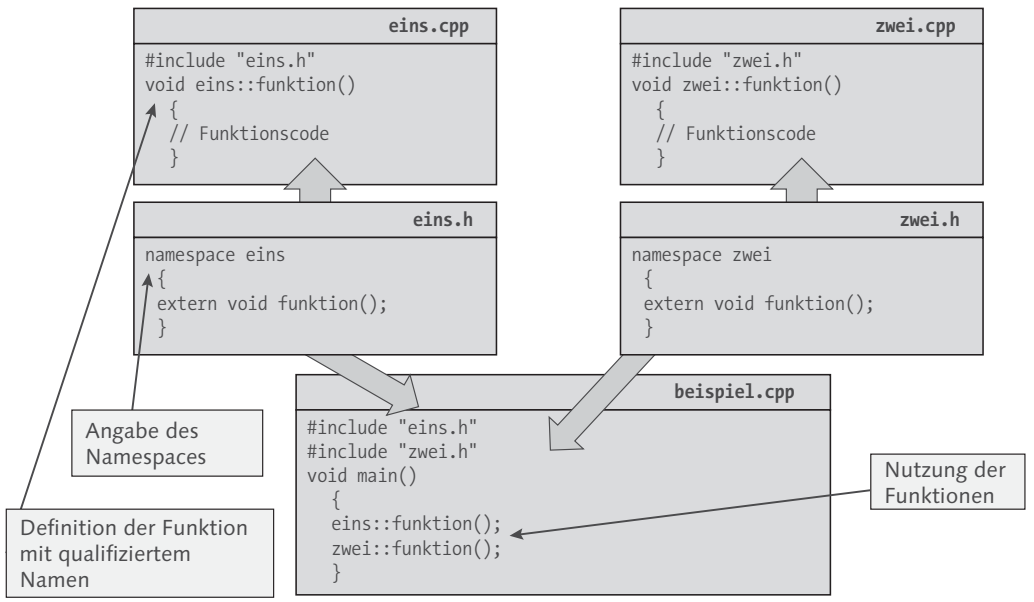


Abbildung 19.2 Auflösung der Namenskonflikte durch Namespaces

Mit der Angabe der voll qualifizierten Namen im Hauptprogramm konnte ich das gesamte Beispiel damit übersetzungsfähig machen.

In C++ können Funktionen in eigenen Namensräumen deklariert werden. Definition und Verwendung der Funktionen erfolgen dann unter Angabe des vollständigen Namens (voll qualifiziert). [!]

Ansprechen von Funktionen in Namensräumen

Generell können Sie mit dem oben geschilderten Vorgehen bereits arbeiten. Die Angabe eines voll qualifizierten Namens wird bei langen Namen aber schnell unübersichtlich. Um sich hier die Arbeit zu erleichtern, verwenden Sie die `using`-Anweisung. Mit ihrer Hilfe kann eine Funktion mit ihrem voll qualifiziertem Namen eingebunden werden und danach mit ihrem »kurzen« Namen genutzt werden:

```
   #include "eins.h"
   #include "zwei.h"

A  using eins::funktion;

   void main()
       {
B      funktion();

C      zwei::funktion();
       }
```

Listing 19.22 Ansprechen von Namensräumen

In dem Beispiel wird `eins::funktion` mit `using` als die präferierte Version von `funktion` eingeführt (A). Damit ist es nun im Weiteren möglich, `funktion` aus dem Namensraum `eins` auch ohne weitere explizite Angabe des Namensraums (B) aufzurufen. Der explizite Aufruf von `zwei::funktion` ist dabei weiter möglich (C).

In einer Datei können mehrere `using`-Anweisungen angegeben werden, auf diese Weise kann ein aufgelöster Namenskonflikt allerdings auch wieder eingeführt werden.

Importieren eines kompletten Namensraums

Die Angabe einzelner `using`-Anweisungen ist zwar eine Verbesserung, aber weiterhin aufwendig und kleinteilig. Über die `using namespace`-Anweisung ist es daher nicht nur möglich, einzelne Funktionen, sondern einen vollständigen Namensraum als Ganzes zu importieren:

```
A  namespace allgemein
       {
       void f1();
       void f2();
       }
```



```
B  namespace speziell
       {
       void f1();
       }

   void main()
       {
C      using speziell::f1;
D      using namespace allgemein;

E      f1(); // speziell::f1
F      f2(); // allgemein::f2
       }
```

Listing 19.23 Import vollständiger Namensräume

In dem Codefragment werden die beiden Namensräume `allgemein` (A) und `speziell` (B) erzeugt. Beide Namensräume enthalten die Funktion `f1` und bergen damit einen potenziellen Namenskonflikt. Über das bereits bekannte `using` wird nun die Funktion `speziell::f1` importiert (C). Anschließend wird mit `using namespace allgemein` der vollständige Namensraum `allgemein` eingebunden.

Der Aufruf von `f1` in (E) richtet sich an die Funktion `speziell::f1`, da die spezifischere Einbindung aus (C) die höhere Priorität besitzt. Der Aufruf von `f2` richtet sich durch die Einbindung des gesamten Namensraums `allgemein` an `allgemein::f2`.

Es gibt noch einige weitere Möglichkeiten und Feinheiten bei der Definition und Nutzung von Namensräumen wie etwa anonyme und geschachtelte Namensräume, diese wollen wir hier aber nicht weiter behandeln.

19.6.1 Der Standardnamensraum std

C++ verwendet die Definition von Namensräumen auch für die Standardbibliothek. Die C-Laufzeitbibliotheken stehen Ihnen aber weiterhin zur Verfügung. Allerdings hat sich deren Einbindung verändert, da die C-Laufzeitbibliotheken für C++-Compiler mit angepassten Header-Dateien versehen worden sind, bei denen auch die Namen angepasst wurden.

Um die Funktion `printf` zur Ausgabe einzubinden, haben Sie bisher den folgenden Eintrag verwendet:

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

In C++ ist den C-Laufzeitbibliotheken jeweils der Buchstabe `c` vorangestellt, und das Suffix ».h« fehlt. In C++ erfolgt die Einbindung daher folgendermaßen:

```
#include <cstdio>
```

Analog sind die Namen der anderen Bibliotheken geändert worden[6]. Die wichtigere Änderung dabei ist aber, dass die Funktionen der Standardbibliothek jetzt einem Namensraum mit dem Namen `std` zugeordnet sind. Die Ansprache der Funktionen in ihrem Namensraum `std` kann jetzt so erfolgen:

```
A  #include <cstdio>

   void main()
      {
B     std::printf( "Hello World!\n");
      }
```

In dem Beispiel wird die C++-Version als Äquivalent zur Header-Datei `stdio.h` für die Standardausgabe eingebunden (A) und die Ausgabe mit dem voll qualifizierten Namen aufgerufen (B).

In den meisten Fällen ist es dabei sinnvoll, gleich den kompletten `std`-Namensraum einzubinden, sodass sich folgendes Vorgehen ergibt:

```
A  #include <cstdlib>

B  using namespace std;

   void main()
      {
      char *str = "123456";
C     int n = atoi( str);
      }
```

Hier wird in (A) die Header-Datei `cstdlib` als C++-Äquivalent zu `stdlib.h` eingebunden. Danach wird der komplette Namensraum `std` importiert (B). Damit können die Funktionen der Laufzeitbibliothek ohne die voll qualifizierte Angabe in (C) aufgerufen werden.

Weitere Informationen und Details für Ihren Compiler sollten Sie in der detaillierten Dokumentation des Compilers finden.

6 Die alten Header sind in den meisten Compilern weiter verfügbar, Sie werden aber den neuen Headern zunehmend öfter begegnen.

Kapitel 20
Objektorientierte Programmierung

Der Prolog zu C++ ist geschafft, in diesem Kapitel erkläre ich Ihnen die Grundlagen der objektorientierten Programmierung.

Im letzten Kapitel haben Sie die nicht-objektorientierten Erweiterungen von C++ kennengelernt.

In diesem Kapitel geht es um Theorie und Praxis der objektorientierten Programmierung. Zu Beginn erkläre ich Ihnen, was die *Objektorientierung* ausmacht und welche Vorteile sie bieten soll. Danach stelle ich Ihnen die wichtigsten Konzepte und Begriffe der Objektorientierung vor. Das ist erst einmal völlig unabhängig von einer konkreten Programmiersprache, führt uns aber dann recht schnell zur konkreten Umsetzung der Konzepte in C++, womit wir uns dann schon mitten in der Praxis befinden werden.

20.1 Ziele der Objektorientierung

Bisher haben wir uns ausschließlich mit der prozeduralen Programmierung beschäftigt. Hier sind Datenstrukturen und Funktionen zumindest im Programmcode strikt getrennt. Diese Trennung besteht aber nur im Code selbst. Im Kapitel über Datenstrukturen haben Sie gesehen, dass die Datenstrukturen und die Funktionen, die mit diesen Strukturen arbeiten, in der Verwendung eine Einheit bilden. Das gilt trotz ihrer Trennung im Programmcode.

Wenn Sie sich an die Datenstruktur *Liste* zurückerinnern, wissen Sie, dass die Struktur die Datenelemente einer Liste speichern kann, z. B. Anker, Vorgänger und Zeiger auf ein Listenelement. Sie wissen aber auch noch, dass die Datenstruktur allein dem Programmierer wenig nutzt.

Erst in Kombination mit den Operationen, die auf dieser Datenstruktur arbeiten, wird die Liste wirklich anwendbar. Ohne ihre separat definierten Operationen wie `create`, `insert`, `remove` und `find` ist die Datenstruktur wenig wert.

Die Operationen benötigen die Datenstruktur aber gleichermaßen. Der Nutzen beider Elemente entsteht aus der Kombination von Daten und Verhalten.

Diese Kombination der beiden Elemente ist also notwendig, sie bedeutet aber auch einen erhöhten Aufwand bei der Wartung und Pflege des Codes. Dies gilt besonders,

weil der Code für die Datenstruktur und die Operationen typischerweise über mehrere Dateien hinweg verteilt ist.

Eine Änderung muss meist parallel an beiden Elementen durchgeführt werden. Wenn neue Operationen hinzugefügt werden, muss oft eine Anpassung der Datenstruktur erfolgen, damit die Operationen die Struktur verwenden können und umgekehrt.

Durch die Trennung von Daten und Verhalten wird damit die Wartbarkeit und Erweiterbarkeit eines Systems erschwert. Dies ergibt sich durch die Aufteilung in die genannten Bereiche und die daraus resultierende Aufteilung auf mehrere Dateien.

Ein Bestreben der objektorientierten Programmierung ist es nun, die Daten und Operationen zu kombinieren, um die Wartung und Weiterentwicklung zu vereinfachen. Wie diese Kombination erreicht wird, werden Sie in diesem Kapitel erfahren.

Moderne Softwareentwicklung ist dadurch geprägt, dass die Systeme immer umfangreicher und komplexer werden. Während der Umfang von Projekten in den 70er-Jahren des vergangenen Jahrhunderts noch in der Größenordnung von Zehn- und Hunderttausenden Zeilen Programmcode gelegen hat, sind heute mehrere Millionen Zeilen von Code, sogenannten *Source Lines of Code (SLOC)* in Projekten durchaus üblich:

Jahr	System	SLOC [Mio.]
1993	Windows NT 3.1	4–5
1996	Windows NT 4.0	11–12
2000	Windows 2000	>29
2001	Windows XP	40
2003	Windows Server 2003	50
2000	Debian 2.2	55–59
2002	Debian 3.0	104
2007	Debian 4.0	283
2005	Mac OS X 10.4	86
2003	Linux-Kernel 2.6.0	5,2
2009	Linux-Kernel 2.6.32	12,6
2012	Linux-Kernel 3.6	15,9
2007	SAP NetWeaver	238

Tabelle 20.1 Anzahl der Codezeilen in verschiedenen Softwareprojekten (Quelle: Wikipedia)

Die Angabe der Codezeilen eines Projekts durch *SLOC* ist für einen Vergleich der Komplexität von Projekten mit Sicherheit nicht geeignet, gibt Ihnen aber eine Vorstellung davon, welche Mengen an Code in großen Projekten verwaltet werden müssen. Es ist einleuchtend, dass bei solchen Projektgrößen diese Themen zunehmend an Bedeutung gewinnen:

- **Wartbarkeit**: Existierender Programmcode muss möglichst einfach gepflegt und erweitert werden können. Dies umfasst Programmcode und Programmstrukturen, die auch dann einfach zu verstehen, zu korrigieren und zu erweitern sein müssen, wenn der Entwickler den Code nicht selbst erstellt hat.
- **Wiederverwendbarkeit**: Einzelne Elemente aus einem Programmsystem sollen möglichst einfach in einem anderen System verwendet werden können. Dazu muss es möglich sein, eine Kopie des Elements möglichst einfach in ein anderes Programm integrieren zu können. Die Abhängigkeiten, die sogenannte *Kopplung*, müssen dazu möglichst gering gehalten werden. Das Element muss außerdem einfach für die Anforderungen weiterer Systeme konfiguriert werden können.
- **Einfache Modellierung**: Die Elemente sollen möglichst einfach modelliert werden können. Dies bedeutet, dass Konzepte aus der realen Welt leicht in Software umgesetzt werden können. Der Bruch, der zwischen den Konzepten in der realen Welt und der Abstraktion in der Software erfolgt, soll klein gehalten werden. Die Elemente der Software sollen am Ende einfach zu verstehen und zu nutzen sein.

Allen drei Punkten ist gemeinsam, dass sie den Aufwand in der Entwicklung verringern und Komplexität reduzieren sollen. Bei Erstellung und Betrieb von Software übersteigen die Kosten für die Menschen, die diese Software erstellen, die Kosten, die für den technischen Betrieb der Maschinen aufgewendet werden, in vielen Fällen. Performance und Effizienz von Programmen sind weiter wichtig, in vielen Bereichen bestimmen aber die Kosten und die Geschwindigkeit der Erstellung, Wartung und Pflege eines Systems den Erfolg im Wettbewerb.

20.2 Objektorientiertes Design

Um die objektorientierte Entwicklung praktisch zu diskutieren, müssen wir uns zuerst den Begriffen widmen, die in diesem Bereich verwendet werden. Sie ahnen vielleicht schon, dass der erste Begriff hier der des *Objekts* ist. Als ein Objekt bezeichnet man die softwaretechnische *Repräsentation* eines Gegenstands oder eines Konzepts aus dem Anwendungsgebiet. Der Gegenstand kann dabei real existieren oder auch gedacht sein. Objekte leiten sich typischerweise aus Substantiven ab.

Generell ist alles, was im Anwendungsgebiet unseres Programms als Substantiv verwendet wird, ein Anwärter dafür, als Objekt repräsentiert zu werden. Beispiele dafür

sind eine Person, eine Datei, ein Bankkonto, ein Datum oder Termin oder auch eine Prüfung. Aber auch abstraktere Konzepte wie Listen oder Warteschlangen sind geeignet, als Objekt repräsentiert zu werden.

Um diese Konzept in eine Software zu überführen, erfolgt eine sogenannte *Modellierung*, bei der man die Eigenschaften der modellierten Konzepte vorab erfassen und darstellen möchte. Für die Modellierung von Objekten und Systemen, das sogenannte *objektorientierte Design*, hat sich eine Methodik entwickelt, die so eng mit der objektorientierten Programmierung verbunden ist, dass sie in der Beschreibung objektorientierter Systeme allgegenwärtig ist. Ich werde die Grundbegriffe der sogenannten *UML*[1]-Methodik für die sogenannten *Klassendiagramme* einführen und für einige Beispiele verwenden. Für eine detaillierte Darstellung von UML und der Modellierung objektorientierter Systeme verweise ich Sie auf weiterführende Literatur.

Für mein Beispiel wähle ich mit einem Auto und einem Datum einen Gegenstand und ein Konzept, die ich als Objekte repräsentieren möchte.

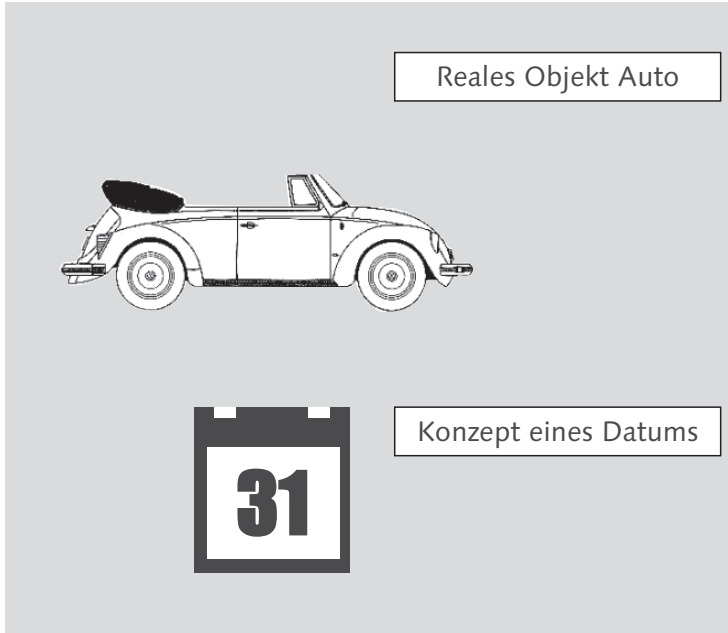


Abbildung 20.1 Beispiel für zu repräsentierende Objekte

In der UML werden Objekte durch ein Rechteck mit drei Feldern repräsentiert. Im oberen Feld steht der Name des Objekts:

1 UML = Unified Modeling Language

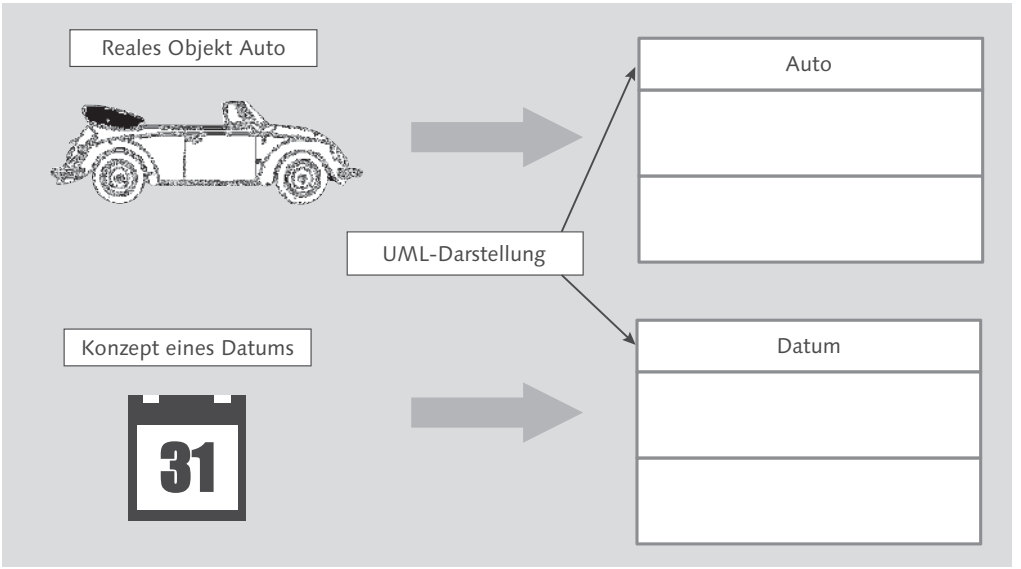


Abbildung 20.2 Darstellung von Objekten in der UML

Ein Objekt befindet sich zu jedem Zeitpunkt in einem genau definierten *Zustand*. Dieser Zustand wird durch die *Attribute* des Objekts beschrieben. Die Attribute enthalten alle Daten, um den Zustand des Objekts als Modell vollständig und konsistent zu beschreiben. Die Attribute eines Objekts werden in der UML im mittleren Feld der Objektbeschreibung dargestellt.

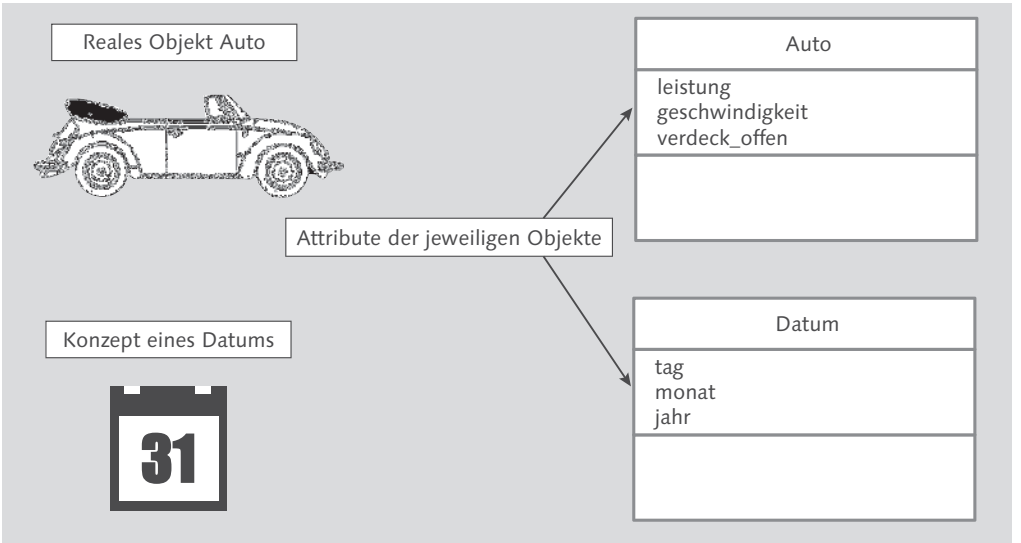


Abbildung 20.3 Darstellung der Attribute von Objekten

Unser Datumsobjekt soll ein Datum speichern können. Dazu möchten wir den Tag, den Monat und das Jahr festhalten. Wir sehen daher die entsprechenden Attribute vor. Die Attribute, die ich für die Modellierung des Autos vorgesehen habe, können Sie dem Diagramm entnehmen. Sie sehen hier auch schon zwei unterschiedliche Arten von Attributen. Die Leistung des modellierten Autos gehört zu den beständigen *Stammdaten*, die oft nur bei der Erstellung eines Objekts erzeugt oder geändert werden. Der Zustand des Verdecks gehört zu den *Bewegungsdaten*, die sich öfter verändern. Für die weitere Darstellung werde ich aber nicht zwischen beiden unterscheiden.

Eine Warteschlange als Objekt hätte als Attribute z. B. die Größe der Warteschlange sowie die enthaltenen Elemente und deren Reihenfolge.

An dieser Stelle könnten Sie einwenden, dass die Attribute, die ich zur Beschreibung verwendet habe, nicht ausreichend sind oder schon zu viele Informationen enthalten. Beides kann richtig sein. Für einen Automobilhersteller wäre »mein« Auto mit den gegebenen Attributen sicherlich nicht ausreichend beschrieben. Auch eine Software für Probleme der Astronomie oder zur Speicherung geologischer Daten würde aufgrund der auftretenden Zeiträume die eher in Millionen von Jahren gemessen werden, sicherlich eine andere Datumsrepräsentation verwenden.

Die hier angegebenen Beispiele stellen natürlich nur eine exemplarische und noch unvollständige Beschreibung dar und müssten weiter detailliert werden – je nachdem, für welchen Zweck das Objekt verwendet werden soll. Die Modellierung eines Objekts muss für die Problemstellung angemessen sein und kann sich von Anwendungsfall zu Anwendungsfall unterscheiden. Vorerst soll die oben beschriebene Modellierung aber ausreichen.

Den hier gezeigten Zustand konnten Sie auch mit Datenstrukturen schon gut abbilden. Objekte haben zu ihrem Zustand aber noch ein dynamisches Verhalten. Die *Methoden* eines Objekts beschreiben alle Operationen, die das Objekt ausführen kann.

In der UML werden die Methoden eines Objekts im unteren Feld der Objektbeschreibung geführt (siehe Abbildung 20.4).

In dem Beispiel sehe ich hier nur die Möglichkeit zum Setzen eines Datums und zum Lesen eines Datums sowie eine Abfrage vor, die darüber Auskunft gibt, ob ein Datum in einem Schaltjahr liegt.

Um den Einstieg in das Thema zu finden, sind die Begriffe in der oben stehenden Erläuterung noch etwas unscharf.

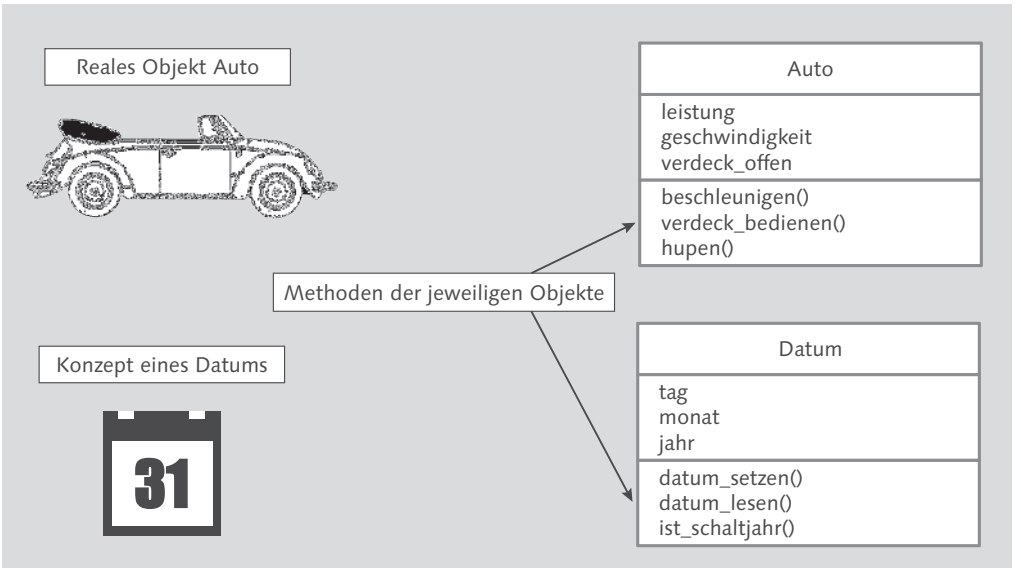


Abbildung 20.4 Darstellung der Methoden von Objekten

Dies will ich jetzt präzisieren. Bisher habe ich nicht unterschieden zwischen einem Datum in seiner allgemeinen Form und einem konkreten Datum. Diese Unterscheidung ist aber ausgesprochen wichtig. So hat ein Datum im Allgemeinen Tag, Monat und Jahr. Der letzte Heiligabend im 20. Jahrhundert hatte aber genau das konkrete Datum 24.12.2000.

Betrachtet man gleichartige Objekte, findet man eine Reihe von Gemeinsamkeiten, aus denen man Klassen bilden kann, wie wir es für Autos und Daten schon getan haben.

Jedes betrachtete Datum hat einen Tag, einen Monat und ein Jahr. Es gibt also eine Vorlage, die ein Datum im Sinne unseres Anwendungsfalls allgemein beschreibt. Diese Vorlage existiert aus sich heraus, sie ist eine generelle Beschreibung. Auch wenn noch kein einziges konkretes Datum im System erfasst ist, können wir eine generelle Klasse zu seiner Beschreibung erstellen.

Unter einer *Klasse* verstehen wir die softwaretechnische Beschreibung einer Vorlage (Attribute und Methoden) für ein Objekt. [!]

Wenn wir ein Objekt als eine spezielle Ausprägung einer solchen Klasse mit einem konkreten Zustand erstellen, wird dieser Prozess *Instanziierung* genannt. Daher werden die aus einer Klasse erzeugten Objekte auch als *Instanzen* bezeichnet.

So, wie Sie mit einer Plätzchenform konkrete Plätzchen ausstechen können, können Sie mit einer Klasse konkrete Instanzen erstellen.

[!] Eine Instanz ist eine konkrete Ausprägung einer Klasse mit spezifischen Attributen, die ihren Zustand bestimmen.

Wenn der Begriff *Objekt* verwendet wird, ist damit nicht klar, ob eine Klasse oder eine Instanz gemeint ist. Im Folgenden werde ich daher vorrangig die Begriffe *Klasse* und *Instanz* nutzen. Von Objekten werde ich nur sprechen, wenn die Unterscheidung unerheblich ist. Wie Sie vielleicht jetzt schon erkennen können, befasst sich das objektorientierte Design eher mit Objekten im Sinne von Klassen als im Sinne von Instanzen.

Die Unterscheidung zwischen der Klasse und der Instanz findet sich auch in der UML. Dort wird eine Instanz dadurch deutlich gemacht, dass der Name der Klasse unterstrichen und der Name der Instanz durch einen Doppelpunkt getrennt hinten angefügt wird. Die konkreten Attributwerte einer Instanz notieren wir mit einem Gleichheitszeichen hinter dem Attributnamen.

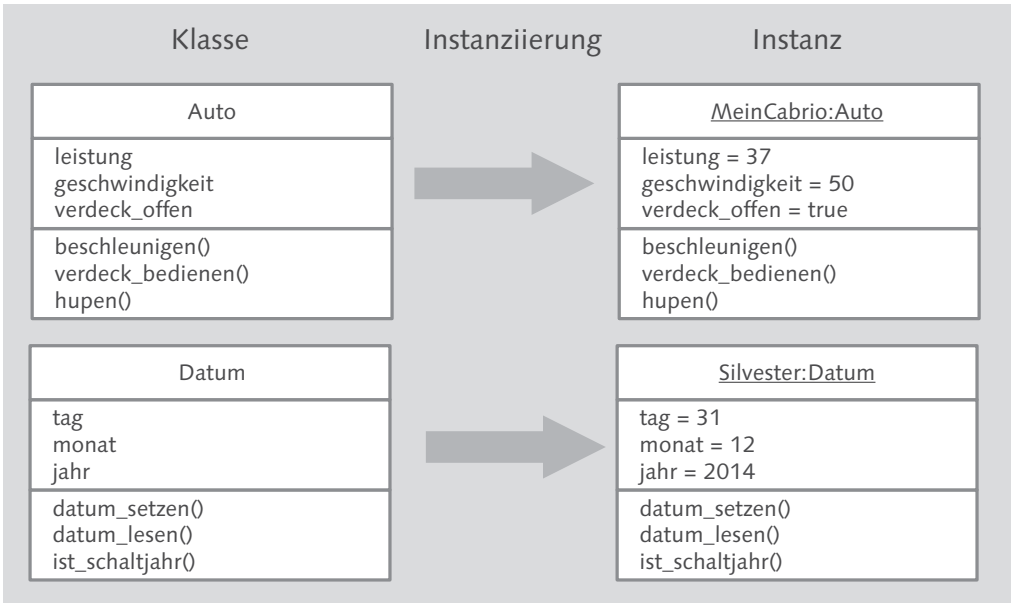


Abbildung 20.5 Darstellung instanziierter Klassen

Weitere Konzepte der Objektorientierung wie die *Vererbung* und die sogenannte *Polymorphie* oder auch Vielgestaltigkeit werden wir vorerst zurückstellen und betrachten, wenn wir sie direkt praktisch umsetzen können.

20.3 Klassen in C++

Ich habe bisher allgemein über Klassen und Instanzen gesprochen und will Ihnen nun zeigen, wie Klassen in C++ umgesetzt werden. Klassen ermöglichen es Ihnen, Datentypen in einer eleganteren und leichter wiederverwendbaren Art zu implementieren, als Sie das bisher mit den Datenstrukturen `struct` konnten.

Als Beispiel werden wir in den folgenden Kapiteln die Elemente eines Logbuchs verwenden und anfangs auch das oben erwähnte Datum wieder kurz aufgreifen. In dem Logbuch sollen bestimmte Ereignisse protokolliert werden. Zu einem Logbucheintrag gehört neben der Bezeichnung des eingetretenen Ereignisses das Datum, an dem es stattgefunden hat. Wir werden das Beispiel im weiteren Verlauf ausbauen, es steht damit aber bereits fest, dass wir eine Klasse zur Repräsentation des Datums benötigen werden. Mit dieser Klasse werden wir starten und wollen dazu die Klasse `datum` erstellen, mit der wir ein Kalenderdatum repräsentieren. Anhand dieses Beispiels werden wir die ersten Schritte in der objektorientierten Programmierung gehen und später auch wesentliche Aspekte der Objektorientierung wie die Vererbung umsetzen.

20.4 Aufbau von Klassen

Die grundlegende Einheit unseres Beispiels ist die Klasse `datum` zur Verwaltung eines Kalenderdatums. Von ihrem äußeren Aufbau ist eine Klasse einer Datenstruktur sehr ähnlich. Es wird lediglich das Schlüsselwort `struct` durch das neue Schlüsselwort `class` ersetzt:

```
A  class datum
B     {
      
      
      
      
C     };
```

Listing 20.1 Der Aufbau einer Klasse

Auf das Schlüsselwort `class` und den Namen der Klasse (A) folgt eine geöffnete geschweifte Klammer, die den Anfang des Blocks mit dem Inhalt der Klasse markiert (B). Wie eine Struktur hat die Klasse eine geschweifte Klammer als schließendes Element, gefolgt von einem Semikolon (C).

Datenstrukturen können ausschließlich Daten enthalten. Klassen können stattdessen Daten und Funktionen als Elemente enthalten. Beide werden auch allgemein als *Member*[2] bezeichnet.

Dabei unterscheiden wir

- *Datenmember*, auch *Attribute* genannt
- *Funktionsmember*, auch als *Methoden* bezeichnet

Der weitere grundlegende Unterschied zwischen Strukturen und Klassen liegt in dem Zugriffsschutz, der für die Member einer Klasse vorgesehen werden kann.

20.4.1 Zugriffsschutz von Klassen

Ein Zugriffsschutz auf bestimmte Objekte ist Ihnen aus dem Alltag vertraut. Dort ist es normal, dass nicht jeder mit allen Objekten alles tun kann. So darf z. B. jeder an meine Haustür gehen und dort klingeln. Der Zugriff darauf ist öffentlich. Jeder kann Post in meinen Briefkasten werfen, Briefe herausnehmen darf aber nur ich. Hier ist offensichtlich der »schreibende« Zugriff (Post einwerfen) öffentlich, der »lesende« Zugriff (Briefe entnehmen) ist privat.

Auch der Zugriff auf meinen Kühlschrank ist privat. Auf meinen Kühlschrank kann allerdings auch meine Familie zugreifen. Diese Form eines geschützten Zugriffs finden wir auch in der objektorientierten Entwicklung bei Objekten, die miteinander verwandt sind. Diese Form der Verbindung werden Sie später als *Vererbung* kennenlernen.

Im täglichen Leben kann ich z. B. auch Freunden (`friend`) einen Zugriff auf meinen Kühlschrank erlauben, sie können sich dann an meinem privaten Kühlschrank bedienen, als wäre es ihr eigener. Die sonst geltende Einschränkung ist für sie aufgehoben.

Diese unterschiedlichen Formen des Zugriffs und deren Aufteilung in verschiedene Bereiche gibt es auch in der objektorientierten Programmierung. Die Unterscheidung zwischen öffentlich (`public`), privat (`private`) und geschützt (`protected`) sowie die Erklärung von Freundschaften (`friend`) finden wir auch dort wieder.

Anders als bei Datenstrukturen gibt es bei Klassen also einen Schutz gegen die missbräuchliche Verwendung der Member der Klasse. Um diesen Schutz wirksam werden zu lassen, werden die Member einer Klasse in entsprechenden Bereichen deklariert. Die entsprechenden Bereiche werden innerhalb der Klassendefinition durch die jeweiligen Schlüsselwörter gekennzeichnet:

2 engl. Member = Mitglied


```
   class datum
      {
A     private:

B     protected:

C     public:

      };
```

Listing 20.2 Der Zugriffsschutz in einer Klasse

Auf Elemente im privaten Bereich `private` (A) besteht besonderer Zugriffsschutz. Auf Elemente im öffentlichen Bereich `public` (C) kann jeder zugreifen. Die Bedeutung des Bereichs `protected` (B) werden Sie später kennenlernen.

Alle Elemente, die keinem Bereich zugeordnet sind, sind privat. Die Bereiche können auch mehrfach und in beliebiger Reihenfolge vorkommen. Die hier angegebene Reihenfolge ist allerdings üblich und bietet sich aus Gründen der Übersichtlichkeit an.

20.4.2 Datenmember

Die Daten, die wir in Klassen speichern, werden *Attribute* oder *Datenmember* genannt. Zur Umsetzung unserer Datumsklasse können wir die notwendigen Daten im öffentlichen Bereich der Klasse speichern. Damit ist ein lesender und schreibender Zugriff von überall her möglich.

```
   class datum
      {
A     public:
B        int tag;
         int monat;
         int jahr;
      };
```

Listing 20.3 Die Klasse »datum«

Nach dem Beginn des öffentlichen Bereichs (A) werden die Attribute zur Speicherung eines Datums in der Klasse deklariert (B).

Der Zugriff auf Attribute erfolgt wie bei Strukturen mit dem Punkt-Operator:

```
int main()
   {
   datum d;
   d.tag = 24;
   d.monat = 12;
   d.jahr = 2014;
   }
```

Listing 20.4 Zugriff auf die Daten mit dem Punkt-Operator

In unserem Beispiel werden bisher nur Integer verwendet, generell können in einer Klasse aber alle elementaren Datentypen und deren Arrays als Attribute verwendet werden[3]. Dies umfasst auch Datenstrukturen:

```
   class klasse
      {
A     private:
         int i;
         float f;
         char c;
B        char string[20];
      public:
         int* pi;
C        struktur str;
         //...
D     public:
         short s;
      };
```

Listing 20.5 Zugriff auf die Attribute der Klasse datum

Im Beispiel ist eine Auswahl verschiedener Datentypen als `private`-Attribute (A) der Klasse deklariert, unter anderem ein Array (B), eine Struktur (C) und in einem zweiten öffentlichen Bereich (D) ein weiterer elementarer Datentyp. Natürlich können auch Klassen in Klassen vorkommen, dies wird im nächsten Kapitel detailliert behandelt.

Bis zu dieser Stelle ist unsere Klasse mit einer Datenstruktur identisch. Faktisch entspricht eine Klasse mit ausschließlich öffentlichen Datenmembern einer Datenstruktur.

3 Ebenso wie bei Datenstrukturen müssen die Werte der Attribute korrekt initialisiert werden.

20.4.3 Funktionsmember

Wir haben in der Vergangenheit bereits ausgiebig mit Datenstrukturen und separaten Funktionen gearbeitet. Dabei waren die Datenstrukturen und Funktionen strikt getrennt. Formal waren sie damit zwar unabhängig, wenn Sie sich deren Zusammenwirken anschauen, sind Daten und Funktionen aber durchaus stark miteinander verflochten.

Werfen Sie z. B. einen Blick auf das behandelte Listen-Modul, dann ist offensichtlich, dass die Datenstruktur erst mit den Listenoperationen (`create`, `insert`, `remove`, `find`) sinnvoll verwendet werden kann. Andererseits können die Operationen nur mit der jeweiligen Datenstruktur arbeiten und sind ohne diese Strukturen nutzlos.

Bei Anpassungen der Datenstruktur folgen typischerweise auch Anpassungen der Operationen. Andersherum erfordern Änderungen an der Funktionalität der Operationen vielfach auch eine Änderung der Datenstrukturen. Es besteht also ein starker Zusammenhang.

Eine unkoordinierte Änderung oder Erweiterung an Datenstrukturen oder Operationen kann dabei schnell zu einem nicht mehr überschaubaren und damit nicht mehr wartbaren System führen.

Es liegt also nahe, Datenstruktur und Operation zu kombinieren. Dies passiert in einem Objekt, indem zusammengehörige Daten und Funktionen als Member einer Klasse zusammengeführt werden.

Wir wollen nun unserer Datumsklasse eine erste Methode und damit ein *Verhalten* hinzufügen. Dabei beginnen wir mit einer *Methode*, die übergebene Parameter für `tag`, `monat` und `jahr` in den entsprechenden Datenmembern speichert.

Dieses Verhalten erscheint im Moment etwas überflüssig, da der Benutzer der Klasse die Werte ja direkt schreiben kann, wie Sie bereits gesehen haben. Der Nutzen diese Methode wird sich aber sehr schnell zeigen.

Wir werden unsere Methode `set` nennen. Methoden, die die Werte von Datenmembern setzen, werden oft auch als *Setter-Methoden* bezeichnet. Unsere Methode soll die folgende Schnittstelle besitzen:

```
void set( int t, int m, int j )
```

Die Methode erhält die Parameter für die zu setzenden Werte von `tag`, `monat` und `jahr` und liefert keinen Rückgabewert.

Wir platzieren unsere Setter-Methode mit ihren drei Parametern ebenfalls im öffentlichen Bereich der Klasse. So kann jeder auf sie zugreifen.

Methoden können direkt in der Klassendeklaration als sogenannte *Inline-Methoden* implementiert werden. Sie werden dann direkt in die Klassendeklaration geschrieben:

```
   class datum
      {
      public:
         int tag;
         int monat;
         int jahr;

A        void set( int t, int m, int j) { tag = t; monat = m; jahr = j;}
      };
```

Listing 20.6 Inline-Implementierung der Methode »set«

Die Methode wird *in* der Klasse mit einer *Inline-Implementierung* angelegt (A). Dies sieht auf den ersten Blick etwas ungewöhnlich aus, wenn Sie die Formatierung anpassen, erkennen Sie aber schnell die gewohnte Form:

```
void set( int t, int m, int j)
   {
   tag = t;
   monat = m;
   jahr = j;
   }
```

Listing 20.7 Umgestellter Code der Methode set

Die Attribute sind für die Klasse definiert, zu der die Methode gehört. Innerhalb der Methode erfolgt der Zugriff auf die Attribute der Klasse selbst ohne Punkt-Operator.

Einen Zugriff auf Attribute und Methoden einer Klasse durch Methoden der Klasse selbst bezeichnen wir auch als einen Zugriff von »innen«. Alle anderen Zugriffe werden auch als Zugriff von »außen« bezeichnet.

Wir haben mit `set` eine öffentliche Methode zum Schreiben der Attribute implementiert. Der Aufruf einer solchen Methode von außen erfolgt wie der Zugriff auf die Attribute einer Klasse über den Punkt-Operator:

```
   int main()
      {
      datum ha; //Heiligabend
A     ha.set( 24, 12, 2014);
B     int t = ha.tag;
      printf( "Der Tag des Datums ist: %d\n", t);
      }
```

Listing 20.8 Die Verwendung der Methode »set«

Aus dem Hauptprogramm erfolgen ein Aufruf der Methode `set` zum Setzen der Attribute für das Datum `ha` (A) und ein lesender Zugriff auf das Attribut `tag` (B), und wir erhalten damit folgende Ausgabe:

```
Der Tag des Datums ist: 24
```

20.4.4 Verwendung des Zugriffsschutzes

Wir haben zuerst alle Attribute in den öffentlichen Bereich gelegt, dort besteht aber keinerlei Schutz gegen ungewollte Änderungen von außen. Die Setter-Methode ist eine Vorbereitung auf dem Weg, die Attribute vor unkontrolliertem Zugriff zu schützen. Im nächsten Schritt werden wir mit der Deklaration der Attribute als privat genau diesen Schutz einrichten.

```
   class datum
      {
      private:
A        int tag;
         int monat;
         int jahr;
      public
B        void set( int t, int m, int j)
                  { tag = t; monat = m; jahr = j;}
   };
```

Listing 20.9 Platzierung der Attribute im privaten Bereich

Wir erklären daher die Datenmember für `private` (A) und belassen nur die `set`-Methode im öffentlichen Bereich (B). Die `set`-Methode bleibt damit auch weiter aus unserem Programm aufrufbar. Die Methode selbst gehört mit zur Klasse, sie befindet sich »innen«, daher darf sie ohne Einschränkungen auf private Elemente der Klasse zugreifen. Hier zeigt sich nun der Nutzen dieser Methode, die jetzt weiter den Zugriff auf die nun privaten Daten ermöglicht.

Wir testen unsere veränderte Klasse:

```
   int main()
      {
      datum ha; //Heiligabend
A     ha.set( 24, 12, 2014);
B     int t = ha.tag; // FEHLER
      }
```

Listing 20.10 Fehler bei lesendem Zugriff

Der Zugriff auf die `set`-Methode im öffentlichen Bereich (A) arbeitet wie erwartet. Der Zugriff von außen auf das private Attribut `tag` der Klasse (B) schlägt fehl.

Durch die Verschiebung in den privaten Bereich können die Attribute nun allerdings auch nicht mehr gelesen werden. Wir wollen unsere Attribute aber natürlich nicht nur setzen, sondern sie auch auslesen können. Die Lösung für dieses Problem liegt nahe. Wir erstellen zu jedem zu lesenden Attribut eine Methode, die das Attribut ausliest und den entsprechenden Wert als Resultat zurückgibt. Solche lesenden Methoden werden auch *Getter-Methoden* genannt. Auch unsere Getter legen wir als Inline-Methoden an und platzieren sie im öffentlichen Bereich:

```
   class datum
      {
      private:
         int tag;
         int monat;
         int jahr;
      public:
A        int getTag() { return tag;}
B        int getMonat() { return monat;}
C        int getJahr() { return jahr;}
         void set( int t, int m, int j)
                  { tag = t; monat = m; jahr = j;}
      };
```

Listing 20.11 Klasse datum mit Getter-Methoden (inline)

Nun können wir auch direkten lesenden Zugriff auf die Attribute durch unsere Getter-Methoden (A, B und C) ersetzen und erhalten den gewünschten Wert. Damit können wir über die Methoden nun mittelbar wieder auf die privaten Attribute zugreifen:

```
int main()
   {
   datum ha; //Heiligabend
   ha.set( 24, 12, 2014);
   int t = ha.getTag();
   }
```

Listing 20.12 Zugriff auf private Attribute über öffentliche Getter

Die Datenmember der Klasse liegen jetzt im privaten Bereich und sind nur noch über öffentliche Getter- und Setter-Methoden zugänglich, die das Verhalten der Klasse darstellen.

Auf den ersten Blick mag es so erscheinen, als wären wir praktisch nicht weiter als am Anfang. Aber wir haben einen großen Fortschritt gemacht. Die Klasse hat ihre Daten nun *gekapselt*. Zugriffe auf die internen Daten, die den Zustand beschreiben, sind nur noch über die bereitgestellten Methoden möglich. Dies eröffnet uns verschiedene Optionen:

- Wir haben nun die Möglichkeit, schreibenden Zugriff so zu gestalten, dass fehlerhafte Eingaben korrigiert oder abgelehnt werden, ohne die Instanz der Klasse in einen inkonsistenten Zustand zu bringen.
- Wenn die Anwender der Klasse nur noch die Getter- und Setter-Methoden verwenden, kann der Entwickler der Klasse z. B. Namen der Attribute oder auch deren Datentyp einfach ändern. Nur die Methoden müssten dann aktualisiert werden. Da alle Zugriffe ausschließlich durch diese entsprechenden Methoden erfolgen (können), müsste ein Nutzer dieser Klasse seinen eigenen Programmcode nicht ändern. Auf diese Weise haben wir in Sachen Wartbarkeit einen großen Fortschritt erzielt, da Anpassungen in der Klasse erfolgen können, ohne dass der Nutzer der Klasse zu Änderungen gezwungen wird.

Auch dieses Konzept kennen Sie aus dem Alltag. So kann der Tageskilometerzähler meines Autos einfach abgelesen werden, das Auslesen des aktuellen Standes ist also öffentlich. Das Verändern des Kilometerstandes ist nur von innen heraus durch die Steuergeräte möglich. Das Fahrzeug bietet mir allerdings einen zusätzlichen öffentlichen Zugriff von außen – mit der eingeschränkten Funktionalität, den Zählerstand auf null zurückzusetzen.

Wir wollen in diesem Sinne unsere Setter-Methode so erweitern, dass sie vor dem Setzen der Attribute eine einfache Prüfung vornimmt und ungültige Werte korrigiert. Die Prüfungen und die Korrektur halten wir hier bewusst einfach. Wir wollen für unser Datum annehmen, dass es gültige Daten zwischen dem 01.01.1970 und dem 30.12.2099 aufnehmen soll. Wir gehen davon aus, dass alle Monate 30 Tage lang sind. Selbst mit dieser Einschränkung ist bereits jetzt abzusehen, dass die Implementierung der Methode mehrere Zeilen in Anspruch nehmen wird. Eine Inline-Implementierung wird hier schnell unübersichtlich. Wir wollen daher dieses Funktionsmember außerhalb der Klasse implementieren. Dazu ersetzen wir zuerst unsere bisherige Implementierung durch eine Deklaration der Methode:

```
   class datum
      {
      private:
         int tag;
         int monat;
         int jahr;
```

```
   public:
      int getTag() { return tag;}
      int getMonat() { return monat;}
      int getJahr() { return jahr;}
A     void set( int t, int m, int j);
   };
```

Listing 20.13 Deklaration einer Methode in der Klasse

Innerhalb der Klasse `datum` wird die Methode `set` nun lediglich deklariert und nicht implementiert (A).

Für die innerhalb der Klasse deklarierte Methode erfolgt nun die Implementierung außerhalb der Klasse[4]:

```
A  void datum::set( int t, int m, int j )
      {
      if( j<1970 || j>2099 )
         j = 1970;
      if( m<1 || m>12 )
         m = 1;
      if( t<1 || t>30 )
         t = 1;
      tag = t; monat = m; jahr = j;
      }
```

Listing 20.14 Implementierung der Methode der Klasse

Zur Implementierung der Methode wird der voll qualifizierte Name der Klasse und Methode als Name der zu implementierenden Funktion verwendet (A). Dieser bildet sich aus dem Klassennamen, gefolgt vom Class-Member-Operator `::` und dem Namen der Methode. Innerhalb der Funktion erfolgt die Implementierung in gewohnter Art und Weise. Den Code zur Korrektur der Daten selbst werden wir hier nicht weiter vertiefen. Ungültige Werte werden einfach immer auf einen vorgegebenen Wert korrigiert.

Mit unserer neuen Setter-Methode werden falsche Datumswerte nun bei der Eingabe entsprechend korrigiert, sodass unser Objekt nach Aufruf der `set`-Methode immer ein gültiges Datum nach den vorgegebenen Regeln enthält:

4 Die Datei, in der die Implementierung erfolgt, benötigt natürlich Kenntnis der Klassendeklaration. Typischerweise erfolgt die Implementierung in einer Datei *datum.cpp*, die die Deklaration in *datum.h* inkludiert.


```
   int main()
      {
      datum dat(
A     dat.set(0, 1 1066);
      printf( "Datum: %d.%d.%d\n",
              dat.getTag(), dat.getMonat(), dat.getJahr());
      }
```

Listing 20.15 Verwendung der korrigierenden set-Methode

Wenn wir der `set`-Methode nun ungültige Parameter übergeben (A), werden die Eingaben auf gültige Werte korrigiert. Damit erhalten wir die folgende Ausgabe:

```
Datum: 1.1.1970
```

20.4.5 Konstruktoren

Wir können nun Instanzen unserer Klasse `datum` anlegen und haben mit der `set`-Methode dafür gesorgt, dass nur noch im Rahmen der Anforderungen gültige Datumswerte in die Attribute eingetragen werden können.

Damit ist aber immer noch nicht garantiert, dass die Attribute des Objekts immer mit konsistenten Werten gefüllt sind. Dies liegt daran, dass wir bisher nicht kontrollieren können, in welchem Zustand das Objekt erstellt wird. Direkt nach seiner Instanziierung ist der Zustand des Objekts noch undefiniert.

Wir benötigen eine Möglichkeit, den Zustand bereits während der Erstellung der Instanz zu kontrollieren. Diese Möglichkeit gibt es mit dem sogenannten *Konstruktor*. Der Konstruktor steuert den Instanziierungsprozess eines Objekts und ist dafür verantwortlich, die Instanz bei der Erstellung direkt in einen konsistenten Zustand zu bringen.

Das Gegenstück zum Konstruktor ist der *Destruktor*, der den Abbau einer Instanz steuert. Diesen werden wir abschließend behandeln.

Der Konstruktor einer Klasse ist eine spezielle Methode. Sie ist so eng mit der Klasse verknüpft, dass sie den gleichen Namen trägt wie die Klasse selbst. Ein Konstruktor hat *keinen* Rückgabetyp, nicht einmal `void`.

Eine Klasse kann keinen, einen oder mehrere Konstruktoren haben. Ebenso wie andere Methoden kann der Konstruktor parameterlos sein. Wenn eine Klasse mehrere Konstruktoren hat, wird wie bei überladenen Funktionen der passende Konstruktor ausgewählt. Dazu werden die Parameter verwendet, die bei der Instanziierung angegeben worden sind. Entsprechend müssen sich auch die Parametersignaturen der Konstruktoren unterscheiden.

Wir wollen nun unserer Klasse einen Konstruktor hinzufügen. Der Konstruktor soll von überall aufrufbar sein, daher platzieren wir ihn im öffentlichen Bereich.

```
   class datum
      {
      private:
         int tag;
         int monat;
         int jahr;
      public:
         int getTag() { return tag;}
         int getMonat() { return monat;}
         int getJahr() { return jahr;}
         void set( int t, int m, int j);
A        datum( int t, int m, int j);
      };
```

Listing 20.16 Klasse mit der Deklaration eines Konstruktors

Innerhalb der Klasse deklarieren wir einen Konstruktor (A) und implementieren ihn wie unseren Setter außerhalb der Klassendeklaration:

```
datum::datum( int t, int m, int j)
   {
   set(t, m, j);
   }
```

Listing 20.17 Die Implementierung des Konstruktors

Die Implementierung selbst greift auf den bestehenden Setter zurück und korrigiert damit eventuell übergebene falsche Werte. Wir verwenden unseren Konstruktor nun bei der Instanziierung eines Datums:

```
   int main()
A     {
      datum em( 1, 5, 2014 ); //1. Mai
      printf( "1. Mai: %d.%d.%d\n",
              em.getTag(), em.getMonat(), em.getJahr());
      }
```

Listing 20.18 Verwendung des Konstruktors

Wir bedienen in unserem Programm die Schnittstelle des Konstruktors (A) und erstellen damit eine Instanz der Klasse. Wenn es mehrere Konstruktoren gibt, wird der passende anhand der Parametersignatur ausgewählt.

Bei der Ausgabe erhalten wir nun das erwartete Ergebnis:

```
1. Mai: 1.5.2014
```

Sobald eine Klasse einen Konstruktor mit Parametern enthält, ist der parameterlose Konstruktor, den wir im Beispiel bisher verwendet haben, nicht mehr verfügbar.

Solange wir keinen Konstruktor explizit bereitgestellt hatten, war er vom System automatisch erstellt worden. Dies ist jetzt nicht mehr der Fall. Die folgende Erstellung eines Objekts ist damit jetzt *nicht* mehr möglich:

```
int main()
   {
   datum ha; // FEHLER
   }
```

Listing 20.19 Fehlender parameterloser Konstruktor

Wenn wir nach der Erstellung des neuen Konstruktors ein Datumsobjekt ohne Parameter erzeugen wollen, erhalten wir einen Compiler-Fehler mit dem Hinweis, der Konstruktor sei nicht verfügbar.

Wenn Sie für eine Klasse keinen Konstruktor definieren, erstellt C++ für diese Klasse *automatisch* einen Konstruktor ohne Parameter. Bisher haben Sie diesen automatisch erstellten Konstruktor verwendet, ohne es zu wissen. Nachdem Sie aber den ersten eigenen Konstruktor erstellt haben, wird dieser parameterlose Konstruktor nicht mehr vom System bereitgestellt. Wenn Sie weiter einen Konstruktor ohne Parameter verwenden wollen, müssen Sie ihn nun selbst implementieren:

```
   class datum
      {
      private:
         int tag;
         int monat;
         int jahr;
      public:
         int getTag() { return tag;}
         int getMonat() { return monat;}
         int getJahr() { return jahr;}
         void set( int t, int m, int j);
         datum( int t, int m, int j);
A        datum() { set( 1, 1, 1970); }
      };
```

Listing 20.20 Ergänzung eines parameterlosen Konstruktors

Nachdem wir einen eigenen parameterlosen Konstruktor als Inline-Implementierung zu der Klasse hinzugefügt haben (A), können wir auch wieder Instanzen in der bisher verwendeten parameterlosen Form erstellen:

```
   int main()
      {
A     datum em( 1, 5, 2014); // 1. Mai
      printf( "1. Mai: %d.%d.%d\n",
              si.getTag(), si.getMonat(), si.getJahr());
B     datum ha; // Heiligabend
      ha.set( 24, 12, 2014);
      }
```

Listing 20.21 Anwendung des neuen parameterlosen Konstruktors

Ausführung des Konstruktor-Codes bei Instanziierung eines Objektes

Ich hatte Sie im vorangegangenen Kapitel darauf hingewiesen, dass es einen Grund dafür gibt, dass in C++ Variablen nicht nur am Anfang eines Codeblocks definiert werden können. An dieser Stelle sollten Sie sich noch einmal klarmachen, dass mit dem Aufruf des Konstruktors (A, B) implizit der Code ausgeführt wurde, der im Konstruktor definiert worden ist. Damit bestimmt die Abfolge der Variablendefinition also auch die Abfolge des Codeablaufs mit.

Ich möchte die generellen Anforderungen an einen Konstruktor noch einmal zusammenfassen:

Anforderungen an einen Konstruktor

Objekte müssen immer in einem konsistenten Zustand sein. Ein Konstruktor eines Objekts instanziiert das Objekt in einem konsistenten Anfangszustand, der dann im weiteren Lebenszyklus der Instanz durch die Methoden konsistent verändert werden kann.

Durch die Konstruktoren werden dem Benutzer genau vorgegebene Optionen zur Verfügung gestellt, wie ein Objekt erstellt werden kann.

Wenn ein Objekt zur Instanziierung Zusatzinformationen von außen benötigt, dann stellt es entsprechend parametrierte Konstruktoren zur Verfügung. Wenn keine solchen Informationen notwendig sind, dann genügt ein Konstruktor mit der Vorgabe »keine Parameter«.

Zum Aufruf von Konstruktoren wollen wir noch einen Blick auf die Notation werfen, da es hier leicht zu einem ganz bestimmten Fehler kommt. Bei den folgenden Notationen handelt es sich um gültige Aufrufe unserer Konstruktoren:

```
A   datum d1;
B   datum d2( 1, 4, 2015);
```

Listing 20.22 Aufruf von Konstruktoren

Zuerst wird ein parameterloser Konstruktor aufgerufen (A), anschließend wird ein Konstruktor mit drei `int`-Parametern verwendet (B).

Die jetzt folgende Konstruktion sieht nur auf den ersten Blick wie der Aufruf eines parameterlosen Konstruktors aus, ist aber kein solcher und führt zu einer Warnung. Es handelt sich formal um die Deklaration einer parameterlosen Funktion mit dem Namen `d` und dem Ergebnistyp `datum`, die der Compiler mit einer Fehlermeldung quittiert.

```
datum d();
```

20.4.6 Destruktoren

Der Destruktor ist das Gegenstück zum Konstruktor. Er wird aufgerufen, wenn ein Objekt zerstört wird. Im Destruktor können Aufräumarbeiten wie die Freigabe der vom Objekt belegten Ressourcen vorgenommen werden. Dies betrifft insbesondere die Freigabe von allokiertem Speicher.

Jede Klasse kann maximal einen Destruktor haben. Der Destruktor hat wie der Konstruktor keinen Rückgabewert, und er ist immer parameterlos. Der Destruktor hat den Namen der Klasse, angeführt von einer vorangestellten Tilde. Der Destruktor kann nicht explizit aufgerufen werden, er wird vom System aufgerufen, wenn eine Instanz abgebaut wird.

Die Klasse `datum` benötigt keine besonderen Aufräumarbeiten, aber zu Demonstrationszwecken erstellen wir hier einen funktionslosen Destruktor:

```
   class datum
      {
      private:
         int tag;
         int monat;
         int jahr;
      public:
         int getTag() { return tag;}
         int getMonat() { return monat;}
```

```
        int getJahr() { return jahr;}
        void set( int t, int m, int j);
        datum( int t, int m, int j);
A       ~datum() {}
    };
```

Listing 20.23 Erweiterung der Klasse um einen Destruktor

Der Destruktor (A) hat in dieser Klasse keine Aufräumarbeiten zu erledigen, er bleibt leer. In diesem Fall könnte der Destruktor komplett entfallen, Sie werden aber noch Klassen kennenlernen, bei denen der Destruktor wichtig ist.

20.5 Instanziierung von Klassen

In den vorangegangenen Abschnitten haben Sie die Konstruktion und Destruktion von Objekten kennengelernt. Ich möchte in diesem Zusammenhang die Instanziierung in C++ mit Ihnen noch einmal genauer betrachten. Dazu werden wir die verschiedenen Formen der Variablenanlage in C durchgehen und parallel dazu die Situation bei der Instanziierung von Objekten in C++ vergleichen.

In C können Variablen auf drei Arten angelegt werden:

- automatisch
- statisch
- dynamisch

In C++ existieren die genannten Methoden weiter. Sie haben aber schon gesehen, dass mit den Konstruktoren und Destruktoren bei Erzeugung und Abbau von Instanzen zusätzlicher Code ausgeführt wird. Die einzelnen Abläufe im Lebenszyklus sind damit in C++ etwas komplexer als in C.

20.5.1 Automatische Variablen in C

Automatische Variablen werden innerhalb von Blöcken angelegt. Sie haben die Lebensdauer des umschließenden Blocks:

```
    void funktion()
       {
A      int auto1;
          {
B         int auto2;
C         }
D      }
```

Listing 20.24 Automatische Variablen in C

Zuerst wird die Variable `auto1` definiert (A), es folgt die Definition von `auto2` (A). Mit dem Schließen des umgebenden Blocks endet die Lebensdauer von `auto2` (C), die Lebensdauer von `auto1` endet in (D).

20.5.2 Automatische Instanziierung in C++

Objekte einer Klasse werden automatisch instanziiert, wenn eine Variable dieser Klasse in einem Block angelegt wird. Dabei wird der Konstruktor aufgerufen, dessen Signatur zu den mitgegebenen Parametern passt. Jedes Mal, wenn der Programmablauf die Definition passiert, wird das Objekt neu instanziiert.

Automatische Objekte werden durch einen gegebenenfalls vorhandenen Destruktor beseitigt, sobald der Block verlassen wird, in dem sie definiert wurden.

Die Destruktoren laufen immer in der umgekehrten Reihenfolge der Konstruktoren ab. Was zuletzt konstruiert wurde, wird zuerst beseitigt.

```
    void funktion()
       {
A      datum d1;
B      datum d2( 1, 1, 2015);
C      }
```

Listing 20.25 Automatische Instanziierung in C++

In dem Beispiel wird zuerst der parameterlose Konstruktor von `datum` aufgerufen (A), danach wird eine Instanz von `datum` mit dem Konstruktor für drei `int`-Werte aufgerufen (B). Der automatische Aufruf der Destruktoren erfolgt zuerst für `d2` (C), danach für `d1`.

20.5.3 Statische Variablen in C

Statische Variablen werden außerhalb von Blöcken angelegt oder mithilfe des Schlüsselwortes `static` gekennzeichnet:

```
A   int statisch1;
B   static int statisch2;
    void funktion()
       {
C      static int statisch3;
       //...
       }
```

Listing 20.26 Statische Variablen in C

Außerhalb eines Blocks angelegte Variablen sind statisch (A), das Schlüsselwort `static` in (B) ist damit redundant. Innerhalb des Blocks werden statische Variablen mit dem Schlüsselwort `static` definiert (C).

20.5.4 Statische Instanziierung in C++

Da in C++ mit den Konstruktoren und Destruktoren Code zur Laufzeit implizit aufgerufen wird, müssen Sie hier unterscheiden, ob statische Objekte innerhalb oder außerhalb von Funktionen angelegt werden:

- **Anlage eines statischen Objekts außerhalb von Funktionen**
  Die Objekte werden vor dem eigentlichen Programmstart, also noch vor der Ausführung des in `main` enthaltenen Codes, instanziiert, entsprechende Konstruktoren werden ausgeführt.
- **Anlage eines statischen Objekts innerhalb von Funktionen**
  Das Objekt wird einmalig instanziiert, wenn der Programmablauf erstmalig dessen Deklaration passiert. Wenn ein statisches Objekt innerhalb einer Funktion einmal angelegt ist, bleibt es bestehen und wird auch bei erneutem Passieren des Codes nicht neu initialisiert. Es wird nicht bei Verlassen der Funktion, sondern erst bei Programmende beseitigt.
- **Beseitigung von statistischen Objekten**
  Statische Objekte werden nach dem Ende des Programms in der umgekehrten Reihenfolge der Instanziierung beseitigt. Für die Angabe zur Reihenfolge der Destruktion gelten die gleichen Regeln wie bei automatischen Objekten. Wenn die Objekte ohne Seiteneffekte arbeiten[5], sollte die Reihenfolge allerdings keine Rolle spielen.

Die Unterscheidung zeigt sich in diesem Beispiel:

```
A   static datum d1;
    void fkt()
       {
B      static datum d3;
       }

    int main()
       {
C      static datum d2;
       fkt();
       fkt();
       }
```

Listing 20.27 Statische Instanziierung in C++

Die statische Variable `d1` wird zum Programmstart initialisiert (A), `d2` wird in `main` initialisiert (C). Die Variable `d3` wird beim ersten Aufruf von `fkt()` initialisiert (B) und bleibt danach weiter bestehen. Alle statischen Variablen werden zum Programmende destruiert .

20.5.5 Dynamische Variablen in C

Dynamische Variablen werden zur Laufzeit angelegt und existieren bis zu ihrer expliziten Beseitigung:

```
    void funktion()
       {
       int *pi;
A      pi = (int*)malloc(sizeof(int));
       //...

B      free( pi );
       }
```

Listing 20.28 Dynamische Variablen in C

Der Speicher wird dynamisch allokiert (A) und nach abgeschlossener Verwendung wieder freigegeben (B). Wird der dynamisch reservierte Speicher nicht freigegeben, ist er bis zum Programmende belegt.

20.5.6 Dynamische Instanziierung in C++

In C haben wir benötigten Speicher mit den Funktionen `malloc` und `calloc` der Laufzeitbibliothek allokiert. Zum Allokieren von Klassen können diese Funktionen *nicht* verwendet werden.

Um ein Objekt dynamisch zu allokieren, reicht es nicht aus, nur den entsprechenden Speicher bereitzustellen. Es ist unbedingt notwendig, dass auch ein geeigneter Konstruktor der Klasse ausgeführt wird. Um dies sicherzustellen, wird in C++ der Operator `new` verwendet, um Klassen dynamisch zu instanziieren, also den zugehörigen Speicher bereitzustellen und den geeigneten Konstruktor aufzurufen. Die Parameter für einen geeigneten Konstruktor werden dem `new`-Operator mitgegeben.

Der `new`-Operator gibt als Ergebnis einen Zeiger auf den erzeugten Objekttyp zurück:

5 Ein Arbeiten mit Seiteneffekt wäre z. B. der gemeinsame Zugriff von Objekten auf globale Variablen.

```
A   datum* d1;
    int main()
       {
B      datum* d2;
C      d1 = new datum;
       d2 = new datum( 30, 12, 2014);
D      delete d1;
       delete d2;
       }
```

Listing 20.29 Dynamische Instanziierung in C++

Nach der Anlage von Zeigern auf entsprechende Objekte entweder global (A) oder lokal (B) erfolgen jeweils die dynamische Allokation ab (C) und die Freigabe des Speichers durch Aufruf des `delete`-Operators (D).

Mit dem Operator `delete` wird für dynamisch angelegte Klassen deren Destruktor ausgeführt.

[!] Objekte, die mit `new` instanziiert worden sind, dürfen auf keinen Fall mit `free` freigegeben werden, mit `malloc` allokierter Speicher nicht mit `delete`. Beides führt zum Programmabsturz.

20.5.7 Instanziierung von Arrays in C++

Arrays können auch in C++ auf die bereits bekannte Art und Weise angelegt werden. Um ein Array von Objekten einer Klasse zu erzeugen, muss die entsprechende Klasse einen parameterlosen Konstruktor besitzen. Ob es sich dabei um einen explizit erstellten oder einen automatisch vom System bereitgestellten Konstruktor handelt, ist egal.

```
datum ds_array[10];
```

Listing 20.30 Anlage eines statischen Arrays mit zehn Elementen

Eine Übergabe von Parametern an den Konstruktor und eine individuelle Instanziierung sind nicht möglich. Der Aufruf der Konstruktoren erfolgt mit wachsendem Index.

Ein Array von Objekten kann auch dynamisch allokiert werden. Die Anzahl der gewünschten Objekte wird auch hier in den eckigen Klammern angegeben. Ein Array von Objekten wird mit dem `delete[]`-Operator freigegeben. Es handelt sich dabei um einen eigenen Operator, der für die Freigabe eines dynamischen Arrays verwendet werden *muss*. Falls der `delete`-Operator zur Freigabe des Arrays verwendet wird, wird nur das erste Element des Arrays beseitigt, die anderen bestehen weiter.


```
    datum *dd_array;
A   dd_array = new datum[10];
    // Nutzung des Arrays
B   delete[] dd_array;
```

Listing 20.31 Dynamische Instanziierung eines Arrays in C++

Nach der Allokation des dynamischen Arrays mit zehn Elementen in (A) erfolgt die abschließende Freigabe des Arrays in (B).

20.6 Operatoren auf Klassen

Sie können auch für Klassen überladene Operatoren definieren. Das eröffnet Ihnen die Möglichkeit, viele Operationen elegant und übersichtlich zu notieren. Mit einem überladenen Operator können Sie z. B. die Anzahl von Tagen zwischen zwei Daten einfach als deren Differenz betrachten und einen entsprechenden Operator erstellen. Die Verwendung sieht folgendermaßen aus:

```
datum ha( 24, 12, 2014 );
datum zf( 26, 12, 2014 );
int diff = zf - ha;
printf( "Heiligabend und zweiter Feiertag liegen %d Tage auseinander\n"
       , diff );
```

Listing 20.32 Anwendung eines Operators auf die Klasse »datum«

Wir erhalten damit die folgende Ausgabe:

```
Heiligabend und zweiter Feiertag liegen 2 Tage auseinander
```

Den Operator, den Sie dafür brauchen, implementieren Sie mit den Ihnen bereits bekannten Mitteln:

```
A   int operator-( datum& l, datum& r)
       {
B      int tage_l = 360*l.getJahr() + 30*l.getMonat() + l.getTag();
C      int tage_r = 360*r.getJahr() + 30*r.getMonat() + r.getTag();
D      return tage_l - tage_r;
       }
```

Listing 20.33 Implementierung der Methode »operator-«

Wir implementieren die Methode `operator-`, die als Eingangsparameter die Referenzen auf zwei Datumsobjekte erhält (A). Für Datumsoperationen bietet es sich oft an,

einen Fixpunkt zu wählen. Dazu ermitteln wir für jedes der beiden Daten die Anzahl der vergangenen Tage, die von einem imaginären »nullten Januar im Jahr null« bereits vergangen sind (B, C). Aus diesen Werten berechnen wir dann die Differenz (D).

20.6.1 Friends

Im vorangegangenen Beispiel haben wir den Operator unter Verwendung der Getter-Methoden implementiert. Dies ist unvermeidlich, da die Funktion `operator-` nicht zur Klasse `datum` gehört und daher keinen Zugriff auf die privaten Datenmember hat. Nicht nur hier kann sich der umfassende Zugriffsschutz auch als lästig erweisen. In Bibliotheken von Klassen wollen wir thematisch zusammengehörigen Klassen untereinander weitergehende Zugriffsrechte einräumen. Um das zu erreichen, können wir hier bisher nur nach dem Motto »alles oder nichts« Member im öffentlichen Bereich der Klasse platzieren.

Um hier differenzierter vorzugehen, kann eine Klasse anderen Funktionen oder Klassen einen besonderen Status zuerkennen und sie zu einem »Freund« erklären. Diese Freunde (`friends`) erhalten dann den gleichen Status wie Memberfunktionen der entsprechenden Klassen und können damit auf deren private Member zugreifen. Funktionen, die auf unterschiedliche Klassen zugreifen, aber keiner der Klassen zugeordnet werden sollen, werden oft als Friend-Funktionen erstellt.

Die Freundschaftserklärung gilt nur in eine Richtung. Ich kann jemand anderen zu meinem Freund erklären und ihm damit besondere Zugriffsrechte an meinen privaten Daten einräumen. Ich kann mich aber nicht selbst zu einem Freund von jemand anderem erklären und mir dadurch besondere Zugriffsrechte an dessen Daten einräumen.

Mit der Erklärung einer Klasse oder Funktion zum Freund sollten Sie sparsam umgehen. Es gibt Situationen, in denen diese Möglichkeit sinnvoll eingesetzt werden kann, ein freigiebiger Umgang mit Freundschaften deutet aber oft auch auf eine ungünstige Modellierung hin. Wir wollen uns ansehen, wie unser Operator als Friend-Funktion implementiert wird.

Wenn unsere Klasse `datum` die Funktion `operator-` zu einem Freund erklärt, erweitert sich die Deklaration der Klasse folgendermaßen:

```
   class datum
      {
A     friend int operator-( datum& l, datum& r);
      private:
         // ...
      public:
         // ...
      };
```

Listing 20.34 Friend-Deklaration in der Klasse

Innerhalb der Klasse wird der `operator-` mit der gewünschten Signatur durch das vorangestellte Schlüsselwort zum `friend` der Klasse erklärt. Damit lässt sich der Operator etwas knapper implementieren, auch wenn die Funktionalität gleich bleibt:

```
   int operator-( datum& l, datum& r)
      {
A     int tage_l = 360*l.jahr  + 30*l.monat + l.tag;
B     int tage_r = 360*r.jahr  + 30*r.monat + r.tag;
      return tage_l - tage_r;
      }
```

Listing 20.35 Implementierung des »operator-« als friend

Durch die Freundschaft zur Klasse hat der implementierte Operator nun direkten Zugriff auf die privaten Attribute (A, B) und muss nicht mehr die Getter verwenden.

20.6.2 Operator als Methode der Klasse

Es gibt noch eine weitere Möglichkeit, einer Klasse einen Operator hinzuzufügen. In diesem Fall wird der Operator als Methode der Klasse deklariert und implementiert:

```
   class datum
      {
      //...
      public:
         //...
A        int operator-( datum& r);
};
```

Listing 20.36 Operator als Methode der Klasse

Die Deklaration des Operators erfolgt hier als öffentliche Methode (A). In diesem Fall benötigt ein zweistelliger Operator nur ein Argument. Der erste Operand ist implizit durch das Objekt gegeben, auf dem der Operator als Memberfunktion ausgeführt wird. Der zweite Operand ist als Parameter der Methode übergeben:

```
   int datum::operator-( datum& r)
      {
A     int tage_l = jahr*360 + 30*monat + tag;
B     int tage_r = r.jahr*360 + 30*r.monat + r.tag;
      return tage_l - tage_r;
      }
```

Listing 20.37 Implementierung des Operators als Methode der Klasse

Die Implementierung ähnelt den vorherigen Fällen. Der linke Operand ist das Objekt selbst, hier erfolgt direkter Zugriff (A). Der zweite Operand, der als Parameter übergeben wurde, ist ein Attribut der aktuellen Klasse, wir haben daher Zugriff auf dieses private Attribut (B).

20.7 Ein- und Ausgabe in C++

Beim Einstieg in C haben wir zu Beginn einfache Ein- und Ausgabeoperationen mit `printf` und `scanf` eingeführt und verwendet.

Wie Sie am Beispiel der `printf`-Funktion schon gesehen haben, arbeiten diese Funktionen auch unter C++ weiter.

Allerdings gibt es für C++ auch ein System zur Ein- und Ausgabe, das das Klassenkonzept und die Möglichkeit zur Überladung von Operatoren ausnutzt. Dieses System werden wir Ihnen im Folgenden kurz vorstellen, ohne in die Details zu gehen.

Um Text auf dem Bildschirm auszugeben, stellt eine Bibliothek in C++ das Objekt `cout` zur Verfügung. Das Objekt ist eine Instanz der Klasse `ostream` (für Output-Stream), die vom System zum Programmstart instanziiert wird. Um `cout` nutzen zu können, muss zuvor die Bibliothek `iostream` inkludiert worden sein, so wie für `printf` die Bibliothek `stdio` eingebunden werden muss.

Die Ausgabe wird dann über die Methoden und überladenen Operatoren des `cout`-Objekts angesprochen. Der wesentliche Operator für die Ausgabe ist der Operator `<<`. Mit diesem Operator können elementare Datentypen wie `int` oder `float` an den Ausgabestrom geleitet werden. Die Ausgabe erfolgt dann folgendermaßen:

```
   #include <iostream>
A  int a = 42;
B  std::cout << a;
```

Listing 20.38 Verwendung des <<-Operators

Nach der Einbindung der Bibliothek, die u. a. die Klasse `ostream` enthält (A), kann die Variable `a` über den überladenen Operator einfach ausgegeben werden (B):

```
42
```

Die Ausgaben nach `cout` können auch »verkettet« werden, indem mehrere Ausgaben hintereinandergestellt werden:


```
#include <iostream>
using std::cout; // Alternativ: using namespace std;
int a = 1;
char c = 'X';
char* s = "Text";
cout << "Der Wert von a ist: " << a << '\n';
cout << "Der Wert von c ist: " << c << '\n';
cout << "Der Wert von s ist: " << s << '\n';
```

Listing 20.39 Verkettung von Ausgaben

In den einzelnen Zeilen wird jeweils die Ausgabe eines Strings und einer Variablen mit `'\n'` zum Zeilenumbruch verkettet, und wir erhalten die folgende Ausgabe:

```
Der Wert von a ist: 1
Der Wert von c ist: X
Der Wert von s ist: Text
```

Oft wird die Ausgabe in einen Stream auch mit dem `endl`-Objekt beendet:

```
cout << 'Ausgabe mit Buffer-Leerung' << endl;
```

Listing 20.40 Zeilenumbruch und Buffer-Leerung

Dies bewirkt nicht nur einen Zeilenumbruch, sondern auch die sofortige Ausgabe des Streambuffers. In großer Menge verwendet, kann sich dies nachteilig auf die Performance der Ausgabe auswirken:

20.7.1 Überladen des <<-Operators

Zusätzlich zu den vom System bereitgestellten `<<`-Operatoren können wir auch entsprechende Operatoren für unsere Klasse überladen. Damit haben wir eine sehr elegante Ausgabemöglichkeit für unsere Objekte. Ausgabe-Operatoren werden häufig außerhalb der Klasse und als `friend`-Funktionen definiert.

```
   class datum
      {
A     friend ostream& operator<<( ostream& os, const datum& d);
      private:
         //...
   };
```

Listing 20.41 Deklaration des Ausgabe-Operators als friend

Rückgabewert des deklarierten <<-Operators (A) ist wieder eine Referenz auf einen `ostream` zur Verkettung der Ausgaben. Der erste Eingangsparameter der Methode ist eine Referenz auf den `ostream`, an den ausgegeben wird. Die übergebenen Datumsobjekte sollen in der Ausgabe nicht verändert werden und werden als konstante Referenz übergeben.

So implementieren Sie den entsprechenden Operator:

```
   ostream& operator<<( ostream& os, const datum& d)
      {
A     os << d.tag << '.' << d.monat << '.' << d.jahr;
B     return os;
      }
```

Listing 20.42 Implementierung des Ausgabe-Operators

Die eigentliche Ausgabe erfolgt über die bereits vorhandenen Ausgabe-Operatoren für die Basisdatentypen (A). Die Rückgabe des `ostream` (B) ist notwendig für die Verkettung von Ausgaben. Die Ausgabe eines Datumsobjekts wird damit sehr einfach:

```
datum einheit( 3, 10, 1990);
cout << "Tag der Einheit: " << einheit<< '\n';
```

Listing 20.43 Ausgabe eines Datumsobjekts

Wir erhalten dieses Ergebnis:

```
Tag der Einheit: 3.10.1990
```

20.7.2 Tastatureingabe

Wie bei der Ausgabe nutzt auch die Eingabe in C++ überladene Operatoren. Hier steht das Objekt `cin` im Zentrum, eine Instanz der Klasse `istream`. Bei der Eingabe wird der Operator >> verwendet. Wie das Zeichen für den Operator schon andeutet, ist der Ablauf der Eingabe wie bei der Ausgabe, lediglich die Flussrichtung dreht sich um.

Wollen wir z. B. einen Wert in eine `int`-Variable einlesen, verwenden wir die folgende Implementierung:

```
   float f;
A  cout << "Bitte geben Sie 'f' ein: "
B  cin >> f;
```

Listing 20.44 Einlesen von Werten mit dem Operator <<

Die Ausgabe des Textes erfolgt ohne Zeilenumbruch (A), sodass der gesamte Ablauf mit dem Einlesen des Wertes von der Tastatur in die Variable (B) auf dem Bildschirm folgendermaßen aussieht:

```
Bitte geben Sie 'f' ein: 1.7
```

Der <<-Operator verwendet Referenzen, um den Wert in die angegebene Variable einzutragen. Daher ist der in C verwendete Adress-Operator für die Eingabe hier nicht notwendig.

Auch für den >>-Operator gilt, dass die Überladungen für die elementaren Datentypen bereits existieren, z. B. ist damit folgende Eingabe möglich:

```
char name[100];
int alter;
cout << "Bitte geben Sie Ihren Namen ein: ";
cin >> name;
cout << "\nBitte geben Sie Ihr Alter ein: ";
cin >> alter;
```

Listing 20.45 Einlesen unterschiedlicher Datentypen

Wir können auch den >>-Operator für unsere eigene Klasse überladen. Als Beispiel wollen wir einen einfachen Eingabe-Operator für unsere Datumsklasse erstellen. Für den Zugriff auf die privaten Attribute unserer Klasse erklären wir auch diesen Operator zum `friend`:

```
friend istream& operator>>( istream& is, datum& d);
```

Listing 20.46 Signatur des >>-Operator zur Überladung

Die Übergabe der Klasse `datum` erfolgt als Referenz, sodass die Variable innerhalb der Methode verändert werden kann. Die Implementierung des Operators sieht dann folgendermaßen aus:

```
   istream& operator>>( istream& is,  datum& d)
      {
A     int tag, monat, jahr;
      is >> tag;
      is >> monat;
B     is >> jahr;
```

```
C     d.set(tag,monat, jahr );
D     return is;
      }
```

Listing 20.47 Implementierung des >>-Operators

Die einzelnen Werte werden zunächst in temporäre Variablen (A) eingelesen (B) und dann mittels des Setters übertragen (C). Abschließend erfolgt die Rückgabe des übergebenen `istreams` zur Verkettung (D).

Nach der Implementierung kann der Operator verwendet werden:

```
datum d;
cout << "Geben Sie Tag, Monat und Jahr ein: " << '\n';
cin >> d;
cout << "Das Datum ist: " << d << '\n';
```

Listing 20.48 Verwendung des >>-Operators

Wir erhalten damit z. B. diese Ausgabe:

```
Geben Sie Tag, Monat und Jahr ein:
15
8
2015
Das Datum ist: 15.8.2015
```

20.7.3 Dateioperationen

Auch für Dateioperationen verlässt sich C++ auf Objekte, Methoden und Operatoren. Diese ersetzen die Datenstrukturen und Funktionen, die Sie von C kennen.

Die Operatoren, die Sie gerade zur Ein- und Ausgabe kennengelernt haben, werden für Operationen auf Dateien ebenso angewandt wie für die Standardein- und -ausgabe. Dazu werde ich Ihnen im Folgenden ein Beispiel zeigen.

Bei der Ausgabe auf den Bildschirm hat uns das System das Objekt `cout` bereitgestellt. Mittels des überladenen Operators << haben wir dann die Daten zur Ausgabe an `cout` weitergegeben.

Um Text in eine Datei statt auf den Bildschirm auszugeben, benötigen wir zuerst ein Objekt, das die entsprechende Datei repräsentiert. Der Datentyp für ein solches Dateiobjekt ist `ofstream` (eine Abkürzung für *Output File Stream*).

Ein solches Objekt können wir nach Einbinden der entsprechenden Header-Datei instanziieren:


```
A   #include <fstream>
    using namespace std;
    int main()
       {

B      ofstream datei( "datum.txt" );

       datum d1( 1, 1, 2015 );
C      datei << d1 << endl;

D      datei.close();
       }
```

In dem Beispiel wird der passende Header inkludiert (A) und ein Objekt `datei` vom Typ `ofstream` erzeugt. Der Name der Datei wird dem Konstruktor übergeben. Die Datei wird im aktuellen Verzeichnis angelegt und zum Schreiben geöffnet (B).

Das Datumsobjekt, das ausgegeben werden soll, wird angelegt und mit seinem `operator<<` an die Datei weitergegeben (C). Nachdem alle Ausgaben im Programm abgeschlossen sind, wird die Datei wieder geschlossen (D).

Wenn Sie das Programm ablaufen lassen und das Programmverzeichnis öffnen, werden Sie dort die Datei *datum.txt* mit dem erwarteten Inhalt finden:

```
1.1.2015
```

Der Operator, den wir für die Bildschirmausgabe erstellt hatten, ist jetzt auch für die Dateiausgabe genutzt worden.

Die Objekte der Klasse `ofstream` schließen geöffnete Dateien selbstständig in ihrem Destruktor. Es ist allerdings gute Praxis, die Datei zu schließen, nachdem alle Ausgaben erfolgt sind. Insbesondere verhindert es Datenverlust, wenn es zu einem unerwarteten Abbruch des Programms kommt und gepufferte Daten noch nicht geschrieben worden sind. Das Pendant zu `ofstream` zum Einlesen von Dateien ist der `ifstream` (*Input File Stream*). Um eine Datei einzulesen und den Inhalt auf dem Bildschirm auszugeben, gehen Sie folgendermaßen vor:

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

    int main()
       {
       char c;
```

```
B    ifstream datei( "datei.txt" );

C    if( !datei )
        {
        cout << " Fehler!\n";
        exit( 1 );
        }

D    while( 1 )
        {
E       datei.get( c );

F       if( datei.eof() )
           break;
G       cout.put( c );
        }

H    datei.close();
     }
```

Sie inkludieren die Dateien für die *File Streams* und für die Standardausgabe (A). Danach kann die gewünschte Datei zum Lesen geöffnet werden, hier z. B. *datei.txt* (B). Vor der Verwendung einer Datei sollten Sie prüfen, ob sie erfolgreich geöffnet worden ist[6]. Das Öffnen schlägt z. B. fehl, wenn die Datei im aktuellen Verzeichnis nicht existiert (C).

Auf den ersten Blick sieht die Prüfung einer Datei der in C sehr ähnlich. Es besteht aber ein grundsätzlicher Unterschied.

In C wird an dieser Stelle mit dem Negationsoperator `!` geprüft, ob der Zeiger auf die Datenstruktur einen Wert ungleich 0 hat. Bei der Prüfung in C++ handelt es sich um einen eigens für den `ifstream` überladenen `!`-Operator, der den Erfolg der Dateioperation anzeigt.

Die Datei wird danach in einer Endlosschleife (D) Zeichen für Zeichen eingelesen (E). Dazu bietet der `ifstream` eine `get`-Methode. Das Ergebnis der Leseoperation wird in den Parameter `c` geschrieben. Der Parameter wird als Referenz übergeben, daher ist keine Übergabe der Adresse erforderlich.

Ist das Dateiende erreicht, gibt die Methode `eof` (*End of File*) des `ifstream` als Ergebnis `true` zurück, und die Schleife wird beendet (F). Die Ausgabe auf dem Bildschirm erfolgt mit `cout` (G). Da `cout` wie eine Datei behandelt wird, hat sie eine `put`-Methode,

6 Diese Prüfung habe ich oben nicht vorgenommen, um das Beispiel kompakt zu halten.

die in die »Datei« schreibt. Alternativ hätte auch die Ausgabe verwendet werden können, die Sie schon kennen. Zum Ende der Verwendung wird auch hier die Datei geschlossen (H).

Generell sind die Dateibehandlung und das darunterliegende Betriebssystem natürlich von der verwendeten Programmiersprache unabhängig. Daher bleiben die wesentlichen Mechanismen der Dateioperationen auch in C++, wie Sie es bereits kennen. Sie sollten die Elemente wiedererkennen, die Sie bereits im ersten Teil des Buches gesehen haben.

Sie finden weitere Informationen zu `ifstream` und `ofstream` in der Dokumentation Ihres Compilers. Insbesondere habe ich im ganzen Kapitel nicht davon gesprochen, wie man die Ausgaben in C++ formatieren kann. Dieses Thema wird auch im Buch nicht behandelt. Diese Art von Ausgabe hat heute auch nur noch geringe Bedeutung. Wenn Sie sie benötigen, werden Sie dazu umfangreiche Dokumentation bei Ihrem Compiler oder im Internet finden. Investieren Sie nicht zu viel Zeit in ausgefeilte Ein- und Ausgaben mit den hier vorgestellten Mitteln. Wenden Sie sich lieber den eigentlichen Algorithmen und den Benutzerschnittstellen grafischer Systeme zu.

20.8 Der this-Pointer

Bei unserer bisherigen Arbeit mit Objekten hat es ausgereicht, dass innerhalb eines Objekts auf Attribute zugegriffen werden konnte. Adressen von Objekten haben wir nur verwendet, wenn wir Objekte dynamisch erstellt oder mit Arrays gearbeitet haben. Wir können auch weiter wie in C die Adresse eines Objekts mit dem Adress-Operator `&` ermitteln:

```
datum d;
datum *zeiger;
zeiger = &d;
```

Listing 20.49 Ermittlung der Adresse eines Elements in C und C++

Für diese Operation müssen wir allerdings einen expliziten Zugriff auf das Objekt haben, im oben angegebenen Beispiel die Instanz in der Variablen `d`.

Innerhalb der Memberfunktion eines Objekts befinden wir uns aber praktisch im »Inneren« des Objekts und haben keinen expliziten Namen, auf den wir zugreifen können.

Um von hier einen Zugriff auf die Adresse der »zugehörigen« Instanz der Klasse zu bekommen, bietet C++ den sogenannten `this`-Pointer.

Der this-Pointer bekommt eine besondere Bedeutung, wenn Objekte untereinander ihre Adressen übermitteln müssen, z. B. weil sie Querverweise untereinander einrichten, um Datenstrukturen wie Listen aufzubauen. Eine weitere Verwendung ist das folgende Konstrukt:

```
void datum::ausgabe()
   {
   cout << *this;
   }
```

Listing 20.50 Verwendung des this-Pointers

Hier erfolgt der Aufruf des Ausgabe-Operators aus der eigenen Klasse mit einem dereferenzierten this-Pointer, der als Referenz an den Operator übergeben wird.

20.9 Beispiele

Sie haben nun die Grundlagen der objektorientierten Entwicklung kennengelernt. Die besonderen Vorteile der Objektorientierung kommen zum Tragen, wenn größere Programme entstehen und Objekte wiederverwendet werden können. Wir bearbeiten ein Beispiel, in dem Sie die gelernten Vorgehensweisen anwenden müssen. Das Beispiel wird dann im folgenden Kapitel wieder aufgegriffen, sodass Sie dort bereits davon profitieren können.

20.9.1 Menge

Mit dem Datentyp *Menge* wollen wir einen Datentyp implementieren, den es in vielen Programmiersprachen bereits als Basisdatentyp gibt. Um eine sinnvolle Implementierung zu ermöglichen, werde ich Ihnen zuerst die Anforderungen an diesen Datentyp vorstellen.

Allgemein ist eine Menge eine ungeordnete Sammlung von Elementen eines bestimmten Datentyps. Jedes Element der Menge kann maximal einmal vorkommen, ist also entweder nicht oder einmal enthalten.

Für unser Beispiel wollen wir eine Menge implementieren, die Zahlen von 0 bis 255 aufnehmen kann. Unser Datentyp soll dabei die wesentlichen aus der Mengenlehre bekannten Operationen ermöglichen:

Eine Menge A, die die Zahlen 1, 3, 5 und 7 enthält, wollen wir folgendermaßen darstellen:

```
A = { 1, 3, 5, 7 }
```


Die Ausgabe erfolgt typischerweise sortiert, die Menge selbst ist aber unsortiert.

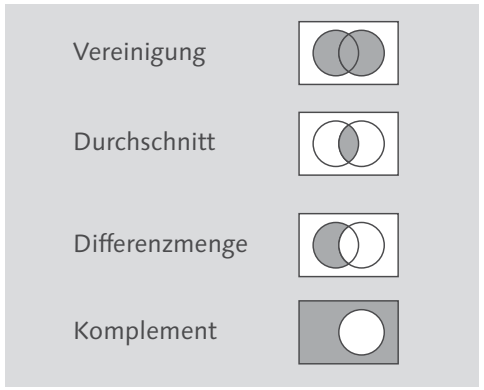


Abbildung 20.6 Darstellung der Basisoperationen

Die naheliegendste Operation ist die Vereinigung von zwei Mengen, also die Zusammenfassung der Elemente beider Mengen. Wir wollen von den beiden Mengen A und B als Beispiel ausgehen:

```
A = { 1, 2, 4 }
B = { 1, 4 , 6, 7 }
```

Wir bestimmen den Operator + als Operator für die Vereinigung. Wenn wir die beiden Mengen zur Menge C vereinigen wollen, erhalten wir also:

```
C = A + B = { 1, 2, 4, 6, 7 }
```

Die nächste Operation soll die Ermittlung des Durchschnitts sein. Zum *Durchschnitt* zweier Mengen gehören alle Elemente, die in beiden Mengen vorhanden sind. Der Durchschnitt wird *Schnittmenge* genannt. Mit dem Operator * für den Durchschnitt erhalten wir:

```
C = A * B = { 1, 4 }
```

Die *Differenzmenge* A – B ist die Menge aller Elemente, die zu A, nicht aber zu B gehören. Mit dem Operator – ergibt das als Ergebnis:

```
C = A - B = { 1, 2 }
```

Das *Komplement* ~B ist die Menge aller Elemente, die in der Grundmenge, aber nicht in B sind:

```
C = ~B = { 0, 2, 3, 5, 8, 9, 10, ..., 255 }
```

Wir wollen in unserer Klasse die folgenden Operatoren implementieren, die jeweils als Ergebnis eine neue Menge erzeugen:

Operation	Beschreibung
A + B	Erzeugt die Vereinigung der Mengen A und B.
A * B	Erzeugt den Durchschnitt der Mengen A und B.
A – B	Erzeugt die mengentheoretische Differenz »A ohne B«.
~A	Erzeugt das Komplement der Menge A.
A + e	Erzeugt die Menge, die alle Elemente aus A und zusätzlich das Element e enthält.
A – e	Erzeugt die Menge, die alle Elemente aus A, aber nicht das Element e enthält.

Tabelle 20.2 Operatoren, die eine neue Menge erzeugen

Es wird auch Operatoren geben, die eine bestehende Menge verändern:

Operation	Beschreibung
A += B	Fügt die Elemente aus B zur Menge A hinzu.
A *= B	Entfernt aus A alle Elemente, die nicht zu B gehören.
A -= B	Entfernt aus A alle Elemente, die zu B gehören.
A += e	Fügt das Element e der Menge A hinzu.
A -= e	Entfernt das Element e aus der Menge A.

Tabelle 20.3 Operatoren, die eine bestehende Menge verändern

Zusätzlich gibt es Operatoren, die eine bestehende Menge prüfen und ein entsprechendes Ergebnis zurückliefern:

Operation	Beschreibung
A <= B	Prüft, ob A Teilmenge von B ist. Das Ergebnis ist 1, wenn die Teilmengenbeziehung besteht, ansonsten 0.
!A	Prüft, ob die Menge A leer ist. Bei einer leeren Menge ist das Ergebnis 1, ansonsten 0.

Tabelle 20.4 Prüfende Operatoren

Operation	Beschreibung
e < A	Prüft, ob die Menge A das Element e enthält. Kommt e in A vor, ist das Ergebnis 1, andernfalls 0.

Tabelle 20.4 Prüfende Operatoren (Forts.)

Abschließend gibt es einen Operator, mit dessen Hilfe wir eine Menge in einen Output-Stream wie `cout` ausgeben können:

Operation	Beschreibung
os << A	Gibt die Menge A auf dem `ostream` os aus.

Tabelle 20.5 Ausgabe-Operator

Wir werden unsere Klasse für die Menge `set` nennen. Da wir die Klasse später wiederverwenden wollen, achten wir auf eine saubere Aufteilung unseres Codes und erstellen drei Dateien:

```
class set
  {
  // Deklaration der Klasse
  //
  };
```

Listing 20.51 Die Datei »set.h«

Die Deklaration der Klasse `set` erfolgt in einer separaten Datei *set.h*.

```
#include "set.h"

set::set()
   {
   //...
   }
// Implementierung der weiteren Methoden
```

Listing 20.52 Die Datei »set.cpp«

Die Implementierung der inkludierten Klassendeklaration in *set.h* liegt in der Datei *set.cpp*.

```
#include "set.h"

int main()
   {
   // Hauptprogramm
   }
```

Listing 20.53 Die Datei »test.cpp«

In *test.cpp* verwalten wir das Hauptprogramm und den Testrahmen für die Klasse. Auch diese Datei inkludiert die Klassendeklaration in *set.h*.

Nachdem die Aufteilung der Dateien feststeht, müssen wir die interne Repräsentation der Daten festlegen. Wir wollen unsere Menge intern als ein Array vorzeichenloser Zeichen (`unsigned char`) repräsentieren. Jeweils ein einzelnes Bit an der entsprechenden Bitposition soll anzeigen, ob eine bestimmte Zahl Element der Menge ist oder nicht. Um die Zahlen von 0 bis 255 als Elemente der Menge zu verwalten, genügt damit ein Array mit 32 Zeichen:

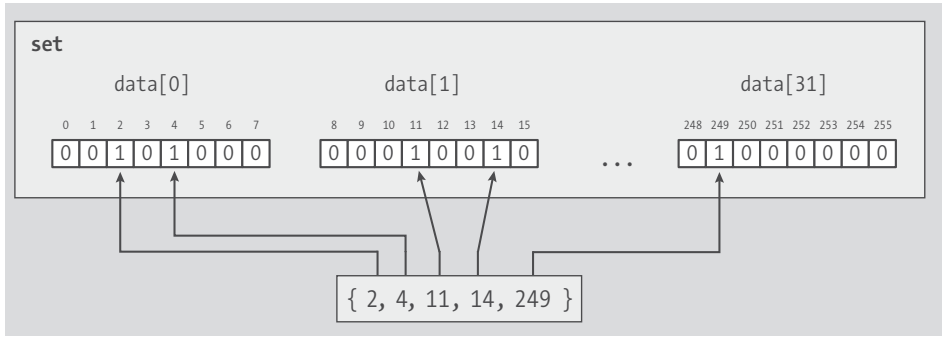


Abbildung 20.7 Interne Repräsentation der Daten

In Abbildung 20.7 sind für die fünf Elemente der Menge die fünf korrespondierenden Bits in dem Array gesetzt.

Die Klassendeklaration für die Menge lautet damit folgendermaßen:

```
class  set
   {
   private:
      unsigned char data[32];
   };
```

Listing 20.54 Deklaration der Klasse »set«

Wir werden alle Operatoren als `friend`-Funktionen implementieren. Die entsprechenden Deklarationen nehmen wir in die Klasse mit auf.

In der Klassendeklaration sind nun die Operatoren enthalten, die ein neues `set` erzeugen:

```
class  set
   {
   friend set operator+( const set& s1, const set& s2 );
   friend set operator-( const set& s1, const set& s2 );
   friend set operator*( const set& s1, const set& s2 );
   friend set operator~( const set& s );
   friend set operator+( const set& s, const int e );
   friend set operator-( const set& s, const int e );
   //...
   };
```

Listing 20.55 Erweiterte Klassendeklaration

Die Operatoren, die eine neue Menge erzeugen, haben als Rückgabetyp ein Objekt des Datentyps `set`. Wir übergeben die Parameter an die Operatoren als konstante Referenzen. So kann der Nutzer der Klasse mit einem Blick in die Klassendeklaration erkennen, dass die per Referenz übergebenen Operanden nicht manipuliert werden.

Die Operatoren wie der Operator +=, die eine Operation mit einer Zuweisung koppeln, geben keine neue Menge zurück, sondern eine Referenz auf den veränderten Operanden. Bei diesen Operatoren wird nur der zweite Operand als konstante Referenz übergeben, da wir den ersten Operanden ja ausdrücklich verändern wollen.

Die prüfenden Operatoren geben jeweils einen Wert vom Typ `int` als Prüfergebnis zurück. Auch hier sollen die Operanden nicht verändert werden und werden entweder als Kopie oder als konstante Referenz übergeben.

```
   class set
      {
A     // .. Erster Teil der Klassendeklaration
      friend set& operator+=( set& s1, const set& s2 );
      friend set& operator-=( set& s1, const set& s2 );
      friend set& operator*=( set& s1, const set& s2 );
      friend set& operator+=( set& s, const int e );
      friend set& operator-=( set& s, const int e );
B     friend int operator<= ( const set& s1, const set& s2 );
      friend int operator< ( int e, const set& s2 );
      friend int operator! ( const set& s );
```

```
C     friend ostream& operator<< ( ostream& os, set s );
      private:
D        unsigned char data[32];
      public:
E        set();
      };
```

Listing 20.56 Vollständige Deklaration der Klasse »set«

Die vollständige Klasse enthält damit die Deklaration für die Operatoren, die den linken Operanden verändern (A), die prüfenden Operatoren (B), den Ausgabe-Operator (C), das Array zur Speicherung der Daten (D) sowie den Konstruktor (E).

Implementierung

Die eigentliche Implementierung startet mit dem Konstruktor. Im Konstruktor müssen wir dafür sorgen, dass eine neu instanziierte Menge der leeren Menge entspricht, also alle Bits des zur Datenspeicherung verwendeten Arrays auf 0 gesetzt sind:

```
set::set()
   {
   int i;
   for( i = 0; i < 32; i++ )
      data[i] = 0;
   }
```

Listing 20.57 Implementierung des Konstruktors von »set«

Dies lässt sich mit einem einfachen Schleifendurchlauf erledigen. Im Anschluss an den Konstruktor wollen wir unseren ersten Operator implementieren. Dazu wählen wir den Operator +=, der die Elemente aus B zur Menge A hinzufügt. Mit der gewählten Datenrepräsentation sieht die Operation schematisch folgendermaßen aus:

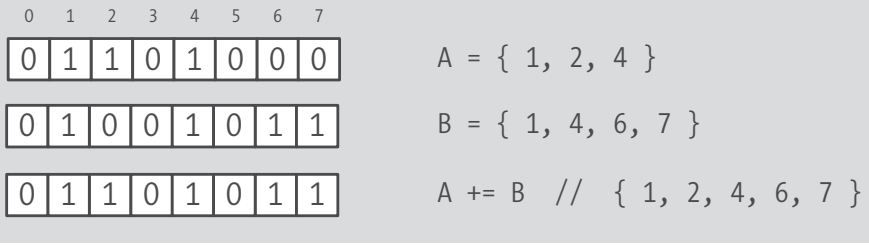


Abbildung 20.8 Verknüpfung zweier Mengen

Es handelt sich um eine simple Oder-Verknüpfung der Array-Elemente, die wir nun implementieren:

```
      set& operator+= ( set& s1, const set& s2 )
         {
         int i;
         for( i = 0; i < 32; i++ )
A           s1.data[i] |= s2.data[i];
         return s1;
         }
```

Listing 20.58 Implementierung des Operators »+=«

Zur Verknüpfung muss unser Operator in dem Array von s1 zusätzlich zu den dort bereits gesetzten Bits diejenigen setzen, die im Array von s2 gesetzt sind. Dies erreichen wir mit einem bitweisen Oder-Operator (A):

Wir haben nun als ersten Operator den Operator += implementiert. Oft kann man weitere Operatoren auf der Basis bereits implementierter leicht umsetzen. So werden wir hier mit dem Operator + verfahren. Wir werden den Operator += jetzt verwenden, um die Vereinigung von Mengen mit operator+ einfach zu implementieren.

```
      set operator+ ( const set& s1, const set& s2 )
         {
A        set r;
B        r = s1;
C        r += s2;
D        return r;
         }
```

Listing 20.59 Implementierung des Operators »+«

Die Vereinigung zweier Mengen erzeugt als Ergebnis eine neue Menge. Innerhalb unseres Operators instanziieren wir daher zuerst die neue Menge, die wir nachher als Ergebnis zurückgeben werden (A), und weisen ihr die Werte des linken Operanden zu (B). Danach besteht der Rest der Implementierung nur noch aus der Anwendung des bereits umgesetzten Operators += (C) und der Rückgabe des Ergebnisses (D).

Die Operation A – B entfernt aus A alle Elemente, die zu B gehören. Die Implementierung der Operationen -= und – läuft damit prinzipiell genauso ab wie bei den beiden vorherigen Operatoren – mit dem Unterschied, dass bei der Differenzbildung die Bits, die im Operanden s2 gesetzt sind, in dem linken Operanden s1 gelöscht werden müssen. Schematisch sieht die Operation so aus:

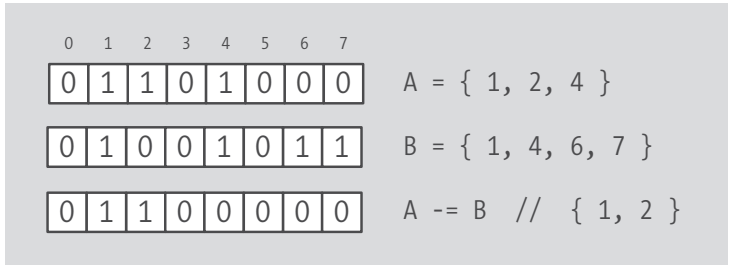


Abbildung 20.9 Differenzbildung zweier Mengen

Die Ähnlichkeit zur Vereinigung ist leicht erkennbar.

```
   set& operator-= ( set& s1, const set& s2 )
      {
      int i;
      for( i = 0; i < 32; i++ )
A        s1.data[i] &= ~s2.data[i];
      return s1;
      }
   set operator- ( const set& s1, const set& s2 )
      {
      set r;
      r = s1;
B     r -= s2;
      return r;
      }
```

Listing 20.60 Implementierung von operator-= und operator-

Die Implementierung des `operator-=` erfolgt durch eine bitweise Und-Verknüpfung (A) von `s1` mit dem Komplement von `s2`, der `operator-` wendet den `operator-=` entsprechend an (B).

Als nächsten Operator haben wir den Durchschnitt deklariert. Der Durchschnitt zweier Mengen wird gebildet, indem eine bitweise Und-Verknüpfung der beiden Daten-Arrays durchgeführt wird, sodass nur die Bits gesetzt bleiben, die in beiden Mengen gesetzt sind:

```
set& operator*= ( set& s1, const set& s2 )
   {
   int i;
   for( i = 0; i < 32; i++ )
      s1.data[i] &= s2.data[i];
   return s1;
```



```
   }
set operator* ( const set& s1, const set& s2 )
   {
   set r;
   r = s1;
   r *= s2;
   return r;
   }
```

Listing 20.61 Implementierung von »operator*=« und »operator*«

Um das Komplement einer Menge zu bilden, müssen wir alle Bits in dem Daten-Array invertieren. Auch dies ist mit den uns bereits aus C bekannten Bitoperationen kein Problem:

```
   set operator~ ( const set& s )
      {
      int i;
A     set r;
      for( i = 0; i < 32; i++ )
B        r.data[i] = ~s.data[i];
      return r;
      }
```

Listing 20.62 Implementierung von »operator~«

Dazu erstellen wir zunächst ein neues `set` (A) und übertragen dann das bitweise Komplement aller Array-Elemente in dieses `set` (B).

Die Vorgehensweise, um ein einzelnes Element zu einer Menge hinzuzufügen, unterscheidet sich etwas von den bisherigen Verfahren. Dazu müssen wir das entsprechende Bit im Array lokalisieren und dieses Bit gezielt setzen. Das Bit für die Position `e` steht im Array-Element `e/8` und dort an der Position `e%8`. Mit diesen Informationen können wir das Bit gezielt manipulieren. Am Beispiel von `e = 13` ergibt sich:

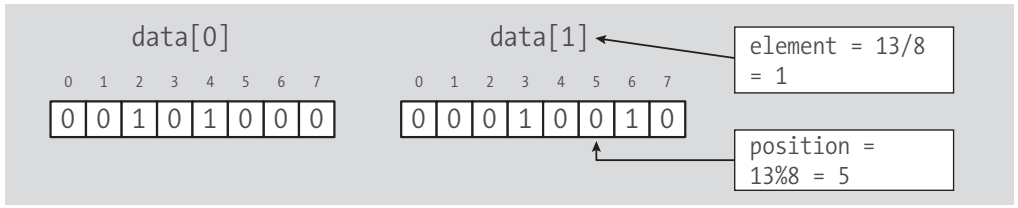


Abbildung 20.10 Berechnung der Position eines Elements

Dieses Vorgehen können wir nun algorithmisch umsetzen:

```
   set& operator+= ( set& s1, const int e )
      {
      if( ( e >= 0 ) && ( e < 256 ) )
A        s1.data[e/8] |=  ( 1 << ( e%8 ) );
      return s1;
      }
   set operator+ ( const set& s1, const int e )
      {
      set r;
      r = s1;
      r += e;
      return r;
      }
```

Listing 20.63 Hinzufügen einzelner Elemente

Um ein einzelnes Elements mit `operator+=` hinzuzufügen, wird in (A) eine bitweise Oder-Verknüpfung des betroffenen Array-Elements mit einem einzeln gesetzten Bit vorgenommen.

Wir implementieren die Methode für den `operator+` zum Hinzufügen einzelner Elemente analog zu den anderen Fällen mithilfe von `operator+=`.

Nachdem Sie das Vorgehen kennengelernt haben, um ein Element hinzuzufügen, können Sie auch leicht Elemente entfernen:

```
   set& operator-= ( set& s1, const int e )
      {
      if( ( e >= 0 ) && ( e < 256 ) )
A        s1.data[e/8] &= ~( 1 << ( e%8 ) );
      return s1;
      }
   set operator- ( const set& s1, const int e )
      {
      set r;
      r = s1;
      r -= e;
      return
```

Listing 20.64 Entfernen einzelner Elemente

Analog dem Hinzufügen wird hier in (A) das entsprechende Element des Arrays mit dem Komplement des betroffenen Bits mit einer bitweisen Und-Verknüpfung manipuliert.

Die manipulierenden Operatoren sind damit abgeschlossen, wir können nun die vergleichenden Operatoren umsetzen.

Die Operation `A <= B` prüft, ob `A` Teilmenge von `B` ist. Für diese Prüfung muss getestet werden, ob mindestens die Bits aus der einen Menge auch in der anderen gesetzt sind:

```
   int operator<= ( const set& s1, const set& s2 )
      {
      int i;
A     for( i = 0; i < 32; i++ )
         {
B        if( ( s1.data[i] & s2.data[i] ) != s1.data[i] )
C           return 0;
         }
D     return 1;
      }
```

Listing 20.65 Prüfung der Teilmengenbeziehung

Dazu werden alle Array-Elemente durchlaufen (A). Werden Bits im Array von `s1` gefunden, die im Array von `s2` nicht gesetzt sind (B), besteht keine Teilmengenbeziehung, die Prüfung wird abgebrochen (C). Wenn kein vorzeitiger Abbruch erfolgt ist, besteht eine Teilmengenbeziehung (D).

Als Nächstes wird die Prüfung auf das Vorhandensein eines einzelnen Elements umgesetzt.

Um mit dem `operator<` zu prüfen, ob ein bestimmtes Element vorhanden ist, müssen wir ermitteln, ob das entsprechende Bit gesetzt ist. Die Prüfung hat starke Ähnlichkeit mit dem Hinzufügen oder Entfernen eines Elements:

```
   int operator<( int e, const set& s )
      {
      if( ( e >= 0 ) && ( e < 256 ) )
A        return s.data[e/8] & ( 1 << ( e%8 ) );
      return 0;
      }
```

Listing 20.66 Prüfung eines enthaltenen Elements

Auch hier wird ein einzeln gesetztes Bit mit einer Und-Verknüpfung mit dem Array-Element kombiniert (A). Das Ergebnis des bitweisen Vergleichs ist bereits das Ergebnis der Prüfung.

Schließlich bleibt die Prüfung auf die leere Menge. Die leere Menge erkennen Sie daran, dass alle Felder des Arrays den Wert 0 haben müssen.

```
   int operator! ( const set& s )
      {
      int i;
A     for( i = 0; i < 32; i++ )
         {
         if( s.data[i] )
B           return 0;
         }
C     return 1;
      }
```

Listing 20.67 Prüfung auf die leere Menge

Es erfolgt eine Prüfung aller Array-Elemente (A). Wenn ein Element ungleich 0 ist, ist die Menge nicht leer, und die Prüfung ist beendet (B). Wenn kein vorzeitiger Abbruch erfolgt ist, ist die Menge leer (C).

Jetzt fehlt nur noch der Ausgabe-Operator, um unsere Menge in einen `ostream` über `cout` auszugeben. Den Ausgabe-Operator implementieren wir mit einem unserer überladenen Prüfoperatoren:

```
   ostream& operator<< ( ostream& os, const set& s )   {
      int i;
      int append;
A     os << '{';

B     for( i = 0, append = 0; i < 256; i++ )
         {
C        if( i < s )
            {
            if( append )
               os << ',';
            append = 1;
D           os << ' ' << i;
            }
         }
E     os << '}' << '\n';
      return os;
      }
```

Listing 20.68 Der Ausgabe-Operator

Die Ausgabe startet mit einer offenen, geschweiften Klammer (A). Im Kopf der Schleife über alle Elemente wird mit `append` ein Kennzeichen gesetzt, um das trennende Komma vor dem ersten Element zu unterdrücken (B). Mit dem überladenen `operator<` wird geprüft, ob `i` im `set` `s` enthalten ist (C). Ist das der Fall, erfolgt eine Ausgabe (D). Eine geschlossene, geschweifte Klammer markiert das Ende der Ausgabe (E).

Der Ausgabe-Operator erzeugt dann z. B. die folgende Ausgabe:

```
{ 2, 4, 6, 8, 10, 12}
```

Wir erstellen nun ein kleines Testprogramm für unsere neue Klasse:

```
int main()
 {
   set A;
   A += 2;
   A += 4;
   A += 6;
   A += 8;
   A += 10;
   A += 12;

   set B;
   B += 2;
   B += 4;
   B += 6;
   B += 7;
   B += 9;
   B += 11;

   cout << " A = " << A;
   cout << " B = " << B << '\n';

   cout << " A + B = " << A + B;
   cout << " A * B = " << A * B;
   cout << " A - B = " << A - B;
   }
```

Listing 20.69 Start des Testprogramms für die Mengenklasse

Mit diesem Code erhalten wir die folgende Ausgabe:

```
A = { 2, 4, 6, 8, 10, 12}
B = { 2, 4, 6, 7, 9, 11}

 A + B = { 2, 4, 6, 7, 8, 9, 10, 11, 12}
 A * B = { 2, 4, 6}
 A - B = { 8, 10, 12}
```

Wir erweitern unser Testprogramm nun um einige zusätzliche Prüfungen:

```
   int main()
      {
      // Erster Teil von main

      cout << "\n"

A     if( ! ( A*B ) )
         cout << "Der Durchschnitt von A und B ist leer\n\n";
      else
         cout << "Der Durchschnitt von A und B ist nicht leer\n\n";

      if( !( A <= B ) )
         cout << "A ist keine Teilmenge von B\n\n";
      cout << "Berechnung einiger Formeln\n";
      cout << "(A +1) * ~(B + 8) = "
           << ( A + 1 ) * ~( B + 8 );
      cout << "((A + 1) * ~(B + 8)) - 10 = "
           << ( ( A + 1 ) * ~( B + 8 ) ) - 10;
      cout << "((A + 1) * ~(B + 8) - 10) + B = \n"
           << ( ( A + 1 ) * ~( B + 8 ) - 10 ) + B;
      cout << "\n"

      if( ( A*B + 15 ) <= ( B + 15 ) )
         cout << " A*B+15 ist Teilmenge von B+15\n\n";

      A += ( B - 11 );
      cout << "A = " << A;
      A *= ( B -= 2 );
      cout << "A = " << A;
      cout << "B = " << B;

      return 0;
      }
```

Listing 20.70 Vollständiges Testprogramm

Wir verwenden in dem Programm auch den überladenen `operator!` (A), der feststellt, ob eine Menge leer ist. Dieser Operator ist in der C-Programmierung so allgegenwärtig, dass eine eventuell vorgenommene Überladung oft übersehen wird.

Mit diesen weiteren Prüfungen erhalten wir dann das folgende Ergebnis:

```
Der Durchschnitt von A und B ist nicht leer

A ist keine Teilmenge von B

Berechnung einiger Formeln
(A +1) * ~(B + 8) = { 1, 10, 12}
((A + 1) * ~(B + 8)) - 10 = { 1, 12}
((A + 1) * ~(B + 8) - 10) + B = { 1, 2, 4, 6, 7, 9, 11, 12}

A*B+15 ist Teilmenge von B+15

A = { 2, 4, 6, 7, 8, 9, 10, 12}
A = { 4, 6, 7, 9}
B = { 4, 6, 7, 9, 11}
```

Damit ist die Menge vollständig implementiert. Die Menge werden wir in der Übung des folgenden Kapitels wiederverwenden.

20.10 Aufgaben

A 20.1 Komplexe Zahlen sind eine für viele mathematische Anwendungen erforderliche Erweiterung der reellen Zahlen. Eine komplexe Zahl z besteht aus einem Real- und Imaginärteil, bei dem es sich jeweils um eine reelle Zahl handelt.

Wir notieren sie in der Form $z = (Re(z), Im(z))$. Reelle Zahlen sind spezielle komplexe Zahlen, deren Imaginärteil den Wert 0 hat.

Für die Addition und Multiplikation komplexer Zahlen (bzw. komplexer mit reellen Zahlen) bestehen die folgenden Rechenregeln:

$x + y = (Re(x) + Re(y), Im(x) + Im(y))$
$x \cdot y = (Re(x)\, Re(y) - Im(x)\, Im(y), Re(x) \cdot Im(y) + Im(x) \cdot Re(y))$

Für die Multiplikation einer reellen Zahl mit einer komplexen Zahl z gilt damit insbesondere:

$a \cdot z = (a \cdot Re(z), a \cdot Im(z))$

Den Betrag einer komplexen Zahl berechnen Sie mit der Formel

$$|z| = \sqrt{Re(z)^2 + Im(z)^2}$$

Modellieren Sie den Datentyp der komplexen Zahl als Klasse in C++. Stellen Sie für alle geläufigen Operatoren mit komplexen Zahlen bzw. mit komplexen und reellen Zahlen sinnvolle Operatoren zur Verfügung.

Schreiben Sie ein Testprogramm, das alle Funktionen dieser Klasse intensiv testet!

A 20.2 Entwerfen und implementieren Sie eine erweiterte Klasse `datum` mit mehreren Operatoren. Die Klasse soll ein Kalenderdatum, bestehend aus Tag, Monat und Jahr, verwalten und an einer von Ihnen festgelegten Schnittstelle die folgenden Funktionen bieten:

- Setzen eines bestimmten Datums
- Vergleich zweier Daten (gleich, vorher, nachher)
- Addition einer bestimmten Anzahl von Tagen zu einem Datum
- Bestimmung der Differenz (= Anzahl Tage) zwischen zwei Daten
- Berechnen des Wochentags eines Datums
- Feststellung, ob es sich um einen Feiertag handelt
- Erzeugen von Textstrings in unterschiedlichen Formaten

Die Klasse soll Schaltjahre und die korrekte Anzahl von Tagen pro Monat berücksichtigen.

Schreiben Sie ein Testprogramm, das alle Funktionen dieser Klasse intensiv testet!

A 20.3 Die Enigma ist eine Verschlüsselungsmaschine, die im 2. Weltkrieg auf deutscher Seite zur Chiffrierung und Dechiffrierung von Nachrichten und insbesondere zur Lenkung der U-Boot-Flotte eingesetzt wurde. Äußerlich ähnelt die Enigma einer Kofferschreibmaschine.

Intern besteht sie aus einem Steckbrett, drei Rotoren und einem Reflektor. Die Verschlüsselung wird durch die Verdrahtung dieser drei Grundelemente erreicht. Abbildung 20.11 zeigt den schematischen Aufbau der Enigma mit einer konkreten Verdrahtung der Bauteile.

Um eine Vorstellung vom mechanischen Aufbau der Enigma zu bekommen, müssen Sie sich die einzelnen Bauteile ausgeschnitten und an den Schmalseiten zu Walzen zusammengeklebt denken. Das Steckbrett und der Reflektor sind fest, die Rotoren dagegen drehbar montiert.

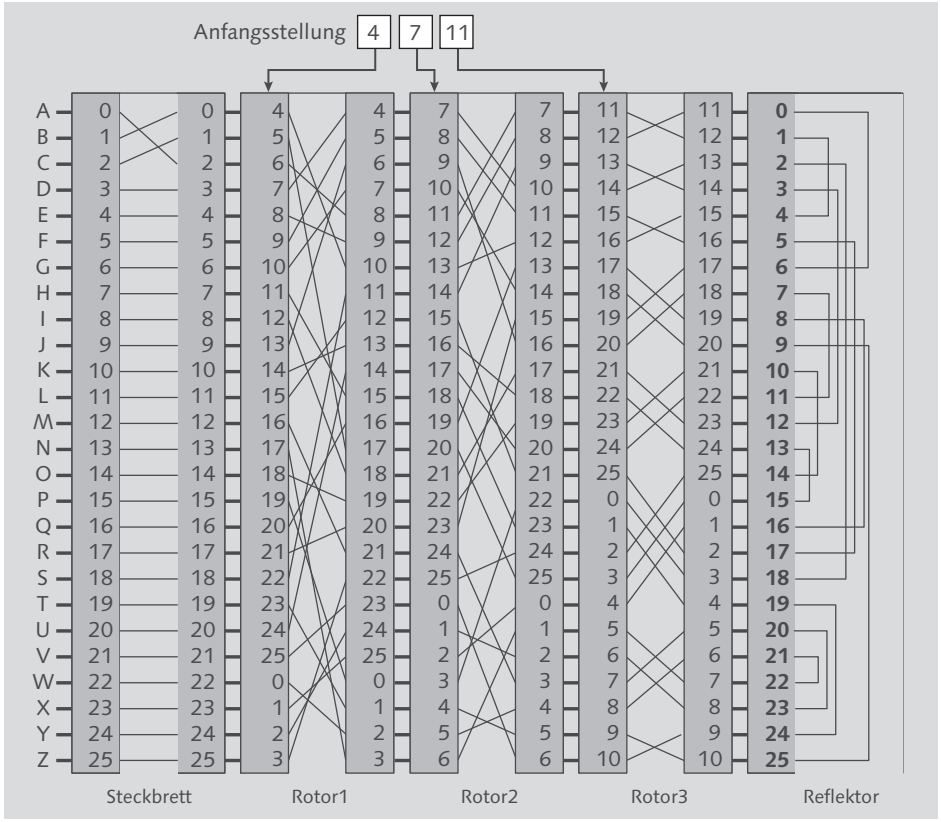


Abbildung 20.11 Schematischer Aufbau der Enigma

Bei einer bestimmten Stellung der Rotoren kann dann ein Buchstabe chiffriert bzw. dechiffriert werden, indem durch das Drücken des Buchstabens auf der Tastatur ein Stromkreis durch die Bauteile geschlossen wird. Dieser bringt dann eine Lampe mit dem Ergebnisbuchstaben zum Aufleuchten.

Die linke Buchstabenreihe in der Abbildung repräsentiert hier die Schreibmaschinentastatur und die leuchtenden Lämpchen gleichzeitig.

In der oben dargestellten Stellung wird etwa der Buchstabe A durch den Buchstaben B verschlüsselt. Um dies abzulesen, starten Sie ganz links auf der Buchstabenreihe mit einem Buchstaben, hier z. B. dem A. Von dort verfolgen Sie die die Linie über die drei Rotoren, den Reflektor und zurück. Der Buchstabe, bei dem Sie landen, ist der, auf den der Startbuchstabe verschlüsselt wird. Hier ist es das B. Umgekehrt kann B wieder zu A entschlüsselt werden. Das ist einsichtig, hier geht es ja nur auf dem gleichen Weg zurück, man landet also wieder am Ausgangspunkt.

Der Verschlüsselungs- und Entschlüsselungsprozess lief nun so ab, dass sich der linke Rotor nach jedem Drücken eines Buchstabens auf der Tastatur um eine Position weiterdrehte. In unserem Beispiel dreht sich der Rotor von 4 auf 5. Dadurch ergibt sich für den nächsten Buchstaben ein geändertes Codierungsschema. Wenn nun der erste Rotor von 25 auf 0 vorrückt, dreht auch der zweite Rotor um einen Schritt weiter. Ebenso verhält es sich mit dem dritten Rotor, der einen Schritt vorrückt, wenn der zweite Rotor wieder auf 0 springt. Dieses Prinzip kennen Sie vielleicht von mechanischen Zählwerken, wie sie früher und auch teilweise noch heute in Autos als Kilometerzähler zum Einsatz kommen.

Eine konkrete Verschlüsselung hängt also von der Verdrahtung der Bauteile und der Anfangsstellung der Rotoren ab. Wie schon beschrieben, ist die Enigma »selbstinvers«. Ein verschlüsselter Text kann also mit exakt der gleichen Prozedur, mit der er verschlüsselt wurde, auch wieder entschlüsselt werden.

Erstellen Sie eine Soft-Enigma mit folgenden Leistungsmerkmalen:

- Konfiguration der Rotoren und des Reflektors über separate Konfigurationsdateien
- interaktive Auswahl der Konfigurationsdateien für die Rotoren und den Reflektor
- interaktive Konfiguration des Steckbretts
- interaktive Eingabe der Anfangsstellung für die drei Rotoren
- Verschlüsseln und Entschlüsseln von Tastatureingaben
- Verschlüsseln und Entschlüsseln von Dateien

Entschlüsseln Sie die folgende Botschaft, die von einer Enigma mit der oben genannten Konfiguration verschlüsselt wurde:

```
PJS UGGQFP LTV
PHAZBRXE VKE YYWLBTBORYC
MGU QA BNQXXK FZL GJ TWGNQKJWR
PQV DXQWINKGGLXKP
ZTMMM GZVCAIVNCQI AGSHRY
```