Programmierung mit ANSI-C

Skript nach einer Vorlesungsausarbeitung von Martin Lowes
Überarbeitet von Carsten Damm und Henrik Brosenne

Georg-August-Universität
Göttingen

Vorwort

Der vorliegende Text ist auf Grundlage einer Vorlesungsausarbeitung von Martin Lowes entstanden, der bis zu seinem Tode 1999 viele Jahre lang Programmierkurse an der Georg-August-Universität in Göttingen durchgeführt hat. Die didaktisch gelungene Struktur und der Inhalt des Urtexts sind weitgehend erhalten geblieben ebenso wie die meisten Programmbeispiele. In die Aktualisierung, Ergänzung, Korrektur, Vereinheitlichung und das Layout des Materials wurde viel Mühe gesteckt, so dass es nun sehr gut sowohl als Begleitmaterial zu Programmierkursen als auch später zum Nachschlagen dienen kann. Ich danke Herrn Johannes Löwe, der den Text (unter Verwendung alter Quellen) einheitlich und sehr sorgfältig mit LaTeX gesetzt hat, aber auch zahlreiche inhaltliche Verbesserungen, Textvorschläge und Korrekturen einbrachte sowie eigene Programmbeispiele beisteuerte. Ich danke auch dem Institut für Numerische und Angewandte Mathematik der Universität Göttingen für die Ermöglichung dieser Arbeiten.

Jedes Kapitel entspricht durchschnittlich im Umfang etwas mehr als einer eineinhalbstündigen Vorlesung, wobei der Schwierigkeitsgrad von Kapitel zu Kapitel steigt. Lässt man einige Abschnitte aus, so kann man in einem Drei-Wochen-Kurs einen Großteil des hier gebotenen Stoffs vermitteln. Die Auslassungen können sich interessierte Teilnehmer im Selbststudium erarbeiten.

C ist eine Programmiersprache, die einer weltweiten Normierung unterliegt. Der aktuelle Standard ist ISO/IEC 9899:1999 (kurz: C99). Jedoch unterstützen nicht alle modernen C-Compiler ohne weiteres alle dort festgeschriebenen Neuerungen gegenüber älteren Versionen des Standards. Sehr weit verbreitet und von den meisten modernen Compilern sehr gut unterstützt ist dagegen der Standard ISO/IEC 9899:1990 (fast identisch zum sogenannten ANSI-Standard (C89) und darum im weiteren so genannt). Es wurde daher Wert darauf gelegt, dass alle Konzepte gründlich und im Sinne einer strengen Auslegung des ANSI-Standards vermittelt werden, ohne die Neuerungen aus C99 zu besprechen. Die noch unzureichende Compiler-Unterstützung würde Anfängern wohl manche Verwirrung bereiten. Die Beschreibung der Programmiersprache erfolgt in diesem Text weitestgehend Betriebssytem-unabhängig. Beschreibung von Compiler-Aufrufen dagegen beziehen sich ausschließlich auf den GNU C-Compiler.

Zielgruppe für den Text sind in erster Linie Studierende, die sauberes Programmieren in ANSI-C erlernen wollen (oder müssen), aber auch Autodidakten und Fortgeschrittene, die sich über einige Aspekte genauer informieren wollen. An Umfang und Präzision des Standards selbst kann und soll der Text selbstverständlich nicht heranreichen.

Die Programmbeispiele im Text sind natürlich nicht ausreichend, um wirklich Programmieren zu lernen. Sie illustrieren nur den einen oder anderen Aspekt der Programmiersprache und sind in keinem Falle Praxisbeispiele. Vielmehr muss das aufmerksame Lesen durch praktische Übungen am Computer ergänzt werden, die vom Ausprobieren der Programmbeispiele und das Herumspielen damit über die Lösung kleiner Übungsaufgaben zu wirklich interessanten und nützlichen Programmen reichen.

Anfängern sei gesagt: Verzagen Sie nicht! Aller Anfang ist schwer, aber wenn der Knoten geplatzt ist, macht C-Programmieren wirklich Spaß. Und noch ein Wort für Fortgeschrittene: Auch wenn Sie schon die eine oder andere ähnliche Programmiersprache kennen und es Ihnen am Anfang zu langsam voran geht — Unterschätzen Sie die Feinheiten und Fallstricke der Programmiersprache C nicht, es wird schon noch interessant!

Viel Erfolg!!

Carsten Damm Februar 2006

Vorwort zur zweiten Auflage

Entsprechend dem Wunsch vieler Hörer wurde für schnelles Nachschlagen ein kurzer Anhang zur Standardbibliothek eingefügt. Johannes Löwe hat das Skript nochmals durchgesehen und die zahlreichen weiteren Verbesserungsvorschläge und Korrekturen eingearbeitet. Außerdem hat er das Stichwortverzeichnis überarbeitet und ergänzt. Herzlichen Dank für diese Arbeit! Danke natürlich auch an die Teilnehmer, die die Verbesserungsvorschläge eingebracht haben.

Carsten Damm Februar 2007

Vorwort zur dritten Auflage

Kleinere Korrekturen wurden vorgenommen und einige Tippfehler beseitigt. Der Inhalt ist weitgehend unverändert, insbesondere die Abfolge und Aufteilung der Kapitel ist gleich geblieben.

Henrik Brosenne Februar 2010

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Die Geschichte von C

Die Programmiersprache C wurde 1972 von Dennis M. Ritchie entwickelt. Der Anlass war eine Rechner–unabhängige Implementation des Betriebssystems UNIX. Dieses Betriebssystem ist zu über 90 % in C geschrieben.

Viele der wichtigsten Ideen von C stammen aus der Sprache BCPL (*Basic Combined Programming Language*), die Martin Richards 1967 entwickelt hat. Diese Sprache wurde von Ken Thomson 1970 für das erste UNIX-System zur Sprache B weiterentwickelt, die schließlich zu C erweitert wurde.

Viele Jahre lang galt die erste Auflage der Sprachbeschreibung *Programmieren in C* von Brian W. Kernighan und Dennis M. Ritchies quasi als „Standard“ für C. Diese Sprachbeschreibung war aber weder vollständig noch exakt; für den Anwender reichte sie zwar aus, für einen Compilerbauer fehlte jedoch vieles. Deshalb richtete 1983 ANSI (*American National Standards Institute*) eine Kommission ein, die eine moderne, umfassende Definition von C erstellen sollte. Das Resultat, der *ANSI–Standard* oder *ANSI–C*, wurde Ende 1988 vorgelegt und ein Jahr später veröffentlicht.

Das oben erwähnte Buch von Kernighan und Ritchie erscheint in seiner zweiten Ausgabe, die den ANSI-Standard als Grundlage beschreibt, und gilt als das Buch über C:

> *Brian W. Kernighan & Dennis M. Ritchie*:
> The C programming language, Second Edition
> Prentice Hall, Englewood Cliffs N.Y. (1988)

1994 wurden die ersten Ergänzungen zum ANSI-Standard (ISO C Amendment 1) eingeführt. 1999 wurde der Standard einer Überarbeitung unterzogen. Dabei wurde auf Abwärtskompatibilität geachtet (ANSI C-Programme nach dem Standard von 1989 genügen weitestgehend auch der Neufassung) und einige von Compiler-Herstellern vorgeschlagene Erweiterungen wurden standardisiert

Der Standard von 1989 beruht in weiten Teilen auf der ursprünglichen Sprachbeschreibung von Kernighan/Ritchie. An den Grundelementen der Sprache hat der Standard nicht sehr viel geändert. Allerdings gibt es durchaus auch wesentliche Neuerungen:

- Die Syntax zur Deklaration und Definition von Funktionen wurde so erweitert, dass der Compiler eventuelle Widersprüche zwischen den Parametern in Deklaration bzw. Definition einer Funktion und den Argumenten in ihren Aufrufen erkennen kann.
- Der Standard legt fest, welchen Umfang die *Standardbibliothek* haben muss. Diese

Bibliothek enthält Funktionen zur Ein–/Ausgabe, zum Zugriff auf das Betriebssystem, zur Manipulation von Zeichenketten, usw.

Der Vorteil dieser Definition ist, dass jedes Standardprogramm, das zum Zugriff auf Betriebssystem und/oder Hardware des Rechners ausschließlich Funktionen aus dieser Bibliothek verwendet, *portabel* ist, d.h. es lässt sich auf jedem beliebigen Rechner, auf dem ein Standardcompiler mit Standardbibliothek zur Verfügung steht, compilieren.

- Die Regeln für numerisches Rechnen wurden grundlegend überarbeitet.

C gilt als höhere Programmiersprache. Es enthält die üblichen Sprachkonstrukte zur Steuerung des Kontrollflusses (Entscheidungen, Schleifen) und bietet Hilfsmittel zur Strukturierung und Modularisierung von Programmen. Gleichzeitig ist C eine maschinennahe Sprache, d.h. man kann in C, ähnlich wie in einem Assembler, Bits, Bytes und Adressen manipulieren.

Ein Hauptvorwurf der Kritiker von C ist, dass C eine zu hohe Disziplin des Programmieres voraussetzt, damit seine Programme nicht völlig unleserlich und instabil geraten. Das ist aber eine fast zwangsläufige Konsequenz der Intentionen für die Entwicklung von C: Die Zielgruppe waren zunächst Systemprogrammierer, also erfahrene Programmierer, die eine Möglichkeit erhalten sollten, auch maschinennahe Programme zum einen maschinenunabhängig und zum anderen kurz und prägnant zu formulieren.

1.2 Schritte der Programmentwicklung

Die Entwicklung eines Programms erfolgt in mehreren Schritten, unabhängig davon, mit welcher Programmiersprache man arbeitet.

1. Die Aufgabe muss sauber definiert werden. Hierzu gehört: Welche Daten stehen in welcher Form zur Verfügung? Welche Resultate möchte man haben?
2. Es muss ein *Algorithmus* zur Lösung der Aufgabe gesucht und formuliert werden.

   Allgemeine Regeln, wie man zu einem Problem einen geeigneten Algorithmus findet, kann es nicht geben; allerdings gibt es Hilfsmittel, wie man Algorithmen übersichtlich formulieren kann. Eines davon ist ein *Flussdiagramm*.

   Die Formulierung von Algorithmen erfolgt, wie die Aufgabenstellung selbst, abseits vom Rechner und auch unabhängig von der zu verwendenden Programmiersprache.
3. Der Algorithmus wird *codiert*, d.h. seine Einzelschritte werden in Anweisungen einer speziellen Programmiersprache übertragen und *erfasst*, d.h. in den Rechner eingegeben und dort z.B. auf der Festplatte gespeichert. Hierzu dienen i.a. spezielle Programme, die *Editor* genannt werden. Sie helfen dem Entwickler meist durch automatische Formatierung des Quelltextes, *Syntaxeinfärbung* – d.h. Schlüsselworte oder Kommentare werden in besonderen Farben hervorgehoben, automatische Vervollständigung von Namen oder zeigen z.B. die möglichen Parameter einer Funktion an.

   Ein Programm in dieser Form wird als *symbolisches Programm* oder auch als *Quellcode* bzw. *Quelltext* bezeichnet.
4. Das gespeicherte symbolische Programm wird mit einem *Compiler* übersetzt.

   In einem neu erfassten Programm wird der Compiler i.a. mehr oder minder viele formale Fehler finden – diese müssen zunächst mit dem Editor korrigiert werden. Die Folge editieren/übersetzen muss so lange wiederholt werden, bis das Programm keine formalen Fehler mehr enthält.

Wenn der Compiler keine Fehler entdeckt, erzeugt er ein *Objekt–Programm*.

5. Das Objekt–Programm wird *gebunden*; dieses geschieht erneut durch ein spezielles Programm, den *Linker*. Der Linker fügt das eigene Programm mit den benötigten Routinen aus den Programm–Bibliotheken zu einem *ausführbaren Programm* („Binary“) zusammen.

   Auch dabei können Fehler auftreten, nämlich dass der Linker benötigte Routinen nicht findet.

6. Das fertige Programm kann ausgeführt werden.

   Allerdings: Man kann nicht davon ausgehen, dass jedes Programm beim ersten Versuch korrekt arbeitet. Jetzt heißt es also i.a., die logischen Fehler im Quellen–Programm zu suchen, sie mit dem Editor zu korrigieren, usw. – so lange, bis das Programm korrekt arbeitet.

Manche Entwicklungssysteme koppeln die Schritte (3) und (4) oder sogar die Schritte (3) bis (5) so eng aneinander, dass sie kaum noch als Einzelschritte zu erkennen sind. In der Sache ändert das am Ablauf allerdings nichts und man darf auch in diesen Fällen nicht aus dem Auge verlieren, dass verschiedene Schritte ausgeführt werden.

1.3 Der Zeichensatz von C

Es ist wichtig zu wissen, welche Zeichen in einer Quelltextdatei stehen dürfen, damit der Compiler sie verarbeiten kann. Der ANSI-Standard schreibt folgende Zeichen vor:

Der Zeichensatz von C umfasst 91 graphische Zeichen (d.h. Zeichen, die durch Druckerschwärze auf dem Papier unmittelbar dargestellt werden können) und weitere Zeichen. Die Zeichen lassen sich in 5 Gruppen einteilen:

- 26 Großbuchstaben des (englischen) Alphabets

  ```
  A  B  C  D  E  F  G  H  I  J  K  L  M
  N  O  P  Q  R  S  T  U  V  W  X  Y  Z
  ```

- 26 Kleinbuchstaben des (englischen) Alphabets

  ```
  a  b  c  d  e  f  g  h  i  j  k  l  m
  n  o  p  q  r  s  t  u  v  w  x  y  z
  ```

- 10 Ziffern

  ```
  0  1  2  3  4  5  6  7  8  9
  ```

- 29 Sonderzeichen

  ```
  !  "  #  %  &  '  (  )  *  +  ,  -  .  /  :
  ;  <  =  >  ?  [  \  ]  ^  _  {  |  }  ~
  ```

- Weitere Zeichen, die keine *graphischen Zeichen* sind. Dazu gehören das Leerzeichen (in Texten wie diesen durch ␣ gekennzeichnet) und Steuerzeichen: zwei Tabulator-Zeichen (horizontal und vertikal), ein Seitenvorschub-Zeichen und die Zeilenende-Kennzeichnung. Diese Zeichen werden kollektiv als *white spaces* bezeichnet.

Die Zeilenende-Kennzeichnung kann implementationsabhängig aus mehreren Zeichen bestehen. UNIX/Linux-Textdateien verwenden nur das linefeed-Zeichen als Zeilenendezeichen. DOS/Windows-Textdateien benutzen dagegen eine Kombination aus linefeed und carriage return (Repositionierungszeichen). Apple-Textdateien wiederum benutzen nur das carriage return.

C-Programme können jedoch während der Ausführung auch andere Zeichen *verarbeiten*. Eine Benutzereingabe kann also durchaus auch die deutschen Umlaute oder das ß enthalten, obwohl der Quelltext diese Zeichen nicht enthalten darf. Zur Darstellung dieser Zeichen im Quelltext z.B. in Zeichenkettenkonstanten gibt es spezielle Zeichenkombinationen, die sogenannten Escapesequenzen.

1.4 Der Aufbau von C–Programmen

Ein C–Programm besteht, formal betrachtet, aus einer Ansammlung von *Funktionen*. Genau eine dieser Funktionen ist ausgezeichnet durch den Namen `main` und realisiert das *Hauptprogramm*. Alle anderen Funktionen sind *Unterprogramme* und müssen andere Namen tragen. Das kleinste Programm besteht nur aus der Funktion `main`. Fehlt sie, gibt der Linker eine Fehlermeldung aus.

Alle Funktionen sind formal völlig gleichberechtigt. Insbesondere heißt das, dass zwar alle Funktionen in beliebiger Reihenfolge angegeben werden können, nicht jedoch ineinander geschachtelt.

Das Schema eines C–Programms sieht so aus:

```
direktiven für den präprozessor
globale deklarationen

... funkt1(...) {
    lokale deklarationen
    anweisungsfolge
}

...

... funktN(...) {
    lokale deklarationen
    anweisungsfolge
}

int main(...) {
    lokale deklarationen
    anweisungsfolge
}
```

`main` ist, wie bereits gesagt, das Hauptprogramm, d.h. die Ausführung des gesamten Programms beginnt am Anfang von `main`. Die Funktionen `funkt1`, ..., `funktN` können vom Hauptprogramm oder auch von den Funktionen aufgerufen werden. Insbesondere ist auch *Rekursion* erlaubt, d.h. eine Funktion darf sich selbst aufrufen.

C unterscheidet bei Unterprogrammen formal nicht zwischen Prozeduren und Funktionen, wie etwa Pascal. Mit Funktionen sind Unterprogramme gemeint, die nach dem Aufruf einen Funktionswert zurückgeben (vergleichbar mit einer mathematischen Funktion), während Prozeduren keinen Wert zurückgeben und allein durch ihre Nebeneffekte wirken. In C gibt es nur Funktionen – wir werden allerdings sehen, dass auch C die Unterscheidung in der Sache durchaus kennt.

Sehen wir uns nun ein erstes vollständiges Programm an. Es soll einfach nur den Text

```
Unser erstes Beispiel!
```

auf der Standardausgabe ausgeben. Aussehen kann es so:

```
/*************************************************************\
 *  Unser erstes Beispiel                                    *
\*************************************************************/

#include <stdio.h>              /* Praeprozessordirektive */

/*= Hauptprogramm ========================================*/
int main(void) {
  printf("Unser erstes Beispiel!\n");    /* Textausgabe    */
  return 0;                              /* fertig         */
}
```

Quelltext 1.1: Unser Erstes Beispiel

Sehen wir uns die Details näher an:

- Das Programm beginnt mit einem *Kommentar*: Zeichenfolgen, die mit `/*` eingeleitet und mit `*/` beendet werden, ignoriert der Compiler. Es ist offensichtlich, dass Kommentare nicht geschachtelt werden können, da das erste Auftreten von `*/` den Kommentar beendet, egal wie oft `/*` bisher gefunden wurde.

  Man unterscheidet zwei Arten von Kommentaren:

  – *Kommentarblöcke* erstrecken sich über die ganze Breite der Zeilen. Jede Quelldatei sollte mit einem solchen Block beginnen, in dem ihr Inhalt kurz erläutert und in der Regel zumindest auch der Autor und das Datum der letzten Änderung genannt werden. In den weiteren Quelltextbeispielen wird der einleitende Kommentarblock nicht abgedruckt. Enthält die Quelldatei mehrere Funktionen, so sollte jede Funktion durch einen eigenen Kommentarblock eingeleitet werden. Für die Umrandungen sollte man dann Zeichen mit unterschiedlichem Schwärzungsgrad verwenden.

  – *Inline–Kommentare* dienen zur Erläuterung des Ablaufs. Es sollte nicht jede einzelne Zeile kommentiert werden, man sollte aber auch nicht zu sparsam mit Kommentaren sein. Dabei sollte man zu beschreiben versuchen, „warum“ etwas geschieht, nicht „was“ geschieht – das sieht man ohnehin! Bei den Inline–Kommentaren sollte man auf eine strikte vertikale Trennung zwischen Code und Kommentar achten.

- Die Routinen zur Ein–/Ausgabe sind, wie bereits angesprochen, nicht direkt als Bestandteil von C definiert. Die Standardbibliothek enthält jedoch unter anderem entsprechende Funktionen, zum Beispiel `printf` für die Ausgabe.

  Ohne weiteres kennt der Compiler die Funktionen der Standardbibliothek nicht, so dass sie ihm bekanntgemacht werden müssen. Da es offenbar zu mühsam (und fehleranfällig) wäre, die entsprechenden Deklarationen jeweils explizit hinzuschreiben, gehören zur Standardbibliothek eine Reihe von Standard–Headerdateien, die die entsprechenden Deklarationen enthalten.

  Der Zugriff auf diese Headerdateien erfolgt letztlich in zwei Schritten:

  1. Zunächst bearbeitet ein *Präprozessor*, dessen Funktionsumfang auch durch den Standard festgelegt ist, das Programm. Ihm gilt die Präprozessor–Direktive

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

     Der Präprozessor ersetzt diese Zeile durch den Inhalt einer Datei `stdio.h` – und das ist gerade die Datei, die die Deklaration der Funktion `printf` enthält.

Das Nummernzeichen (`#`) muss übrigens – abgesehen von eventuellen Leer– und Tabulatorzeichen – das erste Zeichen in der Zeile sein, damit der Präprozessor sich „zuständig fühlt“. Weitere *Präprozessor–Direktiven* werden wir später kennenlernen.

2. Wenn der Präprozessor seine Arbeit beendet hat, beginnt die eigentliche Übersetzung.

- Da das Programm nur aus einer Funktion, nämlich `main`, besteht, muss diese Funktion jetzt folgen.

  Vor dem Funktionsnamen steht der Typ des Funktionswertes, den die Funktion liefert. Für `main` ist festgelegt, dass der Funktionswert ganzzahlig ist; C bezeichnet das mit `int`.

  Dem Funktionsnamen folgt, in runde Klammern eingeschlossen, die Parameterliste, die die Kommunikation zwischen einer Funktion und ihrer Umwelt ermöglicht. Wenn eine Funktion, wie hier `main`, keine Parameter besitzt, ist zwischen den Klammern das Schlüsselwort `void` einzutragen.

  Achtung: In der Literatur findet man oft `main()` statt `main(void)`. Dieses entspricht *nicht* dem Standard. Das Weglassen des Funktionstyps hingegen ist erlaubt, wenn auch unschön: Immer wenn der Typ einer Funktion nicht explizit angegeben ist, ergänzt der Compiler `int`.

- Der Rumpf einer Funktion, d.h. die Anweisungen, die bei Aufruf der Funktion ausgeführt werden, wird in geschweifte Klammern einschlossen.

  Der Übersichtlichkeit halber ist es üblich, die Anweisungen innerhalb eines Rumpfes einzurücken. Editoren, die speziell zum Erfassen von Quelltexten entwickelt wurden, rücken Blöcke meist automatisch ein.

- Die erste Anweisung in unserem Beispiel ist der Aufruf der Bibliotheksfunktion `printf`. Diese Funktion schreibt die in Gänsefüßchen stehende Zeichenkette auf den Bildschirm.

  In der Zeichenkette steht die Zeichenkombination `\n`. Sie gilt in C als *ein* Zeichen und bewirkt bei ihrer Ausgabe den Übergang zu einer neuen Zeile. Es handelt sich um die Umschreibung des newline-Zeichens. Die Notwendigkeit einer solchen Umschreibung ergibt sich aus der Tatsache, dass der Compiler schließende Gänsefüßchen in der gleichen Zeile erwartet. Nicht zulässig ist es also, einen Zeilenumbruch (durch `ENTER`) direkt in die Zeichenkette einzufügen. Beim Editieren würde das so aussehen:

  ```
  printf("Unser erstes Beispiel!
  ");
  ```

  Der Compiler würde in diesem Fall eine Fehlermeldung ausgeben.

- Jede Anweisung muss durch ein Semikolon abgeschlossen werden, anders als etwa in Pascal, wo das Semikolon Anweisungen trennt.

- Die `return`–Anweisung beendet die Funktion und liefert 0 als Funktionswert an die rufende Funktion zurück. Für `main` ist die rufende Funktion das Betriebssystem.

  Verschiedene Funktionswerte von `main` können verabredet werden, um dem Betriebssystem den korrekten (0) oder fehlerhaften (> 0) Ablauf eines Programms mitzuteilen, wobei unterschiedliche Werte zur Kennzeichnung unterschiedlicher Fehler verwendet werden können. Dass der Funktionswert von `main` ganzzahlig sein muss, ist durch den Typ `int` festgelegt.

Um das Programm auszuführen muss es noch compiliert und gelinkt werden. Die Vorgehensweise dazu hängt vom Compiler und Linker ab. Im Anhang A werden die benötigten Schritte für den GCC-Compiler unter UNIX erklärt. Für andere Compiler muss die entsprechende Dokumentation konsultiert werden.

1.5 Literatur

Dieser Text kann nur eine Einführung in die Programmierung mit ANSI-C geben und es bleiben viele interessante und nützliche Aspekte unerwähnt. Ein sehr gutes deutschprachiges Lehrbuch ist [9]. Leider ist es beim Verlag vergriffen und kann derzeit nur antiquarisch erworben oder in Bibliotheken ausgeliehen werden. Auch [3] ist empfehlenswert. Der Klassiker [7] ist auch in deutscher Fassung erhältlich. Das Buch ist sicher lehrreich, kann aber wegen des dort verwendeten (zwar auch standardisierten aber weniger Typsicherheit bietenden) „C-Dialekts“ bei Anfängern einige Verwirrung stiften. Der C-Standard wird immer noch weiter entwickelt. Die aktuelle Entwurfsfassung ist derzeit unter [6] auch im Internet verfügbar, ältere Dokumente zu Standardisierungsvarianten sind über [5] zugänglich. Sehr nützlich ist das Buch [4], das den C-Standard in kommentierender Form beschreibt.

Für genauere Informationen zur Philosophie und auch zur Entstehungs- und Standardisierungsgeschichte der Programmiersprache kann der derzeitige Wikipedia-Eintrag [1] (und ggf. neuere Versionen) empfohlen werden. Weitere verlässliche und empfehlenswerte Internet-Quellen sind die in [8] gesammelten Links (sehr lehrreich ist zum Beispiel die dort erwähnte FAQ-Liste zur C-Programmierung, die in erweiterter Fassung auch als Buch erschienen ist). Diese Webseite wurde in der Newsgruppe `comp.lang.c` eine Zeit lang als nützlichste Website zur C-Programmierung betrachtet.

Zu jeder C-Implementation gehört eine Dokumentation. Diese Dokumentation ist dafür die verlässlichste Informationsquelle, auch wenn es sich um eine Implementation des ANSI-C-Standards handeln soll. Es kann z.B. Abweichungen in der mitgelieferten Standardbibliothek geben, Implementationsfehler sind möglich (sofern sie bekannt sind, wird eine gute Dokumentation sie erwähnen) und die meisten Compiler bieten in Details Erweiterungen, die über die Forderungen des Standards hinausgehen. Wer möglichst portabel programmieren will oder muss, ist allerdings gut beraten, auf die Ausnutzung solcher Erweiterungen weitestgehend zu verzichten. Auf Linux-Systemen ist normalerweise die sogenannte GNU Compiler Collection (GCC) installiert. Dies ist eigentlich eine Sammlung von Programmierwerkzeugen für C und weitere Programmiersprachen — trotzdem sprechen wir GCC hier meist als „den Compiler“ an mit dem Aufruf `gcc`. Zusammen mit der GNU C-Library `glibc` in der jeweiligen Version ist auch noch eine umfangreiche Dokumentation installierbar, die auch Auskunft über verfügbare Programmierwerkzeuge (wie zum Beispiel das in Kapitel 8 erwähnte `make`) gibt. Im Internet ist diese Dokumentation (und auch der Compiler selbst) unter [2] verfügbar.

Kapitel 2

Standarddatentypen, Konstanten und Variablen

2.1 Interne Darstellungen

Heutige Computer verfügen über Schaltelemente, die zwei stabile Zustände annehmen können. Identifiziert man diese beiden Zustände mit den Zahlen 0 und 1, so kann man *Binärzahlen*, d.h. Zahlen im Stellenwertsystem zur Basis 2, in einem Rechner speichern und verarbeiten.

Die *Dezimalzahlen*, mit denen zu rechnen wir gewohnt sind, lassen sich problemlos und eindeutig auch als Binärzahlen darstellen. Sehen wir uns ein Beispiel dafür an:

$$\begin{aligned} 109_d &= 1 \cdot 10^2 + 0 \cdot 10^1 + 9 \cdot 10^0 \\ &= 64 + 32 + 8 + 4 + 1 \\ &= 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 1101101_b \end{aligned}$$

Die dreistellige Dezimalzahl 109 ergibt bereits eine siebenstellige Binärzahl; die dreistellige Dezimalzahl 999 würde bereits eine zehnstellige Binärzahl ergeben. Da Zahlen mit so vielen Ziffern nur sehr schlecht zu lesen sind, zumal wenn sie wie Binärzahlen nur aus Nullen und Einsen bestehen, sind im Zusammenhang mit Rechnern andere Darstellungen üblich, nämlich *Oktalzahlen* (Basis 8; Ziffern: 0, ..., 7) und *Hexadezimalzahlen* (Basis 16; Ziffern: 0, ..., 9, A, ..., F). Die Umwandlung zwischen Binär– und Oktaldarstellung bzw. zwischen Binär– und Hexadezimaldarstellung ist besonders einfach: Jede Oktalziffer entspricht gerade einer Gruppe von 3 Binärziffern, jede Hexadezimalziffer gerade einer Gruppe von 4 Binärziffern. Wir sehen uns auch das an einem Beispiel an:

$$\begin{aligned} 109_d &= 1101101_b \\ &= 1_b \cdot 8^2 + 101_b \cdot 8^1 + 101_b \cdot 8^0 \\ &= 1_d \cdot 8^2 + 5_d \cdot 8^1 + 5_d \cdot 8^0 \\ &= 155_o \end{aligned}$$

$$\begin{aligned} 109_d &= 1101101_b \\ &= 110_b \cdot 16^1 + 1101_b \cdot 16^0 \\ &= 6_d \cdot 16^1 + 13_d \cdot 16^0 \\ &= 6D_h \end{aligned}$$

Verarbeitet werden ohne weiteres nie einzelne Bits, sondern stets Gruppen. Eine gebräuchliche Gruppe ist das *Byte*, das aus 8 Bits besteht.

2.2 Wertebereiche

Wir haben eben bereits gesehen, wie man Zahlen (genauer: nichtnegative ganze Zahlen) in einem Rechner darstellen kann. Diese Zahlen reichen aber nicht aus: Vielfach möchte man auch Zeichen oder reelle Zahlen verarbeiten. Dem tragen die *Standardtypen* der Programmiersprachen Rechnung. Letztlich sind sie nur Vorschriften, Bitfolgen bestimmter Länge in bestimmter Weise zu interpretieren.

Die Standardtypen, die C kennt, werden uns in den nächsten Abschnitten beschäftigen. Ein generelles Problem dabei ist, dass auf der einen Seite der Wertebereich eines Zahlentyps direkt von der Anzahl der Bits abhängt, die für ihn verwendet werden, dass auf der anderen Seite C aber unabhängig von bestimmter Hardware definiert werden sollte.

Wir sehen uns dafür ein Beispiel an: Die am weitesten verbreitete Maßeinheit ist das bereits angesprochene Byte. In ihm kann man die Zahlen $00000000_b = 0_d$ bis $11111111_b = 255_d$ speichern – ein für die Praxis viel zu kleiner Wertebereich. Zwei Bytes ergäben den Wertebereich 0 bis 65535 – auch noch nicht übermäßig viel.

Will man neben den positiven auch negative Zahlen darstellen, so benötigt man ein Bit für das Vorzeichen und erhält dann für Bytes den Wertebereich -127 (bzw. -128) bis $+127$ bzw. für zwei Bytes den Wertebereich -32767 (bzw. -32768) bis $+32767$.

Man hätte nun bei der Definition von C sagen können: In den Programmen können ganzzahlige Werte mit 1, 2 oder 4 Bytes verarbeitet werden. Das wäre aber keine zweckmäßige Definition gewesen, da sie eine optimale Implementation auf vielen Rechnern verhindert hätte:

- Nicht jeder Rechner kennt Bytes – viele Großrechner haben eine *Wortstruktur*, wobei jedes Wort aus 64 oder mehr Bits besteht, und können auf kürzere Bitfolgen gar nicht ohne weiteres zugreifen.
- Auch bei manchem Rechner, der Bytes kennt, ist es für das Rechnen in landläufigem Sinne günstiger, wenn statt einzelner Bytes Gruppen von zwei oder mehr Bytes verwendet werden.

Der ANSI-Standard für C trägt dem Rechnung, indem er keine festen Wertebereiche, sondern nur *Mindestschranken* für die Wertebereiche vorschreibt, und indem er außerdem vorschreibt, dass für die verschiedenen Typen bestimmte *benannte Konstanten* in bestimmten Standard–Headerdateien verfügbar sein müssen, die die konkreten, implementationsspezifischen Werte wiedergeben. Diese Headerdateien sind `limits.h` für die Wertebereiche der ganzzahligen Typen und `float.h` für die Gleitkommatypen.

2.3 Ganzzahlige Typen

Beginnen wir nun konkret mit dem Typ, der bislang exemplarisch behandelt wurde, nämlich dem Typ für die Beschreibung ganzer Zahlen (*integer*). Dieser Typ wird in C mit dem Schlüsselwort `int` bezeichnet.

Genau genommen bezeichnet `int` eine *Gruppe von Typen*, wie es unsere Vorüberlegungen auch bereits nahelegen:

- Neben `int` selbst gibt es `short int` (oder kurz: `short`) mit möglicherweise kleinerem Wertebereich und `long int` (oder kurz: `long`) mit möglicherweise größerem Wertebereich.

- Man hat für alle drei Wertebereiche die Wahl zwischen vorzeichenbehafteten (Schlüsselwort `signed`) und vorzeichenlosen (Schlüsselwort `unsigned`) Zahlen. Gibt man keines der beiden Schlüsselworte an, werden die Zahlen als vorzeichenbehaftet betrachtet.

In Tabelle 2.1 finden Sie die möglichen, alternativen Bezeichnungen für die verschiedenen Ganzzahl–Typen und die Mindestschranken, die der Standard für sie vorschreibt.

Typ	Minimum	Maximum
`signed char`	≤ -127	≥ 127
`unsigned char`	$= 0$	≥ 255

`signed short int`
`signed short`
`short int`
`short`
	≤ -32767	≥ 32767

`unsigned short int` `unsigned short`	$= 0$	≥ 65535

`signed int`
`signed`
`int`
	≤ -32767	≥ 32767

`unsigned int` `unsigned`	$= 0$	≥ 65535

`signed long int`
`signed long`
`long int`
`long`
	≤ -2147483647	≥ 2147483647

`unsigned long int` `unsigned long`	$= 0$	≥ 4294967295

Tabelle 2.1: Vorgeschriebene Wertebereiche für Ganzahltypen

Die im Standard festgelegten Namen der Schranken–Konstanten in `limits.h` sind

- `SHRT_MIN`, `SHRT_MAX` und `USHRT_MAX` für die `short`–Typen,
- `INT_MIN`, `INT_MAX` und `UINT_MAX` für die `int`–Typen und
- `LONG_MIN`, `LONG_MAX` und `ULONG_MAX` für die `long`–Typen.

Im Quellcode kann man Konstanten mit ganzzahligen Typen wahlweise dezimal, oktal oder hexadezimal schreiben:

- Dezimale Konstanten beginnen mit einer Ziffer ungleich 0.
- Oktale Konstanten beginnen mit einer 0.
- Hexadezimale Konstanten beginnen mit `0x` oder `0X`; dieser Kennung muss mindestens eine hexadezimale Ziffer folgen.

Die Dezimalzahl 109, für die wir oben die Umwandlung betrachtet haben, können wir in einem Programm also wahlweise so schreiben:

```
109             dezimal
0155            oktal
0x6D            hexadezimal
0X6D            hexadezimal
```

Eine dezimal geschriebene Konstante erhält den ersten möglichen Typ aus `int`, `long` und `unsigned long`; eine oktal oder hexadezimal geschriebene Konstante erhält den ersten

möglichen Typ aus `int`, `unsigned int` und `unsigned long`. Der Programmierer kann den Typ aber auch explizit festlegen, indem er einer Konstanten die Buchstaben `u` oder `U` für `unsigned`–Typen und/oder `l` oder `L` für `long`–Typen nachstellt. So ist

```
109lu
```

die dezimale Darstellung der Dezimalzahl 109 im Typ `unsigned long`.

Negative Konstanten kennt C nicht! Vorzeichen werden vielmehr stets als einstellige (unäre) Operatoren interpretiert. Da es sich bei Konstanten mit negativem Vorzeichenoperator jedoch um einen konstanten Ausdruck handelt, werden diese trotzdem vom Compiler und nicht zur Laufzeit berechnet. Aber darauf werden wir erst später zurückkommen.

2.4 Der Typ `char`

Zeichen (*character*) werden von C als Teilmenge der ganzen Zahlen behandelt; den Typ bezeichnet das Schlüsselwort `char`. Das funktioniert so: Die verfügbaren Zeichen werden durchnummeriert, beginnend bei 0. Im Rechner werden die Zeichen dargestellt, indem diese *Ordnungszahlen* gespeichert werden. Letztlich geht jede Programmiersprache so vor, nur ist in der Regel die Koppelung von Zahlen und Zeichen nicht so eng.

Wieder ergeben sich Probleme aus der Tatsache, dass C auf möglichst allen Rechnern implementiert werden können soll, auch wenn die Rechner mit unterschiedlichen Zeichensätzen arbeiten. So kann nur vorgeschrieben werden, dass bestimmte Zeichen vorhanden sein müssen, nämlich

- die 26 Großbuchstaben des (englischen) Alphabets,
- die 26 Kleinbuchstaben des (englischen) Alphabets,
- die 10 (Dezimal–)Ziffern,
- 30 druckbare Sonderzeichen wie Leerzeichen, Plus und Minus, Punkt, Komma, Klammern, usw., und
- 7 Steuerzeichen zur Steuerung von Ein–/Ausgabegeräten.

Man beachte: Die deutschen Umlaute und das ß gehören *nicht* zum Zeichensatz von ANSI–C! Weitestgehend entsprechen diese Anforderungen des Standard der Realität des *ASCII–Code* (*American Standard Code for Information Interchange*), der insgesamt 128 Zeichen umfasst:

- Die Zeichen mit den Ordnungszahlen 0 bis 31 sowie das Zeichen mit der Ordnungszahl 127 sind Steuerzeichen.
- Die Zeichen mit den Ordnungszahlen 32 bis 126 sind druckbar.

Dass an dieser Stelle keine Tabelle mit den Ordnungszahlen der verschiedenen ASCII–Zeichen zu finden ist, liegt daran, dass man sich die Ordnungszahlen der Zeichen bewusst nicht merken sollte. Durch die Zeichenkonstanten hat man in C jederzeit Zugriff auf die Ordnungszahl eines Zeichens, ohne den Wert selbst kennen zu müssen.

Zur internen Speicherung von Zeichen wird das Byte verwendet, das seinerseits über mindestens 8 Bits verfügen muss.

Obwohl ein Vorzeichen bei Zeichen nicht plausibel erscheint, unterscheidet man zwischen `signed` und `unsigned`, damit der Typ `char` kompatibel zu den `int`–Typen bleibt, deren Teilmenge er ja ist. Die Wertebereiche der beiden `char`–Typen wurden bereits in Tabelle 2.1 mit aufgelistet.

Die konkreten implementationsspezifischen Werte enthält `limits.h` in Form der Konstanten

- `CHAR_MIN` und `CHAR_MAX` für den Typ `char` ohne Zusatz,
- `SCHAR_MIN` und `SCHAR_MAX` für den Typ `char` mit dem Zusatz `signed` und
- `UCHAR_MAX` für den Typ `char` mit dem Zusatz `unsigned`.

Aus der Existenz der verschiedenen Werte kann man schließen, dass der Typ `char`, anders als die `int`–Typen, je nach Implementation vorzeichenlos oder vorzeichenbehaftet sein kann, wenn das nicht explizit durch `unsigned` oder `signed` festgelegt wird.

Zeichenkonstanten sind einzelne Zeichen, eingeschlossen in Apostrophe, etwa

```
'w'
```

Eine Zeichenkonstante repräsentiert die Ordnungszahl ihres Zeichens – welche das ist, hängt vom Zeichensatz ab, den der Rechner verwendet. Dabei ist eines noch wichtig zu erwähnen, weil es oft vergessen wird: Die Zeichenkonstante `'0'` repräsentiert in der Regel *nicht* die Ordnungszahl 0, die Zeichenkonstante `'1'` in der Regel *nicht* die Ordnungszahl 1, usw. Gesehen haben wir das für den ASCII–Code im Grunde genommen bereits: Die Ordnungszahlen der druckbaren Zeichen beginnen in ihm erst bei 32!

Einige Zeichen können nicht ohne weiteres angegeben werden. Dafür gibt es zwei Gründe:

- Einige druckbare Zeichen haben im Quellcode bestimmte Bedeutung. Von diesen haben wir den Apostroph bereits kennengelernt, der (u.a.) Zeichenkonstanten begrenzt.
- Die Steuerzeichen für Ein–/Ausgabegeräte, die C definiert, stehen auf der Tastatur nicht zur Verfügung.

Für elf solche Zeichen stellt C *Escapesequenzen* bereit. Tabelle 2.2 listet diese auf.

Zeichen	Beschreibung
`\a`	Piepen (wird nicht von allen Ausgabegeräten unterstützt)
`\b`	Backspace
`\f`	Seitenvorschub (wird nur von Druckern beachtet)
`\n`	Zeilenvorschub
`\r`	Positionierung auf Zeilenanfang
`\t`	Horizontaler Tabulator
`\v`	Vertikaler Tabulator (wird nur von Druckern beachtet)
`\'`	Apostroph, der nicht zur Begrenzung einer Zeichenkonstante dient
`\"`	Gänsefüßchen, die nicht zur Begrenzung einer Stringkonstante dienen
`\\`	Backslash
`\?`	Fragezeichen. Nur relevant, wenn man zwei unmittelbar aufeinanderfolgende Fragezeichen in einer Zeichenkettenkonstanten verwenden möchte. Hintergrund ist die selten benötigte Möglichkeit, Sonderzeichen durch eine Kombination aus zwei Fragezeichen und einem weiteren Zeichen darzustellen („Trigraphen“).

Tabelle 2.2: Escapesequenzen

2.5 Zeichenkettenkonstanten

Eine *Zeichenkettenkonstante*, der Kürze wegen gerne mit dem englischen Begriff *String* bezeichnet, ist eine Folge von beliebig vielen Zeichen, die in Gänsefüßchen eingeschlossen ist. Wir haben einen solchen String bereits im einführenden Beispiel verwendet. Strings, die nur durch „white spaces“ getrennt sind, werden vom Präprozessor zu einem einzigen String zusamengefügt.

```
printf("Dieser String ist zu lang, um ihn in einer "
       "einzelnen Zeile unterzubringen.\n");
```

Der Backslash `\` ist in einer Zeichenkettenkonstante nur in Verbindung mit einer Escapesequenz erlaubt. Zeilenumbrüche sind gar nicht erlaubt. Soll ein Zeilenumbruch in der Ausgabe erfolgen, so ist die Escapesequenz `\n` zu verwenden. Da Gänsefüßchen zur Begrenzung von Strings dienen, können diese ebenfalls nur als Escapesequenz `\"` innerhalb eines Strings auftauchen.

Achtung: `'x'` bedeutet nicht dasselbe wie `"x"`! Darauf werden wir in anderem Zusammenhang noch zurückkommen.

2.6 Gleitkommatypen

Zum numerischen Rechnen benötigt man *Gleitkommazahlen*. C stellt diese mit den Typen `float`, `double` und `long double` zur Verfügung. Alle drei Typen müssen mindestens den Wertebereich

$$[-10^{+37}, -10^{-37}] \cup \{0\} \cup [+10^{-37}, +10^{+37}]$$

besitzen. Zur Beschreibung von Gleitkommazahlen reicht die Angabe des Wertebereichs allerdings nicht aus; ebenso benötigt man die Angabe der *Dichte* der Zahlen. Sie kann durch den Abstand der Zahl 1 von der nächstgelegenen größeren Zahl beschrieben werden. Der Standard verlangt die minimalen Dichten

- 10^{-5} beim Typ `float` und
- 10^{-9} bei den Typen `double` und `long double`.

Das entspricht einer Genauigkeit von 6 bzw. 10 dezimal–Stellen. Während sich diese Angaben auf die dezimale Darstellung beziehen, werden Gleitkommazahlen intern binär verarbeitet. Bei der Umwandlung von der einen in die andere Form treten zwangsläufig Rundungsfehler auf. Die Mindestanzahl an Dezimalstellen, die bei Umwandlung in Binärdarstellung und Rückumwandlung erhalten bleibt, wird durch die Anzahl der signifikanten Stellen angegeben. Auch dafür macht der Standard ein Vorgabe.

Die konkreten Werte für die Wertebereiche der Gleitkommatypen findet man in der Datei `float.h`:

- Wertebereich: `FLT_MAX`, `DBL_MAX`, `LDBL_MAX`
- Dichte: `FLT_EPSILON`, `DBL_EPSILON`, `LDBL_EPSILON`
- Anzahl der signifikanten Stellen: `FLT_DIG`, `DBL_DIG`, `LDBL_DIG`.

Im Programm hat man prinzipiell zwei Möglichkeiten der Darstellung:

- Man kann einen Dezimalpunkt schreiben, der vor, inmitten oder hinter der Ziffernfolge stehen kann.
- Man kann einen Exponententeil angeben, der mit dem Kennbuchstaben `e` oder `E` beginnt und danach den Exponenten zur Basis 10 als ganze Zahl mit oder ohne Vorzeichen aufweist.

Außerdem kann man diese beiden Möglichkeiten auch kombinieren.

Einige Beispiele:

```
   .1        0.1        1e-1        1.0E-1
  1.         1.0        1e0         1.0E+0
 10.        10.0        1e1         1.0E+1
```

Ohne weiteres besitzen Gleitkommakonstanten den Typ `double`. Mit den nachgestellten Kennbuchstaben `f` oder `F` bzw. `l` oder `L` können aber auch Konstanten mit dem Typ `float` bzw. `long double` angegeben werden.

2.7 Aufzählungskonstanten

Aufzählungskonstanten sind Konstanten mit ganzzahligem Wert, die durch eine sogenannte `enum`–Deklaration einen Namen erhalten, mit dem sie im weiteren angesprochen werden können.

`enum` *[name]* `{` *konstantenliste* `};`

Dabei:

- Das Schlüsselwort `enum` leitet die Deklaration ein.
- Optional kann der Aufzählungstyp benannt werden. Die eckigen Klammern sollen andeuten, dass der Name *name* optional ist; sie dürfen in keinem Falle mitgeschrieben werden.
- Die Liste der Konstanten wird in geschweifte Klammern eingeschlossen. Die Elemente der Liste werden voneinander jeweils durch ein Komma getrennt. Es ist üblich, wenn auch nicht notwendig, für die Namen der Konstanten ausschließlich Großbuchstaben zu verwenden, was der Lesbarkeit des Programms dienen soll.
- Die Vereinbarung wird durch ein Semikolon abgeschlossen.

Beispiel

```
enum farbe { ROT, GELB, BLAU, GRUEN };
```

Dem ersten Namen in einer `enum`–Liste wird der Wert 0 zugeordnet, allen weiteren der um eins erhöhte Wert des Vorgängers.

Man kann die Werte jedoch auch explizit in der Form `= wert` angeben. Dann wird für Namen, für die kein Wert angegeben ist, aufsteigend weitergezählt. Beispiel:

```
enum escapes { BELL = '\a' , NEWLINE = '\n' };
enum farbe { ROT = 1, GELB = 4, BLAU, GRUEN = 2 };
```

Hier erhält `BLAU` den Wert 5, während allen anderen Namen die angegebenen Werte zugeordnet werden.

Alle Namen in einer oder verschiedenen Aufzählungsdeklarationen im selben Gültigkeitsbereich müssen sich unterscheiden. Die Werte, die verschiedenen Namen zugeordnet werden, auch innerhalb einer einzelnen Aufzählungsdeklaration, brauchen nicht verschieden zu sein. Auch negative Werte können vergeben werden. Einzige Restriktion: Die Werte müssen ganzzahlig sein.

2.8 Benannte Konstanten

Aufzählungskonstanten sind auf ganzzahlige Werte beschränkt. Aber auch für Gleitkommawerte muss man eine Möglichkeit haben, symbolische Namen zu vergeben. Erste Beispiele sind mit `FLT_MAX` oder `DBL_EPSILON` schon genannt worden. Generell sind benannte

Konstanten für den Programmierer im Alltag wichtig, um „magische Konstanten“ zu vermeiden.

Betrachten wir als Beispiel ein Programm, in dem Verkaufspreise berechnet werden. Stand in ihm an allen Stellen, an denen Mehrwertsteuer berechnet wird, explizit die Konstante 14 bzw. 1.14, so gab es zum 1.1.1993 mit der Erhöhung der Mehrwertsteuer viel Arbeit für die Programmierer: Diese Konstanten mussten alle ersetzt werden! Ein besonderes Problem dabei ist: Für jede 14 bzw. 1.14 musste separat entschieden werden, ob die 14 bzw. 1.14 die Mehrwertsteuer war oder eine andere Größe bezeichnete, zum Beispiel den Verschnitt des Materials, die nicht verändert werden durfte.

Mit Hilfe der Präprozessor–Direktive

```
#define name ersatztext
```

kann man solchen Konstanten sinnvolle Namen geben, etwa

```
#define MWST 14.0
```

In allen nachfolgenden Zeilen wird an allen Stellen, an denen *name* nicht innerhalb eines String oder als Teil eines anderen Namens steht, *name* durch *ersatztext* ersetzt. Wie bei den Aufzählungskonstanten ist es üblich, für die Namen ausschließlich Großbuchstaben zu verwenden, auch wenn der Standard das nicht vorschreibt.

Aufzählungskonstanten und durch `#define` benannte Konstanten haben manches gemeinsam. Es gibt aber (natürlich) auch Unterschiede:

- Bei `#define` muss man die Werte stets explizit angeben, während man bei Aufzählungskonstanten den Compiler zählen lassen kann. Das ist insbesondere dann interessant, wenn man nur erreichen will, dass die Konstanten verschiedene Werte repräsentieren und ggf. eine bestimmte Größenrelation zwischen ihnen besteht, die Werte selbst aber unwesentlich sind.
- Aufzählungskonstanten können nur ganzzahlige Werte zugeordnet werden, während bei `#define` beliebige Werte – und wie wir später sehen werden nicht nur Werte – angegeben werden können.

Erweitern wir dazu unser Beispiel der Mehrwertsteuer noch etwas: Benötigt man im Programm neben dem vollen Steuersatz auch den ermäßigten, so könnte (und sollte) man schreiben

```
#define MWST 14.0
#define MWST_ERM (MWST / 2)
```

statt an allen entsprechenden Stellen explizit `MWST / 2` hinzuschreiben. Der ermäßigte Steuersatz lag bislang zwar immer bei der Hälfte des vollen – nur war das eher Zufall und Gewohnheit. Der Vorteil dieser Formulierung: Für die Änderung zum 1.1.1993 brauchten nur diese beiden Zeilen durch

```
#define MWST 15.0
#define MWST_ERM 7.0
```

ersetzt und danach das Programm neu übersetzt und gebunden zu werden.

2.9 Variablen

Konstanten verändern ihren Wert zur Laufzeit eines Programms nicht. Sie besitzen außerdem nur ihren Wert und Typ, nicht jedoch einen Speicherplatz ; zumindest nicht in der Theorie, in der Praxis sieht das gelegentlich anders aus.

Variablen besitzen dagegen einen Speicherplatz und Typ und in der Regel auch einen Wert, nämlich den, der an diesem Speicherplatz steht. Es ist jederzeit möglich, diesen Wert zu ändern. Solange der Wert einer Variablen nicht gesetzt wurde, braucht sie keinen wohldefinierten Wert zu besitzen.

In C müssen alle Variablen vereinbart werden, bevor sie verwendet werden können. Gewöhnlich geschieht dieses am Anfang einer Funktion. Die Vereinbarung von Variablen hat die Form

[speicherklasse] typ namenliste;

Dabei:

- Die Angabe eines Speicherklassen–Attributs ist optional; darauf werden wir später zurückkommen.
- *typ* ist eines der Schlüsselwörter oder eine der Schlüsselwort–Kombinationen, die wir bereits kennengelernt haben, oder auch ein Aufzählungstyp. Auf weitere Möglichkeiten werden wir später zurückkommen.
- Die *namenliste* enthält, durch je ein Komma voneinander getrennt, die Namen der zu vereinbarenden Variablen.
- Die Vereinbarung wird durch ein Semikolon abgeschlossen.

Beispiele:

```
int i;
double a, b, c;
enum farbe kreide;
```

Solch eine Vereinbarung hat zwei Funktionen:

- Zum einen wird der Compiler veranlasst, Speicherplatz für die Variablen bereitzustellen.
- Zum anderen erhält der Compiler die Vorschrift, wie die Bitfolge, die sich an dem Speicherplatz einer Variablen befindet, zu interpretieren ist (als ganze Zahl mit oder ohne Vorzeichen, als Gleitkommazahl, usw.).

Die Vereinbarung einer Variablen bewirkt im Allgemeinen ohne weiteres nicht gleichzeitig einen wohldefinierten Anfangswert für die Variable. Erforderlichenfalls kann man dies aber sicherstellen, indem man die Vereinbarung um ein Gleichheitszeichen und den gewünschten Wert erweitert. Diese Form der Vereinbarung heißt *Initialisierung*. Beispiel:

```
double x = 2.5e-7;
enum farbe kreide = ROT;
int i, summe = 0, quotient = 1, j;
```

Anfangswerte müssen immer individuell zugewiesen werden; eine abkürzende Schreibweise, um mehreren Variablen denselben Anfangswert zuzuweisen, gibt es nicht.

2.10 Namen

Um verschiedene Speicherplätze und andere Objekte, die in C-Programmen eine Rolle spielen, ansprechen zu können, muss man sie mit einem *Namen* oder *Bezeichner* versehen. Dies haben wir schon in verschiedenen Zusammenhängen kennengelernt. Entsprechend wird es Zeit, dass wir uns mit den Regeln für sie beschäftigen. Zur Bildung von Namen stehen folgende Zeichen zur Verfügung:

- uneingeschränkt die 52 Buchstaben, wobei zwischen Groß– und Kleinbuchstaben strikt unterschieden wird,
- die 10 Ziffern, wobei das erste Zeichen eines Namen keine Ziffer sein darf, um eine Unterscheidung von numerischen Konstanten zu erlauben, und
- das Unterstreichungszeichen (_). Dieses Zeichen darf auch am Anfang eines Namens stehen; sollte an dieser Stelle aber für Bibliotheksfunktionen reserviert bleiben.

Namen dürfen zwar beliebig lang sein, jedoch schreibt der Standard nur vor, dass die ersten 31 Zeichen *signifikant* sein müssen. Ein Compiler darf also Namen als gleich interpretieren, die sich erst im 32. oder einem späteren Zeichen unterscheiden. Die Minimalschranke für die Anzahl signifikanter Zeichen von Namen, die über eine Quelldatei hinaus verwendet werden sollen, liegt sogar nur bei 6.

Bestimmte Zeichenfolgen, die *Schlüsselwörter* (Tabelle 2.3), sind zwar wie Namen aufgebaut, dürfen aber nur mit ihrer festgelegten Bedeutung verwendet werden.

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`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 2.3: Schlüsselwörter von C

Da auch bei den Schlüsselwörtern zwischen Groß– und Kleinbuchstaben unterschieden wird, könnte man zum Beispiel zwar `Int` oder `INT` als Namen deklarieren, nicht jedoch `int`. Es ist in C üblich, wenn auch keineswegs vorgeschrieben, für die Namen von Variablen *nur* Kleinbuchstaben und für benannte Konstanten *nur* Großbuchstaben zu verwenden.

Um die Verständlichkeit von Quelltexten zu erhöhen, sollte man bei der Namensvergabe gewissen Konventionen folgen. Dies ist umso wichtiger, je umfangreicher der Quelltext ist. In der Regel sollte man die Namen von Variablen und Konstanten so wählen, dass sie etwas über die Bedeutung ihres Wertes aussagen. Das impliziert, dass Namen häufig 4 bis 6 Zeichen lang sind, oft auch länger. Die kleinen Beispiele in diesem Text können dies nicht hinreichend illustrieren.

In der Regel findet man in gut formulierten Quelltexten nur selten Variablennamen, die nur aus ein oder zwei Buchstaben bestehen. Sinnvoll sind solche kurzen Bezeichner allerdings in folgenden Fällen:

- Bei der Realisierung mathematischer Formeln sollte man die Bezeichnungen aus den Formeln übernehmen, soweit das möglich ist.
- *Hochgradig lokale* Variablen, etwa Zählvariablen in Schleifen, haben häufig keine benennbare Bedeutung. Übliche Namen für solche Variablen sind zum Beispiel

`i, j, k`	(Lauf–)Indizes
`m, n`	Anzahlen
`c, ch`	Zeichen
`s`	Zeichenketten („Strings“)
`p, z`	Zeiger

Man sollte auch daran denken, dass die Länge eines Namen nicht allein seligmachend ist. Zumindest für den Geübten hat der Name `i` *ebenso viel* (oder wenig) Aussagekraft wie der Name `laufvariable`!

Und noch etwas sollte man beachten:

- Zu ähnliche Namen verleiten zu Schreib– und Lesefehlern.
- Die Ziffer 1 und die Buchstaben I und l verwechselt man besonders leicht, ebenso die Ziffer 0 und den Buchstaben O (und auch den Buchstaben Q).

2.11 Typnamen

Das Schlüsselwort `typedef` erlaubt die Deklaration eigener Typnamen, die dann im weiteren Programm wie die entsprechenden Schlüsselwörter verwendet werden können:

```
typedef typ name;
```

Üblich ist es, für die Namen nur Großbuchstaben zu verwenden oder ihnen `_t` anzufügen. Diese beiden Varianten werden auch von der Standardbibliothek verwendet. Beispiel:

```
typedef int INTEGER;
typedef enum farbe farbe_t;

INTEGER i = 1;
farbe_t kreide;
```

Wie man sieht, werden durch `typedef` keine neuen Typen deklariert, sondern nur bekannte Typen benannt. Entsprechend bringt `typedef` an dieser Stelle nichts wirklich Neues, auch wenn man die positiven Auswirkungen auf die Lesbarkeit der Programme nicht unterschätzen sollte. In der Standardbibliothek werden an mehreren Stellen `typedef`– Deklarationen verwendet. Beispiele findet man in Abschnitt 4.14 oder 9.6.

Kapitel 3

Beispielprogramme

Nachdem wir die Standard–Datentypen von C ausführlich besprochen haben, sollen jetzt zunächst mehr informell einige Möglichkeiten der Ein–/Ausgabe und zwei Anweisungen zur Steuerung des Programmablaufs besprochen werden, damit erste einigermaßen sinnvolle Programme entstehen können. Wir werden auf das meiste später auch noch ausführlicher zurückkommen.

3.1 Formatierte Ein–/Ausgabe

Wir haben bereits gesehen: C selbst kennt zwar keine Anweisungen zur Ein–/Ausgabe, jedoch stellt die Standardbibliothek entsprechende Funktionen zur Verfügung, die dem Compiler durch die Präprozessor–Direktive

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

am Anfang einer Quelldatei bekanntgemacht werden können bzw. müssen, wenn man Daten lesen oder schreiben will.

Die Ausgabefunktion `printf` haben wir in unserem ersten Beispiel bereits kennengelernt, wenn auch nur in extrem elementarer Form. Sie schreibt immer auf das *Standardausgabegerät*, in der Regel also auf den Bildschirm. Ihr Pendant für die Eingabe ist die Funktion `scanf`, die stets vom *Standardeingabegerät* liest, in der Regel also von der Tastatur.

Streng genommen muss man `scanf` und `printf` als Funktionen zur *formatierten Ein–/Ausgabe* bezeichnen. Darunter versteht man, dass im Zuge der Ein–/Ausgabe numerischer Werte eine Umwandlung zwischen *interner* (binärer) und *externer* Darstellung erfolgt. Wir sehen uns das an einem Beispiel an:

```
#include <stdio.h>

/*= Hauptprogramm ========================================*/
int main(void) {
  int i;

  scanf("%d", &i);                        /* Wert lesen      */
  printf("%d", i);                        /* Wert ausgeben   */
  return 0;
}
```

Quelltext 3.1: einfache formatierte Eingabe/Ausgabe

Zu den Formalien zunächst nur soviel: Die *Formatstrings* `"%d"` bei beiden Funktionen zeigen an, dass ein Wert mit dem Typ `signed int` zu lesen bzw. zu schreiben ist. Der Operator `&` bei `scanf` zeigt an, dass ein Wert in der Variablen zu speichern ist. Auf beides werden wir gleich noch näher eingehen.

Nun aber zunächst zum Ablauf von Übertragung und Umwandlung:

- `scanf` fordert Eingabe von der Tastatur an. Die Tastatur liefert (Ordnungszahlen von) Zeichen.
- Der gelesene Wert soll in einer `int`–Variablen gespeichert werden. Deshalb müssen die Zeichen interpretiert und in die entsprechende binäre Darstellung umgewandelt werden. Wie viele Bits erzeugt werden müssen, hängt nur von der Anzahl der Bytes ab, die zur Speicherung von `int`–Werten für die Rechnerarchitektur festgelegt ist und hängt nicht von der Größe des Wertes ab.
- `printf` soll einen `int`–Wert schreiben, der in Binärdarstellung vorliegt. Da der Bildschirm mit der Binärdarstellung nichts anzufangen weiß, muss sie zunächst in Zeichen umgewandelt werden.
- Die Zeichen werden auf den Bildschirm übertragen.

Tatsächlich ist der Ablauf sehr viel komplizierter. Da die zusätzlichen Arbeiten aber stets durch elementare Routinen automatisch erledigt werden, braucht uns das nicht zu interessieren.

3.2 Formatierte Ausgabe

Schauen wir uns jetzt zunächst die Ausgabefunktion `printf` näher an, weil sie etwas einfacher als die Eingabefunktion ist. Wir haben sie im ersten Abschnitt dieses Kapitels bereits verwendet.

```
printf("%d", i);
```

Zur Funktionsweise müssen aber zunächst einige Dinge geklärt werden: Funktionen erhalten grundsätzlich extrem wenig Information über die Argumente aus dem Aufruf, nämlich entweder den Wert oder die Speicheradresse. Der Typ der Argumente kann beim Aufruf nicht mitgeteilt werden.

Das ist in der Regel kein Problem, weil die Funktionen die Typen ihrer Argumente meist selbst wissen: Eine Funktion zur Berechnung einer Winkelfunktion kann zum Beispiel sinnvollerweise nur eine Gleitkommazahl als Argument erhalten; sieht man für die drei verschiedenen Gleitkommatypen unterschiedliche Funktionen vor, so weiß jede der Varianten genau, welchen Typ ihr Argument hat.

Bei den Funktionen für die Ein–/Ausgabe sieht das anders aus: Es wäre extrem lästig, wenn man für jeden Typ eine andere Funktion aufrufen müsste. Deshalb geht man anders vor. Nur das erste Argument ist festgelegt und damit den Funktionen bekannt; es muss nämlich ein String sein. Allerdings darf es nicht ein beliebiger String sein, sondern es muss ein Formatstring sein. Damit ist gemeint, dass der String die weiteren Argumente beschreibt, und zwar sowohl die Anzahl als auch die Typen der Argumente.

Jede Beschreibung eines Arguments im Formatstring wird durch ein Prozentzeichen (`%`) eingeleitet. Ihm muss ein Kennbuchstabe (oder auch eine Kennbuchstaben–Kombination) für den Typ des Wertes folgen; dazu können weitere Angaben kommen, die uns zum jetzigen Zeitpunkt aber noch nicht zu interessieren brauchen. Jede dieser Angaben wird als *Formatbeschreiber* bezeichnet.

	"Standard"–Typ	Darstellung
i	`signed int`	ganzzahlig dezimal
d		ganzzahlig dezimal
u	`unsigned int`	ganzzahlig dezimal
o		ganzzahlig oktal
x		ganzzahlig hexadezimal
X		ganzzahlig hexadezimal
f	`double`	exponentenfrei, mit oder ohne Dezimalpunkt
e	`double`	halblogarithmisch
E		
g	`double`	exponentenfrei oder halblogarithmisch
G		
c	`char`	einzelnes Zeichen
s	`char *`	Zeichenfolge
p	`void *`	Adresse

Tabelle 3.1: Kennbuchstaben für Ausgabeformate

Fassen wir zusammen: An der Anzahl der Prozentzeichen im Formatstring erkennt `printf` die Anzahl der noch folgenden Argumente. Wie im ersten Programmbeispiel auf Seite 4, kann die Anzahl auch Null sein. Anhand der Kennbuchstaben, die den Prozentzeichen folgen, erkennt `printf`, welche Typen die Argumente haben. Dabei erfolgt die Zuordnung zwischen den Formatbeschreibern und den Ausgabewerten linear. Dass die Formatbeschreiber in Anzahl und Typen den Ausgabewerten entsprechen, ist *ausschließlich* in die Verantwortung des Programmierers gestellt. Verstöße können zu „kuriosen" Ausgaben führen – oder auch zum „Absturz" des Programms.

Zurück zu den Formatbeschreibern. So wie es eben beschrieben wurde, wäre für jeden Typ ein anderer Kennbuchstabe oder eine andere Kombination von Kennbuchstaben erforderlich. Das ist aber nicht der Fall, weil bei der Übergabe `float`–Werte grundsätzlich in `double` und `short`–Werte grundsätzlich in `int` umgewandelt werden. Auch `char`–Werte werden grundsätzlich in `int` umgewandelt; der Kennbuchstabe für den Typ `char` bewirkt die Rückumwandlung von `int` nach `char` durch `printf`.

Die Kennbuchstaben sind in Abbildung 3.1 wiedergegeben. Für Argumente mit dem Typ `long int` muss dem jeweiligen `int`–Kennbuchstaben der Zusatz `l`, für Argumente mit dem Typ `long double` dem jeweiligen `double`–Kennbuchstaben der Zusatz `L` *voran*gestellt werden.

Die Anzahl der Zeichen, die zur Darstellung eines Wertes verwendet werden, richtet sich nach Standard–Vorgaben, wenn man nur die genannten Kennbuchstaben verwendet, was für uns zunächst ausreicht. Wesentlicher ist: Zeichen im Formatstring, die nicht zu einem Formatbeschreiber gehören, werden unverändert in die Ausgabe übernommen. Genau dieses haben wir in unserem ersten Beispiel genutzt.

Schauen wir uns einige Beispiel an. Vereinbart seien die Variablen

```
int i;
long l;
float f;
double d;
long double ld;
```

Dann können wir programmieren

```
printf ("Ganze Zahlen, dezimal: %d, %ld", i, l);
printf ("Ganze Zahlen, oktal: %o, %lo", i, l);
printf ("Gleitkommazahlen: %e, %E", f, d);
printf ("Gleitkommazahlen: %f, %Lf", f, ld);
```

In allen vier Beispielen erscheint zunächst der Text, dann der Wert der ersten Variablen, dann ein Komma und schließlich der Wert der zweiten Variablen. Wie die Zahlen selbst dargestellt werden? – Schauen Sie es sich am Rechner einmal an!

3.3 Formatierte Eingabe

Sehen wir uns jetzt die Eingabe durch `scanf` an. Der wesentlichste Unterschied zu `printf`: Die Funktion `printf` benötigt nur die *Werte* ihrer Argumente; ob sie Konstanten oder Variablen sind und wo sie ggf. im Speicher stehen, ist gleichgültig. Da der Name einer Variablen in C in den meisten Fällen als Bezeichnung für ihren Wert interpretiert wird, reicht es aus, die Namen der Variablen in den Aufruf von `printf` einzusetzen. Bei `scanf` ist das anders. Diese Funktion muss wissen, *wohin* sie den gelesenen Wert schreiben soll, d.h. sie benötigt nicht die Werte ihrer Argumente, sondern deren *Adressen*. Der *Adressoperator* `&`, der bereits erwähnt und verwendet wurde, sorgt gerade hierfür.

Den Adressoperator bei den Argumenten von `scanf` zu vergessen, ist einer der „beliebtesten“ Fehler – nicht nur bei Anfängern. Wir wollen uns überlegen, was dieser Fehler bewirkt, etwa wenn wir

```
int i = 0;
scanf ("%d", i);
```

schreiben. Da der Operator `&` fehlt, wird der *Wert* von `i` anstelle der *Adresse* von `i` übergeben, hier also 0. Die Funktion `scanf` wird durch den Formatstring informiert, dass ein `int`–Wert zu lesen ist, und unterstellt deshalb, dass das nächste Argument die *Adresse* einer `int`–Variablen ist. Der Versuch, 0 als Adresse zu verwenden, kann in der Regel nur ins Chaos führen. Wenn man Glück hat, überschreibt man nur Daten, die man ohnehin nicht mehr braucht. Nur – Glück hat man in der Regel nicht. Das Programm wird in der Regel mit einem *segmentation fault*, *memory fault* oder *bus error* vom Betriebssystem „gekillt“, hoffentlich ohne Auswirkungen auf andere, gleichzeitig laufende Prozesse.

Im übrigen gilt wie für `printf`: Durch den Formatstring, der als erster Parameter übergeben wird, muss der Funktion mitgeteilt werden, wie viele Argumente folgen und welche Typen diese besitzen; die Verantwortung für die Übereinstimmung von Formatstring und Argumentliste obliegt dem Programmierer. Unterschiede bestehen dabei allerdings:

- Eine eventuelle Typumwandlung wie bei `printf` ist keinesfalls möglich; ist etwa ein Zeichen zu lesen, so umfasst der Speicherplatz der entsprechenden Variablen (in der Regel) ja nur ein Byte, so dass `scanf` auch nur ein Byte an die übergebene Adresse in den Speicher schreiben darf.
- Ob die Eingabezeichen für ganze Zahlen als dezimale, oktale oder hexadezimale Darstellung des Wertes zu interpretieren sind, ist durchaus wesentlich. Offensichtlich gilt ja $11_d \neq 11_o \neq 11_h$. Ebenso ist es wesentlich, ob ein ganzzahliger Eingabewert ein Vorzeichen tragen darf oder nicht. Anders sieht es bei den Gleitkommazahlen aus: Welche der drei möglichen Darstellungen bei der Eingabe verwendet wird, ist für die interne Darstellung der Werte bedeutungslos.
- Zeichen im Formatstring, die nicht zu Formatbeschreibern gehören, müssen bei der Eingabe eingetippt werden. An die rufende Funktion werden sie nicht weitergegeben.

	erwartete Zeichenfolge	"Standard"–Typ
`i`	ganzzahlig, mit oder ohne Vorzeichen	`signed int *`
`d`	ganzzahlig dezimal, mit oder ohne Vorzeichen	
`u`	ganzzahlig dezimal	`unsigned int *`
`o`	ganzzahlig oktal	
`x`	ganzzahlig hexadezimal	
`X`	ganzzahlig hexadezimal	
`f`	Gleitkommawert, mit	`float *`
`e`	oder ohne Vorzeichen,	
`E`	mit oder ohne Dezimal-	
`g`	punkt, mit oder ohne	
`G`	Exponententeil	
`c`	einzelnes Zeichen oder Zeichenfolge	`char *`
`s`	Zeichenfolge	`char *`

Tabelle 3.2: Kennbuchstaben für Eingabeformate

Zweckmäßig ist es deshalb, die Formatbeschreiber durch ein Leerzeichen voneinander zu trennen, da man die Eingabewerte ohnehin auch durch ein Leerzeichen (Zeilenende, o.ä.) trennen muss, andere Zeichen in Eingabeformaten aber nicht zu verwenden.

Die Kennbuchstaben für die „Grundtypen" sind in Abbildung 3.2 wiedergegeben. Andere Typen müssen durch einen zweiten, vorangestellten Kennbuchstaben gekennzeichnet werden:

- `h` `short` bei ganzzahligen Werten
- `l` `long` bei ganzzahligen Werten
  `double` bei Gleitkommawerten
- `L` `long double` bei Gleitkommawerten

In den nächsten Abschnitten werden wir in erster Linie Beispiele für die Eingabe betrachten, weil sie letztlich interessanter als die Ausgabe ist. Ausgabe erweist sich, wenn man „schöne" Ausgaben erzeugen möchte, als reines „Abzählspiel".

3.4 Schleifen

Als erstes Beispiel wollen wir diese Aufgabe lösen: Ein Programm soll zehn (`int`–)Zahlen lesen und wieder schreiben.

Lösen kann man diese Aufgabe mit den bereits bekannten Sprachelementen auf zwei Weisen:

- Man kann eine Variable deklarieren und dann, abwechselnd, zehn Aufrufe von `scanf` und `printf` programmieren.
- Man kann zehn Variablen deklarieren, die man zunächst liest und dann schreibt.

Dass beides keine realistische Lösung ist, sieht man, wenn man die Aufgabe etwas verallgemeinert: Ein Programm soll zehn (`int`–)Zahlen lesen und wieder schreiben. Es soll sich ohne größeren Aufwand auf beliebige andere Anzahlen umstellen lassen.

Die Lösung bringt eine *Schleife*: Man programmiert die Ein– und Ausgabe *einer* Zahl

und bettet diese Anweisungen in ein Sprachkonstrukt ein, das dafür sorgt, dass sie mehrfach ausgeführt werden. Höhere Programmiersprachen kennen in der Regel verschiedene Schleifen–Anweisungen, die sich, von Formalismen abgesehen, darin unterscheiden, *wie* die Anzahl der erforderlichen Wiederholungen bestimmt wird.

C kennt drei Schleifenanweisungen. Als erstes Beispiel soll hier die `while`–Anweisung dienen.

```
while (bedingung) {
    anweisungsfolge
}
```

Vor jedem Schleifendurchlauf wird die Bedingung *bedingung* ausgewertet. Ist das Ergebnis *wahr* (d.h. ungleich 0), so wird die Anweisungsfolge der Schleife ausgeführt. Ist das Ergebnis dagegen *falsch* (d.h. gleich 0), so wird zu der Anweisung übergegangen, die dem *Schleifenrumpf* folgt, d.h. zu der ersten Anweisung, die nicht mehr zur Anweisungfolge der Schleife gehört.

Damit können wir die gestellte Aufgabe realisieren. Aufgrund der modifizierten Aufgabenstellung sollte auch schon klar sein, dass die Anzahl der zu kopierenden Zahlen als benannte Konstante festgelegt werden sollte. Die Lösung kann dann so aussehen:

```
#include <stdio.h>

#define ANZAHL 10               /* Anzahl Wiederholungen   */

/*= Hauptprogramm ========================================*/
int main(void) {
  int i = ANZAHL, wert;

  printf("Geben Sie die Zahlen ein:\n");

  while (i > 0) {               /* Schleife                */
    scanf("%d", &wert);         /* eine Zahl lesen         */
    printf("%d", wert);         /* gelesene Zahl schreiben */
    i = i - 1;                  /* Anzahl verringern       */
  }
  return 0;
}
```

Quelltext 3.2: Wiederholt Zahlen einlesen und ausgeben mit einer Schleife

Diese Lösung erfüllt offensichtlich auch die erweiterte Aufgabenstellung: Sollen 9, 11 oder irgendeine andere Anzahl von Zahlen gelesen und geschrieben werden, so muss nur der Wert in der `#define`–Direktive entsprechend abgeändert und dann das Programm neu übersetzt und gebunden werden.

3.5 Wertzuweisung

Auf eine neue Anweisung, die eben bereits stillschweigend verwendet wurde, muss noch kurz eingegangen werden: Die *Wertzuweisung*

```
i = i - 1;
```

Sie bewirkt, dass der Ausdruck rechts vom Gleichheitszeichen berechnet und der resultierende Wert in der Variablen links vom Gleichheitszeichen gespeichert wird. Diese Beschreibung macht bereits deutlich, dass man eine Wertzuweisung nicht mit einer Gleichung im Sinne der Mathematik verwechseln darf.

Der Ausdruck auf der rechten Seite kann wie eine mathematische Formel aus mehreren Operanden und Operatoren bestehen. Als Operatoren stehen uns (zunächst) die „üblichen“ mathematischen Operatoren `+` (Addition), `-` (Subtraktion), `*` (Multiplikation) und `/` (Division) zur Verfügung. Hinzu kommt der Operator `%` (Rest bei ganzzahliger Division), der bereits in der Einführung angesprochen wurde.

Im Detail werden wir uns im nächsten Abschnitt mit den Wertzuweisungen befassen.

3.6 Zeilenorientierte Verarbeitung

Probiert man das letzte Programm einmal aus, so stellt man fest, dass man damit ein mittelprächtiges Durcheinander auf dem Bildschirm erzeugen kann, wenn man mehrere Zahlen durch Leerzeichen getrennt auf einer Zeile eingibt. Letztlich tut es aber, was es soll. Woran liegt das vermeintliche Chaos?

Der Bildschirm hat, wenn man es genau betrachtet, eine Doppelfunktion: Einerseits werden auf ihm die Tastatur–Anschläge protokolliert; andererseits dient er als Standardausgabe für die Programme. Der Hintergrund für ein eventuelles Chaos ist die *gepufferte Eingabe*: Die Tasten, die man auf der Tastatur anschlägt, werden zwar sofort *geechot*, d.h. auf dem Bildschirm protokolliert – einem Programm werden sie allerdings erst zur Verfügung gestellt, wenn die `ENTER`–Taste gedrückt wird (was als Echo einen Zeilenvorschub ergibt). Unser Beispielprogramm wartet also zunächst auf das Drücken der `ENTER`–Taste. Enthält die Zeile, die es bekommt, mehrere Zahlen, so liest und schreibt es diese, ohne neuerliche Eingabe zu erwarten, auch wieder in eine Zeile. (Da das Ausgabeformat keinerlei Trennung zwischen den Werten vorsieht, erscheinen die Werte bei der Ausgabe auch unmittelbar hintereinander!) Ist die Maximalzahl der zu kopierenden Zahlen noch nicht erreicht, bleibt die Ausgabe direkt hinter dem letzten geschriebenen Wert stehen – jetzt wird wieder Eingabe erwartet.

Übersichtlicher wird es, wenn man jeden Wert in eine eigene Zeile ausgibt und den Benutzer bittet, jeden Eingabewert mit `ENTER` abzuschließen. Außerdem kann man dem Benutzer noch Hinweise zur korrekten Eingabe geben.

```
#include <stdio.h>

#define ANZAHL 10               /* Anzahl Wiederholungen   */

/*= Hauptprogramm ========================================*/
int main (void) {
  int i = ANZAHL, wert;

  printf("Geben Sie die Zahlen ein. ");
  printf("Bitte jeweils mit ENTER abschliessen!\n");

  while (i > 0) {               /* Schleife                */
    printf("Noch %d Zahlen: ", i);
    scanf("%d", &wert);         /* eine Zahl lesen         */
    printf("%d\n", wert);       /* gelesene Zahl schreiben */
    i = i - 1;                  /* Anzahl verringern       */
  }
  return 0;
}
```

Quelltext 3.3: Zahlen kopieren mit Vorschüben

Eine Möglichkeit, den Benutzer dazu zu zwingen, pro Zeile nur einen Wert einzugeben, haben wir derzeit noch nicht. (Dieses „Zwingen“ könnte ohnehin auch nur darin bestehen, dass wir aus jeder Eingabezeile nur den ersten Wert interpretieren und eventuelle weitere Werte ignorieren.)

3.7 Lesen bis zum Ende

Die Aufgabenstellung wird noch einmal modifiziert: Das Programm soll so lange `int`–Zahlen lesen und wieder schreiben, wie der Benutzer das wünscht. Oder, in der Terminologie der Datenverarbeitung: Das Programm soll so lange `int`–Zahlen lesen und wieder schreiben, bis das Ende der Eingabedatei erreicht ist. Dazu müssen wir zunächst zwei Fragen klären:

1. Wie kann der Benutzer anzeigen, dass die Datei zu Ende ist?
2. Wie erfährt das Programm diese Information?

Die Antwort auf die erste Frage ist einfach: Tippt der Benutzer unter UNIX `C-d`, so wird das vom Betriebssystem als Dateiende interpretiert.

Die Antwort auf die zweite Frage ist auch nicht viel komplizierter, nur muss zur Erklärung etwas weiter ausgeholt werden. Wir erinnern uns daran, dass C nur Funktionen kennt. Bisher ist der Funktionswert der verwendeten Funktionen `scanf` und `printf` jedoch noch nicht in Erscheinung getreten. Er liefert jedoch die Antwort auf die Frage: `scanf` liefert als Funktionswert die Anzahl der gelesenen und erfolgreich umgewandelten Werte; tritt jedoch bereits *vor* dem Lesen und der Umwandlung des ersten Wertes ein Fehler ein (und das Dateiende zählt in diesem Sinne als Fehler), so ist der Funktionswert `EOF` (*End Of File*).

Offensichtlich muss `EOF` ein negativer, ganzzahliger Wert sein, da alle anderen möglichen Funktionswerte ganzzahlig und nichtnegativ sind. Häufig ist der Wert -1, aber das wird vom Standard nicht vorgeschrieben. Die benannte Konstante `EOF` ist im übrigen wie `scanf` und `printf` in `stdio.h` deklariert.

Wie kann man den Funktionswert nutzen? Zur Verfügung steht, neben anderen, der *Vergleichsoperator* `!=` (*ungleich*), mit dem wir den Funktionswert mit der Konstanten `EOF` vergleichen können. Das Resultat ist *wahr* bzw. *falsch*, je nachdem, ob die Relation des Operators zwischen den beiden Operanden besteht oder nicht.

Damit können wir die neue Programmvariante realisieren:

```
#include <stdio.h>

/*= Hauptprogramm ========================================*/
int main(void) {
  int wert, anzahl = 0;

  printf("Geben Sie die Zahlen ein:\n");

  while (scanf("%d", &wert) != EOF) {  /* Zahl lesen      */
    anzahl = anzahl + 1;
    printf("Der %d. Wert: %d\n", anzahl, wert);
  }
  return 0;
}
```

Quelltext 3.4: Eingabe bis zum Ende der Eingabe (`EOF`) verarbeiten

Zwei Anmerkungen noch:

- Der Funktionswert von `scanf` erlaubt nicht nur die Erkennung des Dateiendes, sondern auch die Erkennung anderer Fehler bei der Eingabe: Ist der Funktionswert kleiner als die Anzahl der Prozentzeichen im Format und der Variablen in der Argumentliste, so muss etwas schief gelaufen sein. Sollen etwa 10 Werte gelesen werden und ist der Funktionswert 8, so kann man daraus rückschließen, dass beim Lesen des neunten Wertes etwas schief gegangen ist.

  `EOF` wird nur dann geliefert, wenn bereits *vor* dem Lesen des ersten Wertes etwas schief geht. Tritt *beim* Lesen des ersten Wertes ein Fehler ein, so ist der Funktionswert 0. Statt mit `EOF` sollte man den Funktionswert von `scanf` in unserem Beispiel besser mit `1` vergleichen (`scanf (...) == 1`).
- Auch `printf` liefert einen wohldefinierten Funktionswert. Der Standard schreibt vor:
  - Bei korrektem Abschluss der Ausgabe ist der Funktionswert die Anzahl der geschriebenen Zeichen.
  - Bei fehlerhaftem Abschluss ist der Funktionswert negativ. Den Implementatoren ist es freigestellt, unterschiedliche Fehler durch unterschiedliche Werte zu kennzeichnen.

3.8 Alternativen

Neben der Bildung von Schleifen gibt es eine zweite grundlegende Programmiertechnik, nämlich die *Auswahl aus Alternativen*. Realisiert wird sie durch *Verzweigungsanweisungen*.

Wird bei einer Schleife eine Folge von Anweisungen mehrfach (oder ggf. auch keinmal) ausgeführt, so kann man mit den Verzweigungsanweisungen erreichen, dass in Abhängigkeit von bestimmten Bedingungen aus mehreren Anweisungsfolgen eine ausgewählt und ausgeführt wird.

Die wichtigste Verzweigungsanweisung in C ist die `if`–Anweisung:

```
if (bedingung) {
    anweisungsfolge1
}
[ else {
    anweisungsfolge2
} ]
```

Je nachdem, ob die Auswertung von *bedingung* den Wert *wahr* oder *falsch* ergibt, wird entweder *anweisungsfolge1* oder *anweisungsfolge2* ausgeführt.

Die Vergleichsoperatoren `!=` und `==` haben wir eben bereits kennengelernt. Daneben gibt es die vier Vergleichsoperatoren `<`, `<=`, `>` und `>=`, die in ihrer Bedeutung selbsterklärend sind. Bei `==` muss man allerdings sehr aufpassen! Schreibt man nur ein Gleichheitszeichen, so akzeptiert der Compiler das auch – nur ist die Bedeutung eine *vollständig* andere!

Als Beispiel betrachten wir folgende Aufgabe: Der Benutzer soll eine positive ganze Zahl eingeben; nicht positive Zahlen sollen zurückgewiesen werden. In der Praxis kommt diese Aufgabe sicher nur als Bestandteil umfangreicherer Aufgaben vor; wir wollen sie hier aber für sich lösen.

```
#include <stdio.h>

/*= Hauptprogramm ========================================*/
int main(void) {
```

```
  int wert = 0;

  printf("Das Programm wird durch die Eingabe einer"
         "positiven Zahl beendet.\n");

  while (wert <= 0) {
    printf("Geben Sie eine Zahl ein: ");
    if (scanf("%d", &wert) != 1) {     /* Zahl lesen       */
      printf("Eingabefehler!\n");      /* Fehlerbehandlung */
      return 1;
    }
    else {
      if (wert <= 0) {                 /* Eingabe ok ?     */
        printf("Falsch - noch einmal!\n");       /* nein ! */
      }
      else {
        printf("Einverstanden!\n");              /* ja !   */
      }
    }
  }
  return 0;
}
```

Quelltext 3.5: Auswahl aus Alternativen mit `if` und `else`

Hier sehen wir gleich noch etwas: Schleifen und Verzweigungsanweisungen dürfen *geschachtelt* werden.

3.9 Felder

In mathematischen Formeln werden oft Vektoren oder Matrizen verwendet; in anderen Problemkreisen hat man häufig Tabellen. Programmiersprachen stellen entsprechende Datenstrukturen unter dem Oberbegriff *Felder* (oder auch *array*) zur Verfügung. Der sprachlichen Einfachheit halber werden eindimensionale Felder in der Regel als *Vektoren* und zweidimensionale Felder als *Matrizen* bezeichnen.

Das Wesen eines Feldes ist, dass es aus einer festen Anzahl von *Elementen* (*Komponenten*) besteht, die sämtlich denselben Typ besitzen.

In C deklariert man Vektoren, indem man dem Namen des Feldes in der Variablendeklaration die gewünschte Komponentenzahl nachstellt, eingeschlossen in eckige Klammern. Die Deklaration

```
double v[5];
```

oder besser

```
#define LAENGE 5
...
double v[LAENGE];
```

besagt so: `v` ist ein Vektor (eindimensionales Feld) mit 5 Komponenten. Die Dimensionierung *muss* mit Konstanten erfolgen – allerdings sind die benannten Konstanten eben auch Konstanten.

Ansprechen kann man einzelne Komponenten eines Vektors, indem man seinem Namen den *Index* der entsprechenden Komponente nachstellt, erneut eingeschlossen in eckige Klammern. Dabei muss man beachten: Die Numerierung der Komponenten beginnt *immer*

bei 0 , so dass entsprechend der höchste (erlaubte) Index um 1 kleiner ist als die Länge des Vektors. In unseren Beispiel hat `v` also die Komponenten `v[0]`, `v[1]`, ..., `v[4]`.

Die Indizes der Komponenten müssen keine Konstanten sein. Erlaubt sind (u.a.) auch Variablen. Klar sollte sein, dass diese Variablen einen ganzzahligen Typ besitzen müssen und dass ihre Werte innerhalb des Indexbereiches des Vektors liegen müssen.

Als Beispiel betrachten wir ein Programm, das zwei `double`–Vektoren liest, ihr Skalarprodukt berechnet und das Resultat ausgibt:

```
#include <stdio.h>

#define LAENGE 5

/*= Hauptprogramm ========================================*/
int main(void) {
  double v1[LAENGE], v2[LAENGE], prod;  /* Variablendekl.  */
  int i;

  printf("1. Vektor eingeben!\n");      /* 1. Vektor lesen */
  i = 0;
  while (i < LAENGE) {
    scanf ("%lf", &v1[i]);
    i = i + 1;
  }

  printf("2. Vektor eingeben!\n");      /* 2. Vektor lesen */
  i = 0;
  while (i < LAENGE) {
    scanf ("%lf", &v2[i]);
    i = i + 1;
  }

  i = 0;                                /* Skalarprodukt   */
  prod = 0;                             /*   berechnen     */
  while (i < LAENGE) {                  /*   und ausgeben  */
    prod = prod + v1[i] * v2[i];
    i = i + 1;
  }
  printf("Das Skalarprodukt ist: %f\n", prod);

  return 0;
}
```

Quelltext 3.6: Arbeit mit Vektoren: Berechnen eines Skalarproduktes

Das sollte zunächst ausreichen, um Ihnen die Verwendung von (eindimensionalen) Feldern zu erlauben.

Kapitel 4

Ausdrücke und Operatoren

4.1 Aufbau von Ausdrücken

Ausdrücke setzen sich aus *Operanden*, *Operatoren* und *runden Klammern* zusammen, wobei die Operanden in der Regel Konstanten, Variablen (einschließlich Feldkomponenten) und Funktionsaufrufe sein können. Die einfachsten Ausdrücke bestehen nur aus einem Operanden.

An verschiedenen Stellen sind wir bereits Ausdrücken begegnet, ohne dass ihnen besondere Aufmerksamkeit geschenkt wurde:

`i`	als Argument von `printf`
`i - 1`	bei der Schleifenzählung
`v1[i] * v2[i]`	beim Skalarprodukt
`i < LAENGE`	beim Skalarprodukt

Einige weitere Ausdrücke als Beispiele:

```
(a + b) * 7
(a + b) * (c + d)
- a
```

Wie schon erwähnt, sind die Variablen, die wir in den Beispielen als Argumente für `printf` verwendet haben, formal als spezielle Ausdrücke anzusehen. Entsprechend können wir natürlich auch beliebige Ausdrücke als Argumente für `printf` verwenden. Warum das bei `scanf` nicht möglich ist, werden wir uns noch ansehen.

4.2 Die Wertzuweisung

Eine Besonderheit von C gegenüber anderen Sprachen ist, dass auch der *Zuweisungsoperator* (`=`) als Operator betrachtet wird:

```
i = i - 1
```

ist also formal ein Ausdruck und wird, wie in unseren Beispielen geschehen, nur durch das nachgestellte Semikolon zu einer Anweisung. Die Wirkung: Zunächst wird `i - 1` berechnet, dann der resultierende Wert der Variablen `i` zugewiesen. Dieser Wert ist gleichzeitig der Wert des gesamten Ausdrucks.

Damit werden Ausdrücke wie

```
i = (j = (k = 0))
```

möglich: Die drei Variablen `i`, `j` und `k` erhalten jeweils den Wert 0. Da mehrere Gleichheitszeichen von rechts nach links abgearbeitet werden, sind die Klammern in diesem Ausdruck sogar überflüssig und wir können kurz schreiben

```
i = j = k = 0
```

Compiler betrachten den Namen einer Variablen in der Regel als Repräsentanten für den Wert der Variablen. Bei der Funktion `scanf` waren wir deshalb gezwungen, den Argumenten den Adressoperator `&` voranzustellen, damit die Adresse des Arguments anstelle seines Wertes übergeben wird. Gleiches gilt eigentlich auch für die Wertzuweisung: Benötigt wird ja nicht der Wert der Variablen, die verändert werden soll, sondern ihre Adresse.

Da es klar ist, dass auf der linken Seite des Zuweisungsoperators eine Adresse benötigt wird, und da die explizite Angabe des Adressoperators an allen diesen Stellen viel zu mühsam wäre, vereinbart man ganz einfach: Auf der linken Seite des Zuweisungsoperators repräsentiert ein Variablenname die Adresse der Variablen und nicht wie sonst ihren Wert. Schreiben wir etwa den (in der Sache unsinnigen) Ausdruck

```
i = i
```

so steht also `i` links und rechts vom Zuweisungsoperator für ganz verschiedene Dinge.

In der Regel gibt das keine Probleme. Wesentlich wird die Unterscheidung allerdings, wenn man sich überlegt, was aus der linken Seite eines Zuweisungsoperators stehen darf und was nicht. Dass Variablen und Elemente von Feldern erlaubt sind, haben wir bereits gesehen. Nicht erlaubt sind dagegen zum Beispiel

```
5 = i          /* nicht   */
i + j = k      /* erlaubt */
```

da weder die Konstante `5` noch der Ausdruck `i + j` einen eigenen Speicherplatz mit Adresse besitzen.

Offensichtlich unterliegen die Argumente von `scanf` denselben Regeln wie die linken Operanden von Wertzuweisungen: `scanf` soll ja auch den Variablen Werte zuweisen.

4.3 Arithmetische Operatoren

C kennt fünf *arithmetische Operatoren*, nämlich `+`, `-`, `*`, `/` und `%`.

Plus und *Minus* gibt es in zwei Varianten: Als *unäre* Operatoren dienen sie als Vorzeichen, wobei das positive Vorzeichen erst durch den Standard vorgeschrieben wurde; als *binäre* Operatoren bewirken sie Addition bzw. Subtraktion.

Multiplikation bewirkt der Stern, *Division* der Schrägstrich. Bei der Division ganzzahliger Operanden ist eine Besonderheit zu beachten: Der Quotient ist wieder ganzzahlig! Als Ergänzung zum Divisionsoperator gibt es deshalb den *Modulo-Operator* `%`, der den Rest bei der Division ganzer Zahlen liefert. Dieser Operator darf *nur* für ganzzahlige Operanden verwendet werden, nicht für Gleitkommawerte. Wir sehen uns einige Beispiele für die Division an:

```
10.0 / 3   ergibt  3.333333
  10 / 3   ergibt  3
  10 % 3   ergibt  1
```

Der *Divisionsrest* ist nur dann eindeutig bestimmt, wenn beide Operanden nichtnegativ sind; der Divisionsrest ist dann positiv. Bei mindestens einem negativen Operanden ist es den speziellen Implementationen freigestellt, ob der Divisionsrest positiv oder negativ ist. Entsprechend ist auch die ganzzahlige Division nur für nichtnegative Operanden eindeutig, denn die Beziehung

```
a == (a / b) * b + a % b
```

muss für beliebige ganzzahlige Werte `a` und `b` (`b` ungleich 0!) erfüllt sein. Möglich ist zum Beispiel

```
10 / - 3 == - 3                    10 % - 3 == 1
```

ebenso wie

```
10 / - 3 == - 4                    10 % - 3 == - 2
```

Arithmetische Operatoren werden von links nach rechts abgearbeitet. Dabei werden die Regel „Punktrechnung vor Strichrechnung“ natürlich ebenso berücksichtigt wie Klammern. Außerdem besitzen Plus und Minus als unäre Vorzeichenoperatoren höhere Priorität als die binären Operatoren. Man hat also zum Beispiel

```
a + b * c == a + (b * c)
a + b - c == (a + b) - c
- a * b == (- a) * b
```

Auf die Probleme numerischen Rechnens wird in einem eigenen Kapitel noch eingegangen.

4.4 Inkrementierung und Dekrementierung

C verfügt über spezielle Operatoren zum *Inkrementieren* und *Dekrementieren* von Variablen: `++` erhöht seinen Operanden um 1, `--` verringert ihn um 1.

Beide Operatoren können wahlweise in *Präfix*–Schreibweise (vor dem Operanden) oder auch *Postfix*–Schreibweise (hinter dem Operanden) verwendet werden. Äquivalent sind also jeweils die Anweisungen

```
++i;          i++;          i = i + 1;
--i;          i--;          i = i - 1;
```

Als Ausdrücke sind Präfix– und Postfix–Schreibweise keineswegs äquivalent:

- Bei Präfix–Schreibweise wird *erst* inkrementiert bzw. dekrementiert und *danach* der bereits veränderte Wert der Variablen als Wert des Ausdrucks genommen.
- Bei Postfix–Schreibweise wird *erst* der noch unveränderte Wert der Variablen als Wert des Ausdrucks genommen und *danach* inkrementiert bzw. dekrementiert.

So hat nach

```
i = 10; j = ++i;
```

`i` den Wert `11` und `j` ebenfalls den Wert `11`. Nach

```
i = 10; j = i++;
```

hat `i` wiederum den Wert `11`, `j` jedoch den Wert `10`.

Interessant sind beide Operatoren vor allem im Zusammenhang mit Schleifen und Vektoren. Das Beispiel, in dem wir zwei Vektoren gelesen und ihr Skalarprodukt berechnet haben, können wir mit ihnen durchaus kompakter schreiben:

```
#include <stdio.h>

#define LAENGE 5

int main(void) {
```

```
  double v1[LAENGE], v2[LAENGE], prod;   /* Variablendekl. */
  int i;

  printf("1. Vektor eingeben!\n");      /* 1. Vektor lesen */
  i = 0;                               /* Schleife Variante 1 */
  while (i < LAENGE) {
    scanf("%lf", &v1[i]);
    i++;
  }

  printf("2. Vektor eingeben!\n");      /* 2. Vektor lesen */
  i = - 1;                             /* Schleife Variante 2 */
  while (++i < LAENGE) {
    scanf("%lf", &v2[i]);
  }
                  /* Skalarprodukt berechnen und ausgeben  */
  i = 0;                               /* Schleife Variante 3 */
  prod = 0;
  while (i++ < LAENGE) {
    prod = prod + v1[i-1] * v2[i-1];
  }
  printf("Das Skalarprodukt ist: %f\n", prod);

  return 0;
}
```

Quelltext 4.1: Inkrement-Operator

Es sollte klar sein, dass für die Operanden von Inkrementierung und Dekrementierung dieselben Regeln gelten wie für die Operanden auf der linken Seite eines Zuweisungsoperators: Erneut wird ja eine Adresse benötigt, an der der veränderte Wert gespeichert werden soll.

In der Regel wird man Inkrementierung und Dekrementierung für ganzzahlige Operanden verwenden; erlaubt sind sie aber auch für Gleitkommaoperanden.

4.5 Vergleichsoperatoren

C kennt die sechs *Vergleichsoperatoren* `==`, `!=`, `<`, `<=`, `>` und `>=`. Das Resultat eines solchen Ausdrucks ist *wahr* (gleich 1), wenn die Operanden in der Relation zueinander stehen, die der Operator beschreibt, und *falsch* (gleich 0) sonst. Letztlich sind die Resultate von Vergleichen also `int`–Werte; einen besonderen Typ *boolean* und entsprechende logische Konstanten kennt C nicht. Es steht jedem Programmierer natürlich frei, mit `enum` und `typedef` die logischen Konstanten `false` und `true` sowie einen „Typ“ `boolean` geeignet zu definieren.

Da Vergleichsausdrücke `int`–Werte sind, können sie Operanden in arithmetischen Ausdrücken sein. Da die Vergleichsoperatoren eine geringere *Priorität* als die arithmetischen Operatoren haben, kommt man in der Regel ohne Klammern aus. Der Ausdruck

```
a < b + c
```

entspricht also

```
a < (b + c)
```

a	! a
0	1
$\neq 0$	0

a	b	a && b
0	0	0
$\neq 0$	0	0
0	$\neq 0$	0
$\neq 0$	$\neq 0$	1

a	b	a || b
0	0	0
$\neq 0$	0	1
0	$\neq 0$	1
$\neq 0$	$\neq 0$	1

Tabelle 4.1: Wahrheitstafeln für logische Operatoren

was man in der Regel auch beabsichtigen wird. Die Prioritäten werden im einzelnen noch einmal behandelt, wenn wir mehr Operatoren kennen.

Auf etwas anderes soll an dieser Stelle aber noch kurz hingewiesen werden: Wie bereits erwähnt, kann es verhängnisvoll sein, wenn man statt `==` nur `=` schreibt. Jetzt können wir klären, warum das so ist. Dazu betrachten wir das folgende Beispiel:

```
if (i = 1) {  /* immer wahr */
   ...
}
else {
   ...
}
```

Hier wird die Anweisungsfolge, die dem `else` folgt, *nie* ausgeführt! Bei der Auswertung von `i = 1` erhält die Variable `i` den Wert `1`. Auch der Wert des Ausdrucks insgesamt ist damit `1`, wird entsprechend als *wahr* interpretiert und bewirkt die Ausführung der unmittelbar folgenden Anweisungen.

Auch Endlosschleifen kann man so bequem (und ungewollt) „zusammenbasteln“:

```
while (i = 1) {  /* Endlosschleife */
   ...
}
```

Änderungen von `i` im Schleifenrumpf haben hierauf keinen Einfluss, da der Ausdruck `i = 1` vor jedem Schleifendurchgang erneut ausgewertet wird.

4.6 Logische Operatoren

Auch wenn C, wie wir gesehen haben, keinen Typ `boolean` kennt, kennt es doch drei *logische Operatoren*: `&&` (*und*), `||` (*oder*) und `!` (*nicht*).

Anwenden darf man diese Operatoren auf beliebige Operanden, die sich mit 0 vergleichen lassen – also insbesondere Zahlwerte, da wieder nach der bereits bekannten Regel vorgegangen wird: Ein Wert wird als *wahr* interpretiert, wenn er ungleich Null ist, und als *falsch*, wenn er gleich Null ist. Das Resultat ist entweder 1 oder 0. Die *Wahrheitstafeln* der drei logischen Operatoren finden Sie in Tabelle 4.1.

Logische Ausdrücke werden von links nach rechts abgearbeitet, wobei die Prioritäten der Operatoren berücksichtigt werden (erst die Negation, dann *und*, zuletzt *oder*). Eine Besonderheit gegenüber anderen Programmiersprachen legt der Standard ausdrücklich fest: Die Auswertung eines logischen Ausdrucks wird beendet, sobald das Ergebnis feststeht. Das ist gelegentlich ausgesprochen nützlich. Im Ausdruck

```
(d != 0) && (c / d < 10)
```

wird so die Division *nicht* ausgeführt, wenn `d` gleich 0 ist: Der erste Vergleich stellt fest, dass das Resultat *falsch* ist, unabhängig vom Wert des anderen Operanden – und verhindert so die in diesem Falle nicht zulässige Division. Man spricht dabei von „*lazy evaluation*“.

4.7 Zusammengesetzte Zuweisungen

Neben dem einfachen Zuweisungsoperator `=` gibt es in C zehn *zusammengesetzte Zuweisungsoperatoren*. Diese haben einheitlich die Form *op*`=`, wobei *op* ein binärer Operator ist. Zwischen dem binären Operator und dem Gleichheitszeichen *darf kein* Leerzeichen stehen (eine Neuerung des Standards). Für eine Variable *var* und einen Ausdruck *ausdr* kann man die Wirkung der zusammengesetzten Zuweisungsoperatoren so beschreiben:

var *op*`=` *ausdr*

entspricht

var `=` *var* *op* *ausdr*

d.h. zunächst wird der Ausdruck *ausdr* „normal“ ausgewertet, dann sein Resultat mit dem Wert der Variablen *var* durch den Operator *op* verknüpft und schließlich dieses Resultat in der Variablen *var* gespeichert. Der gespeicherte Wert ist gleichzeitig auch der Wert des gesamten Ausdrucks.

Nicht alle binären Operatoren sind bei der Zusammensetzung erlaubt; von den Operatoren, die wir bislang kennengelernt haben, dürfen nur die fünf arithmetischen Operatoren verwendet werden. Sehen wir uns einige konkrete Beispiele an. Die nebeneinanderstehenden Ausdrücke sind hier jeweils äquivalent:

```
x += 3.5            entspricht    x = x + 3.5
i *= j + 10         entspricht    i = i * (j + 10)
a += b += c         entspricht    a = a + (b = b + c)
v[i + j*k] += 3     entspricht    v[i + j*k] = v[i + j*k] + 3
```

Insbesondere das letzte dieser Beispiele zeigt, dass die zusammengesetzten Zuweisungsoperatoren einige Schreibarbeit sparen helfen. Aber nicht nur das: Dem Compiler wird es erleichtert, effizienten Code zu erzeugen. Der Indexausdruck im letzten Beispiel braucht auf jeden Fall nur einmal berechnet zu werden. Verwendet man die zusammengesetzte Zuweisung, so steht der Indexausdruck auch nur einmal im Programm, so dass der Compiler ihn nicht doppelt berechnen muss. Verwendet man dagegen die einfache Zuweisung, so muss man darauf hoffen, dass der Compiler die Identität erkennt und die Doppelberechnung wegoptimiert.

4.8 Bedingte Ausdrücke

Ein auf den ersten Blick ziemlich kurioses, in der Praxis gelegentlich aber durchaus nützliches Konstrukt ist der *bedingte Ausdruck* (`? :`).

bedingung `?` *ausdruck1* `:` *ausdruck2*

Wenn die Bedingung *bedingung* den Wert *wahr* besitzt, ist der Wert des Gesamtausdrucks der Wert von *ausdruck1*, sonst der Wert von *ausdruck2*. Ein zunächst noch mehr formales Beispiel: Statt

```
if (bedingung) {
     x = ausdruck1;
}
else {
     x = ausdruck2;
}
```

kann man kurz

```
x = bedingung ? ausdruck1 : ausdruck2;
```

schreiben. Ein konkretes Beispiel: Durch

```
max = z1 > z2 ? z1 : z2;
```

erhalten wir in der Variablen `max` das Maximum der beiden Zahlen `z1` und `z2`.

Die drei Ausdrücke, die gemeinsam einen bedingten Ausdruck bilden, brauchen auf keinen Fall geklammert zu werden, da die Operatoren `?` und `:` zwar eine höhere Priorität als der Zuweisungsoperator, gleichzeitig aber eine niedrigere Priorität als alle anderen Operatoren haben.

4.9 Der Kommaoperator

Der *Kommaoperator* (`,`) fasst mehrere Ausdrücke zu einem einzigen Ausdruck zusammen.

Beispiel: Die Werte der beiden Variablen `x` und `y` sollen vertauscht werden. Statt

```
h = x;
x = y;
y = h;
```

kann man auch

```
h = x, x = y, y = h;
```

schreiben. Dass dieses kürzer scheint, liegt ausschließlich daran, dass jeweils eine Anweisung in eine Zeile geschrieben wurde, was der Standard keineswegs verlangt.

Durch Kommaoperatoren getrennte Ausdrücke werden von links nach rechts ausgewertet. Der Wert des Gesamtausdrucks ist der Wert des am weitesten rechts stehenden, zuletzt ausgewerteten Teilausdrucks.

Der Kommaoperator sollte nur dann verwendet werden, wenn Konstruktionen sehr eng zusammengehören, wie im Beispiel: Den Tausch der Werte von zwei Variablen kann man durchaus als eine Operation ansehen. Wir werden andere Stellen kennenlernen, an denen sich der Kommaoperator als durchaus nützlich erweist.

Kommata, die in Variablenvereinbarungen und Funktionsaufrufen stehen, haben übrigens *nichts* mit dem Kommaoperator zu tun. Entsprechend haben sie dort auch keinen Einfluss auf die Reihenfolge der Auswertung.

4.10 Priorität der Operatoren

Da wir nun schon eine ganze Reihe von Operatoren behandelt haben, ist es an der Zeit, dass wir uns die *Prioritäten* der Operatoren einmal im Zusammenhang ansehen. Die Aufstellung ist in Abbildung 4.2 enthalten.

Als erstes ist die Prioritätsstufe der Operatoren angegeben: Je höher die Stufe, desto höher die Priorität. Wenn auf einer Stufe mehrere Operatoren stehen, so haben sie dieselbe Priorität.

Für jede Prioritätsstufe ist ganz rechts angegeben, ob die Abarbeitung ihrer Operatoren von links nach rechts (z.B. bei Addition und Subtraktion) oder von rechts nach links (z.B. bei Wertzuweisungen) erfolgt.

Auch wenn noch nicht alle Operatoren bekannt sind, lohnt es sich, die Tabelle einmal in Muße zu studieren und sich die Prioritätsstufen im wesentlichen einzuprägen. Allzu extensive Nutzung der Kenntnis der Prioritätsstufen lohnt allerdings auch nicht, da dann

Stufe	Operator	Beschreibung	Auswertungsreihenfolge
15	`( )`	Auswertung von Parametern	→
	`[ ]`	Auswahl einer Feldkomponente	
	`-> .`	Auswahl einer Strukturkomponente	
14	`! ~`	Negation (logisch, bitweise)	←
	`++ --`	Inkrementierung, Dekrementierung (Präfix oder Postfix)	
	`+ -`	Vorzeichen (unär)	
	`(`*Typ*`)`	Typumwandlung	
	`& *`	Adressbildung, Dereferenzierung (unär)	
	`sizeof`	Bestimmung des Speicherbedarfs	
13	`* /`	Multiplikation (binär), Division	→
	`%`	Rest bei ganzzahliger Division	
12	`+ -`	Summe, Differenz (binär)	→
11	`<< >>`	bitweise Verschiebung nach links, rechts	→
10	`< <=`	Vergleich auf kleiner, kleiner oder gleich	→
	`> >=`	Vergleich auf größer, größer oder gleich	
9	`== !=`	Vergleich auf gleich, ungleich	→
8	`&`	Und (bitweise)	→
7	`^`	exklusives Oder (bitweise)	→
6	`|`	inklusives Oder (bitweise)	→
5	`&&`	Und (logisch)	→
4	`||`	inklusives Oder (logisch)	→
3	`? :`	bedingte Auswertung (nur paarweise!)	←
2	`=`	Wertzuweisung	←
	`+=`	zusammengesetzte Wertzuweisung (auch: `-=`, `*=`, `%=`, `/=`, `<<=`, `>>=`, `&=`, `^=`, `|=`)	
1	`,`	sequentielle Auswertung	→

Tabelle 4.2: Prioritäten der Operatoren

leicht völlig unleserliche Ausdrücke resultieren. Die folgende Auswahl verdeutlicht diesen Umstand:

```
i - j         /*  1: binaer                          */
i -- j        /*  2: unzulaessig                     */
i - - j       /*  3: binaer/unaer                    */
i --- j       /*  4: Dekrement/binaer (wie 6.)       */
i - -- j      /*  5: binaer/Dekrement                */
i -- - j      /*  6: Dekrement/binaer                */
i ---- j      /*  7: unzulaessig                     */
i - --- j     /*  8: unzulaessig                     */
i -- -- j     /*  9: unzulaessig                     */
i --- - j     /* 10: Dekrement/binaer/unaer          */
i ----- j     /* 11: unzulaessig                     */
i - ---- j    /* 12: unzulaessig                     */
i -- --- j    /* 13: unzulaessig                     */
i --- -- j    /* 14: Dekrement/binaer/Dekrement      */
i ---- - j    /* 15: unzulaessig                     */
i ------ j    /* 16: unzulaessig                     */
```

Falls ein Ausdruck dieser Art benötigt wird, kann es nicht schaden, ihn auch dann zu klammern, wenn dies nicht zwingend notwendig ist.

4.11 Nebeneffekte

Unter einem *Nebeneffekt* versteht man bei der Programmierung, dass als Nebenprodukt der Auswertung eines Ausdrucks der Wert einer Variablen verändert wird.

In der Regel wird dringend geraten, Nebeneffekte zu vermeiden. Das ist in C jedoch nicht möglich, denn C ist geradezu die „Sprache der Nebeneffekte“: Da die Wertzuweisung keine Anweisung, sondern ein Operator ist, lassen sich die Werte von Variablen *nur* durch Nebeneffekte verändern!

Neben den Zuweisungsoperatoren bewirken auch die Inkrement– und Dekrementoperatoren sowie alle Funktionen mit Ausgabeparametern Nebeneffekte. Beispiel:

```
while (scanf (...) == 1)
   ...
```

Der Standard schreibt nur sehr eingeschränkt vor, zu welchem Zeitpunkt Nebeneffekte wirksam werden müssen:

- Nebeneffekte können natürlich nicht eintreten, bevor die Auswertung des Ausdrucks beginnt, durch die sie erzeugt werden.
- Wenn die Auswertung eines Ausdrucks abgeschlossen ist, müssen auch seine Nebeneffekte eingetreten sein.

Die Folge: Wenn eine Variable innerhalb eines Ausdrucks mehrfach angesprochen wird und außerdem Nebeneffekte für sie eintreten, ist meist *nicht* definiert, welchen Wert die Variable im Einzelfall besitzt. So können Ausdrücke wie

```
v[i] = i++;
v[i] = w[i++];
s += v1[i] * v2[i++];
b = a++ + a;
```

implementationsspezifisch unterschiedliche Resultate zeitigen.

Ähnliche Probleme können sich aus der Tatsache ergeben, dass die Reihenfolge der Auswertung von Ausdrücken nur teilweise festgelegt ist:

- Für die drei Ausdrücke, die gemeinsam einen bedingten Ausdruck bilden, liegt die Reihenfolge eindeutig fest: Erst wird die Bedingung ausgewertet, dann der Ausdruck, der das Resultat ergibt. Der jeweils andere Ausdruck wird nicht ausgewertet und es treten keine Nebeneffekte für ihn ein.
- Logische Ausdrücke werden von links nach rechts ausgewertet; ihre Berechnung wird beendet, sobald das Resultat feststeht. Bei `||` ist das der Fall, sobald der erste Teilausdruck wahr ist, bei `&&` sobald der erste Teilausdruck falsch ist. Nebeneffekte treten jeweils auf, bevor mit dem nächsten Teilausdruck fortgefahren wird.
- Ausdrücke, die mit dem Kommaoperator verknüpft sind, werden von links nach rechts ausgewertet. Dabei treten alle Nebeneffekte auf, bevor der rechte Ausdruck ausgewertet wird.
- Sonst ist zwar festgelegt, in welcher Reihenfolge aufeinanderfolgende Operatoren gleicher Hierarchiestufe ausgeführt werden, nicht aber die Reihenfolge der Berechnung der Operanden.

Betrachten wir für den letzten Punkt ein Beispiel: Der Ausdruck

```
(a) + (b) + (c)
```

muss wie

```
((a) + (b)) + (c)
```

ausgewertet werden. Sind `a`, `b` und `c` ihrerseits Ausdrücke, so ist keineswegs festgelegt, welcher von ihnen zuerst berechnet wird. Konkret: Die Anweisungsfolge

```
i = 5;
j = i * 3 + ++i - 7;
```

kann in der Variablen `j` je nach Implementation den Wer 14 oder 17 liefern.

Ebenfalls ist nicht festgelegt, in welcher Reihenfolge die Argumente eines Funktionsaufrufs berechnet werden. So kann die Anweisungsfolge

```
n = 3;
printf("%d, %d\n", ++n, 2*n);
```

je nach Implementation die Ausgabe 4, 6 oder 4, 8 ergeben.

Aber auch die scheinbar problemlosen Operatoren bieten „Fallstricke“, weil bei ihnen Teilausdrücke unter Umständen nicht ausgewertet werden und in ihnen steckende Nebeneffekte entsprechend mal eintreten und mal nicht:

```
if (x == y && a == ++b)          /*  1  */
    ...
if (x == y || a == ++b)          /*  2  */
    ...
z = x == y ? a : ++b;            /*  3  */
```

Im ersten Beispiel wird `b` nur dann inkrementiert, wenn `x == y` gilt; im zweiten und dritten Beispiel wird `b` nur dann inkrementiert, wenn `x != y` gilt.

Fazit: Bei Operatoren und Funktionen, die Nebeneffekte erzeugen, sollte man *äußerste* Vorsicht walten lassen – zumal man Fehler im Programm, die aus Nebeneffekten resultieren, oft nur mit großer Mühe findet. Insbesondere sollte man darauf achten, dass Variablen, die inkrementiert oder dekrementiert werden, innerhalb eines Ausdrucks nur an der Stelle vorkommen, an der sie inkrementiert bzw. dekrementiert werden.

4.12 Typumwandlung

Einen wesentlicher Punkt ist bisher noch nicht eindeutig geklärt worden: Welche Typen dürfen bzw. müssen die Operanden eines binären arithmetischen Operators besitzen?

Die Antwort ist ganz einfach: Jede beliebige Kombination ist erlaubt! Oder, um den Fachbegriff zu verwenden: *mixed mode* ist uneingeschränkt erlaubt. Das Beispiel, das wir bereits betrachtet hatten, war

```
10.0 / 3 = 3.333333
```

Die Frage ist: Welchen Wert erhalten solche Ausdrücke? Der Rechner kann grundsätzlich nur Operanden mit identischen Typen verknüpfen, so dass also vor die eigentliche Verknüpfung ggf. eine Typumwandlung vorgeschaltet werden muss. Dieses übernimmt der C–Compiler in der Regel für den Programmierer; man bezeichnet solche Typumwandlungen auch als *implizite Typumwandlungen*. Dass das Resultat einer Verknüpfung den Typ der beteiligten Operanden (ggf. nach der Umwandlung) hat, ist naheliegend, von der

ganzzahligen Division vielleicht einmal abgesehen. Man kann deshalb auch vom „Typ eines Ausdrucks“ sprechen.

Schauen wir uns zunächst an, was bei der Verknüpfung von zwei Operanden durch einen binären arithmetischen Operator geschieht:

Der erste Grundsatz: Mit Werten der Typen `char` und `short`, mit oder ohne Vorzeichen, wird nicht gerechnet, d.h. Werte mit diese Typen werden vor dem Rechnen auf jeden Fall automatisch in `int` bzw. `unsigned int` umgewandelt. Ein Ausdruck wie `'Z' - 'A'` hat also den Typ `int`.

Der zweite Grundsatz: Bei Operanden mit unterschiedlichen Typen wird der Operand mit dem „niederwertigen“ Typ in den „höherwertigen“ Typ umgewandelt. Was unter „niederwertig“ und „höherwertig“ zu verstehen ist, legt die in Tabelle 4.3 dargestellte Hierarchie der Typen fest.

`long` und `int` ungleich		`long` und `int` gleich	
7.	`long double`	5.	`long double`
6.	`double`	4.	`double`
5.	`float`	3.	`float`
4.	`unsigned long`	2.	`unsigned int/long`
3.	`long`	1.	`int/long`
2.	`unsigned int`		
1.	`int`		

Tabelle 4.3: Hierarchie der impliziten Typumwandlung

Dass die Typen `char` und `short` hier nicht vorkommen, liegt daran, dass sie ohnehin umgewandelt werden.

Bei komplexeren Ausdrücken entwickelt sich der Typ im Laufe der Auswertung. Insbesondere wird nicht zuerst der Typ des Resultats bestimmt und erst danach die Berechnung ausgeführt. Dahinter steht, dass jeder Teilausdruck im einfachsten möglichen Typ ausgewertet werden soll. Beispiel:

```
short  s = 65;
int    i = 7;
long   j = 1000;
float  x = 3.0;
double y = 46.5;

printf("%f\n", s * i + j % i + y / x);
```

Zunächst werden die Multiplikationen und Divisionen ausgeführt, wobei die angegebene Reihenfolge nicht notwendig eingehalten werden muss:

- Im Ausdruck `s * i` wird `s` zunächst in `int` umgewandelt. Die Multiplikation liefert den `int`–Wert 455.
- Im Ausdruck `j % i` wird `i` zunächst in `long` umgewandelt. Der Divisionsrest ist der `long`–Wert 6.
- Im Ausdruck `y / x` wird `x` zunächst in `double` umgewandelt. Der Quotient ist der `double`–Wert 15.5.

Die Additionen *müssen* nun von links nach rechts ausgeführt werden:

- Der `int`–Wert 455 wird in `long` umgewandelt und zum `long`–Wert 6 addiert. Die Summe ist der `long`–Wert 461.

- Der `long`–Wert 461 wird in `double` umgewandelt und zum `double`–Wert 15.5 addiert. Die Summe ist der `double`–Wert 476.5. Das ist gleichzeitig das Gesamtresultat.

Die Art und Weise, in der die impliziten Typumwandlungen erfolgen, birgt wieder „Fallstricke“. Ein häufig gemachter Fehler ist zum Beispiel die unbeabsichtigte ganzzahlige Divisionen. So liefert der Ausdruck `1 / 2 * 3.5` *stets* den Wert 0! In derart offensichtlicher Form kommt dieser Fehler in der Praxis natürlich selten vor. Häufiger ist er mit zwei `int`–Variablen, etwa `i / j * 3.5` – was an der ganzzahligen Division nichts ändert.

Die Zuweisungsoperatoren nehmen eine Sonderstellung ein! Es ist einer Variablen ein Wert zuzuweisen – der Typ einer Variablen ist aber nichts, was während der Ausführung eines Programms verändert werden kann. Typumwandlung im Zuge einer Wertzuweisung ist also zwar ggf. erforderlich, kann aber stets nur in einer Umwandlung des Typs des Wertes in den Typ der Variablen bestehen. Dabei kann es erforderlich sein, dass aus einem „höherwertigen“ in einen „niederwertigen“ Typ umgewandelt wird:

- Bei der Umwandlung zwischen ganzzahligen Typen werden nur die wertniedrigsten Bits übernommen.
- Bei der Umwandlung eines Gleitkommawertes in einen ganzzahligen Typ gehen eventuelle Nachkommastellen verloren. Eine Rundung erfolgt bei dieser Umwandlung *nicht*! So erhält die `int`–Variable `i` durch die Wertzuweisung `i = 3.9;` den Wert `3`!
- Wenn bei Gleitkommatypen aus höherer in niedrigere Genauigkeit umgewandelt wird (z.B. von `double` in `float`), bleibt es den Implementatoren überlassen, ob dabei gerundet oder abgeschnitten wird. (Das ist bei binären Darstellungen letztlich gleichwertig!)
- Bei der Umwandlung aus einem „großen“ ganzzahligen Typ in einen „kleinen“ Gleitkommatyp kann Genauigkeit verlorengehen.

Über die impliziten Typumwandlungen, die ggf. für die Argumente von Funktionen durchgeführt werden, wurde bei der Einführung von `printf` schon einiges gesagt. Im Rahmen der Behandlung der Funktionen wird darauf noch weiter eingegangen.

4.13 Castoperatoren

Gelegentlich ist es wünschenswert, den Compiler dazu zwingen zu können, einen Wert in einen bestimmten Typ umzuwandeln, etwa zur Vermeidung von unerwünschten ganzzahligen Divisionen.

Hierfür stellt C die *Castoperatoren* zur Verfügung. Solch ein Castoperator ist eine Typbezeichnung, die in runde Klammern eingeschlossen wird. Castoperatoren sind immer unär und haben sehr hohe Priorität, wie alle anderen unären Operatoren.

Wollen wir etwa erreichen, dass der `double`–Quotient von zwei `int`–Variablen `i` und `j` berechnet wird, so können wir alternativ schreiben

```
(double) i / j
i / (double) j
```

Der Castoperator wirkt dabei wegen seiner hohen Priorität jeweils nur auf einen der beiden Operanden. Der andere Operand wird dann gemäß der Regeln der impliziten Typumwandlung ebenfalls nach `double` umgewandelt.

Alle Probleme der Umwandlungen, die durch Wertzuweisungen erzwungen werden, können offensichtlich auch bei der Verwendung von Castoperatoren auftreten. Darüber hinaus erlauben die Castoperatoren bei unsachgemäßer Verwendung auch viel Unsinn. Dies wird besonders später bei der Verwendung von Zeigern deutlich werden.

4.14 Der sizeof–Operator

Mit dem sizeof–Operator lässt sich der Speicherbedarf für Datentypen ermitteln. Das wird später noch eine Rolle spielen, wenn dynamisch Speicher bereitgestellt werden soll.

```
sizeof ausdruck
sizeof (typ)
```

Im ersten Fall ist das Resultat der Speicherbedarf der benötigt würde, um eine einfache Variable vom Typ des Ausdrucks zu speichern. Im zweiten Fall ist das Resultat der Speicherbedarf einer einfachen Variablen mit dem angegebenen Typ. Die Maßeinheit ist rechnerspezifisch, wird jedoch dadurch festgelegt, dass der Standard `sizeof (char) == 1` vorschreibt. Festgelegt ist außerdem, dass das Resultat den Typ `size_t` besitzt – oder umgekehrt: Der Typ `size_t` ist so zu definieren, dass sein Wertebereich alle möglichen Resultate des Operators `sizeof` umfasst. `size_t` ist demnach kein elementarer Datentyp, sondern wird in den Headerdateien `stddef.h`, `stdlib.h` und `string.h` deklariert.

Bei der Verwendung des `sizeof`–Operators sind einige Dinge besonders zu beachten:

- Handelt es sich bei *ausdruck* um den Namen eines Feldes, bzw. bei *typ* um einen Feldtyp, so wird sowohl die Dimension des Feldes als auch die Größe der einzelnen Elemente berücksichtigt.

  ```
  int feld[7];
  size_t s;
  s = sizeof (feld);          /* s == 28  */
  s = sizeof (char [2][5]);   /* s == 10  */
  ```

- Der `sizeof`–Operator ist trotz seiner Form nicht mit einem Funktionsauruf wie z.B. bei `printf(...)` zu verwechseln. Ganz speziell gilt, dass keine Nebeneffekte für den Ausdruck *ausdruck* auftreten. Der Compiler ermittelt lediglich den *Typ* des Ausdrucks. Zur Laufzeit des Programms werden keine weiteren Operationen ausgeführt. Das kann zu einiger Verwirrung führen:

  ```
  int s = 0;
  printf("%d\n", sizeof (5 / s));  /* Ausgabe: 4 */
  printf("%d\n", sizeof (s = 5));  /* Ausgabe: 4 */
                                   /* s == 0     */
  ```

  Es sind also sogar Ausdrücke erlaubt, die zur Laufzeit einen Fehler verursachen würden (im Beispiel Division mit 0). Da dieser Ausdruck jedoch formal den Typ `int` besitzt, wird der Wert 4 ermittelt (vorausgesetzt der Typ `int` besitzt die Größe 4, was jedoch auf gängigen 32–Bit Architekturen der Fall ist).

Kapitel 5

Zeichen und Strings

5.1 Stringvariablen und –konstanten

Den Datentyp `char` haben wir bereits kennengelernt. Ebenso wurde bereits angesprochen, wie Stringkonstanten aussehen. Wie sieht es mit Stringvariablen aus? Diese werden durch Felder von `char`–Werten realisiert. Ein Feld von Zeichen wird dadurch zu einem String, dass man hinter dem letzten „echten“ Zeichen ein *Null–Zeichen* speichert. Bei Stringkonstanten sorgt der Compiler sogar dafür, dass dieses Null–Zeichen automatisch angehängt wird. Das Null–Zeichen hat die Ordnungszahl `0` und kann im Quelltext als `\0` geschrieben werden.

Damit wissen wir jetzt auch, warum `'w'` und `"w"` nicht dasselbe bedeuten:

- `'w'` hat als Wert die Ordnungszahl des Zeichen `w`.
- `"w"` hat als Wert die Zeichenfolge aus dem Zeichen `w` und einem nachfolgenden Null–Zeichen.

Vereinbart man Stringvariablen, so muss man bei der Längenangabe das Null–Zeichen stets mitzählen. Von den Vereinbarungen

```
char text1[4] = "Text",      /*  nicht korrekt  */
     text2[5] = "Text",
     text3[6] = "Text";
```

ist die erste also *nicht* korrekt, weil sie das abschließende Null–Zeichen *nicht* berücksichtigt. Bei der zweiten Vereinbarung sind die Länge von Variable und Konstante gerade identisch. Die dritte Vereinbarung ist auch zulässig; bei ihr bleibt die letzte der sechs verfügbaren Zeichenpositionen frei.

Eine zusätzliche Möglichkeit für Stringkonstanten sollte hier noch erwähnt werden, weil sie häufiger nützlich ist: Längere Stringkonstanten kann man, in Teile zerlegt, in mehrere Zeilen schreiben. Dazu beendet man die Teile am Zeilenende mit einem Gänsefüßchen und leitet den nächsten Teil wieder mit einem Gänsefüßchen ein; zwischen den Teilen dürfen neben dem Zeilenende–Zeichen auch Leerzeichen und horizontale Tabulatoren stehen. Der Compiler fügt diese Teile unmittelbar hintereinander. Welche der beiden Darstellungen

```
printf("Dieses ist ein nicht besonders langer String\n");
printf("Dieses ist ein nicht besonders "
       "langer String\n");
```

wir wählen, ist also für das Resultat gleichgültig. Man beachte das Leerzeichen am Ende des ersten Teilstring. Es ist notwendig, da beim Zusammenfügen der Zeichenketten keine zusätzlichen Leerzeichen eingefügt werden.

5.2 Arbeiten mit Strings

Offensichtlich erlaubt das Null–Zeichen zweierlei:

- Das Ende eines Strings lässt sich erkennen, ohne dass man seine Länge vorab kennen muss. Die Funktionen `scanf` und `printf` sind zum Beispiel darauf angewiesen, weil sie die Länge ihres Formatierungsstrings ja nicht mitgeteilt bekommen.
- Ein String kann seine Länge während der Ausführung des Programms dynamisch ändern, da er ohne weiteres kürzer sein darf als die Stringvariable, in der er gespeichert ist.

Dass die Umkehrung strikt verboten ist, dass also ein String (einschließlich des abschließenden Null–Zeichens) die Länge des Feldes nie übersteigen darf, ist klar: Da Stringvariablen letztlich Felder sind, gelten natürlich auch dieselben Regeln wie für Felder. Erneut ist ausschließlich der Programmierer selbst dafür verantwortlich, dass keine Indexüberschreitungen vorkommen.

Auch für das Anhängen des abschließenden Null–Zeichens an einen String ist, abgesehen von den Stringkonstanten und einigen Standardfunktionen, der Programmierer selbst verantwortlich.

Als erstes Beispiel betrachten wir eine Anweisungsfolge, die einen String kopiert, dabei seien `quelle` und `ziel` entsprechende Felder, wobei `ziel` groß genug ist um den zu kopierenden String aufzunehmen.

```
i = 0;
while (quelle[i] != '\0') {
  ziel[i] = quelle[i];
  i++;
}
ziel[i] = '\0';
```

Diese Formulierung ist für C allerdings ziemlich untypisch. In der Regel wird man

```
i = 0;
while (ziel[i] = quelle[i]) {
  i++;
}
```

finden. Das funktioniert, weil das Zeichen `'\0'` gerade die Ordnungszahl Null besitzt. Die Bedingung der `while`–Anweisung wird also gerade dann *falsch*, wenn bei der Wertzuweisung das Null–Zeichen übertragen wurde.

Die Vorteile der zweiten Formulierung gegenüber der ersten sind offensichtlich:

- Die Schleife terminiert erst, wenn das Null–Zeichen bereits übertragen ist, so dass es nicht hinter der Schleife besonders behandelt werden muss.
- In jedem Schleifendurchlauf wird nur zweimal auf eine Vektorkomponente zugegriffen und nicht dreimal.

5.3 Ein–/Ausgabe von Zeichen

Da die Verarbeitung von Zeichen und Strings ein wesentlicher Aspekt von C ist, gibt es in der Standardbibliothek, vermittelt durch `stdio.h`, spezielle Funtionen zu ihrer Ein– und Ausgabe. Wir wollen uns zunächst nur die Funktionen für Zeichen ansehen.

Die Funktion `getchar ()` liefert als Funktionswert das nächste Zeichen von der Standardeingabe – oder ggf. den Wert `EOF`. Damit es mit `EOF` keine Probleme gibt, muss der

Funktionswert einen Typ besitzen, in dem sich alle Zeichen und zusätzlich `EOF` darstellen lassen, und das ist gerade der Typ `int`. Entsprechend muss der Funktionswert bei der Zuweisung an eine Zeichenvariable umgewandelt werden. Nach der Zuweisung ist kein Vergleich mit `EOF` mehr möglich, da `EOF` nicht im Wertebereich von `char` liegt.

Die Umkehrung ist die Funktion `putchar (c)`. Sie schreibt das Zeichen `c` auf die Standardausgabe. Dabei ist zu beachten: `c` kann den Typ `int` besitzen oder wird (wenn es den Typ `char` oder `short` besitzt) in den Typ `int` umgewandelt. Die Funktion selbst wandelt diesen Wert in den Typ `unsigned char` zurück um, um das zu schreibende Zeichen zu bestimmen. Der Funktionswert ist das geschriebene Zeichen im Typ `int`, wenn kein Fehler aufgetreten ist, bzw. `EOF` sonst.

Als Beispiel betrachten wir eine Anweisungsfolge, die die Standardeingabe auf die Standardausgabe kopiert:

```
#include <stdio.h>

int c;
while ((c = getchar ()) != EOF) {
  putchar (c);
}
```

Dass diese Anweisungsfolge (bei der üblichen Zuordnung von Standardein– und –ausgabe) *kein* Durcheinander auf dem Bildschirm erzeugt, hat zwei simple Hintergründe: Zum einen arbeitet `getchar` wie `scanf` gepuffert, d.h. es wird zunächst das `ENTER` des Benutzers abgewartet und dann die eingegebene Zeile Zeichen für Zeichen abgearbeitet. Zum anderen liefert `getchar` das Zeilenende–Zeichen wie alle anderen Zeichen auch an das Programm ab, das es mit `putchar` wieder schreibt.

Mit der Funktion `getchar` haben wir jetzt übrigens auch eine Möglichkeit, bei der Eingabe mehrerer Zahlen den Benutzer dazu zu zwingen, jede Zahl einzeln mit `ENTER` abzuschließen: Nachdem wir eine Zahl in der Zeile interpretiert haben, überspringen wir in einer Schleife alle Zeichen bis zum Zeilenende:

```
#include <stdio.h>

#define ANZAHL 10               /* Anzahl Wiederholungen   */

int main(void)
{
  int i = ANZAHL, wert;         /* Variablendeklarationen  */

  printf("Geben Sie die Zahlen ein. "
         "Bitte jeweils mit ENTER abschliessen!\n");

  while (i--) {                   /* Schleife !            */
    scanf("%d", &wert);           /* eine Zahl lesen       */
    while (getchar () != '\n') {  /* Zeilenrest ignorieren */
    }
    printf("%d\n", wert);         /* gelesene Zahl schreiben */
  }
  return 0;
}
```

Quelltext 5.1: Kopieren von 10 Zahlen (Eine Zahl pro Zeile)

Dabei nutzen wir: `scanf` bleibt auf dem ersten Zeichen stehen, das *nicht* mehr interpretiert wurde, spätestens also auf dem Zeilenende–Zeichen. Das ist durchaus wesentlich für das Funktionieren des Programms: Die „Wegwerf"–Schleife wird ja mindestens einmal durchlaufen, holt also zumindest ein Zeichen – wenn sie kein Zeichen der „alten" Zeile mehr finden würde (hier also zumindest das Zeilenende–Zeichen), würde sie über das Betriebssystem eine neue Eingabezeile anfordern – und das wäre hier ein vollständig unerwünschter Effekt.

5.4 Ein–/Ausgabe von Strings

Für die Ein–/Ausgabe von Strings stehen spezielle Funktionen zur Verfügung. Wir haben aber bereits alternative Möglichkeiten kennengelernt: Sowohl mit den Funktionen `scanf` und `printf` als auch mit den Funktionen `getchar` und `putchar` können wir Strings lesen und schreiben. Zur Erinnerung: Der entsprechende Formatbeschreiber für Ein– und Ausgabe ist `%s`.

Sehen wir uns zunächst die Ausgabe an. Sie ist mit beiden Möglichkeiten recht einfach:

```
#define LAENGE ??

char str[LAENGE];
int  i;
/* ... Initialisierung von str ...              */

printf ("%s", str);               /* Variante 1 */

i = 0;                            /* Variante 2 */
while (str[i]) {
  putchar(str[i]);
  i++;
}
```

Bei der Schleife sieht man direkt, dass die Ausgabe nur dann korrekt funktioniert, wenn der String korrekt durch ein Null–Zeichen abgeschlossen ist. Beim Aufruf von `printf` sieht man das zwar nicht, es gilt aber ebenso.

Die Eingabe ist, wenn man sie naiv programmiert, ähnlich einfach:

```
#define LAENGE ??

char str[LAENGE];
int  i;

scanf ("%s", str);                /* Variante 1 */

i = 0;                            /* Variante 2 */
while ((str[i] = getchar ()) != '\n') {
  i++;
}
str[i] = '\0';
```

Ganz äquivalent sind beide Eingaben nicht:

- `scanf` überliest zunächst „white spaces" und liest danach Zeichen, bis ein erneutes „white space" gefunden wird, d.h. ein Leerzeichen, ein (horizontales) Tabulatorzeichen oder ein Zeilenende–Zeichen.

- In der Schleife wird auf jeden Fall von der aktuellen Position in der Zeile bis zum Zeilenende gelesen. Die Anweisung hinter der Schleife ersetzt das Zeilenende–Zeichen durch das Null–Zeichen.

„Naiv“ sind die beiden Realisierungen deshalb, weil in beiden Fällen nicht sichergestellt wird, dass der gelesene String in die Variable `str` hineinpasst.

Um den Aufruf von `scanf` enstprechend sicherer zu machen, benötigt man die erweiterten Möglichkeiten von Formatbeschreibern, wie sie in Kapitel 16 beschrieben werden. Die Schleife sollten wir jedoch geeignet modifizieren, was auch nicht übermäßig schwierig ist:

```
#define LAENGE ??

char str[LAENGE];
int  i;

i = 0;
while (i < LAENGE && (str[i] = getchar ()) != '\n') {
  i++;
}
if (i == LAENGE) {
  printf("Eingegebener String zu lang!\n");
  return 1;
}
else {
  str[i] = '\0';
}
```

5.5 Klassifizierung von Zeichen

Neben `stdio.h` stellen auch die Standardheaderdateien `ctype.h` und `string.h` Standardfunktionen zur Verarbeitung von Zeichen und Strings zur Verfügung. Vielmehr stehen zwei weitere Headerdateien zur Verfügung:

- Die Headerdatei `ctype.h` definiert Funktionen zur Klassifizierung von Zeichen und zur Umwandlung zwischen Groß– und Kleinbuchstaben.
- Die Headerdatei `string.h` definiert Funktionen mit denen man
  - Strings kopieren, konkatenieren oder vergleichen,
  - in Strings nach Zeichen oder Strings suchen oder
  - verschiedene andere Dinge erledigen kann.

Der Header `ctype.h` soll an dieser Stelle kurz behandelt werden, weil zum einen der Umfang der Headerdatei nicht groß, zum anderen die Funktionen sehr grundlegend und einfach sind.

Die Funktionen `tolower` und `toupper` wandeln Groß– in Kleinbuchstaben bzw. Klein– in Großbuchstaben um; andere Zeichen bleiben unverändert. Das Argument ist jeweils das ggf. umzuwandelnde Zeichen in `int`–Darstellung, der Funktionswert das resultierende Zeichen in `int`–Darstellung.

Die übrigen Funktionen haben als Argument ebenfalls ein Zeichen in `int`–Darstellung; ihr Funktionswert ist ungleich oder gleich Null und kann entsprechend als *wahr* oder *falsch* interpretiert werden, je nachdem, ob das Zeichen zu einer bestimmten Gruppe von Zeichen gehört oder nicht. Eine Liste der Funktionen ist in Tabelle 5.1 zu finden.

In alle Funktionen (einschließlich `tolower` und `toupper`) darf man übrigens als Argument auch `EOF` übergeben – und erhält dann `EOF` als Funktionswert zurück.

Funktion	Beschreibung
`islower`	Kleinbuchstabe
`isupper`	Großbuchstabe
`isalpha`	Buchstabe, klein oder groß
`isdigit`	Dezimalziffer
`isxdigit`	Hexadezimalziffer
`isalnum`	Buchstabe oder Ziffer
`iscntrl`	Steuerzeichen (nicht druckbar)
`isgraph`	druckbares Zeichen (ohne Leerzeichen)
`isprint`	druckbares Zeichen (einschließlich Leerzeichen)
`ispunct`	druckbares Sonderzeichen (ohne Leerzeichen)
`isspace`	„white space“

Tabelle 5.1: Funktionen zur Klassifizierung von Zeichen

Als Beispiel betrachten wir einen Programmausschnitt, in dem eine Hexadezimalzahl als Zeichenfolge eingelesen und in die entsprechende interne Darstellung umgewandelt wird. Wir gehen davon aus, dass die eingegebene Zahl nicht größer als `UINT_MAX` ist. Die Interpretation soll stoppen, sobald das erste nicht interpretierbare Zeichen gefunden wird. Diese Aufgabe könnte man zwar auch mit `scanf` lösen – es ist aber durchaus nützlich, wenn man sich einmal Gedanken darüber macht, was bei der Ausführung einer Standardfunktion tatsächlich passiert.

```
#include <stdio.h>
#include <ctype.h>
...
int c;
unsigned int x;

x = 0;
while ( isxdigit( c = getchar() ) )
{
  x = x * 16;
  if ( isdigit(c) )  {
    x = x + c - '0';
  }
  else {
    x = x + toupper(c) - 'A' + 10;
  }
}
```

Diese Anweisungsfolge leistet das Verlangte, falls in dem verwendeten Zeichensatz die zehn Ziffern und die Buchstaben `A` bis `F` jeweils unmittelbar aufeinanderfolgen. Das ist die einzige Voraussetzung. Ansonsten ist dieses Programmstück ohne weiteres portierbar auf Implementationen, denen z.B. nicht der ASCII–Zeichensatz zugrunde liegt.

Die vielen Klammern sind übrigens alle wirklich nötig:

- Die Klammern um die Bedingung der `while`–Anweisung und um die Argumentliste (das Argument) von `isxdigit` sind offensichtlich notwendig.
- Beim Aufruf einer Funktion muss das Klammernpaar auch dann folgen, wenn die Funktion keine Parameter besitzt. (Wenn Sie diese Klammern wegließen, würde der Compiler also „maulen“ – allerdings nicht wegen fehlender Klammern, sondern wegen inkompatibler Typen; die Hintergründe können wir erst später behandeln.)

Ein sachlicher Mangel der Anweisungsfolge soll nicht unerwähnt bleiben: Es ist keinerlei Sicherung gegen eine Bereichsüberschreitung eingebaut!

Kapitel 6

Steuerung des Programmablaufs

6.1 Anweisungen und Blöcke

Mit `while`, `if` und `return` haben wir bereits drei Anweisungen kennengelernt, mit denen sich der Ablauf eines Programms beeinflussen lässt. In diesem Abschnitt wollen wir uns die Anweisungen zur Steuerung des Programmablaufs im Zusammenhang ansehen.

Die einfachsten C–Anweisungen sind die *Ausdrucksanweisungen*. Sie bestehen aus einem Ausdruck, dem ein Semikolon folgt.

Beispiele:

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

Wie bereits erwähnt: Das Semikolon ist in C ein *Abschlusssymbol* und kein *Trennsymbol*, wie etwa in Pascal.

Mehrere Anweisungen können zu einem *Block* zusammengefasst werden, indem man sie in ein Paar *geschweifte Klammern* (`{ }`) einschließt. Ein solcher Block ist syntaktisch einer einzelnen Anweisung äquivalent. Wir haben Blöcke schon bei `while`– und `if`–Anweisung kennengelernt. Achtung: Nach der geschweiften Klammer, die einen Block abschließt, steht *kein* Semikolon!

Ein Block darf auch leer sein, d.h. zwischen seinen Klammern brauchen keine Anweisungen zu stehen. Er ergibt dann eine *leere Anweisung*, d.h. eine Anweisung, die nichts bewirkt. Eine leere Anweisung kann man auch durch aufeinanderfolgende Semikolons oder ein Semikolon als ersten Eintrag in einem Block erzeugen. Auch wenn die leere Anweisung auf den ersten Blick ziemlich nutzlos erscheint, ist sie gelegentlich doch sehr hilfreich: Wir haben bereits ein Beispiel (leeren der Eingabe) gesehen, bei dem die gesamte Operation einer Schleife in der Bedingung steckte, so dass der Anweisungteil der Schleife nur noch aus einer leeren Anweisung bestand:

```
while (getchar () != '\n') {
}
```

Vor der ersten Anweisung in einem Block können Variablenvereinbarungen und beliebige andere Deklarationen stehen. Für die Funktion `main` haben wir das bereits kennengelernt. Gleiches gilt aber auch z.B. für den Block einer Schleife.

6.2 Die `if`–Anweisung

Die `if`–Anweisung wurde bereits informell eingeführt. Ihre vollständige Syntax ist

```
if (bedingung)
     anweisung1
[ else
     anweisung2 ]
```

Was sie bewirkt, sollte inzwischen geläufig sein. Interessanter sind die zusätzlichen formalen Möglichkeiten:

Formal wird hinter dem `if` selbst und ggf. auch hinter dem `else` *genau eine* Anweisung verlangt. Diese Anweisung kann ein Block sein, wie in den bisherigen Beispielen, braucht es aber nicht. Speziell die `if`–Anweisung, mit der der bedingte Ausdruck beschrieben wurde, könnte man also sehr viel kürzer schreiben:

```
if (bedingung)
     x = ausdruck1;
else
     x = ausdruck2;
```

Zum anderen kann man das Schlüsselwort `else` samt nachfolgender Anweisung weglassen, wenn diese Anweisung leer ist – in der Syntaxbeschreibung durch die eckigen Klammern angedeutet. Das ist oft bequem, führt gelegentlich bei geschachtelten `if`–Anweisungen aber auch zu Problemen. Wir betrachten dafür ein Beispiel:

```
if (n > 0)
  if (n % 2)
    printf("positiv und ungerade\n");
else
  printf("nicht positiv\n");
```

So, wie die Anweisung aufgeschrieben ist, suggeriert sie folgenden Ablauf

- Wenn `n` positiv und ungerade ist, wird die entsprechende Meldung geschrieben.
- Wenn `n` positiv und gerade ist, passiert nichts.
- Wenn `n` nicht positiv ist, wird die entsprechende Meldung geschrieben.

Tatsächlich ist der Ablauf anders:

- Wenn `n` positiv und ungerade ist, wird, wie oben, die entsprechende Meldung geschrieben.
- Wenn `n` positiv und gerade ist, wird die Meldung “nicht positiv” geschrieben.
- Wenn `n` nicht positiv ist, passiert nichts.

Woran liegt das? Den Compiler interessiert es überhaupt nicht, wie man sein Programm aufschreibt; das Einrücken dient nur dazu, dem Leser den Überblick über die Struktur eines Programms zu erleichtern. Die Zuordnung eines `else` zu einem `if` durch den Compiler erfolgt vielmehr so, wie die Zuordnung einer schließenden zu einer öffnenden Klammer: Wird eine schließende Klammer gefunden, wird rückwärts gesucht, bis die letzte, passende öffnende Klammer gefunden wird, der noch keine schließende Klammer zugeordnet ist.

Wenn man mit einem Editor arbeitet, der den Quelltext automatisch einrückt, kann es einem übrigens nicht so leicht passieren, dass vermeintliche und tatsächliche Zuordnung nicht übereinstimmen.

Im Beispiel gibt es drei „Reparaturmöglichkeiten“. Die erste ist formaler Natur: Man vervollständigt die geschachtelte `if`–Anweisung durch einen `else`–Zweig:

```
if (n > 0)
  if (n % 2)
    printf("positiv und ungerade\n");
  else
    ;
else
   printf("nicht positiv\n");
```

Die zweite ist ebenfalls formaler Natur: Man macht die geschachtelte `if`–Anweisung zu einem Block – es ist klar, dass mit dem Ende eines Blocks eine `if`–Anweisung abgeschlossen sein muss, auch wenn noch kein `else` für sie da war:

```
if (n > 0) {
  if (n % 2)
    printf("positiv und ungerade\n");
}
else
  printf("nicht positiv\n");
```

Schließlich kann man, zumindest in diesem Beispiel, auch die Logik „umdrehen“:

```
if (n <= 0)
  printf ("nicht positiv\n");
else
  if (n % 2)
    printf("positiv und ungerade\n");
```

Dieses letzte Beispiel legt eine etwas veränderte Schreibweise nahe: Wenn die Anweisung hinter einem `else` ihrerseits eine `if`–Anweisung ist, liegt es nahe, beide Zeilen zu einer zusammenzufassen:

```
if (n <= 0)
  printf("nicht positiv\n");
else if (n % 2)
  printf("positiv und ungerade\n");
```

Die Form, in der man seine Programme aufschreibt, sollte ja, wie eben bereits angesprochen, schon auf den ersten Blick einen möglichst guten Überblick über die Struktur der Programme erlauben. Dazu dient unter anderem das Einrücken. Wenn allerdings zu viele Stufen oder innerhalb der Stufen zu weit eingerückt wird, schadet das wiederum der Übersichtlichkeit. Oft ist man auch zu Kompromissen gezwungen.

Nach so viel Theorie und formalen Beispielen nun noch ein konkretes Beispiel: Der Radius eines Kreises soll abgefragt und, je nach Wunsch, Kreisumfang oder Kreisfläche ausgegeben werden.

```
#include <stdio.h>

#define PI 3.14159

int main(void) {
  int antwort;
  double r;

  printf("Bitte geben Sie den Radius ein: ");
  scanf("%lf", &r);
  while (getchar () != '\n')
    ;
```

```
  printf("Soll der Umfang (u/U) oder die Flaeche (f/F) "
         "des Kreises berechnet werden? ");
  antwort = getchar();
  if (antwort == 'u' || antwort == 'U')
    printf("Umfang: %E\n", 2 * PI * r);
  else if (antwort == 'f' || antwort == 'F')
    printf("Flaeche: %E\n", PI * r * r);
  else
    printf("Falsche Eingabe!\n");

  return 0;
}
```

Quelltext 6.1: Berechnung von Kreisumfang oder Fläche

6.3 Die `switch`–Anweisung

Was im vorherigen Programmbeispiel realisiert ist, kommt oft vor: Abhängig von einem bestimmten Wert ist nicht nur aus zwei, sondern aus mehr Möglichkeiten auszuwählen. C kennt speziell dafür die `switch`–Anweisung

```
switch (ausdruck) {
  case wert1:
    anweisungfolge1
  case wert2:
    anweisungfolge2
  ...
  case wertN:
    anweisungfolgeN
[ default:
    anweisungfolgeD ]
}
```

Der Ablauf ist so: Der Ausdruck *ausdruck* wird ausgewertet. Ist das Resultat einer der einem `case` folgenden Werte *wert1*, ..., *wertN*, so wird die Ausführung mit der Anweisungsfolge fortgesetzt, die diesem Wert folgt. Kommt der Wert des Ausdrucks nicht vor, wird die Anweisungsfolge hinter `default` ausgeführt, wenn eine `default`–Klausel angegeben ist, bzw. sonst direkt hinter das Ende der `switch`–Anweisung verzweigt.

Vor einem Beispiel zunächst noch einige Anmerkungen zu den Formalien:

- Sowohl der Ausdruck *ausdruck* als auch die Werte *wert1*, ..., *wertN* müssen ganzzahlig sein. Zeichenkonstanten sind also erlaubt.
- Die Werte *wert1*, ..., *wertN* müssen sämtlich verschieden sein.
- Die Werte *wert1*, ..., *wertN* müssen Konstanten oder *konstante Ausdrücke* sein. Unter einem konstanten Ausdruck versteht man einen Ausdruck, dessen Operanden sämtlich Konstanten sind, in dem keine Funktionen aufgerufen werden und in dem keine Operatoren mit Nebeneffekten vorkommen. Der Hintergrund: Konstante Ausdrücke werden bereits durch den Compiler ausgewertet, sind bei der Ausführung des Programms also Konstanten.
- Die Reihenfolge, in der *wert1*, ..., *wertN* und ggf. `default` angegeben werden, ist beliebig. Bei der Reihenfolge, die man wählt, sollte man die Lesbarkeit des Programms allerdings besonders im Auge haben.

- Jeder case– und der default–Klausel darf eine *Anweisungsfolge* folgen, nicht nur eine einzelne Anweisung. Insbesondere entfällt damit die Notwendigkeit, Anweisungsfolgen durch geschweifte Klammern zu Blöcken zusammenzufassen.

Wir sehen uns jetzt ein Beispiel an. In einem Programm, das mit Wochentagen arbeitet, wird man die Wochentage mit Kennziffern identifizieren, etwa Sonnabend mit 0, Sonntag mit 1, usw.. Will man diese Kennziffern in Klarschrift umsetzen, so kann man das mit einer switch–Anweisung tun, auch wenn es gerade für dieses Beispiel günstigere Möglichkeiten gibt, wie wir noch sehen werden:

```
enum TAGE {SONNABEND, SONNTAG, ...};
...
switch (wochentag) {
  case SONNABEND:
    printf("Sonnabend\n");
  case SONNTAG:
    printf("Sonntag\n");
  ...
  case FREITAG:
    printf("Freitag\n");
  default:
    printf("Kennzahl unzulaessig\n");
}
```

Dieses Beispiel funktioniert für unzulässige Kennzahlen famos – und liefert für zulässige Kennzahlen nicht das beabsichtigte Ergebnis. Ist die Kennzahl etwa DONNERSTAG, so erhalten wir die drei Ausgabezeilen

```
Donnerstag
Freitag
Kennzahl unzulaessig
```

Diese Ausgabe zeigt auch bereits, was schief geht: Wenn die Anweisungsfolge für einen Fall ausgeführt ist, wird nicht automatisch hinter das Ende der switch–Anweisung verzweigt, sondern linear mit den Anweisungen für die nachfolgenden Fälle fortgefahren.

Abhilfe schafft die Anweisung break: Ihre Ausführung bewirkt, dass die Bearbeitung der switch–Anweisung sofort beendet und mit der ihr folgenden Anweisung fortgefahren wird.

Unser Beispiel müssen wir also so korrigieren:

```
switch (wochentag) {
  case SONNABEND:
    printf("Sonnabend\n");
    break;
  case SONNTAG:
    printf("Sonntag\n");
    break;
  ...
  case FREITAG:
    printf("Freitag\n");
    break;
  default:
    printf("Kennzahl unzulaessig\n");
    break;
}
```

Ob man hinter der letzten Klausel eine break–Anweisung schreibt oder nicht, ist reine Geschmackssache, sie hat auf den Programmablauf keinen Einfluss.

Als weiteres, konkretes Beispiel sehen wir uns noch einmal die Berechnung von Fläche bzw. Umfang eines Kreises an, jetzt mit einer `switch`–Anweisung realisiert:

```
#include <stdio.h>

#define PI 3.14159...

int main(void) {
  int antwort;
  double r;

  printf("Bitte geben Sie den Radius ein: ");
  scanf("%lf", &r);
  while (getchar () != '\n')
    ;

  printf("Soll der Umfang (u/U) oder die Flaeche (f/F) "
         "des Kreises berechnet werden? ");
  antwort = getchar();
  switch (antwort) {
    case 'u':
    case 'U':
      printf("Umfang: %E\n", 2 * PI * r);
      break;
    case 'f':
    case 'F':
      printf("Flaeche: %E\n", PI * r * r);
      break;
    default:
      printf("Falsche Eingabe!\n");
  }

  return 0;
}
```

Quelltext 6.2: Berechnung von Kreisumfang oder Fläche mit `switch`

6.4 Die `while`–Schleife

Die `while`–Schleife haben wir praktisch bereits vollständig kennengelernt:

```
while (bedingung)
  anweisung
```

Solange der Ausdruck *bedingung* wahr – d.h. ungleich Null – ist, wird die folgende Anweisung ausgeführt. Anzumerken ist hier, wie bei der `if`–Anweisung: Formal besteht der Rumpf der Schleife aus genau einer Anweisung. Diese kann ein Block sein, muss es aber nicht.

6.5 Die `do`–Schleife

Die zweite Schleifenanweisung ist die `do`–Anweisung

```
do
  anweisung
while (bedingung);
```

die wir im Falle eines Blocks auch als

```
do {
  anweisung
} while (bedingung);
```

schreiben, um die Verwechslung der `while`–Klausel mit einer eigenständigen `while`–Anweisung zu verhindern.

Im Prinzip sind sich `do`– und `while`–Schleife sehr ähnlich. Der Unterschied zwischen den beiden Schleifen ist, dass bei einer `do`–Schleife sofort der Rumpf ausgeführt und erst danach die Bedingung überprüft wird, während bei einer `while`–Schleife die erste Operation überhaupt eine Überprüfung der Bedingung ist.

Trotzdem lassen sich beide Schleifenanweisungen ohne weiteres durcheinander ersetzen. So ist die Konstruktion

```
do
  anweisung
while (bedingung);
```

der Konstruktion

```
anweisung;
while (bedingung)
  anweisung
```

äquivalent, wobei die Formulierung als `do`–Schleife „natürlicher“ wirkt. Umgekehrt ist die Konstruktion

```
while (bedingung)
  anweisung
```

der Konstruktion

```
if (bedingung)
  do
    anweisung
  while (bedingung);
```

äquivalent, wobei hier die Formulierung als `while`–Schleife „natürlicher“ wirkt. Entsprechend sollte man im Einzelfall diejenige der beiden Schleifenanweisungen verwenden, die die „natürlichere“ Formulierung erlaubt.

In der Praxis kommen `while`–Schleifen häufiger als `do`–Schleifen vor, weil man in der Regel zunächst zu prüfen hat, ob der Schleifenrumpf überhaupt ausgeführt werden darf, bevor man ihn ausführt. Das Kopieren eines String ist insoweit eine Ausnahme, weil ein Zeichen, nämlich das abschließende Null–Zeichen, auf jeden Fall kopiert werden muss. Ein typisches Beispiel für `do`–Schleifen ist auch die Kommunikation mit dem Benutzer: Zunächst muss er seine Eingabe vornehmen – gibt er Unsinn ein, muss die Abfrage nach einer Ermahnung wiederholt werden.

Als Beispiel betrachten wir noch einmal die Aufgabe, vom Benutzer eine positive ganze Zahl zu erfragen. Als wir die Aufgabe zuerst gelöst hatten, hatten wir den „falschen“ Anfangswert Null für die Eingabevariable gesetzt, damit die Bedingung der Schleife anfangs den Wert *falsch* hatte. Darauf können wir jetzt verzichten. (Und auch sonst erlauben uns unsere erweiterten Kenntnisse eine ganze Reihe von Änderungen an der damaligen Lösung.)

```
#include <stdio.h>

int main(void) {
  int wert;                          /* Variablendekl.     */

  printf("Geben Sie die Zahl ein:  ");

  do {
    if (scanf("%d", &wert) != 1) {   /* Zahl lesen         */
      printf("Keine Lust mehr?\n");  /* auf EOF/Fehler     */
      return 1;                      /*   reagieren        */
    }
    if (wert <= 0)                   /* Eingabe zulaessig? */
      printf("Falsch - noch einmal!  ");      /* nein !     */
  } while (wert <= 0);

  printf("Einverstanden!\n");                /* ja !       */
  return 0;
}
```

Quelltext 6.3: Auswahl aus Alternativen (neu)

6.6 Die `for`–Schleife

Die dritte Schleifenanweisung in C ist die `for`–Anweisung

```
for (ausdruck1; bedingung; ausdruck2)
  anweisung
```

Der Ablauf lässt sich am einfachsten durch eine `while`–Schleife beschreiben:

```
ausdruck1;
while (bedingung) {
  anweisung
  ausdruck2;
}
```

Die *Werte* von *ausdruck1* und *ausdruck2* werden nicht verwendet. Man nutzt hier nur die Nebeneffekte, die bei der Berechnung der Ausdrücke auftreten, z.B. beim Auswerten des Zuweisungs– oder Inkrementierungsoperators. *ausdruck1* dient normalerweise der Initialisierung und *ausdruck2* der Aktualisierung der Schleifenvariablen, wie die Übersetzung zur `while`–Schleife bereits angedeutet hat.

In anderen Programmiersprachen gibt es in der Regel eine klare Unterscheidung, wann eine `while`– und wann eine `for`–Schleife einzusetzen ist:

- `for`–Schleifen können nur dann eingesetzt werden, wenn die Anzahl der Schleifendurchläufe bereits beim Eintritt in die Schleife bekannt ist, weil die Ausdrücke *ausdruck1*, *bedingung* und *ausdruck2* nur einmalig ausgewertet und aus ihnen, ebenfalls einmalig, die Anzahl der Schleifendurchläufe berechnet wird. Vor einer eventuellen Wiederholung wird dann nur noch diese (intern gespeicherte) Anzahl geprüft und modifiziert.
- Immer dann, wenn die Wiederholungsbedingung im Rumpf der Schleife modifiziert werden soll, *muss* man eine `while`–Schleife verwenden.

In C ist man zu dieser Unterscheidung zwar nicht gezwungen, wie wir gesehen haben. Es spricht aber manches dafür, sich freiwillig daran zu halten.

Typische Beispiele für den Einsatz der for–Anweisung sind in diesem Sinne Operationen auf Feldern. Das Beispiel, in dem das Skalarprodukt von zwei Vektoren berechnet wird, können wir etwa mit for–Anweisungen sehr viel kompakter formulieren, ohne dass die Lesbarkeit darunter leidet; ganz im Gegenteil – in diesem Fall verbessern die for–Anweisungen die Lesbarkeit sogar erheblich:

```
#include <stdio.h>

#define LAENGE 5

int main(void) {
  double v1[LAENGE], v2[LAENGE], prod;
  int i;

  printf("1. Vektor eingeben!\n");     /* 1. Vektor lesen   */
  for (i = 0; i < LAENGE; i++)
    scanf("%lf", &v1[i]);

  printf("2. Vektor eingeben!\n");     /* 2. Vektor lesen   */
  for (i = 0; i < LAENGE; i++)
    scanf("%lf", &v2[i]);

  prod = 0;
  for (i = 0; i < LAENGE; i++)         /* Skalarprodukt     */
    prod += v1[i] * v2[i];             /*    berechnen und  */
                                       /*    ausgeben       */
  printf("Das Skalarprodukt ist: %f\n", prod);

  return 0;
}
```

Quelltext 6.4: Arbeit mit Vektoren: Skalarprodukt mit for–Schleife

Die Schrittweite einer for–Schleife muss nicht notwendig +1 (oder −1) sein. Durch

```
long potenz;

for (potenz = 1; potenz <= 100000; potenz *= 5)
  printf("%ld\n", potenz);
```

werden zum Beispiel alle Potenzen von 5 ausgegeben, die kleiner oder gleich 100000 sind.

Die drei Ausdrücke einer for–Klausel sind optional, können also auch weggelassen werden. Nur die Semikolons, die die Ausdrücke trennen, müssen stets geschrieben werden. Man könnte also zum Beispiel

```
long potenz = 1;

for (; potenz <= 100000; potenz *= 5)
  printf("%ld\n", potenz);
```

oder auch

```
long potenz = 1;
```

```
for (; potenz <= 100000;) {
  printf("%ld\n", potenz);
  potenz *= 5;
}
```

schreiben. Insbesondere im letzten Beispiel sollte man sich aber überlegen, ob nicht doch eine `while`–Schleife die angemessenere Realisierung ist.

Auch Endlosschleifen kann man erzeugen, die fehlende Bedingung wird hier als „wahr“ interpretiert:

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

Gelegentlich hat man Schleifen mit zwei (oder mehr) Laufindizes. Wollen wir etwa die Reihenfolge der Komponenten eines Vektors umdrehen, so ist es zweckmäßiger, mit zwei Laufindizes zu arbeiten, als aus Länge und einem Index den anderen Index jeweils neu zu berechnen. Enthält die Variable `n` die Länge des Vektors, so können wir schreiben

```
for (i = 0, j = n - 1; i < j; i++, j--)
  h = v[i], v[i] = v[j], v[j] = h;
```

Hier zeigt sich, wie man den Kommaoperator für kompakte Formulierungen einsetzen kann.

Daneben bieten `for`–Anweisungen jede Menge Möglichkeiten zu Verstößen gegen den „guten Ton“! Zum Beispiel kann man die Laufvariable im Rumpf der Schleife ändern. Überlegen Sie sich einmal, was

```
for ( i = 0; i < 100; i += 10) {
  printf("%d\n", i);
  i /= 2;
}
```

als Ausgabe liefert (oder probieren Sie es aus!).

6.7 Sprünge

Anweisungen zur Steuerung des Programmablaufs realisieren grundsätzlich *Sprünge* (*Verzweigungen*), d.h. Unterbrechungen des linearen Programmablaufs: Bei einer vollständigen `if`–Anweisung zum Beispiel wird eine der beiden Anweisungen „übersprungen“; bei einer Schleife wird ggf. vom Ende des Rumpfes zu seinem Anfang „zurückgesprungen“; die `return`–Anweisung bewirkt einen „Rücksprung“ in die rufende Routine.

Wenn man ein Programm liest, wird man bei `if`–, `switch`– oder Schleifenanweisungen an die Auswahl von Alternativen oder Schleifen denken und kaum an die in ihnen steckenden Sprünge. Solche Sprünge werden entsprechend auch als *implizite* Sprünge bezeichnet.

`return`– und `break`–Anweisung sind dagegen *explizite* Sprunganweisungen, d.h. ihr einziger oder zumindest wesentlicher Zweck ist das Ausführen eines Sprunges. Beide Anweisungen sind, wie auch die noch einzuführende `continue`–Anweisung, strukturbezogene Sprunganweisungen, d.h. das Ziel des Sprunges ergibt sich aus der Struktur des Programms und kann nicht frei gewählt werden.

Die vierte explizite Sprunganweisung, die `goto`–Anweisung, erlaubt dagegen die Wahl beliebiger Sprungziele innerhalb einer Funktion. Sie ist bei intensiver Verwendung ein sicheres Mittel, undurchschaubaren Code zu schreiben. Da man in C grundsätzlich ohne `goto`–Anweisungen auskommen kann, wird sie hier nicht weiter behandelt.

Auf die return–Anweisung kommen wir im Zusammenhang mit Funktionen noch einmal zurück.

6.8 Die break–Anweisung

Die break–*Anweisung* haben wir im Zusammenhang mit der switch–Anweisung kennengelernt. Außerdem darf sie in Schleifen aller Art verwendet werden und bewirkt dann die *sofortige* Beendigung der Schleife. Im Innern geschachtelter Schleifen wird immer nur die innerste Schleife beendet, während die umgebenden normal weiter abgearbeitet werden.

Nützlich ist die break–Anweisung in Schleifen vor allem dann, wenn das logische Ende in der Mitte des Schleifenrumpfes liegt. Ein Beispiel haben wir bereits kennengelernt, nämlich die Abfrage einer positiven ganzen Zahl. Hier muss in der Schleife zunächst gelesen und ggf. die Eingabe von EOF behandelt werden. War die Eingabe korrekt, ist die Schleife jetzt logisch zu Ende. Eingetragen werden muss aber noch die Behandlung unkorrekter Eingaben. Mit einer break–Anweisung kann man die Aufgabe so lösen:

```
#include <stdio.h>

int main(void) {
  int wert;                                /* Variablendekl.  */

  printf("Geben Sie die Zahl ein:  ");     /* Zahl anfordern  */

  do {
    if (scanf("%d", &wert) != 1) {         /* Zahl lesen      */
      printf("Keine Lust mehr?\n");        /* auf Abbruch     */
      return 1;                            /*      reagieren  */
    }
    if (wert > 0)                          /* zulaessig ?     */
      break;                               /*  ja !           */
    printf("Auf Wiedersehen!");            /*  nein !         */
  } while (1);

  printf("Einverstanden!\n");              /* alles klar !    */
  return 0;
}
```

Quelltext 6.5: Auswahl aus Alternativen (mit break)

Da das Ende der Schleife jetzt auch formal in der Mitte ihres Rumpfes liegt, spricht manches dafür, die Schleife als Endlosschleife zu formulieren.

Wenn das logische Ende einer Schleife mit ihrem formalen Ende übereinstimmt, kommt man in der Regel ohne größere „Klimmzüge“ auch ohne break–Anweisungen aus. Wollen wir etwa suchen, ob ein bestimmter Wert in einem Vektor enthalten ist, und ggf. bestimmen, an welcher Stelle er steht, so ist vielleicht die Formulierung

```
for (i = 0; i < N; i++)
  if (a[i] == x)
    break;
```

naheliegend. Lösen lässt sich die Aufgabe aber auch durch

```
for (i = 0; i < N && a[i] != x; i++)
  ;
```

wobei wir die Regeln für die Behandlung des Operators `&&` bewusst nutzen. Auch in vielen anderen Fällen kommt man ohne `break`–Anweisungen aus.

6.9 Die `continue`–Anweisung

Erlaubt es die `break`–Anweisung, eine Schleife sofort zu beenden, so erlaubt die `continue`–*Anweisung*, die momentane Abarbeitung eines einzelnen Schleifendurchlaufs zu unterbrechen und sofort zur Prüfung des Endkriteriums für die Schleife überzugehen; vom Ergebnis der Prüfung hängt es dann ab, ob die Schleife beendet oder ein weiterer Schleifendurchlauf ausgeführt wird.

Die `continue`–Anweisung wird nicht sehr oft verwendet. Sie lässt sich auch vermeiden. Sehen wir uns das an einer `while`–Schleife einmal an. Statt

```
while (bedingung1) {
  anweisungsfolge1
  if (bedingung2)
    continue;
  anweisungsfolge2
}
```

können wir ohne weiteres schreiben

```
while (bedingung1) {
  anweisungsfolge1
  if (!bedingung2) {
    anweisungsfolge2
  }
}
```

Man sieht hier auch direkt, was die `continue`–Anweisung leistet: Sie verringert die Schachtelungstiefe – mehr nicht.

6.10 Beispiel

Zum Abschluss des Abschnittes über die Ablaufsteuerung wollen wir uns ein etwas ausführlicheres Beispiel ansehen.

Es soll eine Dezimalziffer gelesen werden. Zu dieser Ziffer sollen alle positiven ganzen Zahlen kleiner als 100 ausgegeben werden, bei denen die Ziffer sowohl in der dezimalen Darstellung der Zahl selbst als auch in der ihres Quadrats vorkommt. Für die Ziffer 2 als Eingabe soll das Programm zum Beispiel diese Ausgabezeilen liefern:

```
23 529
25 625
27 729
32 1024
52 2704
82 6724
```

Es sollte klar sein, dass die obere Schranke als benannte Konstante deklariert wird. Die Wiederholung der Berechnung soll möglich sein, ohne dass dazu das Programm neu gestartet wird. Unzulässige Eingaben sollen (müssen!) zurückgewiesen werden.

Das Programm kann dann so aussehen:

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

#define GRENZE 100

int main(void) {
  int ziffer, c, i, j;

  printf("Geben Sie eine Ziffer ein, "
         "abgeschlossen  mit ENTER: ");

  while (!isdigit(ziffer = c = getchar())
         || (c = getchar()) != '\n') {
    while (c != '\n')
      c = getchar();
    printf("Nur die Ziffern 0 bis 9 sind "
           "erlaubt! Nochmal: ");
  }
  ziffer -= '0';

  printf("Folgende Zahlen < %d erfuellen das "
         "Kriterium:\n", GRENZE);

  for (i = 1; i < GRENZE; i++) {
    for (j = i; j > 0 && j % 10 != ziffer; j /= 10)
      ;
    if (j == 0)
      continue;

    for (j = i*i; j > 0 && j % 10 != ziffer; j /= 10)
      ;
    if (j > 0)
      printf("%d^2 = %d\n", i, i * i);
  }

  return 0;
}
```

Quelltext 6.6: Ziffernübereinstimmungen

Kapitel 7

Funktionen

7.1 Motivation

Je umfangreicher und komplexer ein Problem ist, desto schwieriger ist es zu lösen. Deshalb versucht man, Programmieraufgaben in verschiedene Teile zu zerlegen, die man ihrerseits wieder in Teilprobleme zerlegt, usw. Dabei achtet man besonders darauf, dass zwischen den Teilproblemen möglichst wenig Querverbindungen bestehen. Dieses Vorgehen hat drei Vorteile:

- Jedes Teilproblem ist viel einfacher zu lösen als das Gesamtproblem.
- Die Teilprobleme können von verschiedenen Programmierern gelöst weren.
- Wenn man „Glück“ hat, besitzt man schon Lösungen für einzelne Teilprobleme.

In der Terminologie des Programmierens entspricht eine solche Zerlegung eines Problems der Zerlegung eines Programms in Unterprogramme. C bezeichnet diese Unterprogramme grundsätzlich als *Funktionen*.

Wir haben bereits Funktionen kennengelernt und verwendet, nämlich die Bibliotheksfunktionen `printf`, `scanf`, `isspace`, usw. Auch das Hauptprogramm `main` ist in der Terminologie von C eine Funktion, die vom Betriebssystem beim Start des Programms aufgerufen wird.

7.2 Vereinbarung von Funktionen

Die Vereinbarung einer Funktion besteht in C aus zwei Teilen:

- In einer *Deklaration* werden die Eigenschaften einer Funktion bekanntgemacht: Name, Parameterliste und Typ des Funktionswerts.
- In der *Definition* werden die Operationen festgelegt, Speicherplatz reserviert, usw.

Es ist offensichtlich, dass die Definition einer Funktion ihrer Deklaration entsprechen muss.

Die Funktionsdeklaration hat in ANSI–C die Form

typ name `(`*typ1 [parameter1]*`, ... ,` *typN [parameterN]*`);`

und wird als *Prototyp* bezeichnet.

Für Funktionsnamen gelten dieselben Regeln wie für Namen von Variablen und Konstanten. Wenn eine Funktion keine Parameter besitzt, *muss* dieses durch das Schlüsselwort `void` gekennzeichnet werden.

Beispiel: Wir wollen eine Funktion schreiben, die die Potenz x^y einer `double`–Zahl x und einer ganzen Zahl y berechnet ($y \geq 0$). Der Prototyp kann dann die Form

```
double potenz (double basis, int exponent);
```

oder

```
double potenz (double, int);
```

haben. Obwohl auch die zweite Deklaration den Intentionen von ANSI–C vollständig entspricht, wie wir noch sehen werden, ist doch die erste Deklaration vorzuziehen, weil sie nicht nur die Typen der Parameter, sondern durch die Wahl der Namen auch deren Bedeutung beschreibt.

In der Definition einer Funktion werden die Operationen festgelegt, die beim Aufruf der Funktion durchzuführen sind. Die Funktionsdefinition hat die Form

```
typ name (typ1 parameter1, ..., typN parameterN) {
  lokale deklarationen
  anweisungsfolge
}
```

Man beachte: In der Definition einer Funktion wird der *Funktionsheader nicht* wie in der Deklaration durch ein Semikolon abgeschlossen!

Als erstes Beispiel realisieren wir die bereits angesprochene Funktion `potenz`:

```
double potenz(double basis, int exponent) {
  double pot = 1;

  for (; exponent > 0; exponent--)
    pot *= basis;

  return pot;
}
```

Als weiteres Beispiel betrachten wir eine Funktion `strlen`, die die Länge eines String bestimmt, ohne das abschließende Null–Zeichen mitzuzählen:

```
int strlen(char s[]) {
  int i;

  for (i = 0; s[i] != '\0'; i++)
    ;

  return i;
}
```

Dabei haben wir benutzt, dass sich die Länge des übergebenen Strings aus seinem Inhalt ergibt: Das Null–Zeichen zeigt das Ende an. Im allgemeinen muss bei Feldern als Funktionsparameter jedoch die Länge des Feldes übermittelt werden, z.B. durch einen seperaten Parameter.

Zwei Anmerkungen noch zu grundlegenden Formalien:

- Eine Funktionsdefinition darf nicht innerhalb einer anderen Funktionsdefinition stehen, d.h. C erlaubt keine Schachtelung von Funktionsdefinitionen. Für Funktionsaufrufe gilt das nicht, wie wir bereits gesehen haben: Funktionen können ohne weiteres andere Funktionen aufrufen.
- Deklarationen, die innerhalb eines Funktionsrumpfes vereinbarten werden, (z.B. in den beiden Beispielen jeweils die Variable `i`) sind in der Regel lokal, d.h. sie sind nur innerhalb der jeweiligen Funktion bekannt. Dass diese Einschränkung äußerst sinnvoll ist, werden wir später noch im einzelnen sehen.

7.3 Funktionswerte

Welchen *Funktionswert* eine Funktion an die rufende Funktion zurückliefert, wird durch die `return`–Anweisung

```
return [ausdruck];
```

festgelegt. Stimmt der Typ des Ausdrucks *ausdruck* nicht mit dem Typ der Funktion überein, erfolgt eine Typumwandlung wie bei einer Wertzuweisung.

Wie bereits erwähnt, ist die pauschale Bezeichnung „Funktion“ für alle Unterprogramme eigentlich „Etikettenschwindel“ – eine Funktion braucht nämlich durchaus keinen Funktionswert zurückzuliefern! Dieses kennzeichnet man in Deklaration und Definition, indem man als Typ des Funktionswertes das Schlüsselwort `void` angibt. Außerdem entfällt in der `return`–Anweisung der Ausdruck; eine `return`–Anweisung ohne Ausdruck direkt vor der schließenden Klammer des Funktionsrumpfes kann man sogar ganz weglassen, da die Kontrolle bei ihrem Erreichen auf jeden Fall an die aufrufende Funktion zurückgeht.

Ein triviales Beispiel für eine Funktion, die weder einen Parameter besitzt noch einen Funktionswert liefert, ist die „leere“ Funktion

```
void nichts_tun(void) {
  return;
}
```

Ein ähnlich triviales Beispiel ist eine Funktion, die ein mehrfaches Piepen bewirkt (falls das Terminal dazu imstande ist). Sie hat als Parameter die Anzahl der Piep's und liefert ebenfalls keinen Funktionswert:

```
void piepen(int anzahl) {
  while (anzahl--)
    putchar ('\a');
}
```

Gibt eine Funktion, deren Funktionswert nicht den Typ `void` besitzt, keinen Funktionswert zurück, weil die `return`–Anweisung oder auch nur der Ausdruck hinter `return` fehlt, so ist das in der Regel ein *schwerer Fehler*. Allerdings: Der Standard schreibt nicht vor, wie ein Compiler hierauf zu reagieren hat.

Faktisch ist die Angabe des Typs einer Funktion in Deklaration und Definition optional: Bei jeder Funktion, bei der die explizite Angabe des Typs fehlt, nimmt der Compiler den Typ `int` an! Der Hintergrund ist, dass man bei Compilern (und nicht nur bei ihnen) nach Möglichkeit versucht, *Aufwärtskompatibilität* zu wahren, d.h. ein Compiler nach neuem Standard soll auch Programme nach altem Standard übersetzen, ohne dass Änderungen erforderlich werden. Leider können durch dieses Prinzip alle Absicherungen, die der ANSI–Standard vorsieht, unterlaufen werden. Programmierer – gerade Programmieranfänger – sind gut beraten, sich strikt an die Vorgaben des ANSI–Standards in strengster Auslegung zu halten. Moderne Compiler unterstützen dies durch entsprechende Optionen. Ein solches Vorgehen sichert Portabilität und hilft enorm bei der Fehlersuche.

7.4 Aufruf von Funktionen

Der *Aufruf* einer Funktion besteht aus dem Namen der Funktion, dem, in runde Klammern eingeschlossen, die Liste der *Funktionsargumente* folgt. Das sind die Werte, die die Funktion während ihrer Ausführung als Parameter haben soll.

Funktionsaufrufe kennen wir bereits aus dem ersten Programmbeispiel. Unsere Funktionsbeispiele aus diesem Abschnitt könnten wir etwa durch

```
i = potenz(2.0, 5);
j = strlen("Wie lang ist dieser String?");
nichts_tun();
```

aufrufen. Durch ihren Aufruf geht die Kontrolle auf die Funktion über, bis diese eine weitere Funktion aufruft oder die Kontrolle an die rufende Funktion zurückgibt.

Zulässig, wenn auch sinnlos, wären ebenso die Aufrufe

```
potenz(2.0, 5);
strlen("Wie lang ist dieser String?");
```

nicht jedoch

```
z = nichts_tun();
```

Liefert eine Funktion einen Funktionswert, so darf man diesen ignorieren; bei einer Funktion, die keinen Funktionswert liefert, kann man auf einen Funktionswert natürlich auch nicht zugreifen.

7.5 Parameter und Argumente

Wir haben bereits gesehen, wie die Kommunikation zwischen rufender und gerufener Funktion erfolgt: In der Definition einer Funktion können *Parameter* (auch: *formale Parameter*) definiert werden. Diese Parameter stehen innerhalb der Funktion wie lokale Variablen des entsprechenden Typs zur Verfügung. Welche Werte sie haben, wird beim Aufruf der Funktion durch die *Argumente* (auch: *Aktualargumente* genannt) festgelegt.

Die Argumente werden den Parametern linear zugeordnet; die einzelnen Zuordnungen erfolgen wie Wertzuweisungen. Damit sollte offensichtlich sein, dass die Anzahl der Argumente mit der Anzahl der Parameter übereinstimmen muss und dass die Typen der Argumente ggf. wie bei einer Wertzuweisung in die Typen der Parameter umgewandelt werden. Die eventuellen Typumwandlungen schreibt der ANSI–Standard vor.

Betrachten wir unter diesem Aspekt noch einmal den Funktionsheader der Funktion `potenz` von oben

```
double potenz(double basis, int exponent);
```

und ihren Aufruf

```
potenz(a, j)
```

Die Funktion `potenz` hat die Parameter `basis` und `exponent`. Beim Aufruf wird `basis` der Wert von `a` und `exponent` der Wert von `j` zugewiesen.

Die Parameter einer Funktion entsprechen lokalen Variablen des jeweiligen Typs, in die beim Aufruf die Werte der Argumente kopiert werden. Man bezeichnet das als „*call by value*“. Die Konsequenzen müssen wir uns noch einmal klar machen: Eine Veränderung eines Parameters innerhalb der Funktion hat *keinerlei* Auswirkungen auf die Umwelt der Funktion! Wir haben dieses im Zusammenhang mit der Funktion `potenz` bereits doppelt genutzt:

- Die Funktion verwendet den Parameter `exponent` als Schleifenzähler und verändert ihn dabei. Das hat, wenn wir die Funktion durch

  ```
  potenz (a, j)
  ```

  aufrufen, aber keinerlei Folgen für die Variable `j` der rufenden Funktion, weil ja nur eine Kopie des Wertes von `j` an `potenz` übergeben wird.
- Der Aufruf

```
i = potenz(2.0, 5);
```

ist nur deshalb zulässig, weil die Funktion nicht mit den Argumenten selbst arbeitet, sondern mit lokalen Variablen, in die die Werte der Argumente kopiert werden. Würde die Funktion mit den Argumenten selbst arbeiten, wäre der Aufruf unzulässig, weil das zweite Argument, das in der Funktion verändert wird, als Konstante keine Adresse besitzt.

Nicht in jedem Falle ist es zweckmäßig und wünschenswert, dass Funktionen ihre Parameter nicht verändern können. Mit der Funktion `scanf` kennen wir auch bereits ein Beispiel für eine solche Funktion: Sie soll ja gerade in den Variablen, die als Argumente angegeben werden, die gelesenen Werte speichern! Entsprechend müssen in solchen Fällen die *Adressen* der Argumente anstelle ihrer *Werte* übergeben werden. Wir haben auch bereits gesehen, wie man genau das mit dem Adressoperator `&` erreichen kann.

Übergabe von Adressen bezeichnet man allgemein als „*call by reference*“. Von C wird behauptet, dass es nur „call by value“ kenne, nicht jedoch „call by reference“. Der Hintergrund ist ein kleiner sprachlicher „Klimmzug“: Schreiben wir für eine `int`–Variable etwa

```
scanf("%d", &i)
```

so betrachtet C `&i` nicht als „Adresse der Variablen `i`“, sondern als „Ausdruck mit einem Zeigertyp“ – und was dann übergeben wird, ist eben gerade der Wert dieses Ausdrucks.

Auf die Zeigertypen und Zeiger kann an dieser Stelle noch nicht im einzelnen eingegangen werden. Zunächst nur ein Hinweis: Es reicht natürlich *nicht* aus, einen Zeiger an eine Funktion zu übergeben, um ihr die Veränderung ihrer Parameter zu ermöglichen. Auch die Vereinbarung der Parameter und der Zugriff auf sie müssen entsprechend formuliert sein. Wir werden darauf später noch ausführlich zurückkommen.

7.6 Felder als Parameter

Felder nehmen als Parameter (und Argumente) eine Sonderstellung ein. In der Terminologie von C gilt: Der Name eines Feldes ist eine Zeigerkonstante, deren Wert die Adresse der ersten Komponente des Feldes ist.

Die Konsequenz ist, dass bei Feldern nie eine Kopie der Werte der Komponenten an eine Funktion übergeben wird, sondern stets nur die (Anfangs–)Adresse des Feldes. Sieht man also bei einer Funktion ein Feld als Parameter vor, so muss man darauf achten, dass man nicht unerwünschte Nebeneffekte durch die Veränderung von Feldkomponenten erzeugt.

Als Beispiel für eine Funktion mit einem Feld als Parameter haben wir bereits die Funktion `strlen` betrachtet. Sie greift zwar auf die Komponenten ihres Parameters zu, verändert sie aber nicht; Nebeneffekte treten also nicht auf.

Als weiteres Beispiel wollen wir jetzt eine Funktion betrachten, die einen String invertiert. Die Anweisungsfolge, die diese Aufgabe löst, haben wir bereits betrachtet. Jetzt wollen wir sie als Funktion formulieren. Die Funktion `strlen` werden wir verwenden, um die Länge des String zu bestimmen. Außerdem ist ein Rahmenprogramm zum Lesen und Schreiben der Strings angeben.

```
#include <stdio.h>

#define LAENGE 20

/**  Prototypen  ****************************************/
```

```
void invert(char str[]);
int strlen(char str[]);

/*= Rahmenprogramm ========================================*/
int main(void) {
  char string[LAENGE + 1];
  int anzahl;

  printf("String eingeben (max. %d Zeichen):\n", LAENGE);
  scanf("%s", string);
  invert(string);
  printf("Der invertierte String ist \'%s\'\n", string);

  return 0;
}

/*= Funktionen ============================================*/

/*- Invertieren eines String -------------------------------*/
void invert(char str[]) {
  int i, j, n;
  char c;

  n = strlen(str);
  for (i = 0, j = n - 1; i < j; i++, j--)
    c = str[i], str[i] = str[j], str[j] = c;
}

/*- Bestimmung der Laenge eines String --------------------*/
int strlen(char str[]) {
  int i = 0;

  while (str[i] != '\0')
    i++;

  return i;
}
```

Quelltext 7.1: Invertieren eines Strings

Einige Anmerkungen dazu:

- Das Rahmenprogramm „verlässt“ sich darauf, dass der Benutzer keinen zu langen String eingibt; dieser String darf keine „white spaces“ enthalten.
- Das Hauptprogramm ist wie üblich als erstes angegeben und erst danach die Funktionen.
- Die Funktionsdeklarationen von `invert` und `strlen` stehen *vor* dem Hauptprogramm. Auch das ist üblich. Wir hätten auch `invert` innerhalb des Hauptprogramms und `strlen` innerhalb von `invert` deklarieren können; das hätte zur Folge gehabt, dass `invert` nur innerhalb des Hauptprogramms und `strlen` nur innerhalb von `invert` zur Verfügung steht.

Übrigens: Nur Feldnamen werden als Zeigerkonstanten betrachtet; für Feldkomponenten gilt das nicht. Rufen wir die Funktion `potenz` etwa durch

```
a = potenz(t[k], 4);
```

auf, so wird der Wert von `t[k]` übergeben und nicht die Adresse.

7.7 Das Attribut `const`

Wir haben eben gesehen, dass eine Funktion bei einem Feld, das als Argument übergeben wird, stets den direkten Zugriff auf die Originaldaten hat und diese ggf. verändern kann.

Das ist ziemlich unschön, weil der Programmierer sehr diszipliniert arbeiten muss, damit nicht ungewollte und ungewünschte Veränderungen erfolgen. War Disziplin früher das einzige Mittel, Nebeneffekte zu verhindern, so gibt der Standard dem Programmierer Unterstützung: Versieht man einen Parameter mit dem Attribut `const`, so erzeugt der Compiler bei jeder Zuweisung an den Parameter, bei Feldern auch an eine Komponente, eine Fehlermeldung. Wohlgemerkt: Der Compiler meldet den Fehler nur; er sorgt nicht automatisch dafür, dass eine Kopie angelegt wird.

Bei Parametern ist das Attribut `const` zunächst nur für Felder sinnvoll, weil man bei „einfachen“ Variablen ohnehin nur eine Kopie des Wertes erhält, die man unbesorgt verändern kann. Die Funktion `strlen` sollte man zum Beispiel so deklarieren:

```
int strlen(const char str[]);
```

Hier mag das überflüssig erscheinen, weil man auf den ersten Blick sieht, dass im Rumpf der Funktion nichts „Böses“ passiert. Überlegen Sie sich aber auch dieses:

- Bei vielen Funktionen, etwa den Funktionen der Standardbibliothek, kann man sich nur die Deklarationen in einer Headerdatei und ggf. eine zugehörige Beschreibung ansehen, während die Definition nicht zugänglich ist.
- Je länger eine Funktion ist, desto mühsamer wird die Prüfung, ob sie einen Parameter verändert oder nicht.

In beiden Fällen ist es angenehm, schon aus dem Prototyp ablesen zu können, ob die Funktion ein übergebenes Feld unverändert lässt.

Das Attribut `const` darf man übrigens auch für Variablen vergeben. Dann kann es auch für einfache Variablen sinnvoll sein.

7.8 Prototypen

Prototypen, d.h. Funktionsdeklarationen mit Angabe der Parameterliste, haben verschiedene Zwecke:

- Der Compiler kann prüfen, ob Argumente und Parameter einer Funktion übereinstimmen oder doch zumindest zueinander „passen“, ohne den für die Funktion auszuführenden Code zu kennen. Das ist häufig dann der Fall, wenn die Funktion aus einer Bibliothek stammt, deren Quelltext nicht veröffentlich wurde.
- Der Compiler kann, soweit erforderlich und möglich, die Typen der Argumente in die Typen der Parameter umwandeln. Die Umwandlung erfolgt, wie bereits angesprochen, nach denselben Regeln wie bei Wertzuweisungen.
- Der Compiler kann einen Funktionswert bei Bedarf in einen anderen Typ umwandeln.

Es sollte klar sein: Damit die Prüfungen und ggf. Umwandlungen ausgeführt werden können, muss der Prototyp einer Funktion in der Quelldatei vor ihrem ersten Aufruf stehen. Die Definition kann dann an beliebiger anderer Stelle folgen.

7.9 Rekursion

C erlaubt *Rekursion*. Man spricht von

- *direkter Rekursion*, wenn eine Funktion sich selbst aufruft, und von
- *indirekter Rekursion*, wenn sich zwei oder mehr Funktionen wechselseitig aufrufen.

Rekursion wird in der Mathematik häufig in Definitionen verwendet. Das „klassische“ Beispiel ist die Berechnung der Fakultät $n!$ einer nichtnegativen ganzen Zahl n:

$$n! := \begin{cases} n \cdot (n-1)! & \text{für } n > 0 \\ 1 & \text{für } n = 0 \end{cases}$$

Mit rekursiven Funktionen lassen sich solche Formeln direkt umsetzen:

```
long int fakultaet(int n) {
  if (n < 0)        /* Fehler */
    return 0;

  if (n == 0)       /* Abbruchbedingung */
    return 1;

  return n * fakultaet(n - 1);
}
```

Der Programmierer muss dafür sorgen, dass die Rekursion (irgendwann) „abbricht“. Im Beispiel sorgen die Abbruchbedingung `n > 0` und der rekursive Aufruf mit dem um eins verringerten Argument gemeinsam dafür.

Ob man rekursiv formulierte Berechnungen auch rekursiv realisieren sollte, steht auf einem anderen Blatt. Rekursion ist in der Regel ziemlich aufwendig: Die Parameter müssen kopiert werden („call by value“), lokale Variablen müssen bereitgestellt, der Unterprogrammsprung muss ausgeführt werden. Man sollte sich deshalb zunächst einmal überlegen, ob sich eine rekursiv formulierte Aufgabe nicht auch ohne weiteres *iterativ*, d.h. mit einer Schleife, lösen lässt.

Auch unter diesem Aspekt ist die Berechnung der Fakultät einer nichtnegativen ganzen Zahl das „klassische“ Beispiel. Man kann sie nämlich durch

$$n! = \prod_{i=1}^{n} i$$

definieren und dann wie folgt realisieren.

```
long int fakultaet (int n) {
  long int fak = 1;

  if (n < 0)        /* Fehler */
    return 0;

  while (n > 1)
    fak *= n--;

  return fak;
}
```

Ob eine rekursive oder eine nicht–rekursive Realisierung bestimmter Funktionen vorzuziehen ist, hängt sehr vom Einzelfall ab. Immer dann, wenn sich Rekursion, wie bei den Fakultäten, durch eine einfache Schleife oder ein ähnlich einfaches anderes Konstrukt vermeiden lässt, sollte man sie auch vermeiden. Andererseits gibt es Fälle, in denen Rekursion

nur mit erheblichen „Klimmzügen“ zu vermeiden wäre – und dann sollte man sie auch nutzen.

Die folgenden Beispiele sollten ausreichen, um ein grundlegendes Verständnis für rekursive Algorithmen zu bekommen. Eine ausführlichere Behandlung rekursiver Algorithmen und Techniken kann hier nicht erfolgen.

7.10 Beispiel: Türme von Hanoi

Nicht in jedem Falle gibt es zu einem rekursiv formulierten Problem so offensichtlich eine iterative Lösung wie bei den Fakultäten.

Ein solches Beispiel sind die „Türme von Hanoi“ (Abbildung 7.1): Man hat einen Turm von Scheiben, bei dem jede Scheibe kleiner ist als die, auf der sie liegt. Der Turm soll versetzt werden, wobei zusätzliche Regeln einzuhalten sind:

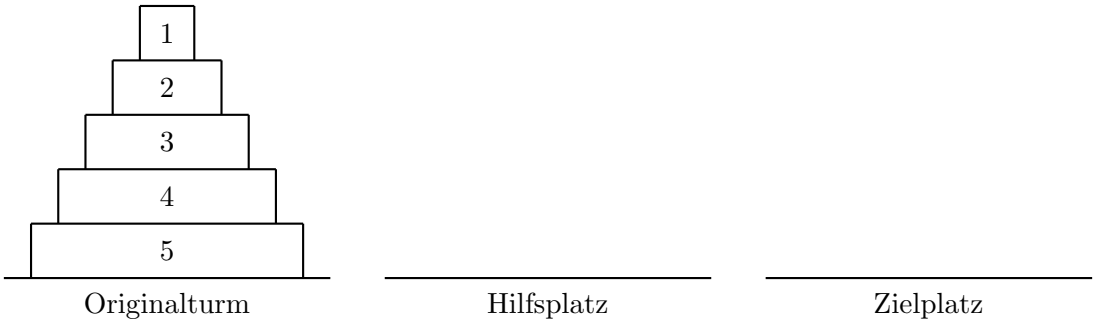

Abbildung 7.1: Türme von Hanoi

- Es darf immer nur die oberste Scheibe eines Turmes bewegt werden.
- Neben dem Ausgangsturm und dem Zielplatz steht ein dritter Platz zur Ablage von Scheiben zur Verfügung.
- Jede Scheibe darf nur auf den Boden oder auf eine größere Scheibe gelegt werden.

Nehmen wir einmal an, dass der Turm aus n Scheiben besteht. Die drei Plätze wollen wir mit „O“ („Originalplatz“), „H“ („Hilfsplatz“) und „Z“ („Zielplatz“) bezeichnen. Dann ist die rekursive Lösung der Aufgabe trivial:

1. Man legt die obersten $n-1$ Scheiben von „O“ nach „H“.
2. Die unterste Scheibe wird nach „Z“ gelegt.
3. Man legt die $n-1$ Scheiben von „H“ nach „Z“.

Damit hat man das Problem der n Scheiben auf zwei Probleme mit jeweils $n-1$ Scheiben zurückgeführt. Diese löst man nach demselben Verfahren, wobei nur die drei Plätze ihre Bedeutung wechseln:

- Im ersten Schritt ist „H“ der „Zielplatz“ für den Turm aus $n-1$ Scheiben und „Z“ der „Hilfsplatz“.
- Im dritten Schritt ist „H“ der „Originalplatz“ des Turms aus $n-1$ Scheiben und „O“ der „Hilfsplatz“.

Dass die Rekursion zum Erfolg führt, ist klar: Bei jedem Rekursionsschritt verringert sich die Höhe des zu verschiebenden Turmes um 1 – und für einen Turm der Höhe 1 besteht seine Verschiebung einfach in der Verschiebung seiner einen Scheibe.

Ein Programm, das diesen rekursiven Algorithmus rekursiv realisiert, kann so aussehen:

```
#include <stdio.h>

/**  Prototyp  ****************************************/
void hanoi(int hoehe, char von, char nach, char ueber);

/*= Rahmenprogramm ========================================*/
int main(void) {
  int n;

  printf("Wie hoch ist der Turm? ");
  scanf("%d", &n);
  hanoi(n, 'O', 'Z', 'H');

  return 0;
}

/*= rekursives Ausfuehren der Zuege =======================*/
void hanoi(int hoehe, char von, char nach, char ueber) {
  if (hoehe > 1)
    hanoi(hoehe - 1, von, ueber, nach);

  printf("Bewege Scheibe '%d' nach '%c'!\n", hoehe, nach);

  if (hoehe > 1)
    hanoi(hoehe - 1, ueber, nach, von);
}
```

Quelltext 7.2: Türme von Hanoi rekursiv

An dieser Stelle sei angemerkt, dass es auch für die Lösung dieses Problems einen iterativen Algorithmus gibt.

7.11 Beispiel: Quicksort

Die beiden Beispiele für Rekursion, die wir bislang betrachtet haben, ließen sich auch iterativ lösen. Bei den Fakultäten war das sehr einfach, bei den Türmen von Hanoi erforderte es schon längere Überlegungen.

Es gibt jedoch auch Aufgaben, die sich prinzipiell *nur* rekursiv lösen lassen. Hierzu zählt „Quicksort“, ein Verfahren zum Sortieren von Vektoren.

Quicksort funktioniert so: Wir wählen zunächst ein beliebiges Element des Vektors, z.B. das an seinem Anfang stehende. Jetzt vertauschen wir Elemente des Vektors so, dass

- das ausgewählte Element an seinen endgültigen Platz kommt, und dass
- links von ihm nur nicht größere und rechts von ihm nur nicht kleinere stehen.

Damit sind wir in einer ähnlichen Situation wie bei den „Türmen von Hanoi“: Anstelle *eines* Vektors mit z.B. n Komponenten haben wir jetzt zwar *zwei* Vektoren zu sortieren – die Gesamtzahl der Komponenten dieser beiden Vektoren ist aber nur $n - 1$. Damit ist jeder der beiden Teilvektoren (echt) kürzer als der Ausgangsvektor und es ist klar, dass der Algorithmus terminiert: Irgendwann im Zuge der Rekursion wird *jeder* Teilvektor auf die Länge 0 oder 1 reduziert – und ein leerer Vektor ist ebenso sortiert wie ein Vektor mit nur einer Komponente.

Damit erweist sich die rekursive Routine selbst wieder als sehr simpel:

```
void quick(int z[], int von, int bis) {
  int p;
  if (von < bis) {
    p = platz(z, von, bis);
    quick(z, von, p - 1);
    quick(z, p + 1, bis);
  }
}
```

Nicht so ohne weiteres hinschreiben kann man die Funktion `platz`. Wir müssen uns zunächst überlegen, wie sie arbeiten soll, und tun das anhand eines Beispiels. In der Zahlenfolge

```
2    8    7    5    3    6    4    1
```

soll das erste Element (2) an seinen endgültigen Platz gebracht werden; dabei sollen die übrigen Zahlen so vertauscht werden, dass links von der 2 nur nicht größere Zahlen und rechts davon nur nicht kleinere Zahlen stehen. Dazu speichern wir die 2 zunächst in einer Hilfsvariablen:

```
2
-    8    7    5    3    6    4    1
```

Dadurch wird die erste Komponente des Vektors frei. Jetzt suchen wir von ganz rechts die erste Zahl, die kleiner als 2 ist, und speichern sie auf den freien Platz um:

```
2
1    8    7    5    3    6    4    -
```

Dadurch wird deren alter Platz frei. Jetzt suchen wir, rechts neben der umgespeicherten Zahl beginnend, aufwärts nach der ersten Zahl, die größer als 2 ist, und speichern sie auf den freien Platz um:

```
2
1    -    7    5    3    6    4    8
```

Dadurch wird deren alter Platz frei. Jetzt suchen wir wieder, links neben der umgespeicherten Zahl beginnend, abwärts nach der ersten Zahl, die kleiner als 2 ist, und speichern sie auf den freien Platz um. Und diese abwechselnde Suche von rechts und links wiederholen wir, bis sich die Indizes überschneiden. Dann haben wir den gesuchten Platz gefunden: Wir können die herausgenommene Zahl an diesem Platz wieder in den Vektor eintragen und den Index des Platzes als Funktionswert zurückliefern.

Die Realisierung kann so aussehen:

```
int platz (int z[], int von, int bis) {
  int h;

  h = z[von];
  while (von < bis) {
    while (von < bis && z[bis] >= h)
      bis--;

    if (von < bis) {
      z[von] = z[bis];
      von++;
    }
```

```
    while (von < bis && z[von] <= h)
      von++;

    if (von < bis) {
      z[bis] = z[von];
      bis--;
    }
  }
  z[von] = h;
  return von;
}
```

Was den Quicksort von den Fakultäten und den „Türmen von Hanoi“ unterscheidet, ist die Tatsache, dass Daten gespeichert werden müssen: Haben wir den Ausgangsvektor unterteilt, können wir nur einen der beiden Teilvektoren sofort weiter verarbeiten; Anfangs– und Endindex des anderen Teilvektors müssen wir speichern. Für den Teilvektor, den wir sofort weiterverarbeiten, gilt dasselbe: Erneut können wir nur einen seiner Teile sofort weiterverarbeiten; Anfangs– und Endindex des anderen Teils müssen wir speichern.

Formulieren wir den Algorithmus rekursiv, so sorgt der Compiler für die Speicherung der Anfangs– und Endindizes der noch zu sortierenden Teilvektoren. Versuchen wir, den Algorithmus iterativ zu lösen, müssen wir selbst für die Speicherung der Anfangs– und Endindizes der noch zu sortierenden Teilvektoren sorgen. Eine scheinbar iterative Realisierung des Quicksort ist letztlich also nur Simulation von Rekursion – und dafür spricht wenig.

Quicksort ist für Vektoren ein sehr schnelles Sortierverfahren – sofern die Vektoren nicht schon weitgehend vorsortiert sind. In der hier vorgestellten Variante wird Quicksort für (auf– oder absteigend) sortierte Vektoren allerdings zum extremen „Slowsort“, weil in jedem Schritt die Ausgangsfolge der Länge m in eine leere Teilfolge und eine Teilfolge mit $m-1$ Komponenten zerlegt wird. Am schnellsten ist Quicksort dann, wenn in jedem Schritt die Ausgangsfolge in zwei gleich lange Teilfolgen zerlegt wird. Dem kann man Rechnung tragen, indem man nicht jeweils das erste Element der Teilfolge an seinen endgültigen Platz bringt. Denkbar wäre, stets das mittlere Element zu nehmen, oder auch von erstem, mittlerem und letztem Element das mit dem in der Mitte liegenden Wert.

Auch bei der rekursiven Funktion kann man noch mancherlei „optimieren“:

- Von den beiden rekursiven Aufrufen ist der zweite überflüssig und kann ohne weiteres durch eine Schleife ersetzt werden:

  ```
  void quick(int z[], int von, int bis) {
    int p;
    while (von < bis) {
      p = platz(z, von, bis);
      quick(z, von, p - 1);
      von = p + 1;
    }
  }
  ```

  Der erste rekursive Aufruf lässt sich dagegen nicht ohne weiteres ersetzen, weil die Rekursion gerade dafür sorgt, dass die Werte von `p` und `bis` (automatisch) gesichert werden.

- Die maximale Rekursionstiefe kann im „worst case“ (in jedem Schritt Abspalten nur eines Elements) bei n zu sortierenden Zahlen $n-1$ erreichen. Das kann man verhindern, indem man jeweils zunächst die kürzere Teilfolge durch Rekursion, danach die

längere Teilfolge durch Iteration sortiert:

```
void quick(int z[], int von, int bis) {
  int p;
  while (von < bis) {
    p = platz(z, von, bis);
    if (p - von < bis - p) {
      quick(z, von, p - 1);
      von = p + 1;
    }
    else {
      quick(z, p + 1, bis);
      bis = p - 1;
    }
  }
}
```

Jetzt kann die Rekursionstiefe bei zunächst n Zahlen nur noch maximal $\log_2 n$ erreichen.

- Die Funktion lässt sich auch mit nur zwei Parametern formulieren, wenn man nicht mit den Anfangs– und Endindizes, sondern mit Zeigern auf die entsprechenden Komponenten arbeitet. Hierfür fehlen uns derzeit aber noch die Mittel.

Wir haben jetzt drei Beispiele für Rekursion betrachtet:

- Bei den Fakultäten haben wir gesehen, dass Rekursion wenig sinnvoll ist.
- Bei den „Türmen von Hanoi“ gibt es zwar eine nichtrekursive Lösung. Ob diese aber effizienter als die rekursive Lösung ist, müsste man erst genauer untersuchen.
- Beim Quicksort ist jede scheinbar iterative Lösung tatsächlich nur Simulation von Rekursion.

7.12 Beispiel: Potenzen

Die bisherige Implementierung der Potenzfunktion lässt sich noch weiter optimieren. Dazu verwendet man die Binärdarstellung des Exponenten.

$$x^{109_d} = x^{1101101_b} = x^{1\cdot 64} \cdot x^{1\cdot 32} \cdot \underbrace{x^{0\cdot 16}}_{=1} \cdot x^{1\cdot 8} \cdot x^{1\cdot 4} \cdot \underbrace{x^{0\cdot 2}}_{=1} \cdot x^{1\cdot 1}$$

Man berechnet also iterativ die Potenzen $x^{(2^k)}, k = 0, 1, \ldots$ und entscheidet in jedem Schritt, ob der Faktor am Ergebnis beteiligt ist. Das erreicht man mit dem Modulo–Operator. Anschließend wird der Exponent durch zwei geteilt, um im nächsten Schritt das nächste Bit ermitteln zu können – Division mit zwei entspricht dem Schieben der Bits um eine Stelle nach rechts.

```
double potenz(double basis, unsigned exponent) {
  double p = 1;

  while (exponent != 0) {
    if (exponent % 2)
      p *= basis;
    exponent /= 2;
    basis *= basis;
  }
  return p;
}
```

Diese iterative Lösung kommt mit maximal $\log_2$(exponent) Schritten aus. Das ist ein enormer Gewinn gegenüber linearem Aufwand.

Kapitel 8

Strukturierung von Programmen

8.1 Gültigkeitsbereiche von Namen

Wo in einem Programm dürfen Vereinbarungen stehen? An verschiedenen Stellen haben wir dazu bereits Details kennengelernt:

- Am Anfang jedes Blocks dürfen Variablen vereinbart werden.
- Funktionen müssen auf gleicher Ebene (wie die Funktion `main`) „global“ definiert werden.
- Benannte Konstanten (mit `#define`) wurden jeweils vor die erste Funktion (das Hauptprogramm) geschrieben.
- Prototypen wurden ebenfalls vor die erste Funktion geschrieben.

Es gibt aber durchaus weitere Möglichkeiten:

- Variablen kann man auch außerhalb von Funktionen vereinbaren.
- Prototypen können auch am Anfang eines Blocks angegeben werden.

Als Beispiel soll das Programm zur Invertierung eines Strings in modifizierter Form dienen:

```
#include <stdio.h>

#define LAENGE 20

/*= Rahmenprogramm ========================================*/
int main(void) {
  char string[LAENGE + 1];
  int anzahl;
  void invert(char str[]);

  printf("String eingeben (max. %d Zeichen):\n", LAENGE);
  scanf("%s", string);
  invert(string);
  printf("Der invertierte String ist '%s'\n", string);

  return 0;
}

/*= Funktionen ============================================*/

int i, j;
```

```
/*- Invertieren eines String ------------------------------*/
void invert(char str[]) {
  char c;
  int strlen(char str[]);

  for (i = 0, j = strlen (str) - 1; i < j; i++, j--)
    c = str[i], str[i] = str[j], str[j] = c;
}

/*- Bestimmung der Laenge eines String --------------------*/
int strlen(char str[]) {
  for (i = 0; str[i] != '\0'; i++)
    ;

  return i;
}
```

Quelltext 8.1: Invertieren eines Strings mit Funktionen

Drei wesentliche Änderungen gegenüber der ursprünglichen Lösung sind hervorzuheben:

- Der Prototyp der Funktion `strlen` wurde in die Funktion `invert` hineingezogen.
- Der Prototyp der Funktion `invert` wurde in die Funktion `main` hineingezogen.
- Die Vereinbarungen der `int`–Variablen aus beiden Funktionen wurde herausgezogen und zu einer einzigen Vereinbarung zusammengezogen.

Welche Konsequenzen hat das? Wir sehen uns das im Rahmen der allgemeinen Regeln an:

1. Namen, die innerhalb eines Blocks (also zum Beispiel im Rumpf einer Funktion) deklariert werden, sind nur innerhalb dieses Blocks bekannt; Namen, die außerhalb von Funktionen deklariert werden, sind von der Stelle ihrer Deklaration bis zum Ende der Quelldatei bekannt.

   Für unser Beispiel hat das zunächst die Konsequenz, dass die Funktion `strlen` nur noch innerhalb der Funktion `invert` bekannt ist. Das Hauptprogramm könnte sie, wenn es wollte, jetzt nicht mehr aufrufen, was in der ursprünlichen Formulierung ohne weiteres möglich gewesen wäre.

   Die Umsetzung des Prototyps von `invert` hat keine Folgen: Die Definition einer Funktion impliziert bei Bedarf deren Deklaration.

   Die weitere Konsequenz: Die `int`–Variablen `i` und `j` sind jetzt beiden Funktionen bekannt. Nun könnte man zwar sagen: Was macht es aus, dass die Funktion `strlen` eine Variable (`j`) kennt, die sie nicht braucht. Das ist auch richtig – der eigentliche Unterschied liegt im Bedeutungswandel des Namen `i`: In der ursprünglichen Formulierung hatten beide Funktionen eine Variable `i`. Und diese Variablen hatten außer dem Namen nichts gemeinsam – insbesondere bezeichneten sie verschiedene Speicherplätze. Jetzt teilen sich die beiden Funktionen eine Variable, die natürlich auch nur einen Speicherplatz besitzt – mit der Folge, dass das Programm so nicht funktioniert.

   Fazit: Bei Prototypen spricht viel dafür, sie gleich an den Anfang eines Programms zu schreiben, noch vor die erste Funktion, damit man sie bei Bedarf an beliebiger Stelle des Programms verwenden kann. Wenn man Funktionen „verstecken“ will, um ihre Verwendung auf ganz bestimmte Stellen zu beschränken, sollte man das Programm modularisieren – darauf werden wir gleich noch zurückkommen. Umgekehrt spricht

bei Variablen viel dafür, sie möglichst lokal zu deklarieren, in der Regel also innerhalb einer Funktion.

2. Die Namen der Parameter in Prototypen und die Namen der Parameter in einer Funktionsdefinition sind nur innerhalb der Funktion bekannt.

   Obwohl sowohl der Parameter von `invert` als auch der Parameter von `strlen` mit `str` bezeichnet wurde, ergibt sich daraus nicht zwangsläufig, dass beide dasselbe Objekt bezeichnen, auch wenn es im Beispiel zufällig so ist.

3. Namen können *verschattet* werden, d.h. ein Name kann vorübergehend eine andere Bedeutung bekommen.

   Genutzt haben wir das beim Prototyp von `strlen`: Der Name `str` bezeichnet innerhalb des Prototyps nicht den Parameter von `invert`, wie in den Zeilen davor und dahinter, sondern den Parameter von `strlen`.

   In gleicher Weise verschattet jede Deklaration eines Namens innerhalb eines Blocks gleichnamige Größen außerhalb dieses Blocks. Wir könnten zum Beispiel programmieren

   ```
   int i;

   for (i = 0; i < laenge; i++) {
     float i;
     /* ... */
   }
   ```

   Dass so etwas kaum zweckmäßig ist, dürfte auf der Hand liegen.

Die Regeln, die wir eben besprochen haben, gelten übrigens `nicht` für Konstanten, die mit `#define` benannt werden. Der Hintergrund ist einleuchtend: Der Präprozessor, der `define`–Direktiven verarbeitet, weiß nichts von C – er führt nur formal Zeichenersetzungen aus. Entsprechend ersetzt er von der `define`–Direktive ab jedes Vorkommen des Namens. Der Compiler muss dagegen die Struktur des Programms analysieren – und berücksichtigen.

8.2 Interne und externe Größen

Eine Größe heißt *intern*, wenn ihre Definition innerhalb einer Funktion steht, sonst *extern*.

Funktionen sind offenbar immer extern, da C die Schachtelung von Funktionen nicht erlaubt. Bei Variablen hat man, wie gesehen, die Wahl: Interne Variablen stehen nur innerhalb des Blocks zur Verfügung, der ihre Definition enthält, während externe Variablen allen Funktionen zur Verfügung stehen, die hinter der Variablendefinition definiert werden – es sei denn, die Funktionen verschatten sie.

In der Regel sollte man versuchen, nur mit internen Variablen auszukommen. Es gibt aber durchaus auch Fälle, in denen externe Variablen zweckmäßig sind.

Als Beispiel betrachten wir Funktionen zur Stapelverwaltung (Stack, lifo = *last in first out*). Für einen Stapel gibt es vier „Standardoperationen“ in zwei Paaren:

- Bevor man ein Element auf den Stapel legen kann (`push`), muss man prüfen, ob noch Platz auf dem Stapel ist (`full`).
- Bevor man das oberste Element vom Stapel holen kann (`pop`), muss man prüfen, ob überhaupt ein Element auf dem Stapel liegt (`empty`).

Wie die Funktionen den Stapel verwalten, braucht und sollte den Benutzer der Funktionen nicht interessieren. Wir werden den Stapel durch einen Vektor realisieren; denkbar wäre

aber auch eine verkettete Liste. Neben dem Vektor selbst (`stapel`) brauchen wir eine Indexvariable (`spitze`), die die erste freie Komponente des Vektors bezeichnet.

Da alle vier Funktionen auf Vektor und/oder Indexvariable operieren sollen, haben wir zwei Möglichkeiten:

- Wir können Vektor und/oder Indexvariable als Parameter vorsehen. Das hat zur Folge, dass der Benutzer beide deklarieren und immer wieder als Argumente übergeben muss. (Die maximale Höhe des Stapels sollte bzw. müsste dann ebenfalls als Parameter übergeben werden.)
- Wir können Vektor und Indexvariable als externe Variablen vereinbaren. Sie brauchen dann nicht mehr als Argumente übergeben zu werden. Und wir halten uns gleich noch eine Möglichkeit offen: Wir können die Funktionen jederzeit von der Vektorspeicherung auf eine verkettete Liste umstellen, ohne dass irgendwelche Änderungen an den Aufrufen vorzunehmen sind.

Für die konkrete Realisierung soll angenommen werden, dass Zeichen zu speichern sind. Ein Rahmenprogramm soll dem Benutzer die Möglichkeit geben, ein Zeichen auf den Stapel zu legen oder herunterzunehmen bzw. das Programm zu beenden.

```
#include <stdio.h>

/**  Prototypen  ****************************************/
int full(void);
int empty(void);
void push(char zeichen);
char pop(void);

char gelesen(void);

/*= Rahmenprogramm ========================================*/
int main(void) {
  do {
    printf("0 = Zeichen auf Stapel ablegen\n");
    printf("1 = Zeichen von Stapel nehmen\n");
    printf("2 = Programmende\n");
    switch (gelesen()) {
      case '0':
        if (full ())
          printf("Stapel ist voll!\n");
        else {
          printf("Zeichen eingeben: ");
          push(gelesen ());
        }
        break;
      case '1':
        if (empty())
          printf("Stapel ist leer!\n");
        else
          printf("Oberstes Zeichen ist: %c\n", pop());
        break;
      case '2':
        return 0;
        break;
      default:
        printf("Nur 0, 1 oder 2 ist erlaubt!\n");
    }
```

```
  } while (1);
}

/*- Lesen eines Zeichen  (Hilfsroutine) -------------------*/
char gelesen(void) {
  char c1, c2;
  c1 = c2 = getchar();
  while (c2 != '\n')
    c2 = getchar();
  return c1;
}

/*= Stapel-Funktionen mit Daten ============================*/

#define HOEHE 10

char stapel[HOEHE];
int spitze = 0;

int full(void) {
  return spitze >= HOEHE;
}

int empty(void) {
  return spitze <= 0;
}

void push(char zeichen) {
  stapel[spitze++] = zeichen;
}

char pop(void) {
  return stapel[--spitze];
}
```

Quelltext 8.2: Stapelverwaltung

Sind externe Variablen gelegentlich auch sehr nützlich, so sollte man sich doch von Fall zu Fall sehr genau überlegen, ob man sie verwendet oder nicht. Bei extensiver Verwendung externer Variablen geht sehr schnell der Überblick verloren, wann wo was verändert wird – entsprechend sind nachträgliche Änderungen an den Funktionen extrem gefährlich.

8.3 Das Modulkonzept

Das letzte Beispiel legt mit seiner geschilderten Intention nahe, dass man das nutzende Programm und die Funktionen zur Stapelverwaltung nicht nur optisch innerhalb einer Quelldatei voneinander trennt, sondern dass man sie in verschiedene Quelldateien schreibt. So, wie das Beispiel vorab beschrieben wurde, soll die Stapelverwaltung ja beliebigen Programmen zur Verfügung stehen; unser Hauptprogramm muss man auch eher als „Testrahmen“ bezeichnen – eine Stapelverwaltung als Selbstzweck wie im Beispiel ist natürlich ziemlich unsinnig.

Eine Zerlegung in zwei Quelldateien ist auch mit den uns bekannten Mitteln bereits möglich: Wenn wir die Funktionen zur Stapelverwaltung in eine separate Quelldatei schreiben, etwa `stapel.c`, können wir diese Datei mit

```
#include "stapel.c"
```

ohne weiteres in beliebige andere Dateien einfügen. (Die Gänsefüßchen bewirken, dass der Präprozessor an anderen Stellen im Dateibaum sucht, als wenn spitze Klammern verwendet werden.)

Dieses Vorgehen erweist sich bei der praktischen Arbeit an größeren Projekten schnell als unzureichend. Was man braucht, ist eine Möglichkeit, Teile eines Programms voneinander unabhängig zu übersetzen und erst mit dem Linker zusammenzufügen. Solche Teile, die unabhängig voneinander übersetzt werden können, werden *Module* genannt. C geht sogar noch einen Schritt weiter: Ein Modul, der nur Unterprogramme enthält, wird in zwei Quelldateien gespeichert:

- Eine Datei enthält nur die Deklarationen der Größen, die der Modul *exportiert*, also in der Regel Typdeklarationen und Prototypen von Funktionen, gelegentlich auch Konstanten und Variablen. Solch eine Datei wird als *Headerdatei* bezeichnet.
- Die andere Datei enthält die Definitionen der zu exportierenden Größen, also insbesondere die Definitionen der Prototypen aus der Headerdatei. Sie kann aber auch zusätzliche lokale Funktionen realisieren, die dann nur innerhalb des Moduls verwendet werden können. Auf die Headerdatei greift sie mit einer `include`–Direktive zu.

Üblicherweise erhalten Headerdatei und Implementation eines Moduls den selben Namen, bei der Headerdatei um `.h` und bei der Implementation um `.c` erweitert.

Will man auf einen separaten Modul zugreifen, muss man zwei Dinge tun:

- Dem Compiler muss man im *rufenden* Modul die Größen des *gerufenen* Moduls bekanntmachen. Das geschieht mit einer `include`–Direktive für die Headerdatei des gerufenen Moduls.
- Dem Linker müssen die Module, die er zusammenbinden soll, einzeln genannt werden.

Sehen wir uns zunächst die Zerlegung unseres Programmbeispiels an: Die Headerdatei zur Stapelverwaltung, die `stapel.h` heißen soll, kann so aussehen:

```
/** Prototypen ********************************************/
int full (void);
int empty (void);
void push (char zeichen);
char pop (void);
```

Quelltext 8.3: Stapelverwaltung (modularisiert I): `stapel.h`

Dazu gehört die Implementation `stapel.c`:

```
#include "stapel.h"

/*= Daten ==========================================*/
#define HOEHE 10

char stapel[HOEHE];
int spitze = 0;

/*= Funktionen =====================================*/
int full (void) {
  return spitze >= HOEHE;
}
```

```
int empty (void) {
  return spitze <= 0;
}

void push (char zeichen) {
  stapel[spitze++] = zeichen;
}

char pop (void) {
  return stapel[--spitze];
}
```

Quelltext 8.4: Stapelverwaltung (modularisiert I): `stapel.c`

Schließlich haben wir den *Hauptmodul*, d.h. den Modul, der das Hauptprogramm enthält. Sein Name sei `stapeltest.c`.

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

/**  Prototypen  ****************************************/
char gelesen( void);

/*= Rahmenprogramm ========================================*/
int main (void)
{
  do {
    printf ("0 = Zeichen auf Stapel ablegen\n"
            "1 = Zeichen von Stapel nehmen\n"
            "2 = Programmende\nEingabe: ");
    switch (gelesen ()) {
      case '0':
        if (full ())
          printf ("Stapel ist voll!\n");
        else {
          printf ("Zeichen eingeben: ");
          push (gelesen ());
        }
        break;
      case '1':
        if (empty ())
          printf ("Stapel ist leer!\n");
        else
          printf ("Oberstes Zeichen ist: %c\n", pop());
        break;
      case '2':
        return 0;
        break;
      default:
        printf ("Nur 0, 1 oder 2 ist erlaubt!\n");
    }
  } while (1);
}

/*- Lesen eines Zeichen  (Hilfsroutine) -------------------*/
char gelesen (void) {
```

```
  int c1, c2;
  c1 = c2 = getchar ();
  while (c2 != '\n')
    c2 = getchar ();
  return (char) c1;
}
```

Quelltext 8.5: Stapelverwaltung (modularisiert I): `stapeltest.c`

Da ein Hauptmodul nichts exportiert, gibt es zu ihm auch keine Headerdatei.

8.4 Separate Compilation, `make`

Wir müssen uns jetzt noch ansehen, wie man Programme, die aus mehreren Modulen bestehen, übersetzen und binden kann. Letzlich ist das ganz einfach. Für unser Beispielprogramm können wir etwa schreiben

```
gcc -Wall -ansi stapeltest.c stapel.c
```

wenn wir mit `a.out` als Namen für das ausführbare Programm zufrieden sind, bzw.

```
gcc -Wall -ansi -o stapeltest stapeltest.c stapel.c
```

wenn das ausführbare Programm den Namen `stapeltest` erhalten soll.

Konnten beide Module nacheinander fehlerfrei übersetzt werden, schließt sich der Lauf des Linkers unmittelbar an, so dass wir das ausführbare Programm erhalten. Stellt der Compiler dagegen einen Fehler fest, etwa im Modul `stapeltest.c`, so übersetzt er zwar auch noch `stapel.c` (oder versucht es zumindest); der Linker wird jedoch nicht mehr gestartet.

Der nächste Schritt ist nun natürlich, die Fehler in `stapeltest.c` zu beseitigen. Danach kann man dann neu übersetzen.

Für größere Projekte ist dieses Vorgehen offensichtlich ungeeignet: Wer mag schon bei jedem Compileraufruf zehn oder mehr Module hinschreiben, ganz davon abgesehen, dass es bei so vielen Modulen nicht besonders effizient ist, alle immer neu zu übersetzen. Günstiger ist es, die Objektdateien aufzubewahren und nur dann neu zu erzeugen, wenn sie nicht mehr auf dem aktuellen Stand sind. Dafür gibt es zwei Gründe:

- Offensichtlich muss ein Modul neu übersetzt werden, wenn man an seiner Implementation etwas geändert hat.
- Bei etwas näherem Hinsehen ist es ebenso offensichtlich, dass ein Modul neu übersetzt werden muss oder zumindest der Sicherheit halber neu übersetzt werden sollte, wenn an einer der Headerdateien, die er benutzt, eine Änderung vorgenommen wurde: Es kann ja zum Beispiel die Parameterliste einer Funktion verändert worden sein. Statt sich nun den Kopf darüber zu zerbrechen, wo die Funktion überall aufgerufen wird, lässt man den Compiler arbeiten. Er wird (wenn er dem Standard entspricht) jede Abweichung der Argumente eines Aufrufs von der (geänderten) Parameterliste „bemeckern“!

UNIX bietet für solche Fälle eine Hilfe an, nämlich das Programm `make`. Für eine detaillierte Besprechung fehlt leider die Zeit. Aber exemplarisch soll das Programm wegen seiner Bedeutung doch kurz vorgestellt werden:

Im ersten, nur einmal auszuführenden Schritt, erstellt man mit einem Texteditor eine *Beschreibungsdatei*, d.h. man schreibt auf, was zu tun ist, um ein Programm auf den neuesten Stand zu bringen. Für unser Beispiel könnte das so aussehen:

```
# GCC Complier-Optionen
GCCFLAGS = -ansi -pedantic -Wall

# Abhaengigkeiten und Erzeugungskommandos
stapeltest : stapeltest.o stapel.o
        gcc -o stapeltest stapeltest.o stapel.o

stapeltest.o : stapeltest.c stapel.h
        gcc $(GCCFLAGS) -c stapeltest.c

stapel.o : stapel.c stapel.h
        gcc $(GCCFLAGS) -c stapel.c
```

Quelltext 8.6: `makefile` für Stapelverwaltung

.

Gibt man dieser Datei den Namen `makefile`, so kann man ihre Ausführung durch das Kommando `make` starten.

Die Zeilen der Beschreibungsdatei sind so zu interpretieren:

- Die erste Zeile ist eine Kommantarzeile.
- Die Zeile

  ```
  stapeltest : stapeltest.o stapel.o
  ```

  bedeutet: Wenn die Datei `stapeltest` älter als eine der beiden nachfolgend angegebenen Objektdateien ist, werden das oder die nachfolgenden Kommandos ausgeführt, sonst übersprungen.
- Die nächste Zeile

  ```
  gcc -o stapeltest stapeltest.o stapel.o
  ```

  ist ein solches Kommando: Die Objektdateien werden gebunden. Solche Kommandozeilen müssen mit einem (horizontalen) Tabulator beginnen, während die Beschreibungen der Abhängigkeiten jeweils am Zeilenanfang beginnen müssen.
- Bevor `make` das Alter von `stapeltest`, `stapeltest.o` und `stapel.o` vergleicht, prüft es zunächst, ob weiter unten Regeln angegeben sind, wie ggf. `stapel.o` und `stapeltest.o` auf den neuesten Stand zu bringen sind. Diese Regeln folgen jetzt also, mit der gleichen Syntax wie die „Hauptregel“ am Anfang. Anzumerken ist nur noch: Die Option `-c` bewirkt, dass nur eine Objektdatei erzeugt, nicht jedoch der Linker aufgerufen wird. Das Erstellen des Programms wird hier also tatsächlich in die Teilaufgaben übersetzen und binden zerlegt.

Die Option `-Wall` im ersten `gcc`–Aufruf wurde übrigens nicht vergessen: Er bewirkt ja keine Übersetzung von Quellcode, sondern nur noch das Binden der Objektdateien.

8.5 Lokale und globale Größen

Das Beispiel zur modularisierten Lösung der Stapelverwaltung ist noch nicht ganz fertig. Es tut nämlich noch nicht wirklich das, was beschrieben wurde: Die Variablen `stapel` und `spitze` sind *noch nicht* im Modul „versteckt“!

Der Hintergrund ist, dass bei der Beschreibung der Gültigkeitsbereiche von Namen das Modulkonzept noch völlig außer Acht gelassen wurde. Jede externe Größe eines Moduls

ist, wenn man nicht ausdrücklich anderes bestimmt, gleichzeitig eine *globale* Größe, d.h. man kann auf sie auch aus anderen Modulen heraus zugreifen – ob sie in der zugehörigen Headerdatei deklariert wird oder nicht, ist dafür gleichgültig. Interne Variablen sind dagegen gleichzeitig stets *lokale* Größen, d.h. man kann auf sie nur innerhalb der Funktion zugreifen, in der ihre Vereinbarung steht. Ebenso sind benannte Konstanten, die in einer Implementation in einer `define`–Direktive definiert werden, stets lokale Größen. Dass eine benannte Konstante, die in einer Headerdatei in einer `define`–Direktive definiert wird, allen Modulen zur Verfügung steht, die die Headerdatei verwenden, sollte offensichtlich sein.

Es gibt jedoch eine einfache Möglichkeit, für externe Größen zu verhindern, dass sie gleichzeitig auch globale Größen sind: Man fügt der Definition der Größe das Speicherklassen–Attribut `static` hinzu. Es ist üblich, wenn auch nicht notwendig, dieses Attribut an den Anfang der Definition setzen.

Entsprechend sollte unsere Datei `stapel.c` so beginnen, damit die Variablen `stapel` und `spitze` im Modul „versteckt“ werden:

```
#include "stapel.h"

/*= Daten ================================================*/
#define HOEHE 10

static char stapel[HOEHE];         /* Die Daten sind jetzt  */
static spitze = 0;                 /*   im Modul versteckt  */

...
```

Quelltext 8.7: Stapelverwaltung (modularisiert II): `stapel.c`

In diesem Beispiel sind es zwei Variablen, denen wir das Attribut `static` gegeben haben. Ebenso kann man aber auch (Hilfs–)Funktionen damit versehen, wenn diese Funktionen nur innerhalb des Moduls aufgerufen werden können bzw. sollen. Ein Beispiel ist die Funktion `gelesen` im (Test–)Hauptmodul der Stapelverwaltung. Definieren wir sie als `static`, so ist sie im Modul „versteckt“; verzichten wir dagegen auf das Attribut `static`, so wird der Name `gelesen` an den Linker weitergegeben – und der meckert, falls in irgendeinem anderen der einzubindenden Module auch eine globale Variable oder Funktion mit dem Namen `gelesen` definiert ist.

Fazit: Ebenso, wie man Variablen möglichst als interne Größen einer Funktion vereinbaren sollte, sollte man darauf achten, dass in jedem Modul alle Größen als lokal vereinbart werden, die nur innerhalb des Moduls (sinnvoll) verwendet werden können.

Man hat letztlich zwei voneineinander unabhängige Ebenen des „Versteckens“, nämlich einmal die Ebene des Übersetzens und einmal die Ebene des Bindens.

8.6 Deklarationen und Definitionen

Bei den Funktionen haben wir bereits gesehen, dass man zwischen Deklaration und Definition zu unterscheiden hat. Mit dem Modulkonzept haben wir jetzt auch gesehen, warum diese Unterscheidung erforderlich ist: Die Deklaration einer Funktion erlaubt den Zugriff auf sie, d.h. sie erlaubt den Aufruf der Funktion und damit die Ausführung ihrer Operationen; sie besagt aber nichts darüber, welche Operationen ihr Aufruf bewirkt. Die Definition einer Funktion legt ihre Operationen im Fall des Aufrufes fest, bewirkt aber nicht ihre Ausführung.

Es ist offensichtlich, dass eine Funktion innerhalb genau eines Moduls eines Gesamtprogramms *definiert* werden darf und muss, während sie in beliebig vielen Modulen, ggf. auch keinem, *deklariert* werden darf.

Wir haben bereits gesehen, dass auch Variablen global oder lokal sein können. Entsprechend muss es auch für Variablen die Unterscheidung zwischen Deklaration und Definition geben. Hierzu dient das Speicherklassen–Attribut `extern`: Eine Vereinbarung, die es enthält, ist eine Deklaration, und eine Vereinbarung, die es nicht enthält, eine Definition.

Sehen wir uns ein etwas konstruiertes Beispiel an – es dient nur der Demonstration des Konzepts. Angenommen man müsste – aus welchen Gründen auch immer – darauf verzichten, eine seperate Funktion `empty` für die Stapelverwaltung zu definieren. Dann könnte man wieder auf den ursprünglichen Einsatz der Variablen `spitze` zurückgreifen: Der Ausdruck `!spitze` leistet ja im Grunde dasselbe wie der Aufruf der Funktion `empty`.

```
/** Prototypen ********************************************/
int full (void);
void push (char zeichen);
char pop (void);
extern int spitze;          /* auf spitze kann jetzt aus   */
                            /*   beliebigen Modulen        */
                            /*   zugegriffen werden        */
```

Quelltext 8.8: Stapelverwaltung (modularisiert III): `stapel.h`

Entsprechend müssen in der Implementation `stapel.c` das Attribut `static` für die Variable `spitze` und die Definition der Funktion `empty` gelöscht werden:

```
#include "stapel.h"

/*= Daten ==========================================================*/
#define HOEHE 10

static char stapel[HOEHE];
int spitze = 0;

/*= Funktionen =====================================================*/
int full (void) {
  return spitze >= HOEHE;
}

void push (char zeichen) {
  stapel[spitze++] = zeichen;
}

char pop (void) {
  return stapel[--spitze];
}
```

Quelltext 8.9: Stapelverwaltung (modularisiert III): `stapel.c`

Schließlich muss im Rahmenprogramm noch der Aufruf von `empty` durch den Ausdruck `!spitze` ersetzt werden.

Vier Anmerkungen noch dazu:

- Noch einmal: Dies dient nur der Demonstration des Konzepts extern definierter Größen. Tatsächlich beschreibt `spitze` einen inneren Zustand des Stapels. Solche

inneren Zustände sollte man „privat“ halten und Zugriffe darauf nur durch spezielle Funktionen ermöglichen, wie z.B. Lesezugriff durch `empty()`. Durch obige Realisierung wird aber sogar Schreibzugriff von außen ermöglicht – ein fataler Entwurfsfehler! Es fehlen leider die Zeit und der Platz, um an dieser Stelle ein nichttriviales, sinnvolles Beispiel für extern definierte Variablen zu erläutern.

- Unsere Variablenvereinbarungen bislang waren streng genommen Definitionen, wie wir jetzt gesehen haben; auf Deklarationen haben wir bislang ganz verzichtet. Das ist ohne weiteres möglich, auch für Funktionen, weil jede Definition bei Bedarf eine Deklaration impliziert.

  Um diese Implikation für Funktionen zu nutzen, müsste man die Reihenfolge ihrer Definitionen ggf. umdrehen, d.h. jede rufende Funktion müsste *hinter* den Funktionen definiert werden, die sie ruft. Das ist in C jedoch nicht üblich und auch nicht immer möglich, wenn sich z.B. Funktionen gegenseitig aufrufen.
- Es sollte selbstverständlich sein, dass innerhalb eines Gesamtprogramms für eine Variable *höchstens* an einer Stelle ein Anfangswert angegeben werden darf. Die Restiktion, dass ein Anfangswert nur in einer Definition zugewiesen werden darf, ist damit naheliegend.
- Wir haben den Begriff *extern* jetzt mit zwei Bedeutungen kennengelernt, bei denen sich allenfalls mit Mühe Gemeinsamkeiten entdecken lassen. Man muss also sauber zwischen *externen* Variablen und Variablen mit dem Speicherklassen–Attribut `extern` unterscheiden.

Wir haben gesehen: Variablen kann man sowohl außerhalb als auch innerhalb von Funktionen deklarieren! Einerseits besagt die Deklaration einer Variablen (gekennzeichnet durch das Attribut `extern`) auf jeden Fall, dass die Definition der Variablen irgendwo anders steht. Andererseits hat das Attribut `extern` keinen Einfluss auf den Gültigkeitsbereich des Namens:

- Eine interne Variable mit dem Attribut `extern` steht nur in dem Block zur Verfügung, der ihre Deklaration enthält; wenn in mehreren Blöcken derselbe Name mit dem Attribut `extern` deklariert ist, stellt der Linker die Verbindung her.
- Eine externe Variable mit dem Attribut `extern` steht von der Stelle ihrer Deklaration an bis zum Ende des Moduls zur Verfügung.

8.7 Statische und automatische Variablen

Eine ähnliche, wenn auch nicht ganz so gravierende Begriffsüberschneidung gibt es bei *statisch*/`static`.

Eine *statische* Variable steht prinzipiell während der gesamten Dauer der Ausführung eines Programms zur Verfügung, auch wenn man zeitweilig nicht auf sie zugreifen kann, weil sie in einigen Modulen nicht deklariert oder in einigen Funktionen verschattet ist. Statisch sind alle externen Variablen – gleichviel, ob sie das Attribut `static` tragen oder nicht.

Für statische Variablen wird der Speicherplatz bereits durch den Compiler bereitgestellt und zugeordnet. Der Compiler löscht die Speicherbereiche dieser Variablen auch mit binären Nullen, wenn der Programmierer nicht in der Definition explizit Anfangswerte vorgibt. Dass ein Anfangswert neben einer Konstante auch ein konstanter Ausdruck sein darf, sollte klar sein: Ausdrücke, die der Compiler auswerten kann, sind Konstanten gleichgestellt, wie wir bereits gesehen haben.

Das Pendent zu den statischen Variablen sind die *automatischen* Variablen. Sie sind interne Variablen, d.h. Variablen, die innerhalb eines Blocks (zum Beispiel im Rumpf einer

Funktion) definiert werden. Diese Variablen besitzen nur einen Speicherplatz, während der Block ausgeführt wird, in dem sie definiert sind. Oder anders formuliert: Einer automatischen Variablen wird in dem Moment ein Speicherplatz zugeordnet, in dem der Block betreten wird, der ihre Definition enthält; beim Verlassen des Blocks wird die Zuordnung wieder gelöst. Wird der Block erneut betreten, erfolgt eine neue Zuordnung – allerdings unter Umständen zu einem anderen Speicherplatz.

Da der Standard außerdem nicht vorschreibt, dass automatische Variablen bei der Speicherplatzzuordnung einen wohldefinierten Anfangswert erhalten, *muss* man ihnen erst einen Wert zuweisen, bevor man auf ihren Wert zugreifen kann. Diese Zuweisung kann in der Form einer Anfangswertzuweisung erfolgen oder auch durch eine explizite Wertzuweisung; beides löst der Compiler letztlich ohnehin in gleicher Weise auf. Ob wir

```
void beispiel(double a, double b) {
  double f = sin(a) * exp(b);
  ...
}
```

oder

```
void beispiel(double a, double b) {
  double f;
  f = sin (a) * exp (b);
  ...
}
```

schreiben, ist also gleichgültig. Das erklärt letztlich auch, warum bei der Anfangswertzuweisung für automatische Variablen *beliebige* Ausdrücke erlaubt sind und nicht nur *konstante* Ausdrücke: Da der Compiler den Speicherplatz (noch) nicht bereitstellt, kann er den Wert selbst noch nicht zuweisen, sondern nur eine Befehlsfolge erzeugen, die den Wert berechnet und speichert. Falls diese Befehlsfolge die Aufrufe von Funktionen enthält, kann (und muss) der Linker diese ergänzen. Falls auf Parameter der Funktion zugegriffen wird, ist das auch kein Problem: Die Befehlsfolge wird ja erst ausgeführt, wenn die Funktion aufgerufen wird – und dann sind den Parametern bereits die Argumente des Aufrufs zugeordnet. Übrigens: `sin` und `exp` sind Standardfunktionen, die in der Headerdatei `math.h` deklariert sind. Darauf werden wir im nächsten Abschnitt zurückkommen.

Bei automatischen Variablen darf man das Speicherklassen–Attribut `auto` angeben – aber wer wird das schon tun, da sie dieses Attribut ohnehin automatisch erhalten!

Interessanter ist eine andere Möglichkeit: Variablen, die in einem Block definiert werden, dürfen das Attribut `static` erhalten! Solche Variablen sind eine Mischung aus externen `static`–Variablen und internen Variablen:

- Wie auf interne Variablen kann man auf sie nur in dem Block zugreifen, der ihre Definition enthält. Das impliziert insbesondere, dass sie wie `static`–Variablen lokal sind, also nicht aus anderen Modulen heraus gelesen oder verändert werden können.
- Wie bei externen Variablen erfolgt die Zuordnung des Speicherplatzes einmalig durch den Compiler, was entsprechend auch wieder nur konstante Ausdrücke als Anfangswerte erlaubt.

Wir sehen uns dazu zwei Funktionen an, die bis auf das Attribut `static` für ihre interne Variable übereinstimmen:

```
int f1(void) {
  int z = 1;
  return z++;
```

```
}

int f2(void) {
  static int z = 1;
  return z++;
}
```

Rufen wir diese Funktionen durch

```
for (i = 1; i <= 3; i++)
  printf("Aufruf %d: %d - %d\n", i, f1(), f2());
```

auf, so erhalten wir folgende Ausgabe:

```
Aufruf 1: 1 - 1
Aufruf 2: 1 - 2
Aufruf 3: 1 - 3
```

Die Interpretation ist klar:

- Bei `f1` wird der Variablen `z` bei jedem Aufruf erneut ein Speicherplatz und diesem Speicherplatz der Anfangswert 1 zugeordnet, den `f1` auch als Funktionswert liefert. Die Erhöhung des Wertes von `z` als Nebeneffekt bleibt letzlich ohne Wirkung, weil die Zuordnung der Variablen zu dem Speicherplatz mit dem Rücksprung wieder gelöst wird.
- Bei `f2` wird der Variablen `z` bereits durch den Compiler ein Speicherplatz zugeordnet und mit dem Anfangswert 1 belegt. Bei jedem Aufruf liefert `f2` als Funktionswert den Wert, den `z` gerade besitzt – und erhöht als Nebeneffekt den Wert von `z`. Da die Zuordnung zwischen der Variablen `z` und ihrem Speicherplatz mit dem Rücksprung *nicht* gelöst wird, hat `z` beim nächsten Aufruf den veränderten Wert des vorhergehenden Aufrufs.

8.8 `register` und `volatile`

Drei Speicherklassen–Attribute haben wir inzwischen kennengelernt, nämlich `auto`, `static` und `extern`. Daneben gibt es zwei weitere Speicherklassen–Attribute, die allerdings nur selten benutzt werden:

- Das Speicherklassen–Attribut `register` zeigt dem Compiler, dass die Variable keine globale Bedeutung besitzt und nach Möglichkeit in einem Register des Prozessors gehalten werden sollte.

  Dieses Attribut soll die Laufzeit eines Programms verringern – ob es das wirklich tut, hängt vom jeweiligen Compiler ab:

  – Bei einem „guten“ (optimierenden) Compiler und sauberer Programmierung wird das Attribut kaum Wirkung haben, weil der Compiler nach Möglichkeit ohnehin nur mit den Registern arbeiten wird.

  – Bei einem „schlechten“ (nicht optimierenden) Compiler kann das Attribut tatsächlich zu einer Beschleunigung führen – es kann aber auch zu einer Verlangsamung führen, wenn man „zu viele“ Variablen mit ihm versieht.

  Da der Standard keine detaillierten Vorschriften enthält (und auch nicht enthalten kann), ist man im Einzelfall auf Ausprobieren angewiesen. Vorgeschrieben ist auf jeden Fall: Nur automatische Variablen und Parameter dürfen das Attribut `register` erhalten.

- Das Speicherklassen–Attribut `volatile` verbietet dem Compiler die Optmierung des Zugriffs auf die entsprechend vereinbarten Variablen. Das bedeutet, dass der Wert bei jedem Zugriff neu aus dem Speicher geladen werden muss und nicht z.B. in einem Register vorgehalten werden darf. Erforderlich ist so etwas zum Beispiel dann, wenn man einen Treiber für ein externes Gerät programmiert. Dann kann der Inhalt einer Variablen z.B. durch das Gerät verändert werden und bei einer Optimierung des Zugriffs würde diese Änderung nicht bemerkt. Das geht aber weit über einen Programmierkurs für Anfänger hinaus.

Kapitel 9

Übersicht über die Standardbibliothek

Zu jeder C–Implementierung gehört standardmäßig eine Sammlung vordefinierter Routinen, die Ein–/Ausgabe und andere Operationen ermöglichen. Diese Sammlung wird *Standardbibliothek* genannt.

9.1 Headerdateien

Die Nutzung der Standardbibliothek erleichtern die *Standard–Headerdateien*, die der Standard vorschreibt, wenn eine Standardbibliothek vorhanden ist, und auf die man mit `#include`–Direktiven zugreifen kann. Man könnte die entsprechenden Deklarationen natürlich auch explizit in seine Programme schreiben – aber das wäre ziemlich unsinnig.

Insgesamt schreibt der Standard 15 Headerdateien vor. Einige davon haben wir bereits kennengelernt:

`ctype.h`	Zeichenverarbeitung
`float.h`	Interne Datenformate (Gleitkommatypen)
`limits.h`	Interne Datenformate (ganzzahlige Typen)
`stdio.h`	Ein–/Ausgabe
`string.h`	Stringverarbeitung

Einige weitere Headerdateien werden in diesem Abschnitt näher behandelt, auch wenn für eine vollständige Besprechung die Zeit fehlt:

`assert.h`	Testhilfen
`errno.h`	Fehlernummern
`math.h`	mathematische Funktionen
`stddef.h`	elementare Typen
`stdlib.h`	diverse Hilfsroutinen
`time.h`	Termine und Zeiten

Der Vollständigkeit halber seien auch die übrigen Headerdateien erwähnt:

`locale.h`	länderspezifische Darstellungen (Zeiten, Geldbeträge, usw.)
`setjmp.h`	Sprünge zwischen Funktionen
`signal.h`	Behandlung von Signalen („Interrupts“)
`stdarg.h`	Funktionen mit variabler Argumentzahl (wie `printf`)

Manche Deklarationen sind in mehreren Headerdateien enthalten. Das erlaubt es, alle Headerdateien unabhängig voneinander zu nutzen, und insbesondere, die `include`'s für

die Headerdateien in beliebiger Reihenfolge anzugeben. Andererseits erfordert es, die Headerdateien so gegeneinander abzusichern, dass der Compiler keine Deklarationen doppelt findet. Dieses ist mit bedingter Compilation möglich; die im nächsten Kapitel behandelt wird.

Für das Verständnis des weiteren ist es hilfreich daran zu erinnern, dass die Headerdateien die genannten Funktionalitäten nur bekannt machen. D.h. sie enthalten z.B nur Prototypen von Funktionen. Dadurch wird es dem Compiler möglich, den korrekten Funktionsaufruf im Quelltext zu überprüfen. Die eigentlichen Implementationen können dagegen in Binärdateien gesammelt sein, die durch Vorkompilation oder anders erzeugt wurden.

9.2 Mathematische Funktionen

Die numerischen Funktionen, die man üblicherweise in Programmiersprachen zur Verfügung hat, bietet auch C. Die Prototypen sind in `math.h` enthalten.

Funktion	Beschreibung
`sin`	Sinus
`cos`	Cosinus
`tan`	Tangens
`asin`	Arcus Sinus
`acos`	Arcus Cosinus
`atan`	Arcus Tangens
`sinh`	Sinus hyperbolicus
`cosh`	Cosinus hyperbolicus
`tanh`	Tangens hyperbolicus
`exp`	Exponentialfunktion
`log`	natürlicher Logarithmus
`log10`	Logarithmus zur Basis 10
`sqrt`	Quadratwurzel
`ceil`	Aufrundung auf ganze Zahl (das Resultat ist ein `double`–Wert!)
`floor`	Abrundung auf ganze Zahl (das Resultat ist ein `double`–Wert!)
`fabs`	Absolutbetrag (Achtung: `abs` für `int`–Argumente und `labs` für `long`–Argumente sind in `stdlib.h` deklariert)

Tabelle 9.1: Mathematische Funktionen mit einem Argument

Viele der Funktionen haben die Deklaration

```
double name (double);
```

d.h. sie haben einen `double`–Parameter und liefern einen `double`–Funktionswert. Diese Funktionen sind in Tabelle 9.1 aufgelistet. Dazu ist noch wichtig zu erwähnen, dass die Winkelfunktionen alle im Bogenmaß (d.h. voller Kreisumfang entspricht 2π) rechnen. Von den übrigen Funktionen sind drei besonders erwähnenswert, die jeweils zwei `double`–Parameter besitzen und einen `double`–Funktionswert liefern:

`atan2` liefert bei einem Aufruf `atan2 (y, x)` den Arcus Tangens von y/x, wobei auch eines der beiden Argumente Null sein darf (aber nicht beide gleich-

zeitig). Der Wertebereich ist $[-\pi, \pi]$, unter Berücksichtigung der Vorzeichen beider Argumente, während der Wertebereich bei `atan` nur $[-\pi/2, \pi/2]$ ist.

`fmod` liefert bei einem Aufruf `fmod (x, y)` die Nachkommastellen des Quotienten $x/|y|$ ($y \neq 0$). Nützlich ist diese Funktion zum Beispiel, wenn man Winkel modulo 2π rechnen möchte.

`pow` liefert bei einem Aufruf `pow (x, y)` die Potenz x^y.

Der Standard spricht nur von *der* Standardbibliothek. Trotzdem können die Funktionen auf mehrere *Bibliotheksdateien* (d.h. Binärdateien die Implementationen von Funktionen enthalten) verteilt sein. Nicht immer durchsucht in solchen Fällen der Linker auch alle Bibliotheksdateien. Für diesen Fall und auch für die Möglichkeit der Erzeugung und des Testens eigener Bibliotheken gibt es Linker–Optionen, mit denen man die zu verwendenden Bibliotheksdateien angeben kann.

Speziell im Falle der Verwendung der mathematischen Bibliotheksfunktionen ist dies meist erforderlich und wird gern vergessen. Speziell im Falle des `gcc` benötigt man dazu die Option `-lm` im Aufruf des `gcc`, die *hinter* die Namen der Dateien geschrieben wird, etwa so:

```
gcc -Wall prog.c -lm
gcc -Wall -o prog prog.c -lm
```

Hintergrund: Bei einigen Rechnern beherrscht der Hauptprozessor selbst nur ganzzahlige Arithmetik. Für Gleitkommaarithmetik gibt es zwei Möglichkeiten: Man kann einen zusätzlichen Gleitkommaprozessor einbauen oder man muss die Gleitkommaarithmetik durch entsprechende Funktionen realisieren. Dass die Bibliotheken in beiden Fällen zwar dieselben Eingangspunkte haben, darüber hinaus aber sehr verschieden aussehen, sollte offensichtlich sein. Sieht man nun für die mathematischen Funktionen eine besondere Bibliothek vor, so braucht man nur diese in verschiedenen Varianten zu unterhalten, während die „Hauptbibliothek“ nur in einer Variante existiert.

9.3 Fehlerbehandlung

Die Headerdatei `errno.h` enthält zwar nur drei Deklarationen, bietet aber trotzdem interessante Möglichkeiten:

Die Variable `errno` wird von vielen Bibliotheksfunktionen genutzt, um eventuelle Fehler zu markieren. Besitzt `errno` den Wert Null, so bedeutet das, dass kein Fehler aufgetreten ist, während positive Werte Fehler anzeigen.

Allerdings: `errno` wird nur einmal automatisch auf Null zurückgesetzt, nämlich beim Start des Programms. Möchte man eine bestimmte Operation auf fehlerfreien Ablauf hin überwachen, so ist es zweckmäßig, `errno` vor der Operation selbst auf Null zurückzusetzen.

Welche Fehler durch welche Nummern gekennzeichnet werden, überlässt der Standard den Implementatoren. Er schreibt jedoch eine Funktion (in `string.h`) vor, mit der man Fehlernummern in Klarschrift umsetzen kann. Außerdem erlauben die beiden anderen Deklarationen in `errno.h` die direkte Behandlung der häufigsten Fehler, nämlich Fehler bei den mathematischen Funktionen:

- `EDOM` ist der Wert, den `errno` erhält, wenn ein Argument außerhalb des zulässigen Wertebereichs liegt. Der Funktionswert der mathematischen Funktion ist in diesem Falle undefiniert.
- `ERANGE` ist der Wert, den `errno` erhält, wenn der Funktionswert außerhalb des zulässigen Wertebereichs liegt. Der Funktionswert der mathematischen Funktion wird dann

auf den Wert `HUGE_VAL` gesetzt, mit dem korrekten Vorzeichen. `HUGE_VAL` ist in `math.h` deklariert.

Durch den Aufruf `sqrt (- 2.0)` würde etwa `errno` auf den Wert `EDOM` und durch den Aufruf `exp (2000.0)` auf den Wert `ERANGE` gesetzt. Beim ersten Aufruf wäre der Funktionswert undefiniert, beim zweiten `HUGE_VAL` (vergleichbar mit $\pm\infty$).

Damit `errno` in allem Modulen eines komplexen Programms zur Verfügung stehen kann, darf die Variable mit nur einem Speicherplatz verbunden sein. Darum ist die Variable in `errno.h` extern deklariert und steht in allen Modulen zur Verfügung, die `errno.h` durch `#include` einbinden.

Sogar nur eine Deklarationen enthält die Headerdatei `assert.h`, die zum Test von Programmen dient: Die Funktion

```
void assert(int bedingung);
```

bewirkt nichts, wenn *bedingung* den Wert *wahr* besitzt. Sollte der Wert *falsch* sein, wird das Programm mit einer Fehlermeldung abgebrochen.

Eine zweite Definition wird in der Headerdatei zwar verwendet, ist aber nicht in ihr enthalten: Wenn der Name `NDEBUG` *nicht* definiert ist, werden die Tests wie beschrieben ausgeführt; ist der Name dagegen definiert, *unterbleiben* die Tests. Das erlaubt, die Aufrufe von `assert` im Quellcode stehen zu lassen, obwohl sie nicht ausgeführt werden sollen – vielleicht braucht man sie nach Änderungen am Programm ja noch einmal. Man muss nur eine Definition für `NDEBUG` ins Programm einfügen. Namen werden mit der `#define`–Direktive definiert; auf Details geht das nächste Kapitel ein.

9.4 Elementare Typen

Die Typdeklarationen und Konstantendefinitionen, die `stddef.h` enthält, sind weitgehend auch in anderen Headerdateien enthalten. Auf einige dieser Deklarationen werden wir noch zurückkommen.

9.5 Diverse Hilfsroutinen

Wundert man sich bei mancher Headerdatei über die Zusammenstellung der Deklarationen, so enthält insbesondere `stdlib.h` ein ziemlich buntes Gemisch von Deklarationen:

- Es gibt Funktionen, die den Inhalt eines String als Zahl interpretieren – z.B. `atoi`.

  ```
  int atoi(const char *s);
  ```

  Daneben gibt es noch eine Reihe weiterer Umwandlungsfunktionen.

- Es gibt Funktionen zur dynamischen Bereitstellung von Speicher – und natürlich auch zur Freigabe solchen Speichers. Wegen ihrer besonderen Bedeutung ist ihnen ein extra Kapitel gewidmet.

- Zwei Funktionen (`abort` und `exit`) erlauben die Beendigung eines Programms an beliebiger Stelle, also ohne geschachtelte Funktionsaufrufe erst bis ins Hauptprogramm zurückverfolgen zu müssen:

  ```
  void abort(void);
  void exit(int status);
  ```

- Mit `getenv` kann man sich Informationen über die Betriebssystemumgebung beschaffen oder mit `system` die Ausführung von Betriebssystemkommandos verlangen.

- Die Funktion `qsort` sortiert einen Vektor; mit `bsearch` kann man in einem sortierten Vektor nach einer Komponente mit einem bestimmten Wert suchen.
- Es gibt Funktionen, die die Verarbeitung erweiterter Zeichensätze erlauben.

Zwei Gruppen von Funktionen aus `stdlib.h` fehlen in diesem sehr pauschalen Überblick. Sie sollen jetzt ein wenig ausführlicher behandelt werden:

Die Funktion

```
int rand(void);
```

realisiert einen *Pseudo–Zufallszahlen–Generator*: Sie liefert bei jedem Aufruf eine ganze Zahl aus dem Bereich $[0, \texttt{RAND\_MAX}]$, wobei $\texttt{RAND\_MAX} \geq 32767 (= 2^{15} - 1)$ gelten muss.

Von einem „Pseudo“–Zufallszahlen–Generator spricht man, weil die Zahlen, die `rand` liefert, nicht wirklich zufällig sind. Vielmehr gibt es eine bestimmte Formel, wie aus dem Startwert bzw. dem letzten gelieferten Wert die nächste zu liefernde Zahl zu berechnen ist.

Pseudo–Zufallszahlen sind zum Programmtest oft nützlich. Betrachten wir als Beispiel den Test einer Sortierfunktion: Man möchte zwar möglichst gut verteilte Zahlen haben, wofür der Generator sorgt – man kann aber auch nicht bei jedem Test andere Werte brauchen, weil eventuelle Fehler reproduzierbar sein müssen. Um verschiedene Zahlenfolgen erzeugen zu können, kann man den Startwert (standardmäßig 1) mit der Funktion `srand` angeben; ihr Prototyp ist

```
void srand(unsigned int start);
```

Steckt man hier etwa Sekunden und Minuten der aktuellen Uhrzeit hinein, so lassen sich die Zahlenfolgen ohne weiteres nicht mehr nachvollziehen. Zumindest werden die Zahlen so „zufällig“, dass sie für Spiele und ähnliche Programme vollständig ausreichen.

Genau genommen liefert `rand` „gleich verteilte“ Zahlen, d.h. die Zahlen sind gleichmäßig über den erlaubten Wertebereich verteilt. Braucht man für einen Test etwa Zahlen mit einer Gauß–Verteilung, so kann man `rand` *nicht* ohne weiteres verwenden.

Zur zweiten Gruppe gehören vier Funktionen zur ganzzahligen Arithmetik: Die Funktionen `abs` und `labs` haben als Argument einen `int`– bzw. `long`–Ausdruck und liefern als Funktionswert dessen Absolutbetrag; die Funktionen `div` und `ldiv` haben als Argumente zwei `int`– bzw. `long`–Ausdrück und liefern als Funktionswert sowohl den ganzzahligen Quotienten als auch den Rest – wie man erreichen kann, dass Funktionswerte aus mehreren Einzelwerten bestehen, werden wir noch kennenlernen. Das Stichwort ist „Strukturen“.

9.6 Termine und Zeiten

Die verwendete Prozessorzeit des aktuellen Prozesses liefert die Funktion

```
clock_t clock(void);
```

die in `time.h` deklariert ist. Dabei ist `clock_t` ein ganzzahliger Typ, der ebenfalls in `time.h` deklariert ist. Dies geschieht durch eine `typedef`–Deklaration – eine typische Anwendung von `typedef`.

Dieses ist erneut eine typische Vorgehensweise des Standard: Vorgeschrieben wird zwar ein ganzzahliger Typ, nicht jedoch ein konkreter ganzzahliger Typ. Das erlaubt den Implementatoren, zwischen `short`, `int` und `long` zu wählen, abhängig von der Hardware des Rechners, auf dem die Implementation laufen soll. Für den Programmierer wird die konkrete Wahl durch den speziellen Typ weitestgehend transparent, ohne dass sich Restriktionen

ergeben: Bei arithmetischen Operationen sorgt der Compiler für die entsprechenden Umwandlungen! Probleme gibt es allenfalls bei der Ausgabe mit `printf`, weil man nicht weiß, welchen Formatbeschreiber man zu verwenden hat; das lässt sich aber auch lösen, indem man den Wert hierfür zunächst in `long` umwandelt.

Die Maßeinheit von `clock` ist implementations–spezifisch. In Sekunden kann man den Funktionswert von `clock` umrechnen, indem man ihn durch die ebenfalls in `time.h` definierte Konstante `CLOCKS_PER_SEC` dividiert.

Ebenso lässt der Standard offen, zu welchem Zeitpunkt die Zeitmessung gestartet wurde. Es handelt sich dabei um Implementationsabhängiges Verhalten, das entsprechend in der jeweiligen Dokumentation der Implementation beschrieben wird. Mögliche Zeitpunkte für den Start der Zeitmessung sind der Zeitpunkt des Programmstarts oder der erste Aufruf von `clock` im Programm.

Die übrigen Funktionen aus `time.h`, etwa die Bestimmung von Datum und Uhrzeit, erfordern Kenntnisse, die wir derzeit noch nicht haben.

Kapitel 10

Der Präprozessor

10.1 Überblick

Den *Präprozessor*, den der Standard vorsieht, haben wir mit zwei seiner *Direktiven* bereits kennengelernt:

- Eine `include`–Direktive wird durch den Inhalt der in ihr genannten Datei ersetzt.
- Eine `define`–Direktive bewirkt nach unserem derzeitigen Kenntnisstand, dass der in ihr definierte Name im Rest des Moduls durch den ebenfalls in ihr definierten Ersatztext ersetzt wird.

Die grundsätzliche Arbeitsweise des Präpozessors kennen wir damit: Er überarbeitet den Text, indem er Einsetzungen, Ersetzungen – und unter Umständen auch Streichungen – vornimmt. Alles geschieht auf rein formaler Ebene, also ohne Rücksicht auf die logische Struktur der Funktionen, die der Modul enthält.

Man kann sich den Präprozessor als Programm vorstellen, das vor der eigentlichen Übersetzung läuft, oder auch als ersten Durchlauf des Compilers durch das Programm.

Was ein Standard–Präprozessor können muss, ist ebenso im Standard festgelegt wie die Form der Direktiven:

- Jede Direktive beginnt mit einem Nummernzeichen (`#`). Diese Nummernzeichen muss, von „white spaces“ abgesehen, das erste Zeichen in der Zeile sein.
- Dem Nummernzeichnen folgt der Name der Direktive. Ob weitere Einträge erforderlich sind oder nicht, hängt von der jeweiligen Direktive ab.
- Jede Direktive wird durch ein Zeilenende–Zeichen beendet. Allerdings: Reicht der Platz in einer Zeile nicht aus, so kann man Fortsetzungszeilen schreiben. Dazu muss man *direkt* vor das Zeilenende–Zeichen der fortzusetzenden Zeile einen Backslash (\) setzen.

Wir haben für Direktiven also die allgemeine Form

`#`*name* *[text]*

Die Direktiven, die es neben `include` und `define` gibt, dienen im wesentlichen zur *bedingten Compilation*. Darunter versteht man, dass Teile des Quellcodes in Abhängigkeit von bestimmten Bedingungen übersetzt werden oder auch nicht. Bedingte Compilation ist in zwei Fällen besonders nützlich:

- Sie erlaubt es, Headerdateien so zu formulieren, dass man in beliebiger Reihenfolge auf sie zugreifen kann.

- Varianten von Funktionen, die sich nur in Details unterscheiden, können in einer Quelldatei gespeichert werden. Die Entscheidung darüber, welche Variante übersetzt werden soll, braucht erst beim Aufruf des Compilers getroffen zu werden.

10.2 Die Direktive `#include`

Die `include`–Direktive kennen wir bereits vollständig. Ihre beiden Formen sind

```
#include <name>
#include "name"
```

Zur Erinnerung noch einmal der Unterschied zwischen den beiden Formen: Bei der zweiten Form sucht der Präprozessor die angegebene Datei zunächst im aktuellen Verzeichnis. Findet er sie dort nicht, sucht er in anderen Directories, die implementationsspezifisch festgelegt sein müssen. Die erste Form arbeitet ebenso, nur dass die Suche in der aktuellen Directory entfällt.

Faktisch heißt das:

- die spitzen Klammern werden bei Standard–Headerdatei verwendet,
- die Gänsefüßchen bei eigenen Dateien.

Der Name darf jeweils auch Pfadangaben enthalten.

10.3 Makros (`#define`)

Dass man mit der Direktive `#define` Kontanten benennen kann, haben wir bereits gelernt und genutzt. Tatsächlich ist diese Möglichkeit nur ein Teilaspekt der Direktive. Zur Erinnerung noch einmal die Form der Direktive:

```
#define name [ersatztext]
```

An dem, was wir über den Eintrag *name* bereits wissen, ändert sich zunächst nur die Nomenklatur: Namen, die in einer `define`–Direktive definiert werden, werden vielfach als *Makros* bezeichnet. Die Ersetzungen, die ein Makro bewirkt, bezeichnet man als *Expandierung des Makro*.

Als *ersatztext* sind, über unsere bisherigen Kenntnisse hinaus, beliebige Zeichenfolgen erlaubt. Diese Zeichenfolgen dürfen insbesondere auch bereits zuvor definierte Makros enthalten oder leer sein.

Zulässig sind so zum Beispiel

```
#define BREITE 10
#define HOEHE (BREITE + 5)
```

Makros können Parameter erhalten:

```
#define name(parameter) ersatztext
```

Wichtig ist, dass die öffnende Klammer dem Namen *unmittelbar* folgen muss – damit wird gekennzeichnet, dass die Klammer eine Parameterliste einleitet und nicht Bestandteil des Ersatztextes ist. Zum Beispiel können wir also

```
#define QUADRAT(x) x * x
```

definieren. An den Stellen, an denen der Makro expandiert werden soll, müssen Argumente angegeben werden, ähnlich wie bei einer Funktion. Die Folge ist praktisch eine *doppelte* Textersetzung: Der Makro einschließlich seiner Argumentliste wird durch den Ersatztext ersetzt; im Ersatztext werden die Parameter durch die Argumente ersetzt. So ergeben zum Beispiel die Aufrufe

```
x = QUADRAT(7.4);
y = QUADRAT(x + 1);
```

letztlich die Quellanweisungen

```
x = 7.4 * 7.4;
y = x + 1 * x + 1;
```

Das zweite Beispiel zeigt, dass der Makro `QUADRAT` *unvollständig* definiert ist: Für `y` resultiert ein Ausdruck, der offensichtlich *nicht* der Intention entspricht. Abhilfe schafft

```
#define QUADRAT(x) ((x) * (x))
```

Die äußeren Klammern sind nötig, da es Operatoren gibt, deren Priorität höher als die der Multiplikation ist.

Makros mit Parametern können wie Funktionen die Lesbarkeit eines Programms erhöhen und Schreibarbeit sparen. Man muss sich aber darüber klar sein, dass zwischen Makros und Funktionen ganz wesentliche Unterschiede bestehen, die letztlich alle aus der Tatsache resultieren, dass Makros expandiert werden, *bevor* die eigentliche Übersetzung beginnt und dass sie damit im ausführbaren Programm überhaupt nicht mehr in Erscheinung treten. Einige Details:

- Dass auch bei Makros die Anzahlen von Parametern und Argumenten übereinstimmen müssen, sollte klar sein. Die Parameter von Makros besitzen jedoch *keine* Typen, so dass die Argumente beliebige Typen besitzen können – Typen spielen ja erst später eine Rolle, wenn der Compiler übersetzt.
- Jeder Aufruf eines Makro wird durch seinen Ersatztext ersetzt. Das hat einerseits zur Folge, dass der Maschinencode, der aus dem Ersatztext resultiert, mehrfach im ausführbaren Programm steht, während der Maschinencode, der aus einer Funktion resultiert, nur einmal im Programm steht. Dadurch entfallen andererseits die Sprünge und die Parameterzuordnung, mit denen Funktionsaufrufe verbunden sind.

Aus der Tatsache, dass ein Makro–Aufruf durch Textersetzung aufgelöst wird, folgt auch, dass man vorsichtig sein muss, wenn man als Argumente Ausdrücke mit Nebeneffekten einsetzt. Aus dem Aufruf

```
i = QUADRAT(j++);
```

resultiert so

```
i = ((j++) * (j++));
```

Abgesehen davon, dass der doppelte Nebeneffekt in der Regel nicht beabsichtigt sein wird, ist er auch gefährlich. Mit den Gefahren von Nebeneffekten haben wir uns bereits in Abschnitt 4.11 befasst.

Man kann die Argumente von einem Makro auch als Zeichenkette verwenden. Dazu muss man ihnen bei der Verwendung `#` voranstellen. Ein nützliches Beispiel ist im Abschnitt 10.5 angegeben.

10.4 Bedingte Compilation (#if, #elif, #else)

Die Direktiven–Konstrukte, die bedingte Compilation erlauben, arbeiten mit der Logik der `if`–Anweisungen. Ihre Form ist allerdings durchaus anders:

`#if` *bedingung*
quellcode

```
[ #elif bedingung
  quellcode ]
...
[ #else
  quellcode ]
#endif
```

Die Bedingungen müssen konstante Ausdrücke sein und werden in der C–typischen Weise als *wahr* oder *falsch* interpretiert. Der Quellcode, der für die einzelnen Fälle vorgesehen ist, kann aus vollständigen Anweisungen bestehen, braucht es aber nicht. Selbstverständlich muss der Programmierer immer dafür sorgen, dass der Quellcode, der durch die Auswahl entsteht, vollständige Quellanweisungen ergibt.

Wir betrachten ein Beispiel: Wir wollen einen Modul testen und dazu Ausgabeanweisungen in den Quellcode einfügen. Das können wir etwa in dieser Form tun:

```
#define TESTEN 1
...
#if TESTEN
  printf ("Kontrollpunkt A erreicht\n")
#endif
```

Haben wir unsere Tests abgeschlossen, so brauchen wir die `define`–Direktive für `TESTEN` nur durch

```
#define TESTEN 0
```

zu ersetzen, während alle übrigen Testzusätze im Modul stehen bleiben können: Der Präprozessor entfernt sie, so dass sie in Zukunft nicht mit übersetzt werden. Der wesentliche Vorteil dieses Verfahrens zeigt sich, wenn wir Änderungen am Modul vornehmen und ihn danach erneut testen müssen: Durch erneute Änderung der `define`–Direktive können wir alle Testausgabe wieder aktivieren.

In einem Punkt ist das Beispiel noch ziemlich untypisch: Man wird dem Namen `TESTEN` keinen Wert geben, sondern nur

```
#define TESTEN
```

schreiben. Der Name wird dadurch zwar definiert, repräsentiert jedoch keinen Wert. Abfragen kann man das durch den Präprozessor–Operator

```
defined (name)
```

Er prüft nur, ob der Name *name* definiert ist oder nicht; welchen Wert der Name ggf. repräsentiert, wird *nicht* geprüft. Wohlgemerkt: Dieser Operator steht nur in Präprozessor–Direktiven zur Verfügung. Unser Beispiel könnte damit so aussehen:

```
#define TESTEN
...
#if defined (TESTEN)
  printf ("Kontrollpunkt A erreicht\n")
#endif
```

Dieses kann man mit einer weiteren Direktive noch kürzer und prägnanter schreiben: Die Direktive

```
#ifdef name
```

hat gerade dieselbe Wirkung wie

```
#if defined (name)
```

Bei `#ifdef` kann immer nur ein Name angegeben werden; bei `if` könnten wir zum Beispiel aber auch

```
#if defined (name1) || defined (name2)
```

schreiben. `ifdef` ist also tatsächlich nur eine Kurzform für Spezialfälle.

Es sollte jetzt auch schon klar sein, wie man Größen in mehreren Headerdateien definieren kann, ohne dass es zu Konflikten kommt, unabhängig von der Reihenfolge, in der auf die Headerdateien zugegriffen wird. Schreiben wir etwa

```
#if !defined (EOF)
  #define EOF (- 1)
#endif
```

so ist `EOF` beim ersten Auftreten dieser Direktivenfolge noch nicht definiert – die `define`–Direktive wird also wirksam. Findet der Präprozessor dieselbe Direktivenfolge erneut, ist `EOF` bereits definiert, so dass er die Wiederholung aus dem zu übersetzenden Quellcode eliminiert.

10.5 Weitere Möglichkeiten (#ifndef, #undef)

Neben den Direktiven, die wir kennengelernt haben, schreibt der Standard vier weitere Direktiven vor:

`ifndef` ist das Pendent zu `ifdef`: Geprüft wird, ob der angegebene Name *nicht* definiert ist.

`undef` streicht den angegebenen Namen aus der Liste der definierten Namen, löscht also praktisch den Makro.

`error` erlaubt die Ausgabe eigener Fehlermeldungen während der Compilation.

`line` erlaubt die Einflussnahme auf die Form der Meldungen des Compilers.

Daneben schreibt der Standard die Vordefinition von fünf Makros vor:

`__FILE__` Name der Quelldatei, die gerade übersetzt wird

`__LINE__` Nummer der Quellzeile, die gerade übersetzt wird

`__DATE__` Datum der Übersetzung

`__TIME__` Uhrzeit der Übersetzung

`__STDC__` *wahr* oder *falsch*, je nachdem, ob der Compiler ein Standardcompiler ist oder nicht

Diese Makros sind faktisch Schlüsselworten gleichgestellt und dürfen vom Programmierer nicht für eigene Zwecke verwendet werden.

Für die Fehlersuche in großen Projekten mit mehreren Quelldateien können diese Makros sehr nützlich sein. Zum Beispiel liefert

```
#define DEBUG
...
#ifdef DEBUG
#define REPORT printf("Zeile %d in %s erreicht\n", \
                      __LINE__, __FILE__)
#else
#define REPORT
#endif
```

eine einfache und ausbaufähige Hilfe bei der Überwachung des Programmablaufs. Manchmal will man sich bei der Ausführung eines Programms davon überzeugen, dass eine bestimmte Programmzeile auch erreicht wurde. An solchen kritischen Stellen fügt man einfach die Zeile `REPORT;` in den Quelltext ein und erhält dann eine aussagekräftige Kontrollausgabe, sofern `DEBUG` bei der Compilation definiert wurde. Ausbaubar ist das Makro z.B. so:

```
#define REPORTINT(x) printf("%s(%d) : Wert: %d\n", \
                            __FILE__, __LINE__, x)
```

Damit kann der aktuelle Wert von `int`–Variablen ausgegeben werden. Auf `double`–Variablen angewendet liefert das Makro natürlich Unsinn.

Schließlich kann man durch folgende Variante auch noch dafür sorgen, dass der Name der angesprochenen Variablen mit ausgegeben wird.

```
#define REPORTINT(x) printf("%s(%d) : " #x " == %d\n", \
                            __FILE__, __LINE__, x)
```

Wie schon erwähnt sorgt `#` dafür, dass das Argument in Anführungszeichen eingeschlossen wird. `REPORTINT(a);` wird also zu

```
printf("%s(%d) : " "a" " == %d\n", ..., ..., a);
```

expandiert. Durch die Zusammensetzung von Zeichenkettenkonstanten ergibt sich ein zusammenhängender Formatstring für `printf`. Maßnahmen wie solche raffinierten Testausgaben fass man untr dem Begriff „low-level debugging“ zusammen. Geschickt eingesetzt können sie Programmierer sehr bei der Fehlersuche unterstützen.

10.6 Makro–Definition im Compileraufruf

Wir haben gesehen, wie man Makros mit den Direktive `define` definieren bzw. mit der Direktive `undef` löschen kann. Beides kann man auch durch Optionen im Aufruf des Compilers erreichen. Der `gcc`–Compiler unterstützt dazu die Kommandozeilenparameter:

`-D`*name*
`-D`*name*`=`*text*
`-U`*name*

Im Zusammenhang mit bedingter Compilation wird man diese Möglichkeit häufig nutzen, weil sie es einem erspart, zwischen den verschiedenen Compilerläufen den Quellcode zu modifizieren. Man kann zum Beispiel direkt im Compileraufruf angeben, ob Testausgaben erzeugt werden soll oder nicht, ohne den entsprechenden Makro im Quellcode zu definieren oder zu löschen.

Kapitel 11

Felder

11.1 Rückblick

Wir haben uns verschiedentlich bereits mit *Feldern* beschäftigt. Was wir dabei gelernt haben, ist im Folgenden noch einmal kurz zusammengefasst:

- Felder können durch

  typ *name*`[`*anzahl*`];`

  vereinbart werden. Der Wert *anzahl* muss ein konstanter Ausdruck mit ganzzahligem, positivem Wert sein.

- Die *Komponenten* (*Elemente*) eines Feldes sind Variablen mit dem Typ des Feldes. Sie können mit

  name`[`*index*`]`

  angesprochen werden. Da die Numerierung bei 0 beginnt, ist der größte zulässige *Index* um 1 kleiner als die Anzahl der Komponenten. Die Indizes selbst können beliebige Ausdrücke mit einem ganzzahligen Typ sein.

- Der Programmierer ist selbst dafür verantwortlich, dass er nur zulässige Indizes verwendet.

- Vektoren mit dem Typ `char` werden als Strings bezeichnet, wenn sie in einer Komponente ein Null–Zeichen enthalten.

- Feldnamen bezeichnen selbst keine Variablen, sondern sind Zeigerkonstanten. Das hat Auswirkungen auf die Übergabe von Feldern an Funktionen.

11.2 Vereinbarung von Feldern

Ähnlich wie eindimensionale Felder (*Vektoren*) können auch Felder mit mehreren Dimensionen vereinbart werden: Dem Namen folgen dann, jeweils in eckigen Klammern eingeschlossen, die Längenangaben für die verschiedenen Dimensionen. So ergibt

```
int m[2][3];
```

ein zweidimensionales Feld (*Matrix*). In gleicher Weise hat man mehrere Indizes anzugeben, wenn man auf ein einzelnes Element zugreifen will; im Beispiel hat man etwa die 6 Elemente

```
m[0][0], m[0][1], m[0][2], m[1][0], m[1][1], m[1][2].
```

Sowohl die Vereinbarung eines mehrdimensionalen Feldes als auch die Zugriffe auf seine Komponenten kann man auch in anderer Weise interpretieren: Eine Matrix zum Beispiel kann man auch als Vektor betrachten, dessen Komponenten ihrerseits Vektoren sind. Bei einem dreidimensionalen Feld hat man sogar drei Möglichkeiten der Interpretation (Vektor von Matrizen, Matrix von Vektoren, Vektor von Vektoren von Vektoren). Entsprechende Zugriffe sind erlaubt: `m[0]` und `m[1]` bezeichnen so die beiden Vektoren der Länge 3, aus denen `m` besteht. `m[1][0]` bezeichnet in diesem Sinne die erste Komponente im zweiten Vektor von `m`.

11.3 Anordnung von Feldern im Speicher

Für den C–Programmierer ist es in manchen Situationen unumgänglich zu wissen, wie Felder im Speicher angeordnet sind. Grundsätzlich wird für jedes Feld ein zusammenhängender Speicherbereich verwendet.

Bei Vektoren sind die Komponenten mit aufsteigenden Indizes hintereinander angeordnet: Den Index einer Komponente kann man als ihren Abstand vom ersten Element des Vektors interpretieren. Für den Vektor

```
int v[6];
```

ist die Anordnung der Komponenten in Abbildung 11.1 angegeben.

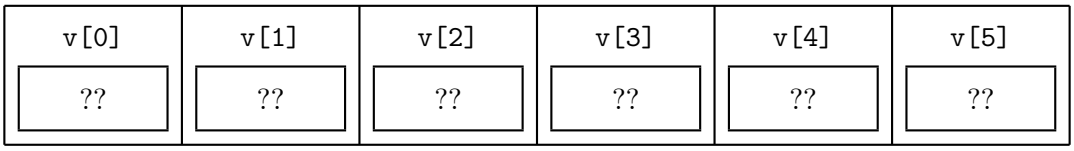

Abbildung 11.1: Speicheranordnung eines Vektors mit 6 Komponenten

Das ist übrigens auch der Grund, warum bei C Felder ab 0 indiziert werden. Bei der Realisierung speziell mathematischer Formeln kann das verwirren, weil in ihnen die Indizierung in der Regel *nicht* bei Null beginnt.

Die Beschreibung einer Matrix als Vektor von Vektoren sollte schon nahelegen, wie Matrizen angeordnet werden: Am Anfang steht der erste Zeilenvektor, dann der zweite Zeilenvektor, usw. Die Elemente der einzelnen Zeilenvektoren folgen jeweils unmittelbar hintereinander. Für die Matrix

```
int m[2][3];
```

ist dies in Abbildung 11.2 wiedergegeben.

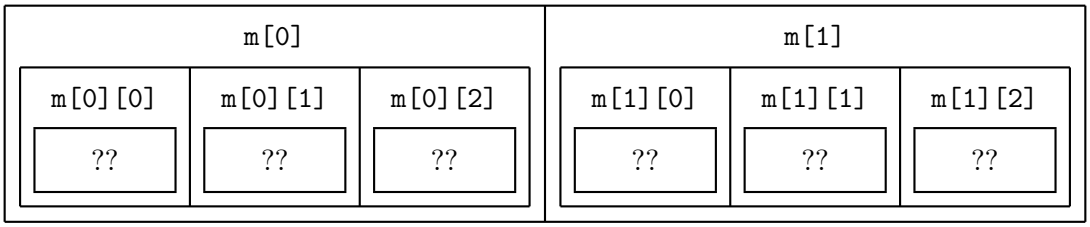

Abbildung 11.2: Speicheranordnung einer (2×3)–Matrix

Für den Zugriff auf eine Komponente einer Matrix muss der Abstand vom ersten Element jetzt wirklich „ausgerechnet“ werden. Haben wir etwa den Zugriff `m[i][j]`, so ist der Abstand gerade $i \cdot 3 + j$, oder allgemeiner: Zeilenindex mal Zeilenlänge plus Spaltenindex. Und genau diesen Ausdruck setzt der Compiler bei jedem Zugriff auf eine Matrixkomponente ein. Die Zuordnung $(i, j) \longmapsto i \cdot 3 + j$ ist in dieser Situation die *Speicherabbildungsfunktion*. Ihre Gestalt hängt offenbar von der Dimensionierung der Matrix ab. Ähnliche Formeln kann man sich übrigens auch für Felder mit höherer Dimension überlegen. Dass jede Feldkomponente möglicherweise mehrere Bytes belegt und deshalb der Abstand noch einmal mit dieser Anzahl multipliziert werden muss, braucht uns nicht zu interessieren.

Damit ist jetzt auch schon angedeutet, was passiert, wenn man mit unzulässigen Indizes auf ein Feld zugreift: Der Abstand wird ausgerechnet und zum Zugriff verwendet. So bewirkt `v[-1]` gerade den Zugriff auf den Speicherbereich direkt *vor* dem Beginn des Vektors. Versuchte Zugriffe auf Feldkomponenten jenseits der Feldgrenzen nennt man *Indexüberschreitung*.

Ob und ggf. was für Folgen eine Indexüberschreitung hat, hängt von der Situation ab. Manche bleiben, von eventuell verfälschten Resultaten abgesehen, ohne Wirkung. So bewirkt zum Beispiel

```
m[0][3] = m[1][0];
```

auf *jeden Fall nichts*! Das Indexpaar `[0][3]` ist zwar nicht zulässig – da es aber denselben Abstand vom Anfang der Matrix liefert wie das Indexpaar `[1][0]`, nämlich 3, wird nur ein `int`–Wert mit sich selbst überschrieben.

Nicht jeder Verstoß endet so ohne Folgen. Da ist es fast schon angenehmer, wenn die Abstände so falsch sind, dass das Programm wegen Zugriffs auf einen „verbotenen“ Speicherbereich „gekillt“ wird, als wenn es mit verfälschten Daten weiterrechnet. An der Stelle, auf die zum Beispiel durch `v[-1]` zugegriffen wird, steht ja in der Regel eine andere Variable des Programms, die durch

```
v[-1] = 0;
```

verfälscht würde.

Wird ein Programm mit einer Meldung wie „segmentation fault“, „bus error“ bzw. „memory fault“ abgebrochen, sollte man das Programm zunächst auf Indexüberschreitungen hin untersuchen.

11.4 Felder als Parameter

Erinnerung: Die *Länge eines Strings*, der durch ein `char`–Feld realisiert ist, ist etwas anderes als die *Länge dieses Feldes*. Die Länge des Strings ist die Anzahl der Bytes vor dem Nullbyte im Feld. Die Länge des `char`–Feldes in dem die Zeichen des Strings gespeichert sind ist die Anzahl der Bytes, die das *komplette* Feld belegt.

Was eine Funktion dagegen nicht bestimmen kann, ist die Länge eines übergebenen Feldes. Wenn eine Funktion die Länge eines als Argument übergebenen Feldes benötigt, so muss ihr diese mit Hilfe eines seperaten Parameters mitgeteilt werden. Selbst berechnen kann sie die Länge nämlich nicht, denn im Speicher folgt einfach Byte auf Byte – eine besondere Grenze, an der die Funktion erkennen könnte, dass das übergebene Feld „zu Ende“ ist, gibt es nicht.

Wegen des Unterschieds zwischen Länge eines Strings und Länge des Feldes durch den ein String realisiert ist, bilden Strings nur scheinbar eine Ausnahme. Als Beispiel betrachten wir eine Funktion, die die Summe der Komponenten eines Vektors berechnet:

```
float vektorsumme(const float v[], int laenge) {
  float s = 0;
  while (--laenge >= 0)
    s += v[laenge];
  return s;
}
```

Der Vorteil dieses Vorgehens ist, dass die Funktion für Vektoren beliebiger Länge eingesetzt werden kann: Der Funktion wird mitgeteilt, wo der Vektor beginnt (`v[]`). Sie unterstellt, dass weitere `laenge - 1` Komponenten folgen. Dass das so ist, ist wieder ausschließlich in die Verantwortung des Programmierers gestellt. Zulässig wäre zum Beispiel

```
float v1[7], v2[50], m[5][20], summe;
...
summe = vektorsumme(v1, 7) + vektorsumme(v2, 50);
summe = vektorsumme(m[3], 20);
```

Nicht zulässig wäre dagegen, bei gleichen Definitionen, wie bereits angesprochen

```
summe = vektorsumme(v1, 10);
```

da die Funktion auf Speicher zugreifen würde, der nicht zu `v1` gehört. Andererseits dürfen wir ohne weiteres

```
summe = vektorsumme(v2, 10);
summe = vektorsumme(&v2[10], 10);
```

schreiben – es steht uns ja frei, einen Teil der Komponenten von `v2` zu ignorieren, hier im ersten Fall die letzten 40 und im zweiten Fall die ersten 10 und die letzten 30. Und schließlich ist auch

```
summe = vektorsumme(m[0], 100);
```

zulässig. Das mag auf den ersten Blick verwunderlich erscheinen – spätestens auf den zweiten Blick wird es klar: Die Matrix `m` besteht aus insgesamt 100 Komponenten, die im Speicher unmittelbar aufeinanderfolgen. Was sollte uns also daran hindern, diesen Speicherbereich bei Bedarf auch als Vektor zu betrachten? Nur `m` als Parameter anzugeben reicht jedoch nicht aus. Ohne Zusatz bezeichnet `m` einen Vektor von Zeilenvektoren, nicht einen Vektor von `float`–Werten. `m[0]` bezeichnet den ersten Zeilenvektor, die weiteren Zeilenvektoren folgen jedoch unmittelbar im Speicher.

Schwieriger wird es, wenn ein mehrdimensionales Feld auch in der Funktion als solches betrachtet werden muss – und wenn obendrein die Größe des Feldes in den einzelnen Dimensionen von Aufruf zu Aufruf verschieden können sein soll. In Deklaration und Definition einer Funktion dürfen wir Felder zwar auch „echt“ dimensionieren, d.h. in die Klammern etwas eintragen – nur müssen das Konstanten oder konstante Ausdrücke sein, deren Wert dann natürlich von Aufruf zu Aufruf nicht verändert werden kann. Andererseits dürfen wir in einer Funktionsvereinbarung nur für die erste Dimension die Längenangabe weglassen. Der Prototyp

```
float f(float m[][3], int zeilen);
```

ist also zwar zulässig, erlaubt den Einsatz der Funktion `f` aber nur für Matrizen mit 3 Spalten – und ist deshalb für die Praxis unzureichend. Der Prototyp

```
float f(float m[][], int zeilen, int spalten);
```

ist *nicht* erlaubt. Der Grund sollte klar sein: Ohne die Längenangabe der Zeilen hat der Compiler keine Möglichkeit, die Speicherabbildungsfunktion zu berechnen.

Die Abhilfe: Man übergibt die Matrix formal als Vektor und gibt diesem Vektor in der Funktion wieder Matrixstruktur, indem man die Speicherabbildungsfunktion selbst berechnet. Das ist zwar nicht schön, in C aber nicht zu vermeiden.

Ein Beispiel ist das Matrizenprodukt $C = A \cdot B$ für Matrizen $A = (a_{ij})$, $B = (b_{jk})$ und $C = (c_{ik})$ mit $i = 1, \ldots, \ell$, $j = 1, \ldots, m$ und $k = 1, \ldots, n$:

$$c_{ik} = \sum_{j=1}^{m} (a_{ij} \cdot b_{jk})$$

Der Prototyp einer entsprechenden Funktion kann

```
void matprod(float c[], const float a[], const float b[],
             int l, int m, int n);
```

sein. Bevor wir die Funktion definieren, deklarieren wir zunächst einen Makro, der zu einem Indexpaar den Abstand vom Anfang des Vektors berechnet. Dieser Makro benötigt zusätzlich zu den beiden Indizes offensichtlich auch die Länge der Zeilen der Matrix als Parameter. In diesen Makro können wir dann auch gleich noch die Transformation der logischen Indizes $1, \ldots, x$ in die C–Indizes $0, \ldots, x-1$ hineinstecken:

```
#define lin(zeile, spalte, laenge)              \
              (((zeile)-1)*(laenge)+(spalte)-1)
```

Jetzt können wir die Funktion definieren – in einer Schreibweise, die der Formel zumindest einigermaßen nahekommt:

```
void matprod (float c[], const float a[], const float b[],
              int l, int m, int n) {
  int i, j, k;
  for (i = 1; i <= l; i++)
    for (k = 1; k <= n; k++) {
      c[lin(i,k,n)] = 0;
      for (j = 1; j <= m; j++)
        c[lin(i,k,n)] += a[lin(i,j,m)] * b[lin(j,k,n)];
    }
}
```

Diese Formulierung entspricht in etwa der Schreibweise der Formel, dafür aber nicht so sehr der Schreibweise von C. Dieses Beispiel wird deshalb im nächsten Abschnitt noch einmal aufgegriffen.

Ein Aufruf der Funktion könnte so aussehen:

```
#define DIM_L ...
#define DIM_M ...
#define DIM_N ...
/* ...                                              */
float A[DIM_L][DIM_M], B[DIM_M][DIM_N],
      C[DIM_L][DIM_N];
/* ... Initialisierung der Matrizen             ... */

matprod (C[0], A[0], B[0], DIM_L, DIM_M, DIM_N);
```

Im übrigen lässt sich auch diese Routine noch nicht wirklich universell für die Multiplikation von zwei `float`–Matrizen einsetzen: Ein Programm, das Matrizen verarbeitet, soll in der Regel bei jedem Aufruf Matrizen unterschiedlicher Größe verarbeiten können. Um das zu realisieren, gibt es in C zwei Möglichkeiten:

- Man arbeitet mit *dynamischem Speicher*. Das bedeutet, man bestimmt zunächst die Größe der zu bearbeitenden Matrizen und beschafft sich dann vom Betriebssystem den entsprechenden Speicher. Methoden dazu werden im Kapitel 13 vorgestellt. Der Einsatz unserer Funktion `matprod` wäre dann möglich.
- Man arbeitet mit „Maximalgrößen“ für die Matrizen, d.h. man überlegt sich eine obere Schranke für die Größe der vorkommenden Matrizen und definiert alle Matrizen mit dieser Größe. Das wird in mathematischen Programmen oft gemacht, hat aber auch seine Nachteile:
  - Sollte der Benutzer des Programms größere Matrizen verarbeiten wollen, muss die Verarbeitung abgelehnt werden.
  - Funktionen wie `matprod` benötigen zusätzliche Parameter: Für die Schleifen wird die „logische“ (genutzte) Größe der Matrizen benötigt; für die Berechnung der Speicherabbildungsfunktion wird dagegen die „physikalische“ (vereinbarte) Größe der Matrizen benötigt.

  Leider fehlt die Zeit, auf Einzelheiten einzugehen. Vielleicht formulieren Sie selbst die Funktion `matprod` einmal so um, dass sie auch dann korrekt arbeitet, wenn die Matrizen nur einen Teil des für sie bereitgestellten Speichers belegen.

11.5 Initialisierung von Feldern

Felder können initialisiert werden. Für Strings haben wir diese Möglichkeit auch bereits genutzt. Allgemein sieht die Anfangswertzuweisung für Felder aber anders aus: Die Werte der Komponenten werden einzeln aufgeführt, voneinander jeweils durch ein Komma getrennt und insgesamt in geschweifte Klammern eingeschlossen:

```
int v[6] = {6, 5, 4, 17, 18, 19};
```

Gibt man zu wenig Werte an, so werden die übrigen auf Null gesetzt; gibt man zu viele Werte an, wird der Compiler einen Fehler anzeigen. Wenn man für jede Komponente einen Wert angibt, kann man den Compiler auch abzählen lassen und die Feldlänge ganz weglassen: Die Definition

```
int v[] = {6, 5, 4, 17, 18, 19};
```

ist also zulässig und ergibt einen Vektor mit 6 Komponenten.

Bei der Initialisierung mehrdimensionaler Felder muss man sich wieder klar machen, dass sie formal Vektoren von Vektoren sind. Entsprechend kann man dann zum Beispiel

```
int m[2][3] = {{ 6,  5,  4},
               {17, 18, 19}};
```

schreiben.

Wenn man Anfangswerte angibt, darf man bei mehrdimensionalen Feldern die Länge in der ersten Dimension weglassen; der Compiler bestimmt sie dann durch Abzählen. Das ist nicht nur für Schreibfaule interessant, sondern erspart einem einerseits das Abzählen der Daten und verhindert andererseits ein Verzählen dabei.

Auch mehrdimensionale Felder lassen sich teilweise initialisieren. Alle nicht angegebenen Komponenten erhalten dann den Wert `0`. Dabei wird die durch geschweiften Klammern angegebene Struktur eingehalten.

```
float v[12] = { 0 };   /* alle Komponenten = 0  */

int m[4][2][3] = {{{ 1, 2, 3 }, {  4, 5, 6 }},
                  {{ 7, 8 }},
                  {{ 9 }, { 10 }}};
/* entspricht  = {{{ 1, 2, 3 }, {  4, 5, 6 }},
                  {{ 7, 8, 0 }, {  0, 0, 0 }},
                  {{ 9, 0, 0 }, { 10, 0, 0 }},
                  {{ 0, 0, 0 }, {  0, 0, 0 }}}  */
```

Im Beispiel hätte man wiederum die erste Dimension des Feldes `m` weglassen können. Der Compiler hätte für sie dann 3 bestimmt, da für drei Zeilen Anfangswerte angegeben wurden.

Für Strings haben wir bereits eine Kurzform der Initialisierung kennengelernt: Wenn eine Stringkonstante angegeben ist, zerlegt der Compiler diese in einzelne Zeichen und sorgt selber für das abschließende Null–Zeichen. Die beiden Definitionen

```
char str1[6] = "hallo",
     str2[6] = {'h', 'a', 'l', 'l', 'o', '\0'};
```

sind also äquivalent. Und da wir auch hier wieder den Compiler abzählen lassen können, sind die beiden Definitionen

```
char str3[] = "hallo",
     str4[] = {'h', 'a', 'l', 'l', 'o', '\0'};
```

ebenfalls äquivalent.

Alle Konstanten, die in den Initialisierungen angegeben wurden, dürften übrigens auch konstante Ausdrücke sein, bei automatischen Feldern sogar beliebige Ausdrücke.

Ein Manko ist sicher, dass man alle Anfangswerte explizit angeben muss. Eine Abkürzung wie „trage 10mal den Wert 7 ein“ gibt es nicht! Auch kann man nicht nur für einzelne Feldkomponenten Anfangswerte angeben.

Kapitel 12

Zeiger

12.1 Adressen und Zeiger

Eine Variable besitzt sowohl einen Speicherplatz als auch einen Wert. Beide benötigen wir je nach Kontext. Sehen wir uns das für eine Wertzuweisung `v = e;` im einzelnen an:

- Im Ausdruck auf der rechten Seite benötigen wir als Operanden *Werte*; wenn als Operanden Variablen angegeben sind, müssen die Werte der Variablen verwendet werden, nicht ihre Adressen.
- Links vom Zuweisungsoperator benötigen wir die *Adresse* der Variablen, in der der Wert des Ausdrucks gespeichert werden soll.

Die Theorie spricht von *Dereferenzierung*: Der Name einer Variablen repräsentiert die Adresse der Variablen; durch Dereferenzierung erhält man ihren Wert. In den Programmiersprachen ist es allgemein *nicht* üblich, dies auch formal nachzuvollziehen, weil ohnehin Verwechslungen nicht zu befürchten sind:

- Weil als Operand in einem Ausdruck meist nur der Wert einer Variablen Sinn macht, nicht jedoch ihre Adresse, wird die Dereferenzierung automatisch vorgenommen.
- Weil links vom Zuweisungsoperator einer Wertzuweisung nur eine Adresse Sinn macht, nicht jedoch der Wert einer Variablen, unterbleibt die Dereferenzierung.

Man muss aber noch weiter differenzieren: Bei den kombinierten Wertzuweisungen und den Inkrement– und Dekrementoperatoren muss sowohl dereferenziert als auch die Adresse verwendet werden. Betrachen wir als Beispiel die beiden gleichwertigen Anweisungen

```
i++;
i += 1;
```

Zunächst wird `i` (automatisch) dereferenziert, um den *Wert* von `i` zu erhalten. Dann wird dieser Wert um 1 erhöht. Schließlich wird die *Adresse* von `i` verwendet, um das Resultat der Operation zu speichern. Aber auch dieses erfolgt, wie gesehen, automatisch und damit letztlich für den Programmierer verdeckt.

Nicht in jedem Falle ist die automatische Dereferenzierung erwünscht. Gesehen haben wir das zum Beispiel bei der Funktion `scanf`: Da die Funktion Werte in Variablen speichern soll, ist ihr mit *Werten* nicht gedient – sie benötigt *Adressen*! Wir haben auch folgendes bereits gesehen (jetzt in der neuen Terminologie): Der *Adressoperator* `&` verhindert, dass Variablen dereferenziert werden! Einsetzen kann man diesen Operator prinzipiell überall – nur macht er ohne weiteres an den meisten Stellen keinen Sinn.

Die Umkehrung des Adressoperators `&` ist der *Dereferenzierungsoperator* `*`: Er bewirkt, dass sein Operand derefenziert wird, oder, anders formuliert: Er bewirkt, dass der Wert

seines Operanden dereferenziert und der dabei resultierende Wert als Wert des Ausdrucks betrachtet wird.

Der Dereferenzierungsoperator hebt den Adressoperator in seiner Wirkung offensichtlich auf. Wir sehen uns dazu das Beispiel

```
int i = 42, j;
j = *&i;
```

an: Der Teilausdruck `&i` liefert die Adresse von `i` als Resultat. Durch den Operator `*` wird der Wert des Teilausdrucks `&i` dereferenziert und liefert den Wert von `i` als Resultat. Die Wertzuweisung entspricht insgesamt also `j = i;` – nur umständlich ausgedrückt.

In C bezeichnet man Adressen als *Zeiger*. Ein Ausdruck wie `&i` wird entsprechend als *Zeigerausdruck* bezeichnet.

12.2 Zeiger als Funktionsparameter (I)

Wie bereits erwähnt erfolgt bei Funktionsaufrufen ein „call by value“. Das bedeutet, nur der *Wert* des Aufrufarguments – z.B. einer Variable `i` – wird übergeben. Demzufolge kann ein Funktionsaufruf nicht auf die Variable `i` zurückwirken, ihren Wert also nicht verändern. Dies wird weiter unten noch einmal an einem Beispiel erläutert.

Bei Feldern als Funktionsparameter ist das Verhalten scheinbar anders. Wie schon erwähnt, wird der Name eines Feldes (ohne Indizes) von C als Zeigerkonstante betrachtet. Bedeutung hatte das für uns im Zusammenhang mit Funktionen: Sehen wir als Parameter ein Feld vor und übergeben wir als Argument ein Feld durch seinen Namen, so haben wir in der Funktion direkten Zugriff auf das Feld. Entsprechend war es uns möglich, nicht nur auf die Werte der Komponenten zuzugreifen, sondern auch die Werte nachhaltig zu verändern. Wie lässt sich dasselbe für einfache Variablen erreichen?

Wie der Aufruf einer solchen Funktion auszusehen hat, wissen wir bereits von `scanf`: Anstelle des Namens der Variablen, etwa `v`, tragen wir den Ausdruck `&v` ein. Das bewirkt, wie wir eben noch einmal gesehen haben, dass die Dereferenzierung von `v` unterbleibt und entsprechend der Zeiger auf den Speicherplatz von `v` anstelle des Wertes von `v` an die Funktion übergeben wird.

Bei der Einführung der Funktionen wurde schon erwähnt, dass auch in der Definition einer Funktion Änderungen erforderlich sind, wenn Zeiger übergeben werden. Diese Änderungen betreffen zwei Punkte:

- Im Funktionsheader muss, sowohl in der Deklaration als auch in der Definition, markiert werden, dass ein Parameter ein Zeiger ist. Das geschieht, indem dem Namen des Parameters ein Stern (`*`) vorangestellt wird.
- Im Funktionsrumpf muss dem Namen des Parameters in der Regel ebenfalls an jeder Stelle ein Stern vorangestellt und damit explizit dereferenziert werden.

Als Beispiel betrachten wir eine „Trivialfunktion“: Sie soll nur den Wert einer Variablen der rufenden Funktion um 1 erhöhen. Bisher hätte ein Versuch wohl so ausgesehen:

```
void inkrement1(int i) {
  i = i + 1;
}
```

So funktioniert das nicht: `inkrement1` erhält eine Kopie des Wertes von `i`. Wenn, wie hier geschehen, `inkrement1` diese Kopie verändert, bleibt das für die rufende Funktion ohne Folgen. Wir müssen deshalb

```
void inkrement2(int *i) {
  *i = *i + 1;
}
```

schreiben: `i` selbst ist jetzt eine *Zeigervariable*. Das zeigt die Parametervereinbarung `int *i` an. Die (automatische) Dereferenzierung von `i` im Zuge der Auswertung eines Ausdrucks liefert wie bisher den Wert von `i` – nur ist das jetzt ein Zeiger auf eine `int`–Variable und nicht mehr ein `int`–Wert. Wollen wir auf den `int`–Wert zugreifen, müssen wir also noch einmal explizit dereferenzieren. Und das geschieht gerade durch `*i`.

Rufen wir die Funktion jetzt durch

```
j = 1;
inkrement2(&j);
```

so hat `j` nach dem Aufruf den Wert 2.

12.3 Zeigervariablen

Nicht nur Parameter von Funktionen dürfen Zeigertypen erhalten, sondern auch beliebige andere Variablen. Dabei ist der Formalismus derselbe: In der Vereinbarung wird dem Namen ein Stern vorangestellt. Sollen mehrere Variablen in einer Zeile definiert werden, so wird für jede einzeln durch das vorangestellte Sternchen entschieden, ob es sich um eine Zeigervariable oder eine normale Variable des Typs handelt. So werden durch

```
int i, *k, j;
```

zwei `int`–Variablen `i` und `j` deklariert und eine Zeigervariable `k`, deren Wert einen Zeiger auf eine `int`–Variable repräsentiert. Für `i` und `j` gibt es keine Unterschiede zur bisherigen Verwendung. Mit `k` dürfen wir jetzt zum Beispiel

```
k = 10;
```

keinesfalls mehr schreiben: Die Anweisung würde ja bedeuten, dass der Wert 10 in der Folge als Zeiger auf eine `int`–Variable zu interpretieren wäre – es gibt jedoch für den C–Programmierer keine Möglichkeit, das sicherzustellen, weil die Zuordnung von Adressen zu den Variablen eines Programms *ausschließlich* dem Compiler (und Linker) bzw. den Funktionen zur dynamischen Speicherbereitstellung der Standardbibliothek vorbehalten ist. Eine zulässige Anweisungsfolge wäre dagegen

```
int i, j, *k;

k = i < j ? &i : &j;
*k = 10;
```

Sehen wir uns an, was im einzelnen passiert:

- Der Wert, der `k` letztlich zugeordnet wird, ist entweder `&i` oder `&j`. Beide Werte sind Zeiger auf eine `int`–Variable und stimmen damit im Typ mit `k` überein. Entsprechend ist die Wertzuweisung zulässig und speichert in `k` einen Zeiger auf eine `int`–Variable.
- Bei der Dereferenzierung von `k` resultiert der Zeiger auf eine `int`–Variable, hier also aufgrund der Wertzuweisung entweder der Zeiger auf `i` oder `j`. Entsprechend wird der `int`–Wert entweder in die `int`–Variable `i` oder in die `int`–Variable `j` übertragen.

C verlangt auch für Zeiger eine Typbindung. So haben wir bislang in den Beispielen „Zeiger auf `int`“ verwendet. Das hat Vorteile: Im letzten Beispiel etwa „weiß“ der Compiler, dass `*k` den Typ `int` besitzt. Er kann damit bei Bedarf den Typ eines zuzuweisenden Wertes oder beim Zugriff mit `*k` den Typ `int` „geeignet“ umwandeln. Die Wertzuweisung

```
*k = 7.567;
```

würde so ein Abschneiden der Stellen hinter dem Punkt bewirken. Hätte `k` nur den Typ „Zeiger“, wäre eine solche automatische Umwandlung prinzipiell nicht möglich.

12.4 Zeiger und `const`

Das Zusammenspiel zwischen Zeigervariablen und dem Schlüsselwort `const` ist komplex. Bei der Deklaration von Zeigervariablen kann das Schlüsselwort `const` an verschiedenen Stellen stehen. Steht es links von `*`, so bedeutet es, dass die von der Zeigervariablen referenzierten Werte konstant sind. Die Werte, auf die der Zeiger verweist, können also nicht geändert werden. Man kann der Zeigervariablen jedoch jederzeit einen neuen Wert zuweisen. An welcher Stelle `const` links von `*` steht, hat keine Bedeutung. Im Folgenden bezeichnen `i` und `j` Variablen vom Typ `int`.

```
const int *p = &i;   /* oder: int const *p = &i; */

*p = 2;  /* FEHLER: referenzierter Wert konstant */
p = &j;  /* erlaubt                              */
```

Eine andere Bedeutung hat `const` jedoch rechts von `*`. Dort bewirkt es, dass die Zeigervariable selbst konstant ist. D.h. einmal initialisiert kann man ihr keinen neuen Adresswert zuweisen. Der von ihr referenzierte Wert kann jedoch geändert werden.

```
int * const p = &i;

*p = 3;  /* erlaubt                              */
p = &j;  /* FEHLER: Zeigervariable konstant      */
```

Eine Kombination aus beiden Fällen ist ebenfalls möglich. Es gelten dann entsprechend beide Einschränkungen für die Zeigervariable.

```
int const * const p = &i;

*p = 4;  /* FEHLER: referenzierter Wert konstant */
p = &j;  /* FEHLER: Zeigervariable konstant      */
```

Man kann sich die Bedeutung leicht merken, indem man die Deklaration von rechts nach links liest. Im zweiten Beispiel kann man so lesen: „`p` ist ein *konstanter* Zeiger auf `int`–Werte“.

Beim Umgang mit Zeigern und `const` ist besondere Aufmerksamkeit erforderlich. Durch Zuweisungen kann das `const`–Attribut versehentlich verloren gehen. Der Standard sieht das nicht als Fehler an und der Compiler produzieren entsprechend meist nur eine Warnung.

```
const int k = 42;
const int *p = &k;
int *q = &k;           /* WARNUNG: const Attribut */
                       /*   wird abgestreift      */
*q = 43;               /* k hat nun den Wert 43!! */

q = p;                 /* WARNUNG: ...            */
```

Das Schlüsselwort `const` kann natürlich auch für Funktionsargumente mit der gleichen Bedeutung wie bei Variablendeklarationen verwendet werden. In der Praxis wird man es dort sogar wesentlich häufiger antreffen.

Zeiger als Parameter von Funktionen werden in C verwendet, um „*call by reference*“ zu simulieren. So wird das Zeigerargument selbst zwar kopiert, durch Dereferenzierung erhält man aber direkten Zugriff auf die eigentlichen Daten und kann diese ändern. Oft möchte man mit einem Zeigerargument aber nur das unter Umständen aufwändige automatische Kopieren der Argumente umgehen, da sich bei großen Datenstrukturen die Parameterübergabe mit „*call by value*“ negativ auf die Laufzeit auswirken könnte. Das Schlüsselwort `const` im Prototyp einer Funktion ist dann ein wertvoller Hinweis, dass die Daten nicht verändert werden.

12.5 Zeiger und Felder (I)

In C besteht ein enger Zusammenhang zwischen Zeigern und Feldern. Wir haben auch bereits gesehen:

- C betrachtet den Namen eines Feldes ohne Indizes als Zeigerkonstante. Ebenso ist der Name eines Feldes ein Zeiger, wenn er mit weniger Indizes angegeben wird als das Feld Dimensionen hat.
- Als Zeiger repräsentiert ein Feldname die Adresse der ersten Komponente des Vektors.
- Der Zugriff auf eine Feldkomponente erfolgt durch Berechnung der Speicherabbildungsfunktion für ihre Indizes.

Verfolgt man diese Prinzipien konsequent weiter, so wird es möglich, Formulierungen wie

```
int v[10], *p;
...
p = &v[0];
p = v;
```

zu verwenden.

Dass die erste der beiden Zuweisungen zulässig ist, haben wir oben schon gesehen: Zunächst wird die Speicherabbildungsfunktion berechnet, weil die eckigen Klammern als Operatoren höchste Priorität besitzen. Dann unterbleibt wegen des Adressoperators die Dereferenzierung, der Wert des Ausdruck `&v[0]` ist ein Zeiger und der Typ des Ausdrucks ist „Zeiger auf eine `int`–Variable“.

Die zweite der beiden Zuweisungen ist die Folgerung aus den eben angestellten Überlegungen: Der Ausdruck `v` ist der Zeiger auf die erste Komponente des Feldes `v`.

Der Standard legt fest, dass Variablennamen (einschließlich Feldkomponenten) automatisch dereferenziert werden, Feldnamen dagegen nicht. Laut Standard ist auch die Anweisung

```
p = &v;
```

nicht verboten, aber der Typ des Ausdrucks `&v` ist nur „Zeiger“ und nicht „Zeiger auf eine `int`–Variable“.

Wir können das Beispiel nun fortsetzen. Erlaubt ist jetzt zum Beispiel

```
int v[10], *p;

p = v;
*p = 17 + 4;
```

Der Wert des Ausdrucks `17 + 4` wird ausgewertet und sein Resultat in der Variablen gespeichert, auf die der Wert der Zeigervariablen `p` zeigt, also in `v[0]`. Die letzte Zuweisung entspricht also gerade

```
v[0] = 17 + 4;
```

Zur Erinnerung noch einmal: Namen von Feldern sind Zeigerkonstanten. Das schließt Zuweisungen wie

```
int v[10], *p, i;

v = p;
v = &i;
```

von vornherein aus, auch wenn in beiden Zuweisungen die Typen der Ausdrücke links und rechts vom Zuweisungsoperator übereinstimmen. Von der Logik her verlangen sie dasselbe wie die Wertzuweisung

```
7 = i;
```

12.6 Zeigerarithmetik

Was wir bislang über Zeiger gelernt haben, könnte als Sammlung von Formalien erscheinen, die zumindest auf den ersten Blick keinen Nutzen haben – abgesehen einmal von der Möglichkeit, „call by reference“ zu simulieren. Wirklich interessant werden Zeiger tatsächlich auch erst durch zwei zusätzliche Möglichkeiten:

- Da Zeigervariablen ihre Werte während der Ausführungszeit eines Programms ändern können, kann man mit ihnen auf Speicher zugreifen, der erst während der Ausführung des Programms bereitgestellt wird. Diese Möglichkeit kennen praktisch alle modernen Programmiersprachen; das Stichwort ist „Speicher auf dem Heap“. Die Besprechung dieser Möglichkeit wird auf Kapitel 13 verschoben.
- Die andere Möglichkeit ist eine Eigenart von C: Mit Zeigern kann man – in gewissen Grenzen – auch rechnen. Und damit müssen wir uns jetzt beschäftigen.

Der Ausgangspunkt sind erneut die Felder: Schreiben wir etwa

```
int v[10];

v[7] = 9;
```

so bewirkt `v[7]` ja gerade, dass auf das Element zugegriffen wird, das den Abstand 7 vom Anfang des Feldes hat. Anders formuliert: Der Zeiger auf den Anfang des Feldes, repräsentiert durch den Namen `v`, wird um 7 erhöht und der resultierende Wert für den Zugriff verwendet. Wir dürften das auch als

```
*(v + 7) = 9;
```

hinschreiben; die Wirkung entspricht gerade der eben gegebenen Beschreibung. Die Klammern sind dabei *unbedingt* nötig, da der Dereferenzierungsoperator wie alle unären Operatoren eine höhere Priorität besitzt als die binären Operatoren.

Entsprechend kann man nun natürlich auch

```
int v[10], *p;

p = v + 7;
*p = 9;
```

schreiben – und sind damit bei „echter“ Zeigerarithmetik angelangt: Was hindert uns daran, etwa

```
p++;
p += 3;
```

zu schreiben?

Damit kommt man zu formal ganz anderen Formulierungen von Programmen, die auf Feldern operieren. Im letzten Abschnitt hatten wir zum Beispiel zur Berechnung der Summe der Komponenten eines Vektors die Funktion

```
float vektorsumme(const float v[], int laenge) {
  float summe = 0;
  while (--laenge >= 0)
    summe += v[laenge];
  return summe;
}
```

formuliert. Die C–typische Formulierung wäre aber

```
float vektorsumme(const float *v, int laenge) {
  float summe = 0;
  while (laenge--)
    summe += *v++;
  return summe;
}
```

Einige Anmerkungen hierzu:

- Die Parameterdefinitionen `const float v[]` und `const float *v` sind gleichwertig:
  - Beide Definitionen besagen, dass das Argument ein Zeiger auf eine `float`–Variable mit konstantem Wert sein muss/wird.
  - Dass die Variable als erste Komponente eines Vektors zu betrachten ist, ergibt sich bei der ersten Definition direkt aus der Definition, während es bei der zweiten Definition aus der Logik der Funktion entnommen werden muss. Wenn `v` keinen Vektor bezeichnet, sollte man deshalb zur Verdeutlichung `const float *const v` schreiben.
- Beim Aufruf wird der Wert des ersten Arguments in die lokale Variable `v` der Funktion kopiert, so dass die Änderung von `v` durch `v++` auch dann zulässig ist, wenn das Argument eine Konstante ist (hier: der Name eines Vektors).
- Beim Ausdruck `*v++` nutzen wir, dass die unären Operatoren alle dieselbe Priorität besitzen und dass sie ggf. von rechts nach links abgearbeitet werden. Der Ausdruck ist also gleichwertig mit `*(v++)`.

12.7 Operationen mit Zeigern

Arithmetik mit Zeigern ist nur eingeschränkt möglich. Wir müssen uns deshalb einmal im Zusammenhang ansehen, welche Operationen für Zeiger überhaupt zulässig sind:

- Zeigern können Werte zugewiesen werden. Wir haben bereits verschiedentlich Wertzuweisungen für Zeiger gesehen.
- Ganze Zahlen können auf Zeiger addiert und von ihnen subtrahiert werden. Welche Operatoren man dazu verwendet, ist gleichgültig. So sind für eine Zeigervariable `p` und einen `int`–Ausdruck `i` die Ausdrücke

```
p++
p + i
p--
p - i
```

sämtlich (formal) zulässig.

Über das, was logisch sinnvoll ist, müssen wir uns noch besonders Gedanken machen.

- Die Differenz von zwei (typgleichen) Zeigern kann gebildet werden. Das funktioniert so: Es wird unterstellt, dass beide Zeiger auf Komponenten ein und desselben Vektors zeigen. Die Differenz entspricht dann gerade der Differenz der Indizes der beiden Komponenten.

  Eine solche Differenz besitzt einen ganzzahligen, vorzeichenbehafteten Typ – welcher es im einzelnen ist, muss durch die Deklaration des Typs `ptrdiff_t` in der Standard–Headerdatei `stddef.h` festgelegt werden.

  Die Anweisungsfolge

  ```
  int v[10], *p, *q;
  ptrdiff_t d1, d2;

  p = &v[3];
  q = &v[7];
  d1 = p - q;
  d2 = q - p;
  ```

  liefert so in `d1` den Wert -4 und in `d2` den Wert $+4$.

  Die Addition zweier Zeiger ist dagegen unzulässig. Was sollte für ein Feld `v` auch `v + v` bedeuten? Diesem Umstand ist besonders Rechnung zu tragen, wenn Terme vereinfacht werden sollen. Seien z.B. `start` und `ende` Zeiger auf den Anfang und das Ende eines Feldes. Soll nun ein Zeiger auf die Mitte des Feldes berechnet werden, so kann das folgendermaßen geschehen:

  ```
  int v[20], *start = v, *ende = v + 20, *mitte;
  mitte = start + (ende - start)/2;
  ```

  Wendet man nun einfache mathematische Termumformungen an, könnte man auf folgende kürzere Schreibweise kommen

  ```
  mitte = (start + ende)/2;    /* nicht zulaessig */
  ```

  die für beliebige Ganzzahlwerte sicher richtig wäre, und mit einer Operation weniger auskommt. Für Zeiger ist dieser Ausdruck jedoch nicht zulässig.

- Zeiger können verglichen werden. Dafür sind alle 6 Vergleichsoperatoren zulässig. Der Ablauf entspricht der Bildung der Differenz von zwei Zeigern: Es wird unterstellt, dass beide Zeiger auf Komponenten ein und desselben Vektors zeigen. Der Vergleich liefert dasselbe Resultat wie ein Vergleich der Indizes.

  Die Berechnung der Vektorsumme könnten wir zum Beispiel auch so formulieren:

  ```
  float vektorsumme(const float *v, int laenge) {
    float summe = 0, *p = v + laenge;
    while (v < p)
      summe += *v++;
    return summe;
  }
  ```

Auf den ersten Blick scheint hier `v + laenge` nicht erlaubt, weil das Resultat nicht auf eine Komponente des Feldes zeigt, sondern direkt *hinter* das Feld. Dies lässt der Standard allerdings ausdrücklich zu. Diese Technik wird häufig verwendet, wenn Bereiche aus einem Feld durch zwei Zeiger begrenzt werden sollen. Der erste Zeiger wird auf das Anfangselement gesetzt, während der zweite genau hinter das letzte Element des Bereichs zeigt. Das hat den Vorteil, dass die Differenz der beiden Zeiger gerade die Anzahl der Elemente im Bereich angibt und `for`–Schleifen wie gewohnt mit `<` als Abbruchbedingung formuliert werden können.

- Es gibt einen Makro `NULL`, der den *Nullzeiger* definiert. Der Wert dieses Makros muss von den Implementatoren so gewählt werden, dass kein „echter“ Zeiger ihn haben kann. Er erlaubt die Markierung von Zeigervariablen als „ohne Wert“. Definiert wird der Makro `NULL` in den Standard–Headerdateien `stddef.h`, `stdio.h` und `string.h`. Trotz der Bezeichnung sollte man sich nicht darauf verlassen, dass der Nullzeiger den Wert 0 hat und stattdessen immer `NULL` verwenden. So wird auch sofort klar, dass es sich um einen Zeiger handelt.

12.8 Zeiger als Parameter (II)

Beim Beispiel der Vektorsumme haben wir bereits gesehen, dass es bei Funktionen letztlich gleichgültig ist, ob wir einen Parameter als Vektor oder als Zeiger deklarieren. Zeiger sind allerdings typischer für C – und erlauben vielfach extrem kompakte Formulierungen von Funktionen. Zur Demonstration sollen zwei Beispiele dienen. Dass beide Beispiele Strings bearbeiten, ist nicht nur Zufall.

Erstes Beispiel: Zu kopieren ist ein String. Unter Verwendung von Feldern können wir schreiben

```
void strcpy1(char s[], const char t[]) {
  int i = 0;
  while (s[i] = t[i])
    i++;
}
```

Zeiger erlauben die Formulierung

```
void strcpy2(char *s, const char *t) {
  while (*s++ = *t++)
    ;
}
```

Zweites Beispiel: Zwei Strings sind zu vergleichen. Der Funktionswert soll kleiner, gleich oder größer als Null sein, je nachdem, ob der erste String (lexikographisch) kleiner, gleich oder größer als der zweite String ist. Für Vektoren können wir die Lösung so formulieren:

```
int strcmp1(const char s[], const char t[]) {
  int i = 0;
  while (s[i] == t[i] && s[i])
    i++;
  return s[i] - t[i];
}
```

Mit Zeigern kann die Lösung so aussehen:

```
int strcmp2 (const char *s, const char *t) {
  while (*s == *t && *s)
    s++, t++;
```

```
    return *s - *t;
}
```

Ganz groß sind die Unterschiede nicht, wie man sieht. Was bei den Zeiger–Varianten entfällt, sind die lokalen Indexvariablen. Im übrigen sind, wie wir bereits gesehen haben, die Formulierung des Funktionsheaders und des Funktionsrumpfes voneinander unabhängig. Wir können also ohne weiteres im Header eine Vektor– und im Rumpf eine Zeigerformulierung verwenden und umgekehrt.

12.9 Probleme mit Zeigern

Mit Zeigern kann man mancherlei Unsinn treiben. Bei der Beschreibung der Operationen mit Zeigern wurde zweimal vorausgesetzt, dass die Zeiger in einem speziellen Zusammenhang zueinander stehen müssen. Nämlich bei der Berechnung der Differenz und beim Vergleich von zwei Zeigern. Sehen wir uns dazu ein Beispiel an:

```
int v1[10], v2[20];

if (v1 < v2)
   ...
```

Als Programmierer hat man keinen Einfluss darauf, wie der Compiler die Variablen (und Felder) eines Programms im Speicher anordnet! Ob die Bedingung `v1 < v2` in unserem Beispiel *wahr* oder *falsch* ist, hängt aber gerade davon ab, ob das Feld `v1` vom Compiler *vor* oder *hinter* dem Feld `v2` in den Speicher gelegt wurde. Ein Vergleich von zwei Zeigern macht in der Regel nur dann Sinn, wenn beide Zeiger in das gleiche Feld zeigen.

Auch ein falscher Indexzugriff oder allgemein das Dereferenzieren eines ungültigen Zeigers kann zu Problemen führen und ist häufig Ursache von Programmabstürzen. Das untersuchen wir anhand eines Beispiels:

```
int v[10], w[50], i, *p = v;
/* Berechnung von i */
v[i] = 44;
*(p + i) = 44;
```

Welchen Wert `i` hat, liegt erst fest, wenn die Wertzuweisungen ausgeführt werden. Allerdings:

- Es ist für den Compiler natürlich ein Kleines, zusätzliche Maschinenbefehle zu erzeugen, mit denen geprüft wird, ob der Wert von `i` zwischen 0 und 9 liegt, bevor die erste Wertzuweisung ausgeführt wird.
- Bei der zweiten Wertzuweisung ließen sich solche Prüfungen allenfalls mit erheblichem Aufwand erzeugen: Worauf `p` zeigt, wird erst während der Ausführung des Programms festgelegt. Der zulässige Wertebereich im Beispiel kann also 0 bis 9 sein; er kann aber auch 0 bis 49 sein, wenn `p` auf den Anfang von `w` „umgesetzt“ wurde. Wenn `p` durch Zeigerarithmetik verändert wurde, wird es endgültig unmöglich, den zulässigen Wertebereich für `i` zu bestimmen.

Dass die Auswertung von Zeigerausdrücken immer vernünftige Werte ergibt, ist ausschließlich in die Verantwortung des Programmierers gestellt. Verstöße wirken sich in der Regel als Abstürze des Programms aus sobald ein ungültiger Zeigerausdruck dereferenziert wird.

Besonders „schöne“ Möglichkeiten eröffnen aber auch die Castoperatoren. Zeiger sind ja an Typen gebunden. Schreiben wir etwa

```
int *ip;
float *fp;
...
fp = ip;
```

so wird der Compiler hoffentlich eine Fehlermeldung ausgeben. Wir könnten jetzt auf die Idee kommen, das durch

```
fp = (float *) ip;
```

zu „reparieren“. Um die Abläufe etwas zu verdeutlichen, muss das Beispiel etwas erweitert werden:

```
int i, *ip = &i;
float *fp;
...
fp = (float *) ip;
*fp = 6.5f;
printf ("%d\n", i);
```

Letztlich haben wir hier den Typ von `i` (temporär) geändert: Mit `*fp` greifen wir auf den Speicherplatz der `int`–Variablen `i` zu. Bei der Wertzuweisung wäre also eigentlich eine Umwandlung des Typs des Ausdrucks `6.5f` von `float` in `int` erforderlich – diese Umwandlung unterbleibt aber, weil `*fp` als Zeiger auf `float` definiert ist. Statt dessen wird die Bitfolge unverändert übertragen. Durch die Funktion `printf` wird diese Bitfolge nur wieder als `int`–Wert interpretiert – dass das nicht funktionieren kann, sollte klar sein; was man erhält, lässt sich dagegen allgemein *nicht* vorhersagen. Vielleicht probieren Sie es selbst einmal aus.

Hinter der Typbindung der Zeiger steht aber nicht nur die Notwendigkeit, ggf. Typen umzuwandeln. Auch für Operationen mit den Zeigern selbst wird der Typ vielfach benötigt. Um bei Zeigerarithmetik eine konkrete Adresse im Speicher zu berechnen, wird die Größe des entsprechenden Typs benötigt. Beim Inkrementieren eines Zeigers muss die Adresse ggf. um *mehrere* Bytes erhöht werden, damit der Zeiger auf das nächste Element zeigt. Durch die Bindung des Typs an den Zeiger steht auch diese Information zur Verfügung.

Ein Beispiel: Auf gängigen Maschinen braucht man mehrere Bytes um einen int-Wert zu speichern - z.B. auf 32-Bit Maschinen genau vier. Bei dem Vektor

```
int v[10];
```

ist die Speicheradresse von `v[1]` also um 4 größer als die Speicheradresse von `v[0]` und nicht nur um 1. In der Regel ist das kein Problem, weil der Compiler bei allen Zeigeroperationen den Speicherbedarf des jeweiligen Typs berücksichtigt. Schreiben wir aber etwa

```
int v[10];
char *s1, *s2;
ptrdiff_t i;
...
s1 = (char *) &v[0];
s2 = (char *) &v[1];
i = s2 - s1;
```

so erhält `i` allenfalls auf Rechnern mit sehr ausgefallener Organisation den Wert 1; in der Regel wird der Wert 2 oder 4 sein, je nachdem, wie viele Bytes der Rechner für einen `int`–Wert verwendet.

12.10 Zeiger und Felder (II)

Verschiedentlich wurden in Variablendefinitionen bereits Anfangswertzuweisungen für Zeigervariablen verwendet, ohne darauf besonders einzugehen. Daraus können Sie entnehmen, dass dieses ohne weiteres möglich ist. Trotzdem gibt es einige Dinge zu beachten, die sich insbesondere bei Strings auswirken.

Am Ende des letzten Abschnitts hatten wir vier weitgehend äquivalente Möglichkeiten kennengelernt, einer Stringvariablen einen Anfangswert zu geben:

```
char str1[6] = "hallo",
     str2[6] = {'h', 'a', 'l', 'l', 'o', '\0'},
     str3[]  = "hallo",
     str4[]  = {'h', 'a', 'l', 'l', 'o', '\0'};
```

Hierdurch werden, wohlgemerkt, nur Anfangswerte festgelegt. Wir dürfen also ohne weiteres

```
str1[1] = 'o';
str1[4] = 'a';
```

schreiben, wodurch der Wert zu `"holla"` wird.

Eine letztlich durchaus andere Wirkung hat die ebenfalls zulässige Anfangswertzuweisung

```
char *str5 = "hallo";
```

Hier bezeichnet `str5` nicht mehr den Vektor, in dem der String gespeichert wird, sondern eine Variable, deren Anfangswert der Zeiger auf den Anfang des Vektors ist.

Während die Wertzuweisung

```
str1 = str5;
```

unzulässig ist, weil der Name eines Feldes eine Zeigerkonstante ist und entsprechend *nicht* verändert werden kann, darf man

```
str5 = str1;
```

schreiben – der Wert einer Zeigervariablen darf natürlich jederzeit verändert werden. Ob das allerdings vernünftig ist, ist eine andere Frage: Wenn wir im Beispiel den Wert von `str5` verändern, ohne dass wir den Anfangswert zuvor in eine andere Zeigervariable umspeichern, haben wir in der Folge *keinen Zugriff* mehr auf den String (auch wenn er natürlich unverändert im Speicher steht)! Solche Zuweisungen sind also, obwohl formal zulässig, logisch ein Fehler, von seltenen Ausnahmen vielleicht abgesehen.

Zweckmäßig ist es also, solche Zeigervariablen mit dem Attribut `const` zu versehen:

```
char *const str5 = "hallo";
```

Jetzt darf der Wert von `str5` im Programm nicht mehr verändert werden.

Entsprechend kann und sollte man `const` verwenden, wenn die Strings selbst unveränderbare Konstanten sein sollen. Typische Formulierungen sind zum Beispiel

```
const char str3[] = "hallo";
const char *const str5 = "hallo";
```

Alles hier gesagte, gilt für Vektoren und Zeiger mit anderen Typen natürlich in gleicher Weise.

Eines müssen wir an dieser Stelle aber noch klären: Was bedeutet bzw. bewirkt eine Wertzuweisung wie

```
s = "hallo";
```

C kennt keine Wertzuweisung, bei der der Wert eines String (oder allgemeiner: der Wert eines Feldes) übertragen wird! Entsprechend muss der String hier faktisch als Zeiger auf den Anfang der Zeichenfolge interpretiert werden, wobei die Zeichenfolge selbst vom Compiler irgendwo im Speicher abgelegt wird. Die Variable `s` *muss* damit den Typ `char *` besitzen!

12.11 Zeigervektoren

Vergleicht man die letzten beiden Anfangswert–Beispiele

```
const char str3[] = "hallo";
const char *const str5 = "hallo";
```

genauer, so stellt man fest, dass die erste Formulierung vorzuziehen ist: `str3` und `str5` repräsentieren zwar letztlich beide konstante Zeiger auf konstante Strings, die Behandlung ist jedoch durchaus verschieden:

- `str3` ist formal eine Konstante: Sie besitzt keinen Speicherplatz, sondern zeigt selbst auf den Anfang des konstanten String.
- `str5` ist eine Variable: Sie besitzt einen Speicherplatz, so dass erst ihre Dereferenzierung den Zeiger auf den Anfang des konstanten Strings liefert.

In bestimmten Zusammenhängen können Zeigervariablen mit konstanten Werten durchaus sinnvoll sein. Wir wollen uns das an einem etwas ausführlicheren Beispiel ansehen: Auf Zahlungsanweisungen und anderen Belegen müssen Beträge in Buchstaben wiederholt werden. Man behilft sich gelegentlich damit, die Ziffern einzeln in Buchstaben umzusetzen und irgendein Trennzeichen dazwischen zu setzen, so dass etwa 138 die Zeichenfolge

```
eins = drei = acht
```

ergibt.

Die Buchstabenfolgen wird man in einem Feld speichern. Wir können etwa definieren

```
const char ziffern[][7] = {
  "null", "eins", "zwei", "drei", "vier",
  "fuenf", "sechs", "sieben", "acht", "neun"
};
```

Dabei „verschwendet“ man allerdings Speicherplatz: Jede Zeile der Matrix kann 7 Zeichen aufnehmen; benötigt werden diese 7 Zeichen aber nur für die `sieben`, während für 7 Buchstabenfolgen jeweils 5 Zeichen ausreichen. Hier spielt das keine große Rolle – dass wir 70 Bytes für die Matrix bereitstellen, obwohl wir nur 54 Bytes wirklich brauchen, macht nichts aus. Aber es geht um das Prinzip: Bei einer Folge von 100 Strings, von denen einer 99 Zeichen, die übrigen dagegen nur jeweils 4 Zeichen lang sind, macht es schon einen Unterschied, ob wir eine (100×100)–Matrix definieren und damit 10000 Bytes belegen, oder ob wir nur die benötigten 595 Bytes für die Strings verwenden.

Prinzipiell kann man das Problem lösen, indem man einen Vektor von Zeigern auf Strings definiert:

```
const char *const ziffern[] = {
  "null", "eins", "zwei", "drei", "vier",
  "fuenf", "sechs", "sieben", "acht", "neun"
};
```

Jetzt ist jede Komponente des Vektors ein Zeiger auf einen String; die Strings selbst speichert der Compiler irgendwo – mit genau der Länge, die sie einschließlich des abschließenden Null–Zeichens haben. Es soll nicht verschwiegen werden, dass diese Lösung im konkreten Beispiel aufwendiger ist als die erste Formulierung: Bei den Strings sparen wir zwar 16 Bytes ein – dafür definieren wir den Zeigervektor zusätzlich, der, je nach Rechner, in der Regel 20 oder 40 Bytes benötigen wird!

Nach diesen Vorüberlegungen zu den Daten erweist sich die Realisierung als einfach: Eine rekursive Funktion erledigt die eigentliche Arbeit. Sie wird in einen nichtrekursiven Rahmen eingebettet, weil der Trennstring nur zwischen je zwei Ziffern geschrieben werden darf, der dafür erforderliche zusätzliche Parameter für den Benutzer der Funktion aber „versteckt“ werden soll:

```
void umsetzen (int wert) {
  um_rek (wert, FALSE);
}

void um_rek (int wert, int trennen) {
  if (wert > 10)
    um_rek (wert / 10, TRUE);
  printf ("%s", ziffern[wert % 10]);
  if (trennen)
    printf (" = ");
}
```

12.12 Zeiger auf Zeiger

Mit den Daten des letzten Beispiels haben wir eine weitere Möglichkeit kennengelernt, nämlich Zeiger auf Zeiger: Der Name des Vektors ist der Zeiger auf den Anfang einer Folge von Variablen, die ihrerseits Zeiger auf Strings sind. Auch solche Variablen kann man explizit definieren: Durch

```
char **tabelle;
```

ist `tabelle` ein Zeiger auf eine Variable mit dem Typ `char *`, also Zeiger auf eine Variable mit einem Zeigertyp.

Dass Zeiger auf Zeiger keine theoretische Spielerei sind, kann durch eine kleine Erweiterung des Beispiels „Zahlen in Klarschrift“ gezeigt werden: Die Klarschrift soll wahlweise in Deutsch oder Englisch (oder einer beliebigen anderen Sprache) erfolgen. Die Entscheidung darüber soll erst bei der Ausführung des Programms getroffen werden.

Im wesentlichen müssen wir die Datendefinition erweitern:

```
const char *const ziffern_d[] = {
  "null", "eins", "zwei", "drei", "vier",
  "fuenf", "sechs", "sieben", "acht", "neun"
};
const char *const ziffern_e[] = {
  "zero", "one", "two", "three", "four",
  "five", "six", "seven", "eight", "nine"
};
const char *const *ziffern;
```

Im Programm brauchen wir jetzt nur noch dafür zu sorgen, dass `ziffern` der „richtige“ Zeiger zugewiesen wird, zum Beispiel durch

```
if (...)
   ziffern = ziffern_d;
else
   ziffern = ziffern_e;
```

Generell entsprechen mehrdimensionale Felder Zeigern auf Zeiger, wie bereits kurz angesprochen. Haben wir zum Beispiel die Definition

```
int m[2][3];
```

so ist `m` Zeiger auf den Vektor der Zeilenvektoren, während `m[i]`, einen „zulässigen“ Wert von `i` vorausgesetzt, der Zeiger auf das erste Element in der $(\texttt{i}+1)$–ten Zeile der Matrix ist. Damit haben wir, zulässige Werte der Indizes vorausgesetzt, vier Möglichkeiten, auf ein bestimmtes Element zuzugreifen, nämlich

```
m[i][j]
(*(m + i))[j]
*(m[i] + j)
*(*(m + i) + j)
```

12.13 Zeiger als Funktionswerte

Auch wenn es selbstverständlich sein sollte: Funktionen können als Funktionswert einen Zeiger liefern.

Speziell bei Funktionen zur Verarbeitung von Strings ist der Funktionswert häufig ein Zeiger auf einen (Teil–)String. Das ist auch der Grund, warum auf die Funktionen bislang nicht näher eingegangen wurde, die in der Standard–Headerdatei `string.h` deklariert sind. Deshalb jetzt ein kleiner Nachtrag zu dieser Header–Datei.

Zum Kopieren von Strings stehen die beiden Funktionen

```
char *strcpy(char *s1, const char *s2);
char *strncpy(char *s1, const char *s2, size_t n);
```

zur Verfügung: `strcpy` kopiert so lange, bis das Null–Zeichen von `s2` übertragen wurde. `strncpy` schreibt dagegen genau `n` Zeichen, wobei ggf. am Ende Nullen ergänzt werden. Der Funktionswert ist der Zeiger auf `s1`. Bei beiden Funktionen dürfen sich Quellstring `s2` und Zielstring `s1` nicht überschneiden; dass der Zielstring genug Platz für den Quellstring bietet, liegt ausschließlich in der Verantwortung des Programmierers.

Der Typ `size_t` ist übrigens ein vorzeichenloser ganzzahliger Typ, der in `stddef.h`, `stdlib.h` und `string.h` durch `typedef` deklariert ist.

In analoger Weise kopieren die Funktionen

```
char *strcat(char *s1, const char *s2);
char *strncat(char *s1, const char *s2, size_t n);
```

den String `s2` hinter den String `s1`. Man bezeichnet das als *Konkatenation*. Dass der Zielstring genug Platz bietet, liegt ausschließlich in der Verantwortung des Programmierers.

Ebenfalls analog erlauben die Funktionen

```
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
```

den Vergleich von zwei Strings. Der Funktionswert ist kleiner, gleich oder größer als Null, je nachdem, ob `s1` kleiner, gleich oder größer als `s2` ist. Beim Vergleich werden die Ordnungszahlen der Zeichen verwendet, so dass Groß– und Kleinbuchstaben als verschieden betrachtet werden.

Verschiedene, teils ziemlich komplizierte Funktionen erlauben das Suchen von Zeichen oder Strings in Strings. Die wichtigsten dieser Funktionen sind:

```
char *strchr(const char *s, int c);
char *strrchr(const char *s, int c);
char *strstr(const char *s1, const char *s2);
```

Die ersten beiden Funktionen suchen das Zeichen `c` im String `str` und liefern als Funktionswert den Zeiger auf die erste gefundene Position oder den Nullzeiger, wenn `c` in `s` nicht vorkommt. Während `strchr` die Suche am Anfang von `s` startet, startet `strrchr` am Ende von `s`. `strstr` sucht die Zeichenfolge von `s2` in `s1`. Der Funktionswert ist der Zeiger auf das erste Zeichen der gefundenen Teilfolge bzw. der Nullzeiger.

Schließlich gibt es die von uns bereits betrachtete Funktion

```
size_t strlen(const char *s);
```

zur Bestimmung der aktuellen Länge des String `s`.

Als Beispiel soll einmal die Funktion `strchr` implementiert werden:

```
char *strchr(const char *s, char c) {
  while (*s != c)
    if (*s++ == '\0')
      return NULL;
  return s;
}
```

Hierbei ist anzumerken, dass der Compiler bei der `return`–Anweisung warnen sollte, dass das `const`–Attribut verlorengeht. Als Rückgabewert erhält man ja einen Zeiger auf *nicht*–konstante Werte. Folgende zwei Zeilen sollten ohne Warnung übersetzt werden können:

```
char *p = strchr("Konstanter Strung", 'u');
*p = 'i';
```

Das Ergebnis bei der Ausführung ist nicht definiert. Beim Argument von `strchr` handelt es sich schließlich um eine Zeichenketten*konstante*, auf die auch der zurückgegebene Zeiger verweist.

12.14 Parameter des Hauptprogramms

Jedes C–Programm muss, wie wir wissen, mit der Funktion `main` ein Hauptprogramm besitzen. Diese Hauptfunktion wird beim Start des Programms vom Betriebssystem aufgerufen. Und sie erhält beim Aufruf zwei Argumente! Wir haben diese Argumente bislang durch die Definition

```
int main (void)
```

ignoriert. Tatsächlich ist der Prototyp für `main` jedoch

```
int main (int argc, char *argv[]);
```

Was man übergeben erhält, ist der bereits zerlegte Inhalt der Aufrufzeile des Programms:

- `argc` ist die Anzahl der Einträge in der Aufrufzeile, wobei als Trennzeichen zwischen den Einträgen Leerzeichen und horizontaler Tabulator interpretiert werden.
- `argv` ist ein Vektor von Zeigern auf die einzelnen Einträge in der Form von C–Strings.

Der String, auf den `argv[0]` zeigt, ist der Name des Programms oder ein leerer String, wenn das Betriebssystem den Namen des Programms nicht liefert. Die nächsten `argc - 1` Komponenten zeigen auf die weiteren Einträge der Aufrufzeile, in der Reihenfolge, in der sie in der Aufrufzeile angegeben sind. Dem letzten „echten“ Zeiger folgt noch eine Komponente mit dem Nullzeiger (`argv[argc]`), so dass man sich wahlweise an diesem Nullzeiger oder an dem Wert von `argc` orientieren kann.

Nicht übergeben werden dabei alle Einträge der Aufrufzeile, die sich auf die Umleitung der Ein– und Ausgabe oder auf Filter beziehen – das sollte aber auch klar sein, da dieses vom Betriebssystem bearbeitet wird und für das Programm transparent sein soll.

Was ein Programm mit seinen Parametern anfängt, ist ihm selbst überlassen. Es kann sie ignorieren, wie wir es bislang getan haben, oder aber auch darauf reagieren.

Wir betrachten zunächst ein eher formales Beispiel: Ein Programm soll die Einträge seiner Aufrufzeile auflisten. Da der Nullzeiger hier erstmals für uns Bedeutung gewinnt, wird er hier verwendet, um das Ende der Liste der Einträge abzufangen. Die Realisierung kann dann so aussehen:

```
int main(int argc, char *argv[]) {
  while (*++argv != NULL)
    printf("%s\n", *argv);
  return 0;
}
```

Ein realistischeres Beispiel könnte so aussehen: Wir hatten die Umsetzung von Zahlen in Klarschrift programmiert, in der letzten Version mit der Möglichkeit, die Klarschrift auf deutsch oder englisch zu erhalten. Es liegt nahe, die gewünschte Sprache nicht abzufragen, sondern als Parameter in der Aufrufzeile zu verlangen, zum Beispiel `-Sd` für deutsch und `-Se` für englisch. Man kann dann weiter festlegen, dass bei fehlendem Parameter `-S`*x* wie bei `-Sd` verfahren wird und dass bei `-S`*x* mit einem anderen Zeichen als `d` und `e` das Programm mit einer Fehlermeldung abgebrochen wird. Ebenso sollen mehr als 2 Parameter zum Programmabbruch führen.

Die Realisierung des Hauptprogramms kann dann so aussehen:

```
#include <string.h>

enum sprachen {DEUTSCH, ENGLISCH};

int main(int argc, char *argv[]) {
  enum sprachen sprache = DEUTSCH;

  if (argc > 2) {
    printf ("Zu viele Parameter!\n");
    return 1;
  }

  if (argc == 2) {
    if (strcmp(argv[1], "-Se") == 0) {
      printf("Sprache Englisch gewaehlt.\n");
      sprache = ENGLISCH;
    }
    else if (strcmp(argv[1], "-Sd") == 0) {
      printf("Sprache Deutsch gewaehlt.\n");
      sprache = DEUTSCH;
    }
    else {
```

```
        printf("Unzulaessiger Parameter.\n");
        return 2;
      }
    }
    ...
    return 0;
  }
```

12.15 Zeiger auf Funktionen

C erlaubt nicht nur Zeiger auf Variablen, sondern ebenso Zeiger auf Funktionen – auch *Funktionszeiger* genannt.

Zeiger auf Funktionen benötigt man zum Beispiel dann, wenn man Funktionen als Parameter an Funktionen übergeben will. Das ist nichts „Exotisches“:

- In der Mathematik hat man sehr häufig Algorithmen, die für ganze Funktionenklassen definiert sind. Ein „klassisches“ Beipiel hierfür ist die numerische Integration; eine Integrationsfunktion soll ja für beliebige integrierbare Funktionen verwendet werden können.
- Häufig hat man bei allgemein formulierten Sortier– und Suchfunktionen Funktionen als Parameter. Führt man zum Beispiel Vergleiche nicht explizit aus, sondern sieht eine Vergleichsfunktion vor, so kann man ohne Änderung an der eigentlichen Sortierfunktion nach beliebigen Kriterien sortieren.

Sehen wir uns aber zunächst die Formalien an: Die Definition eines Zeigers auf eine Funktion ähnelt der Deklaration einer Funktion:

funktionstyp `(*`*name*`)(`*parametertyp*`, ..., `*parametertyp*`);`

So ist durch

```
int (*fptr)(char);
```

die Variable `fptr` ein Zeiger auf eine Funktion, die einen `char`–Parameter besitzt und einen `int`–Funktionswert liefert. Die Klammern um `*fptr` sind hier zwingend notwendig, denn durch

```
int *fptr (char);
```

würde deklariert, dass `fptr` selbst eine Funktion ist, die einen `char`–Parameter besitzt und als Funktionswert einen Zeiger auf `int` liefert! Die Deklaration von Funktionszeigern verlangt einem also einiges an Schreibarbeit ab – besonders wenn man mehrere von ihnen braucht. Hier kommt das Schlüsselwort `typedef` gelegen. Dazu betrachten wir folgendes Beispiel:

```
#include <stdio.h>

typedef double (*fptr_t)(double, double);

double summe(double a, double b) {
  return a + b;
}

double produkt(double a, double b) {
  return a * b;
}
```

```
/*= Hauptprogramm ========================================*/
int main (void) {
  /* Feld von Funktionszeigern */
  fptr_t vfptr[2] = { summe, produkt };
  unsigned int op = 0;
  double a = 0.0, b = 0.0;

  printf("Eingabe: Operator Argument1 Argument2\n"
         "Operatoren: 0 = Summe, 1 = Produkt\n");
  scanf("%u %lf %lf", &op, &a, &b);

  printf("Ergebnis: %f\n", vfptr[op](a, b));

  return 0;
}
```

Quelltext 12.1: Verwendung von Zeigern auf Funktionen und `typedef`

Durch die Typ–Deklaration von `fptr_t` spart man bei der weiteren Verwendung das wiederholte Ausschreiben des Funktionszeiger–Typs. Selbst die Deklaration eines Feldes von Zeigern auf Funktionen ist dann nicht schwieriger als z.B. beim Typ `int`. Als Ausgabe des Programms erhalten wir

```
Eingabe: Operator Argument1 Argument2
Operatoren: 0 = Summe, 1 = Produkt
1
42
23
Ergebnis: 966.000000
```

Bemerkenswert an diesem kleinen Beispiel ist, dass die Auswahl der gewünschten Operation ohne `if`-Anweisung oder ähnliches auskommt. Wie für Felder ist auch für Funktionen der Adressoperator nicht erforderlich: Wenn eine Funktion aufgerufen werden soll, muss dem Namen die Argumentliste folgen, also mindestens ein leeres Klammernpaar. Steht der Name einer Funktion ohne Argumentliste im Quellcode, dann ist der Zeiger auf sie gemeint.

Als konkretes Beispiel für Funktionszeiger als Argument soll die Funktion `qsort` dienen, die in der Standard–Headerdatei `stdlib.h` deklariert ist. Ihr Prototyp ist

```
void qsort(void *start, size_t anzahl, size_t groesse,
           int (*relation)(const void *, const void *) );
```

Diese Funktion sortiert grundsätzlich aufsteigend – nur kann und muss ihr Anwender mit einer eigenen Relationsfunktion festlegen, was „aufsteigend“ bedeutet. Daneben unterstellt die Funktion, dass der Vektor zu sortieren ist, den die ersten drei Argumente beschreiben:

- Das erste Argument ist der Zeiger auf die erste Komponente des Vektors.
- Das zweite Argument ist die Anzahl der Komponenten des Vektors.
- Das dritte Argument ist die Größe der Komponenten des Vektors.

Diese Formulierung, insbesondere die Verwendung von `void *`, erlaubt es, beliebige Vektoren als Argumente zu übergeben. Man beachte dabei: Variablen, Parameter und Funktionswerte dürfen zwar den Typ `void *` erhalten; dereferenzieren kann man solche Werte allerdings erst, nachdem man sie mit einem Castoperator in einen typgebundenen Zeiger

umgewandelt hat. Dass man dabei nur „sinnvolle“ Umwandlungen vornehmen darf, sollte klar sein, kann jedoch vom Compiler nicht überprüft werden.

Der Name der Funktion legt nahe, dass sie Quicksort realisiert – aber das ist nicht vorgeschrieben.

Jetzt zum Beispiel: Wir wollen `short`–Zahlen auf– oder absteigend sortieren. Dann können wir schreiben

```
#include <stdlib.h>

#define LAENGE ...

typedef int (*REL)(const void *, const void *);

int aufsteigend(const short *const links,
                const short *const rechts);
int absteigend(const short *const links,
               const short *const rechts);

int main(void) {
  short v[LAENGE];
  int (*rel)(const short *const,
             const short *const);
  ...
  if (...)
    rel = aufsteigend;
  else
    rel = absteigend;
  ...
  qsort((void *) v, (size_t) LAENGE,
        sizeof (short), (REL) rel);
  ...
  return 0;
}

int aufsteigend(const short *const links,
                const short *const rechts) {
  return *links - *rechts;
}

int absteigend(const short *const links,
               const short *const rechts) {
  return *rechts - *links;
}
```

Damit sollte das Prinzip klar sein, auch für Vektoren mit anderen Typen. Der Castoperator für die Vergleichsfunktion mag auf den ersten Blick ziemlich „exotisch“ erscheinen, aber er ist ohne weiteres zulässig und wandelt den Typ von `rel` entsprechend dem Prototyp von `qsort` um.

Man kann sich nun leicht überlegen, dass man selbst die „merkwürdigsten“ Sortierreihenfolgen leicht realisieren kann, zum Beispiel aufsteigend nach Beträgen oder erst alle geraden Werte aufsteigend und dann alle ungeraden Werte absteigend. Aber das überlegen Sie sich in den Details vielleicht selbst einmal.

12.16 Kopieren und umspeichern von Speicherblöcken

Da die Funktion `qsort` mit `void`–Zeigern arbeitet, besitzt sie keine Information über den Typ der zu sortierenden Daten. Die Zeiger können also auch nicht dereferenziert und die Elemente durch den Zuweisungsoperator vertauscht werden. Stattdessen werden sie einfach als binäre Daten kopiert. Diese Aufgabe übernehmen auch die folgende beiden Funktionen aus der Standardbibliothek.

```
void *memcpy(void *s1, const void *s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
```

Beide kopieren `n` aufeinanderfolgende Zeichen, auf deren erstes `s2` zeigt, in den Speicherbereich, auf dessen Anfang `s2` zeigt.

Die Intention dieser beiden Funktionen ist letztlich jedoch nicht, wie `strcpy` und `strncpy` Zeichen zu kopieren, sondern beliebige zusammenhängende Speicherbereiche, auch *Speicherblöcke* genannt. Diese werden nur temporär als Zeichenvektoren betrachtet. Entsprechend werden die Werte der „Zeichen“ auch nicht untersucht, so dass ein eventuelles Null–Zeichen das Kopieren *nicht* beendet.

Unterschiede bestehen zwischen den beiden Funktionen, wenn Speicherbereiche kopiert werden sollen, die sich überlappen. Während `memmove` auch in solch einem Fall korrekt arbeitet, ist das bei `memcpy` nicht garantiert.

Beispiel:

```
#define LAENGE ???
...
double v1[LAENGE], v2[LAENGE];

/* korrekt:  */
memcpy(v1, v2, sizeof v2);
memcpy(v1, v2, LAENGE * sizeof (double));
/* fraglich: */
memcpy(v1, v1+1, (LAENGE - 1) * sizeof (double));

/* korrekt:  */
memmove(v1, v2, sizeof v2);
memmove(v1, v2, LAENGE * sizeof (double));
memmove(v1, v1+1, (LAENGE - 1) * sizeof (double));
```

Mit der gleichen Logik arbeitet die Funktion

```
void memset(void *v, int c, size_t n);
```

Sie schreibt den Wert von `c`, umgewandelt in den Typ `char`, `n`–mal in den Vektor, auf dessen Anfang `v` zeigt.

Kapitel 13

Speicher auf dem Heap

13.1 Speichertypen

Wir haben bisher zwei logische Arten von Speicher kennengelernt:

- Externe Variablen sind grundsätzlich statisch, ebenso interne Variablen mit dem Attribut `static`. Das hieß, dass bereits der Compiler den erforderlichen Speicherplatz bereitstellt, der dann während der gesamten Ausführung des Programms zur Verfügung steht.
- Automatische Variablen sind dynamisch. Das hieß, dass der Speicher für sie erst in dem Moment bereitgestellt wird, in dem bei der Ausführung des Programms der Block betreten wird, der ihre Definition enthält, und dass dieser Speicher beim Verlassen des Blocks auch wieder freigegeben wird. Diese Art von Variablen wird in der Regel auf einem *Stack* bereitgestellt. Dabei handelt es sich um eine Datenstruktur, die linear wächst. D.h. Speicherplatz einer Variablen auf dem Stack kann erst wieder freigegeben werden, wenn alle später erzeugten Variablen ebenfalls wieder freigegeben wurden. Das entspricht genau der Verwendung von automatischen Variablen. Es handelt sich also um strukturierten dynamischen Speicher.

Es gibt eine dritte logische Art von Speicher: Ein Programm hat die Möglichkeit, während es läuft Speicher vom Betriebssystem anzufordern, wobei sich die Größe des angeforderten Speichers erst bei Ablauf des Programms z.B. durch eine Benutzereingabe ergibt. Dieser Speicher steht ihm dann so lange zur Verfügung, bis es ihn explizit wieder freigibt oder bis es terminiert. Die Verfügbarkeit dieses Speichers ist insbesondere nicht an die Blockstruktur des Programms gebunden. Man bezeichnet die Gesamtheit dieses Speichers als *Heap*. Da dieser Speicher in beliebiger Reihenfolge angefordert und wieder freigegeben werden kann, ist klar, dass er nicht besonders strukturiert sein kann. Die Funktionen zur Anforderung von Speicher auf dem Heap müssen also einen gewissen Aufwand betreiben, um den Speicher zu verwalten. Der von automatischen Variablen genutzte Speicher auf dem Stack erfordert fast keinen zusätzlichen Aufwand.

Wieviel Speicher auf dem Heap zur Verfügung steht, hängt vom jeweiligen Rechner und Betriebssystem ab. Die Größe des Heap kann durch den physikalisch vorhandenen Speicher begrenzt sein. Die meisten Betriebssystem unterstützen jedoch virtuellen Speicher. D.h. jedem Programm steht der gesamte Adressraum zur Verfügung. Sollte mehr als der tatsächlich vorhandene Speicher benötigt werden, so werden gerade nicht benötigte Speicherbereiche auf die Festplatte ausgelagert. Belegt ein Zeiger 32 Bit, so sind theoretisch maximal 4 Gigabyte adressierbar.

Bei statischem Speicher und bei Speicher auf dem Stack sorgt der Compiler dafür, dass eine Zuordnung zwischen Variablen und Adressen erfolgt. Bei Speicher auf dem Heap sieht das ganz anders aus: Die Anforderung von Speicher vom Betriebssystem erfolgt natürlich durch Aufruf einer entsprechenden Funktion. Die kann – als einzige Möglichkeit – einen Zeiger auf den Anfang des bereitgestellten Speichers als Funktionswert zurückgeben , falls genug Speicher vorhanden ist. Man ist also gezwungen, diesen Wert in einer „passenden“ Zeigervariablen zu speichern, damit man ihn später für die Zugriffe auf den Speicher verwenden kann.

13.2 Funktionen zur Heapverwaltung

C kennt drei Funktionen, mit denen man Speicher auf dem Heap anfordern kann, und eine Funktion zur Freigabe. Alle vier Funktionen sind in der Standard–Headerdatei `stdlib.h` deklariert. Wichtig sind zunächst nur die Funktionen

```
void *malloc(size_t groesse);
void free(void *zeiger);
```

Die Funktion `malloc` stellt, wenn möglich, einen zusammenhängenden Speicherbereich der Größe `groesse` zur Verfügung und liefert als Funktionswert den Zeiger auf den Anfang dieses Speicherbereichs. Die Größe bezieht sich dabei auf ein Vielfaches des Speicherbedarfs eines Zeichens – wir erinnern uns: Der Standard schreibt `sizeof (char) == 1` vor. Falls der angeforderte Speicher nicht bereitgestellt werden kann, ist der Funktionswert der Nullzeiger (`NULL`). Da die Speicherbereitstellung bei jedem Versuch fehlschlagen kann, sollte man den Rückgabewert von `malloc` *immer* auf `NULL` überprüfen.

Der Typ dieser Funktion ist `void *`. Ein Zeiger dieses Typs ist kompatibel zu allen anderen Zeigertypen, so dass man seinen Wert jeder beliebigen Zeigervariablen zuweisen kann, unabhängig von deren Typ und ohne einen Castoperator hinschreiben zu müssen. Dass man Zeiger mit dem Typ `void *` selbst nicht dereferenziert werden kann, haben wir bereits gesehen.

Die Funktion `free` gibt einen Speicherbereich wieder frei. Voraussetzung für ordnungsgemäßes Arbeiten der Funktion ist, dass man ihr nur Zeiger als Argument übergibt, die man sich zuvor mit `malloc` oder einer der beiden anderen Bereitstellungsfunktionen beschafft hat. Dahinter steht, dass zumindest manche Betriebssysteme „in der Nähe“ der Zeiger, die sie beim Aufruf von `malloc` liefern, zusätzliche interne Informationen unterbringen, die beim Aufruf von `free` benötigt werden, etwa Angaben über die Größe des Speichers. Einzige Ausnahme ist `NULL`. Wird `NULL` als Argument von `free` verwendet, tut die Funktion nichts.

13.3 Bereitstellung von Speicher

Bei der Matrizenmultiplikation, oder auch vielen anderen mathematischen Programmen, die mit Vektoren und Matrizen arbeiten, weiß man in der Regel beim Schreiben des Programms nicht, wie groß die Vektoren und Matrizen maximal werden können. Dies macht die Verwendung von statischen Variablen problematisch.

Dieses Problem wird in C durch die Bereitstellung der benötigten Vektoren und Matrizen auf dem Heap gelöst. Durch

```
float *v;
int laenge;
/* ... Initialisierung von laenge              */
v = (float *) malloc(laenge * sizeof(float));
```

```
if (v == NULL) {
  /* Fehlerbehandlung, ggf. Programmabbruch   */
}
```

stellt das Betriebssystem ja gerade den Speicherplatz bereit, den ein `float`–Vektor mit `laenge` Komponenten benötigt. Entsprechend kann man sich mit

```
float *m;
int n;
/* ... Initialisierung von n                  */
m = (float *) malloc(n * n * sizeof(float));
if (m == NULL) {
  /* Fehlerbehandlung, ggf. Programmabbruch   */
}
```

den Speicherplatz für eine $(n \times n)$–Matrix mit `float`–Komponenten beschaffen.

Zeigervariablen erlauben es immer, ihren Wert als Zeiger auf den Anfang eines Vektors zu interpretieren. Das haben wir hier für den Speicher auf den Heap genutzt. Ohne weiters ist es dagegen nicht möglich, Speicher auf dem Heap mit einer Matrixstruktur zu versehen. Wir sind also, wie in der Regel auch in Funktionen, dazu gezwungen, die *Speicherabbildungsfunktion* selber zu definieren und sie für jeden Zugriff selber auszuwerten. Die entsprechende Vorgehensweise wurde schon im Abschnitt über Funktionen besprochen.

Am Ende dieses Kapitels wird noch einmal auf Matrizen auf dem Heap eingegangen. Wir werden sehen, dass sich, wenn auch mit einigem Aufwand, letztlich doch auch Matrizen erzeugen lassen.

Speicher, den man sich mit `malloc` beschafft, besitzt übrigens keine wohldefinierten Werte. Will man bestimmte Anfangswerte haben, muss man selber explizit für deren Zuweisung sorgen.

13.4 Beispiel: Speichern von Zahlen

Es wird Zeit für ein konkretes Beispiel. Wir wollen nur die letzten Zahlen einer Eingabe verarbeiten, zum Beispiel die letzten 10. Allerdings soll die Anzahl letztlich erst beim Aufruf des Programms festgelegt werden: Wenn ein entsprechender Aufrufparameter (in der Form `-`*nnn*) angegeben ist, soll sein Wert verwendet werden; sonst soll 10 verwendet werden.

Um das Programm nicht ausufern zu lassen, werden zwei Vereinfachungen gegenüber der „Soll–Lösung“ vorgenommen:

- Unzulässige Aufrufparameter sollen nicht besonders bemängelt, sondern nur durch den Standardwert 10 ersetzt werden.
- Die Verarbeitung soll nur im Auflisten der Zahlen bestehen.

Die Lösung verwendet die Funktion `sscanf`, die in `string.h` deklariert wird. Sie arbeitet genau wie `scanf`, jedoch liest sie nicht von der Standardeingabe, sondern verarbeitet als Eingabe den String, auf den das erste Argument zeigt.

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

#define DEFLEN 10
```

```
/**  Prototypen  ****************************************/
static int lesen(unsigned int maxlen, float *v);

/*= Hauptprogramm ========================================*/
int main(int anzahl, char *argument[]) {
  unsigned int maxlen, len, i;
  float *v;

  if (anzahl < 2 || sscanf(argument[1], "-%u", &maxlen) != 1)
    maxlen = DEFLEN;

  v = (float *) malloc(maxlen * sizeof (float));
  if (v == NULL) {
    printf("Speicher nicht verfuegbar!\n");
    return 1;
  }

  printf("Bitte Zahlen eingeben: ");
  len = lesen (maxlen, v);

  for (i = 0; i < len; i++)
    printf("%f\n", v[i]);

  free(v);
  return 0;
}

/*= Lesen der Eingabe ====================================*/
static int lesen(unsigned int maxlen, float *v) {
  int i = 0;
  float z;

  while (i < maxlen) {
    if (scanf("%f", &v[i]) == EOF)
      return i;
    i++;
  }

  while (scanf("%f", &z) != EOF) {
    memmove(&v[0], &v[1], (maxlen - 1) * sizeof (float));
    v[maxlen - 1] = z;
  }

  return maxlen;
}
```

Quelltext 13.1: Auflisten der letzten Zahlen der Eingabe

Eine Frage stellt sich noch: Ist es eigentlich zweckmäßig, die gespeicherten Werte umzuspeichern? Lässt sich dieses ggf. vermeiden?

Zur ersten Teil der Frage: Ganz ohne Umspeicherungen kommen wir nicht aus. Da `float`–Werte in der Regel nur 4 Bytes beanspruchen, spricht im Beispiel wenig dagegen, sie selbst umzuspeichern – `short`– und `char`–Werte benötigen zwar in der Regel weniger Speicherplatz, aber diese Typen helfen hier nicht weiter.

Anders sieht es aus, wenn die Werte mehr Speicherplatz benötigen – etwa wenn sie Strings

oder, allgemeiner, Vektoren sind. Das führt zum zweiten Teil der Frage: Natürlich brauchen wir die Werte selbst nicht umzuspeichern! Um das Umspeichern zu vermeiden, müssen wir allerdings die Datenstruktur ändern: Anstelle eines Vektors, der selbst die Daten enthält, müssen wir einen Vektor definieren, der Zeiger auf die Daten enthält. Dann reicht es aus, die Zeiger umzuspeichern – und die belegen in der Regel jeweils 2 oder 4 Bytes, je nach Rechner.

Realisiert man diese Idee, so hat man zwei Schritte auszuführen: Im ersten muss man den Zeigervektor bereitstellen und im zweiten die Zeiger auf die Daten. Und dabei kann man sich geschickt oder ungeschickt anstellen: Ungeschickt wäre es, in einer Schleife für jede Vektorkomponente separat den Zeiger auf einen passenden Speicherbereich zu beschaffen, weil `malloc` in der Regel eine ziemlich aufwendige Funktion ist. Es reicht durchaus aus, einen einzigen, zusammenhängenden Speicherbereich zu beschaffen und dann Zeiger auf Teile dieses Speicherbereichs zu berechnen. Das können wir, wenn wir beim Werttyp `float` bleiben, zum Beispiel so lösen:

```
float **v, *z;
int j;
...
v = (float **) malloc(maxlen * sizeof(float *));
if (v != NULL) {
  z = (float *) malloc(maxlen * sizeof(float));
  if (z != NULL) {
    for (j = 0; j < maxlen; j++)
      v[j] = &z[j];
  }
  else {
    free(v);
    v = NULL;
  }
}
```

Jetzt zeigt die `j`–te Komponente von `v` anfangs auf die `j`–te Komponente von `z` – doch das ändert sich unter Umständen während der Ausführung des Programms.

In der Folge sind weitere Änderungen am Programm erforderlich. Sie sollten sie sich selbst einmal überlegen.

13.5 Speicherfreigabe

Am Beginn des Abschnitts wurde erwähnt, dass Speicher auf dem Heap dem Programm so lange zur Verfügung steht, bis es ihn wieder freigibt. Das stimmt, leider, nur bei formaler Betrachtung: Der Speicher steht zwar formal zur Verfügung – um ihn nutzen zu können, braucht man zusätzlich aber auch einen Zeiger auf ihn. Und diese Zeiger können in der Regel selbst nicht auf dem Heap liegen, sondern müssen statische oder automatische Variablen sein.

Wir sehen uns das an einem Beispiel an: Formal ist die Funktion

```
void muellerzeugung(int menge) {
   float *v;
   v = (float *) malloc(menge * sizeof(float));
}
```

sicher korrekt. Allerdings verschwindet ihre lokale, automatische Variable `v` mit Abschluss der Funktion wieder – und mit ihr auch der Zeiger auf den Speicherbereich, den wir uns mit `malloc` beschafft haben.

Mit einer Funktion wie `muellerzeugung` können wir entsprechend jedes Programm zum „Absturz“ bringen – wir brauchen sie nur oft genug mit einem hinreichend großen Argument aufzurufen: Bei jedem Aufruf wird Speicher angefordert. Da dieser Speicher nicht wieder freigegeben wird, muss der Heap notwendig irgendwann voll sein.

Es ist also ein schwerwiegender Fehler, Speicher auf dem Heap zu beschaffen, diesen aber nicht wieder freizugegeben. Wird der Speicher nur lokal in einer Funktion benötigt, so ist diese auch dafür zuständig, ihn wieder freizugeben. Das kann so aussehen:

```
void kein_muell (int menge) {
   float *v;
   v = (float *) malloc(menge * sizeof(float));
   /* ... */
   free (v);
}
```

Oft soll aber der bereitgestellte Speicher über einen längeren Zeitraum verwendet werden. Dann sollte es möglich sein, die Speicherreservierungen zu protokollieren, um während der Testphase des Programms eventuelle Speicherlecks mithilfe des Protokolls zu entdecken. Der Protokollmechanismus kann so implementiert werden, dass man ihn z.B. durch bedingte Compilation leicht abschalten kann. Damit reservierter Speicher leicht wieder freigegeben werden kann, sollte es zu jeder Funktion, die Speicher bereitstellt, eine passende Funktion geben, die ihn wieder freigibt. Besonders bei komplexen Datenstrukturen kann das sehr hilfreich sein: Man muss am Ende nur zählen, ob beide Funktionen gleichoft erfolgreich aufgerufen wurden. Im nächsten Abschnitt wird für Matrizen ein solchen Funktionenpaar vorgestellt.

Außerdem gilt natürlich erneut, was schon für Zeiger auf Konstanten gilt: Überschreibt man den Zeiger auf einen Speicherbereich im Heap durch einen anderen Wert, hat man anschließend keinen Zugriff mehr auf ihn. Insbesondere kann man ihn nicht einmal mehr freigeben!

So offensichtlich wie in der Anweisungsfolge

```
float f, *v;
v = (float *) malloc(laenge * sizeof(float));
v = &f;
```

wird das sicher kein Programmierer machen – in verdeckter Form kommt es leider allzu häufig vor!

13.6 Matrizen auf dem Heap

Jetzt soll endlich ein allgemein brauchbares Beispiel für beliebig dimensionale Matrizen implementiert werden. Wie man sich den Speicher für sie auf dem Heap beschaffen kann, haben wir oben gesehen. Wie man eine vernünftige Indizierung erreichen kann, wollen wir uns jetzt überlegen.

Dazu müssen wir uns noch einmal in Erinnerung rufen, wie für eine Matrix `m` der Ausdruck `m[i]` zu interpretieren ist. Zunächst zwei elementare formale Überlegungen:

- Paare eckiger Klammern sind Operatoren; aufeinanderfolgende Paare werden von links nach rechts abgearbeitet.
- Der Name einer Matrix repräsentiert einen Zeiger auf den Vektor der Zeiger auf die Zeilen.

Beide Überlegungen ergeben zusammen: Der Wert des Ausdrucks `m[i]` ist ein Zeiger auf eine Zeile der Matrix.

Da `m` der Name einer Matrix ist, ist der Vektor der Zeiger auf die Zeilen und entsprechend der Wert des Ausdrucks `m[i]` eine Konstante. Es steht allerdings nirgends, dass bei doppelter Indizierung, etwa `m[i][j]`, der Teilausdruck `m[i]` eine Konstante sein *muss*.

Und dieses können wir nutzen: Wir wollen einmal unterstellen, dass wir eine $n \times m$–Matrix A mit `double`–Komponenten benötigen. Dann beschaffen wir uns in einem ersten Schritt einen Vektor mit n Komponenten, die jeweils den Zeiger auf eine Zeile aufnehmen können. In einem zweiten Schritt beschaffen wir uns die Speicher für die Zeilen und tragen die Zeiger darauf in die Komponenten des zunächst beschafften Vektors ein.

Im Programm kann das so aussehen:

```
int n, m;
...
double **a;
int i;
...
a = (double **) malloc(n * sizeof (double *));
if (a == NULL)
  /* ... Fehlerbehandlung */

for (i = 0; i < n; i++) {
  a[i] = (double *) malloc(m * sizeof (double));
  if (a[i] == NULL)
    /* ... Fehlerbehandlung */
}
```

Schreiben wir jetzt etwa `a[i][j]`, so wird wieder zunächst `a[i]` ausgewertet. Dabei resultiert der Zeiger auf die entsprechende Zeile. Danach erfolgt die zweite Indizierung und liefert uns den Wert der entsprechenden Komponente der Zeile, also genau das, was wir haben wollen.

Bei der Freigabe einer so bereitgestellten Matrix muss man offensichtlich genau umgekehrt vorgehen und erst die einzelnen Zeilen und dann den Vektor der Zeiger auf die Zeilen freigeben. Das Arbeiten mit Matrizen wird dadurch sehr einfach, da man so bereitgestellte Matrizen auch in Funktionen vernünftig indizieren kann. Die Parameter, die die Matrizen repräsentieren, sind formal ja Vektoren (von Zeigern) – und für Vektoren funktioniert die Indizierung auch in Funktionen.

Der Preis, den man dafür zahlen muss, sind die relativ aufwendige Bereitstellung und Freigabe der Matrizen. Man kann ihn aber auf ein Minimum reduzieren, indem man einmal die entsprechenden Operationen als Funktionen programmiert und in einem separaten Modul unterbringt, auf den man dann bei Bedarf zurückgreifen kann.

Wenn man diese Funktionen schreibt, kann man gleich noch etwas optimieren: Die Funktionen `malloc` und `free` sind in der Regel ziemlich aufwendig, so dass man sie nicht unnötig oft aufrufen sollte. Entsprechend ist es hier in der Regel günstiger, sich den gesamten Speicherbereich für die Matrix mit einem Aufruf von `malloc` zu beschaffen und diesen Speicher dann in einer Schleife aufzuteilen, wie in einem früheren Beispiel bereits gesehen:

```
#include <stdlib.h>

static int anzahl = 0;
```

```
/*= Matrixallokation ====================================*/
double **matrixalloc(unsigned int n, unsigned int m) {
  double **a;

  a = (double **) malloc(n * sizeof (double *));
  if (a != NULL) {
    a[0] = (double *) malloc(n * m * sizeof (double));
    if (a[0] == NULL) {
      free(a);
      a = NULL;
    }
    else {
      unsigned int i;
      for (i = 1; i < n; i++)
        a[i] = a[i-1] + m;

      anzahl++;                    /* Matrix bereitgestellt */
    }
  }
  return a;
}

/*= Matrixfreigabe ======================================*/
void matrixfree(double **a) {
  if (a != NULL)
    free(a[0]);

  free(a);
  anzahl--;                        /* Matrix freigegeben    */
}

/*= Matrix-Zaehler ======================================*/
int matrixanzahl(void) {
  return anzahl;              /* falls matrixanzahl() != 0, */
}                             /*   wurden nicht alle        */
                              /*   Matrizen freigegeben     */
```

Quelltext 13.2: Matrizen mit beliebigen Dimensionen bereitstellen

Beachte: Zeilenvertauschungen durch Vertauschen von Zeigern sind jetzt ohne weiteres nicht mehr möglich. Zum Freigeben der Matrizen wird vorausgesetzt, dass der Zeiger auf die erste Zeile der Matrix auch auf den Anfang des verwendeten Speicherbereichs zeigt. Sollte dies nach Zeilenvertauschungen nicht mehr der Fall sein, scheitert die Freigabe des Speichers mit `free`. Ein Lösung wäre, in `matrixfree` den kleinsten Zeiger zu suchen und als Argument von `free` zu verwenden.

Auch kann und sollte man in die Funktionen gleich noch eine halbwegs komfortable Behandlung von Fehlern bei der Speicherbereitstellung einbauen, so wie es hier gemacht wurde. Mit der Funktion `matrixanzahl` steht eine einfache Möglichkeit bereit, um zu prüfen, ob bei Programmende alle Matrizen ordnungsgemäß freigegeben wurden.

Kapitel 14

Strukturen

14.1 Vereinbarung von Strukturen

Wir haben gesehen, dass man Elemente mit gleichem Typ zu Feldern zusammenfassen kann. Häufig ist es aber auch wünschenswert, Objekte mit unterschiedlichem Typ zu logischen Einheiten zusammenfassen zu können: Will man etwa eine Studentendatei verwalten, benötigt man (u.a.) den Namen, die Matrikelnummer und die Adresse jedes Studenten. Aber auch für die Zusammenfassung von Objekten mit gleichem Typ bieten sich Felder in manchen Fällen nicht an: Einen Typ „komplexe Zahl“ kann man zwar durch

```
typedef double complex_t[2];
```

deklarieren; der Zugriff auf Real– und Imaginärteil durch Indizierung macht das Arbeiten mit komplexen Zahlen aber nicht gerade übersichtlich.

C bietet die Möglichkeit, unterschiedliche Objekte zu *Strukturen* zusammenzufassen.

Die Deklaration eines Strukturtyps beginnt mit dem Schlüsselwort `struct`. Ihm kann ein Name folgen, mit dem dann im weiteren Programm der Strukturtyp bezeichnet werden kann. Ihm folgt, in geschweifte Klammern eingeschlossen, die Aufzählung der Komponenten des Strukturtyps. Abgeschlossen wird die Deklaration durch ein Semikolon:

```
struct name {
  typ komponente1;
  typ komponente2;
  ...
  typ komponenteN;
};
```

Als Beispiel wollen wir die bereits angesprochenen Studentendaten als Strukturtyp deklarieren:

```
struct student_s {
  char vorname[30];
  char nachname[30];
  int matrikelnr;
  int plz;
  char wohnort[30];
  char strasse[30];
  int hausnr;
};
```

Die Gültigkeit der Namen der Komponenten eines Strukturtyps ist auf den Strukturtyp beschränkt. Damit entstehen auch dann keine Namenskonflikte, wenn eine Komponente

denselben Namen wie eine Variable, ein Feld, eine Funktion oder auch eine Komponente eines anderen Strukturtyps besitzt.

In der beschriebenen Form handelt es sich um eine reine Deklaration, d.h. es wird noch keine Variable mit dem Typ definiert und entsprechend kein Speicherplatz bereitgestellt. Variablen mit einem Strukturtyp, kurz: Strukturen, kann man auf verschiedene Weisen definieren.

Zunächst kann man die Namen direkt zwischen die schließende geschweifte Klammer und das nachfolgende Semikolon der Typdeklaration setzen:

```
struct student_s {
  char vorname[30];
  /* ... wie zuvor */
} student1, student2;
```

Will man im Rest des Programms keinen Bezug mehr auf den Strukturtyp nehmen, kann man den Namen, hier also `student_s`, auch weglassen. Wenn der Name angegeben ist, kann man Stukturen auch so definieren:

```
struct student_s student1, student2;
```

Für das Aufschreiben und auch die Lesbarkeit eines Programms ist es in der Regel günstiger, Strukturtypen mit `typedef` Namen zu geben:

```
typedef struct student_s {
  char vorname[30];
  /* ... wie zuvor */
} student_t;
```

Jetzt kann man Strukturen etwa durch

```
struct student_s student1, student2;
student_t student3, student4;
```

definieren. Bei der Deklaration kann man den Strukturnamen `student_s` weglassen. Die erste Variante der Variablenvereinbarung ist dann jedoch nicht mehr möglich. Das ist jedoch kein Nachteil, da die zweite Variante ohnehin kürzer ist. Eine Kombination von `typedef`–Deklaration und Strukturdefinition ist *nicht* möglich.

Strukturnamen wurden bisher mit einem nachgestellten `_s` versehen. Das ist nicht zwingend erforderlich, verdeutlicht jedoch, dass es sich um einen Strukturnamen handelt. Generell haben Strukturnamen jedoch einen eigenen Namensraum, da sie ja nur zusammen mit `struct` verwendet werden können.

14.2 Operationen mit Strukturen

Operationen mit Strukturen als Ganzem sind nur sehr eingeschränkt möglich:

- Wertzuweisungen zwischen Strukturen sind möglich. Dabei werden die Werte aller Komponenten übertragen. Offensichtliche Voraussetzung ist die Identität der Strukturtypen.
- Funktionen können Strukturen als Parameter besitzen. Die Abarbeitung der Aufrufe erfolgt wie bei (einfachen) Variablen: Im Zuge des Aufrufs wird der Wert des Arguments in eine lokale Struktur der Funktion kopiert. Zumindest bei umfangreicheren Strukturen sollte man sich also genau überlegen, ob die Übergabe eines Zeigers auf die Struktur nicht zweckmäßiger ist.

- Funktionen können Strukturen als Funktionswerte zurückliefern. Wie bei den Parametern sollte man sich aber genau überlegen, ob das zweckmäßig ist. Auch hier kann ein Zeiger-Parameter die bessere Lösung sein.
- Mit dem Adressoperator `&` kann man die Adresse einer Struktur ermitteln.
- Bei der Definition einer Struktur kann man ihren Anfangswert festlegen, in der gleichen Form wie bei Feldern:

```
student_t student = {
  "Klaus", "Meier",
  12345,
  37073, "Goettingen",
  "Einbahnstrasse", 12
};
```

Mit den Komponenten von Strukturen kann man dagegen „ganz normal“ arbeiten. Allerdings reicht die Angabe des Namens einer Komponente offensichtlich nicht aus, da alle Strukturen mit gleichem Typ ja gleichnamige Komponenten besitzen. Erforderlich ist, ähnlich wie bei Zeigern, eine Dereferenzierung: Angegeben werden der Name der Struktur *und* der Name der Komponente; zwischen beide wird der *Punktoperator* (`.`) gesetzt:

```
student_t student;
...
student.plz = 1000;
```

Als Beispiel schreiben wir ein Programm, das den Datensatz eines Studenten liest und wieder schreibt:

```
#include<stdio.h>

/**  Strukturdeklaration  ***********************************/
typedef struct {
  char vorname[30];
  char nachname[30];
  int  matrikelnr;
  int  plz;
  char wohnort[30];
  char strasse[30];
  int  hausnr;
} student_t;

/**  Prototypen  ****************************************/
student_t gelesener_satz (void);
void satz_schreiben (student_t s);

/*= Hauptprogramm ========================================*/
int main(void) {
  student_t satz;
  printf("Datensatz eingeben:\n");
  satz = gelesener_satz ();
  printf("Eingegebener Datensatz:\n");
  satz_schreiben (satz);
  return 0;
}

/*= Eingabefunktion ======================================*/
student_t gelesener_satz(void) {
```

```
  student_t s;

  printf("Vorname: ");
  scanf("%s", s.vorname);
  printf("Nachname: ");
  scanf("%s", s.nachname);
  printf("Matrikelnummer: ");
  scanf("%d", &s.matrikelnr);
  printf("Postleitzahl: ");
  scanf("%d", &s.plz);
  printf("Wohnort: ");
  scanf("%s", s.wohnort);
  printf("Strasse: ");
  scanf("%s", s.strasse);
  printf("Hausnummer: ");
  scanf("%d", &s.hausnr);
  return s;
}

/*= Ausgabefunktion ======================================*/
void satz_schreiben(student_t s) {
  printf("Vorname: %s\n", s.vorname);
  printf("Nachname: %s\n", s.nachname);
  printf("Matrikelnummer: %d\n", s.matrikelnr);
  printf("Postleitzahl: %d\n", s.plz);
  printf("Wohnort: %s\n", s.wohnort);
  printf("Strasse: %s\n", s.strasse);
  printf("Hausnummer: %d\n", s.hausnr);
}
```

Quelltext 14.1: Arbeiten mit Strukturen

14.3 Schachtelung strukturierter Typen

Jede Strukturkomponente darf einen beliebigen Typ besitzen. Wir haben das zum Beispiel schon genutzt, indem wir Felder als Strukturkomponenten deklariert haben. Entsprechend kann man Felder definieren, deren Komponenten Strukturen sind. Außerdem darf jede Strukturkomponente ihrerseits eine Struktur sein.

Zum Beispiel können wir vereinbaren

```
typedef struct {
  char vorname[30];
  char nachname[30];
} name_t;

typedef struct {
  int plz;
  char wohnort[30];
  char strasse[30];
  int hausnr;
} adresse_t;

typedef struct {
  name_t name;
  int matrikelnr;
  adresse_t adresse;
```

```
} student_t;

student_t student, studenten[50];
```

Für die Zugriffe auf einzelne Komponenten muss man dann die „Pfade“ nachverfolgen, zum Beispiel

```
student.adresse.hausnr = 17;
if (studenten[17].name.vorname[0] == 'K')
  ...
```

Die Abarbeitung solcher Pfade erfolgt, wie es nahe liegt, von links nach rechts.

14.4 Zeiger auf Strukturen

Es wurde bereits angesprochen, dass man sich sehr genau überlegen sollte, ob man umfangreiche Strukturen als Parameter von Funktionen oder als Funktionstypen vorsieht, weil beides das Kopieren ganzer Strukturen impliziert. In der Regel wird man deshalb im Zusammenhang mit Funktionen Zeiger auf Strukturen verwenden. Es ist übrigens nicht so recht nachvollziehbar, warum C nicht grundsätzlich Strukturen wie Felder behandelt, also Strukturnamen als Zeiger auf den Anfang der Struktur betrachtet.

Für Zeiger auf Strukturen gelten dieselben Regeln wie für Zeiger auf einfache Variablen:

```
student_t student, *p = &student;
```

Beim Zugriff auf die Komponenten muss man berücksichtigen, dass der Punktoperator höhere Priorität besitzt als der Dereferenzierungsoperator (`*`). Wir *müssen* also

`(*`*strukturzeiger*`).`*strukturkomponente*

schreiben. Abkürzend und suggestiver ist die äquivalente Schreibweise mit dem Pfeil–Operator (`->`).

strukturzeiger`->`*strukturkomponente*

Sehen wir uns als Beispiel noch einmal die Ein–/Ausgabe an. Diesmal sind die Strukturen verschachtelt und es werden den Funktionen Zeiger auf die Strukturen übergeben, um das automatische Kopieren der ganzen Strukturen zu vermeiden.

```
#include<stdio.h>

/**  Strukturdeklarationen  *********************************/
typedef struct {
  char vorname[30];
  char nachname[30];
} name_t;

typedef struct {
  char strasse[30];
  int  hausnr;
  int  plz;
  char wohnort[30];
} adresse_t;

typedef struct {
  name_t    name;
  int       matrikelnr;
```

```
  adresse_t adresse;
} student_t;

/**  Prototypen  ****************************************/
void satz_lesen(student_t *const s);
void satz_schreiben(const student_t *const s);

/*= Hauptprogramm ============================================*/
int main(void) {
  student_t satz;
  printf("Datensatz eingeben:\n");
  satz_lesen (&satz);
  printf("Eingegebener Datensatz:\n");
  satz_schreiben (&satz);
  return 0;
}

/*= Eingabefunktion ==========================================*/
void name_lesen(name_t *const n) {
  printf("Vorname: ");
  scanf("%s", n->vorname);
  printf("Nachname: ");
  scanf("%s", n->nachname);
}

void adresse_lesen(adresse_t *const a) {
  printf("Postleitzahl: ");
  scanf("%d", &a->plz);
  printf("Wohnort: ");
  scanf("%s", a->wohnort);
  printf("Strasse: ");
  scanf("%s", a->strasse);
  printf("Hausnummer: ");
  scanf("%d", &a->hausnr);
}

void satz_lesen (student_t *const s) {
  name_lesen(&s->name);

  printf("Matrikelnummer: ");
  scanf("%d", &s->matrikelnr);

  adresse_lesen(&s->adresse);
}

/*= Ausgabefunktion ==========================================*/
void satz_schreiben (const student_t *const s) {
  printf("Vorname: %s\n",        s->name.vorname);
  printf("Nachname: %s\n",       s->name.nachname);
  printf("Matrikelnummer: %d\n", s->matrikelnr);
  printf("Postleitzahl: %d\n",   s->adresse.plz);
  printf("Wohnort: %s\n",        s->adresse.wohnort);
  printf("Strasse: %s\n",        s->adresse.strasse);
  printf("Hausnummer: %d\n",     s->adresse.hausnr);
}
```

Quelltext 14.2: Arbeiten mit verschachtelten Strukturen und Zeigern auf Strukturen

14.5 Strukturen auf dem Heap

Für Zeiger auf Strukturen gelten dieselben Regeln wie für Zeiger auf einfache Variablen. Insbesondere heißt das auch: Definiert man eine Zeigervariable, so wird nur der Speicher für die Aufnahme des Zeigers bereitgestellt. Durch Zuweisung eines Zeigers auf eine statische oder automatische Struktur kann man solch einer Zeigervariablen einen Wert geben. Aber auch Bereitstellung von Speicher auf dem Heap ist möglich.

```
student_t *p;
...
p = (student_t *)malloc(sizeof (student_t));
```

Hier ist der Operator `sizeof` unentbehrlich! Wieviel Speicher eine Struktur benötigt, kann nämlich von Rechner zu Rechner sehr unterschiedlich sein, selbst wenn die einzelnen Komponenten auf den Rechnern jeweils gleichviel Speicher benötigen. Dahinter steht eine Vorschrift des Standards: Die Komponenten einer Struktur müssen im Speicher des Rechners zwar dieselbe Reihenfolge besitzen wie in der Deklaration – sie müssen aber nicht lückenlos aufeinanderfolgen! Entsprechend addiert der Operator `sizeof` nicht einfach den Speicherbedarf der Komponenten, sondern berücksichtigt auch eventuelle Lücken. Als Grundregel sollte man sich einfach merken: Bei jedem Aufruf von `malloc` steckt im Argument der Operator `sizeof`!

14.6 Datum und Uhrzeit

Mit der Kenntnis von Strukturen kann nun auch die Headerdatei `time.h` besprochen werden. Aktuelles Datum und Uhrzeit kann man sich von der Funktion

```
time_t time(time_t *zeit);
```

liefern lassen – in einer implementationsspezifischen Darstellung. Dabei:

- Der Funktionswert `(time_t)(-1)` (*Typecast*) markiert, dass Datum und Uhrzeit nicht verfügbar sind.
- Wenn `zeit` der Nullzeiger ist, wird nur der Funktionswert geliefert; sonst wird das Resultat zusätzlich auch in die angegebene Variable geschrieben.

Die Headerdatei bietet aber auch die Möglichkeit, daraus eine allgemein verständliche Darstellung erzeugen zu lassen. In ihr ist der Strukturtyp

```
struct tm
```

definiert. Er muss zumindest die folgenden Komponenten besitzen:

`int tm_sec`	Sekunde in der Minute, 0 . . . 61
`int tm_min`	Minute in der Stunde, 0 . . . 59
`int tm_hour`	Stunde seit Mitternacht, 0 . . . 23
`int tm_mday`	Tag im Monat, 1 . . . 31
`int tm_mon`	Monat im Jahr, 0 . . . 11
`int tm_year`	Jahr seit 1900
`int tm_wday`	Tag in der Woche (ab Sonntag), 0 . . . 6
`int tm_yday`	Tag seit 1. Januar, 0 . . . 365
`int tm_isdst`	Sommerzeit-Marke (daylight saving time) (< 0 : unbekannt)

wobei die Reihenfolge der Komponenten durch den Standard nicht festgelegt ist. Die Umwandlung eines Wertes mit dem Typ `time_t` in den Typ `tm` nehmen die Funktionen

```
struct tm *gmtime(const time_t *zeit);
struct tm *localtime(const time_t *zeit);
```

vor – in Greenwich–Standardzeit bzw. in die lokale Zeit. Daneben gibt es auch die Umkehrung: Die Funktion

```
time_t mktime(struct tm *zeitpunkt);
```

wandelt den Wert der Struktur in die interne Darstellung um. Sie unterstellt, dass es sich um eine lokale Zeit handelt.

14.7 Verkettete Listen

Strukturen und Zeiger auf Strukturen bilden die Grundlage einer wichtigen nicht elementaren Datenstruktur, nämlich der *verketteten Liste*. Wegen der Bedeutung der verketteten Listen soll ein kurzer Überblick geben werden.

Anschaulich kann man verkettete Listen mit „Schnitzeljagd“ vergleichen:

- Man hat einen Verweis, wo man beginnen muss (Zeigervariable).
- Am Start findet man einen Zettel mit einer Aufgabe und einen neuen Verweis (Struktur mit Daten– und Zeigerkomponente).
- Folgt man den Verweisen, kommt man irgendwann zum Ende (Zeigerkomponente mit Nullzeiger).

Abbildung 14.1 verdeutlicht das.

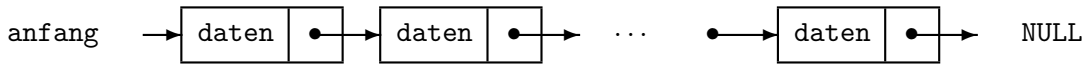

Abbildung 14.1: Verkettete Liste

Im Programm kann man das durch Deklarationen wie

```
struct liste {
  DATENTYP daten;
  struct liste *nachfolger;
};
```

nachvollziehen. Interessant ist, dass hier scheinbar eine „rekursive Deklaration“ vorliegt. Tatsächlich ist die zweite Komponente aber ein Zeigertyp und steht nur für eine Adresse, nicht für den Strukturtyp, der gerade deklariert wird.

Standardoperationen für verkettete Listen sind

- Einfügen eines neuen Eintrags.
- Suchen eines Eintrags mit bestimmtem Wert.
- Entfernen eines Eintrags.

Typischerweise werden verkettete Listen auf dem Heap angelegt. Der Fantasie sind beim Aufbau verketteter Strukturen keine Grenzen gesetzt:

- In einer verketteten Liste kann jeder Eintrag Anfang einer verketteten Liste sein.
- Mehrere Zeigerkomponenten in jeder Struktur erlauben den Aufbau von mehrfach verketteten Listen oder auch von *Bäumen*.

14.8 Verbunde

Formal den Strukturen sehr ähnlich sind die *Verbunde*: Statt `struct` schreibt man `union`. Inhaltlich bestehen jedoch ganz gravierende Unterschiede:

- Bei Strukturen belegen aufeinanderfolgende Komponenten aufeinanderfolgende Plätze im Speicher, wenn auch vielleicht nicht lückenlos, wie wir gesehen haben.
- Bei einem Verbund beginnen alle Komponenten an derselben Speicheradresse. Der Verbund belegt insgesamt so viel Speicher wie seine längste Komponente. Dadurch wird erreicht, dass der gleiche Speicherplatz zu unterschiedlichen Zeitpunkten in unterschiedlicher Weise genutzt werden kann, z.B. je nach Situation zur Speicherung eines eines `float`-Wertes oder (wenn dieser gerade nicht benötigt wird) zur Speicherung eines `int`-Wertes.

In der Praxis kommen Verbunde recht selten vor. Ein kurzes Beispiel soll zeigen, dass ihre Verwendung extrem leicht zu Fehlern führt. Wir hatten gesehen, wie man sich selber mit Castoperatoren „austricksen“ kann, etwa indem man

```
int i, *ip = &i;
float *fp;
...
fp = (float *) ip;
*fp = 6.5f;
printf("%d\n", i);
```

schreibt. Mit einem Verbund kann man dasselbe einfacher haben:

```
union {
  int i;
  float f;
} quatsch;
...
quatsch.f = 6.5f;
printf("%d\n", quatsch.i);
```

Kapitel 15

Probleme der Rechnerarithmetik

15.1 Gleitkomma–Ausnahmebehandlung

Numerisches Rechnen mit Computern ist nicht unproblematisch. Das gilt besonders für das Rechnen mit Gleitkommawerten, in gewissem Maße aber auch für das Rechnen mit ganzzahligen Werten. Diese Probleme sind keineswegs C–spezifisch, sondern treten bei allen Sprachen auf, die die entsprechenden Datentypen kennen, und haben eine einzige Ursache, nämlich die begrenzte Stellenzahl, die dargestellt werden kann.

Erinnern Sie sich noch einmal an die Minimalanforderungen für `double`–Zahlen, die am Anfang des Kurses genannt wurden: Der geforderte minimale Wertebereich ist

$$[-10^{+37}, -10^{-37}] \cup \{0\} \cup [+10^{-37}, +10^{+37}]$$

bei einer minimalen Dichte von 10^{-9}. Hieraus ergeben sich direkt die drei „problematischen“ Fälle:

- Es kann ein *Überlauf* (*overflow*) eintreten, d.h. ein Wert wird dem Betrage nach größer als `DBL_MAX`.
- Es kann ein *Unterlauf* (*underflow*) eintreten, d.h. ein Wert wird dem Betrage nach kleiner als `DBL_MIN`.
- Ein Wert fällt in eine „Lücke“ zwischen zwei darstellbaren Zahlen, d.h. er muss gerundet werden.

Wie auf solche Fälle reagiert wird, kann von Rechner zu Rechner sehr unterschiedlich sein. Mann kann deshalb nur „übliche“ Reaktionen beschreiben; andere Reaktionen sind denkbar.

- Beim Überlauf gibt es drei „übliche“ Reaktionen:
  - Das Programm wird abgebrochen.
  - Der Wert wird auf „Infinity“ gesetzt, einen ausgezeichneten Wert.
  - Der Wert wird undefiniert.

  Division durch Null wird mit gewisser Berechtigung in der Regel wie ein Überlauf behandelt.
- Bei Unterlauf ist es üblich, den Wert durch Null zu ersetzen, ohne dass dieses dem Programm angezeigt wird.
- Gerundet wird in der Regel automatisch, ohne dass dieses dem Programm angezeigt wird.

Nur für den Überlauf schreibt C vor, dass ein Programm selbst festlegen kann, wie darauf zu reagieren ist. Die entsprechenden Mittel stellt C mit der Headerdatei `signal.h` zur Verfügung.

15.2 Überlauf bei ganzen Zahlen

Die ganzzahligen Typen sind auf der einen Seite weniger problematisch, da bei ihnen nur der Überlauf eintreten kann, und auf der anderen Seite problematischer, da die Verhinderung/Vermeidung von Überläufen grundsätzlich und ausschließlich in die Verantwortung des Programmierers gestellt ist.

Überlauf bei ganzzahligen Operationen bewirkt nämlich in der Regel nichts – außer dass mit falschen Werten weitergerechnet wird. Eine Ausnahme sind Divisionen durch Null und natürlich Berechnungen des Restes mit Divisor Null, die vielfach einen Programmabbruch bewirken.

Der Hintergrund für die üblicherweise verschiedenen Vorgehensweisen sind Hardware–Eigenschaften der Rechner: Bei Gleitkommaoperationen ist jeder Überlauf und Unterlauf ein Fehler, der ohnehin besonders behandelt werden muss. Bei ganzzahligen Operationen tritt dagegen häufiger ein Hardware–Überlauf auf, *ohne* dass das ein Fehler ist. Wir sehen uns das an einem Beispiel an. Der Einfachheit halber wird Byte–Arithmetik betrachtet, auch wenn C sie nicht explizit vorsieht. Wir haben etwa

$$\begin{array}{rr} & 1111\ 1111_b \\ + & 0000\ 0001_b \\ \hline = & 1\ 0000\ 0000_b \end{array}$$

wobei das werthöchste, neunte Bit gerade das überlaufende Bit ist, das nicht mehr gespeichert werden kann. Ob dieser Überlauf ein Fehler ist oder nicht, hängt davon ab, ob wir die Operanden als vorzeichenbehaftet oder vorzeichenlos betrachten:

- Bei vorzeichenbehafteter Operation haben wir gerade $-1 + 1 = 0$, also das korrekte Resultat!
- Bei vorzeichenloser Operation haben wir $255 + 1 = 0$ – und das ist offenbar falsch!

Sieht man sich das ganze genauer an, so stellt man fest, dass bei vorzeichenlosen Operationen modulo `..._MAX + 1` gerechnet wird. Bei vorzeichenbehafteten Operationen kommt man zu einem ähnliche Ergebnis.

Übrigens: Bei den impliziten Typumwandlungen im Zuge von arithmetischen Operationen sind die Regeln gerade so angelegt, dass Überlauf und Unterlauf nicht auftreten können. Werden Typumwandlungen dagegen durch Wertzuweisungen oder Castoperatoren erzwungen, können Überlauf und Unterlauf wie bei arithmetischen Operationen auftreten.

15.3 Normalisierte Gleitkomma–Darstellungen

Wir haben gesehen, dass man im Programm oder auch bei der Eingabe für Gleitkommawerte beliebig viele verschiedene Darstellungen verwenden kann. Der Wert `1.2` kann etwa als `1.2`, `12e-1`, `0.12e1`, usw., dargestellt werden.

Im Rechner gibt es dagegen für jeden darstellbaren Gleitkommawert nur genau eine Darstellung, die als *normalisierte Darstellung* bezeichnet wird. Prinzipiell gibt es unterschiedliche normalisierte Darstellungen (beschrieben in einer Norm von IEEE), viele Rechner verwenden von diesen jedoch nur genau eine.

Der Anschaulichkeit halber wird die normalisierte Darstellungen für dezimale Rechenwerke beschrieben. Als Beispiel nehmen wir ein Rechenwerk, das mit fünf signifikanten Stellen

und einem einstelligen Exponententeil arbeitet. Eine normalisierte Darstellung setzt den (gedachten) Dezimalpunkt direkt links neben die werthöchste signifikante Ziffer und passt den Exponenten entsprechend an; in ihr hat der Wert `1.2` also die eindeutige Darstellung

$$0.12000 \cdot 10^{1}$$

Der Wertebereich dieses Rechenwerkes ist dann

$$[-0.99999 \cdot 10^{+9}, -0.10000 \cdot 10^{-9}] \cup \{0\} \cup [+0.10000 \cdot 10^{-9}, +0.99999 \cdot 10^{+9}]$$

wobei Null ($0.00000 \cdot 10^{0}$) besonders definiert werden muss, da die Darstellung nicht normalisiert ist.

Wenn man mit solchen Darstellungen rechnen will, muss man sie in die beiden Bestandteile zerlegen, die beiden Bestandteile getrennt verarbeiten und dann wieder zu einer normalisierten Darstellung zusammenfügen.

15.4 Assoziativ– und Kommutativgesetz

Ausdrücke werden Schritt für Schritt ausgewertet – und in jedem Schritt muss der berechnete Wert innerhalb des Wertebereichs des Typs liegen. Wir sehen uns das für den Ausdruck $\frac{ab}{cd}$ an, der in `float`–Arithmetik ausgewertet werden soll:

`A`	`B`	`C`	`D`	`A*B/C/D`	`A/C*B/D`
10^{30}	10^{30}	10^{30}	10^{30}	Überlauf	1
10^{30}	10^{-30}	10^{-30}	10^{30}	1	Überlauf
10^{-30}	10^{-30}	10^{-30}	10^{-30}	Unterlauf	1
10^{-30}	10^{30}	10^{30}	10^{-30}	1	Unterlauf

Obwohl das theoretische Resultat in allen Fällen 1 ist, können Überlauf und Unterlauf auftreten!

Assoziativ– und Kommutativgesetz sind also auf einem Rechner für Gleitkommazahlen nicht erfüllt, da die Voraussetzungen für beide Gesetze nicht erfüllt sind: Zur Verfügung hat man ja nicht die reellen Zahlen, sondern nur eine Teilmenge, deren Werte nicht dicht in den reellen Zahlen liegen!

Folgerung: Man sollte sich bei Gleitkommaausdrücken genau überlegen, in welcher Reihenfolge man sie ausgewertet haben möchte. Der Standard unterstützt das, im Gegensatz zum „alten“ C: Scheinbar redundante Klammern dürfen nicht mehr „wegoptimiert“ werden! War ein Compiler früher berechtigt, in einem Ausdruck wie

```
a + (b + c)
```

die Klammern schlichtweg zu ignorieren (was unter Umständen zu „Klimmzügen“ im Programm zwang), so *muss* ein Standardcompiler sie jetzt berücksichtigen!

Beispiel: Sind `a = 1`, `b = -1e12` und `c = 1e12`, dann ergibt sich bei einem Rechenwerk mit 10 signifikanten Stellen

```
a + (b + c) = 1
(a + b) + c = 0
```

15.5 Rundungsfehler

Damit kommen wir auch schon zum nächsten Problem, nämlich der Rundung, die aufgrund der begrenzten Stellenzahl in vielen Fällen unumgänglich ist, und den sich daraus ergebenden *Rundungsfehler*.

Nehmen wir einmal an, wir hätten ein dezimales Rechenwerk zur Verfügung, das mit zwei Dezimalstellen Genauigkeit arbeitet. Dann lässt sich in diesem Rechenwerk die Zahl 1.5 exakt speichern. Bildet man jetzt jedoch das Produkt $1.5 * 1.5$, so resultiert die Zahl 2.25 mit drei signifikanten Stellen, die nicht mehr exakt gespeichert werden kann, sondern zunächst durch Rundung auf zwei signifikante Stellen verkürzt werden muss. In unserem Rechenwerk ist das Resultat, je nach Art der Rundung, also entweder 2.2 oder 2.3.

Genau diese Probleme treten bei jedem Rechner auf. Allerdings lässt sich das, was bei einem Rechner passiert, für einen Menschen nicht mehr so auf den ersten Blick überschauen: Der Mensch ist in seinen Zahlenvorstellungen sehr stark auf das Dezimalsystem fixiert, während die Rechner ja mit binären Zahlendarstellungen arbeiten.

Und so wird vielfach übersehen, dass Rundungsfehler nicht erst beim eigentlichen Rechnen, sondern bereits bei der Umwandlung der dezimalen Konstantendarstellung in die interne Darstellung auftreten. Dazu ein Beispiel: Der „schöne“ Dezimalbruch 0.2_d ergibt bei Umwandlung in Binärdarstellung den periodischen Bruch $0.001100110011\ldots_b$, lässt sich also im Rechner nur gerundet darstellen.

Bei arithmetischen Operationen mit gerundeten Zahlen können die Rundungsfehler unter Umständen erheblich anwachsen. Dazu das folgende Beispiel, das 10000 Mal 0.2 addiert und das Ergebnis ausgibt:

```
float schritt = 0.2f, summe = 0.0f;
int i;

for (i = 0; i < 10000; i++)
  summe += schritt;

printf("%f\n", summe);
printf("%f\n", 10000 * schritt);
```

Ein Testlauf ergab für die erste Ausgabe den Wert 1999.805835 und für die zweite Ausgabe 2000.000000.

Der Programmierer muss sich also sehr genau überlegen, wie er sein Programm formuliert, um den Einfluss der Rundungsfehler möglichst gering zu halten. Dazu ein Beispiel: Eine Funktion $f(x)$ sei in einem Intervall $[a, b]$ zu tabellieren, wobei die Anzahl N der Teilintervalle vorgegeben ist.

Als “naive” Lösung bietet sich an, zunächst einmal die Schrittweite zu berechnen und ihren Wert dann bei jedem Schleifendurchlauf zur Bestimmung des nächsten Tabellierungspunktes zu verwenden. Das Programm hat dann das folgende Schema:

```
h = (b - a) / N;
x = a;
while (x <= b)
{
  printf("%f %f\n", x, f (x));
  x += h;
}
```

Mit den Werten `a = 0`, `b = 4` und `N = 40` erhält hier die Variable `x` vor dem letzten Schleifendurchlauf den Wert 3.9999980, erreicht also nicht exakt den rechten Rand des Ta-

bellierungsintervalls. Entsprechend ungenau werden natürlich auch die tabellierten Funktionswerte.

Günstiger ist es, nicht mit der Schrittweite zu arbeiten, sondern die Tabellierungspunkte jeweils direkt zu berechnen, etwa nach dem folgenden Schema

```
i = 0;
while (i <= N)
{
  x = ((N - i) * a + i * b) / N
  printf("%f %f\n", x, f (x));
  i++;
}
```

Hier wird, wieder mit den Werten `a = 0`, `b = 4` und `N = 40`, beim letzten Schleifendurchlauf der rechte Rand des Tabellierungsintervalls exakt erreicht.

Auch bei Typumwandlungen können Rundungsfehler auftreten – und das nicht nur bei erzwungenen Umwandlungen: Für den Typ `float` sind nur 6 signifikante Stellen vorgeschrieben, während der Typ `long` über mindestens 10 Stellen verfügen muss. Bei einer Umwandlung von `long` in `float`, wie sie ggf. automatisch bei der Auswertung von Ausdrücken erfolgt, können also durchaus die letzten Stellen des `long`–Wertes verloren gehen.

15.6 Auslöschung

Wir haben eben gesehen, dass sich die Rundungsfehler bei Gleitkomma–Arithmetik in bestimmten Situationen beherrschen oder doch zumindest reduzieren lassen. In anderen Situationen ist das nicht so einfach oder sogar unmöglich.

Wir betrachten zunächst wieder ein Beispiel: Gegeben seien Zahlen

$$\begin{aligned} x_1 &= 0.42178 \cdot 10^2 \\ x_2 &= 0.23722 \cdot 10^3 \\ y_1 &= 0.99986 \cdot 10^2 \\ y_2 &= 0.99987 \cdot 10^2 \end{aligned}$$

Dann gilt

$$\begin{aligned} x_1 x_2 &= 0.1000546516 \cdot 10^5 \\ y_1 y_2 &= 0.9997300182 \cdot 10^4 \end{aligned}$$

und

$$\begin{aligned} x_1 x_2 - y_1 y_2 &= 0.10005465160 \cdot 10^5 \\ &\quad - \; 0.09997300182 \cdot 10^5 \\ &= 0.00008164978 \cdot 10^5 \\ \text{normalisiert:} \qquad & 0.81649780000 \cdot 10^1 \end{aligned}$$

Das zwischenzeitliche Vorkommen führender Nullen wird als *Auslöschung* bezeichnet.

Erlaubt die verfügbare Arithmetik nicht die exakte Darstellung der Zwischenresultate, sondern z.B. nur fünfstellige Resultate, so werden bereits die Produkte $x_1 x_2$ und $y_1 y_2$ gerundet geliefert:

$$x_1 x_2 \approx x = 0.10005 \cdot 10^5 \quad \text{und} \quad y_1 y_2 \approx y = 0.99973 \cdot 10^4$$

Erst danach kann die Subtraktion ausgeführt werden und liefert

$$\begin{aligned} x - y &= 0.100050 \cdot 10^5 \\ &- \ 0.099973 \cdot 10^5 \\ \hline &= 0.000077 \cdot 10^5 \\ \text{normalisiert:} \quad & \quad 0.770000 \cdot 10^1 \\ \text{gerundet:} \quad & \quad 0.77000 \ \cdot 10^1 \end{aligned}$$

Der Vergleich der Differenzen $x_1x_2 - y_1y_2$ und $x - y$ zeigt, dass das gerundete Resultat zwar dieselbe Größenordnung besitzt wie das exakte Resultat, dass aber nicht einmal die werthöchsten Ziffern übereinstimmen.

Auslöschung tritt immer dann ein, wenn zwei beinahe gleichgroße Zahlen subtrahiert werden. Selbst wenn die einzelne Subtraktion fehlerfrei ausgeführt wird, ist die Auslöschung numerisch gefährlich, wenn die zu subtrahierenden Daten selber bereits gerundet wurden.

Im Beispiel, das wir eben betrachtet haben, hatte das tatsächliche Resultat noch die Größenordnung des theoretischen Resultats. Es bereitet jedoch keine Probleme, Beispiele zu finden, in denen auch die Größenordnungen voneinander abweichen. Dazu betrachten wir die Taylorreihe der Exponentialfunktion

$$e^z = \sum_{k=0}^{\infty} \frac{z^k}{k!}$$

Diese Reihe ist absolut konvergent für jedes z aus der komplexen Zahlenebene. Für reelles, negatives z haben die Summanden alternierende Vorzeichen. In diesem Fall ist der Fehler, den man begeht, wenn man die n–te Partialsumme als Wert für e^z nimmt, kleiner als der $(n+1)$–te Summand, der als erster vernachlässigt wird.

Wertet man diese Reihe unter Verwendung einer Arithmetik mit 6–stelliger dezimaler Mantisse für $z = -14$ aus, so erhält man nach 50 Additionen den Wert -0.159691 – das korrekte Resultat wäre 0.00000083152900! (Der Abbruch der Summation nach 50 Schritten erfolgt, weil alle weiteren Summanden so klein sind, dass sie bei der verwendeten Arithmetik keinen Beitrag mehr zu der Summe liefern.)

Hier lässt sich mit Mitteln der Programmiertechnik nichts mehr ausrichten. Abhilfe schaffen nur noch besondere Algorithmen, die die Probleme der Gleitkommaarithmetik berücksichtigen, oder der Einsatz der Intervallrechnung.

Kapitel 16

Ein–/Ausgabe

16.1 Verarbeitung von Dateien

Längst nicht in jedem Falle kommt man mit Standardein– und –ausgabe aus, auch nicht unter Berücksichtigung der Möglichkeit, beide umzuleiten.

Ein Beispiel: Ein Programm soll zwei in sich sortierte Dateien „mischen“, d.h. es soll die beiden Dateien parallel lesen und ihre Inhalte sortiert in eine dritte Datei schreiben. Dieses lässt sich prinzipiell mit Umleitung der Standardeingabe *nicht* realisieren, weil Umleitung immer voraussetzt, dass man die Eingabe aus einer einzigen Datei liest.

Abhilfe schafft die Möglichkeit, direkt auf Dateien zuzugreifen. Bevor man aber direkt auf eine Datei zugreifen kann, muss man eine Vorarbeit vornehmen: Dateien werden ja vom Betriebssystem verwaltet; zu ihrer Identifikation dienen Namen. Diese Namen kann man in einem C–Programm für die eigentlichen Zugriffe nicht verwenden, sondern nur im Zuge der Vorarbeit, nämlich durch Zuordnung des (externen) Namens der Datei zu einer (programminternen) *Dateivariablen*.

Für diese Zuordnung stellt `stdio.h` die Funktion

```
FILE *fopen (const char *dateiname,
             const char *zugriff);
```

zur Verfügung:

- Der erste Parameter ist der Zeiger auf den String, der den externen Namen der Datei enthält.
- Der zweite Parameter ist der Zeiger auf einen String, der die gewünschte Art des Zugriffs beschreibt. Eine Übersicht über die Optionen ist in Tabelle 16.1 aufgelistet.
- Der Funktionswert ist der Zeiger auf eine Dateivariable, falls die Zuordnung vorgenommen werden konnte, oder der Nullzeiger. Der Typ `FILE` ist ebenfalls in `stdio.h` deklariert.

Die Zeiger, die `fopen` liefert, sind ausschließlich zur Weitergabe an die anderen Funktionen bestimmt, die Dateien bearbeiten. Der Standard legt auch keinerlei Einzelheiten des Aussehens des Typs `FILE` fest; in der Regel wird es sich um einen Strukturtyp handeln, der aber je nach Implementation völlig unterschiedlich aussehen kann.

Das Pendent zu `fopen` ist die Funktion

```
int fclose (FILE *datei);
```

Option	Beschreibung
`r`	Datei lesen; die Datei muss bereits existieren
`w`	Datei schreiben; falls die Datei existiert, wird ihr Inhalt überschrieben
`a`	Datei fortsetzen; erstellt Datei, falls sie nicht existiert
`r+`	Datei lesen und schreiben; die Datei muss bereits existieren
`w+`	Datei lesen und schreiben; falls die Datei existiert, wird ihr Inhalt überschrieben
`a+`	Datei lesen und fortsetzen; erstellt Datei, falls sie nicht existiert

Tabelle 16.1: Optionen zum Öffnen von Dateien mit `fopen`

Sie löst die Zuordnung für die übergebene Dateivariable und meldet mit dem Funktionswert Null bzw. `EOF` ihren Erfolg bzw. Misserfolg.

Lesen und schreiben können wir jetzt nicht mehr mit `scanf`, `getchar`, `printf` und `putchar`, weil wir natürlich die betroffene Datei zusätzlich nennen müssen. Die entsprechenden Funktionen

```
int fscanf(FILE *datei, const char *format, ...);
int fgetc(FILE *datei);
int fprintf(FILE *datei, const char *format, ...);
int fputc(int c, FILE *datei);
```

arbeiten aber letztlich genauso wie die bekannten Funktionen, nur dass sie eben gerade auf die angegebene Datei anstelle der Standardein– bzw. –ausgabe zugreifen.

Als Beispiel wird die Konkatenation von zwei Dateien realisiert:

```
#include <stdio.h>

/**  Prototypen  ****************************************/
static FILE *zugeordnete_datei(const char *frage,
                               const char *zugriff);
static void datei_kopieren(FILE *ziel, FILE *quelle);

/*= Hauptprogramm ==========================================*/
int main(void) {
  FILE *quelle, *ziel;

  ziel = zugeordnete_datei("Zieldatei: ", "w");
  if (ziel == NULL) {
    printf("Zieldatei nicht verfuegbar -- stop\n");
    return 1;
  }

  quelle = zugeordnete_datei("1. Quelldatei: ", "r");
  if (quelle == NULL) {
    printf("1. Quelldatei nicht verfuegbar -- stop\n");
    return 2;
  }
  datei_kopieren(ziel, quelle);
  fclose(quelle);

  quelle = zugeordnete_datei("2. Quelldatei: ", "r");
```

```
  if (quelle == NULL) {
    printf("2. Quelldatei nicht verfuegbar -- stop\n");
    return 3;
  }
  datei_kopieren(ziel, quelle);
  fclose(quelle);

  fclose(ziel);
  return 0;
}

/*= Zuordnung einer Datei (mit Abfrage ihres Namens) ======*/
static FILE *zugeordnete_datei(const char *frage,
                               const char *zugriff) {
  char name[FILENAME_MAX];
  int i;
  FILE *datei;

  printf("%s", frage);
  i = 0;
  while ((i < FILENAME_MAX) && ((name[i] = getchar()) != '\n'))
    i++;
  if (i >= FILENAME_MAX)
    i--;
  if (name[i] != '\n')
    while (getchar() != '\n')
      ;
    name[i] = '\0';

    datei = fopen(name, zugriff);
    return datei;
}

/*= Kopieren einer Datei (zeichenweise) ====================*/
static void datei_kopieren(FILE *ziel, FILE *quelle) {
  int c;
  while ((c = fgetc(quelle)) != EOF)
    fputc (c, ziel);
}
```

Quelltext 16.1: Dateiverarbeitung: Konkatenation

16.2 Formatierte und binäre Ein–/Ausgabe

Wie schon erwähnt, muss man die Ein–/Ausgabe, die wir bislang verwendet haben, genauer als „formatierte Ein–/Ausgabe“ bezeichnen. Im Zuge der Übertragung erfolgt in der Regel eine Umwandlung der Darstellung:

- Bei der Eingabe müssen die Zeichen *interpretiert* werden, um herauszufinden, wo ein Eingabewert anfängt und wo er aufhört. Bei numerischen Werten muss dann die Zeichenfolge in die interne, binäre Darstellung des Wertes umgewandelt werden.
- Bei der Ausgabe numerischer Werte muss die interne, binäre Darstellung in eine Zeichenfolge umgewandelt werden.

Das Pendent zur formatierten Ein–/Ausgabe ist die *binäre* (*unformatierte*) Ein–/Ausgabe: Bei ihr werden die Bitfolgen der Werte im Zuge der Übertragung *nicht* verändert.

Binäre Ein–/Ausgabe hat einen wesentlichen Vorteil: Sie ist in der Regel *sehr viel* schneller als formatierte Ein–/Ausgabe. Das ist auch ohne weiteres einsichtig: Die Umwandlungen zwischen interner Darstellung und Zeichendarstellung erfordern eine Vielzahl von arithmetischen Operationen; wir haben das in einem Beispiel für eine ganze Zahl bereits einmal gesehen. Spart man diese Operationen ein, *muss* das die Übertragung beschleunigen.

Andererseits lässt sich binäre Ein–/Ausgabe aus zwei Gründen nur eingeschränkt einsetzen:

- Nicht jedes Ein–/Ausgabegerät erlaubt binäre Ein–/Ausgabe! Eine Tastatur kann zum Beispiel nur Zeichen liefern; ein Bildschirm oder Drucker kann mit binären Darstellungen nichts anfangen. Bei all diesen Geräten ist man also auf formatierte Ein–/Ausgabe angewiesen.
- Binärdateien sind in der Regel nicht portabel: Ein `int`–Wert zum Beispiel benötigt in der Datei wie im Speicher des Rechners 2 oder 4 Bytes (oder was auch immer); auf einem Rechner, der `int`–Werte anders darstellt, kann die Binärdatei ohne weiteres also nicht gelesen werden. Formatierte Dateien sind zwar auch nicht ohne weiteres portabel, weil die Zeichencodes auf verschiedenen Rechern verschieden sein können. Die Umsetzung von Zeichencodes ist jedoch extrem simpel, verglichen mit der Umsetzung der internen Darstellungen numerischer Werte.

  Anders formuliert: Bei formatierten Dateien hat man immer eine implizite Interpretationsvorschrift, nämlich Interpretation als Zeichen; bei Binärdateien fehlt diese Vorschrift.

Unter UNIX ist es deshalb üblich, weitestgehend formatierte Ein–/Ausgabe und entsprechend formatierte Dateien zu verwenden. Auf die beiden Funktionen `fread` und `fwrite`, mit denen binäre Ein–/Ausgabe möglich ist, wird deshalb auch nicht näher eingegangen.

Die wesentlichsten Funktionen zur formatierten Ein–/Ausgabe kennen wir bereits seit längerem; was nachzutragen bleibt, sind die vielfältigen Möglichkeiten, Einfluss auf die Form der Umwandlung zu nehmen.

16.3 Ausgabeformate

Die Formatbeschreiber für die Ausgabe haben die allgemeine Form

%<*modus*><*laenge*><.*stellen*><*typ*>*kennung*

Wir sehen: Außer dem Prozentzeichen am Anfang und dem Kennbuchstaben *kennung* am Ende sind alle Einträge optional.

Der Formatierungsstring und die Liste der Ausgabewerte werden linear abgearbeitet. Die Abarbeitung beginnt mit der Interpretation des Formatierungsstring. Was weiter passiert, hängt von den dabei gefundenen Zeichen ab:

- Wenn im Formatierungsstring ein Prozentzeichen gefunden wird, wird zunächst der vollständige Formatbeschreiber interpretiert. Dann wird der nächste Ausgabewert ihm entsprechend in eine Zeichenfolge umgewandelt und in die Ausgabedatei übertragen.
- Wenn im Formatierungsstring ein anderes Zeichen gefunden wird, wird es ohne weiteres in die Ausgabedatei übertragen.

Die Kennungen für die verschiedenen Datentypen bzw. ihre externen Darstellungen, die weitgehend am Anfang des Kurses bereits aufgezählt wurden, lassen sich in vier Gruppen unterteilen:

1. Ausgabe ganzer Zahlen

   Der Ausgabewert muss den Typ `int` bzw. `unsigned int` besitzen. Die zu erzeugende externe Darstellung wird durch die folgenden Kennbuchstaben beschrieben:

   - `i` der Wert wird als `signed int` betrachtet, die Darstellung erfolgt dezimal, ggf. mit Vorzeichen
   - `d` wie `i`
   - `u` der Wert wird als `unsigned int` betrachtet, die Darstellung erfolgt dezimal
   - `o` der Wert wird als `unsigned int` betrachtet, die Darstellung erfolgt oktal
   - `x` der Wert wird als `unsigned int` betrachtet, die Darstellung erfolgt hexadezimal unter Verwendung von Kleinbuchstaben
   - `X` wie `x`, jedoch mit Großbuchstaben

2. Ausgabe von Gleitkommazahlen

   Der Ausgabewert wird als `double` erwartet. Die zu erzeugende externe Darstellung wird durch die folgenden Kennbuchstaben beschrieben:

   - `f` Darstellung ohne Exponententeil
   - `e` Darstellung mit Exponententeil, der Exponenten–Kennbuchstabe ist `e`; die Darstellung wird so normiert, dass links vom Dezimalpunkt genau eine Ziffer steht, die nur dann Null ist, wenn der Wert insgesamt Null ist
   - `E` wie `e`, jedoch mit Exponenten–Kennbuchstabe `E`
   - `g` wie `f` oder `e`, je nach Größenordnung des Ausgabewertes; Nullen am Ende der Ziffernfolge werden unterdrückt; der Dezimalpunkt wird nur geschrieben, wenn ihm eine Ziffer folgt
   - `G` wie `g`, jedoch wird ggf. `E` anstelle von `e` verwendet

   Die letzte Ziffer wird jeweils gerundet.

3. Ausgabe von Zeichen und Strings

   - `s` der Ausgabeparameter wird als Zeiger auf einen String betrachtet, der Wert des String geschrieben
   - `c` der `int`–Wert wird in `unsigned char` umgewandelt geschrieben

4. Kennungen für verschiedene Zwecke

   - `p` der Wert wird als Zeiger betrachtet und in implementations–spezifischer Darstellung geschrieben
   - `n` es wird keine Ausgabe erzeugt, sondern die Anzahl der bislang übertragenen Zeichen in den nächsten Parameter der Ausgabeliste geschrieben; dieser Parameter muss entsprechend den Typ `int *` besitzen
   - `%` ein Prozentzeichen wird geschrieben (dem Formatbeschreiber wird kein Ausgabewert zugeordnet)

Wenn in einem Formatbeschreiber nur die Kennung *kennung* angegeben ist, werden Werte mit bestimmten Typen erwartet, erfolgt die Darstellung in einem Standardformat. Zum Beispiel wird bei Verwendung des Formatbeschreibers `u` ein `unsigned int`–Wert erwartet. Auch werden bei der Umwandlung nur die signifikanten Ziffern des Wertes erzeugt. Abweichende Werte und Darstellungen lassen sich mit den optionalen Einträgen der Formatbeschreiber anzeigen bzw. erzeugen.

Der Eintrag *laenge* legt die Mindestzahl der zu erzeugenden Zeichen fest, die ggf. durch voran– oder nachgestellte Leerzeichen zu erreichen ist. Falls bei der Umwandlung des Wertes mehr Zeichen entstehen, wird die Angabe ignoriert. *laenge* kann eine dezimale Konstante oder ein Stern sein; wenn ein Stern angegeben ist, wird der nächste Wert der Ausgabeliste als Wert von *laenge* genommen und nicht geschrieben. Dieser Wert *muss* den Typ `int` besitzen.

Negative Längen sind nicht möglich (abgesehen davon, dass sie auch nicht sinnvoll sind): Ein Minuszeichen wird als Modifikator betrachtet und bewirkt linksbündige Ausgabe anstelle der sonst rechtsbündigen, so dass für *laenge* nur die angegebene Ziffernfolge übrigbleibt.

Einige Beispiele: Seien `x` und `y` durch

```
int x = 743, i = 4;
float y = 123.4567;
```

definiert. Dann erhalten wir

```
printf("%d", x);                  → 743
printf("%2d", x);                 → 743
printf("%6d", x);                 → ␣␣␣743
printf("%-6d", x);                → 743␣␣␣
printf("%*d", i, x);              → ␣743

printf("%f", y);                  → 123.456703
printf("%4f", y);                 → 123.456703

printf("%s", "Hallo");            → Hallo
printf("Hallo");                  → Hallo
printf("%s", "Hallo%Hallo");      → Hallo%Hallo
printf("Hallo%Hallo");            → Fehler!
printf("%10s", "Hallo");          → ␣␣␣␣␣Hallo
```

Wie *stellen* interpretiert wird, hängt in erster Linie von *kennung* ab. Formal kann *stellen* wie *laenge* eine dezimale Konstante oder ein Stern sein; wenn ein Stern angegeben ist, wird der nächste Wert der Ausgabeliste als Wert von *stellen* genommen und nicht geschrieben. Dieser Wert *muss* den Typ `int` besitzen:

- Bei ganzzahligen Werten ist *stellen* die Mindestzahl der zu schreibenden Ziffern, auch wenn dadurch führende Nullen erscheinen.
- Bei den Kennungen `f`, `e` und `E` ist *stellen* die Anzahl der Stellen hinter dem Dezimalpunkt; wird die Anzahl der Stellen hinter dem Dezimalpunkt auf Null gesetzt, entfällt auch der Dezimalpunkt selbst.
- Bei den Kennungen `g` und `G` wird genauso verfahren; nur wird ggf. bei *stellen* der Wert 0 durch 1 ersetzt.
- Bei Strings bewirkt *stellen*, dass höchstens so viele Zeichen des String übertragen werden.

Seien `x` und `y` wie oben definiert. Dann gilt

```
printf("%6.4d", x);               → ␣␣0743
printf("%-6.4d", x);              → 0743␣␣

printf("%7.2f", y);               → ␣123.46
```

```
printf("%7.3s", "Hallo");          → ␣␣␣␣Hal
printf("%7.6s", "Hallo");          → ␣Hallo␣
```

Dass man Ausgabewerte mit `long`–Typen und dem Typ `long double` durch Angabe des Kennbuchstabens `l` bzw. `L` für *typ* besonders kennzeichnen muss, haben wir bereits gesehen.

Ebenfalls erwähnt wurde bereits das Zeichen `-` als *modus*. Ein weiteres Zeichen, das hier zugelassen ist, ist `+`. Es bewirkt für vorzeichenbehaftete Zahlen, dass auch positive Vorzeichen geschrieben werden.

Für weitere Einzelheiten zu den Ausgabeformaten sei auf die Literatur verwiesen.

16.4 Eingabeformate

Die Formatbeschreiber für die Eingabe haben die Form

`%<*><`*laenge*`><`*typ*`>`*kennung*

Dabei sind die in spitzen Klammern stehenden Einträge erneut optional.

Der Formatierungsstring, die Zeichen aus der Eingabedatei und die Liste der Eingabevariablen werden linear abgearbeitet. Die Abarbeitung beginnt mit der Interpretation des Formatierungsstring. Was weiter passiert, hängt von den dabei gefundenen Zeichen ab. Zunächst soll der einfachste Fall unterstellt werden, dass nämlich die Formatbeschreiber nur aus dem einleitenden Prozentzeichen und der Kennung *kennung* bestehen, dass keine optionalen Einträge enthalten sind:

- Wenn im Formatierungsstring ein „white space“ gefunden wird, werden so lange Zeichen aus der Eingabedatei geholt, bis das nächste Zeichen *kein* „white space“ ist. Die übertragenen „white spaces“ werden ignoriert.
- Wenn im Formatierungsstring ein Zeichen gefunden wird, das kein Prozentzeichen und kein „white space“ ist, wird es mit dem nächsten Zeichen der Eingabedatei verglichen. Wenn Übereinstimmung besteht, werden beide Zeichen ignoriert. Sonst werden so lange Zeichen in der Eingabedatei übersprungen, bis ein „white space“ gefunden wird, dann die Routine mit Fehlerstatus abgebrochen.
- Wenn im Formatierungsstring ein Prozentzeichen gefunden wird, wird zunächst der vollständige Formatbeschreiber interpretiert. Dann werden so lange Zeichen aus der Eingabedatei geholt, wie diese zur Kennung *kennung* des Formatbeschreibers „passen“. Anschließend wird diese Zeichenfolge in die entsprechende interne Darstellung umgewandelt und das Resultat in der nächsten Eingabevariablen gespeichert.

Die Kennungen für die verschiedenen Datentypen bzw. ihre externen Darstellungen lassen sich wieder in vier Gruppen unterteilen. Wir kennen sie weitgehend bereits:

1. Eingabe ganzer Zahlen

   Die Eingabevariable muss den Typ `int` bzw. `unsigned int` besitzen. Die erwartete externe Darstellung wird durch die folgenden Kennbuchstaben beschrieben:

   - `i` ganze Zahl mit oder ohne Vorzeichen in C–Schreibweise, d.h. mit `0x` oder `0X` beginnende Zahlen werden als hexadezimal, andere mit `0` beginnende Zahlen als oktal dargestellt betrachtet; Zahlen, die nicht mit Null beginnen, werden als dezimal dargestellt betrachtet
   - `d` ganze Zahl mit oder ohne Vorzeichen in dezimaler Darstellung
   - `u` ganze Zahl ohne Vorzeichen in dezimaler Darstellung

- `o` ganze Zahl mit oder ohne Vorzeichen in oktaler Darstellung
- `x` ganze Zahl mit oder ohne Vorzeichen in hexadezimaler Darstellung, jedoch ohne einleitendes `0x` bzw. `0X`
- `X` wie `x`

2. Eingabe von Gleitkommazahlen

   Die Eingabevariable muss den Typ `float` besitzen. Die verfügbaren Kennungen sind die Buchstaben `e`, `E`, `f`, `g` und `G`. Sie erwarten sämtlich eine beliebige zulässige Darstellung einer Gleitkommazahl, also mit oder ohne Vorzeichen, mit oder ohne Dezimalpunkt, mit oder ohne Exponententeil.

3. Eingabe von Zeichen und Strings

   Die Eingabevariable muss den Typ `char` besitzen; in der Regel wird sie die erste Komponente eines Feldes sein müssen, das hinreichend groß ist, alle übertragenen Zeichen aufzunehmen. Die erwartete externe Darstellung wird durch die folgenden Kennungen beschrieben:

   - `s` beliebige Zeichenfolge, beendet durch ein „white space“; an die Zeichenfolge wird automatisch ein Null–Zeichen angehängt
   - `[` wie `s`, jedoch kann explizit angegeben werden, bei welchen Zeichen die Übertragung stoppen soll (vgl. unten)
   - `c` einzelnes Zeichen, das auch ein „white space“ sein kann

4. Kennungen für verschiedene Zwecke

   - `p` Erwartet wird ein Zeigerwert in implementations–spezifischer Darstellung; die Eingabevariable muss einen Zeigertyp besitzen. Die Kennung ist, wenn überhaupt, nur dann sinnvoll, wenn die zu lesende Eingabe unter derselben Implementation geschrieben wurde.
   - `n` Es werden keine Zeichen aus der Eingabedatei geholt, sondern die Anzahl der bisher umgewandelten Werte in die nächste Eingabevariable übertragen. Diese Eingabevariable muss den Typ `int` besitzen.
   - `%` Kennung für ein Prozentzeichen, das als Eingabezeichen erwartet wird.

Die Kennbuchstaben für numerische Werte müssen durch die Typkennung `h`, `l` oder `L` modifiziert werden, wenn die Eingabevariable *nicht* den Typ `int` bzw. `float` besitzt. Durch *laenge* kann man angeben, wie viele Zeichen maximal für den Eingabewert interpretiert werden sollen.

Als Beispiel betrachten wir die Anweisungsfolge

```
int i, anzahl;
char c;
float f;
double d;
...
anzahl = scanf("%5d %c %f %lf", &i, &c, &f, &d);
```

Dann bewirkt die Eingabezeile

```
12345Y3.14 2.12
```

die Zuweisungen

```
i = 12345;
c = 'Y';
f = 3.14;
d = 2.12;
anzahl = 4;
```

Durch die Eingabezeilen

```
12345 Y3.14 2.12
12345 Y 3.14 2.12
```

ändert sich am Resultat nichts (wegen des Leerzeichens in ␣`%c` !!). Schreiben wir dagegen

```
123456Y3.14 2.12
```

so werden die Zuweisungen

```
i = 12345;
c = '6';
anzahl = 2;
```

ausgeführt, während die Zuweisungen für `f` und `d` unterbleiben.

Beginnt ein Formatbeschreiber mit einem Stern, so wird er zwar bei der Interpretation der Eingabe „normal“ abgearbeitet – der gelesene Wert wird jedoch nicht in einer Eingabevariablen gespeichert. Auch dafür ein Beispiel:

```
int i, o, anzahl;
char c;
float f;
...
anzahl = scanf("%3d zahl %4o %*d %c %f", &i, &o, &c, &f);
```

Mit den Eingabezeilen

```
473 zahl1563 567 R2.33E+2
473 zahl1568 567 R2.33E+2
473z ah l1563 567 R2.33E+2
```

erhalten wir jetzt

```
i = 473;          i = 473;          i = 473;
o = 883;          o = 110;
c = 'R';          c = '5';
f = 233.0;        f = 67;
anzahl = 4;       anzahl = 4;       anzahl = 1;
```

Eine interessante Möglichkeit ist, für Strings die zulässigen oder nicht zulässigen Zeichen explizit anzugeben:

`[`*zeichenfolge*`]`
`[^`*zeichenfolge*`]`

Wir haben in einem Beispiel einmal mit `getchar` in einer Schleife dafür gesorgt, dass aus jeder Eingabe nur die Zahl am Anfang interpretiert und der Rest der Eingabezeile ignoriert wurde:

```
scanf("%d", &wert);          /* eine Zahl lesen */
while (getchar() != '\n')    /* Rest ignorieren */
   ;
```

Jetzt können wir auch kürzer

```
scanf("%d%*[^\n]", &wert); getchar ();
```

schreiben. Achtung: Das Format `%d%*[^\n]%*c` bei gleichzeitigem Verzicht auf den Aufruf von `getchar` ist *nicht* möglich. Wird für den Formatbeschreiber `%*[^\n]` kein zulässiges Zeichen gefunden, so wird die Funktion mit einem Fehler beendet, der Formatbeschreiber `%*c` also gar nicht mehr bearbeitet. Immer dann, wenn das Zeilenende–Zeichen der Zahl unmittelbar folgt, würde es also im Puffer stehenbleiben.

16.5 Funktionen zur Ein–/Ausgabe

Das Funktionenpaar

```
int scanf(const char *format, ...);
int printf(const char *format, ...);
```

kennen wir bereits seit Beginn des Kurses: Es erlaubt den Zugriff auf Standardein– und –ausgabe. Den Zugriff auf beliebige Dateien erlaubt das Funktionenpaar

```
int fscanf(FILE *datei, const char *format, ...);
int fprintf(FILE *datei, const char *format, ...);
```

das am Anfang dieses Abschnitts eingeführt wurde. `scanf` und `printf` dienen ausschließlich der Bequemlichkeit des Programmierers: Der Standard schreibt vor, dass für die Standardein– und –ausgabe die Namen `stdin` bzw. `stdout` vordefiniert sind, so dass man grundsätzlich auch

```
fscanf(stdin, ...);
fprintf(stdout, ...)
```

schreiben könnte. (Es gibt übrigens mit `stderr` noch eine dritte „Standarddatei“ für die Ausgabe von Fehlermeldungen. Sie wird bei interaktiven Programmen in der Regel auch dem Bildschirm zugeordnet.)

Es gibt noch ein drittes, in manchen Fällen interessantes Funktionenpaar, nämlich

```
int sscanf(const char *s, const char *format, ...);
int sprintf(char *s, const char *format, ...);
```

Auch diese Funktionen führen Umwandlungen zwischen interner und externer Darstellung durch, nur greifen sie nicht auf eine Datei zu, sondern statt dessen auf den String, der als erster Parameter übergeben wird. Letztlich werden damit dem Programmierer nur die Umwandlungsroutinen für seine Zwecke zur Verfügung gestellt, die hinter den Formatbeschreibern stehen.

16.6 Vorausschau beim Lesen

Eine weitere Funktion möchte zum Abschluss dieses Abschnitts noch ansprechen, nämlich

```
int ungetc(int c, FILE *datei);
```

Dahinter steht: Bei der Interpretation von Eingabe merkt man in der Regel erst ein Zeichen zu spät, dass man die Interpretation schon hätte beenden müssen. Für solche Fälle ist `ungetc` gedacht: Die Funktion bietet die Möglichkeit, ein Zeichen in die Eingabedatei *zurückzuschreiben*! Liest man die Datei anschließend erneut, so erhält man das zurückgeschriebene Zeichen erneut als Eingabe.

Wir sehen uns dafür ein Beispiel an: Eine Eingabezeile soll eine bestimmte Anzahl von Einträgen enthalten, die nacheinander abzuarbeiten sind. Der Benutzer soll nur die ersten

Einträge angeben müssen; fehlende Angaben sollen durch Standardwerte ersetzt werden. Der Einfachheit halber soll angenommen werden, dass die Einträge Zeichen sind und dass der Standardwert das Leerzeichen sein soll. Bei der Lösung müssen wir darauf achten, dass wir das Zeilenende–Zeichen nicht entfernen, solange noch weitere Eingabezeichen folgen. Realisieren lässt sich das durch die Funktion

```
int naechstes_zeichen(void) {
  int c;
  if ((c = getchar ()) == '\n') {
    ungetc (c, stdin);
    c = ' ';
  }
  return c;
}
```

Erwarten wir etwa 40 Zeichen je Zeile, so können wir diese Funktion durch

```
for (i = 0; i < 40; i++) {
  c = naechstes_zeichen();
  ...
}
getchar ();
```

nutzen. Der Aufruf von `getchar` hinter der Schleife sorgt dafür, dass das Zeilenende–Zeichen der nun vollständig abgearbeiteten Eingabezeile aus der Eingabe entfernt wird. Es gibt übrigens eine Reihe von Restriktionen für `ungetc`, auf die hier nicht näher eingegangen werden kann.

Anhang A

Einführung in UNIX

A.1 Grundlagen

UNIX ist ein *time–sharing–Betriebssystem*. Dieses bedeutet, dass sich mehrere Benutzer an verschiedenen Terminals gleichzeitig die Resourcen eines Rechners (Speicher, Prozessorzeit, Platten, Drucker, usw.) teilen können. Es ist aber auch möglich, dass ein einzelner Benutzer mehrere Prozesse gleichzeitig laufen lassen kann. Wenn mehrere Prozesse eines oder mehrerer Benutzer gleichzeitig laufen, muss das Betriebssystem die Prozessorzeit den einzelnen Prozessen abwechselnd nach einem Prioritätenschema zuweisen. Während die Prozessorzeit also für kurze Intervalle ausschließlich einem einzelnen Prozess zur Verfügung steht, werden andere Resourcen (wie z.B. der Speicher) permanent einem Prozess zugeordnet. Ein Prozess belegt also auch dann Speicher, wenn zwar gestartet wurde, sich aber nicht in der Ausführung befindet.

Jeder Benutzer besitzt einen Benutzernamen und ein hoffentlich nur ihm und dem System bekanntes Passwort. Unter den Benutzern gibt es einen, der unbeschränkte Privilegien besitzt, mit dem Namen *root*. Der Benutzer root kann dem System einen neuen Benutzernamen mit Passwort mitteilen und somit einem neuen Benutzer den Zugang zum System ermöglichen. Genauso kann er den Namen eines Benutzer aus dem System entfernen und somit diesem die Rechenberechtigung entziehen.

A.2 Ein– und Ausloggen, Passwort

Jeder Benutzer muss sich vor jeder Sitzung am Terminal einer Zulassungsprozedur zum Rechnen unterwerfen, dem *login*–Verfahren. Wie diese Prozedur im einzelnen abläuft, hängt sehr stark von der jeweiligen Umgebung ab.

Im wesentlichen werden Benutzername und Passwort abgefragt. Das kann in einer Konsole oder einem grafischen login-Fenster geschehen. Heutzutage werden fast alle Systeme auch eine grafische Benutzeroberfläche anbieten. Dann kann man nach erfolgreichem login über die Oberfläche ein Konsolenfenster öffnen. Der Rest dieses Abschnittes wird weiter auf die Nutzung der Konsole eingehen. Von dort aus kann man alle wichtigen Operationen ausführen. Eine Behandlung von grafischen Benutzeroberflächen ist angesichts derer Vielfalt hier nicht möglich.

Hat man nun Zugriff auf eine Konsole, erscheint dort dann der Textcursor hinter dem Prompt des Systems, das die Form

rechnername>

hat. Das Beenden einer Sitzung erfolgt durch die Eingabe des Kommandos

```
exit ↵
```

oder noch einfacher durch Eingabe von `C-d` im Terminalfenster. Das Zeichen ↵ bedeutet Drücken der `ENTER`–Taste; `C-d` bedeutet, dass man die Taste `Ctrl` gedrückt hält und dazu die Taste `d` anschlägt.

A.3 Hilfen

Die Beschreibung von beliebigen Kommandos kann man sich auf dem Bildschirm mit dem Befehl

`man` *kommandoname* ↵

(*manual*) ansehen. Man erhält dann die entsprechenden Seiten des UNIX–Manuals aufgelistet. So erhält man die Erläuterung des Kommandos `passwd` durch

```
man passwd ↵
```

und die Erläuterung des Kommandos `man` selbst durch

```
man man ↵
```

Die UNIX–Manuals sind in verschiedene Abschnitte aufgeteilt. `man` liefert das erste Ergebnis, egal in welchem Bereich es gefunden wurde. Möchte man speziell z.B. die Dokumentation zu einer Bibliotheksfunktion erhalten, so muss man den Abschnitt mit angeben. Dies geschieht durch eine zusätzliche Zahl vor dem eigentlichen Begriff. Mit

```
man 3 printf ↵
```

erhält man so die Dokumentation zu `printf` aus der C-Standardbibliothek.

A.4 Das Dateisystem

Der gesamte Massenspeicher, über den ein UNIX–System verfügt, ist in Form einer einheitlichen baumartigen Dateistruktur organisiert, in der jeder Benutzer über einen eigenen Teil des Dateibaumes (einen Ast) verfügt. Hier darf der Benutzer Dateien erzeugen, löschen und ausführen; dem Superbenutzer root gehört der gesamte Baum (vgl. Abbildung A.1). Ob der gesamte Dateibaum auf einer einzigen Platte gespeichert oder auf eine Vielzahl von Platten verteilt ist, ist für den Benutzer weitgehend transparent.

Das UNIX–Dateisystem unterscheidet zwischen *Verzeichnissen* und (anderen) *Dateien*. Die Verzeichnisse enthalten andere Verzeichnise und Dateien; Dateien können zum Beispiel Texte oder ausführbare Programme enthalten. Alle inneren Knoten des Dateibaumes, die Wurzel `/` eingeschlossen, sind Verzeichnisse. Die Blätter des Baumes sind (noch) leere Verzeichnisse oder andere Dateien. Man beachte: Verzeichnisse sind letztlich auch Dateien, nur mit ganz speziellem Inhalt.

Die Lage einer Datei (oder eines Verzeichnisses) in der Baumstruktur kann durch ihren *Pfad* beschrieben werden, d.h. den Weg von der Wurzel des Baumes zu der betreffenden Datei. Der *Pfadname* besteht aus der Folge der Namen der dabei durchlaufenen Knoten, wobei die einzelnen Dateinamen durch einen Schrägstrich (`/`) voneinander getrennt werden. Die Wurzel wird nur durch einen Schrägstrich bezeichnet.

Der so angegebene Weg von der Wurzel zu einer Datei ist gleichzeitig ihr vollständiger Name. Zum Beispiel sind die vollen Namen der Dateien `prog.c` und `a.out` im Baumabschnitt von `benutzer2` im Beispiel (vgl. Abbildung A.1)

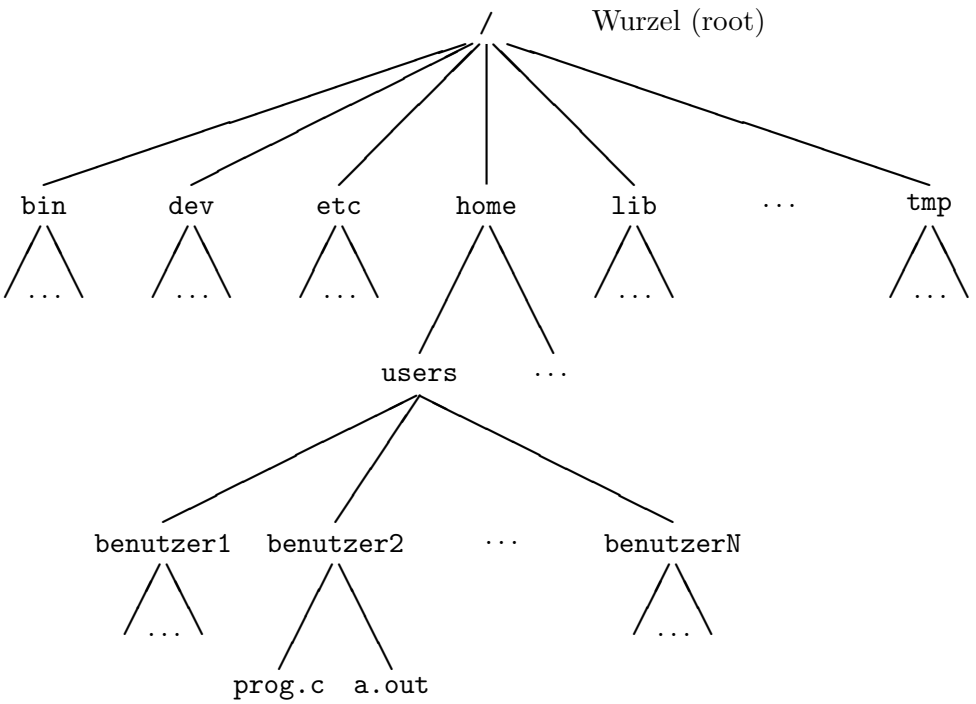

Abbildung A.1: Beispiel eines UNIX–Filesystems

```
/home/users/benutzer2/prog.c
/home/users/benutzer2/a.out
```

Kein Benutzer muss die Namen seiner Dateien stets vollständig angeben, denn das System gibt jedem Dateinamen, der nicht vollständig angegeben wird, ein Präfix, das den Weg von der Wurzel zum aktuellen Baumabschnitt des Benutzers beschreibt. Im Beispiel ist für `benutzer2` das Präfix nach dem Einloggen `/home/users/benutzer2/`. So sind die Eingabe

```
a.out ↵
```

und die Eingabe

```
/home/users/benutzer2/a.out ↵
```

äquivalent. Welches Präfix das System momentan den Dateinamen gibt, lässt sich durch die Eingabe des Kommandos

```
pwd ↵
```

(*print working directory*) feststellen. Das Präfix ist der Pfadname eines Verzeichnisses, das man auch Arbeitsverzeichnis (*working directory*) nennt. Eine Abkürzung dafür ist der Punkt (`.`). So kann man alternativ auch

```
./a.out ↵
```

für obigen Aufruf verwenden. Das kann sogar notwendig sein, wenn nach ausführbaren Programmen nicht im Arbeitsverzeichnis gesucht wird.

Auf eine andere Möglichkeit, ein Präfix vom System automatisch ergänzen zu lassen, wird gleich noch näher eingegangen.

A.5 Dateiverwaltung

Das momentan gültige Namenspräfix kann man durch das Kommando

`cd` *pfadname* ↵

(*change directory*) ändern, wobei *pfadname* den Weg von der Wurzel zu einem Verzeichnis beschreibt. (Bei der Ausführung des Kommandos wird der angegebene Pfadname ggf. noch durch das alte Präfix ergänzt.)

Mit dem Kommando

```
cd .. ↵
```

wird das gegenwärtige Präfix um den letzten Namen gekürzt. Dieses entspricht einer Bewegung in der Richtung zur Wurzel im Dateibaum. Mit dem Kommando

`cd` *name* ↵

wird das gegenwärtige Präfix um den angegebenen Namen erweitert, sofern in der gegenwärtigen Verzeichnisdatei eine Verzeichnisdatei mit diesem Namen vorhanden ist. Dieses entspricht einer Bewegung in der Richtung weg von der Wurzel im Dateibaum.

Nach dem Einloggen befindet man sich in seinem Heimatverzeichnis (*home directory*), d.h. in der Wurzel des eigenen Teilbaumes. Für `benutzer2` heißt das etwa, dass nach dem Einloggen

```
/home/users/benutzer2/
```

das aktuelle Präfix ist. Dieses Präfix kann man jederzeit wiederherstellen, indem man nur

```
cd ↵
```

eintippt. Für das Heimatverzeichnis eines Benutzers gibt es auch noch eine spezielle Kurzform, nämlich

```
~benutzer2/
```

Das System bestimmt dann selbst, wie der Pfad zu diesem Verzeichnis korrekt zu ergänzen ist.

Welche Dateien ein Verzeichnis mit dem Namen *name* enthält, lässt sich mit dem Kommando

`ls` *name* ↵

(*list*) feststellen. Beispiele:

```
ls / ↵
ls /home ↵
ls /home/users/benutzer2 ↵
ls ~benutzer2 ↵
```

Das erste Kommando zeigt alle Einträge in der Wurzel, das zweite alle Einträge im Verzeichnis `/home`, das dritte und vierte alle Einträge im home–Verzeichnis des Benutzers `benutzer2`.

Gibt man nur das Kommando

```
ls ↵
```

ein, so werden alle Einträge im aktuellen Verzeichnis angezeigt, d.h. dem Verzeichnis, dessen Pfadname durch `pwd` geliefert wird.

Mit der Option `-l` (*long*)

```
ls -l ↵
```

wird für jeden Eintrag des Verzeichnisses mehr Information ausgegeben: Jede Zeile enthält Angaben zu einer Datei, deren Name am Ende der Zeile steht. Die ersten 10 Zeichen haben die Form

```
d rwx rwx rwx
```

und bedeuten:

- `d` es handelt sich um ein Verzeichnis
- `r` die Datei darf gelesen werden
- `w` in die Datei darf geschrieben werden
- `x` die Datei darf (als Programm) ausgeführt oder (als Verzeichnis) aufgelistet werden

Steht an der Position von `d` ein Bindestrich (`-`), so handelt es sich *nicht* um ein Verzeichnis. An den anderen Stellen bedeutet ein Bindestrich, dass das entsprechende Zugriffsrecht *nicht* besteht. Die drei Gruppen `rwx` markieren, von links nach rechts, die Zugriffsrechte für den Besitzer der Datei, für seine Gruppe und für andere Benutzer.

Die Zugriffsrechte, also der Schutz der Dateien, können mit dem Kommando `chmod` (*change mode*) geändert werden. Sehen Sie sich die Einzelheiten einmal selbst mit dem Kommando `man` an.

Mit der Optionenkombination `-al`

`ls -al` ↵

werden auch Dateien erfasst, deren Name mit einem Punkt beginnt und die sonst nicht aufgelistet werden.

Neue Verzeichnisse kann man mit dem Kommando

`mkdir` *name* ↵

(*make directory*) anlegen, leere Verzeichnisse mit dem Kommando

`rmdir` *name* ↵

(*remove directory*) löschen, entsprechende Zugriffsrechte vorausgesetzt. Andere Dateien kann man mit

`rm` *name1 name2 ... nameN* ↵

(*remove*) löschen, ebenfalls entsprechende Zugriffsrechte vorausgesetzt. Beim Löschen ist oft die Option `-i`

`rm -i` *name1 name2 ... nameN* ↵

zweckmäßig. Sie bewirkt, dass man für jede einzelne Datei das Löschen durch Eintippen von `y`(es) noch einmal ausdrücklich bestätigen muss.

Kopieren bzw. Umbenennen kann man Dateien mit den Kommandos

`cp` *quelle ziel* ↵

(*copy*) bzw.

`mv` *quelle ziel* ↵

(*move*). Durch Umbenennen kann eine Datei auch aus einem Verzeichnis in ein anderes verlegt werden.

A.6 Metazeichen

Bei vielen Kommandos ist es zweckmäßig, Dateinamen nicht explizit anzugeben, sondern die *Metazeichen* (*Jokerzeichen*, *wild cards*) Fragezeichen (`?`) und Stern (`*`) zu verwenden:

- Ein Fragezeichen steht für genau ein beliebiges Zeichen.
- Ein Stern steht für eine beliebig lange (auch leere) Folge beliebiger Zeichen.

Beispiele:

```
rm *.c ↵
rm *~ ↵
rm * ↵
rm -i * ↵
rm aufg?.c ↵
```

Setzen wir voraus, dass die entsprechenden Zugriffsrechte bestehen, so gilt: Im ersten Beispiel werden in der aktuellen Directory alle Dateien mit dem Suffix `.c` gelöscht und im zweiten Beispiel alle Dateien, bei denen das letzte Zeichen des Namen `~` ist; im dritten Beispiel werden in der aktuellen Directory **alle** Dateien ausnahmslos und ohne Rückfrage gelöscht; im vierten Beispiel werden in der aktuellen Directory alle Dateien gelöscht, für die dieses ausdrücklich bestätigt wird; im letzten Beispiel werden in der aktuellen Directory alle Dateien gelöscht, deren Name mit `aufg` beginnt, dann ein beliebiges Zeichen aufweist und danach mit `.c` endet.

A.7 Auflisten von Datei–Inhalten

Durch

`cat` *datei* ↵

wird der Inhalt der entsprechenden Datei aufgelistet. Dieses Kommando empfiehlt sich allerdings nur für ziemlich kurze Dateien, da die Ausgabe „durchrollt“, wenn sie länger als eine Bildschirmseite ist. Seitenweise kann man sich den Inhalt einer Datei mit dem Kommando

`more` *datei* ↵

ansehen: Die Ausgabe stoppt, sobald der Bildschirm voll ist; durch Drücken der Leertaste wird die nächste Bildschirmseite gezeigt, durch Drücken der `ENTER`–Taste nur die nächste Zeile.

Zur Ausgabe auf einen Drucker dient das Kommando

`lpr` *datei* ↵

(*lineprinter*) oder, wenn verschiedene Drucker zur Verfügung stehen

`lpr -P`*xyz datei* ↵

wobei *xyz* für den Namen des Druckers steht. Leider kommt es immer wieder vor, dass Drucker durch Fehlbedienung blockiert werden. Den Status eines Druckers kann man sich mit dem Kommando

`lpq -P`*xyz* ↵

(*lineprinter queue*) ansehen. Stellt man fest, dass die Warteschlange des Druckers nicht leer ist, der Drucker aber trotzdem nicht arbeitet, kann es daran liegen, dass er blockiert ist. Falls eine eigene Datei der „Übeltäter“ ist, kann man versuchen, die Datei aus der Warteschlange zu löschen. Hierzu dient das Kommando

```
lprm -Pxyz # ↵
```

(*lineprinter remove*), bei dem man als Nummer die Zahl einzusetzen hat, die das Kommando `lpq` für die Datei gemeldet hat.

A.8 Umleitung von Ein– und Ausgabe

UNIX–Kommandos holen ihre Eingabe vielfach vom *Standardeingabegerät* (Tastatur) und schreiben ihre Ausgabe vielfach auf das *Standardausgabegerät* (Bildschirm).

Die Standardein–/ausgabe lässt sich aber beim Aufruf eines Kommandos ohne weiteres umleiten.

programm `<` *name*

bewirkt, dass die Datei (oder das Gerät) mit dem Namen *name* als Standardeingabegerät verwendet wird;

programm `>` *name*
programm `>>` *name*

bewirken, dass die Datei (oder das Gerät) mit dem Namen *name* als Standardausgabegerät verwendet wird.

Beide Formen unterscheiden sich nur dann, wenn die Datei, in die die Ausgabe umgeleitet wird, bereits existiert: Bei einem Winkel wird der bisherige Inhalt der Datei zunächst gelöscht; bei zwei Winkeln wird die Ausgabe an den bisherigen Inhalt der Datei angehängt.

Umleitung von Ein– und Ausgabe ist bei vielen Kommandos möglich. Besonders interessant ist die Umleitung für eigene Programme. Man kann für umfangreichere Tests die benötigte Eingabe in einer Datei speichern und diese als Eingabequelle verwenden. So muss die Eingabe nicht jedesmal von Hand erneut eingegeben werden.

A.9 Editieren

Meist hat man die Wahl zwischen verschiedene *Editoren*, d.h. Programmen, mit denen man Texte, zum Beispiel auch C–Quellprogramme eingeben kann. Wichtig ist, dass der Editor den eingegebenen Quelltext ohne Formatierungsanweisungen oder Anweisungen zur Dokumentorganisation speichert. Ein normales Textverarbeitungsprogramm, mit dem man z.B. seine Briefe verfasst ist daher in der Regel nicht geeignet. Mit der Zeit entwickelt sicher jeder eine Vorliebe für einen bestimmten Editor.

Es würde wenig bringen, an dieser Stelle zu versuchen, die Funktionsweise eines Editors zu erklären oder auf spezielle Funktionen eines bestimmten Editors einzugehen. Neben den oft vorinstallierten Editoren gibt es eine Reihe freier Editoren, die man konstenlos im Internet herunterladen kann. Es lohnt sich auf jeden Fall zunächst einige verschiedene Editoren auszuprobieren und sich erst dann zu entscheiden.

A.10 Übersetzen und Binden

Prinzipiell sind zwei Schritte erforderlich sind, um aus dem Quellcode eines C–Programms ein ausführbares Programm zu erzeugen:

1. Der Quellcode wird mit dem *Compiler* übersetzt. Der Compiler erzeugt eine *Objektdatei*.
2. In der Regel werden Bibliotheksroutinen benötigt. Die Objektdatei und diese Routinen werden durch den *Linker* gebunden.

Dazu benötigt man einen Compiler und Linker. Auf Linux-basierten Systemen ist meist die GNU Compiler-Sammlung (GNU Compiler Collection, GCC, *http://gcc.gnu.org*) zusammen mit der GNU C Bibliothek (GNU C Library, glibc, *http://www.gnu.org/software/libc*) installiert. Beide kann man kostenlos im Internet herunterladen.

Die beide Schritte – compilieren und linken – werden vom `gcc` mit einem einzigen Kommando bewirkt:

```
gcc prog.c
```

Dabei ist `prog.c` der Name der C–Quelldatei; das Suffix `.c` zeigt dem Compiler an dass es sich um C–Quellcode handelt. Das ausführbare Programm erhält den Namen `a.out`. Die Eingabe dieses Namens als UNIX–Kommando bewirkt die Ausführung des Programms.

Soll das ausführbare Programm einen anderen Namen als `a.out` erhalten, zum Beispiel `prog`, so muss man die Option `-o` und den neuen Namen zusätzlich in das `gcc`–Kommando eintragen:

```
gcc -o prog prog.c
```

Eine zusätzliche Option sollte man beim Aufruf des `gcc` auf jeden Fall angeben, nämlich `-Wall`, so dass der Aufruf

```
gcc -Wall [-o prog] prog.c
```

ist. Ist diese Option angegeben, schreibt der Compiler zu „verdächtigem“ Code, den er sonst kommentarlos akzeptieren würde, Warnungen. In vielen Fällen weisen diese Warnungen auf Programmierfehler hin, ohne deren Beseitigung das Programm nicht laufen wird. Merke: Ein Programm, das ein Compiler widerspruchslos akzeptiert, ist noch lange kein korrektes Programm!

Die Option `-O` optimiert den erzeugten Maschinencode hinsichtlich Größe und Geschwindigkeit. Die Ausführung eines Programms kann so deutlich beschleunigt werden. Der Compiler kann aber auch mehr Warnungen ausgeben. Durch die bei der Optimierung verwendete Datanfluss–Analyse wird zum Beispiel auch erkannt, ob eine Variable ggf. uninitialisiert verwendet wird.

Zwei weitere Optionen, nämlich `-ansi` und `-pedantic`, unterstützen den Programmierer, der sauberes Standard–C programmieren und keine Erweiterungen des `gcc` verwenden möchte.

Der normale einfache Compileraufruf sieht also so aus:

```
gcc -Wall -ansi -pedantic -O [-o prog] prog.c
```

Eine weitere oft benötigte Option ist `-lm`. Sie sorgt dafür, dass der Bibliotheksteil mit den mathematischen Funktionen gebunden wird. Da es sich um eine Linker-Option handelt, wird sie nach den Quelltextdateien angegeben.

Anhang B

Die C-Standardbibliothek

Die Header-Dateien der C-*Standardbibliothek* sind hier vollständig aufgezählt, nicht jedoch die der enthaltenen Elemente. Hier sollen nur die (subjektiv) wichtigsten Konstanten, Typen, Funktionen und Variablen aufgelistet werden. Auf UNIX/LINUX-Systemen sollte `man 3` ***`fkt`*** bzw. `man 3` ***`datei`*** die Dokumentation zur Funktion ***`fkt`*** bzw. zur Headerdatei ***`datei`***`.h` der Standardbibliothek liefern.

- `assert.h`
  debugging-Hilfen:

  Makro `NDEBUG` - Funktion `assert`

- `ctype.h`
  Funktionen zur Klassifizierung/Umwandlung von Zeichen

  Argument und Funktionswert der Funktionen sind jeweils vom Typ `int`. Hat das übergebene Argument den Wert EOF, so ist der Funktionswert bei allen Funktionen EOF. Wird die Funktion mit einer `char`-Variablen als Argument aufgerufen, so wird deren Wert implizit in einen `int`-Wert umgewandelt und es werden folgende Werte zurückgeliefert:

  - `int isalnum(int c)`
    - ungleich 0, falls `c` ein alphanumerisches Zeichen repräsentiert (Ziffer oder Buchstabe), 0 sonst.
  - `int isalpha(int c)`
    - ungleich 0, falls `c` Buchstaben repräsentiert, 0 sonst.
  - `int iscntrl(int c)`
    - ungleich 0, falls `c` ein Steuerzeichen repräsentiert (siehe Zeichensatz), 0 sonst
  - `int isdigit(int c)`
    - ungleich 0, falls `c` Ziffer repräsentiert, 0 sonst.
  - `int isgraph(int c)`
    - ungleich 0, falls `c` druckbares Zeichen (ohne Leerzeichen) repräsentiert, 0 sonst.
  - `int islower(int c)`
    - ungleich 0, falls `c` Kleinbuchstaben repräsentiert, 0 sonst.
  - `int isprint(int c)`
    - ungleich 0, falls `c` druckbares Zeichen (mit Leerzeichen) repräsentiert, 0 sonst.
  - `int ispunct(int c)`
    - ungleich 0, falls `c` ein druckbares Sonderzeichen repräsentiert, 0 sonst.

  - `int isspace(int c)`
    - ungleich 0, falls `c` *Leerraum* repräsentiert (also Leerzeichen oder eines der Zeichen `\f, \n, \r, \t, \v`), 0 sonst
  - `int isupper(int c)`
    - ungleich 0, falls `c` Großbuchstaben repräsentiert, 0 sonst.
  - `int isxdigit(int c)`
    - ungleich 0, falls `c` hexadezimale Ziffer repräsentiert, 0 sonst.
  - `int tolower(int c)`
    - Falls der Argumentwert ein Wert aus dem Bereich `'A'` bis `'Z'` ist, so wird der entsprechende „Kleinbuchstabenwert“ (aus `'a'` bis `'z'`) zurückgegeben. Anderenfalls wird der Argumentwert zurückgegeben.
  - `int toupper(int c)`
    - wie `tolower` nur umgekehrt: zu „Kleinbuchstabenwert“ wird (soweit möglich) entsprechender „Großbuchstabenwert“ zurückgegeben

- `errno.h`
  Fehlerbehandlung für mathematische Funktionen:

  Makro `EDOM` - Makro `ERANGE` - globale Variable `errno`
- `float.h`
  Angaben zu den Gleitkommazahlwertebereichen:

  `FLT_MAX` -

  `FLT_MIN` -

  `DBL_MAX` -

  `DBL_MIN` -

  `LDBL_MAX` -

  `LDBL_MIN`
- `iso646.h`
  alternative Schreibweisen für logische Operatoren
- `limits.h`
  Angaben zu den Ganzzahlwertebereichen:

  `CHAR_BIT` - `CHAR_MAX` - `CHAR_MIN` - `SCHAR_MAX` - `SCHAR_MIN` - `UCHAR_MAX` - `SHRT_MAX` - `SHRT_MIN` - `USHRT_MAX` - `INT_MAX` - `INT_MIN` - `UINT_MAX` - `LONG_MAX` - `LONG_MIN` - `ULONG_MAX`
- `locale.h`
  Sprach-, Schrift-, Zeitrechnungs- und währungsspezifische Angaben
- `math.h`
  Gleitkomma-Arithmetik:

  Makro `HUGE_VAL`, Funktionen: `fabs` - `ceil` - `floor` -

  `acos` - `asin` - `atan` -

  `cos` - `sin` - `tan` - `cosh` - `sinh` - `tanh` - `exp` -

  `log` - `log10` -

  `pow` - `sqrt`
- `setjmp.h`
  Sprünge in andere Programmteile

- `signal.h`
  Signal(Interrupt)behandlung
- `stdarg.h`
  Variable Argumentlisten für Funktionen mit einer variablen Anzahl von Parametern (wie z.B. `printf` und `scanf`):

  `va_list` - `va_start` - `va_arg` - `va_end`
- `stddef.h`
  Definitionen von einfachen Typen und Werten
  - `NULL`
    - der „Nullzeiger“, eine benannte Konstante vom Zeigertyp deren Wert verschieden ist von jedem ëchtenSZeigerwert
  - `size_t`
    - vorzeichenloser ganzzahliger Typ dessen Wertebereich alle zulässigen Speicherbereichsgrößen umfasst (m.a.W.: Menge der Werte, die vom `sizeof`-Operator (in der jeweiligen Implementation) zurückgeliefert werden können)
- `stdio.h`
  Ein- und Ausgabe-Funktionen (siehe die Kapitel des Skripts zur Ein- und Ausgabe)
- `stdlib.h`
  Umwandlung von Strings zu Zahlwerten:

  `atof` - `atoi` - `atol` - `strtod` - `strtol` - `strtoul`

  Pseudozufallsgenerator:

  `RAND_MAX` - `rand` - `srand`

  dynamische Speicherverwaltung:

  `malloc` - `calloc` - `realloc` - `free`

  ganzzahlige Mathematik:

  `div_t` - `ldiv_t` - `abs` - `div` - `labs` - `ldiv`

  Programmabbruch:

  `EXIT_FAILURE` - `EXIT_SUCCESS` - `exit` - `abort` - `atexit`

  Systemaufrufe:

  `system` - `getenv`

  Algorithmen:

  `bsearch` - `qsort`

  Umwandlungen zwischen verschiedenen Zeichensätzen
- `string.h`
  Stringbearbeitung
  - `char *strerror(int Kennzahl);`
    - liefert den Zeiger auf den Anfag des Strings, der die Klarschrift-Fehlermeldung zum Fehler mit dem Fehlercode `Kennzahl` enthält
  - `size_t strlen(const char *s);`
    - liefert die Anzahl der Zeichen des Strings, auf dessen Anfang `s` zeigt. Das Stringende-Zeichen des Strings wird dabei nicht mitgezählt.

- `char *strcpy(char *s1, const char *s2);`
  - kopiert (byteweise) die Zeichen des Strings auf dessen Anfang `s2` zeigt (inklusive Stringende-Zeichen) in einen Speicherbereich auf dessen Anfang `s1` zeigt. Der Wert von `s1` wird als Funktionswert zurückgeliefert.
- `char *strncpy(char *s1, const char *s2, size_t n);`
  - wie `strcpy`, jedoch werden *genau* `n` Zeichen geschrieben. Außerdem wird das Kopieren beendet, sobald das Stringende-Zeichen im Quellstring gefunden wurde und ergänzt danach nur noch (binäre) Nullen. Das bedeutet insbesondere: Wenn der zu kopierende String aus `n` oder mehr Zeichen besteht, wird der kopierte String nicht durch ein Stringende-Zeichen abgeschlossen. Der Wert von `s1` wird als Funktionswert zurückgeliefert.
- `char *strcat(char *s1, const char *s2);`
  - kopiert den String auf dessen Anfang `s2` zeigt hinter den String auf dessen Anfang `s1` zeigt. Das Stringende-Zeichen von `s1` wird dabei überschrieben. Hinter dem letzten kopierten Zeichen wird das Stringende-Zeichen automatisch angehängt. Der Wert von `s1` wird als Funktionswert zurückgeliefert.
- `char *strncat(char *s1, const char *s2, size_t n);` - wie `strcat`, jedoch wird das Kopieren vorzeitig beendet, sobald `n` Zeichen übertragen wurden.
- `int strcmp(const char *s1, const char *s2)`
  - liefert einen `int`-Wert kleiner, gleich oder größer Null, je nachdem, ob der String, auf dessen Anfang `s1` zeigt, *lexikographisch* kleiner, gleich oder größer ist als der String, auf dessen Anfang `s2` zeigt. [Ein String heißt lexikographisch kleiner als ein anderer, wenn in ihm an der ersten Unterscheidungsstelle ein kleinerer `char`-Wert steht als im anderen String.
- `int strncmp(const char *s1, const char *s2, size_t n);`
  - wie `strcmp`, jedoch werden höchstens `n` Zeichen verglichen
- `char *strchr(const char *s, int c)`
  - liefert Zeiger auf das zuerst gefundene Vorkommen des Zeichens `c` in dem String, auf dessen Anfang `s` zeigt bzw. den Nullzeiger, falls das Zeichen nicht gefunden wurde.
- `strrchr`
  - wie `strchr`, jedoch wird beginnend beim Stringende-Zeichen in Richtung Anfang gesucht.
- `strspn`
- `strcspn`
- `strpbrk`
- `char *strstr(const char *s1, const char *s2)`
  - sucht das erste Vorkommen des Strings auf dessen Anfang `s2` zeigt, im String, auf dessen Anfang `s1` weist (dabei wird das Stringende-Zeichen nicht mit verglichen). Der Funktionswert ist der Zeiger auf den Anfang der gefundenen Teilfolge bzw. der Nullzeiger.
- `strtok`

Speicherbearbeitung

- `memchr`
- `memcmp`

  - `void *memcpy(void *s1, const void *s2, size_t n)`
    - kopiert die ersten `n` Bytes eines Speicherbereichs auf dessen Anfang `s2` zeigt in die ersten `n` Bytes des Speicherbereichs auf dessen Anfang `s1` zeigt. Funktionswert ist `s1`.
  - `memmove`
  - `memset`

- `time.h`
  Zeitmessung:

  `CLOCKS_PER_SEC` - `clock_t` - `clock`

  Datums- und Uhrzeitbestimmung:

  `time_t` - `time` - `difftime` - `struct tm` - `gmtime` - `localtime` - `asctime` - `ctime`

- `wchar.h`
  Umwandlung von Strings zu Zahlwerten für den erweiterten Zeichensatz
  String- und Speicherbearbeitung für den erweiterten Zeichensatz
  Ein- und Ausgabe für den erweiterten Zeichensatz

- `wctype.h`
  Zeichenuntersuchung für den erweiterten Zeichensatz

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextbeispiele

Literaturverzeichnis

[1] C Programming Language. Wikipedia-Eintrag: `http://en.wikipedia.org/w/index.php?title=C_programming_language&oldid=40429557`.

[2] GNU Compiler Collection (GCC). Internet-Website: `http://gcc.gnu.org/`.

[3] Joachim Goll, Ulrich Bröckl, and Manfred Dausmann. *C als erste Programmiersprache*. Teubner, 2003.

[4] Samuel P. Harbison and Guy L. Steele Jr. *C. A Reference Manual*. Prentice Hall, 2002.

[5] Sammlung von Standardisierungsdokumenten. Internet-Website: `http://www.open-std.org/JTC1/SC22/WG14/www/standards`, Juni 2005.

[6] ISO/IEC 9899:TC2. Internet-Dokument: `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`, Mai 2005. ISO/IEC.

[7] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.

[8] Linksammlung zur C-Programmierung. Internet-Website: `http://www.lysator.liu.se/c/`, 2001.

[9] Martin Lowes and Augustin Paulik. *Programmieren mit C. ANSI Standard*. Teubner, 1999.

Stichwortverzeichnis

V

W

Z