Markt&Technik

Peter Rosenbeck

# Programmierung unter TOS

# ATARI ST

Einführung in die Programmiersprache C

★ Systemprogrammierung am Beispiel eines Diskettenmonitors

★ Einsatz von BIOS-Routinen

★ Software-Engineering

C-Programmierung unter TOS
ATARI ST

Peter Rosenbeck

# C-Programmierung unter TOS ATARI ST

- Einführung in die Programmiersprache C
- Systemprogrammierung am Beispiel eines Diskettenmonitors
- Einsatz von BIOS-Routinen
- Software-Engineering

Markt&Technik Verlag

CIP-Kurztitelaufnahme der Deutschen Bibliothek

**Rosenbeck, Peter:**
C-Programmierung unter TOS, ATARI ST : Einf. in d. Programmiersprache C ;
Systemprogrammierung am Beispiel e. Diskettenmonitors ;
Einsatz von BIOS-Routinen ; Software engineering / Peter Rosenbeck. –
Haar bei München : Markt-und-Technik-Verlag, 1986.
ISBN 3-89090-226-X



15 14 13 12 11 10 9 8 7 6 5 4 3
89 88 87 86

ISBN 3-89090-226-X


Einbandgestaltung: Grafikdesign Heinz Rauner
Druck: Kösel, Kempten
Printed in Germany

# Inhaltsverzeichnis

# 0 Vorwort: Was sollten Sie schon wissen?

Es gehört beinahe schon zum Einleitungszeremoniell vieler Computer-Bücher, das Unmögliche zu versprechen: nämlich daß (ein fiktives Beispiel) mit dem vorliegenden Buch der absolut blutige Laie, gerade des Lesens und Schreibens mächtig, in drei leicht verdaulichen Lektionen auch noch die letzten Geheimnisse der Systemprogrammierung beigebracht bekommt. Das Buch, das diesen Anspruch einlöst, muß allerdings erst noch geschrieben werden.

Um mich nicht dieser zweifelhaften Praxis anzuschließen, will ich gleich zu Beginn das Risiko auf mich nehmen und eventuell ein paar Käufer verprellen: wenn Sie absoluter EDV-Anfänger sind, "Byte" für ein irrtümlich großgeschriebenes englisches Tätigkeitswort halten, bei "Adresse" zuerst mal an "Straße" denken und "Diskette" mit "Diskus" und "Olympiade" assoziieren - tja, dann sollten Sie sich das Geld für dieses Buch sparen.

Wer sich jetzt nicht angesprochen fühlte, der möge weiterlesen. Beobachtet man den Bücher- und vor allem den Zeitschriften-Markt auf dem EDV-Sektor, dann kann man sehen, daß die deutschen Computer-Fans der Pubertät zu entwachsen beginnen: das Niveau steigt, und entsprechend ist nicht mehr nur alles BASIC, BASIC, BASIC. In Leserbeiträgen von Computerzeitschriften findet man Programme von oft erstaunlich hohem Niveau, und die Zeitschriften würden sowas nicht veröffentlichen, wenn nicht die Nachfrage dafür gegeben wäre. Das war nicht immer so; die hunderte von BASIC-Bücher mit immer der gleichen Strickart (FOR-Schleife, PRINT und als krönender Abschluß ein einfaches Spiel zum Abtippen) belegen es.

Die Hobbyisten beginnen sich also zu emanzipieren. Das Interesse an moderneren Entwicklungen innerhalb der Informatik wächst, man ist zunehmend bereit, sich auch mit anderen Programmiersprachen auseinanderzusetzen. Das Wissen über Computer, deren Funktionsweise und Möglichkeiten hat zugenommen, kurz: der Bildungsgrad ist gestiegen und als Fachautor muß man nicht immer bei Adam und Eva anfangen.

Diesen EDV-gebildeten Leserkreis möchte ich mit meinem Buch erreichen. Der ideale Leser sollte die ein- oder andere Fachzeitschrift verfolgen - möglichst eine, die sich nicht auf das Abschreiben von Produktankündigungen beschränkt -, vom Jargon der Branche nicht verschreckt sein, schon mal mit einem Computer gearbeitet haben und - das ist das wichtigste - neugierig sein. Kenntnisse einer anderen Programmiersprache sind nicht unbedingt nötig, wenn auch gelegentlich nützlich. Insbesondere nehme ich

nicht an, daß Sie bereits einmal mit einer anderen Compilersprache gearbeitet haben; was ein Compiler ist, wie er funktioniert und wozu er taugt, das wird ausführlich erläutert. Bestimmte Grundkenntnisse der Computerei (Was ist eine Variable? Was ist eine Schleife?) sind jedoch sicher von Nutzen. Zwar werde ich alle zum Verständnis nötigen Konzepte einführen, aber oft ist es ganz hilfreich, etwas bereits Bekanntes noch einmal in anderer Darstellung zu lesen.

Gegenstand dieses Buches ist die C-Programmierung. Alle Programmiersprachen verwenden ein dem Englischen entlehntes Vokabular. Auch viele Fachausdrücke der Computerwissenschaft sind dem Englischen entlehnt. Ein deutscher Autor hat hier zwei Möglichkeiten: die konsequente Eindeutschung der Begriffe, oder die weitgehende Übernahme der englischen Original-Ausdrücke. Ich habe mich für den zweiten Weg entschieden.

Einmal sind die mir bekannten Eindeutschungen meist ziemlich unbeholfen. Zum anderen gehen sie an der Realität vorbei: außer deutschen Professoren spricht so kein Mensch. Unter Programmierern hat sich ein deutsch-englischer Slang herausgebildet, eine Art Fliegerdeutsch, das für den Außenstehenden oft schwer zu durchschauen ist. Neben der Beherrschung der Programmiersprache ist es jedoch wichtig, diesen Slang zu kennen, um mitreden (und mitlesen) zu können. Deshalb bediene ich mich in diesem Buch des Programmierer-Jargons, weise aber bei jedem neu eingeführten Begriff auf seine Bedeutung hin; Sie müssen also nicht Englisch können.

Um das Buch sinnvoll nutzen zu können, benötigen Sie Zugang zu einem C-Compiler. Nun ist C eine ziemlich standardisierte Sprache und es dürfte keine Rolle spielen, welchen Compiler von welchem Hersteller Sie benutzen. Die Ausführungen über C in diesem Buch sind möglichst allgemein gehalten und sollten für eine Vielzahl von Compilern gelten. Zum Zeitpunkt der Erstellung dieses Buchs war jedoch im wesentlichen ein Compiler für den ATARI ST im Umlauf: der mit dem Entwicklungssystem ausgelieferte C-Compiler von Digital Research.

Dieser Compiler wies einige (vor allem für den Anfänger) z.T. sehr unangenehme Fehler auf, die in späteren Versionen sicher behoben werden. Dennoch werde ich gelegentlich auf die Besonderheiten dieses Compilers eingehen, da - wenn Sie mit ihm arbeiten - nicht sicher ist, ob Sie die neueste Version besitzen, oder ob auch wirklich alle Fehler aus dem Compiler entfernt wurden.

Das Buch ist in zwei Teile gegliedert; der erste Teil ist eine vollständige und ausführliche Einführung in die Sprache C. In diesem Teil werden alle Sprachkonstrukte in Einzelbeispielen besprochen, wie das eben so bei Einführungen üblich ist. Ich habe besonderen Wert darauf gelegt, auch die

fortgeschritteneren Konzepte von C (dynamische Datenstrukturen und Funktions-Pointer) besonders ausführlich zu behandeln; dies auch deshalb, weil nur durch ihr Verständnis die anspruchsvolle GEM-Programmierung, der ich ein eigenes Buch widmen werde, angegangen werden kann.

Aber kurze Einzelbeispiele reichen alleine - auch wenn die Erklärung noch so ausführlich ist - nicht aus, um den Umgang mit einer Sprache zu beherrschen. Deshalb der zweite Teil, der aus einem ausführlich dokumentierten und kommentierten längeren Programm, einem Diskmonitor, besteht. Die Programmierung eines Diskmonitors ist bestens geeignet, um die systemnahe Programmierung, das direkte Arbeiten mit den Funktionen des Betriebssystems, zu erlernen. Da die Aufgabenstellung verhältnismäßig komplex ist, können Sie an ihrer Lösung die Prinzipien modularer strukturierter Programmentwicklung einüben.

Sie werden feststellen: C ist eine elegante, professionelle Sprache, mit der auf dem ATARI zu arbeiten großen Spaß macht!

# Teil I:
# Einführung in C

# 1 Zum Warmwerden

Für viele Leser bedeutet das Erlernen von C auch den ersten Kontakt mit einer Compilersprache. Da werden sie feststellen, daß vor die Beherrschung der Programmiersprache die Beherrschung des Compilers gestellt ist. Deshalb legt dieses Kapitel dar, was ein Compiler ist, worin sein Zweck besteht und wie er funktioniert. Wer das schon alles weiß, der findet darin ferner einen Vergleich zwischen C und anderen Compiler-Sprachen. Abschließend zeige ich Ihnen alles, was Sie für den Umgang mit einem speziellen C-Compiler wissen müssen am Beispiel des C-Compilers, der mit dem ATARI-Entwicklungssystem ausgeliefert wird.

## 1.1 C ist eine Compilersprache

Sie wissen, daß Computer eigentlich nur in ihrer Muttersprache mit sich reden lassen. Und Sie wissen, daß diese Muttersprache sich eines äußerst dürftigen Alphabets bedient, das nur aus zwei Zeichen besteht: '0' und '1'. Diese Zeichen, auch "Bits" genannt, bilden die Grundlage des unter Computern gesprochenen "Binärcodes"; weil er so wichtig ist, wird der Binärcode Sie später noch etwas mehr beschäftigen.

Vorerst interessiert jedoch nur die Tatsache, daß die Umgangssprache der Computer herzlich wenig mit der des Menschen zu tun hat. Sie basiert auf dem Binärcode und wird - wie naheliegend! - "Maschinensprache" genannt. Seine Wünsche der Maschine in einer endlosen Folge von Nullen und Einsen mitteilen zu müssen (so arbeiteten die ersten Programmierer tatsächlich!) ist nicht nur mühsam, sondern auch fehleranfällig. Der Mensch in seiner Faulheit - diese ist eben oft die Triebfeder für große Erfindungen - sann auf Abhilfe. Erstmal brachte er die Maschinensprache in eine etwas gefälligere und entschärfte Form, die sogenannte "Assemblersprache". Die arbeitet nicht mehr mit dem Binärcode, sondern mit sprechenden Befehlsnamen. Da aber ihre Befehle völlig auf die 'Denkweise' der Maschine ausgerichtet sind (sie spiegeln unmittelbar das wieder, was die Maschine kann), ist das Arbeiten mit ihr immer noch sehr mühsam und fehlerbehaftet.

Dann aber erfand der Mensch die höheren Programmiersprachen. Der Trick dabei: man kann den Computer zwar nicht von seiner Maschinensprache abbringen, man kann ihm jedoch mittels Maschinensprache beibringen, wie er in einem andern Code verfaßte Befehle erstmal in die ihm genehme

Weise bringen und dann befolgen kann. Zwischen der Sprache, mit der der Programmierer arbeitet und der, die der Computer versteht, wird ein Übersetzungsprogramm eingeschaltet.

Dieses Programm spielt also die Rolle eines Dolmetschers zwischen Mensch und Maschine. Da es für den Programmierer dank seiner Hilfe so aussieht, als würde der Computer jetzt eine andere Sprache sprechen, setzt man kurzerhand den Übersetzer mit der Sprache, die er dolmetscht, gleich. Es hat sich dafür der Ausdruck "höhere Programmiersprache" eingebürgert. "Höher" deshalb, weil diese Sprache sich näher an die menschliche Ausdrucksweise anlehnt als die Maschinensprache, die entsprechend eine 'niedere' Programmiersprache ist.

Programmiersprachen gibt es viele; denn die ultimate 'Programmiersprache', die natürliche Umgangssprache des Menschen, kann man bisher noch nicht in Maschinensprache übersetzen. Sie ist zu vieldeutig; in ihr bleibt vieles ungesagt, was zum Verständnis notwendig ist und was sich die arme Maschine nicht einfach - wie der Mensch - 'dazudenken' kann. Deshalb schaffen sich die Menschen für verschiedene Zwecke unterschiedlich geartete Programmiersprachen, von denen C - die Sprache, die Sie interessiert - eine ist.

Aber Sie sollten daran denken, daß eine Programmiersprache letztlich nichts anderes ist, als ein Übersetzungsprogramm in die Maschinensprache. Bei der Organisation dieses Übersetzungsvorgangs hat man verschiedene Wahlmöglichkeiten, die einen unmittelbaren Einfluß auf die Arbeit mit der Programmiersprache haben. Dies kann man sich am Beispiel eines menschlichen Dolmetschers leicht klarmachen.

## 1.2 Interpreter vs. Compiler

Wenn Sie in Japan mit einem Ihrer Sprache nicht mächtigen Geschäftspartner verhandeln, dann bedienen Sie sich eines Dolmetschers. Die größten Könner dieser Zunft sind die Simultandolmetscher, die es verstehen, quasi gleichzeitig mit Ihren deutschen Äußerungen eine Übertragung ins Japanische vorzunehmen. Dies hat für Sie den Vorteil, daß Sie die Wirkung des Gesagten unmittelbar an den Reaktionen Ihres Gesprächspartners, an den (simultan übersetzten) Antworten, Einwürfen etc. kontrollieren können: der Dialog ist wegen des beinahe unmittelbaren Feedbacks natürlicher.

Aber das Verfahren kann auch Nachteile haben. Hat man es beim Dolmetscher nicht mit einem Könner seines Faches zu tun, der aber dennoch zur Simultanarbeit gezwungen ist, dann wird dieser in seiner Not zur wört-

lichen Übersetzung greifen. Und das kann katastrophal enden, wie folgender (nicht erfundene - ich schwör's!) Ausschnitt aus einer Computer-Bedienungsanleitung zeigt:

> *... Der Microcomputer in der Tastatur führt viele Funktionen aus, einschliesslich der Selbstprüfung des Kraftbetriebs wenn ersucht vom Systemeinheit. Dies Prüfung prüft den Mikrocomputer ROM, Erinnerung, und die festsitzten Tasten. Die zusätzlichen Funktionen sind: die genaue Prüfung der Tastatur, der Puffer der Blickscodes, das Erhalten der seriemässigen Kommunikation mit der Systemeinheit, und das Ausführen des hand-schüttelnden Protokolls benötigt von jedem Transfer der Blickscodes.*

Diese aus dem Japanischen 'übersetzte' Bedienungsanleitung für eine Computer-Tastatur versteht auch der Spezialist nur nach schärfstem Nachdenken (und wenn der Blick nicht mehr von Lachtränen getrübt ist)!

Das Problem hierbei ist, daß sich Struktur und Wortschatz einer Sprache nicht Eins zu Eins auf eine andere übertragen lassen, sondern daß für die Übersetzung komplizierte Umstrukturierungen nötig sind. Der Könner unter den Dolmetschern nimmt diese wie beiläufig vor; weniger Geübte gestatten oftmals unfreiwillig Einblicke in diesen hochkomplexen Prozeß, was dem Verständnis nicht gerade förderlich ist.

Besonders katastrophal würde sich eine wörtliche Übertragung bei einem Gedicht oder einem anderen sprachlichen Kunstwerk auswirken. Übersetzer, die solche Aufgaben angehen - beispielsweise das Nibelungenlied ins Japanische übertragen - brauchen daher viel Zeit, detaillierte Kenntnis beider Sprachen und meterweise Lexika.

Warum ich Ihnen das erzähle? Weil es auch bei Computern Simultandolmetschern und literarische Übersetzer gibt. Die Simultanarbeiter nennt man hier jedoch "Interpreter"; der Kunstwerker heißt "Compiler".

Ein Interpreter ist ein Übersetzungsprogramm, das Ausdrücke (Programme) einer höheren Programmiersprache unmittelbar in die Maschinensprache überträgt und zur Ausführung bringt. Der bekannteste Interpreter ist der für die auf Home- und Personalcomputern allgegenwärtige Sprache BASIC. Etliche unter Ihnen werden BASIC bereits kennen und die Eigenschaft schätzen, mit einem einfachen RUN der Maschine sofort ein Ergebnis zu entlocken. Für den Programmierer, der in dieser Sprache arbeitet, hat dies den Vorteil des unmittelbaren Feedback, und dies kann die Programmentwicklung beträchtlich erleichtern.

Es gibt jedoch auch Nachteile bei diesem Verfahren. Um bei der Dolmetscher-Analogie zu bleiben: die Übertragungen, die der Interpreter von BASIC in die Maschinensprache vornimmt, sind oft zu wortreich und umständlich. Das bedeutet: ein in BASIC geschriebenes Programm ist oft viel umfangreicher (benötigt mehr Speicherplatz) und wird wesentlich langsamer von der Maschine ausgeführt, als das entsprechende Programm in Maschinensprache oder in Compilersprache.

Compilersprachen sind die zweite Alternative. Anstatt so früh wie möglich loszuübersetzen, bekommt ein Compiler das gesamte Programm vorgesetzt und kann dann daraus in aller Ruhe ein möglichst effizientes Maschinenprogramm bosseln. Das bedeutet nun nicht, daß Sie auf eine Compiler-Übersetzung tagelang warten müßten; meist ist der Übersetzungsvorgang in einigen Dutzend Sekunden abgeschlossen. Das Ergebnis der Übersetzung, das Maschinenprogramm, ist dann direkter auf die 'Denkweise' des Computers zugeschnitten und kann von diesem auch in einem Zug ausgeführt werden, ohne daß dauernd ein Übersetzer dazwischenquasseln muß.

Aber dennoch: zwischen dem Eingeben des Programmes und seiner Ausführung durch die Maschine vergeht Zeit, die Sie sich bei der Arbeit mit einem Interpreter sparen. Entwickelt jedoch wird ein Programm nur einmal, angewendet hingegen (hoffentlich) tausende Male. Und da zeigen sich die vom Compiler gelieferten Ergebnisse in ihrem besten Licht, denn sie sind in der Regel zehnmal schneller als die Produkte eines Interpreters, manchmal ist der Geschwindigkeitsvorteil sogar erheblich höher!

Compilersprachen haben gegenüber den Interpretersprachen noch einen Vorteil, der nicht so offensichtlich ist. Interpretersprachen verführen zum Drauflosprogrammieren: kaum haben sich die ersten unscharfen Umrisse einer Idee im Kopf gebildet, setzt man sich meist schon hin und hackt das Ideen-Fragment in die Maschine. Das mag bei kleineren Problemen angehen; für umfangreiche Programmieraufgaben ist es jedoch oftmals tödlich. Das hat man Ihnen ja schon in der Schule beigebracht, daß man für einen längeren Aufsatz erstmal eine Stoffsammlung anlegt, dann eine Gliederung entwirft und erst anschließend mit dem Schreiben anfängt.

Nun ist schon das Schreiben eines einfachen Adressverwaltungs-Programms im Vergleich mit einem Schulaufsatz ein Zentnerproblem. Sie können sich vorstellen, wie weit man mit der Methode kommt, halbgare Gedanken gleich auszuführen. Das entstehende Code-Chaos durchblickt oft nur mehr sein Schöpfer, und der nach zwei, drei Wochen auch nicht mehr.

Darum arbeiten nur mehr die wenigsten Software-Profis in BASIC; und wenn, dann erlegen sie sich strengste Disziplin auf. Wenn zeitkritische Anwendungen entwickelt werden müssen, dann ist mit BASIC ohnehin nichts

zu machen. Viele greifen dann zurück zur Maschinensprache und tun damit flugs den Schritt zurück in die Computer-Steinzeit; darüber im nächsten Kapitel mehr.

Deshalb spricht alles für Compilersprachen: ihre Modernität, die Geschwindigkeit und Kompaktheit der Programme und die Portabilität. Was "Portabilität" ist, wird im nächsten Kapitel erklärt; was jedoch bedeutet "Modernität" im Zusammenhang mit einer Programmiersprache?

Computerwissenschaftler forschen nun schon seit 30 Jahren darüber, wie die optimale Programmiersprache auszusehen hat. Die Erkenntnisse auf diesem Gebiet haben besonders in den letzten 20 Jahren drastisch zugenommen. Als Ergebnis hat sich ein zeitgemäßer Programmierstil entwickelt, der unter der Bezeichnung "strukturierte Programmierung" bekannt ist. Um diesen als besonders effizient erkannten Stil praktizieren zu können, bedient man sich moderner "strukturierter Programmiersprachen", von denen C eine ist.

Die Entwicklung von BASIC datiert jedoch auf den Anfang der sechziger Jahre zurück. Alles, was die Wissenschaft seither an Einsichten gewonnen hat, insbesondere die Methode der strukturierten Programmierung, ist an BASIC spurlos vorübergegangen. So ist BASIC - weil es jeder Mikrocomputer-Hersteller mit seinen Geräten als Dreingabe mitliefert - zwar immer noch eine aktuelle, aber alles andere als eine moderne Programmiersprache.

Die Forschungsergebnisse der Informatiker haben ihren Niederschlag überwiegend in modernen Compilersprachen gefunden. Eine davon ist C; womit wir schon angelangt wären bei der

## 1.3 Geschichte von C

In der deutschen Personal- und Heimcomputergemeinde ist C erst seit einigen Jahren bekannt. Profis (Softwarehäuser) benutzen es seit ca. 4 Jahren, die Hobbyisten wurden wohl erst in diesem Jahr (1986) nachhaltig damit konfrontiert. Letzteres kann man hauptsächlich auf das Erscheinen der ST-Modelle von Atari (Atari 520 ST+ und 260 ST) zurückführen und die Entscheidung dieser Firma, C zur Entwicklungssprache zu machen.

Daraus könnte man die Vermutung ableiten, daß C eine sehr junge Sprache ist. Doch dies täuscht. Sie wurde bereits Mitte der siebziger Jahre entwickelt, und sie ist das Werk eines einzigen Mannes, Dennis M. Ritchie.

Daß eine Programmiersprache das Werk eines Einzelnen ist, ist nicht selbstverständlich. Sprachen wie COBOL, das hauptsächlich in der Groß-EDV

für Verwaltungszwecke eingesetzt wird, Algol, eine Sprache für technisch-naturwissenschaftliche Anwendungen, oder neuerdings ADA, ein vom amerikanischen Verteidigungsministerium gesponsorter neuer Standard, wurden und werden von einem vielköpfigen Komittee entworfen. Doch dies tut den Sprachen nicht immer gut.

So ist auch die wohl bekannteste Programmiersprache, BASIC, das Werk zweier Männer. Der Bekanntheitsgrad von BASIC rührt nicht zuletzt daher, daß es für Anfänger besonders leicht und schnell zu erlernen ist; und genau dies war die Intention von Kemeny und Kurtz, den Vätern von BASIC. Nicht immer erfüllen Programmiersprachen in so vollkommener Weise die Intentionen ihrer Schöpfer; C ist ein weiterer solcher Glücksfall.

D. Ritchie war in den siebziger Jahren bei den amerikanischen Bell Laboratorien mit der Entwicklung des Betriebssystems UNIX beschäftigt. UNIX sollte unter anderem Multi-User- und Multi-Tasking-Fähigkeiten haben: mehrere Benutzer sollten (scheinbar) gleichzeitig einen Computer benutzen können und dabei auch noch mehrere Aufgaben (scheinbar) parallel erledigen können. Dies zu realisieren ist keine leichte Aufgabe und man kann sich vorstellen, daß es bei einem so zeitkritischen System sehr auf die Geschwindigkeit ankommt. Die einzige Möglichkeit zur Programmierung eines solchen Systems schien damals Maschinensprache zu sein. Doch wer Assembler kennt, der weiß, wie mühselig und umständlich das Arbeiten in dieser Sprache ist. Auch erhöht sich die Fehleranfälligkeit eines solchen umfangreichen Projektes durch Benutzung von Maschinensprache beträchtlich.

D. Ritchie hätte daher ganz gerne eine höhere Programmiersprache mit der Möglichkeit zur Systemprogrammierung - der Erstellung maschinen- bzw. systemnaher Programme - gehabt. Das gab es damals noch nicht, ja, es schien sogar ein Widerspruch zu sein: Systemprogrammierung macht man in Assembler und damit Basta!

Nun, seit den Anstrengungen von D. Ritchie macht man es eben in C. Denn dessen Reaktion war, sich nicht mit den bestehenden Verhältnissen zufriedenzugeben, sondern sich unverdrossen das Werkzeug zu verfertigen, das er brauchte. Und so entstand C, eine Sprache, die zwei bisher unvereinbar erscheinende Welten zu vereinen vermochte: die systemnahe, maschinennahe Programmierung, bis dato eine Domäne der Assembler-Freaks, und die abstrakte, strukturierte Programmierung, die modernen höheren Programmiersprachen, vor allem Pascal, vorbehalten zu sein schien.

Erklärungsbedürftig ist noch die Namenswahl; "C" als Name für eine Programmiersprache erscheint befremdlich. Der Grund liegt darin, daß es zu C einen Vorläufer gab, die Sprache "B", die Ritchie zu C weiterentwickelte. Schon wieder ein Einbuchstaben-Name! Dieser ist aber leichter erklärt: B

hatte nämlich ein Vorbild, die Sprache BCPL. BCPL ist die Abkürzung für "Basic Cambridge Programming Language" (zu Deutsch etwa: Elementare Programmiersprache aus Cambridge). Die Namenswahl ist nicht nur amüsant; sie ist auch typisch für den C-Programmierstil. C-Programme können nämlich verblüffend kurz und kompakt gehalten werden. C bietet dem Programmierer die Möglichkeit, sich elegant und prägnant auszudrücken; wer diesen eleganten Stil jedoch noch nicht gewohnt ist, tut sich manchmal (besonders als Anfänger) schwer mit dem Verstehen. Das hat C das Vorurteil eingebracht, eine 'unverständliche' Sprache zu sein. Sie werden noch selbst sehen, daß dieses Vorurteil unbegründet ist.

Das Betriebssystem UNIX, das momentan langsam aber sicher auch in die PC-Welt einsickert, ist als Ergebnis der Bemühungen von D. Ritchie zum größten Teil in C geschrieben. Von den ca. dreizehntausend Programmzeilen, die der Systemkern benötigt, ist lediglich ein kleiner Kern von etwa 800 Zeilen in Maschinensprache verfaßt. Alles, was ein Betriebssystem sonst noch nützlich macht - die sog. "Utilities" wie z.B. Editoren - ist gänzlich in C geschrieben. Zuvor hätte es niemand für möglich gehalten, daß so etwas überhaupt möglich ist!

C ist aber nicht nur gut, um Betriebssysteme damit zu entwickeln. Immer mehr Softwarehäuser machen es zur Haussprache. So hat die Firma MicroPro, weltweit bekanntgeworden mit ihrem Textprogramm WordStar, eine völlig neue Version dieses Programms unter dem Namen Wordstar 2000 auf den Mark gebracht, das völlig in C geschrieben ist. Auch Word, der Textprogramm-Bestseller von Microsoft, ist ein C-Programm.

Was macht C für diese Firmen so attraktiv? Es ist die Portabilität von C-Programmen. Man nennt ein Programm portabel, wenn es ohne Probleme von einem Rechner auf einen anderen gebracht werden kann. Dies setzt zweierlei voraus: einmal muß sich der Programmierer jeder Hardwareabhängigkeit enthalten. Er darf also in seinem Programm nicht auf Möglichkeiten zurückgreifen, die nur seine Maschine und sonst keine andere in dieser Form zu bieten hat. Die Verlockung zur maschinenabhängigen Programmierung ist besonders bei Fragen der Bildschirmausgabe groß. Viele Home- und Personal Computer verfügen über 'schlaue' Bildschirme mit einer Menge eingebauter Fähigkeiten (z.B. Zeichen oder Zeilen blinken zu lassen, zu unterstreichen oder revers darzustellen), und da übermannt einen leicht der Spieltrieb. Aber dies büßt man spätestens dann, wenn man sich einen neuen Rechner zulegt und das Programm jetzt auch gern auf diesem einsetzen will. Da es die Industrie versteht, so etwas wie Kompatibilität (technische Verträglichkeit) unter allen Umständen zu vermeiden, sind die Chancen groß, daß das gesamte Programm - oder zumindest der Teil, der sich um den Bildschirm kümmert - neugeschrieben werden muß.

Aber selbst wenn der Programmierer sich diszipliniert: wenn seine Programmiersprache nicht portabel ist, dann ist alle Liebesmüh vergeblich. Portabel sind Programmiersprachen dann, wenn sie standardisiert sind. Das sind aber nur die wenigsten (BASIC ist es nicht). Und auch mit standardisierten Sprachen hat es so seine Bewandtnis: denn selbst wenn eine Sprache standardisiert ist, so tastet man diesen Standard meist nur dann nicht an, wenn die Sprache perfekt ist. Hierzu bietet Pascal ein gutes Beispiel.

Für diese Sprache gibt es einen Quasi-Standard, nämlich das Buch, in dem Pascal erstmals einer größeren Öffentlichkeit vorgestellt wurde. Dieser nach seinen Autoren Jensen/Wirth-Standard genannte Sprachumfang ist jedoch in Punkto Ein-/Ausgabe nicht gerade üppig ausgestattet. Deshalb fühlt sich jedes Softwarehaus, das ein Pascal entwickelt, bemüßigt, sein Pascal in diesem Punkt etwas aufzumöbeln. Diese Aufmöbelei macht natürlich jeder Hersteller etwas anders - und aus ist's mit der Kompatibilität.

Bei C sieht der Fall anders aus. Die Sprache ist hinreichend mächtig und vollständig, um auch verwöhnte Programmierer zufriedenzustellen; und sie ist klein. Dies scheint ein Widerspruch zu sein, der sich jedoch im weiteren Verlauf des Buches auflösen wird. "Klein" bedeutet, daß es nicht allzuviele Befehlsworte in C gibt. Das bedeutet aber auch, daß man C relativ leicht auf einen neuen Computer bringen kann. Und da C vollständig ist, muß man es auch nicht erst aufpolieren.

Das macht C so attraktiv. Es dürfte wirklich die einzige Sprache sein, die in gleicher Weise für ganz kleine Maschinen (wie z.B. den weitverbreiteten C64 von Commodore) und für ganz große Maschinen (z.B. die viele Millionen Dollar teueren Supercomputer von Cray Research) zu haben ist. Wenn Sie also ein Programm in C entwickeln, dann erschließt sich Ihnen damit theoretisch der gesamte EDV-Markt.

C ist somit eine Sprache mit Vergangenheit, aber auch mit großer Zukunft. Denn nicht nur Atari hat sich für C entschieden; auch Commodore ist mit seinem Amiga diesen Weg gegangen.

Aber auch wenn man keine professionellen Ambitionen hat, sondern als Hobbyist über gesteigerte Programmiermöglichkeiten verfügen möchte, empfiehlt es sich, zu C zu greifen. Bisher war ja BASIC die Leib- und Magensprache des kleinen Mannes. Eine Zeitlang kann man mit BASIC auch uneingeschränkt glücklich sein. Aber besonders die Neu- und Wißbegierigeren unter uns, diejenigen, die der Maschine auf die Schliche kommen oder das letzte aus ihr herausholen wollen, stoßen bald an die Grenzen von BASIC. Viele greifen dann zur Maschinensprache, legen sich damit aber stark auf einen - ihren - Computer fest. Dies ist genaugenommen ein bedenklicher Schritt, denn bei der rapiden Entwicklung der

Computertechnik ist eine Computergeneration nach spätestens vier Jahren veraltet. Dann ist man vielleicht firm im Assembler des 6502, aber die neueren Entwicklungen gehen an einem vorbei.

Für ATARI-Besitzer gibt es noch ein weiteres wichtiges Argument. Die Funktions-Vielfalt von GEM - der grafischen Benutzeroberfläche - kann bisher nur unter C (und in Maschinensprache) genutzt werden. GEM-Programme sind hinreichend kompliziert, um die Vorteile einer höheren Programmiersprache wie C schätzen zu lernen!

Ein weiterer Grund, sich von BASIC abzuwenden, ist die unbestrittene Langsamkeit von BASIC-Programmen. BASIC als Interpretersprache kann es an Geschwindigkeit mit keiner Compilersprache - wie es C ist - aufnehmen. Und unter den Compilersprachen - FORTRAN, Algol, COBOL, PL/I, Pascal und wie sie alle heißen - ist C beileibe nicht die langsamste.

## 1.4 C im Vergleich

Die Hauptvorteile von C wurden ja bereits angerissen:

- o die Geschwindigkeit und Kompaktheit der Programme, die C als Compilersprache bietet,
- o die Modernität von C, das eine strukturierte Programmiersprache ist,
- o die Möglichkeit, in C maschinennah zu programmieren, ohne direkt auf die Maschinensprache zurückgreifen zu müssen
- o die Portabilität von C-Programmen, weil C auf unterschiedlichsten Computern einheitlich verfügbar ist.

All diese Vorteile - und daß es Spaß machen kann, in C zu programmieren - will Ihnen dieses Buch noch näherbringen. Aber C ist nicht die einzige Programmiersprache auf der Welt. Der Gerechtigkeit halber soll sie hier kurz mit den verbreitetsten Konkurrenten verglichen werden, denn nichts auf der Welt ist vollkommen, und so hat C auch im Vergleich mit anderen Sprachen nicht nur Stärken aufzuweisen.

### 1.4.1 BASIC und C

Der Hauptvorteil, den BASIC gegen C ins Feld zu führen hat: BASIC ist eine Interpretersprache. Wie Sie nun wissen, bedeutet dies, daß BASIC-Programme schneller getestet werden können, da der Programmierer sie unmittelbar ausprobieren und auf Fehler reagieren kann. Man sollte jedoch nicht vergessen, daß die Schöpfer von BASIC dieses hauptsächlich deshalb als Interpreter konzipierten, weil sie BASIC als ausgesprochene Lehr- und

Anfängersprache vorsahen. So ist auch unbestritten, daß man sich einen minimalen BASIC-Wortschatz schnell aneignen kann. Aber daß BASIC auch für größere Programme benutzt wird, ist eher ein historischer Zufall und entspricht nicht der ursprünglichen Intention seiner Entwickler: Es war die erste höhere Programmiersprache, die für Microcomputer verfügbar war und hat sich deshalb zum Quasi-Standard entwickelt.

### 1.4.2 FORTRAN und C

FORTRAN ist die älteste höhere Programmiersprache. Als sie entwickelt wurde, wußte man noch nichts über den Bau einer Programmiersprache. Im Vergleich zu den stromlinienförmig gestylten modernen Sprachen nimmt sich FORTRAN denn auch aus wie ein Sammelsurium erratischer Einfälle. Besonders die Theoretiker des strukturierten Programmierens haben mit Vorliebe FORTRAN als abschreckendes Beispiel herangezogen. Dennoch: im technisch-wissenschaftlichen Bereich, in den Universitäten und Labors erfreut sich FORTRAN noch immer großer Beliebtheit. Dies mag zwei Gründe haben: einmal die Bequemlichkeit - kann man erst mal eine Sprache, dann bleibt man auch dabei -, dann aber sicher die numerischen Fähigkeiten der Sprache.

Wenn Sie nämlich mit sehr großen Zahlen oder sehr genau rechnen müssen, wenn Sie gar mit Exotischem wie z.B. komplexen Zahlen hantieren sollten, dann ist FORTRAN noch immer die Sprache der Wahl. Allerdings sieht's bei Microcomputern damit nicht sehr rosig aus; weil FORTRAN doch ein ziemliches Sprach-Durcheinander ist, ist der Compiler auch sehr umfangreich und, wenn er den vollen Sprachumfang bieten soll, auch ziemlich teuer.

Das soll nicht heißen, daß man mit C nicht genau rechnen oder komplexe Arithmetik treiben könnte. Wie Sie noch sehen werden, ist C durchaus geeignet, mit sehr großen Zahlen genau zu hantieren. Nur wenn diese Genauigkeit für Ihre Zwecke nicht ausreicht, dann können Sie entweder die entsprechenden hochgenauen Arithmetik-Routinen in C selbst programmieren, oder zu einem guten FORTRAN greifen, in dem das schon eingebaut ist.

### 1.4.3 COBOL vs. C

Eine der größten Schwächen von FORTRAN sind seine Ein-/Ausgabe-Möglichkeiten. Das heißt, FORTRAN kann sehr gut mit Daten rechnen, aber diese mit einem FORTRAN-Programm in die Maschine zu bekommen und ihr anschließend wieder zu entlocken, ist eine Quälerei. Bald nach der Entwicklung der Computer begann man ihren Wert für Verwaltungsauf-

gaben (Personaldaten, Lohnabrechnung, andere Abrechnungen wie Strom, Wasser, Gas etc.) zu erkennen, und da wird hauptsächlich ein- und ausgegeben; die dazu nötige Rechnerei ist nicht der Rede wert.

Weil also FORTRAN diesen Aufgaben nicht gerecht wurde, rief die amerikanische Regierung im Jahre 1959 ein Komittee in die Welt (unter dem Namen CODASYL bekanntgeworden), das COBOL ausbrütete. Dessen Eignung für das Hin- und Herschaufeln großer Datenmengen ist unbestritten; aber ansonsten ist COBOL ein Sprachkoloß, und kein Computerwissenschaftler würde heute noch ernsthaft behaupten, daß man damit etwas anderes als ein Programm zum Ausdrucken von Listen vernünftig programmieren kann. Aber: in der Administration - sei's staatlich, sei's privatwirtschaftlich - ist COBOL immer noch die Umgangssprache.

Prinzipiell kann man alles, was in COBOL möglich ist, in C erledigen, das meiste sogar besser. Denn viele Merkmale von COBOL sind noch auf die Lochkarten-Verarbeitung zugeschnitten; diese umständliche Arbeitsweise haben Sie als Microcomputer-Benutzer gottseidank nicht nötig. Aber einige Bequemlichkeiten hat COBOL auch dem modernen Menschen zu bieten. So sind z.B. Sortierbefehle direkt in die Sprache eingebaut; der C-Programmierer muß sich, wenn er Daten ordnen will, selbst ein Sortierprogramm schreiben. Aber wie das geht, erfahren Sie hier ohnehin!

### 1.4.4 Pascal vs. C

Pascal hat auf Microcomputern den Siegeszug der strukturierten Programmierung eingeläutet. Es wurde von dem Schweizer Professor N. Wirth entwickelt und 1971 publiziert. Pascal wurde als Lehrsprache konzipiert, die Studenten mit den Prinzipien moderner Software-Entwicklung vertraut machen sollte. Deswegen wurde bei seiner Entwicklung großer Wert auf Fehlerprüfung während des Übersetzungsvorganges gelegt. Von den Universitäten drang Pascal bald 'nach außen', in die Wirtschaft und hier besonders in die Welt der Microcomputer.

Viele und gute Programme wurden in Pascal geschrieben; doch weil es als Studentensprache gedacht war, legte N. Wirth weniger Wert auf Eigenschaften, die in der Profi-Programmierung wichtig sind. Da ist einmal die Ein-/Ausgabe, die in Pascal umständlich ist und zum anderen die Möglichkeit zur maschinennahen Programmierung, die in Pascal gänzlich fehlt. Da die Zahl der Pascal-Fans groß ist, wurden Modifikationen an der Sprache vorgenommen, um sie den Bedürfnissen der Programmierer anzupassen. Dies ging jedoch sehr zu Lasten der Standardisierung: Pascal-Programme sind meist nicht portabel.

Auch geht dem Fortgeschrittenen die ewige Fehlerprüferei - für den Anfänger sicher ein Vorteil - bald auf die Nerven. Verhindert sie doch auch bestimmte Programmier-Tricks, die zwar von zweifelhaftem pädagogischen, dafür von umso größerem praktischen Nutzen sind. C macht im Vergleich zu Pascal keinerlei Fehlerprüfung; dadurch sind äußerst elegante Problemlösungen möglich, bei Anfängern führt dies aber oft zu ausgesprochen schwer zu entdeckenden Programmfehlern. Während Pascal eine Studentensprache sein sollte, war C eben von Anfang an als 'Erwachsenensprache' angelegt.

## 1.5 Wie arbeitet man mit C?

Compiler-Sprachen sind in ihrer Handhabung etwas umständlicher zu bedienen als Interpretersprachen. In der Regel erfolgt die Programmentwicklung in einem (unfreiwilligen) Zyklus, der aus den Schritten Edieren, Compilieren, Linken, Programmtest, Edieren, Compilieren... usw. besteht, bis das Programm fehlerfrei ist oder dafür gehalten wird.

### 1.5.1 Edieren des Programmes

Wenn man in BASIC programmiert, dann ruft man (z.B. mit dem Befehl BASIC, oder - bei Homecomputern - durch einfaches Einschalten des Computers) den Interpreter und kann dann sofort mit dem Schreiben des Programms beginnen. Ihre von der Tastatur eingegeben Zeichen werden vom Interpreter gelesen und von diesem im Computer-Speicher als Programmtext abgelegt und verwaltet. Der Interpreter gestattet es auch, das eingegebene Programm zu Testzwecken wieder anzusehen und zu modifizieren. Außerdem (und das ist eigentlich das Wichtigste) kümmert sich der Interpreter auch noch um die Ausführung Ihres Programms; er ist also ziemlich vielseitig.

Bei Compilersprachen ist das anders. Das Übersetzungsprogramm - der Compiler - erwartet, daß der Programmtext bereits vollständig in maschinenlesbarer Form vorliegt. Dies zu bewerkstelligen hilft er Ihnen jedoch nicht! Dazu brauchen Sie einen Editor.

Text - Programmtext oder anderen - in maschinenlesbarer Form bereitstellen bedeutet, ihn in einer Datei abzulegen. Dateien sind benannte Zusammenfassungen von (logisch zusammengehörigen) Informationen auf einem Datenträger, meist einer Diskette oder Festplatte. In Dateien können Texte allgemeiner Art gespeichert sein: Briefe, Berichte, auch die einzelnen Kapitel dieses Buches wurden, da es mit Computerhilfe erstellt ist, in Dateien gespeichert. Neben Klartext für den Menschen speichert man in

Dateien aber auch Programme, die die Maschine ausführen kann. Die Frage ist nur: wie kriegt man die Informationen in die Datei?

Dazu gibt es spezielle Programme, die man "Editoren" nennt. Editoren sind nützliche Hilfsmittel bei der Textverarbeitung, da sie neben dem normalen Erfassen von Texten nach Schreibmaschinen-Manier auch vielfältige Bearbeitungsmöglichkeiten erlauben. So können in bereits geschriebenen Text Zeichen eingefügt oder daraus wieder gelöscht werden, es können Textausschnitte (Wörter, Sätze, ganze Absätze und Seiten) im Text verschoben werden, es können Wörter oder andere beliebige Zeichenfolgen gesucht und/oder durch andere ausgetauscht werden usw. Editoren erleichtern die Arbeit eines jeden erheblich, der mit Text zu tun hat; und auch Programme sind erstmals nichts anderes als Text, wenn auch Text mit einer besonderen Bedeutung für den Computer.

Sie benötigen für die Arbeit mit C also einen Editor, mit dessen Hilfe Sie in Programmform das niederschreiben, was der Computer für Sie tun soll. Da man mit einem Editor viel mehr als nur Programme schreiben kann, lohnt sich diese Anschaffung in den meisten Fällen. Mit dem Entwicklungssystem des ATARI wird mittlerweise der Editor von Metacomco ausgeliefert, der leicht zu erlernen ist, da er die Funktionstasten auf der ATARI-Tastatur verwendet.

Der erste Schritt bei der Programmerstellung besteht also darin, mit einem Editor eine Programm-Datei zu erstellen, die Ihr C-Programm enthält. Übrigens ist in einem modernen Computersystem fast alles eine Datei. Neben dem, was der Prozessor (das 'Gehirn' Ihres Computers) kann - es ist herzlich wenig! - und ein paar sehr elementaren Fähigkeiten, die in ROMs (Festwertspeicher) fest verdrahtet sind, muß alles, was den Computer interessant macht, diesem erst beigebracht werden. Das geht, indem man ein Programm schreibt, und dieses Programm liegt in einer Datei. Oder in mehreren; denn zwischen der Form, in der ein Computer ein Programm verarbeiten kann (Sie erinnern sich: die Maschinensprache) und der Form, in der Sie es erstellen (Sie arbeiten mit C, einer höheren Programmiersprache) besteht ein Unterschied. Den herauszuarbeiten, hat man sich einige Fachbegriffe einfallen lassen:

Das Programm in der höheren Maschinensprache nennt man "Quellprogramm". Es ist in der vorliegenden Form vom Computer noch nicht ausführbar, sondern muß erst durch einen Übersetzer umgewandelt werden. Entsprechend bezeichnet man die Datei, in der sich dieses Quellprogramm befindet, auch als "Quelldatei".

Was sich nach der Umwandlung des Quellprogramms ergibt, ist ein "ablauffähiges Programm". Es liegt nämlich nun in Maschinensprache vor und

kann so vom Computer verstanden werden. Aus etwas dunklen Gründen nennt man dieses ablauffähige Programm auch "Objektprogramm"; die Datei, in der das Objektprogramm steht, wird entsprechend als "Objektdatei" bezeichnet.

### 1.5.2 Compilieren des Programmes

Das Programm liegt nun also in der Quelldatei. Damit die Maschine die Programmbefehle auch befolgen kann, muß es übersetzt werden (Sie erinnern sich noch an das Dolmetscher-Beispiel?). Daß diese Übersetzung ebenfalls von einem Programm besorgt wird, wissen Sie bereits. Dieses Übersetzer-Programm nennt man im Programmierer-Jargon auch einen "Compiler"; dies ist der Grund, warum ich C als "Compiler-Sprache" bezeichnet habe.

Zum Verständnis des Folgenden sind einige Kenntnisse über die Techniken notwendig, die dem Übersetzungsvorgang (auch "Kompilieren" genannt) zugrundeliegen. Dieser ist in drei logische Schritte gegliedert. Im ersten Schritt untersucht ein sog. "Scanner" die Quelldatei und versucht das, was für ihn erstmals wie ein unorganisierter Strom von Zeichen aussieht, aufzugliedern in bedeutungstragende Einheiten wie z.B. Befehlsworte.

Auch dies kann man mit den Verhältnissen in der natürlichen Sprache vergleichen. Hört man nämlich einer Unterhaltung in einer Sprache zu, die man nicht versteht, dann hat man meist das Gefühl, daß die Gesprächspartner sich außergewöhnlich schnell unterhalten; besonders beim Italienischen und anderen südländischen Sprachen drängt sich dieser Eindruck auf. Das Gefühl täuscht jedoch, wie psychologische Untersuchungen erwiesen haben. Das Phänomen hat seinen Ursprung darin, daß Sie in dem auf Sie einstürzenden Schwall an Lauten nicht ausmachen können, wo die Wortgrenzen liegen. Ihr auf das Deutsche 'geeichter' Scanner weiß nicht, wo ein Wort aufhört und das nächste anfängt, und deshalb geht alles für ihn zu schnell.

Sie sehen also, daß das Ausfindigmachen von Wörtern (bei Programmiersprachen spricht man nicht von Wörtern, sondern von "Tokens") für das weitere Verarbeiten einer Sprache wichtig ist. Dazu gehört auch das Erkennen von Nicht-Wörtern: Ihr eingebauter Deutsch-Scanner weiß z.B., daß es sich bei "Tisch" um ein Wort des Deutschen handelt, daß "Gewurbel" hingegen kein deutsches Wort ist (auch wenn es ein mögliches deutsches Wort wäre; im Unterschied dazu ist "Zphxkrwmpkrt" beim besten Willen kein mögliches deutsches Wort - auch das weiß Ihr Scanner!). Der Scanner des C-Compiler zerlegt ähnlich wie Ihr angeborener Deutsch-Scanner das

Quellprogramm in bedeutungstragende Einheiten, die obenerwähnten Tokens, und gibt diese weiter an die nächste Instanz.

Diese nächste Instanz kümmert sich um die "Syntaxanalyse". Mit "Syntax" bezeichnet man die Regeln, die angeben, wie Sätze einer Sprache gebildet werden. Natürliche Sprachen haben eine Syntax, die für das Deutsche z.B. festlegt, daß "Ich lese gerade ein Buch über C" ein deutscher Satz ist, während "Mich wird gerade ein deutscher Buch gelesen von C" das nicht von sich behaupten kann.

Ein großer Teil der Arbeit beim Erlernen einer neuen Programmiersprache - natürliche Sprachen bilden da übrigens keine Ausnahme - wird für die Beherrschung der Syntax dieser Sprache aufgewendet. Und die Aufgabe ist gar nicht so einfach; selbst alte Hasen, die bereits längere Zeit mit einer Programmiersprache arbeiten, machen da gelegentlich noch Fehler. Beim Anfänger passiert das entsprechend häufiger. Aber keine Angst: im Erkennen von solchen syntaktischen Fehlern sind die Compiler ziemlich gut. Wenn der Parser - dies ein anderer Ausdruck für den Compiler-Teil, der die Syntaxanalyse besorgt - einen Syntaxfehler erkennt, dann weist er Sie sofort darauf hin. Meist liefert er auch noch eine Fehlerdiagnose, mit der Sie leicht den Fehler korrigieren können.

"Wenn der Parser so schlau ist, warum korrigiert er den Syntaxfehler dann nicht gleich selbst?", könnte man nun Fragen. Aber dieses automatische Korrigieren geht nicht: denn dazu müßte der Computer ja wissen, was das Programm eigentlich bewerkstelligen soll, und dazu müßte es in Maschinensprache vorliegen und dazu braucht man den Compiler und der kann ja gerade deshalb nicht weitermachen, weil ein Syntaxfehler im Programm ist!

Ist das Programm aber einmal frei von Syntaxfehlern, dann kann der Parser seine vollständige syntaktische Struktur ermitteln, welche Voraussetzung für den dritten und wichtigsten Schritt beim Kompilieren ist: die Codeerzeugung.

Die dritte logische Komponente eines Compilers nimmt nämlich das Analyseergebnis des Parsers und erzeugt auf dieser Basis die Maschinenbefehle, aus denen sich das Objektprogramm zusammensetzt. Man spricht hier von "Codegenerierung", wobei der erzeugte Code noch nicht unbedingt in ablauffähiger Form vorliegen muß; aber darüber mehr im nächsten Unterkapitel.

Wenn der Compiler fertig ist, bedeutet das noch nicht, daß Ihr Programm jetzt garantiert fehlerfrei ist. Denn auch wenn es allen vom Compiler überprüften formalen Anforderungen genügt, kann das Programm doch immer noch logische Fehler enthalten, d.h. es macht zwar etwas, aber nicht das,

was Sie wollen. Gegen solche logische Fehler gibt es bisher noch kein Heilmittel außer Nachdenken. Syntaktische Fehler können von einem Programm - also maschinell - erkannt werden. Aber ob ein Quellprogramm seinen vorgesehenen Zweck auch erfüllt, kann nur der Mensch letztgültig bestimmen. Die Beseitigung von logischen Fehlern, die im Jargon der Zunft "Bugs" (englisch "Wanzen") genannt werden, nimmt daher auch bei der Programmierarbeit viel Zeit in Anspruch.

Sie sollten auf keinen Fall beunruhigt oder gar entmutigt sein, wenn in Ihrem Programm noch Bugs sind. Auch Profis passiert das immer wieder. Allerdings kann durch einen klaren und disziplinierten Programmierstil die Fehleranfälligkeit von Programmen reduziert werden. Wie man diesen Stil in C praktiziert, wird ebenfalls Gegenstand der folgenden Kapitel sein.

Mit den Einzelheiten des Kompilierens habe ich Sie deshalb behelligt, weil man als Compiler-Bauer (das ist keine neumodische Agronomen-Variante!) die drei logischen Schritte Scanner, Parser und Codegenerator auf verschiedene Weise zusammenspielen lassen kann (weil also verschiedene "Compiler-Architekturen" möglich sind). Einige Compiler sind so konstruiert, daß die Schritte hintereinander ablaufen, jeder Schritt als eigenes Programmmodul realisiert ist und die drei Instanzen sich Informationen über (temporäre) Hilfsdateien aneinander weiterreichen. Weil dadurch drei Durchgänge durch das Programm notwendig werden, spricht man in jenem unvergleichlichem Gemisch von Deutsch und Englisch, das die EDV-Zunft auszeichnet, von einem "Drei-Pass-Compiler" ("pass" = engl. "Durchgang"). Dies kann sich auf die Art der Handhabung Ihres Compilers oder auf den Speicherplatz-Bedarf auf Diskette oder Festplatte auswirken; lesen Sie dazu den entsprechenden Passus in Ihrem Handbuch, für dessen Verständnis Sie jetzt gerüstet sind.

Bei anderen Architekturen arbeiten die drei Instanzen verzahnt: kaum hat der Scanner ein Token erkannt, reicht er es an den Parser, der jede gefundene syntaktische Einheit sofort zur Codeerzeugung weitergibt. Wir haben es dann mit einem "Ein-Pass-Compiler" zu tun.

Das eben Gesagte gilt für jeden Compiler, also auch für einen Pascal-, FORTRAN- oder COBOL-Compiler. Bei C gibt es jedoch noch eine Besonderheit. In C kann man mit Makros arbeiten (darüber später Näheres); um diese Makros korrekt verarbeiten zu können, benötigt der Compiler noch einen eigenen Durchgang, der den anderen dreien vorgeschoben ist. Diese allererste Programmanalyse betreibt der sog. "Makro-Präprozessor".

### 1.5.3 Bibliotheken und der Linker

Zum Programmierer-Jargon, der sich im weiteren Verlauf des Buches noch häufen wird, möchte ich ein paar Worte verlieren. Natürlich gibt es für alles deutsche Namen; statt "Compiler" können Sie auch "Übersetzer" sagen. Und auch das in der Überschrift enthaltene Wort "Linker" hat eine deutsche Entsprechung: "Binder". Aber diese Bezeichnungen haben sich - wie alle anderen eingedeutschten Ausdrücke der Computerwissenschaft - nicht durchgesetzt. Dies ist verständlich, denn die Befehlsworte aller Programmiersprachen (ob BASIC, Pascal oder C) sind ohnehin der englischen Sprache entnommen; da gibt es keinen deutschen Ersatz. Weil dies so ist, haben sich viele Programmierer einen typischen deutsch-englischen Jargon angewöhnt, eine Art Fliegerdeutsch der Computerzunft. Wenn Sie mitreden wollen, dann sollten Sie diesen Jargon kennen. Anstatt also mit aller Gewalt deutsch zu bleiben, werde ich vielmehr versuchen, Ihnen neben der Programmiersprache C auch noch die Sprache zu erläutern, in der man fachgerecht über C spricht. Außerdem: ob das 'deutsche' "Interpretierer" wirklich so viel verständlicher ist wie "Interpreter"?

Jetzt also zum Linker. Das Endprodukt des Compilers (das, was die Codeerzeugung abliefert) ist - anders, als man es erwarten würde - noch nicht ablauffähig. Obwohl dies unpraktisch erscheint: es hat praktische Gründe. Die wurzeln in der Technik des modularen Programmierens.

Das Wort "Modul" bzw. "modular" kennen Sie vielleicht von der Fernseh- und HiFi-Technik. Moderne Geräte sind in einzelne sauber gegliederte funktionale Einheiten aufgeteilt, die Module. Die Reparatur eines Fernsehgerätes sieht heutzutage so aus, daß der Techniker mit seinen Meßgeräten das fehlerhafte Modul lokalisiert und dann als Ganzes austauscht. Dieses Arbeiten mit Moduln ist auch bei der Programmierung möglich.

In vielen Programmen werden immer wieder die gleichen Verfahren benötigt. Hierzu drei Programm-Beispiele: Sie haben Ihre Bücher im Computer erfaßt und wollen sie nun nach Autoren sortiert ausdrucken; Sie haben ein elektronisches Telefonverzeichnis, das Sie sortiert ausdrucken wollen; Sie wollen die Sammlung Ihrer im Computer gespeicherten Kochrezepte nach Rezeptdauer anordnen. All diese Vorhaben haben eines gemeinsam: es muß sortiert werden.

Sie könnten nun für jedes der drei Programme wieder einen Sortierteil schreiben, aber das ist unnütze Mehrarbeit. Die Lösung in C: man verfaßt einmal eine Sortiermodul. Das ist ein selbständiges Programm, das, mit Daten versorgt, diese sortieren kann. Dieses Modul bewahrt man dann an einem Ort auf, an dem es jederzeit herangezogen werden kann, falls es

benötigt wird. Dieser Ort ist die Modul-Bibliothek, die "Library", wie der C-Freak sagt.

Die Library ist eine Ansammlung von Moduln, die dort in der Form abgelegt sind, wie sie der Compiler liefert. Wenn Sie einen C-Compiler kaufen, dann gehört zum Lieferumfang mindestens eine Library, in der viele wichtige und nützliche Moduln abgelegt sind. Diese ist als "Standard-Library" bekannt. Sie können sich auch eigene Bibliotheken anlegen, in denen Sie Module aufbewahren, die für Ihre speziellen Zwecke nützlich und wichtig sind. Auch die Erweiterung der Standard-Library um eigene Module ist möglich, wenngleich davon abzuraten ist.

Da man das Rad nicht zweimal erfinden soll, greift so gut wie jedes C-Programm auf ein oder mehrere Moduln aus der Library zurück. Um ausgeführt werden zu können, müssen diese Moduln mit dem Programm erst verknüpft werden. Diesen Prozeß des Verknüpfens nennt man "linken" (englisch "to link" = verknüpfen) und er ist Aufgabe des Linkers.

Wie man den Linker handhabt, also mit welchem Kommando er aufgerufen wird und welche Informationen er benötigt, ist von Produkt zu Produkt verschieden. Hierzu muß ich Sie wieder auf das Handbuch verweisen; da Ihnen Wesen und Wollen eines Linkers jetzt bekannt ist, sollten Sie damit keine Probleme haben.

### 1.5.4 Ein Zwischenschritt: Assembler-Mnemonics

Als ob es nicht schon kompliziert genug wäre, schieben manche Compiler in den Verarbeitungsprozeß zwischen Kompilieren und Linken noch einen Zwischenschritt ein. Hierbei erzeugt der Compiler nämlich noch keine Objektdatei, sondern eine Datei in sog. Assemblersprache. Dies ist eine weniger verschärfte Form der Maschinensprache, in der die binären Befehle des Computers durch leichter zu merkende Befehlsworte, sog. Befehlskürzel oder "Mnemonics" ersetzt sind. Statt den Befehl zur Addition zweier Werte in der Form

```
0100 1101 0001 0101
```

zu erteilen, schreibt man in Assembler:

```
ADD   A,BETRAG
```

(dieses Beispiel ist fiktiv und entstammt nicht der Assemblersprache des ATARI!) Das Befehlskürzel "ADD" ist nichts anderes als eine einprägsame Abkürzung für den Maschinenbefehl "0100 1101". Außerdem sieht man, daß die Verwendung von "symbolischen Namen" erlaubt ist. Das bedeutet, daß der Programmierer für Daten und Speicherstellen, mit denen er arbei-

ten will, nicht die tatsächlichen Binärwerte, sondern sinnvolle Abkürzungen (wie z.B. "BETRAG") schreiben kann.

Die Assembler-Programmierung ist zwar wesentlich einfacher als die Arbeit mit Maschinensprache. Da aber in Assembler nur Befehle verfügbar sind, die der Computer direkt ausführen kann - und das ist nichts besonders Komfortables - ist die Assemblerprogrammierung im Vergleich zum Arbeiten mit einer höheren Programmiersprache immer noch mühselig genug.

Warum aber legt dann der Compiler diesen Zwischenschritt ein? Dies hat Gründe, die mit der Optimierung des Programms zusammenhängen. Denn das Arbeiten in Assembler ist zwar mühsam, da man in dieser Sprache aber den Bedürfnissen der Maschine am meisten entgegenkommt, kann man sehr effiziente Programme damit schreiben. Erinnern Sie sich: auch UNIX ist zu einem kleinen Teil in Assembler geschrieben.

Wenn der Compiler Assembler-Mnemonics erzeugt, dann kann der Programmierer zeitkritische Teile seines Programms in dieser Assembler-Datei untersuchen und gegebenenfalls umschreiben, um sie effizienter zu machen. Wer diese Möglichkeit der Handoptimierung nicht nutzen will, der braucht sich mit Assembler gar nicht erst herumzuschlagen. Er muß jedoch aus der Assembler-Datei eine Objektdatei machen lassen, und diese besorgt ein Programm, das man "Assembler" nennt. Hier heißt also der Übersetzer einmal genauso wie die Sprache, die er übersetzt.

## 1.6 Ein Beispiel: das ATARI-C

Stellvertretend für die anderen auf dem ATARI verfügbaren C-Compiler sollen hier die Schritte aufgeführt werden, die man bei der Arbeit mit dem C des Entwicklungssystems durchzuführen hat. Dieses C stammt von der Firma Digital Research, die auch für das Betriebssystem TOS verantwortlich zeichnet. Im folgenden wird es kurz als "ATARI-C" bezeichnet. Beachten Sie daher: die im folgenden aufgeführten Schritte gelten nur für dieses C. Sollten Sie mit einem anderen Compiler arbeiten, dann müßen Sie dessen Beschreibung zu Rate ziehen, um zu erfahren, wie es nach dem Edieren des Programmes weitergeht.

Der Compiler des ATARI-C ist ein Drei-Pass-Compiler mit vorgeschaltetem Makro-Präprozessor. Zum Kompilieren eines Programmes müssen nacheinander (in der angegebenen Reihenfolge) die folgenden Programme bemüht werden:

1. CP68 der Makro-Präprozessor
2. C068 Tokenisierer und Syntaxanalyse
3. C168 Code-Generierung

Das Ergebnis von C168 wird an den Assembler weitergegeben:

4. AS68

Schließlich kombiniert der Linker das Programm mit allen benötigten Modulen; er trägt den Namen

5. LINK68

Außerdem benötigen Sie noch drei weitere Programme:

6. RELMOD wandelt Linker-Ergebnis in lauffähiges Programm
7. RM zum Löschen nicht mehr benötigter Dateien
8. BATCH für die Stapel-Verarbeitung.

Angenommen, Sie haben Ihr erstes C-Programm mit dem Editor in eine Datei mit dem Namen ERSTES.C geschrieben. Quell-Dateien für den C-Compiler sollten Sie stets mit dem Namenszusatz (der Extension) ".C" versehen. Dann erhalten Sie mit folgenden Schritten ein lauffähiges Programm: (die folgenden Ausführungen gehen davon aus, daß Sie ein doppelseitiges Laufwerk besitzen und alle hier genannten Programme auf einer Diskette unterbringen. Was Besitzer von einseitigen Laufwerken zu tun haben, wird weiter unten gesagt.)

```
1. CP68 ERSTES.C ERSTES.I
2. C068 ERSTES.I ERSTES.1 ERSTES.2 ERSTES.3 -f
3. C168 ERSTES.1 ERSTES.2 ERSTES.S
4. AS68 -u ERSTES.S
5. LINK68 [u] ERSTES.68K=GEMS,APSTART,ERSTES,OSBIND,GEMLIB,LIBF
6. RELMOD ERSTES.68K ERSTES.PRG
```

Diese Schritte sollen nun im Einzelnen erläutert werden. Als erstes kommt der Präprozessor, der alle Makros expandiert (darüber erfahren Sie später mehr). Er erzeugt als Ergebnis eine Datei mit dem Namen ERSTES.I, die Eingabe für Scanner und Parser (C068) ist. Dieser Schritt arbeitet mit einer Hilfsdatei ERSTES.3 und erzeugt zwei Ergebnisdateien, die die Namen ERSTES.1 und ERSTES.2 haben und Eingabe für den nächsten Schritt sind. Der Codegenerator C168 erzeugt aus ERSTES.1 und ERSTES.2 eine Datei mit Assembler-Mnemonics, die hier ERSTES.S genannt wird. Daraus macht der Assembler AS68 eine Datei in dem Format, das mit dem Linker verträglich ist. Die Datei heißt ERSTES.O, sie scheint jedoch in der Kommandozeile nicht auf.

Am kompliziertesten ist der Aufruf des Linkers; sein Ergebnis trägt den Namen ERSTES.68K und entsteht, indem eine Reihe von Moduln aus verschiedenen Bibliotheken zusammen mit der Datei ERSTES.O kombiniert werden. Voraussetzung dafür ist, daß Sie im Besitz der Modul-Dateien bzw. der Bibliotheken sind, die in der Kommandozeile des Linkers aufgeführt sind. Im Einzelnen handelt es sich um die Dateien:

GEMS.O
APSTART.O
OSBIND.O
GEMLIB
LIBF.

Die letzten beiden Dateien machen die Standard-Library aus, wobei diese getrennt ist nach Funktionen, die mit der Gleitpunkt-Arithmetik zu tun haben (die in LIBF untergebracht sind) und allen anderen Funktionen (Datei GEMLIB). Die ersten drei Moduln sorgen dafür, daß Ihre Programme unter TOS gelinkt werden.

Dies bedarf der Erläuterung. Auf dem ATARI können Sie Programme entwickeln, die die grafischen Möglichkeiten von GEM und die Maus nutzen. Windows (Fenster), Icons (Bildsymbole), Menüs, all diese schönen Dinge, die Sie vom GEM-Desktop kennen, können aus C heraus programmiert werden. Doch die GEM-Programmierung ist eine hohe Kunst; sie setzt die vollkommene Beherrschung von C voraus.

Einfacher ist es, Programme für die unter GEM liegende Schicht des Betriebssystems zu erstellen, für TOS. TOS ist ein kommandoorientiertes System. Was unter GEM das Auswählen eines Icons und doppeltes Anklicken mit der Maus bewirkt (nämlich das Laufenlassen eines Programms), das erreicht man unter TOS, indem man den Namen des Programms eintippt und dann die *Return*-Taste betätigt. Was unter GEM durch Öffnen einer Diskette bewirkt wird (nämlich die Anzeige des Disketten-Inhalts), das erreicht man unter TOS durch das Kommando DIR. Was unter GEM durch Verschieben eines Icons von einem Fenster in ein anderes bewirkt wird (nämlich das Kopieren von Dateien), das erreicht man unter TOS mit dem Kommando COPY (vgl. dazu jedoch das Kapitel 7.7!).

TOS mag Ihnen als die umständlichere Lösung erscheinen; dies ist jedoch Geschmacksfrage. Jedenfalls ist die kommandoorientierte Art des Arbeitens die, die auf anderen Micro- und Homecomputern üblich ist. Diejenigen unter Ihnen, die bereits mit einem anderen System gearbeitet haben, werden sich unter TOS zuhause fühlen. Für die Zwecke dieses Buchs ist von ausschlaggebender Bedeutung, daß die C-Programmierung unter TOS einfacher ist als unter GEM. Deshalb ist sie für eine Einführung geeigneter.

Der letzte Schritt (Programm RELMOD) macht aus dem Linker-Ergebnis ein ablauffähiges Programm. Die in diesem Schritt vorgenommenen Operationen zu erklären, übersteigt den Rahmen des Buches. Ihre Kenntnis ist für das Verständnis der C-Programmierung nicht von Bedeutung.

Sie können natürlich das ATARI-C auch von GEM aus benutzen; wie, das verrate ich Ihnen gleich. Aber über eines müßen Sie sich im Klaren sein: keines der in diesem Buch vorgestellten Programme arbeitet mit Windows, Menüs oder der Maus. Diese Möglichkeiten bleiben fortgeschrittenen C-Programmierern vorenthalten. Sie sind Gegenstand eines eigenen Buches. Soviel sei jedoch gesagt: soll ein Programm die Möglichkeiten von GEM nutzen, dann muß es "unter GEM" gelinkt werden. Der sechste und letzte Schritt muß dazu abgeändert werden, da eine Reihe von GEM-spezifischen Moduln vom Linker in das Programm einzubinden sind.

Um mit TOS direkt zu verkehren, benötigen Sie ein Programm mit dem Namen COMMAND.PRG. Wenn Sie dieses starten (doppeltes Anklicken mit der Maus), dann verschwindet das Desktop und Sie sehen stattdessen die Bereitschaftsmeldung - der "Prompt" von TOS, nämlich:

```
{a}          oder
{b}
```

Je nachdem, welches Laufwerk (Laufwerk A oder B) gerade das aktuelle Laufwerk ist, ändert sich der Buchstabe in den geschweiften Klammern.

Nun ist es äußerst umständlich, zum Übersetzen eines einzigen Programms jedesmal sechs nicht gerade kurze Kommandos zu geben. Deshalb gehört zum Lieferumfang des Entwicklungssystems auch ein Programm zur Stapel-Verarbeitung, das den Namen BATCH.TTP trägt. Ein Stapel ist nichts anderes als eine Datei mit Kommandos, bei denen für einige Angaben Platzhalter gelassen wurden. Die Stapel-Datei zur Übersetzung von C-Programmen sieht so aus:

```
CP68 %1.C %1.I
C068 %1.I %1.1 %1.2 %1.3 -f
RM %1.I
C168 %1.1 %1.2 %1.S
RM %1.1 %1.2
AS68 -u %1.S
RM %1.S
LINK68 [u] %1.68K=GEMS,APSTART,ERSTES,OSBIND,GEMLIB,LIBF
RELMOD %1.68K %1.PRG
RM %1.68K
```

**Abb. 1.1:** *Stapel-Datei für ATARI-C (doppelseitiges Laufwerk)*

In neueren Versionen des Entwicklungssystems ist eine Batch-Datei mit diesem Inhalt unter dem Namen CLOS.BAT enthalten. Ich empfehle Ihnen, der Datei den Namen C.BAT zu geben. Der kurze Name erspart Ihnen später eine Menge Tipparbeit; die Extension (den Namenszusatz) ".BAT" muß die Datei jedoch auf jeden Fall haben, gleichgültig für welchen anderen Namen Sie sich entscheiden. Das Umbenennen von Dateien unter GEM erreicht man durch einmaliges Anklicken des Datei-Symbols. Anschließend wählen Sie aus der Menü-Zeile das Menü "Datei" und darin wieder die Option "Info-Anzeige". Jetzt können Sie den Namen in der Dialogbox ändern und mit OK bestätigen.

Um eine Datei unter TOS umzubenennen, benutzen Sie das Kommando REN. Dieses ist in TOS eingebaut und wird wie folgt verwendet:

```
REN CLOS.BAT C.BAT <RETURN>
```

(Alle TOS-Kommandos werden erst durch Drücken der *Return*-Taste ausgeführt.) Damit haben Sie die Datei von CLOS.BAT in C.BAT umbenannt.

Sie sehen, daß in der Stapeldatei die bereits bekannten Kommandos enthalten sind, anstelle des Programmnamens ERSTES findet sich jedoch der Platzhalter "%". Zusätzlich sind zwischen die einzelnen Schritte jedoch noch Aufrufe eines neuen Programms. mit dem Namen RM eingestreut. Auch dies gehört zum Entwicklungssystem. Es löscht nicht mehr benötigte Hilfsdateien von der Diskette und sorgt so dafür, daß diese nicht allzu schnell voll wird.

Einige der Kommandos sind übrigens mit Optionen versehen (mit einem Bindestrich eingeleitete, einbuchstabige Hinweise an die Programme), die die Art der Fehlerdiagnose betreffen, welche die Programme durchführen; ihre Bedeutung entnehmen Sie dem Handbuch des Compilers.

Um die Stapeldatei zu benutzen, rufen Sie das Stapelprogramm BATCH. TTP auf. Unter GEM müßen Sie zuerst dieses Programm mit der Option "TOS übernimmt Parameter" anmelden (im Menü "Optionen"). Unter TOS schreiben Sie

```
BATCH.TTP C ERSTES <RETURN>
```

Dies hat (beinahe) dieselbe Wirkung wie die sechs ausführlichen Schritte zu Beginn dieses Abschnitts; "beinahe" deshalb, weil überflüssige Zwischendateien gelöscht werden. BATCH setzt in der Stapeldatei namens C für jedes "%1" den Namen der Quelldatei ERSTES ein und arbeitet dann die einzelnen Kommandos der Reihe nach ab. Sollte bei der Verarbeitung eines Kommandos ein Fehler auftreten, so können Sie den Kommandostapel

durch Drücken der *Return*-Taste anhalten. BATCH fragt Sie dann, ob Sie weitermachen wollen. Durch Eingeben von "N" bricht der Prozeß ab.

Wenn Sie unter TOS arbeiten, können Sie den Aufruf des Stapelprogramms durch Umbenennen beträchtlich vereinfachen. Geben Sie folgendes Kommando:

```
REN BATCH.TTP B.PRG <RETURN>
```

Damit wird das Stapelprogramm BATCH.TTP in B.PRG umbenannt und kann künftig unter TOS so aufgerufen werden:

```
B C ERSTES <RETURN>
```

"B" ist der Name des Stapelprogramms; "C" ist der Name der Stapeldatei, in der die Kommandos stehen; ERSTES ist der Name der Quelldatei. Letzterer ändert sich natürlich, wenn Sie ein anderes Programm übersetzten lassen wollen. Als Ergebnis der ganzen Prozedur erhalten Sie nach ca. 5 Minuten (so lange dauert die Abarbeitung des Stapels mit Disketten) ein Programm mit dem Namen ERSTES.PRG auf Ihrer Diskette, das Sie unter TOS einfach dadurch laufenlassen können, daß Sie

```
ERSTES <RETURN>
```

tippen. Übrigens macht TOS keinen Unterschied zwischen Groß- und Kleinschreibung; wenn Ihnen das besser gefällt, können Sie alle Kommandos und Dateinamen ebensogut mit Kleinbuchstaben schreiben.

Wer nicht mit doppelseitigen Laufwerken gesegnet ist, der bringt nicht alle für die Verarbeitung des Stapels nötigen Programme auf einer Diskette unter. In diesem Fall müssen sie wie folgt auf zwei Disketten verteilt werden.

| Diskette 1 (in Laufwerk A): | | Diskette 2 (in Laufwerk B): |
|---|---|---|
| CP68.PRG | | LINK86.PRG |
| C068.PRG | | RELMOD.PRG |
| C168.PRG | | GEMS.O |
| AS68.PRG | | APSTART.O |
| RM.PRG | | OSBIND.O |
| .STDIO.H | | GEMLIB |
| OSBIND.H | | LIBF |
| PORTAB.H | | |
| TOSDEFS.H | | |
| HEADER.H | (Siehe unten!) | |
| Ihr Editor | (falls noch Platz bleibt) | |
| Ihre Quelldatei | | |

**Abb. 1.2:** *Disketten-Inhalt für die Arbeit mit 2 Laufwerken*

Die Batch-Datei, die mit dieser Konfiguration arbeitet, hat folgenden Inhalt:

```
CP68 %1.C %1.I
C068 %1.I %1.1 %1.2 %1.3 -f
RM %1.I
C168 %1.1 %1.2 %1.S
RM %1.1 %1.2
AS68 -u %1.S
RM %1.S
B:LINK68 [u] B:%1.68K=B:GEMS,B:APSTART,ERSTES,B:OSBIND,B:GEMLIB,B:LIBF
B:RELMOD B:%1.68K B:%1.PRG
RM B:%1.68K
```

**Abb. 1.3:** *Stapel-Datei für ATARI-C (zwei einseitige Laufwerke)*

Diese Batch-Datei bewirkt, daß das fertige Programm auf der Diskette im Laufwerk B zu stehen kommt. In der Zusammenstellung der Abbildung 1.2 sind übrigens fünf Dateien mit der Extension ".H" für das Laufwerk A aufgetaucht, zu denen ich Ihnen noch eine Erklärung schuldig bin.

Die Dateien haben mit dem Makro-Präprozessor zu tun. Ein volles Verständnis dieser Materie ist erst möglich, wenn Sie in die Geheimnisse der C-Makros eingeweiht wurden. Soviel jedoch sei vorweggenommen: der Compiler benötigt für seine Arbeit bestimmte Informationen, die in sog. "Include"-Dateien mit der Extension ".H" enthalten sind. Nicht für jedes Programm werden alle Informationen benötigt. Doch schadet es nichts, dem Compiler jedesmal alle Informationen mitzugeben. Ich empfehle Ihnen deshalb, mit Ihrem Editor auf Laufwerk A eine Datei Namens HEADER. H zu erstellen ( in der obigen Aufstellung ist sie bereits enthalten), die folgenden Inhalt hat:

```
#include <stdio.h>
#include <osbind.h>
#include <portab.h>
#include <tosdefs.h>
```

**Abb. 1.4:** *Die Include-Datei HEADER.H*

Jedes Programm, das im folgenden besprochen wird, sollten Sie mit der folgenden Zeile beginnen:

```
#include <header.h>
```

Jedes Beispielprogramm sollte so eingeleitet werden. In den Beispiel-Programmen des Buches fehlt diese Zeile jedoch stets, weil es sich dabei um ein ausgesprochenes Spezifikum des ATARI-C handelt. Andere Compiler benötigen diese Informationen vermutlich nicht. Die Beispielprogramme enthalten nur Informationen, die für jedes standardisierte C zutreffen.

Wenn Sie die *#include*-Zeile in Ihre C-Programme mit aufnehmen (denken Sie daran: es sollte die erste Zeile in Ihrem Programm sein und das #-Zeichen muß in der ersten Spalte dieser Zeile stehen!), dann ist dafür gesorgt, daß der ATARI-C-Compiler (genauer: der Makro-Präprozessor) alle benötigten Informationen findet. Die Dateien STDIO.H, OSBIND.H, PORTAB.H und TOSDEFS.H sind Bestandteil des Entwicklungssystems. Wenn Sie in C etwas versierter sind, dann können Sie sich diese Dateien ansehen und nur mehr das aufnehmen, was für Ihr Programm gerade benötigt wird. Bis dahin jedoch rate ich Ihnen zu dem eben beschriebenen Verfahren.

## 1.7 Zusammenfassung

In diesem Kapitel haben Sie erfahren

- daß Programme in einer höheren Programiersprache erst übersetzt werden müssen, ehe sie vom Computer verstanden werden;
- daß es zwei Arten von Übersetzern gibt: Interpreter, die Programme sofort übersetzen und ausführen - analog einem Simultandolmetscher - und Compiler, die ein Programm als Ganzes betrachten und übersetzen - analog einem literarischen Übersetzer;
- daß C eine Compilersprache ist, die von D. Ritchie als strukturierte höhere Programmiersprache entwickelt wurde, mit der auch maschinennahe Systemprogrammierung möglich ist;
- daß ein Compiler in die logischen Arbeitsschritte Scanner, Parser und Codegenerator gegliedert ist und daß bei C noch ein Makro-Präprozessor vorgeschaltet ist;
- daß in C Module in einer Bibliothek oder Library abgelegt sind, die mit dem eigentlichen Programm durch einen Linker zusammengebunden werden;
- daß manche C-Compiler noch einen Zwischenschritt einlegen und erst Assembler-Mnemonics erzeugen, die vor dem Linken noch durch einen Assembler in Objektdateien gewandelt werden müssen;
- daß Programmierer manchmal recht seltsam daherreden.

# 2 Zur Sache

Nachdem jetzt ziemlich viel über C geredet wurde, wird es Zeit, endlich mal C zu reden. Ihr Tatendrang ist vermutlich groß; aber ich muß Sie bitten, sich noch ein wenig zu gedulden. Denn das erste Programm, das Sie kennenlernen werden (Prog. 2.1), tut nichts. Daß auch das zu etwas gut sein kann, werden Sie gleich sehen.

```
main()
{
}
```

**Prog. 2.1:** *Das leere Programm*

Dieses scheinbar nutzlose Beispiel hat seine Berechtigung darin, daß es Ihnen klar zeigt, was man in C mindestens hinschreiben muß, um ein Programm zu bekommen. Sie können daran die minimalen Bestandteile eines C-Programmes erkennen: es hat einen Namen (in diesem Falle *main()*), hinter dem in geschweiften Klammern umschlossen das steht, was das Programm tun soll: Anders ausgedrückt: die geschweiften Klammern enthalten die Anweisungen des Programms, oder - noch mal anders gesagt - sie umschließen einen "Anweisungsblock". Diese Erläuterungen sind jetzt natürlich noch etwas akademisch, weil das Programm 2.1 ja nichts tun soll. Darum steht eben auch nichts zwischen den Klammern; der Anweisungsblock ist leer. Für die BASIC-Kundigen unter Ihnen will ich (in Abbildung 2.1) zur Erinnerung nochmal zeigen, wie man dasselbe (also nichts) in BASIC tut.

```
10
```

**Abb. 2.1:** *Nichts in BASIC*

```
main(){}                    ——— Variante 1: alles auf einer Zeile

main   (    )  {

}                           ——— Variante 2: die Anzahl der Leerzeilen ist
                                    beliebig

main( )  {                  ——— Variante 3
}

main()                      ——— Variante 4: zugleich knapp und
{}                                  übersichtlich
```

**Abb. 2.2:** *Varianten des Nichts*

Am leeren Programm läßt sich bereits eine nützliche Eigenschaft von C demonstrieren, die sog. "Formatfreiheit". Dazu sollten Sie die Abb. 2.2 betrachten.

Hier sehen Sie vier unterschiedliche aber zulässige Arten, das Programmbeispiel 2.1 hinzuschreiben. Sie können daran erkennen, daß C (genauer: der C-Compiler) einen freizügigen Umgang mit Leerzeichen (oder Tabulator-Vorschüben) sowie mit der Plazierung auf einer oder mehreren Zeilen erlaubt. Aber nicht alles ist möglich; werfen Sie einen Blick auf Abb. 2.3:

```
mai  n() {}          Da ist der Wurm drin: keine Tokens
                     trennen!
```

**Abb. 2.3:** *Nichtswürdige Variante des Nichts*

Hier wurden Leerzeichen in den Programmnamen *main* eingestreut, und das nimmt der Compiler übel. Sie können Leerzeichen in C mit Sprachpausen in der gesprochenen Rede vergleichen: zwischen Worten und Sätzen kann man sich beim Reden (beinahe) beliebig viel Zeit lassen; aber innerhalb der Worte tut man das nicht, weil die Verständlichkeit des Gesagten darunter leidet. Ebenso hält's der Compiler: zwischen Tokens darf beliebig viel leerer Raum sein; aber die Token müssen ungetrennt bleiben.

Somit haben Sie jetzt bereits ein Token kennengelernt, nämlich *main*. Diesen Programmnamen habe ich nicht deshab gewählt, weil ich etwa Anwohner jenes fränkischen Flusses wäre. Vielmehr ist es der obligatorische Name der Hauptfunktion.

## 2.1 main: die Basis für alles Weitere

Keine Angst; so knüppeldick wie im letzten Absatz kommen die Fremdworte so schnell nicht wieder. "Obligatorischer Name der Hauptfunktion" bedeutet folgendes: C-Programme sind aus einzelnen Moduln aufgebaut, die man im C-Jargon Funktionen nennt. Funktionen sind die elementaren Bauklötzchen von C; und weil das an Lego-Bausteine erinnert, will ich das Beispiel noch etwas weiter strapazieren und daran "Hauptfunktion" erklären.

Wenn Kinder mit Lego-Steinchen spielen, dann bauen sie komplexere Gebilde, indem sie einzelne Steinchen (die Module des Lego-Systems) ineinanderstecken. Aber für dieses Ineinanderstecken und Aufeinandertürmen brauchen sie einen Anfang; das ist die Grundplatte. Lego benutzt Grundplatten, auf denen die Konstruktionen aufgebaut werden. Ähnlich ist es in C.

Ein größeres C-Programm entsteht, indem Funktionen in geeigneter Weise ineinandergeschachtelt werden; dazu gleich noch ein Beispiel. Für das Ineinanderschachteln muß jedoch in C - ähnlich wie beim Lego-Spiel - ein Anfang gegeben sein. Dieser ist die Hauptfunktion *main*, die also für C eine ähnliche Rolle wie die Grundplatte bei Lego spielt. In jedem C-Programm muß die Hauptfunktion vorhanden sein (darum nannte ich sie "obligatorisch") und sie trägt stets den Namen *main* (was eine verkürzte Schreibweise für das Englische "main program", zu Deutsch "Hauptprogramm" ist). Das versprochene Beispiel für verschachtelte Funktionen finden Sie in Programm 2.2.

```
main()
{   tue_nichts();
}

tue_nichts()
{
}
```

**Prog. 2.2:** *Das leere Programm: Hauptfunktion mit Unterfunktion*

Programm 2.2 unterscheidet sich in seiner Wirkung nicht von Programm 2.1; es wurde lediglich das gleiche (nämlich: nichts) mit anderen Mitteln erreicht. Wie Sie sehen, steht jetzt Text zwischen den geschweiften Klammern von *main*: der Anweisungsblock ist nicht mehr leer. Es wurde ja bereits erwähnt, daß in diesen Klammern das steht, was das Programm zu tun hat. In diesem Fall wird innerhalb von *main* eine selbstgemachte Funktion aufgerufen, nämlich *tue_nichts*. (Warum sowohl hinter *main* als auch hinter *tue_nichts* stets ein Paar ziemlich sinnlos erscheinender Klammern steht, wird gleich noch erklärt!) Daß diese Anweisung nach rechts eingerückt im Programm niedergeschrieben ist, hat lediglich ästhetische Gründe. Bei umfangreicheren Programmen macht diese Methode der optischen Textgliederung das Programm lesbarer. Die Vorgehensweise und Wirkung von Programm 2.2 wird in dem BASIC-Beispiel von Abb. 2.4. ziemlich genau nachvollzogen.

```
10  GOSUB 100  ———————  Dies entspricht dem Aufruf von tue_nichts
20   END
100 RETURN     ———————  Dies entspricht tue_nichts
```

**Abb. 2.4:** *Warum einfach, wenn's auch umständlich geht!*

Beachten Sie den Strichpunkt hinter dem Aufruf von *tue_nichts* in *main*! Wenn man dem Computer in C sagt, was er innerhalb einer bestimmten Funktion zu tun hat, dann geschieht das in Form von sogenannten "Anweisungen". Diese sind vergleichbar den Sätzen der natürlichen Sprache. Bei einer einfachen Aufgabe - wie im Programm 2.2 - kommt man mit einer

Anweisung aus, komplexe Aufgaben erfordern mehrere Anweisungen. Wie in der natürlichen Sprache auch, ist es beim Programmieren in C erforderlich, das Ende von Anweisungen zu markieren (man macht dadurch dem Compiler das Leben leichter). Im Deutschen benutzen Sie dazu die Satzzeichen Punkt, Ausrufe- und Fragezeichen. In C beendet man eine Anweisung mit dem Strichpunkt.

In *main* wird eine einzige Anweisung ausgeführt; diese gehört jedoch nicht zum Grundwortschatz von C. Kein C-Compiler der Welt kann mit *tue_nichts* etwas anfangen, es sei denn, man sagt ihm, was darunter zu verstehen ist. Dies wird dem Computer in Programm 2.2 jedoch beigebracht: Hinter der schließenden geschweiften Klammer von *main* findet er die Definition von *tue_nichts*, also die Festlegung dessen, was bei der Ausführung von *tue_nichts* zu geschehen hat. Wie Sie wissen, ist das nichts! In der Abbildung 2.5 sind die einzelnen Bestandteile des Programms noch einmal erläutert.

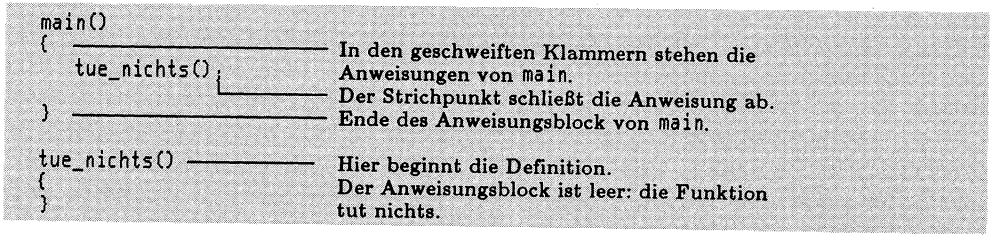



**Abb. 2.5:** *Die Anatomie des Programms 2.2*

Programm 2.2 demonstriert so das Prinzip der Schachtelung von Funktionen: innerhalb von *main* wird *tue_nichts* aufgerufen, oder - anders gesehen - der Aufruf von *tue_nichts* ist in *main* eingeschachtelt. Doch beiden Beispielprogrammen fehlt noch etwas sehr Wesentliches: der Kommentar.

Nur verantwortungslose Programmierer liefern Programme ab, in denen nicht jeder Schritt erklärt ist. Wenn man ein Programm schreibt, steckt man noch tief in dessen Logik drin und glaubt deshalb, alles wäre selbstverständlich und -erklärend. Dies ist eine besondere Form der Betriebsblindheit, deren Auswirkungen der Programmierer oft selbst schmerzhaft zu spüren bekommt. Wendet man sich nämlich nach längerer Zeit einem Programm zu, dann hat man oft die größte Mühe, das zu verstehen, was zum Zeitpunkt der Entstehung so selbstverständlich und klar schien.

Deshalb versieht der Profi seine Programme mit Kommentaren. Diese sind lediglich Erläuterungen für den menschlichen Leser des Programms; sie haben keinerlei Auswirkungen auf das Programm selbst. Das ist anders als in der Interpretersprache BASIC, bei der Kommentare vom Interpreter über-

lesen werden müssen und so die Ausführungszeit verlangsamen. Eine der Aufgaben des C-Scanners ist es jedoch, Kommentare zu überlesen und die anderen Instanzen damit gar nicht mehr zu behelligen.

In C werden Kommentare durch die Zeichenfolge /* eingeleitet und durch */ abgeschlossen. Alles, was dazwischen steht, ist Kommentar. Dieser kann sich auch über mehrere Zeilen erstrecken. Das Programm 2.3 demonstriert dies; es ist lediglich eine kommentierte Variante von Programm 2.2. Allerdings erlaubt C nicht die Verschachtelung von Kommentaren: Kommentare innerhalb von Kommentaren bringen den Compiler durcheinander.

```
main()                                  /*  Hauptprogramm; ruft    */
{   tue_nichts();                       /*  selbstdefinierte       */
}                                       /*  Funktion               */

tue_nichts()                            /* Diese Funktion tut      */
{                                       /*  nichts.                */
}
```

**Prog. 2.3:** *Ein wohldokumentiertes, wenn auch uninteressantes Beispiel*

Da die Wirkung von Programm 2.3 trivial ist, ist auch der Kommentar dazu nicht sonderlich erhellend. Sie können Kommentare in Ihrem Programm plazieren, wie und wo Sie wollen. Der besseren Lesbarkeit wegen empfiehlt es sich jedoch, Kommentare stets an den gleichen Spalten der Zeile beginnen und enden zu lassen.

## 2.2 Einfache Ausgabe: printf

Jetzt ist es aber an der Zeit, ein Programm zu schreiben, das wirklich etwas tut! Sie finden es in Programm 2.4. Ähnlich wie im letzten Beispiel wird hier in *main* eine Funktion aufgerufen; sie trägt den Namen *printf*. Dieser Funktionsaufruf ist die einzige Anweisung in *main*. Wie alle Anweisungen wird er mit einem Strichpunkt abgeschlossen.

```
main()                                  /*  Gib eine Meldung auf   */
{   printf("Jetzt geht es los!");       /*  dem Bildschirm aus.    */
}
```

**Prog. 2.4:** *Ausgabe in C*

Im Unterschied zu *tue_nichts* im letzten Beispiel ist *printf* eine Systemfunktion; d.h. sie ist C bereits bekannt (genauer: sie ist Bestandteil der Standard-Library) und muß deshalb nicht gesondert definiert werden. Wie die anderen Funktionen auch, ist *printf* mit einem Paar runder Klammern

dekoriert; doch diesmal sind die Klammern nicht leer. Sie enthalten vielmehr das Argument von *printf*.

Die meisten Funktionen in C sind dazu da, um mit Daten etwas zu tun. So ist es zum Beispiel die Aufgabe von *printf*, Botschaften auf dem Bildschirm auszugeben. Dazu muß man der Funktion aber mitteilen, welche Botschaft sie ausgeben, also mit welchen Informationen sie arbeiten soll. Diese aktuell von einer Funktion zu bearbeitenden Informationen werden Argumente genannt. Sie werden der Funktion in dem Klammerpaar hinter dem Funktionsnamen übergeben. Die Meldung "Jetzt geht es los!" ist somit das Argument von *printf*.

Beachten Sie, daß die Zeichenfolge, die *printf* ausgeben soll, von Anführungszeichen umschlossen ist. Kenner der Materie sagen hier, daß das Argument von *printf* ein String sein muß; dazu später mehr. Fürs Erste sollen Sie sich lediglich merken, daß *printf* Strings auf dem Bildschirm ausgibt und daß ein String eine Zeichenfolge ist, die von Anführungszeichen umrahmt wird.

Es gibt Funktionen mit einem Argument, wie *printf* eine ist, aber auch solche mit mehreren Argumenten und auch Funktionen ohne jedes Argument. Die Beispielsfunktion *tue_nichts* fällt in letztere Kategorie, weswegen das runde Klammerpaar beim Aufruf dieser Funktion leerbleibt.

Bei der optischen Gestaltung eines Funktionsaufrufs hat der Programmierer wieder mehrere Alternativen, die sich durch Einstreuen von Leerzeichen und -zeilen ergeben. Alle in der Abbildung 2.5 aufgeführten Varianten sind zulässig.

```
printf ("Jetzt geht es los!");
printf( "Jetzt geht es los!" );
printf (
        "Jetzt geht es los!"
       );
```

**Abb. 2.6:** *Formatmöglichkeiten für Funktionsaufrufe*

Welcher Variante Sie den Vorzug geben, hängt einzig von Fragen des persönlichen Geschmacks ab. Wenn Sie sich jedoch einmal für einen bestimmten Stil entschieden haben, dann sollten Sie ihn durchgängig in Ihren Programmen praktizieren.

## 2.3 Der Compiler muß ran

Sicher verspüren Sie den Wunsch, das Programm 2.4 auch mal laufen zu lassen; dies bedeutet, den Compiler anzuwerfen. Dazu müssen Sie mit Ihrem Editor den Programmtext in eine Datei schreiben. Dieser Datei geben Sie am besten den Namen BSP24.C, um zu dokumentieren

a) daß es sich dabei um das C-Beispiel 2.4 handelt,
b) daß Sie wie alle echten Computer-Fans zu unverständlichen Abkürzungen neigen.

Um keine Verwirrung aufkommen zu lassen: der Name, den Sie der Datei geben, ist zugleich der Programmname. Wenn Ihr Programm fertig übersetzt und gelinkt ist, dann rufen Sie es mit "BSP24" auf.

Wer mit dem ATARI-C arbeitet und die im letzten Kapitel enthaltenen Ratschläge zur Namensgebung des Batch-Programms und der Stapeldatei befolgt hat, der kann den Übersetzungsvorgang unter TOS mit folgendem Kommando anstoßen:

```
B C BSP24
```

Sollten Sie beim Übersetzen Fehlermeldungen erhalten, dann überprüfen Sie, ob Sie auch alles richtig abgeschrieben haben. Beim Arbeiten mit der Stapeldatei können Sie die Abarbeitung unterbrechen, indem Sie die *Return*-Taste betätigen. Im Fehlerfall meldet Ihnen der Compiler (genauer: der Teil mit dem Namen C068) die Nummer der Zeile, in der der Fehler enthalten ist. Sie können jetzt in den Editor gehen und das Programm verbessern. Allerdings sollten Sie die Zeilenangabe in der Fehlermeldung nicht allzu wörtlich nehmen. Der Fehler kann durchaus auch eine oder zwei Zeilen vor der gemeldeten lokalisiert sein. Dies sollten Sie sich besonders für spätere, längere Programme merken.

Ist Ihr Programm fehlerfrei, dann können Sie es - wie bereits erwähnt - mit "BSP24" aufrufen und sich an der Meldung auf dem Bildschirm erfreuen.

Wenn Sie unter GEM arbeiten, dann trübt Ihnen leider ein Wermutstropfen die Freude: die Programmausgabe erscheint ganz kurz am Bildschirm und anschließend wird sie wieder vom Desktop und dem Datei-Fenster überlagert. All dies geht so schnell, daß man keine Chance hat, die Meldung zu lesen. Das Problem haben Sie allerdings nicht, wenn Sie unter TOS arbeiten; dies würde ich Ihnen auch für alle praktischen Übungen empfehlen.

Sollten Sie jedoch aus dem einen oder anderen Grund vorziehen, die Übungs-Programme aus GEM zu starten, dann gibt es eine Möglichkeit zur

Abhilfe, die jedoch voraussetzt, daß Sie über das Programm WAIT aus dem Entwicklungssystem verfügen. Sollte dies nicht der Fall sein, dann finden Sie in der Abbildung 2.7 ein C-Programm, das den gleichen Zweck erfüllt und das Sie mit dem Namen WAIT übersetzen sollten. Tippen Sie das Programm einfach wie gezeigt ab; was es macht, wird Ihnen in späteren Kapiteln klar werden.

```
#include <header.h>                              /*  Siehe Kapitel 1.6!     */
main()

{   puts("\nBitte eine beliebige Taste druecken...");
    getchar();
}
```

**Abb. 2.7:** *Das Programm WAIT*

Erstellen Sie nun mit Ihrem Editor eine Stapel-Datei (bei zwei einseitigen Laufwerken: auf Laufwerk A), der Sie den Namen RUN.BAT geben und die folgenden Inhalt hat:

```
%1
WAIT
```

Von nun an können Sie aus GEM ihre Programme starten, indem Sie das Batch-Programm (haben Sie es auch "B" genannt?) anklicken und ihm dann folgende Angaben mitgeben:

```
RUN BSP24
```

bzw.

```
RUN B:BSP24
```

Die erste Variante setzt voraus, daß sich Batch-Programm, Stapeldatei und das Beispielprogramm BSP24 alle auf derselben Diskette befinden. Die zweite Variante wenden Sie dann an, wenn sich das Batch-Programm und die Stapeldatei auf Laufwerk A befindet, ihr Programm jedoch auf Laufwerk B steht. Letztere Situation ist dann gegeben, wenn Sie - wie im ersten Kapitel beschrieben - mit zwei einseitigen Laufwerken arbeiten.

Wenn Sie Ihre Programme über die Stapeldatei aufrufen, dann sehen Sie erst die Ausgabe des Beispielprogramms, gefolgt von einer freundlichen Aufforderung, eine Taste zu betätigen. Wenn Sie mit dem Original-WAIT-Programm aus dem Entwicklungssystem arbeiten, dann müssen Sie die *Return*-Taste betätigen; beim selbstgestrickten WAIT ist's egal, welche Taste Sie wählen.

## 2.4 Ersatzdarstellung mit Escape-Sequenzen

Das, was Programm 2.4 bewirkt, kann man auch auf viele andere Arten notieren; einige davon sind recht verblüffend. Sehen Sie sich dazu einmal das Programm 2.5 an.

```
main()
{
    printf("Jetzt ge");                    /*  printf macht ohne be-  */
    printf("ht es los!");                  /*  sondere Anweisung kei- */
}                                          /*  nen Zeilenvorschub!    */
```

**Prog. 2.5:** *Die Eigenheiten von* printf

Der Anweisungsblock von *main* enthält jetzt zwei Aufrufe der Funktion *printf*. Aber nicht das ist das Besondere an diesem Beispiel. Ins Auge fällt vielmehr die brutale Art, mit der die Meldung "Jetzt geht es los!" entzweigerissen wurde; Herr Duden würde sich im Grabe umdrehen. Wenn Sie dieses Programm jedoch übersetzen, linken und laufen lassen, dann werden Sie feststellen, daß sich sein Ergebnis in keiner Weise vom Programm 2.4 unterscheidet.

Vielleicht hätten Sie erwartet, daß die Meldung ebenso zerstückelt und über zwei Zeilen verteilt auf dem Bildschirm erscheinen würde, wie sie im Programm steht. Doch nichts dergleichen passiert.

Daran erkennt man: *printf* gibt die Zeichen beginnend mit der aktuellen Cursorposition der Reihe nach auf dem Bildschirm aus, bis es mit seiner Arbeit fertig ist (Zur Erinnerung: der Cursor ist die blinkende Schreibmarke am Bildschirm). Dann hält es einfach inne und bewegt nicht etwa den Cursor auf den Anfang einer neuen Zeile. Ein nachfolgender *printf*-Aufruf beginnt mit der Zeichenausgabe unmittelbar hinter der Stelle, an der das letzte *printf* innegehalten hat.

Wenn ich das aber nicht will? Wie schaffe ich es, in C eine über zwei Zeilen verteilte Meldung auszugeben? Nun, dazu müssen Sie *printf* anweisen, einen Zeilenvorschub zu erzeugen. Da *printf* aber nur für die Ausgabe von Zeichen zuständig, ein Zeilenvorschub aber kein Zeichen im herkömmlichen Sinne ist, muß es dafür eigene Vorkehrungen geben.

```
main()
{                                          /*  Die Zeichenfolge \n im */
    printf("Diese Meldung ist");           /*  Argument von printf    */
    printf("\nauf zwei Zeilen verteilt."); /*  bewirkt einen Zeilen-  */
}                                          /*  vorschub.              */
```

**Prog. 2.6:** *Ersatzdarstellung für den Zeilenvorschub*

Man ist dazu auf die Idee gekommen, für den Zeilenvorschub und andere Steuerzeichen (also 'Zeichen', die keine Buchstaben oder Ziffern sind, sondern Instruktionen für ein Ausgabegerät darstellen) eine Ersatzdarstellung zu verwenden. Wie das geht, sehen Sie in Programm 2.6. Das Argument des zweiten *printf* im Anweisungsblock wird jetzt durch eine seltsame Zeichenfolge eingeleitet, nämlich durch

```
\n
```

Dem liegt folgende Konvention zugrunde. Dem umgekehrten Schrägstrich ist eine besondere Aufgabe zuteil geworden. Er signalisiert nämlich, daß das auf ihn folgende Zeichen (das "n") etwas besonderes ist und als Steuerzeichen interpretiert werden muß. Die beiden Zeichen bilden zusammen eine Ersatzdarstellung für den Zeilenvorschub, der sich direkt nicht hinschreiben läßt. Daß man ausgerechnet diese Zeichenkombination gewählt hat, liegt an der englischen Sprache; "n" soll nämlich an das Englische "newline" erinnern, das auf Deutsch "Zeilenvorschub" bedeutet. Es gibt noch eine Reihe weiterer Ersatzdarstellungen, die in der Abb. 2.7 zu besichtigen sind.

| **Ersatz-darstellung** | **Bedeutung** |
|---|---|
| \n | Zeilenvorschub |
| \t | Tabulator |
| \b | Rückschritt ("backspace") |
| \r | Wagenrücklauf ("return") |
| \f | Seitenvorschub ("form feed") |
| \" | Doppelte Anführungszeichen (innerhalb eines Strings) |
| \' | Einfaches Anführungszeichen (als Zeichenkonstante) |
| \nnn | Beliebiges Bitmuster mit Oktalwert "nnn" |

**Abb. 2.8:** *Die Ersatzdarstellungen von C.*

Ist in die auszugebende Zeichenfolge die Ersatzdarstellung für den Tabulator eingestreut, so wird das nächste darauf folgende Zeichen beginnend mit der nächsten Tabulatorposition auf dem Bildschirm ausgegeben. Klingt kompliziert? Versuchen Sie es einfach einmal mit Programm 2.7.

```
main()
{
    printf("Das\tist\tein\n");                    /*  \t steht fuer Tabula-  */
    printf("Beispiel\tf}r\tden\tTabulator");      /*  tor. Der Zeilenvor-   */
}                                                 /*  schub kann auch am En- */
                                                  /*  de erfolgen.          */
```

**Prog. 2.7:** *Ersatzdarstellung für den Tabulator*

Wie Sie an der Ausgabe des Programms am Bildschirm sehen können, sind jetzt die Wörter, vor denen das Tabulator-Zeichen steht, auf die jeweils nächste Tabulatorposition gerückt. Das "für" in der zweiten Zeile steht dabei an Tabulatorposition 3, da das erste Wort dieser Zeile ("Beispiel") sich bereits über die zweite Position erstreckt.

Das Prinzip der Ersatzdarstellungen ist eigentlich ganz einfach: etwas, das auf normalem Wege nicht dargestellt werden kann, wird einfach durch ein Stellvertreterzeichen signalisiert. Als Stellvertreter fungieren in C die kleingeschriebenen Abkürzungen der englischen Bezeichnungen für die Funktionen. Um zu signalisieren, daß diese Abkürzungen nicht für sich selbst stehen, braucht es noch ein besonderes Kennzeichen, in diesem Fall den umgekehrten Schrägstrich. Solche Kennzeichen, die darauf hinweisen, daß das, was nach ihnen kommt, aus der Reihe fällt (nicht wie üblich zu interpretieren ist), nennt der Programmierer "Escape"-Zeichen (vom engl. "to escape" = entkommen). Auf ein Escape-Zeichen folgt stets (mindestens) ein weiteres Zeichen, das angibt, was eigentlich passieren soll. Deshalb - weil man es also immer mit ganzen Zeichenfolgen zu tun hat - spricht man auch von "Escape-Sequenzen".

Nun kann es gelegentlich erforderlich sein, auch das Escape-Zeichen selbst auszugeben. Wie sollte man sonst z.B die Meldung

```
Mit "\n" erreicht man einen Zeilenvorschub
```

auf den Bildschirm bekommen? Hier stellt sich gleich noch ein zweites Problem. Denn diese Meldung enthält die Anführungszeichen, mit denen normalerweise das Ende des auszugebenden Textes für *printf* signalisiert wird. Man kann also nicht einfach schreiben:

```
printf("Mit "\n" erreicht man einen Zeilenvorschub.");
```

weil sonst bereits bei dem Leerzeichen hinter "mit" für *printf* das Argument zuende ist. Anführungszeichen, die von *printf* ausgegeben werden sollen, bedürfen also einer Sonderbehandlung - und schon denken Sie an Ersatzdarstellung! Daß Sie sich da nicht geirrt haben, bestätigt das Programm 2.8.

```
main()
{   printf("\nMit \"\\n\" erreicht man ");        /*  \\ steht fuer Escape   */
    printf("einen Zeilenvorschub.");              /*  \" steht fuer Anfueh-  */
}                                                 /*  rungszeichen.          */
```

**Prog. 2.8:** *Die Ersatzdarstellung der Ersatzdarstellung.*

Damit erst mal genug von diesem Thema; die restlichen Escape-Sequenzen aus der Abbildung 2.8 werden bei Gelegenheit noch besprochen. Es sollte Sie jedoch nichts daran hindern, selbst damit zu experimentieren...

## 2.5 Erste Bekanntschaft mit dem Präprozessor

C-Feinde dichten der Sprache manchmal an, in ihr könne man keine übersichtlichen Programme schreiben. Dabei besitzt C eine einzigartige Möglichkeit zur übersichtlichen und klaren Programmgestaltung, die man in anderen Programmiersprachen vergeblich sucht: den Makro-Präprozessor. Schon wieder so ein furchtbares Fremdwort! Aber schauen Sie mal im Programm 2.9 nach; dann wird die Sache klarer.

```
#define MELDUNG "\n\tJetzt geht es erst richtig los!"

main()
{
    printf(MELDUNG);
}
```

**Prog. 2.9:** *Ein Makro-Beispiel*

Wenn Sie dieses Beispiel laufen lassen, dann sehen Sie - um eine Tabulatorposition eingerückt - den Text "Jetzt geht es erst richtig los!" auf Ihrem Bildschirm. Dieser ist im Programm jedoch nicht Argument von *printf*, wie es in den bisherigen Beispielen üblich war. Vielmehr steht da ein geheimnisvolles *MELDUNG*. Aber C weiß, was das bedeuten soll; denn in der ersten Zeile des Programms haben Sie dem Compiler gesagt, was damit gemeint ist.

Mit *#define* können Sie beliebige Zeichenfolgen durch andere ersetzen. Damit haben Sie die Möglichkeit, für Programmteile selbstgewählte Namen zu vergeben. Im Beispiel 2.9 wurde einfach die auszugebende Botschaft auf den Namen MELDUNG getauft. Da, wo sonst im Programm der String zu stehen käme (in den Argumentklammern von *printf*), wurde jetzt einfach dessen Name verwendet.

```
#define <eigener_name>  <alte_zeichenfolge>
```

**Abb. 2.9**: *Format für #define-Makros*

Alle *#define*-Anweisungen folgen in ihrem Aufbau den gleichen Regeln. Erst kommt das 'Kennwort' *#define*, dann folgt Ihr eigener Name, und dahinter kommt die alte Zeichenfolge, die Sie solcherart umtaufen wollen. Programmierer haben eine eigene Art, solche Regeln aufzuschreiben, die sie für übersichtlich halten. Sie sehen das in Abbildung 2.9; naja - Geschmacksache!

Wenn Sie eigene Namen definieren wollen, dann müssen Sie darauf achten, daß das "#" von *#define* unmittelbar am Anfang der Zeile steht. Nur dann wird es von der dafür zuständigen Instanz des Compilers erkannt und rich-

tig verarbeitet. Diese Instanz ist übrigens der bereits im ersten Kapitel erwähnte Makro-Präprozessor.

Man nennt ihn "Präprozessor", weil er vor dem eigentlichen Übersetzungsvorgang auf eigene Faust durch das Programm geht und (unter anderem) nach *#define*-Anweisungen sucht. Dann merkt er sich, was Sie umdefiniert haben und setzt jedesmal, wenn er Ihren selbstgewählten Namen findet, dafür den Text ein, der in der *#define*-Zeile hinter dem vereinbarten Namen kommt. Daß der Präprozessor ein sehr nützliches Merkmal von C und die Makros äußerst leistungsfähig sind, werden Sie im Laufe des Buches noch feststellen.

## 2.6 Der Rechner rechnet

'Rechner' heißen die Computer im Volksmund; daß sie noch ganz anderes können als rechnen haben Sie bereits gesehen: das Ausgeben von Botschaften hat mit Arithmetik nichts zu tun. Aber rechnen kann der Computer natürlich auch; und das soll er jetzt (Programm 2.10).

```
                                        /*************************/
main()                                  /* Einfache Arithmetik.  */
{                                       /*************************/
    int Summe;                          /* Deklaration von drei  */
    int Zahl_1;                         /* Variablen, die ganze  */
    int Zahl_2;                         /* Zahlen aufnehmen.     */
                                        /*************************/
    Zahl_1 = 5;                         /* Wertzuweisung an die  */
    Zahl_2 = 13;                        /* Variablen.            */
    Summe = Zahl_1 + Zahl_2;            /*                       */
                                        /*************************/
    printf("\nDie Summe von %d und %d ist %d.",  /* Ausgabe Ergebnis:     */
           Zahl_1,                      /* zusaetzliche Argumente */
           Zahl_2,                      /* fuer printf auf eige- */
           Summe);                      /* ner Zeile.            */
                                        /*************************/
}
```

**Prog. 2.10:** *Einfache Arithmetik in C*

Bei diesem Beispiel tut sich im Anweisungsblock von *main* einiges. Wie auch der Kommentar deutlich macht, gliedert sich der Anweisungsblock in drei Abschnitte. Der erste Abschnitt erledigt die Variablen-Deklaration; der zweite kümmert sich um Wertzuweisung an die Variablen und um die Berechnung des Ergebnisses; der dritte Abschnitt sorgt dafür, daß der Benutzer die Ergebnisse auch zu sehen bekommt, hat also mit Ausgabe zu tun.

### 2.6.1 Variablen: Namen und Deklaration

Variablen sind in C-Programmen (wie in BASIC auch) dazu da, um Werte aufzunehmen. Man kann diesen Variablen frei wählbare Namen geben. Bei der 'Taufe' von Variablen sollten Sie stets Namen wählen, die über den in der Variablen abgelegten Wert etwas aussagen. Die Bildung solcher aussagekräftiger ("mnemotechnischer") Namen ist in C besonders leicht.

Ein Name darf aus Buchstaben, Ziffern und dem Unterstrich "_" zusammengesetzt sein. Die Buchstaben können klein- oder großgeschrieben sein, wobei C den Unterschied beachtet. Obwohl vom Standard dies nicht ausdrücklich gefordert wird, sollten Sie Ihre Namen stets mit einem (großen oder kleinen) Buchstaben beginnen lassen, da einige Compiler auf dieser Regelung bestehen. Auch ist für die Namenslänge nichts bestimmtes vorgeschrieben. Die in diesem Buch verfolgte Konvention dürfte jedoch bei keinem Compiler auf Probleme stoßen: Namen können prinzipiell beliebig lang sein, sollten jedoch in den ersten 8 Zeichen eindeutig sein. Der Unterstrich wird zur optischen Auflockerung in C-Programmen häufig eingesetzt. Da in C-Namen keine Leerzeichen enthalten sein dürfen (denken Sie an die Abbildung 2.3!), dient er oft als Leerzeichen-Ersatz.

Es folgen einige Beispiele für richtig und falsch gebildete Namen mit Erläuterung:

a) variable
b) Variable
c) 1_var
d) var/name
e) variable_1
f) variable_2
g) do
h) Do
i) dodo

Die Beispiele a, b, h und i sind korrekt gebildet; beachten Sie, daß es sich bei a) und b) für C um zwei verschiedene Variablen handelt, da sie sich in der Schreibweise des Anfangsbuchstabens unterscheiden.

Das Beispiel d ist falsch, da der Name ein nicht erlaubtes Zeichen (den Schrägstrich) aufweist. Alle anderen Beispiele sind nicht unbedingt falsch, aber doch problematisch. Beispiel c beginnt mit einer Ziffer; dies bereitet manchen Compilern Kopfzerbrechen, weswegen Sie es vermeiden sollten. Die Beispiele e) und f) sind zwar korrekt gebildet, in den ersten 8 Zeichen jedoch nicht zu unterscheiden. Kommen diese beiden Variablen in demselben Programm vor, dann kann das zu Problemen führen.

Daß das so harmlos erscheinende Beispiel g) ebenfalls nicht in Ordnung ist, läßt sich aus dem bisher Gesagten nicht erkennen. Dennoch: *do* ist als selbstgewählter Name nicht erlaubt, da dieser Name bereits vergeben ist, denn *do* ist eines der Schlüsselworte von C.

C ist ja eine Programmiersprache; und eine Sprache hat einen Wortschatz. Der Wortschatz von C ist ziemlich klein und deshalb verhältnismäßig rasch zu erlernen. Sie finden ihn in der Abbildung 2.10. Dies ist eine List der Worte, die der C-Compiler versteht, der sogenannten "Schlüsselworte" (engl.: "keywords"). Die Mächtigkeit von C rührt daher, daß zu diesen 'eingebauten' Worten auch selbstdefinierte hinzukommen dürfen. Sie können nicht nur Variablen mit Namen belegen, sondern auch eigene Funktionen schreiben und diesen einen Namen geben; das Programm 2.2 hat dies bereits gezeigt.

```
int             extern          else
char            register        for
float           typedef         do
double          static          while
struct          goto            switch
union           return          case
long            sizeof          default
short           break           entry
unsigned        continue        auto
if
```

**Abb. 2.10:** *Schlüsselworte von C*

Aber bei der Wahl aller selbstdefinierten Namen - gleichgültig, ob für Funktionen oder Variable - müssen Sie darauf achten, nicht in Konflikt mit den Schlüsselworten zu geraten. Wie Sie der Abbildung 2.10 entnehmen können, sind alle Schlüsselwörter kleingeschrieben; daran müssen Sie sich halten. Wie Sie ebenfalls wissen, unterscheidet C zwischen groß- und kleingeschriebenen Namen. Deshalb ist das Beispiel h) aus der obigen Sammlung in Ordnung.

Ebenfalls in Ordnung ist Beispiel i). Es wurde nur mit aufgenommen, weil in manchen Programmiersprachen (und auch in einigen BASIC-Dialekten) in Variablennamen Schlüsselwörter auch nicht als Teil enthalten sein dürfen. C erlegt Ihnen diese Beschränkung nicht auf.

In C können Variablen nur verwendet werden, wenn sie zuvor deklariert wurden. Prinzipiell könnte diese Deklaration irgendwo im Programm vor der erstmaligen Verwendung der Variablen erfolgen. Aber es ist übersichtlicher, die Deklaration der in einer Funktion benötigten Variablen an den

Anfang der Funktion zu setzen. Auch gehen einige Compiler - so auch der des ATARI-C - davon aus, daß Sie sich an diese Konvention halten. das ATARI-C erlaubt Ihnen nicht, nach der ersten ausführbaren Anweisung im Programm noch Variablen zu deklarieren.

So habe ich mich auch im Programm 2.9 an diese Konvention gehalten: der Anweisungsblock beginnt mit drei Deklarationen.

In einer Variablendeklaration macht man dem Compiler (unter anderem) bekannt, welche Namen man für die Variablen zu verwenden gedenkt und von welchem Datentyp diese Variablen sein sollen. Das Kapitel "Variablennamen" wurde eben erschöpfend abgehandelt. Wenn Sie sich von Ihrer Erschöpfung erholt haben, dann können wir uns dem Thema "Datentyp" zuwenden.

Daten sind 'Dinge', mit denen der Computer etwas macht. Mit verschiedenen Dingen kann man verschiedenes machen; so kann man mit Zahlen rechnen, mit Buchstaben aber nicht. Andererseits kann man aus einem großgeschriebenem Buchstaben einen kleingeschriebenen machen; bei Zahlen ist dies ein sinnloses Unterfangen. Man faßt in der EDV die Objekte, mit denen dieselben Operationen möglich sind, in einem Datentyp zusammen. Die wichtigsten Datentypen von C sind Zahlen, Zeichen und Zeiger (hübscher Stabreim!). Um den letzten Datentyp, die Zeiger zu verstehen, ist einiges an Vorkenntnissen nötig. Zeiger und alles, was damit zusammenhängt, bleiben daher dem Kapitel 5 vorenthalten.

Anders ist's mit Zahlen und Zeichen; die kennt jeder. Oder meint es jedenfalls. Denn der Zahlen gibt es mehrere. Bereits von klein auf vertraut sind uns die ganzen Zahlen (1,2,3 usw.), die auch schon in zwei Spielarten 'in der Natur' vorkommen: als positive und als negative Zahlen. Als man begann, uns in der Schule mit der Division zu quälen, da wurde es unerläßlich, sich noch mit einer weiteren Zahlengattung vertraut zu machen: die Zahlen mit Nachkommastellen (z.B. die allgegenwärtige Kreiszahl 3,1415). In einem Computer gehören diese Zahlen jeweils einem eigenen Datentyp an.

Als ob die Materie nicht schon kompliziert genug wäre, unterscheiden Computer auch noch Zahlen von unterschiedlicher Größe und Genauigkeit. Aber dies wird Sie noch in einem späteren Kapitel (Kapitel 4) beschäftigen. Die Variablen, die in Programm 2.9 gebraucht werden, sollen nur ganz bescheidene ganze Zahlen aufnehmen. Im Englischen heißt eine ganze Zahl "integer"; jetzt können Sie sich auch denken, warum im Programm 2.9 vor den Variablennamen immer *int* steht! Sie sehen: C neigt zur Kürze und schreibt alles klein.

Programm 29 hebt also mit der Deklaration dreier Integer-Variablen an, die die Namen *Summe*, *Zahl_1* und *Zahl_2* tragen. Eine Variablen-Deklaration ist eine Anweisung wie jede andere auch und muß deshalb mit einem Strichpunkt abgeschlossen werden. Allerdings ist es keine Anweisung, die etwas 'tut'; vielmehr hält sie nur Informationen für den Compiler bereit, damit dieser Speicherplatz für die Daten besorgen kann. Dieser Speicherplatz kann dann mit Hilfe einer Wertzuweisung mit Informationen vollgeschrieben werden; womit wir schon beim nächsten Thema wären...

### 2.6.2 Die Wertzuweisung; einfache und zusammengesetzte Ausdrücke

Anweisungen, die etwas tun, finden sich erst im zweiten Abschnitt des Programms 2.9; dort wird auf etwas umständliche Weise die Summe zweier Zahlen berechnet. Dazu werden die beiden Summanden 5 und 13 an die beiden Variablen *Zahl_1* und *Zahl_2* zugewiesen. Genauer: in den Speicherplatz, den der Compiler für diese beiden Variablen reserviert hat, werden diese beiden Werte eingetragen. Da Speicherplätze später noch eine große Rolle spielen werden, will ich Ihnen diesen Sachverhalt mit einem Speicherdiagramm verdeutlichen; Sie finden es in Abbildung 2.11.

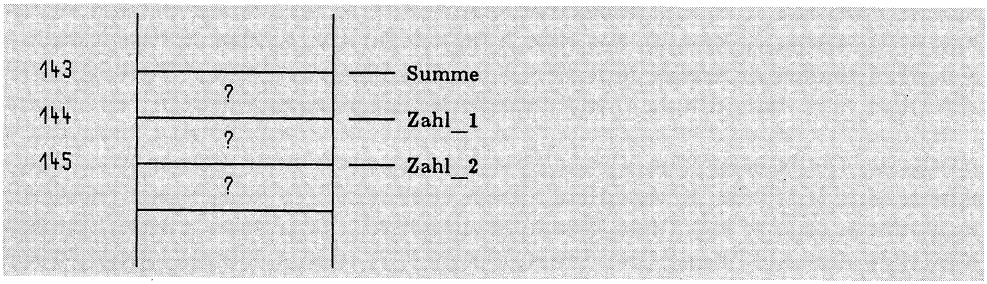



**Abb. 2.11:** *Speicherdiagramm der Variablendeklaration*

Wie Sie wissen, ist der Arbeitsspeicher eines Computers in einzelne Speicherzellen unterteilt, die fortlaufend durchnumeriert sind. In der Abbildung 2.11 sind die Speicherzellen durch Kästchen symbolisiert, die zur Aufnahme eines Wertes dienen können. Die Nummer einer Speicherzelle wird auch deren "Adresse" genannt; über diese Adresse kann eine Speicherzelle angesprochen werden, um einen Wert in ihr zu speichern oder um nachzusehen, welchen Wert sie enthält. Natürlich sind die in der Abbildung stehenden Adressen fiktiv; der Compiler wird mit Sicherheit andere Adressen als die hier gezeigten für die Variablen benutzen (mit Sicherheit deshalb, weil sich an dieser Stelle im ATARI wichtige Tabellen des Betriebssystems befinden. Aber das ist eine andere Geschichte...)

In der Frühzeit der Computerei mußten Programmierer noch mit den numerischen Adressen des Arbeitsspeichers umgehen. Dieses Verfahren ist ungemein umständlich und fehleranfällig. Einer der Vorteile höherer Programmiersprachen besteht darin, anstatt mit numerischen Adressen mit symbolischen Adressen arbeiten zu können. Die in einem C-Programm vereinbarten Variablen sind nichts anderes als solche symbolische Adressen.

Wenn Sie in C eine Variable deklarieren, dann besorgt sich der Compiler irgendwo im Arbeitsspeicher des Computers Platz für die Variablen und merkt sich, an welchen numerischen Adressen dieser Platz zu finden ist. Die Abbildung 2.11 gibt somit einen Ausschnitt aus dem Arbeitsspeicher wieder, wie er nach der Deklaration aussehen könnte. Der Compiler hat sich für die drei gewünschten Variablen drei aufeinanderfolgende Speicherstellen - beginnend bei der numerischen Adresse 143 - besorgt und hat sich gemerkt, daß die Speicherstelle 143 jetzt auf den Namen *Summe* getauft ist, 144 auf *Zahl_1* usw. Die numerischen Adressen sind völlig willkürlich gewählt; der Compiler nimmt eben den nächsten Speicherplatz, der gerade frei ist. In der Regel haben Sie als Programmierer auch keinen Einfluß darauf, an welchen Adressen Ihre Variablen abgelegt werden (es sei denn, Sie arbeiten mit Zeigern; damit können Sie Werte an von Ihnen frei wählbaren Adressen speichern). Wo der Compiler Ihre Variablen hinlegt, müssen Sie aber auch gar nicht wissen, denn Sie arbeiten ja mit symbolischen Adressen, also mit Ihren selbstgewählten Variablennamen.

Nach der Deklaration der Variablen enthalten diese noch keinen Wert. Genauer: ihr Wert ist undefiniert (denn daß sich an einer Speicherstelle tatsächlich nichts befindet, ist nicht möglich). Deswegen stehen in den Kästchen der Abbildung 2.11 Fragezeichen. Erst durch Wertzuweisung kann man die Kästchen füllen; in Abbildung 2.12 ist dies teilweise geschehen:

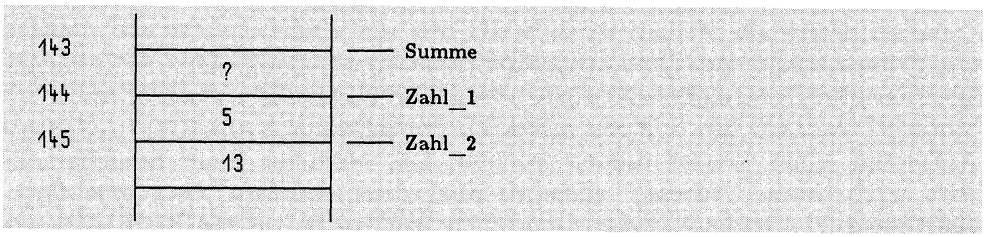



**Abb. 2.12:** *Speicherdiagramm der einfachen Wertzuweisung*

Das Diagramm in dieser Abbildung gibt die Verhältnisse wieder, die nach Ausführung der beiden Anweisungen

```
Zahl_1 = 5;
Zahl_2 = 13;
```

im Speicher herrschen. Diese beiden Anweisungen sind sogenannte "Wertzuweisungen"; sie bewirken, daß einer Variablen ein (definierter) Wert zugewiesen wird. Wertzuweisungen haben immer den gleichen Aufbau:

```
<lvalue> = <ausdruck>;
```

Das Gleichheitszeichen "=" signalisiert die Zuweisung. Links vom Gleichheitszeichen muß etwas stehen, dem ein Wert zugewiesen werden kann. Im Jargon der C-Programmierer bezeichnet man das als *lvalue* (sprich: "Ell-Vällju"); Ihr Compiler wird Ihnen gelegentlich Fehlermeldungen präsentieren, in denen dieser Terminus Technicus auftaucht. Jetzt wissen Sie, was das bedeutet.

Als *lvalues* kommen bisher nur einfache Variablen in Frage; Sie werden aber noch sehen, daß die Gattung der *lvalues* äußerst artenreich ist und einige exotische Vertreter aufzuweisen hat.

Rechts vom Zuweisungszeichen "=" muß - so entnehmen Sie dem Diagramm - ein "Ausdruck" stehen. In der Computerei ist ein Ausdruck alles, was einen Wert (z.B. einen Zahlen- oder Zeichenwert) hat bzw. durch Ausrechnen einen solchen ergibt. Die Ausdrücke in den beiden obigen Anweisungen sind von der einfachsten denkbaren Art, es sind nämlich die "Konstanten" 5 und 13. Daß es auch noch andere Ausdrücke gibt, zeigt bereits die nächste Anweisung von Programm 2.10. Ihre Wirkung gibt die Abbildung 2.13 wieder.

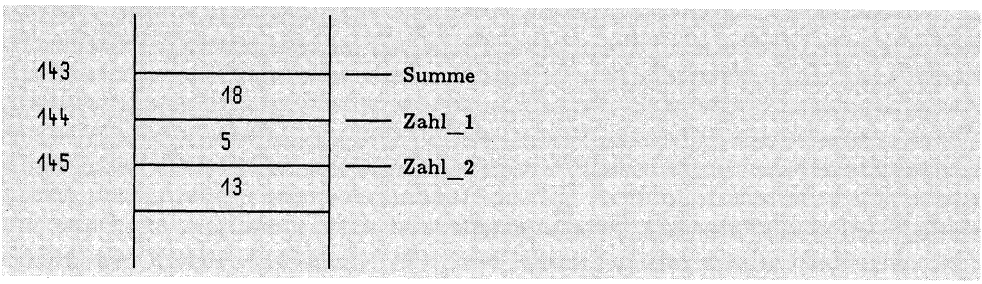



**Abb. 2.13:** *Speicherdiagramm der arithmetischen Wertzuweisung*

Der Ausdruck

```
Summe = Zahl_1 + Zahl_2;
```

hat als *lvalue* die Variable *Summe*, weist also an die betreffende Speicherstelle einen Wert zu. Der zugewiesene Wert ist jedoch diesmal keine Konstante, sondern ergibt sich durch Ausrechnen, in diesem Fall durch Addition zweier Zahlen; dies signalisiert das wohlvertraute Pluszeichen. Die zu addierenden Zahlen sind jedoch nicht direkt - als Konstanten - hin-

geschrieben. Stattdessen wurden die Variablen angeführt, die als Wert die zu addierenden Zahlen enthalten. Natürlich hätte man genausogut

```
Summe = 5 + 13;
```

schreiben und damit das gleiche bewirken können. Dann hätte ich allerdings nicht demonstrieren können, daß in Ausdrücken neben Konstanten auch Variablen vorkommen dürfen und daß zur Berechnung des Ausdrucks der in den Variablen gespeicherte Wert herangezogen wird. Natürlich können Sie eine Variable nur dann in Ausdrücken verwenden, wenn sie zuvor einen Wert erhalten hat.

Ausdrücke können einfach sein, wie die ersten beiden im Programm 2.10; sie können aber auch zusammengesetzt sein, wie der dritte und eben besprochene. In zusamengesetzten Ausdrücken gibt es immer einen Operator, der zwei oder mehrere andere Ausdrücke miteinander kombiniert, um daraus einen neuen Wert zu produzieren. Hier kombiniert der Operator '+' die beiden Ausdrücke *Zahl_1* und *Zahl_2*, um deren Summe (18) zu produzieren. Weil in dem zusammengesetzten Ausdruck gerechnet wird (oder, vornehmer ausgedrückt: weil '+' ein arithmetischer Operator ist), nennt man ihn auch einen arithmetischen Ausdruck. All diese Begriffe versammelt noch einmal die Abbildung 2.14.

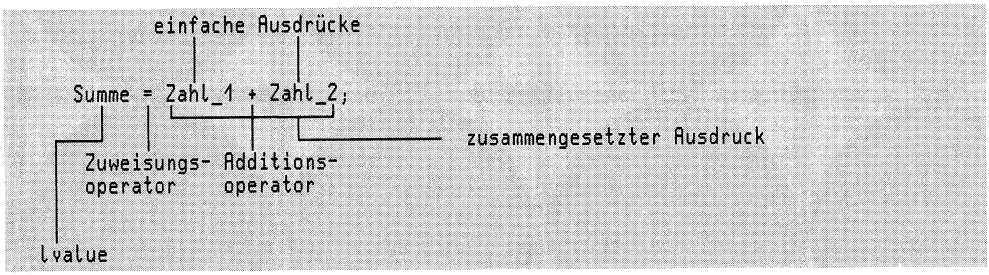



**Abb. 2.14:** *Anatomie einer Wertzuweisung*

### 2.6.3 Ausgabe von Ergebnissen mit printf

Nachdem die Maschine nun ausgerechnet hat, wieviel 5 plus 13 ist, soll sie uns das auch mitteilen. Das Ergebnis muß also ausgegeben werden, und dazu ist *printf* da. Wurde damit aber bisher ausschließlich Meldungen - also feststehende Werte oder Konstanten - ausgegeben, so muß *printf* jetzt etwas anzeigen, was zum Zeitpunkt der Programmerstellung noch nicht bekannt ist (nämlich das Ergebnis der Addition). Es tut dies wie folgt:

```
printf("\nDie Summe von %d und %d ist %d.",Zahl_1,Zahl_2,Summe);
```

So steht das allerdings nicht im Programm 2.10; dort habe ich mir zunutze gemacht, daß man in C so gut wie überall Kommentare einstreuen kann. Doch darüber später mehr. Obige Version ist jedenfalls für den C-Compiler gleichbedeutend mit der im Programm und soll daher zuerst besprochen werden.

Wenn Sie Programm 2.10 laufen lassen, dann erhalten Sie wie erwartet die Meldung

```
Die Summe von 5 und 13 ist 18.
```

aber der Wortlaut dieser Meldung ist nicht Argument von *printf*! Dort findet sich vielmehr dreimal die Zeichenfolge "%d"; außerdem hat *printf* diesmal nicht nur einen String als Argument, sondern es folgen noch die drei bereits bekannten Variablen.

An dem Ergebnis des Programms sehen Sie, daß *printf* neben konstanten Meldungen auch die Werte von Variablen ausgeben kann, und - was noch mehr ist - Text und Variablenwerte auf einfache Weise gemischt werden können. Dazu schreiben Sie als erstes Argument von *printf* ganz einfach den gewünschten Meldungstext, setzen aber an die Stelle, an der die Variablen erscheinen sollen, die Zeichenfolge "%d".

Das Prozentzeichen ist ein weiteres Escape-Zeichen für *printf*. Wo es steht, soll von *printf* bei der Ausgabe der Wert einer Variablen eingesetzt werden. Auf das Prozentzeichen folgt ein Buchstabe, der angibt, von welchem Typ die Variable ist: "d" steht für "dezimal" und bedeutet, daß eine Zahl (eine Integer) in dezimaler Darstellung (so wie Sie es gewohnt sind) ausgegeben werden soll. *printf* liest sich den String durch und sieht die drei Prozentzeichen; daran erkennt es, daß 3 Variablen ausgegeben werden müssen. Sie müssen der Funktion jedoch noch sagen, welche Variablen das sind. Dazu geben Sie einfach weitere Argumente in den Argumentklammern von *printf* an. Das erste Argument nach dem String bezieht sich auf das erste "%d" im String, das zweite Argument nach dem String auf das zweite "%d" usw.

Hat eine Funktion mehrere Argumente, so müssen Sie diese durch Kommas trennen. Wegen der Formatfreiheit von C ist es jedoch möglich, dazwischen auch beliebig viele Leerzeichen, Zeilenvorschübe und sogar Kommentare einzustreuen. Genau das habe ich im Programm 2.10 gemacht: jedes *printf*-Argument steht auf einer eigenen Zeile, alle sind sie ordentlich untereinander aufgereiht und das erhöht die Lesbarkeit.

Ehe Sie die Anatomie einer erweiterten *printf*-Anweisung in der Abbildung 2.15 studieren, will ich noch einmal die Informationen zusammenfassen, die Sie bisher über *printf* erhalten haben:

- *printf* übernimmt die Ausgabe von Meldungen auf den Bildschirm.
- Die auszugebenden Meldungen müssen in Form eines Strings erstes Argument von *printf* sein; dieser String wird auch "Kontrollstring" genannt.
- Der Kontrollstring kann Escape-Sequenzen für Steuerzeichen enthalten; als Escape-Zeichen dient der umgekehrte Schrägstrich; die zulässigen Folgezeichen finden Sie in Abbildung 2.8.
- Der Kontrollstring kann Escape-Zeichen für die Ausgabe von Variablen enthalten; als Escape-Zeichen dient das Prozentzeichen. Als Folgezeichen dient "d" für die Ausgabe von Dezimalzahlen.
- Sollen Variablen ausgegeben werden (der Kontrollstring enthält Prozentzeichen), dann müssen diese als zusätzliche Argumente von *printf* hinter dem Kontrollstring folgen. Achten Sie stets darauf, daß Sie auch ebensoviele Variablen angeben, wie im Kontrollstring Prozentzeichen auftauchen. Wenn Sie zuwenig Argumente liefern, dann tut Ihr Computer Seltsames!

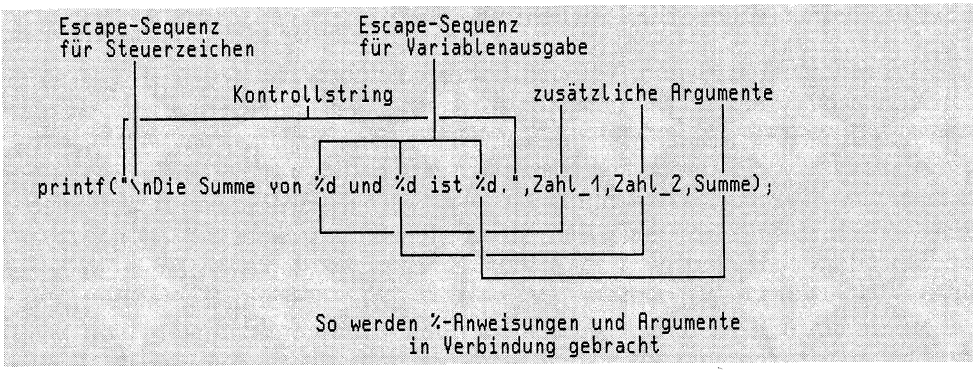



**Abb. 2.15:** *Erweiterte* printf-*Anweisung*

# 3 Einfache interaktive Programmierung

Im letzten Kapitel haben Sie die Möglichkeit zur Ausgabe von Daten mit *printf* kennengelernt. Die mit den bisher vorgestellten Sprachmitteln möglichen Programme ähneln aber ein wenig der Unterhaltung mit einem Gehörlosen: der Computer denkt eine Weile vor sich hin und gibt dann was aus. Er hört jedoch nicht auf Sie.

So richtig schön wird die Sache erst, wenn man dem Computer im Dialog etwas sagen kann und er dann darauf mit einem Ergebnis reagiert. Dies umschreibt man mit dem schönen Fachausdruck "interaktive Programmierung". Elementare Möglichkeiten dieses Programmierstils werden in diesem Kapitel besprochen.

## 3.1 Eingabe mit scanf

"Der Rechner rechnet", hieß es im Kapitel 2.6. Aber was, werden Sie sagen, ist das für eine Rechnerei, wenn er immer nur dieselben beiden Zahlen zusammenzählt? Die Daten, mit denen das Programm 2.10 arbeitet, sind ja als Konstanten im Programm enthalten, sind also gleichsam in diesem 'fest verdrahtet'. Interessanter wird die Sache doch erst, wenn man verschiedene Zahlen addieren kann, ohne dazu jedesmal das Programm neuschreiben zu müssen. In dieser Situation schafft das Programm 3.1 Abhilfe.

```
                                                /*****************************/
main()                                          /* Einfache Arithmetik.      */
{                                               /*****************************/
    int Summe;                                  /*  Variablen-Deklaration.   */
    int Zahl_1;                                 /*                           */
    int Zahl_2;                                 /*                           */
                                                /*****************************/
    printf("\nZahl 1: "); scanf("%d",&Zahl_1);  /*  Wertzuweisung an die     */
    printf("\nZahl 2: "); scanf("%d",&Zahl_2);  /*  Variablen durch Benut-   */
    Summe = Zahl_1 + Zahl_2;                    /*  zereingabe!              */
                                                /*****************************/
    printf("\nDie Summe von %d und %d ist %d.",  /*  Ausgabe Ergebnis.        */
           Zahl_1,                              /*                           */
           Zahl_2,                              /*                           */
           Summe);                              /*                           */
                                                /*****************************/
}
```

**Prog. 3.1:** *Werteingabe durch* scanf

**Anmerkung:** In der mir verfügbaren Version des ATARI-C funktionierte *scanf* nicht richtig. Sollten Sie mit diesem Compiler arbeiten und feststellen, daß das Programm 3.1 nicht wie gewünscht reagiert, dann ist dieser Fehler auch noch in Ihrer Version enthalten. Lesen Sie dann dieses Kapitel durch, machen Sie jedoch die Beispiele so, wie im Abschnitt 3.1.1 beschrieben!

Im Vergleich zum letzten Programmbeispiel hat sich hier im Mittelteil (dem Verarbeitungsteil) etwas geändert. Wenn Sie das Programm laufen lassen, werden Sie sehen, daß Sie durch die Meldung "Zahl 1:" zur Eingabe einer Zahl aufgefordert werden. Danach hält das Programm inne und der Computer wartet, daß Sie ihn mit einer Zahl versorgen. Sie können jetzt eine beliebige Zahl eintippen und die Eingabe mit der *Return*-Taste abschließen. Erst dann werden Sie durch eine weitere Meldung aufgefordert, die zweite Zahl einzugeben. Sie verfahren wie eben beschrieben und erhalten als Resultat Ihrer Mühen in der bereits vom letzten Programm bekannten Weise das Ergebnis angezeigt.

Wie die Meldungen erzeugt werden, die zur Eingabe auffordern und das Ergebnis darstellen, ist kein Geheimnis mehr: mit *printf*. Das Einlesen der Werte von der Tastatur besorgt jedoch eine neue Funktion, ein naher Verwandter von *printf* mit dem Namen *scanf*. Die Verwandschaft ist zweifach: beide Funktionen sind für interaktive Programmierung zuständig; beide Funktionen arbeiten mit einem Kontrollstring.

"Interaktive Programmierung" ist die Würze der Micro-Computerei. Sie ermöglicht dem Benutzer einen Dialog mit seinem Gerät: der Computer zeigt Informationen an, auf die der Benutzer reagieren kann, diese Reaktionen werden vom Programm sofort verarbeitet, das Ergebnis angezeigt, der Benutzer kann wieder darauf reagieren usw. Paradebeispiel für interaktive Programmierung sind Textprogramme (wie z.B. der Editor, mit dem Sie Ihre C-Programme erstellen), die den (von der Tastatur) eingegebenen Text auf dem Bildschirm darstellen und je nach Reaktion des Benutzers (Eingabe weiterer Zeichen oder Kommandos zur Bearbeitung des Textes) in aktualisierter Form wieder ausgeben. Bei der interaktiven Programmierung muß der Programmierer Ausgabe- und Eingabegeräte bedienen können.

Das klassische Ausgabegerät für Micro-Computer ist der Bildschirm; von dessen Bedienung mit *printf* wissen Sie schon einiges. Als Eingaberät für die interaktive Programmierung dient primär die Tastatur, aber auch fortschrittliche Medien wie z.B. die Maus. Zur Verarbeitung von Tastatureingaben gibt es *scanf*; die Verarbeitung von Maus-Informationen erfordert im Vergleich dazu sehr fortgeschrittene Kenntnisse und kann in diesem Buch noch nicht behandelt werden.

Wenn Sie Daten von der Tastatur einlesen wollen, dann müssen Sie *scanf* mitteilen, von welchem Typ diese Daten sind und wo Sie sich diese Daten aufheben wollen. Ersteres tun Sie mit einem Kontrollstring; letzteres durch Angabe einer oder mehrerer Variablenadressen vom passenden Typ.

*scanf* ist - wie Sie noch sehen werden - sehr vielseitig und kann Daten unterschiedlichen Typs einlesen, darunter Zahlen in verschiedener Darstellung und Genauigkeit, Zeichen und Zeichenketten. Der Kontrollstring sagt der Funktion, auf welche Daten sie sich einzustellen hat. Mit *%d* wird im Programm 3.1 mitgeteilt, daß eine einfache Zahl in Dezimalschreibweise erwartet wird. Da *%d* angibt, welches Format die zu erwartenden Daten haben, spricht man in diesem Zusammenhang auch von einer "Formatanweisung". Bei Formatanweisungen ist offensichtlich derselbe Escape-Mechanismus am Werk, der bei der Ausgabe mit *printf* benutzt wird. Der Benutzer des Programms muß die Eingabe seiner Informationen übrigens mit der *Return*-Taste abschließen: daran sollten Sie auch beim Testen des Programmbeispiels denken.

Im Kontrollstring erfährt *scanf*, daß es eine Zahl einlesen soll; jetzt muß es nur noch wissen, wo es die gelesene Zahl hintun soll. Dazu benötigt es, nein! nicht eine Variable, sondern die Adresse einer Variablen.

Das ist eigentlich auch ganz logisch. Denn wenn eine Variable Argument einer Funktion ist, dann wird der momentan in der Variablen gespeicherte Wert zur Verarbeitung herangezogen. So ist es auch bei der abschließenden *printf*-Anweisung von Programm 3.1: hier wird mit den Argumenten *Zahl_1*, *Zahl_2*, *Summe* der gerade in diesen Variablen gespeicherte Wert angesprochen und ausgegeben. Aber in der Funktion *scanf* wollen Sie ja nicht, daß der Wert der Variablen manipuliert wird (die hat zu diesem Zeitpunkt ja noch gar keinen!), sondern die Variable selbst, indem sie einen solchen Wert erst erhält. Aus der Abbildung 2.12 wissen Sie bereits, daß eine Variable nichts anderes als ein frei wählbarer Name für eine Speicherstelle - eine Adresse - ist. *scanf* soll diese Adresse manipulieren (soll an ihr etwas ablegen), und deshalb muß man sie *scanf* mitteilen. Aber wie, wenn man sie nicht kennt?! Müssen Sie sich jetzt doch wieder mit numerischen Adressen herumschlagen?

Zum Glück nicht! Denn in C gibt es - darin ist die Sprache einzigartig - eine einfache Möglichkeit, sich die Adresse einer Variablen zu besorgen, nämlich den *&*-Operator. Schreiben Sie *&* vor eine Variable, dann hat dies im Programm den Effekt, daß statt der Variablen ihre Adresse benutzt wird.

Sollte Ihnen bei dieser Erklärung etwas mulmig geworden sein, so ist das nicht tragisch. Sie werden bald selbst Funktionen schreiben können, die

Variablenadressen manipulieren und dann werden Ihnen die Zusammenhänge sicher klar. Merken Sie sich vorerst ganz einfach: die Variablen, in denen *scanf* die eingelesenen Werte ablegen soll, werden in der Argumentklammer von *scanf* mit einem vorgestellten &-Zeichen dekoriert.

& ist das erste Beispiel für einen einstelligen Operator; mit der Addition (+) findet sich im Programm 3.1 auch ein Vertreter der zweistelligen Operationen. Das Kapitel 3.2 stellt Ihnen weitere zweistellige Operatoren vor.

```
iscan(var)
      int *var;

{  char buff[80];
   char *get_s();

   *var = atoi(get_s(buff));
}



fscan(var)
      float *var;

{  char buff[80];
   double atof();
   char *gets_s();

   *var = atof(get_s(buff));
}

extern long gemdos();



char *get_s(line)
     char *line;

{  register char *cp;
   register char i;

   cp = line;
   while ((i = gemdos(0x1)) != '\r')
      *cp++ = i;
   *cp = 0;
   if (cp == line)
      return (char *) 0;
   return line;
}
```

```
/*************************/
/* Wert fuer Integer-Va- */
/*  riable von der Tasta- */
/*  tur besorgen.        */
/*************************/
/*                       */
/*                       */
/* Eingebautes gets funk- */
/*  tioniert nicht!      */
/*************************/

/*************************/
/* Wert fuer Float-Varia- */
/*  ble von Tastatur holen.*/
/*************************/
/*                       */
/*                       */
/*                       */
/*                       */
/*                       */
/*************************/
/*                       */
/* Fuer Einbindung des Be- */
/*  triebssystems.       */
/*                       */
/*************************/
/* String von Tastatur le- */
/*  sen.                 */
/*************************/
/*                       */
/*                       */
/* Buffer-Anfang merken. */
/*  Zeichen lesen ueber  */
/*  Funktion des Betriebs- */
/*  systems.             */
/* String abschliessen.  */
/* Nichts gelesen?       */
/*                       */
/* String-Anfang zurueck. */
/*************************/
```

**Abb. 3.1:** *Ein Ersatz für* scanf

### 3.1.1 Ein Ersatz für scanf

Leser, deren C über ein funktionierendes *scanf* verfügt, können diesen Abschnitt beim ersten Durchlesen getrost überspringen. Wenn Sie bereits fortgeschrittener in der C-Programmierung sind, dann können Sie ja das Programm in der Abbildung 3.1 studieren.

Wenn bei Ihrem Exemplar des ATARI-C *scanf* nicht funktioniert, dann haben Sie mit den Informationen in diesem Abschnitt die Möglichkeit, die Beispielprogramme sinngemäß nachzuvollziehen. Dazu müssen Sie aber an das Ende jedes Beispielprogramms den 'Anhang' mit Aufnehmen, den Sie in der Abbildung 3.1 sehen. Dies ist jedoch kein großes Problem, da alle guten Editoren (so auch der METACOMCO-Editor des Entwicklungssystems) die Möglichkeit vorsehen, in eine Datei andere Dateien einzumischen. Schreiben Sie also die Anweisungen der Abbildung 3.1 in eine eigene Datei (die Sie SCAN.C nennen können) und fügen Sie diese Datei dann an das Ende jedes Beispielprogramms des dritten Kapitels an. Auf die Kommentare in der Abbildung können Sie natürlich verzichten, um die Tipparbeit zu reduzieren.

Sie müssen dieses umständliche Verfahren zum Glück nur für einige Beispiele dieses Kapitels anwenden. In den folgenden Kapiteln wird kein Gebrauch von *scanf* mehr gemacht.

Die Funktionen *iscan* und *fscan* stellen einen - allerdings sehr schwachen - Ersatz für die Standard-Funktion *scanf* dar. Aber Sie können wenigstens die Beispiele nachvollziehen. Eine Erklärung dessen, was in der Abbildung 3.1 getan wird, ist aus verständlichen Gründen jetzt noch nicht möglich. Wenn Sie jedoch die ersten fünf Kapitel dieses Buches durchgearbeitet haben, dann können Sie sich die Abbildung getrost noch einmal zu Gemüte führen.

*iscan* und *fscan* sind zwei spezialisierte Funktionen, die in einer Variablen einen Integer- bzw. einen Gleitpunktwert ablegen. Sie erhalten als Wert eine Variablenadresse und gleichen darin *scanf*. Sie werten jedoch keinen Kontrollstring aus. Wenn Sie mit *iscan* arbeiten wolle, dann müssen Sie das Programm 3.1 an einer Stelle ändern. Statt

```
.
.
.                                        /*************************/
printf("\nZahl 1: "); scanf("%d",&Zahl_1);   /*  Wertzuweisung an die  */
printf("\nZahl 2: "); scanf("%d",&Zahl_2);   /*  Variablen durch Benut- */
.
.
.
```

schreiben Sie:

```
.
.
.
printf("\nZahl 1: "); iscan(&Zahl_1);          /*  Wertzuweisung an die   */
printf("\nZahl 2: "); iscan(&Zahl_2);          /*  Variablen durch Benut- */
.
.
.
```

Der Rest des Programms kann unverändert übernommen werden.

## 3.2 Arithmetische Operatoren

Natürlich kann man mit zwei Zahlen viel mehr machen, als sie zu addieren. Selbstverständlich können Sie auch in C subtrahieren, multiplizieren und dividieren. All diese Operationen führt Ihnen das Programm 3.2 vor.

```
                                                /*************************/
main()                                          /* Die Grundrechnungsarten */
{                                               /*************************/
    float Ergebnis;                             /*  Deklaration von drei   */
    float Zahl_1;                               /*  Gleitpunkt-Variablen.  */
    float Zahl_2;                               /*                         */
                                                /*************************/
    printf("\nZahl 1: "); scanf("%f",&Zahl_1);  /*  Interaktive Wertzuwei- */
    printf("\nZahl 2: "); scanf("%f",&Zahl_2);  /*  sung (Gleitpunktzahl!) */
                                                /*************************/
    Ergebnis = Zahl_1 + Zahl_2;                 /*  Bildung von Summe,     */
    printf("\nSumme: %f",Ergebnis);             /*  Differenz, Produkt,    */
                                                /*  und Quotienten.        */
    Ergebnis = Zahl_1 - Zahl_2;                 /*                         */
    printf("\nDifferenz: %f",Ergebnis);         /*  Da bei der Division    */
                                                /*  nicht-ganzzahlige Er-  */
    Ergebnis = Zahl_1 * Zahl_2;                 /*  gebnisse auftreten     */
    printf("\nProdukt: %f",Ergebnis);           /*  koennen, ist die Be-   */
                                                /*  nutzung von Gleit-     */
    Ergebnis = Zahl_1 / Zahl_2;                 /*  punktvariablen und     */
    printf("\nQuotient: %f",Ergebnis);          /*  den entsprechenden     */
                                                /*  Formatangaben in       */
                                                /*  "printf" noetig.       */
                                                /*************************/
}
```

**Prog. 3.2:** *Die Grundrechnungsarten*

**Anmerkung:** Die Leidensgenossen mit defektem *scanf* müssen für die Eingabe diesmal die beiden folgenden Anweisungen verwenden:

```
printf("\nZahl 1: "); fscan(&Zahl_1);          /*  Interaktive Wertzuwei- */
printf("\nZahl 2: "); fscan(&Zahl_2);          /*  sung (Gleitpunktzahl!) */
```

Die Bildung von Summe, Differenz, Produkt und Quotient bietet keine Überraschungen. Wie Sie sehen können, verwendet C dafür die (zweistelligen) Operatoren *+*, *-*, * und */*. Daß für Multiplikation ein Stern und für Division ein Schrägstrich geschrieben wird, ist bei den meisten Programmiersprachen üblich.

Aber die Variablendeklarationen am Programmanfang und ebenso die Kontrollstrings für *printf* und *scanf* haben sich geändert. Programm 3.2 arbeitet nicht mehr mit drei Integers, sondern verwendet jetzt Gleitpunktzahlen. Diese werden im Englischen "floating point numbers" genannt, weswegen bei der Deklaration die Typangabe *float* steht; C liebt's eben kurz.

Gleitpunktzahlen sind Zahlen mit Nachkommastellen. Computer haben im Umgang mit Gleitpunktzahlen eine eigene Art, die sich deutlich von der Arbeit mit Ganzzahlen (Integers) unterscheidet. Das geht sogar so weit, daß für ganze Zahlen und Gleitpunktzahlen gesonderte Rechenverfahren benutzt werden; der Fachmann spricht in diesem Zusammenhang von "Integerarithmetik" bzw. "Gleitpunktarithmetik". Ich will hier vorerst nicht weiter ins Detail gehen; Sie können sich jedoch merken, daß Gleitpunktzahlen mehr Speicherplatz beanspruchen und daß das Rechnen mit ihnen für den Computer aufwendiger ist als bei Integers (Details dazu im Kapitel 4).

Warum wurde dann in Programm 3.2 dem Computer diese Mehrbelastung aufgebürdet? Nun, hätte sich das Programm auf Addition, Subtraktion und Multiplikation beschränkt, wäre der Aufwand nicht nötig gewesen. Denn diese Operationen erzeugen bei ganzzahligen Argumenten nur ganzzahlige Ergebnisse. Anders die Division; dividiert man 2 durch 3, so kann man das Ergebnis (1.33333....) nicht mehr als ganze Zahl darstellen (Übrigens: Computer verwenden statt eines Dezimalkommas - wie im Deutschen üblich - stets einen Dezimalpunkt!). Wir brauchen also Gleitpunktarithmetik.

Deshalb haben im Programm 3.2 die Variablen den Datentyp gewechselt. Sie können jetzt auch nicht-ganzzahlige Ergebnisse aufnehmen. Damit ändert sich aber auch etwas für die Ein- und Ausgabefunktionen (*printf* und *scanf*), die sich auf den neuen Datentyp einstellen müssen. Erstmal muß *scanf* angewiesen werden, von der Tastatur Gleitpunktzahlen zu lesen; diese dürfen - im Unterschied zu den Integers - auch einen Dezimalpunkt enthalten. Dann muß *printf* auf die Ausgabe von Ergebnissen mit Nachkommastellen vorbereitet werden. Beides geschieht im Kontrollstring, der die Escape-Sequenz *%f* enthält. Dies ist eine weitere Formatanweisung (*%d* haben Sie schon kennengelernt), wobei "f" die Abkürzung für *float* ist, welches wiederum eine Kurzschreibweise des englischen Wortes für Gleitkommazahl darstellt. Einen Beispiellauf des Programmes sehen Sie in Abbildung 3.2.

**Anmerkung:** Einige C-Compiler (darunter der GST-C-Compiler) unterstützen keine Gleitpunktarithmetik und kennen daher auch nicht den Datentyp *float*. Sollte Sie einen dieser Compiler besitzen, dann können Sie das Programmbeispiel 3.2 nicht testen. Doch trösten Sie sich: die Gleitpunktarithmetik wird im weiteren Verlauf des Buches keine große Rolle mehr spielen.

```
Zahl 1:     5
Zahl 2:     2
Summe:      7.00000
Differenz:  3.00000
Produkt:    10.
Quotient:   2.50000
```

**Abb. 3.2:** *Verarbeitung von Gleitpunktzahlen*

Wie Sie sehen können, werden die Ergebnisse mit fünf Stellen nach dem Komma ausgegeben. Dieses Ausgabeformat kann jedoch von Compiler zu Compiler verschieden sein und läßt sich je nach Bedarf des Programmierers steuern.

## 3.3 Für Schreibfaule

Mehrfach schon wurde es angemerkt: C neigt zur Kürze. Nicht nur in der Wahl der Schlüsselworte drückt man sich in C knapp aus; auch Programme können in unterschiedlichen Varianten geschrieben werden, wobei man einen hohen Grad an Schreib-Ökonomie erzielen kann (sprich: weniger Tastenanschläge auf der Tastatur; also genau das Richtige für Schreibfaule). Dazu ein Beispiel in Programm 3.3.

```
                                        /*************************/
main()                                  /*  Einfache Arithmetik  */
{                                       /*     Kurzversion       */
                                        /*************************/
    float Zahl_1, Zahl_2;               /*  Kurzschreibweise der */
                                        /*  Variablen-Deklaration.*/
                                        /*************************/
    printf("\nZahl 1, Zahl 2: ");       /*  Eingabe-Aufforderung. */
    scanf("%f%f",&Zahl_1,&Zahl_2);      /*  Beide Werte auf einmal*/
                                        /*  lesen.               */
                                        /*************************/
    printf("\nSumme: %f",Zahl_1 + Zahl_2);    /*  Hier macht man sich  */
    printf("\nDifferenz: %f",Zahl_1 - Zahl_2);/*  zunutze, dass die    */
    printf("\nProdukt: %f",Zahl_1 * Zahl_2);  /*  arithmetischen Opera-*/
    printf("\nQuotient: %f",Zahl_1 / Zahl_2); /*  toren einen Wert pro-*/
                                        /*  duzieren.            */
}                                       /*************************/
```

**Prog. 3.3:** *Kurzversion von Programm 3.2*

**Anmerkung:** Mit der selbstgebauten *fscan*-Funktion ist es nicht möglich, mehrere Variablen auf einmal mit Werten zu versorgen. Wenn Sie mit dieser Funktion arbeiten, dann muß Ihr Eingabeteil genauso wie im letzten Programm aussehen.

In jedem der Verarbeitungs-Schritte des Programmes (Variablendeklaration, Eingabe, Verarbeitung mit Ausgabe) hat sich jetzt etwas geändert. Bei der Deklaration sehen Sie, daß das Schlüsselwort *float* nicht für jede Variable von diesem Typ gesondert hingeschrieben werden muß. Es ist ebenso möglich, hinter *float* eine Liste aller Variablen - durch Komma getrennt - folgen zu lassen, die von diesem Typ sein sollen. Natürlich muß diese Deklarations-Anweisung wie jede andere Anweisung auch durch einen Strichpunkt abgeschlossen werden.

Bei der Dateneingabe wurde berücksichtigt, daß *scanf* ebenso wie *printf* eine beliebige Zahl an Argumenten haben kann. Ebenso, wie man mit einem einzigen *printf*-Aufruf mehrere Zahlen auf einmal ausgeben kann, ist es mit *scanf* möglich, mehrere Werte auf einmal einzulesen. Dazu muß man nur im Kontrollstring entsprechend viele Formatanweisungen angeben - und natürlich anschließend an den Kontrollstring genügend Variablenadressen bereithalten, an denen die gelesenen Werte abgelegt werden sollen. Weil im Programm zwei Zahlen eingelesen werden sollen, findet sich im Kontrollstring zweimal die Formatangabe *%f* und auf diesen Kontrollstring folgen zwei Variablenadressen, *&Zahl_1* sowie *&Zahl_2*.

Bei der Eingabe der Daten sollten Sie diese übrigens durch ein (oder mehrere) Leerzeichen trennen, damit *scanf* den Anfang der einen und das Ende der nächsten Zahl auch finden kann. Die Abbildung 3.3 zeigt einen Beispiellauf dieses Programms.

```
Zahl 1, Zahl 2:  5 2
Summe:           7.00000
Differenz:       3.00000
Produkt:        10.00000
Quotient:        2.50000
```

**Abb. 3.3:** *Was das Programm 3.3 produziert*

Wenn Sie das Programm 3.3 genauer betrachten, wird Ihnen auffallen, daß die Variable *Ergebnis* weggefallen ist. Statt die Resultate der einzelnen Berechnungen jedesmal in dieser Variablen zu speichern, sind dafür jetzt die Berechnungen in die Argumentenklammern von *printf* gewandert. Das ist möglich, weil *printf* im Kontrollstring erfährt, daß es eine Gleitpunktzahl ausgeben soll; die auszugebenden Werte erwartet es in der Argumentklammer hinter dem Kontrollstring. Aber es ist *printf* gleichgültig, ob diese Werte in einer Variablen zur Verfügung gestellt werden, oder ob die Be-

rechnungen selbst dort zu finden sind. Sie können sich merken: Alles, was einen Wert produziert, kann *printf* als Argument für die Ausgabe übergeben werden. Neben Variablen produzieren alle Operationen einen Wert; außer den hier vorgestellten arithmetischen Operationen gibt es in C noch eine Menge anderer Operationen, die Sie noch kennenlernen werden.

## 3.4 Die for-Schleife

"Wenn der Computer schon rechnet, dann soll er auch was Ordentliches machen; addieren, subtrahieren und den ganzen Kram - das kann ich selber!"

Bitte: hier ist Programm 3.4 mit einer Menge Neuigkeiten!

```
                                                /*************************/
main()                                          /*  Quadratzahlen-Tabelle  */
{                                               /*************************/
    int von, bis, Zahl;                         /*  Deklarationen.         */
                                                /*************************/
    printf("\nTabelle von (?): ");              /*  Anfangs- und Endwert   */
    scanf("%d",&von);                           /*  fuer Tabelle einlesen. */
    printf("\nbis: ");                          /*                         */
    scanf("%d",&bis);                           /*                         */
                                                /*************************/
    for (Zahl = von; Zahl <= bis; ++Zahl)       /*                         */
        printf("\nDas Quadrat von %d ist %d",   /*  Tabelle berechnen und  */
                Zahl, Zahl * Zahl);             /*  ausgeben.              */
                                                /*                         */
}                                               /*************************/
```

**Prog. 3.4:** *Quadratzahlen-Tabelle; eine einfache* for*-Schleife.*

**Anmerkung:** Für die Eingabe mit *iscan* schreiben Sie:

```
                                                /*************************/
    printf("\nTabelle von (?): ");               /*  Anfangs- und Endwert   */
    iscan(&von);                                /*  fuer Tabelle einlesen. */
    printf("\nbis: ");                          /*                         */
    iscan(&bis);                                /*                         */
```

Wie Sie ja wissen, besteht eine der größten Stärken eines Computers darin, gleiche oder ähnliche Arbeitsschritte wiederholt auszuführen. Im Programm 3.4. wurde dieser manische Wiederholungszwang des Computers zur Erzeugung einer Quadratzahlen-Tabelle ausgenutzt. Das Programm fragt den Benutzer nach dem Zahlenbereich, für den es die Quadratzahlentabelle erstellen soll und legt dann los, wie z.B. in Abbildung 3.4.

Das Einlesen der Anfangs- und Endwerte samt Erzeugen der zugehörigen Meldungen ist für Sie mittlerweile nichts Neues mehr. Interessanter dürfte

es schon sein, wie man - im dritten Programmabschnitt - mit lediglich drei Programmzeilen die Ausgabe von 15 Tabellen-Zeilen bewerkstelligt.

```
Tabelle von (?): 13 ──────┐
                          ├── (Anm.: das geben Sie ein)
bis: 28 ──────────────────┘
Das Quadrat von 13 ist 169
Das Quadrat von 14 ist 196
Das Quadrat von 15 ist 225
Das Quadrat von 16 ist 256
Das Quadrat von 17 ist 289
Das Quadrat von 18 ist 324
Das Quadrat von 19 ist 361
Das Quadrat von 20 ist 400
Das Quadrat von 21 ist 441
Das Quadrat von 22 ist 484
Das Quadrat von 23 ist 529
Das Quadrat von 24 ist 576
Das Quadrat von 25 ist 625
Das Quadrat von 26 ist 676
Das Quadrat von 27 ist 729
Das Quadrat von 28 ist 784
```

**Abb. 3.4:** *Eine Quadratzahlen-Tabelle*

Dazu gibt es die *for*-Schleife. In einer *for*-Schleife werden eine oder mehrere Anweisungen wiederholt ausgeführt, wie z.B. die letzte *printf*-Anweisung im Programm 3.4. Deshalb muß in der *for*-Schleife gekennzeichnet werden, welche Anweisung wiederholt werden soll.

Meist ist man nur an einer bestimmten Anzahl von Wiederholungen interessiert, möchte also nicht, daß die Anweisung endlos wiederholt wird (aber auch das gibt's!). Im Programm 3.4 kann der Benutzer die Anzahl der Wiederholungen durch Eingabe von der Tastatur bestimmen; im Beispiel von Abbildung 3.4 fanden 15 Wiederholungen statt. Deshalb muß die *for*-Schleife Möglichkeiten haben, die Wiederholungen zu kontrollieren.

Die Wiederholung immer wieder desselben Arbeitsschrittes ist meist nicht sinnvoll (aber auch das gibt's). In der Regel wünscht man, daß bei jeder Wiederholung sich ein oder mehrere Werte ändern. Im Programm 3.4. ändert sich bei jedem Durchgang die Zahl, deren Quadrat berechnet wird (und mit ihr natürlich auch das berechnete Quadrat). Deshalb muß die *for*-Schleife Möglichkeiten vorsehen, die sich ändernden Werte anzugeben.

Wenn sich bei jedem Durchgang einer oder mehrere Werte ändern sollen, dann muß man auch angeben, wie sie sich ändern sollen. Im Programm 3.4 ändert sich die zu berechnende Zahl jedesmal um den Wert 1. Das muß jedoch nicht so sein; anstatt nacheinander die Quadrate von 13, 14, 15 usw. zu berechnen, könnte man auch vorziehen, die Werte für 13, 23, 33 usw. zu bestimmen, also mit einer Schrittweite von 10 zu arbeiten. Deshalb muß die

*for*-Schleife Möglichkeiten vorsehen, die Schrittweite(n) der sich ändernden Werte zu bestimmen.

All dies erreicht die *for*-Schleife mit den Sprachmitteln aus der Abbildung 3.5. Das ist wieder so ein Syntax-Diagramm, wie es Ihnen in der Abbildung 2.8 schon einmal vorgekommen ist. Mit diesen Diagrammen beschreibt man den Aufbau komplizierterer Sprachmittel, wie die *for*-Schleife eines ist; zum Glück gibt es davon in C nicht allzu viele. Alles, was in diesen Diagrammen in spitzen Klammern steht (wie z.B. "<initialisierung>"), bedarf noch weiterer Erläuterung. Alles andere (wie z.B. das Schlüsselwort *for*, die runden Klammern und Strichpunkte) müssen Sie wie im Diagramm aufgeführt übernehmen, wenn Sie eine *for*-Anweisung in eigene Programme einbauen wollen.

```
for ( <initialisierung> ; <kontrollbedingung> ; <re-initialisierung> )
                <anweisung>
```

**Abb. 3.5:** *Anatomie einer* for-*Schleife*

Dem Syntax-Diagramm der Abbildung 3.5 entnehmen Sie also folgendes: Nach dem Schlüsselwort *for* muß eine runde Klammer kommen, in der durch Strichpunkt getrennt Informationen über die Initialisierung, die Kontrolle und die Re-Initialisierung der Schleife zu finden ist (was das ist? kommt gleich, kommt gleich!). An diese Steuer-Informationen in der runden Klammer schließt sich die Anweisung an, die in der Schleife wiederholt wird. Im Programm 3.4 entsprechen den einzelnen Bestandteilen folgende Komponenten:

```
<initialisierung>    : Zahl = von
<kontrollbedingung>  : Zahl <= bis
<re-initialisierung> : ++Zahl
<anweisung>          : printf("\nDas Quadrat von %d ist %d",Zahl, Zahl * Zahl);
```

Im Initialisierungsteil erhalten innerhalb der Schleife verwendete Variablen ihren Anfangswert. Das vorliegende Beispiel arbeitet in der Schleife nur mit einer Variablen, nämlich mit *Zahl*; diese erhält als Anfangswert das, was in der Variablen *von* gespeichert ist und das wiederum wurde mit *scanf* vom Benutzer abgefragt. So kommt es, daß die Quadratzahlentabelle genau mit der Zahl beginnt, die Sie eingeben.

In der Kontrollbedingung wird festgelegt, wie lange die Schleife durchlaufen werden soll. Solange die Kontrollbedingung erfüllt ist (d.h. der dort stehende Ausdruck den Wert "wahr" hat), wird die von der *for*-Schleife kontrollierte Anweisung ausgeführt. Die Kontrollbedingung des Programmbeispiels ist im Klartext so zu lesen: "solange der in *Zahl* gespeicherte Wert

kleiner oder gleich dem in *bis* gespeichertem Wert ist". In C schreibt man das so:

```
Zahl <= bis
```

und das ist auch gleich ein Beispiel für einen "Vergleichsoperator". So wie *+* zwei Werte nimmt und deren Summe liefert, so nimmt *<=* zwei Werte, vergleicht sie miteinander und liefert den Wahrheitswert "wahr", wenn der erste Wert kleiner oder gleich dem zweiten ist und ansonsten (wenn also der erste Wert größer als der zweite ist) den Wahrheitswert "falsch".

Im Beispielprogramm hat die Variable *bis* ihren Wert durch Tastatureingabe erhalten; beim ersten Schleifendurchgang hat *Zahl* den Wert, den *von* über Tastatureingabe erhielt. Ersetzt man die Variablen durch ihre Momentanen Werte, dann ergibt sich für den Beispiellauf der Abbildung 3.4 vor dem ersten Durchgang:

```
13 <= 28
```

und weil dies tatsächlich wahr ist, wird die Anweisung durchgeführt. Wie jedoch sorgt man dafür, daß die Kontrollbedingung falsch wird und die Schleife somit abbricht?

Durch den Re-Initialisierungsteil (was für ein Wort!) der *for*-Schleife! Ist die Anweisung nämlich einmal ausgeführt, denn tut C das, was in diesem Teil steht, ehe es erneut die Kontrollbedingung überprüft und gegebenenfalls die Anweisung der Schleife wiederholt. Und was steht da im Beispiel?

```
++Zahl
```

Womit Sie wieder einen Operator kennengelernt haben. Das unscheinbare *++* heißt mit vollem Namen "Auto-Prä-Inkrement", und wenn Sie dieses Wort so ganz beiläufig auf der nächsten Party fallen lassen, dann sind Sie ein für alle mal als der Ober-Computerspezialist legitimiert (wenn man's nicht versehentlich für ein Schimpfwort hält)!

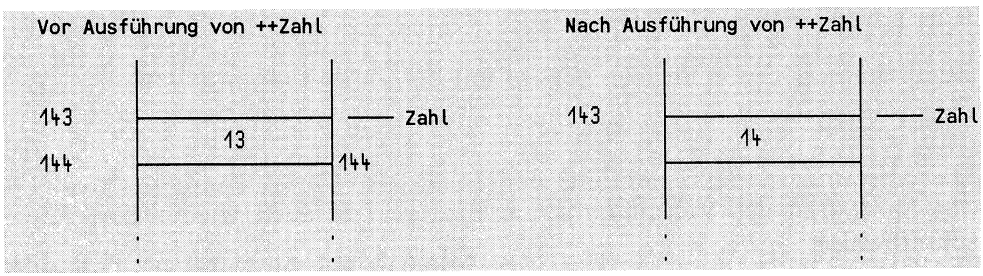



**Abb. 3.6:** *Speicherdiagramm für ++*

Trotz der furchteinflößenden Bezeichnung ist schnell gesagt, was *++* tut: es zählt den Inhalt einer Variablen um Eins hoch. Ist in *Zahl* der Wert 13 gespeichert, dann findet sich nach Ausführung von *++Zahl* dort der Wert 14. Das zeigt auch die Abbildung 3.6. Wer's salopper liebt, kann sagen, daß *++* eine Variable hochzählt. Der auf Etikette bedachte Freak wird dazu (zum Hochzählen) "Inkrementieren" sagen, was einen Teil des Namens erklärt.

## 3.5 Inkrement und Dekrement

Wer BASIC kennt, den wird die knappe Art verblüffen, mit der in C das Hochzählen (Inkrementieren) von Variablen möglich ist. Vermutlich hätte er erwartet, daß dies mit einer Anweisung wie z.B.:

```
Zahl = Zahl + 1;
```

bewirkt wird. Wird es auch, liebe BASIC-Bewanderte; die andere Methode ist jedoch nicht nur eleganter, sonder auch schneller in der Ausführung!

Dennoch: der herkömmlichen Methode des Inkrementierens mittels Wertzuweisung muß noch etwas Aufmerksamkeit geschenkt werden. Wer zum ersten mal einen Ausdruck wie den obigen sieht und dabei an die Schulmathematik denkt, der wird ihn für baren Unsinn halten. Denn wir sind gewohnt, das = als Gleichheitszeichen zu lesen; niemals aber ist eine Zahl so groß wie ihr Nachfolger!

Sie aber wissen bereits, daß = das Zeichen für Zuweisung darstellt; in C ist es ein Operator, was bestimmte Folgen hat, zu denen ich noch kommen werde. Bei der Zuweisung wird erstmal der (in diesem Fall arithmetische) Ausdruck rechts vom = berechnet und dann dessen Ergebnis dem *lvalue* links vom = zugewiesen. Will man auf eine Variable nicht den Wert 1, sondern beispielsweise 13 draufzählen, dann schreibt man einfach:

```
Zahl = Zahl + 13;
```

Zum 'runterzählen kann man ebenso mit der Zuweisung arbeiten:

```
Zahl = Zahl - 2;
```

Dies vermindert (dekrementiert) den Inhalt der Variablen *Zahl* um 2. Für den Spezialfall des Verminderns um Eins gibt es auch wieder eine Kurzschreibweise:

```
--Zahl;
```

Das hat dieselbe Wirkung wie:

```
Zahl = Zahl - 1;
```

Das -- ist ein einstelliger Operator, eng verwandt mit *++* und trägt den schönen Namen "Auto-Prä-Dekrement".

Für der Erhöhen bzw. Vermindern einer Variablen gibt es also ++ und --; aber C wäre nicht C, wenn es für die anderen Fälle nicht auch eine kürzere Schreibweise gäbe:

| **statt:** | **kürzer:** |
|---|---|
| `Zahl = Zahl + 5;` | `Zahl += 5;` |
| `Zahl = Zahl - 12;` | `Zahl -= 12;` |
| `Zahl = Zahl * 2;` | `Zahl *= 2;` |
| `Zahl = Zahl / 4;` | `Zahl /= 4;` |

Sie sehen: für Zuweisungen, in denen der Wert einer (und nur einer) Variablen manipuliert wird (gleichgültig, mit welcher arithmetischen Operation) gibt es in C einen spezialisierten Zuweisungs-Operator, der nicht nur schneller hingeschrieben ist, sondern auch für schnellere Programme sorgt!

Mit diesen Zutaten ausgestattet, können Sie bereits eine Vielzahl von Programmieraufgaben mit der *for*-Schleife erledigen. In der Abbildung 3.7 sind verschiedene Varianten der *for*-Schleife zusamengefaßt, die jeweils auf unterschiedliche Art eine Integer-Variable *i* im Schleifenkörper manipulieren. Die Beispiele führen auch einen neuen Vergleichsoperator ein: *>=* steht für "größer-gleich". Die Kommentare erläutern die Wirkung der einzelnen Varianten:

```
                                    /*************************/
for (i = 1; i <= 10; ++i)           /* Druckt die Zahlen von */
    printf("\n%d",i);               /*  1 bis 10 in aufstei- */
                                    /*  gender Reihefolge.   */
                                    /*************************/
for (i = 10; i >= 0; --i)           /*  Druckt die Zahlen von */
    printf("\n%d",i);               /*  10 bis 0 in absteigen- */
                                    /*  der Reihenfolge.     */
                                    /*************************/
for (i = 1; i <= 20; i += 2)        /* Druckt 1,3,9...,19    */
    printf("\n%d",i);               /*                       */
                                    /*************************/
for (i = 20; i >= 0; i -= 2)        /* Druckt 20,18,16,...,0 */
    printf("\n%d",i);               /*                       */
                                    /*************************/
```

**Abb. 3.7:** *Varianten der* for-*Schleife*

## 3.6 Der Profi sorgt für Sicherheit

Obwohl das Programm 3.4 nicht sonderlich aufregend ist, lassen sich daran doch bereits einige wichtige Prinzipien der Software-Entwicklung demonstrieren. Als Programmierer sollte man nämlich stets bedacht sein, in seinem Programm vorhersehbare Benutzerfehler so weit als möglich abzufangen und weitestgehend zu korrigieren. Dadurch wird das Programm robuster und fehlertoleranter. Halten Sie sich an dieses Prinzip; Ihr Benutzer wird's Ihnen danken.

Ein möglicher Problempunkt des Programms 3.4 ist die Festlegung der Anfangs- und Endwerte der Tabelle durch den Benutzer. Der kann sich ja vertun und aus Versehen Anfangs- und Endwert vertauschen. Probieren Sie doch mal mit Ihrem Programm aus, was in diesem Fall passiert? Nichts! Die Kontrollbedingung der *for*-Schleife ist nämlich nun schon beim Eintritt in die Schleife falsch. In diesem Fall wird der Schleifenkörper gar nicht erst durchlaufen: die Bedingung wird vor dem erstmaligen Eintritt in den Schleifenkörper getestet. Diese Schleifenvariante bezeichnet der Informatiker übrigens als "abweisende" Schleife (denn es kann sein, daß sie gar nicht durchlaufen wird).

Programm 3.4 wäre ein Stück benutzerfreundlicher, wenn es in diesem Fehler-Fall gutwillig reagieren und den kleineren Wert stillschweigend als Anfangs-, den größeren als Endwert heranziehen würde. Die nächste Variante, Programm 3.5, tut genau dies.

```
                                        /*************************/
main()                                  /*  Quadratzahlen-Tabelle  */
{                                       /*************************/
    int von, bis, Zahl;                 /*  Deklarationen.         */
    int zwi;                            /*  Zwischenspeicher.      */
                                        /*************************/
    printf("\nTabelle von (?): ");      /*  Anfangs- und Endwert   */
    scanf("%d",&von);                   /*  fuer Tabelle einlesen. */
    printf("\nbis: ");                  /*                         */
    scanf("%d",&bis);                   /*                         */
                                        /*************************/
    if (bis > von)                      /* Falls Endwert größer   */
    {   zwi = von;                      /*  als Anfangswert die    */
        von = bis;                      /*  beiden Werte vertau-   */
        bis = zwi;                      /*  schen.                 */
    }                                   /*                         */
                                        /*************************/
    for (Zahl = von; Zahl <= bis; ++Zahl)   /*                     */
        printf("\nDas Quadrat von %d ist %d",  /*  Tabelle berechnen und */
                Zahl, Zahl * Zahl);     /*  ausgeben.              */
                                        /*                         */
}                                       /*************************/
```

**Prog. 3.5:** *Qudratzahlen-Tabelle mit Fehlersicherung*

### 3.6.1 Die if-Anweisung

Die Überprüfung darauf, ob die eingegebenen Werte plausibel sind, erfolgt in einer *if*-Anweisung. Die Anatomie der *if*-Anweisung finden Sie in Abbildung 3.8

```
if (<bedingung>)
    <anweisung>
```

**Abb. 3.8:** *Anatomie der* if-*Anweisung*

Mit *if* kontrolliert man in C-Programmen die Ausführung einer anderen Anweisung. Dies ist ähnlich wie bei *for*, das die - allerdings wiederholte - Ausführung einer Anweisung kontrolliert. Die Kontrolle erfolgt bei *if*, indem das Vorliegen einer bestimmten Bedingung überprüft wird. Diese Bedingung steht in runden Klammern hinter dem Schlüsselwort *if*; es muß sich dabei um einen wahrheitswertigen Ausdruck handeln. Meist steht hier irgendeine Art von Wertevergleich, wie Sie es ja auch schon aus der Kontrollbedingung der *for*-Schleife kennen.

Ist die in runden Klammern stehende Bedingung wahr, dann wird die Anweisung ausgeführt, die hinter der runden Klammer folgt. Ansonsten (bei nicht erfüllter Bedingung) wird diese Anweisung einfach übersprungen und C fährt im Programm an der Stelle hinter dem gesamten *if*-Komplex fort. Die Wirkung der *if*-Anweisung könnte man umgangssprachlich so fassen: "Wenn das-und-das der Fall ist (die Bedingung erfüllt) ist, dann tue dieses-oder-jenes; ansonsten lasse es bleiben".

Übertragen auf das Programmbeispiel 3.5 hieße das: "Wenn der eingegebene Anfangswert größer als der eingegebene Endwert ist, dann vertausche diese beiden Werte, ehe du die *for*-Schleife ausführst". Das Zeichen > steht für "größer" und ist ein weiterer Vergleichsoperator.

Das Vertauschen zweier Variablenwerte erfordert in C drei Schritte. Denn würde man einfach schreiben

```
von = bis;
```

dann wäre der alte Wert der Variablen *von* verloren und könnte nicht mehr an *bis* zugewiesen werden. Deshalb wird ein Zwischenspeicher benötigt, in diesem Fall die Integer-Variable *zwi*. Hier noch einmal die gesamte Vertauschungs-Sequenz mit Erläuterungen:

```
zwi = von; ---- Speichere alten Anfangswert
von = bis; ---- Jetzt sind Anfangs- und Endwert gleich
bis = zwi; ---- Der zwischengespeicherte Anfangswert wird nun
                zum neuen Endwert.
```

Jetzt aber zu einem offensichtlichen Widerspruch: der Anatomie des *if* können Sie entnehmen, daß dieses die Ausführung genau einer Anweisung steuert; zum Vertauschen werden aber drei Anweisungen benötigt! Damit nun wieder alles seine Ordnung hat, muß man diese drei Anweisungen zu einer machen, indem man sie zu einem Anweisungsblock zusammenfaßt. Dies geschieht mit genau den selben geschweiften Klammern, mit denen auch der Anweisungsblock einer Funktion (und von *main*) umschlossen wird.

Soll also eine Folge von Anweisungen syntaktisch wie eine einzige Anweisung behandelt werden, dann müssen Sie diese Anweisungsfolge mit geschweiften Klammern zu einem Block zusammenfassen. Durch diese Blockbildung entsteht eine sogenannte "zusammengesetzte Anweisung". Anders als die Einzelanweisungen müssen zusammengesetzte Anweisungen nicht mit einem Strichpunkt abgeschlossen werden; auch dies können Sie im Programm 3.5 erkennen.

```
#include <stdio.h>
#define TABELLE "\nDas Quadrat von %d ist %d"
#define MELDUNG "\nWeiter mit beliebiger Taste: "

main()                                          /***************************/
{                                               /*  Quadratzahlentabelle.  */
    int von, bis, Zahl;                         /***************************/
    int zlr;                                    /*                         */
    char Eingabe;                               /* Zeichenvariable         */
                                                /***************************/
    printf("\nTabelle von (?): ");               /* Werteingabe wie gehabt  */
    scanf("%d",&von);                           /*                         */
    printf("\nbis: ");                          /*                         */
    scanf("%d",&bis);                           /***************************/
                                                /*                         */
    if (bis > von)                              /* Falls Endwert grösser   */
    {   zwi = von;                              /*  als Anfangswert die    */
        von = bis;                              /*  beiden Werte vertau-   */
        bis = zwi;                              /*  schen.                 */
    }                                           /*                         */
                                                /***************************/
                                                /*                         */
    for (Zahl = von, zlr = 1; Zahl <= bis; ++Zahl, ++zlr)  /*              */
    {   printf(TABELLE, Zahl, Zahl * Zahl);     /*                         */
        if (zlr == 20)                          /*  20 Zeilen ausgegeben;  */
        {   printf(MELDUNG);                    /*  Meldung ausgeben und   */
            Eingabe = getchar();                /*  auf Tastendruck warten */
            zlr = 1;                            /*  Zaehler ruecksetzen.   */
        }                                       /*<--- Ende "if"-Anweisung */
    }                                           /*<--- Ende "for"-Schleife */
                                                /***************************/
}
```

**Prog. 3.6:** *Zeichenvariable und Zeicheneingabe*

### 3.6.2 Der Datentyp char: Zeichenvariablen

Nicht nur fehlertolerant sollte man seine Programme machen; auch für Bequemlichkeit muß der Programmierer sorgen. In diesem Punkt läßt das Quadratzahlen-Programm noch einiges zu Wünschen übrig. Denn wenn sich der Benutzer eine Tabelle anzeigen läßt, die mehr als 24 Zeilen umfaßt, dann gehen ihm wegen der Beschränkung des Bildschirms auf 25 Zeilen die ersten Einträge verloren. Auch erfolgt die Ausgabe so schnell, daß man keine Chance zum Mitlesen hat.

Es wäre darum wünschenswert, die Ausgabe der Tabelle nach jeweils 20 Zeilen anzuhalten, damit die Ergebnisse in aller Ruhe betrachtet werden können. Auf Knopf- bzw. Tastendruck sollte das Programm dann mit seiner Arbeit fortfahren. Dazu wurde das Programm 3.5. nochmals modifiziert (siehe Programm 3.6).

**Anmerkung:** Wenn Sie mit dem ATARI-C arbeiten, dann sollten Sie - gemäß den Ausführungen im ersten Kapitel) Ihr Programm mit folgender Anweisung beginnen:

```
#include <header.h>
```

Außerdem ist in sehr frühen Compiler-Versionen die Funktion *getchar( )* nicht in Ordnung. Sollte Ihr Programm nicht funktionieren, dann müssen Sie folgende Vorkehrungen treffen. An den Programmanfang (unter die Deklarationen) schreiben Sie

```
extern long gemdos();
```

Anstelle von *getchar( )* schreiben Sie

```
gemdos(0x1);
```

Wenn Sie die Funktion *get_s* aus der Abbildung 3.1 aufmerksam durchgelesen haben, dann werden Ihnen diese Vorkehrungen bekannt vorkommen!

Dieses Beispiel wartet gleich mit mehreren Neuigkeiten auf. Gleich in der ersten Zeile finden Sie einen neue Anweisung für den Makro-Präprozessor, kenntlich an dem # am Zeilenanfang. Die *include*-Anweisung dient dazu, den Inhalt einer anderen Datei mit Ihrer Programmdatei zu verknüpfen. Sieht der Präprozessor eine *include*-Anweisung, dann hört er mit der 'Lektüre' Ihres Programms auf, sucht erst mal die hinter *include* angegebene Datei und liest sich diese durch. Der Dateiname muß dabei (wie im Beispielprogramm) mit spitzen Klammern dekoriert werden. Auch doppelte Anführungszeichen sind möglich, und einige Compiler reagieren unterschiedlich auf diese verschiedenen Begrenzungszeichen. Im wesentlichen

geht es hier um die Frage, in welchem Teilverzeichnis der Diskette die *include*-Datei gesucht wird. Um in diesem Punkt letzte Gewißheit zu erhalten, sollten Sie in der Dokumentation Ihres Systems nachsehen.

Durch die *include*-Datei wird der Compiler über alle darin enthaltenen Definitionen informiert. Danach erst setzt er die Abarbeitung Ihres Programmes fort. Dieses kann jetzt somit auf alle eben eingeführten Definitionen zurückgreifen. Eine *include*-Anweisung hat demnach den Effekt, den Inhalt einer anderen Datei in Ihre eigenen Datei mit aufzunehmen (engl. "to include" = aufnehmen).

**Anmerkung:** Wenn sie in Ermangelung eines funktioneierenden *scanf* mit *iscan* und *fscan* arbeiten, dann können Sie künftig einfach die Anweisung

```
#include "scanf.c"
```

an den Anfang (oder das Ende) Ihres Programms schreiben und so das Einlesen der nötigen Definitionen dem Präprozessor überlassen. Sie sparen sich damit das händische Einlesen im Editor.

In C ist es der Brauch, oft benötigte Präprozessor-Definitionen in Dateien zusammenzufassen, denen - aus traditionellen Gründen - meist der Namenszusatz *.h* angehängt wird. Sie haben den vollen Leistungsumfang des Präprozessors noch nicht kennengelernt, weswegen ich jetzt, was diese Definitionen betrifft, noch nicht ins Detail gehen kann. In jedem C-System sind jedoch in der Datei *stdio.h* einige wesentliche Definitionen enthalten, die mit der Ein-/Ausgabe zusammenhängen (*stdio* ist denn auch die Abkürzung für "Standard Input/Output"). Die Funktion *getchar*, die weiter unten im Programm benötigt wird, gehört zu den in dieser Datei definierten Dingen.

Sie sollten sich zur Gewohnheit machen, diese *include*-Anweisung in alle Ihre Programme aufzunehmen, die mit Ein-/Ausgabe zu tun haben. Das war zwar bei den bisherigen Programmbeispielen auch der Fall, ich habe jedoch darauf geachtet, nur solche Sprachmittel zu verwenden, die die Definitionen in *stdio.h* nicht benötigen. Ab jetzt jedoch wird diese Einleitungszeile in keinem Programm mehr fehlen.

Es schließen sich zwei Definitionen an, mit denen in bereits bekannter Weise im Programm verwendeten Meldungen ein Name gegeben wird. Dies geschieht aus Gründen der Übersichtlichkeit. Es ist aber auch ansonsten keine üble Praxis, Meldungstexte auf diese Art zu definieren; so haben Sie sie alle übersichtlich am Programmanfang versammelt und können sie gegebenenfalls leicht ändern.

Neuigkeiten gibt es auch bei den Variablendeklarationen. Das Programm soll ja die Bildschirmausgabe nach jeweils 20 Zeilen anhalten und erst auf Tastendruck des Benutzers wieder fortsetzen. In einer späteren Version des Programmes wird die vom Benutzer gedrückte Taste (d.h. das von ihm eingegebene Zeichen) noch weiter ausgewertet. Deshalb speichert das Programm diese Benutzereingabe auch ab. Dazu wird jedoch ein eigener Variablentyp benötigt, nämlich eine Zeichenvariable, die im Beispielprogramm den Namen "Eingabe" trägt. Variablen dieses Typs deklariert man mit dem Schlüsselwort *char*, einer Abkürzung des Englischen "character", was im Deutschen "Zeichen" bedeutet. Über Zeichen und Zeichenvariable werden Sie später noch detailliertere Informationen erhalten.

Um die Ausgabe zum rechten Zeitpunkt anhalten zu können, wird innerhalb der *for*-Schleife eine Variable (mit dem Namen *zlr*, was ein typischer C-Name für einen Zähler ist) hochgezählt. Bei jedem Schleifendurchgang erfolgt (mit *if*) eine Überprüfung, ob diese Zählvariable bereits den Wert 20 erreicht hat. Interessant ist, wie diese Abfrage vonstatten geht:

```
if (zlr == 20)
.
.
.
```

Hier haben Sie einen weiteren Vergleichsoperator von C: die zwei Gleichheitszeichen (==) dienen nämlich zum Testen auf Gleichheit! Hier liegt eine der beliebtesten Fallgruben für C-Anfänger, denn viele andere Sprachen, darunter auch BASIC, verwenden für den Gleichheitstest das einfache Gleichheitszeichen. Das aber ist in C ausschließlich für die Zuweisung reserviert. Aus Gründen, die jetzt noch nicht dargelegt werden können, ist es dem C-Compiler nicht möglich, diesen Fehler durch syntaktische Überprüfung abzufangen. Sollten Sie versehentlich geschrieben haben:

```
if (zlr = 20)
.
.
.
```

dann erhalten Sie keinen Hinweis bei der Programmübersetzung! Sie können das Programm auch laufen lassen, werden aber feststellen, daß Sie jetzt bei jedem Schleifendurchgang um einen Tastendruck angegangen werden. Daran (spätestens) können Sie Ihren Fehler in diesem Fall erkennen.

Deshalb der gute Rat, bei Gleichheitstest besonders in der Anfangsphase der C-Programmierung geradezu höllisch aufzupassen; es lohnt sich! Sollte (wie im vorliegenden Beispiel) mit einer Konstante verglichen werden, dann können Sie sich behelfen, indem Sie den Vergleich in der Form

```
if (20 == zlr)
.
.
.
```

notieren, die Konstante also zuerst hinschreiben. Vergessen Sie jetzt ein Gleichheitszeichen, dann wird der Fehler bei der Syntaxüberprüfung erkannt, denn *20 = zlr* ist eine Zuweisung ohne zulässigen *lvalue* (Sie erinnern sich: links vom Zuweisungszeichen dürfen nur Variablen stehen).

Ehe wir zu der vom *if* kontrollierten Anweisung kommen, ist noch die runde Klammer der *for*-Schleife beachtenswert. Hier werden nämlich im Initialisierungsteil zwei Variablen initialisiert und auch bei jedem Durchgang im Reinitialisierungsteil zwei Variablen inkrementiert. In jeder der drei Klammer-Komponenten einer *for*-Schleife (also auch bei der Kontrollbedingung) können mehrere Aktionen durchgeführt werden. Diese müssen aber dann durch ein Komma voneinander getrennt sein. C garantiert Ihnen, daß solcherart getrennte Anweisungen von links nach rechts berechnet werden.

Jetzt aber zurück zur *if*-Anweisung. Bei jeder zwanzigsten Ausgabezeile wird der vom *if* kontrollierte Anweisungsblock ausgeführt. Nach der Ausgabe einer Meldung wartet das Programm hier auf einen Tastendruck des Benutzers. Es tut dies, indem es die Funktion *getchar( )* aufruft. Diese wartet, bis an der Tastatur eine Taste betätigt wurde und liefert dann als Wert das entsprechende Zeichen zurück. Natürlich wäre es auch möglich gewesen, diese Tastatureingabe über *scanf* (zusammen mit einer entsprechenden Formatanweisung für Zeicheneingabe) zu erledigen. Doch die Lösung mit *getchar* ist einfacher.

**Anmerkung:** Wenn *getchar( )* nicht arbeitet, wie es sollte, dann benutzen Sie wie weiter oben beschrieben stattdessen *gemdos(0x1a)*!

Nach Empfang des Zeichens, das sich das Programm in der Variablen "Eingabe" aufhebt, wird der Zeilenzähler auf den Wert 1 zurückgesetzt und die Schleife fortgesetzt. Innerhalb des Schleifenkörpers müssen jetzt mehrere Aktionen wiederholt werden: neben der Ausgabe der Tabellenzeilen mit *printf* eben auch die *if*-Anweisung zum Anhalten der Ausgabe. Dazu ist es wieder nötig, diese Anweisungen mit geschweiften Klammern zu einem Anweisungsblock zusammenzufassen.

Im Programm 3.6. sind jetzt somit drei Anweisungsblöcke ineinandergeschachtelt. Der äußerste macht den Anweisungsteil der Hauptfunktion *main* aus; darin eingeschachtelt findet sich der Block, den *for* kontrolliert, innerhalb dessen wiederum ein von *if* kontrollierter Block steckt. Diese Verschachtelung wird optisch durch die unterschiedlich tiefe Einrückung der

Blocks akzentuiert; dies ist aber wieder eine Stilfrage und nichts, was Ihnen C vorschreiben würde.

Das Programm 3.7 zeigt eine weitere (und letzte) Steigerung im Komfort für den Benutzer. In dieser Version hat er nämlich die Möglichkeit, die Programmausgabe durch Betätigen einer speziellen Taste abzubrechen. Das kann z.B. immer dann nützlich sein, wenn das Programm mehr Quadratzahlen erzeugt, als er überhaupt anzusehen Lust hat.

```
#include <stdio.h>
#define TABELLE "\nDas Quadrat von %d ist %d"
#define MELDUNG "\nWeiter mit beliebiger Taste: "

main()                                          /*************************/
{                                               /*  Quadratzahlentabelle.  */
    int von, bis, Zahl;                         /*************************/
    int zlr;                                    /*                        */
    char Eingabe;                               /* Zeichenvariable        */
                                                /*************************/
    printf("\nTabelle von (?): ");              /* Werteingabe wie gehabt */
    scanf("%d",&von);                           /*                        */
    printf("\nbis: ");                          /*                        */
    scanf("%d",&bis);                           /*************************/
                                                /*                        */
    if (bis > von)                              /* Falls Endwert grösser  */
    {   zwi = von;                              /*  als Anfangswert die   */
        von = bis;                              /*  beiden Werte vertau-  */
        bis = zwi;                              /*  schen.                */
    }                                           /*                        */
                                                /*************************/
                                                /*                        */
    for (Zahl = von, zlr = 1; Zahl <= bis; ++Zahl, ++zlr)  /*             */
    {   printf(TABELLE, Zahl, Zahl * Zahl);     /*                        */
        if (zlr == 20)                          /*  20 Zeilen ausgegeben; */
        {   printf(MELDUNG);                    /*  Meldung ausgeben und  */
            Eingabe = getchar();                /*  auf Tastendruck warten */
            if (Eingabe == 'A' || Eingabe == 'a') /* Programmabruch noetig? */
                break;                          /* Ja; Schleife verlassen. */
            zlr = 1;                            /*  Zaehler ruecksetzen.  */
        }                                       /*<--- Ende "if"-Anweisung */
    }                                           /*<--- Ende "for"-Schleife */
                                                /*************************/
}
```

**Prog. 3.7:** *Reaktion auf Benutzereingabe*

Wie sein Vorgänger wartet auch dieses Programm nach jeder zwanzigsten Zeile auf eine Reaktion des Benutzers; anders als sein Vorgänger wertet es sie jedoch aus. Denn wenn dieser die "A"-Taste betätigt, dann soll das Programm abgebrochen werden. Jetzt hat es auch einen Sinn, sich das eingegebene Zeichen in einer Variablen aufzuheben, da es für den nachfolgenden Vergleich noch verfügbar sein muß.

Dieser Vergleich wird von *if* kontrolliert. Falls der Benutzer ein "A" eingegeben hat, soll die Schleife abgebrochen werden. Nun kann es aber sein, daß ein "A" eingegeben wurde, aber daß es sich um den entsprechenden Kleinbuchstaben handelt. Da das Programm da nicht so streng sein will, wird auch in diesem Fall abgebrochen. Im *if* muß also überprüft werden, ob ein großes oder kleines "A" von *getchar* geliefert wurde. Dies schreibt man in C so:

```
if (Eingabe == 'A' || Eingabe == 'a')
.
.
.
```

Daran ist zuerst mal der Test auf Gleichheit mit einer Zeichenkonstanten interessant. Wie Sie sehen können, werden Zeichenkonstanten in einfache Hochkommas eingeschlossen. Dadurch ist es dem C-Compiler möglich, sie von einbuchstabigen Variablen zu unterscheiden. Ferner ist die Art neu, in der hier zwei Vergleiche miteinander verknüpft werden. Die beiden senkrechten Striche stehen nämlich für das umgangssprachliche "Oder"; die vom *if* kontrollierte Anweisung wird nur ausgeführt, wenn entweder ein kleines "a" oder ein großes "A" in der Variablen "Eingabe" gespeichert ist.

Damit haben Sie eine neue Art von wahrheitswertigem Operator kennengelernt. Er dient zur Verknüpfung mehrere Wahrheitswerte, im vorliegenden Fall für die sog. "Oder"-Verknüpfung. Mehr darüber werden Sie im nächsten Kapitel erfahren.

Wenn nun aber der Benutzer einen dieser beiden Buchstaben eingegeben hat, dann soll die Schleife abgebrochen werden. Wie macht man das in C? Ganz einfach mit *break* (englisch für "unterbrechen" oder "abbrechen")! Diese Anweisung führt, wenn sie ausgeführt wird, zum sofortigen Verlassen der Schleife, in der sie steht. Das Programm wird hinter der Schleife fortgesetzt; da im Beispiel 3.7 hinter der *for*-Schleife nichts mehr kommt, entspricht dies einer sofortigen Beendigung des Programms.

## 3.7 Benutzerdefinierte Funktionen mit Parametern

Die große Attraktivität von C für professionelle Programmierung rührt hauptsächlich von der Möglichkeit zum modularen Programmieren her. Dies wiederum basiert hauptsächlich auf der Verschachtelung von Funktionen. Neben "vorgefertigten" Systemfunktionen wie *printf*, *scanf* und *getchar* können Sie sich auch eigene Funktionen definieren. Das zweite Kapitel deutete dies bereits an (Programm 2.2); zu diesem Zeitpunkt war aber das

Definieren von einigermaßen nützlichen eigenen Funktionen noch nicht möglich.

Jetzt aber sind Sie so weit; Sie haben genügend Kenntnisse von C, um sich an diese Materie heranzuwagen. Und ganz so kompliziert ist es ja auch wieder nicht!

### 3.7.1 Die while-Schleife

Es wird öfter vorkommen, daß Sie - wie im Programm 3.7 - auf einen Tastendruck des Benutzers reagieren müssen. Viele Programme konfrontieren den Anwender mit Ja/Nein-Auswahlmöglichkeiten und verfahren entsprechend, je nachdem, ob ein 'J' oder eine andere Taste gedrückt wurde. Nun kann man vom Benutzer nicht verlangen, immer nur in Großbuchstaben mit seinem Programm zu verkehren. Eine Lösung dieses Problems zeigt Programm 3.7: man überprüft eben immer beide Fälle.

Übersichtlicher aber wäre es, die Benutzereingabe erstmal (falls es nötig ist) in einen Großbuchstaben umzuwandeln und dann diesen für weitere Entscheidungen im Programm heranzuziehen. Sowas kann öfter gebraucht werden, und deshalb macht man es in einer Funktion.

Die Aufgabenstellung lautet: eine Funktion zu schreiben, die ein Zeichen nimmt und es entweder unverändert zurückgibt, falls es sich um einen Großbuchstaben handelt, oder dieses Zeichen in den entsprechenden Großbuchstaben verwandelt und diesen zurückgibt. Programm 3.7 tut genau dies.

Dieses Programm bietet eine ganze Menge an Neuigkeiten. Wenn Sie es laufen lassen, dann werden Sie feststellen, daß es alle Zeichen, die Sie von der Tastatur eingeben, in Großschreibung zurückgibt (sofern das möglich ist; Zahlen und Sonderzeichen passieren das Programm natürlich unverändert), bis Sie ein großes "E" eingeben. Dies ist für das Programm das Endesignal.

Als erstes muß die Hauptfunktion *main* analysiert werden, da sie das allgemeine Verhalten des Programms bestimmt. Es sollen fortgesetzt Zeichen eingelesen und - in veränderter Form - wieder ausgegeben werden. Die fortgesetzte Wiederholung von Verarbeitungsschritten erledigt man in C bekanntlich mit einer Schleife. In diesem Fall ist es jedoch nicht nötig, innerhalb der Schleife irgendwelche Variablen hoch- oder niederzuzählen. Lediglich eine Kontrollbedingung muß angegeben werden, die darüber wacht, daß solange mit der Schleife fortgefahren wird, bis der Benutzer ein großes "E" tippt.

```
#include <stdio.h>

main()                                  /*************************/
{                                       /*  Eingelesene Zeichen in */
    char c, toupper();                  /*  Grossbuchstaben umwan- */
                                        /*  deln, bis Benutzer 'E' */
    while ((c = getchar()) != 'E')      /*  eingibt.               */
        putchar(toupper(c));            /*                         */
}                                       /*************************/


char toupper(zei)                       /*************************/
     char zei;                          /*  Parameter in Gross-    */
                                        /*  buchstaben wandeln     */
{                                       /*************************/
    if (islower(zei))                   /* Falls Kleinbuchstabe:   */
        return (zei + 'A' - 'a');       /*    umwandeln            */
    else                                /* ansonsten:              */
        return(zei);                    /*    unveraendert zurueck */
}                                       /*************************/

int islower(c)                          /*************************/
     char c;                            /*  Ist der Parameter      */
                                        /*  kleingeschrieben?      */
{                                       /*************************/
    return (c >= 'a' && c <= 'A');      /*  Bereich ueberpruefen.  */
}                                       /*************************/
```

**Prog. 3.8:** *Umwandlung in Großbuchstaben:* toupper

Für diese Fälle ist die *for*-Schleife viel zu ausdrucksstark, weswegen C dafür eine eigene Schleife bereithält, die *while*-Schleife. Eine Aufgabenstellung, die sich allgemein so umreißen läßt:

```
solange eine Bedingung wahr ist
        tue dieses-oder-jenes
```

wird am elegantesten mit diesem Schleifentyp formuliert. Eine Problembeschreibung wie die obige, die zwar im Klartext gehalten, aber bereits an die Programmstruktur angelehnt ist, nennt man übrigens "Pseudocode"-Beschreibung. Man kann diesen Pseudocode schrittweise verfeinern, ihn immer enger an des endgültige Programm anlehnen. Die Pseudocode-Beschreibung der *main*-Funktion von Programm 3.8 lautet etwas detaillierter:

```
solange (das eingelesene Zeichen ungleich 'E' ist)
     gib das Zeichen in Großschreibung aus
```

Ähnlich wie bei der *while*-Schleife steht hier die Kontrollbedingung bereits in runden Klammern; das, was die Schleife kontrolliert, steht eingerückt eine Zeile tiefer.

Die Kontrollbedingung der *while*-Schleife hat's in sich; in ihr ist nämlich das Einlesen eines Zeichens, das Zuweisen an eine Variable und der Vergleich mit einer Zeichenkonstante in sehr kompakter Form integriert. Wie bereits im Programm 3.7 wird das Zeichen von der Tastatur mit *getchar* eingelesen. Da es später noch gebraucht wird (zum Ausgeben nämlich), wird es auch gleich an eine Variable *c* zugewiesen (C-Programmierer lieben es, Variablen gelegentlich kurz und knapp zu benamsen!).

Die Zuweisung aber ist in C ein Operator.

Sie erinnern sich: Operatoren produzieren Werte. Am klarsten wird dies bei den Paradebeispielen für Operatoren, der Addition, Subtraktion etc. Die Addition nimmt zwei Zahlen und produziert daraus eine neue, nämlich deren Summe. Diese (die Summe) ist der Wert der Operation, und dieser Wert kann weiterverwendet werden. Weiterverwendet wurde er bisher meist für eine Zuweisung.

Doch auch diese produziert einen Wert. In C ist der Wert einer Zuweisungsoperation das, was an den *lvalue* zugewiesen wird. Hierzu ein Beispiel: in dem C-Ausdruck

```
a = 3 + 4;
```

produziert der arithmetische Operator + den Wert 7, welcher weiterverwendet und an die Variable (den *lvalue*) *a* zugewiesen wird. Dies besorgt - wie Sie bereits wissen - der Zuweisungsoperator =; aber auch = produziert einen Wert, nämlich ebenfalls 7, und diesen könnte ich weiterverwenden, etwa in folgender Weise:

```
b = (a = 3 + 4);
```

Diese Mehrfachzuweisung, in der *a* und *b* in einer einzigen Anweisung der Wert 7 zugewiesen wird, ist nur deshab möglich, weil C die Zuweisung als Operator behandelt und dieses deshalb wertbehaftet ist. Das unterscheidet sich von den Konventionen, die in BASIC oder Pascal befolgt werden, in denen die Zuweisung 'wertlos' ist.

Noch ein weiteres Beispiel:

```
c = (1 + (b = (1 + (a = 3 + 4))));
```

Hier erhält - wie bereits in den letzten beiden Beispielen - *a* den Wert 7 durch Addition. Der Wert der Zuweisung wird jetzt aber nicht sofort für eine weitere Zuweisung herangezogen. Vielmehr geht er in eine Addition ein, die das Ergebnis 8 liefert, welches *b* zugewiesen wird. Die Zuweisung an *b* hat ebenfalls einen Wert (nämlich 8), der wieder für eine Addition verwendet wird, die das Ergebnis 9 liefert, welches nun *c* zugewiesen wird.

Ich habe in diesen Beispielen freizügigen Gebrauch von runden Klammern gemacht, um die Verhältnisse für Sie etwas durchsichtiger zu gestalten: jede Zuweisung hat ihr eigenes Klammerpaar. C legt Ihnen beim Gebrauch der Klammern keine Beschränkung auf; gerade für den Anfang würde ich Ihnen raten, lieber zu viel als zu wenig Klammern zu verwenden. Wer gerne ohne Klammern programmiert, der muß die Ausführungen im Kapitel 4 abwarten.

Nach Ausführung dieser Anweisung liegen somit folgende Verhältnisse vor:

a hat den Wert 7
b hat den Wert 8
c hat den Wert 9

Zurück zum Programm 3.8; auch hier wird der Wert einer Zuweisung weiterverwendet, diesmal jedoch für einen Vergleich. Mit *c = getchar()* bewirkt man ja, daß *c* als Wert das eingelesene Zeichen erhält; wie Sie jetzt wissen, ist dieses eingelesene Zeichen zugleich auch Wert der Zuweisung. Deshalb kann man den folgenden C-Ausdruck

```
(c = getchar()) != 'E'
```

so lesen:

```
das eingelesene (und in c gespeicherte Zeichen) ist ungleich dem Buchstaben "E"
```

vorausgesetzt, man weiß, daß in C für das Testen auf Ungleichheit der Operator *!=* geschrieben wird. Noch ein Wort zu den Klammern: die Zuweisung ist eingeklammert (und muß es in diesem Fall auch sein; warum, wird später noch erklärt). Ein weiteres Klammerpaar wird durch *getchar* eingeführt, das eine Funktion ist und deshalb stets als Zeichen seines Standes mit Klammern geschmückt sein muß (auch wenn in denen nichts drinsteht!). Die Abbildung 3.9 legt noch einmal die Bestandteile dieser komplexen Kontrollbedingung dar.

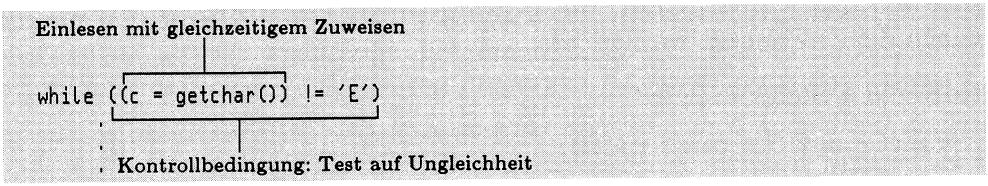



**Abb. 3.9:** *Anatomie einer komplexen Kontrollbedingung*

Ebenso wie die *for*-Schleife kontrolliert auch die *while*-Schleife eine Anweisung (oder, falls nötig, einen Anweisungsblock). Die Anweisung im Programm 3.8 soll dafür sorgen, daß das soeben eingelesene Zeichen in

Großschreibung wieder ausgegeben wird. Dazu braucht's zwei Ingredienzien: etwas, das das Zeichen in Großschreibung umwandelt und etwas, das das Umwandlungs-Ergebnis ausgibt.

Für die zweite dieser Aufgaben hält C (genauer: die C-Bibliothek) die Systemfunktion *putchar* bereit. Dieser gibt man ein Zeichen, und sie bringt es auf den Bildschirm. Schreiben Sie also in einem Programm

```
putchar('O');
```

dann sehen Sie als Wirkung dieser Anweisung ein großes "O" auf Ihrem Bildschirm. Wohlgemerkt: *putchar* kann nur mit Zeichen etwas anfangen. Es hat keinen Sinn (und zeitigt im Programm die seltsamsten Ergebnisse), ihm eine Zeichenkette (einen String) zum Ausgeben vorzulegen. Schreiben Sie also keinesfalls etwas wie das Folgende:

```
putchar("OTTO");  /*  FALSCH!!! */
```

Hierfür (zum Ausgeben eines Strings) müssen Sie vorläufig - Sie werden auch dafür etwas anderes kennenlernen - das altbewährte *printf* heranziehen.

Im Programm 3.8 soll aber nicht irgendein Zeichen ausgegeben werden, sondern das umgewandelte Eingabezeichen. Dazu basteln wir uns eine eigenen Umwandlungsfunktion (mit dem sinnfälligen Namen *toupper*); wie das geht, wird im nächsten Abschnitt beschrieben.

Aber was soll das sein: *putchar(toupper(c))*?

Es ist ein Beispiel für die Verschachtelung von Funktionen.

### 3.7.2 Was ist eine Funktion?

Dies ist der rechte Augenblick, nochmal vom Wesen und Wirken der Funktionen zu reden. Vielleicht geht es Ihnen wie mir: bei dem Wort "Funktion" fühlte ich mich lange an schmerzhafte Erfahrungen mit der Schulmathematik erinnert, dachte an endlose Rumrechnereien, Differentiale und Integrale und was der Unerfreulichkeiten mehr sind. Doch dabei sind Funktionen etwas ganz harmloses.

Dem Mathematiker ist alles eine Funktion, was aus einem oder mehreren Dingen ein anders macht. Genaugenommen ist also sogar ein Backofen eine Funktion, denn er macht aus einem bläßlichen Stück Teig einen duftenden Kuchen. Oder: eine (berufliche) Beförderung ist eine Funktion, denn sie macht aus einem armen Menschen einen weniger armen Menschen. Leider beschäftigt sich der Mathematiker meist nicht mit so handfesten Dingen; er

liebt die Zahlen und betrachtet deshalb hauptsächlich Funktionen, die aus einer oder mehreren Zahlen eine neue Zahl machen, darunter so gräßliche (oder nützliche Dinge; je nachdem, wie man's sieht), wie die Sinus-Funktion, die aus einem Winkel eine Verhältniszahl zweier Strecken macht - aber sprechen wir nicht mehr davon!

Für den Funktionsbegriff, wie er in der C-Programmierung verwendet wird, sind ohnehin die "Äußerlichkeiten" der Funktionen wichtiger: sie tun Dingen 'etwas an' (sprich: haben Argumente) und sie liefern ein Ergebnis (sprich: haben einen Wert).

Das tun jedenfalls die normalen Angehörigen der Funktionsfamilie. Natürlich gibt es auch hier Abweichler: Funktionen, die keine Argumente haben (*getchar* ist ein Beispiel dafür), solche, die keinen Wert zurückliefern (*printf* gehört zu ihnen) und solche, die weder Argument noch Wert aufweisen (wie das läppische *tue_nichts* aus dem zweiten Kapitel).

Aber die normale Funktion mit Argument(en) und Wert kann man sich graphisch mit dem berühmten schwarzen Kästchen veranschaulichen: es hat einen oder mehrere Eingänge und (höchstens) einen Ausgang. Die Abbildung 3.10 zeigt das Beispiel einer Funktion mit drei Eingängen (drei Argumenten).

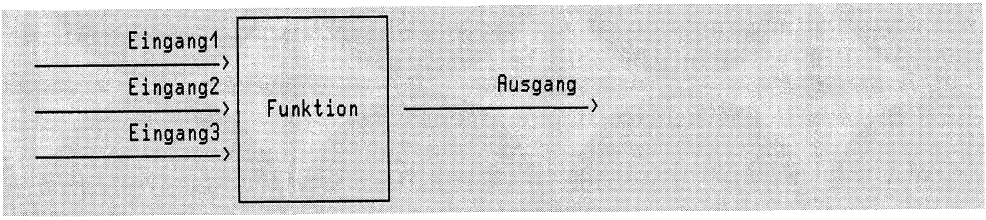



**Abb. 3.10:** *Black-Box-Diagramm einer Funktion*

Möchte man das Diagramm einer bestimmten Funktion zeichnen, dann gibt man an, was sie an ihrem Eingang (bzw. ihren Eingängen) erwartet und was sie am Ausgang liefert. In Abbildung 3.11 sehen Sie das "Black-Box"-Diagramm für die Funktion *toupper*. Sie hat als Argument und Wert Daten vom Typ "Zeichen" (*char*); dies steht unter den Ein- bzw. Ausgangspfeilen. Die Beschriftung auf den Pfeilen gibt nochmal genauer an, um was für Zeichen es sich dabei handelt.

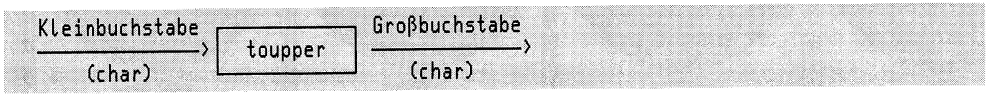



**Abb. 3.11:** *Black-Box-Diagramm der Funktion* toupper

Etwas anders verhält es sich mit den Funktionen *getchar* und *putchar*. Die erste hat kein Argument und nur einen Wert, die zweite hat ein Argument, aber keinen Wert. Wenn Sie also *putchar* anwenden, dann erzeugen Sie damit zwar eine Bildschirmausgabe, bekommen aber von dieser Funktionen keinen weiterverwendbaren Wert zurückgeliefert. Dies verdeutlicht noch einmal die Abbildung 3.12.

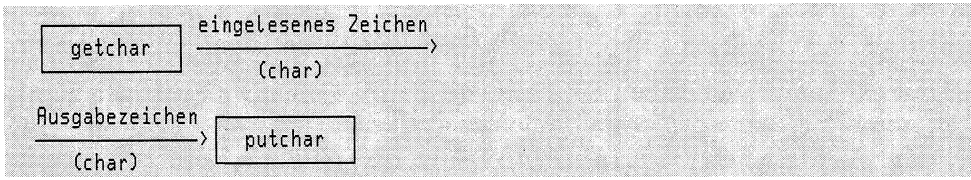



**Abb. 3.12:** *Black-Box-Diagramm von* getchar *und* putchar

Hat man mehrere dieser schwarzen Kästchen und stimmen die Ein- und Ausgänge der Funktionen in ihrem Typ überein, dann hindert einen nichts daran, diese 'hintereinanderzuhängen' oder 'zusammenzuschalten'. So kann man durch Zusammenschalten von *toupper* und *putchar* die Ausgabe eines in Großschreibung umgewandelten Zeichens bewirken. Dies veranschaulicht die Abbildung 3.13.

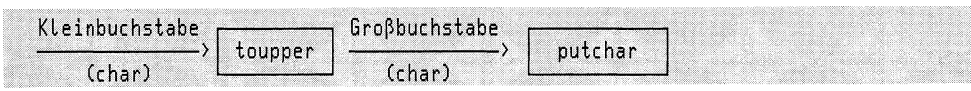



**Abb. 3.13:** *Hintereinanderschalten von Funktionen*

Genau dieses Hintereinanderschalten von Funktionen geschieht auch in der *while*-Schleife des Programms 3.8. Nur schreibt man es da anders. Was in der Grafik durch Hintereinanderschalten ausgedrückt ist, bewirkt man in C durch Ineinanderschachteln der Funktionen: mit

```
putchar(toupper(c))
```

drückt man also aus, daß die Funktion *putchar* das ausgeben soll, was ihr die Funktion *toupper* liefert; dies ist das Argument von *putchar*. *toupper* wiederum hat als Argument ganz einfach eine Variable (die ihren Wert durch Zuweisung erhalten hat).

### 3.7.3 Der Aufbau einer Funktionsdefinition

Systemfunktionen - also solche Funktionen, die Sie als Programmierer mit Ihrem C-Compiler hinzuerworben haben und die in der Standardbibliothek enthalten sind, wie etwa *printf*, *getchar* und *putchar* - sind aus Programmierersicht wirklich Black Boxes: Sie müssen nur wissen, welche

Argumente sie verlangen und welchen Wert sie produzieren. Aber wie sie ihre Arbeit tun, das kann Ihnen gleichgültig sein.

Anders bei selbstgeschriebenen Funktionen; da muß das schwarze Kästchen von Ihnen zusammengeschraubt werden, Sie müssen also über das Funktionieren der Funktion Bescheid wissen. Wie man so etwas macht, zeigt Ihnen Programm 3.8.

Sie wissen ja bereits aus Kapitel 2, daß man sich in C eigene Funktionen definieren kann. Hier haben Sie das erste Beispiel einer vollwertigen Funktion, die ein Argument hat und einen Wert zurückgibt. Die Definition einer solchen Funktion folgt dem Aufbau, wie er in Abbildung 3.14 zu sehen ist; nicht erschrecken!

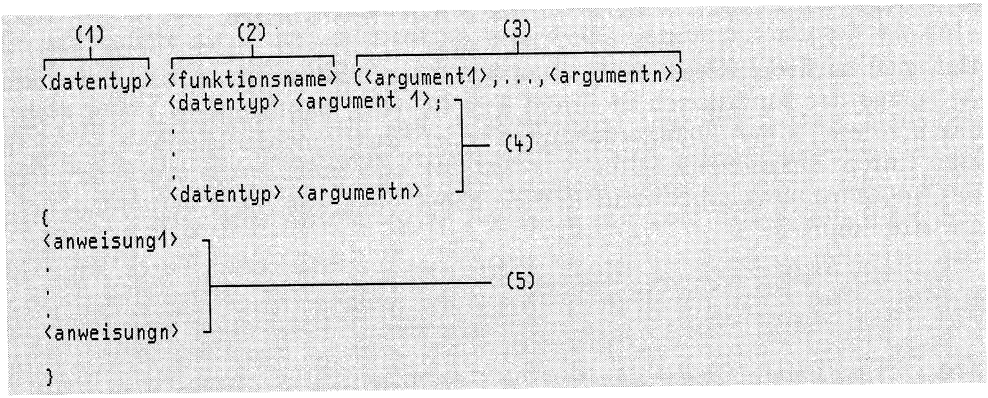



**Abb. 3.14:** *Anatomie einer Funktionsdefinition*

Ich habe die einzelnen Komponenten der Abbildung numeriert und werde sie nun einzeln durchsprechen.

1. Funktionen geben im Normalfall einen Wert zurück; dieser Wert ist von einem bestimmten Datentyp (*char* im Beispiel *toupper*) und das muß dem Compiler mitgeteilt werden. Deshalb werden Funktionsdefinitionen mit einer Datentyp-Deklaration begonnen, ähnlich wie bei Variablen ebenfalls der Typ deklariert werden muß.
2. Dann hat eine Funktion natürlich auch einen Namen; für die Bildung von Funktionsnamen gelten dieselben Gesetzmäßigkeiten wie für die Variablennamen.
3. Funktionen tun Daten etwas an, oder, vornehmer: sie haben Argumente. Bei der Definition der Funktion müssen Sie natürlich angeben, wieviele Argumente die Funktion haben soll. Sie tun das, indem sie in der Definition für die zu erwartenden Argumente Namen vergeben; diese Argumentname der Definition nennt man auch "formale Parameter". Hat eine

Funktion mehrere Argumente, so müssen Sie auch mehrere formale Parameter angeben, die dann durch Komma zu trennen sind. Im Beispielprogramm habe ich für den formalen Parameter von *toupper* den Namen *zei* gewählt (was an "Zeichen" erinnern soll). Für Parameternamen gelten dieselben Gesetzmäßigkeiten wie für alle Namen in C.

4. Formale Parameter haben viel mit Variablen gemeinsam: sie sind diejenigen Speicherstellen, in denen eine Funktion bei ihrem Aufruf die Daten empfängt, mit denen sie arbeiten soll. Ähnlich wie Variablen müssen sie daher auch deklariert werden. Im vierten Teil einer Funktionsdefinition teilen Sie deshalb dem Compiler mit, von welchem Datentyp die Argumente sind, die die Funktion in ihrem formalen Parameter empfängt. Die Deklaration der Parameter muß vor dem Anweisungsblock der Funktion hingeschrieben werden.
   Die Parameter einer Funktion sind streng lokal. Das bedeutet, daß - gleichgültig, wie sie heißen - niemals ein Namenskonflikt mit Variablen auftreten kann, die in einer anderen Funktion deklariert sind. Ich hätte statt des Namens *zei* ebensogut *c* verwenden können, obwohl es auch in der Hauptfunktion *main* eine Variable mit diesem Namen gibt. Der Compiler wäre dadurch nicht durcheinandergekommen: das *c* in der Funktion *toupper* wäre für ihn eine völlig andere Variable als das *c* in *main*. Ein Beispiel dafür bietet die zweite selbstdefinierte Funktion *islower*. Genaueres über diese Materie finden Sie im Kapitel 6.1
5. Natürlich muß eine Funktion auch was tun. Was sie tut, steht im Anweisungsblock. Dem Anweisungsblock von *toupper* gilt jetzt die Aufmerksamkeit des nächsten Unterkapitels.

### 3.7.4 Von Zeichen und Zahlen

In Pseudocode ausgedrückt, sieht die Aufgabe der Funktion *toupper* so aus:

```
wenn (das Argument zei ist ein Kleinbuchstabe)
     wandle es um in Großschreibung und gib es zurück;
ansonsten
     gib es  unverändert zurück;
```

Die Aufgabe, einen Kleinbuchstaben zu erkennen, wird in bester arbeitsteiliger Manier auf eine weitere selbstgeschriebene Funktion *islower* abgeschoben, zu der ich noch kommen werde.

Daß das umgangssprachliche "wenn...dann" in C mit *if* ausgedrückt wird, ist bereits bekannt. Die erste Zeile des Pseudocodes kann daher so neuformuliert werden:

```
if (islower(zei))
.
.
.
```

Aber hier hat man es ja nicht alleine mit "wenn...dann" zu tun. Vielmehr muß in der Funktion eine (und nur eine) von zwei möglichen Aktionen ausgewählt werden; die andere muß unterbleiben. Dieses "wenn ... dann ... ansonsten" heißt in C *if...else*; der Pseudocode kann also wie folgt verfeinert werden:

```
if (islower(zei))
     "wandle es um in Großschreibung und gib es zurück";
else
     "gib es  unverändert zurück";
```

(**Anmerkung:** Was noch nicht fertiger C-Code ist, habe ich in Anführungszeichen geschrieben). Von den verbleibenden zwei Schritten ist am leichtesten der zweite erklärt. Hier geht es um die Frage, wie man dafür sorgen kann. daß eine Funktion einen bestimmten Wert zurückgibt. "Zurückgeben" heißt im Englischen "return", und ebenso nennt sich auch die entsprechende Anweisung:

```
if (islower(zei))
     "wandle es um in Großschreibung und gib es zurück"
else
     return (zei);
```

Die *return*-Anweisung hat eine zweifache Wirkung. Wird sie bei der Programmausführung erreicht, dann wird sofort die gerade ausgeführte Funktion (im Beispiel *toupper*) verlassen und die Kontrolle über die Programmausführung geht an die Stelle im Programm zurück, die die Funktion gerufen hat (im Beispiel *putchar* innerhalb von *main*). Außerdem wird das, was in den Klammern hinter *return* steht, als Wert an die aufrufende Stelle übergeben.

In den Klammern kann einfach eine Variable stehen, aber auch ein beliebig komplexer Ausdruck, der einen Wert produziert. Ein Beispiel dafür finden Sie im *if*-Zweig der Funktion *toupper*. Um dies zu verstehen, sind ein paar Worte darüber nötig, was in C ein Zeichen ist.

Im Inneren des Computers gibt es nur Zahlen; nichts anderes kann er verarbeiten. Um dennoch mit Zeichen umgehen zu können, greift man zu einem Trick: man codiert die Zeichen, faßt sie in einer Codetabelle zusammen, in der jedes Zeichen seinen Platz hat, so daß man sie über die Platznummer identifizieren kann - und schon hat man die benötigten Zahlen.

Wie die allermeisten Home- und Personalcomputer arbeitet der ATARI mit einer genormten Codetabelle, dem American Standard Code for Information Interchange (ASCII-Code). Die ASCII-Norm definiert 128 Zeichen: Ziffern, Sonderzeichen, Groß- und Kleinbuchstaben und einige Steuerzeichen

für periphere Geräte. Darüber hinaus hält der ATARI weitere 128 Zeichen bereit, wie z.B. Grafikzeichen, nationale Sonderzeichen, griechische und mathematische Symbole, die jedoch nicht genormt sind. Die interne Codetabelle des Atari umfaßt somit 256 Zeichen, von denen die ersten 128 Positionen international genormt sind.

Was für den Menschen ein Zeichen ist, ist für den Computer lediglich eine Zahl zwischen 0 und 255 (das erste Zeichen der Codetabelle besetzt die Position 0; in der EDV liebt man es ohnehin, bei Null zu beginnen). So ist es in allen Programmiersprachen; C ist jedoch einzigartig, weil es dem Programmierer erlaubt, diese Zeichen-Zahlen mit den üblichen arithmetischen Operationen zu traktieren. Zu einem sichtbaren Zeichen werden die Zahlen erst wieder, wenn der Computer sie an ein Ausgabegerät schickt (zum Beispiel mit *putchar* an den Bildschirm). Dieses weiß dann, daß es die Zahlen umzuwandeln hat in die für den Menschen lesbaren Buchstaben.

In der ASCII-Tabelle sind die Buchstaben alphabetisch sortiert; außerdem kommen die Großbuchstaben vor den Kleinbuchstaben. Die Großbuchstaben "A" bis "Z" nehmen die Positionen 65 bis 90 ein, die Kleinbuchstaben "a" bis "z" findet man zwischen 197 und 122. Um von einem Kleinbuchstaben zu dem entsprechenden Großbuchstaben zu kommen, braucht man daher nur die Distanz zwischen diesen beiden Tabellenausschnitten vom Kleinbuchstaben zu subtrahieren. Diese erhält man ganz einfach, indem man den Zahlenwert der Zeichenkonstante 'A' von dem der Konstante 'a' subtrahiert: 'A' - 'a'. Dies ergibt den negativen Wert -32, weswegen ich auch in *toupper* schreiben könnte:

```
return (zei - 32);
```

Aber dann hätten Sie nicht so viel über die Zeichen-Arithmetik gelernt!

Daß man in C mit Zeichen nicht nur Rechnen, sondern daß man auch die üblichen Vergleichsoperationen darauf anwenden kann, zeigt die Funktion *islower*. Die überprüft einfach, ob ihr Argument sich im Tabellenbereich für Kleinbuchstaben aufhält. In Pseudocode:

```
das Zeichen liegt zwischen 'a' und 'z'
```

bzw., in verfeinerter Version:

```
das Zeichen ist größer-gleich 'a'
UND das Zeichen ist kleiner-gleich 'z'
```

UND schreibt man in C mit *&&*; somit erhält man:

```
(c >= 'a' && c <= 'A')
```

Das Resultat dieses Vergleichs (entweder "wahr" oder "falsch") wird als Wert der Funktion zurückgegeben. Die Wahrheitswerte haben in C übrigens den gleichen Datentyp wie die ganzen Zahlen. Deswegen ist der Typ der Funktion *islower* auch *int*.

Noch etwas: die Funktionen *toupper* und *islower* müssen bei einigen C-Compilern nicht selbst definiert werden. Sie sind in diesen Systemen Bestandteil der Standard-Bibliothek. In dem C-Compiler, der Bestandteil des ATARI-Entwicklungssystem ist, waren Sie jedoch zum Zeitpunkt der Erstellung dieses Buches (Winter 1985) noch nicht vorhanden. Sie sollten also erst einen Blick in Ihre Systemdokumentation werfen, um sich bei der praktischen Arbeit unnötigen Mehraufwand zu ersparen.

# 4 Datentypen, Operatoren und Kontrollstrukturen

In den letzten beiden Kapiteln wurde bereits eine Vielzahl von Sprachelementen von C vorgeführt. Das vierte Kapitel hat die Aufgabe, dieses Material zu vervollständigen und zu systematisieren.

## 4.1 Datentypen

Variablen und selbstdefinierte Funktionen, die einen Wert zurückgeben, müssen hinsichtlich ihres Datentyps deklariert werden. Dazu muß der Programmierer wissen, welche elementaren Datentypen es in C gibt. Hierbei erscheint das Angebot von C auf den ersten Blick recht kärglich, denn die Sprache kennt nur Zahlen und Zeichen (und Zeiger auf diese Datentypen; aber dies ist Gegenstand eines eigenen Kapitels). Wie sich aber noch zeigen wird, ist dies völlig ausreichend für alle Programmieraufgaben. Außerdem entspricht es der Philosophie von C, nach der die Sprache möglichst eng an die Fähigkeiten der Maschine angelehnt sein soll, um dem Programmierer maximale Kontrolle darüber zu erlauben.

Die Schlüsselworte, mit denen man Daten eines elementaren Datentyps deklarieren kann, finden Sie in Abbildung 4.1. Die Tabelle ist in zwei Teile gegliedert; im ersten Teil finden Sie die eigentlichen elementaren Datentypen mit dem zugehörigen Schlüsselwort für die Deklaration. Die Schlüsselworte im zweiten Teil sind Modifikatoren, mit denen die elementaren Datentypen genauer bestimmt bzw. eingeschränkt werden können.

| Schlüsselwort | Bedeutung |
|---|---|
| *char* | Zeichen |
| *int* | Ganze Zahl |
| *long* | Doppelt genaue ganze Zahl |
| *float* | Gleitkommazahl |
| *double* | Doppelt genaue Gleitkommazahl |
| *short* | Modifikator für *int* |
| *long* | Modifikator für Zahlen |
| *unsigned* | Modifikator für *char* und *int* |

**Abb. 4.1:** *Die elementaren Datentypen von C*

Bei jedem dieser Datentypen muß man sich zwei grundsätzliche Fragen stellen: wie sehen Angehörige dieses Datentyps für den Menschen aus, und wie sieht sie die Maschine?

### 4.1.1 Zeichen

Zeichen sind die elementaren Bausteine unserer Kommunikation mit dem Computer. Wir unterteilen die Zeichen in mehrere Kategorien: Buchstaben (klein-und großgeschrieben), Ziffern (0,1,2,...9) und Sonderzeichen (für die Interpunktion, aber auch Klammern, mathematische Symbole usw.). Das sind die Zeichen, die Sie auf der Tastatur Ihres ATARI finden, auf den Tasten, die bei Betätigung (alleine oder zusammen mit den Umschalttasten *Shift* und *Alternate*) ein sichtbares Ergebnis auf dem Bildschirm zeitigen.

Daneben gibt es aber auch noch Zeichen, die man nicht sehen kann. Diese "Zeichen" zu nennen ist etwas ungewöhnlich; genaugenommen sind es sog. "Steuerzeichen". Sie dienen der Steuerung peripherer Geräte, sorgen etwa dafür, daß Ihr Terminal piepst oder daß am Bildschirm oder Drucker ein Zeilen- oder Seitenvorschub erzeugt wird. Als Programmierer müssen Sie wissen, wie Sie gewöhnliche Zeichen und Steuerzeichen notieren (um sie als Konstanten in Ihren Programmen verwenden zu können).

Zeichen können als Konstanten in Programmen benötigt werden. Ist ein Zeichen darstellbar (also kein Steuerzeichen), so wird es als Konstante einfach zwischen einfache Anführungszeichen gesetzt; hier einige Beispiele:

```
'A'  /* Buchstabe */
'7'   /* Ziffer */
';'   /* Semikolon */
' '   /* Leerzeichen */
'_'   /* Unterstrich */
```

Da die einfachen Anführungszeichen als Begrenzer für Zeichenkonstanten dienen, ist es nicht ohne weiteres möglich, ein einfaches Anführungszeichen selbst als Zeichenkonstante hinzuschreiben. Dafür und für die nicht-darstellbaren Zeichen benutzt man den bereits im Kapitel 2 im Zusammenhang mit *printf* vorgestellten Escape-Mechanismus zur Ersatzdarstellung. Auch hierzu noch einige Beispiele:

```
'\''        /* Einfaches Anfuehrungszeichen */
'\n'        /* Neue Zeile */
'\\'        /* Der umgekehrte Schraegstrich */
'\7'        /* Der Atari piepst */
'\101'      /* Dasselbe wie 'A' */
```

Die ASCII-Tabelle (siehe Kapitel 3) umfaßt 32 Steuerzeichen; einige dieser Steuerzeichen (z.B. Neuzeile, Tabulator) haben in C einen eigenen (einbuchstabigen) Namen; dies zeigt Ihnen die Tabelle der Abbildung 2.7. Die

meisten Steuerzeichen sind aber anonym. Will man sie dennoch darstellen, dann benutzt man die Methode, die das letzte Beispiel illustriert: hinter den umgekehrten Schrägstrich schreibt man eine dreistellige Zahl, die die Position des gewünschten Zeichens in der ASCII-Tabelle darstellt. Dummerweise muß man diese Zahl in Oktaldarstellung hinschreiben (das oktale Zahlensystem ist der Favorit der UNIX-Hacker). Aber keine Angst: die ASCII-Tabelle im Anhang des Buches enthält für jedes Zeichen auch die Oktalwerte, die Sie also nur dort nachzusehen brauchen. Durch Angabe einer oktalen Tabellenposition erreichen Sie übrigens auch die nicht-standardisierten oberen 128 Zeichen (Grafikzeichen etc.) des Atari-Zeichensatzes. Das Programm 4.1 gibt Ihnen ein Beispiel für die Verwendung von Zeichenkonstanten und insbesondere für die Ersatzdarstellungen.

```
#include <stdio.h>

main()                                  /****************************/
                                        /*  Ausgeben verschiedener  */
{   char c;                             /*  Zeichenkonstanten.      */
                                        /****************************/
    c = 'A'; putchar(c);                /*  Buchstabe               */
    c = '\\'; putchar(c);               /*  Umgekehrter Schraegstr.*/
    c = '\''; putchar(c);               /*  Einf. Anfuehrungsz.     */
    c = '\n'; putchar(c);               /*  Zeilenvorschub          */
    c = '\7'; putchar(c);               /*  Pieps!                  */
    c = 'A'; putchar(c);                /*  Buchstabe A             */
    c = '\101'; putchar(c);             /*  ditto                   */
    c = '\210'; putchar(c);             /*  Grafikzeichen: e mit    */
                                        /*    Akzent                */
                                        /****************************/
}
```

**Prog. 4.1:** *Zeichenkonstanten*

Was für den Menschen ein Zeichen ist, das ist für den Computer eine Zahl; im Kapitel 3 wurde dieser Sachverhalt bereits erwähnt. Da es im Zeichensatz des ATARI insgesamt 256 Zeichen gibt, muß der Computer zur Speicherung eines Zeichens soviel Platz im Arbeitsspeicher reservieren, daß Zahlen zwischen 0 und 255 darin dargestellt werden können. Dazu benötigt man bekanntermaßen 8 Bits oder ein Byte. Somit kann man zusammenfassen: zur Speicherung eines Zeichens reserviert der C-Compiler ein Byte; für den Computer ist ein Zeichen nichts anderes als eine Zahl zwischen 0 und 255. Der Zahlenwert bezieht sich auf die Position des Zeichens in der Codetabelle.

Diese enge Verwandschaft zwischen Zahlen und Zeichen geht sogar soweit, daß man C anstelle von Zeichen auch Zahlen unterschmuggeln kann (vorausgesetzt man sorgt dafür, daß sich diese Zahlen im erlaubten Bereich zwischen 0 und 255 bewegen). Dies ist der Grund, warum das Programm

4.2 in einer *for*-Schleife alle abdruckbaren Zeichen ausgibt (diese beginnen ab der Position 32 mit dem Leerzeichen), obwohl in der *for*-Schleife eine Integer-Variable hochgezählt und in der Kontrollbedingung mit einer Zahlenkonstante verglichen wird.

```
#include <stdio.h>

main()                                  /*****************************/
                                        /*  Zeichen sind Zahlen!     */
{                                       /*****************************/
    int c;                              /*                           */
                                        /*  Zaehlt die Positionen    */
    for (c = 32; c <= 255; ++c)         /*  in der ASCII-Tabelle.    */
    {   putchar(c);                     /*  Zeichen ausgeben.        */
        putchar('\n');                  /*  Neue Zeile: Zeilen-      */
        putchar('\r');                  /*   vorschub und Wagen-     */
    }                                   /*   ruecklauf.              */
}                                       /*****************************/
```

**Prog. 4.2:** *Zusammenhang von Zeichen und Zahlen*

Eine Anmerkung zu diesem Programm ist jedoch nötig: dies ist kein portabler Programmierstil. Sie können Programme, die mit der absoluten Position von Zeichen in der Codetabelle arbeiten, nur auf solche Rechner übertragen, die genau die gleiche Codetabelle wie Ihr Computer verwenden. Bezüglich der ersten 128 Zeichen sollte es da wegen der ASCII-Norm jedoch keine Schwierigkeiten geben.

### 4.1.2 Integers

Integers sind ganze Zahlen. Sie sind in C in mehreren Geschmacksrichtungen erhältlich. Mit dem Schlüsselwort *int* werden vorzeichenbehaftete ganze Zahlen deklariert, die sich (auf dem ATARI) im Bereich zwischen - 32768 und +32767 bewegen.

Zur Darstellung eines Werts vom Typ *int* benötigt die Maschine 16 Bit bzw. zwei Byte an Speicherplatz. Das höchstwertige Bit (Bit 15) ist dabei für die Darstellung des Vorzeichens reserviert (Bit 15 gesetzt bedeutet: die Zahl ist negativ; Bit 15 nicht gesetzt signalisiert eine positive Zahl). Die Bits 14 mit 0 im Byte verbleiben somit für die Darstellung des Zahlenbetrags, wobei negative Zahlen im Zweierkomplement dargestellt sind.

Wenn Sie in einem Programm keine positiven Zahlen benötigen, dann können Sie zu dem Datentyp *unsigned int* greifen (das Englische *unsigned* bedeutet "vorzeichenlos"). Auch eine *unsigned int* benötigt 16 Bits (zwei Byte) zu ihrer Speicherung, die jedoch allesamt zur Darstellung des Zahlenbetrages verfügbar sind (ein Vorzeichenbit wird ja nicht benötigt). Damit

können Daten vom Typ *unsigned int* Werte zwischen 0 und 65535 annehmen.

Wie ihr Name schon sagt, kann eine *unsigned int* nicht negativ werden. Hierin liegt eine Gefahr, wenn man Variablen dieses Typs in Kontrollbedingungen von Schleifen benutzt. Was passiert, wenn folgendes Programmfragment ausgeführt wird?

```
main()

{   unsigned int i;

    for (i = 20; i >= 0; --i)
    .
    .
    .
}
```

Man könnte annehmen, daß in diesem Beispiel die *for*-Schleife insgesamt 21 mal durchlaufen wird. Aber da ist man einem Irrtum aufgesessen. Zwar wird die Schleifenvariable gegen 0 gezählt. Wenn sie jedoch den Wert 0 erreicht hat und noch einmal dekrementiert wird, dann ist das Ergebnis keinesfalls -1 (und das Endekriterium für die Schleife somit erfüllt). Stattdessen wird C versuchen, das sich beim Dekrementieren von 0 ergebende Bitmuster (16 binäre Einsen) als positive Zahl zu interpretieren (als den Wert 65535): Sie sind in einer Endlosschleife gefangen!

Wem der Wertevorrat des Datentyps *unsigned int* immer noch zu wenig ist, der sollte mit *long int* arbeiten (englisch *long* bedeutet "lang"). Für eine "lange Integer" reserviert der Computer 32 Bits oder 4 Byte und ermöglicht Ihnen so einen Wertevorrat von -2.147.483.649 bis +2.147.483.648.

Die Modifikatoren *unsigned* und *long* können auch ohne das Schlüsselwort *int* hingeschrieben werden. Es sind also in einem Programm als Deklarationen äquivalent:

| | | |
|---|---|---|
| *long int* | und | *long* |
| *unsigned int* | und | *unsigned* |
| *short int* | und | *int* |

Letzteres ist der Fall, weil C eine Integer ohnehin als *short int* vereinbart, wenn Sie nichts anderes angeben: *short* ist der Standard-Typ von C.

Integer-Konstanten können Sie in der gewohnten Manier im Dezimalsystem einfach hinschreiben. Übersteigt der Zahlenwert der Konstanten das, was noch in eine vorzeichenbehaftete Integer paßt, so weist der Compiler dieser Konstanten automatisch den Typ *long* zu. Sie können jedoch - was manchmal ganz praktisch ist - Ihre Konstanten in einem anderen Zahlensystem

als dem dezimalen notieren. Sie haben hier die Wahl zwischen hexadezimalen und oktalen Konstanten. Beginnt eine Konstante mit der Ziffer 0 ("Null"), so wird sie als Oktalzahl interpretiert (und es sind nur mehr die Ziffernzeichen '0' mit '7' in ihr zugelassen). Beginnt die Konstante mit der Zeichenfolge *0x* oder *0X*, dann stellt sie eine Hexadezimalzahl vor, in der neben den Ziffernzeichen '0' mit '9' auch die "Ziffern" 'a' bzw. 'A' bis 'f' bzw. 'F' vorkommen können.

Manchmal ist es notwendig, einer Konstanten, die ihrem Betrag nach als einfache *int* durchgehen würde, den Datentyp *long* unterzuschmuggeln. Sie erreichen dies, indem Sie hinter die Konstante (die dezimal, oktal oder hexadezimal notiert sein kann) den Buchstaben 'L' schreiben (Achtung: diese Möglichkeit ist im ATARI-C nicht gegeben!). Hier einige Beispiele:

```
99        /* Eine Schnapszahl */
0x63      /* dasselbe hexadezimal */
0143      /* dasselbe oktal */
0x63L     /* die Dezimalzahl 99 als long-Konstante; sie
             beansprucht 4 Byte Speicherplatz */
```

### 4.1.3 Gleitpunktzahlen

Zwar stellt der Datentyp *long int* Zahlen von beträchtlicher Größe bereit; doch leider kommt man in der Mathematik mit den ganzen Zahlen nicht aus. Für alle Zwecke, in denen Zahlen mit Nachkommastellen benötigt werden, gibt es in C die Datentypen *float* und *double*.

Leider sind die momentan für den ATARI verfügbaren Compiler bei den Gleitpunktzahlen noch etwas schwach auf der Brust (auch der Compiler des Entwicklungssystems bildet da keine Ausnahme). Üblicherweise werden zur Speicherung einer Gleitpunktzahl vom Typ *float* 4 Byte und für eine *double* (was eine andere Schreibweise für *long float* ist) 8 Byte reserviert. Aber um letzte Aufklärung sollten Sie in dieser Materie das Handbuch Ihres Compilers bemühen.

### 4.1.4 Wahrheitswerte

Bei Vergleichen (z.B. mit *==*, *<=*, *!=* und deren Verwandten), bei logischen Verknüpfungen (*&&* haben Sie bereits kennengelernt) und in der Kontrollbedingung der *for* und *while*-Schleife werden Wahrheitswerte benötigt. Vergleiche und logische Verknüpfungen liefern die Werte *wahr* oder "falsch", Schleifen werden solange ausgeführt, solange eine Bedingung "wahr" ist.

Was aber ist in C "wahr" oder "falsch"?

C verfolgt hier einen besonders eleganten Weg. Die Wahrheitswerte "wahr" und "falsch" sind Zahlen. Dabei wird nur festgelegt, was "falsch" ist, nämlich die Zahl 0. Alles andere (alles, was ungleich 0 ist) gilt als "wahr".

Es gibt also nur eine Wahrheitswertkonstante für "falsch", aber beliebig viele für "wahr". Es hat sich jedoch eingebürgert, in C mit zwei über *define* engeführten Konstanten zu arbeiten:

```
#define TRUE 1   /* wahr */
#define FALSE 0  /* falsch */
```

Diese Konvention ermöglicht einige sehr elegante Formulierungen. Benötigt man in C aus irgendeinem Grund eine Endlosschleife, so schreibt man einfach:

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

oder, etwas leserlicher:

```
while(TRUE)
{
     .    /* Endlosschleife */
}
```

oder, am saubersten:

```
#define FOREVER while(1)

FOREVER
{
     .    /* Endlosschleife */
}
```

Die Gleichsetzung des Wahrheitswerts "falsch" mit der Zahl 0 macht es außerdem möglich, die Ergebnisse arithmetischer Operationen oder sogar nur Integer-Variablen als Kontrollbedingung zu verwenden. Die *for*-Schleife im Programm 4.3 zählt von 10 nach 1; beachten Sie ihre Kontrollbedingung.

```
main()                                  /* Numerische Variable als */
                                        /*  Kontrollbedingung. Die */
{   int i;                              /*  Schleife wird durch-   */
                                        /*  laufen, solange der    */
    for (i = 10; i ; --i)               /*  Wert von "i" ungleich  */
        printf("%d",i);                 /*  0 ist.                 */
}
```

**Prog. 4.3:** *Integer-Variable als Kontrollbedingung einer Schleife*

## 4.2 Operatoren

Daten sind Dinge, mit denen der Computer etwas tun kann. Was kann er mit ihnen tun? Darüber geben die Operatoren Auskunft, über die eine Sprache verfügt. Hier ist das Angebot von C besonders reichlich.

Alle Operatoren haben eines miteinander gemeinsam, was Sie bereits von den arithmetischen Operatoren (und den selbstdefinierten Funktionen) kennen: sie arbeiten mit einem oder mehreren Datenwerten (den Argumenten) und produzieren damit einen neuen Wert. Um Ordnung in das Angebot der Operatoren zu bringen, ist es üblich, sie in verschiedene Kategorien zu unterteilen. Bei der Kategorisierung ist ausschlaggebend, von welchem Datentyp die Argumente und das Ergebnis sind. Damit gelangt man zu folgenden Klassen: den arithmetischen Operatoren, den Vergleichsoperatoren, den Bitoperatoren, und den Operatoren für logische Verknüpfungen. Außerdem ist - wie Sie bereits erfahren haben - in C auch die Zuweisung eine Operation, so daß als fünfte Klasse noch die Zuweisungsoperatoren hinzukommen. Als - äußerst praktisches! - Kuriosum, das sich in keiner anderen Programmiersprache findet, kennt C auch noch einen dreistelligen "Wenn-Dann-Operator", der in eine eigene sechste Kategorie einzuordnen ist.

| Einstellige Operatoren | Bedeutung |
|---|---|
| ++ | Auto-Inkrement |
| -- | Auto-Dekrement |
| - | Negative Zahl bilden |

| Zweistellige Operatoren | Bedeutung |
|---|---|
| + | Addition |
| - | Subtraktion |
| * | Multiplikation |
| / | Division |
| % | Divisionsrest |

**Abb. 4.2:** *Die arithmetischen Operatoren von C*

### 4.2.1 Arithmetische Operatoren

Arithmetische Operatoren sind Operatoren, die mit Zahlen operieren und als Ergebnis wieder eine Zahl liefern. Dabei darf man das nicht so eng sehen; denn in C sind ja auch Zeichen Zahlen und ebenso die Wahrheitswerte. Es ist also möglich, mit Daten dieses Typs arithmetische Operationen

durchzuführen. Allerdings muß der Programmierer dann schon genau wissen, was er macht! In der Abbildung 4.2 finden Sie eine Tabelle der arithmetischen Operatoren in C.

In dieser Tabelle sind die Operatoren nach ihrer Stelligkeit untergliedert, also nach der Anzahl der Argumente, die sie benötigen. Die einstelligen Operatoren für Auto-Inkrement und Auto-Dekrement sind Ihnen - wie die meisten anderen Einträge in dieser Tabelle - ja bereits bekannt. Neu ist jedoch, daß es für diese beiden Operatoren zwei unterschiedliche Schreibweisen gibt, die auch Unterschiedliches bewirken, Dies macht das Programmbeispiel 4.4 deutlich.

```
#define AUSGABE printf("\na = %d, b = %d",a,b)

main                                    /*************************/
{                                       /*  Inkrement u. Dekrement */
    int a,b;                            /*  in Prä- und Postfix-   */
                                        /*  Schreibweise.          */
    a = 10;                             /*************************/
    b = a++;                            /*                        */
    AUSGABE;                            /*  Postinkrement.        */
    a = 10;                             /*                        */
    b = ++a;                            /*                        */
    AUSGABE;                            /*  Präinkrement.         */
    a = 10;                             /*                        */
    b = a--;                            /*                        */
    AUSGABE;                            /*  Postdekrement.        */
    a = 10;                             /*                        */
    b = --a;                            /*                        */
    AUSGABE;                            /*  Prädekrement.         */
                                        /*                        */
}                                       /*************************/
```

**Prog. 4.4:** *Auto-Inkrement und Auto-Dekrement*

Wie Sie sehen, kann man diese beiden Operatoren vor bzw. hinter ihr Argument schreiben. Die Wirkung der Operatoren ist in beiden Fällen die gleiche: eine Variable wird hoch- bzw. heruntergezählt. Aber *++* und *--* produzieren auch einen Wert (jeder Operator muß das tun!) und der unterscheidet sich bei den verschiedenen Schreibweisen. Es gelten diese Regeln:

Steht *++* vor seinem Argument, so ist der Wert der Operation der Inhalt der Variablen nach dem Hochzählen. Steht *++* hinter seinem Argument, so ist der Wert der Inhalt der Variablen vor dem Hochzählen. Entsprechendes gilt für *--*. Das Programm 4.4 erzeugt somit folgendes Ergebnis:

```
a = 11, b = 10
a = 11, b = 11
a = 9, b = 10
a = 9, b = 9
```

Das Zeichen "-" führt - in C ebenso wie in der Arithmetik - ein Doppelleben. Als einstelliger Operator dient es zur Kennzeichnung von negativen Zahlen. Als zweistelliger Operator bewirkt es die Subtraktion seiner Argumente.

Die anderen zweistelligen Operatoren sind Ihnen ja - bis auf % - bereits vertraut. Dieser Operator dient zur Bildung des Divisionsrests. Sein Funktionieren wird durch Programm 4.5 verdeutlicht. Dazu ist zu sagen, daß der 'normale' Divisionsoperator / nur ganzzahlige Divisionsergebnisse liefert, wenn er mit Integer-Argumenten arbeitet. Mehr darüber finden Sie im Kapitel über die Typumwandlung.

```
main()                                          /***************************/
                                                /*  Division und Rest      */
{                                               /***************************/
    int i;                                      /*                         */
                                                /*                         */
    for (i = 1; i <= 10; ++i)                   /*                         */
        printf("\n25 / %d ist %d Rest %d",      /*                         */
               i,                               /*  der Divisor            */
               25 / i,                          /*  ganzzahlige Division   */
               25 % i);                         /*  Divisionsrest          */
}                                               /***************************/
```

**Prog. 4.5:** *Division (/) und Divisionsrest (%)*

Das Programm 4.5 erzeugt folgende Ausgabe:

```
25 / 1 ist 25 Rest 0
25 / 2 ist 12 Rest 1
25 / 3 ist 8 Rest 1
25 / 4 ist 6 Rest 1
25 / 5 ist 5 Rest 0
25 / 6 ist 4 Rest 1
25 / 7 ist 3 Rest 4
25 / 8 ist 3 Rest 1
25 / 9 ist 2 Rest 7
25 / 10 ist 2 Rest 5
```

| Operator | Bedeutung |
|---|---|
| > | *arg1* größer als *arg2*? |
| < | *arg1* kleiner als *arg2*? |
| >= | *arg1* größer oder gleich *arg2*? |
| <= | *arg1* kleiner oder gleich *arg2*? |
| == | *arg1* gleich *arg2*? |
| != | *arg1* ungleich *arg2*? |

**Abb. 4.3:** *Die Vergleichsoperatoren von C*

### 4.2.2 Vergleichsoperatoren

Vergleichsoperatoren arbeiten mit Zahlen und/oder Zeichen und produzieren einen Wahrheitswert. Die Abbildung 4.3 gibt Ihnen einen Überblick über die verfügbaren Vertreter dieser Klasse.

Alle Vergleichsoperatoren sind - natürlich - zweistellig und bieten weiter keine besonderen Überraschungen. Eine Warnung sei jedoch an dieser Stelle nochmal eindringlichst wiederholt: den Test auf Gleichheit schreibt man in C mit *zwei Gleichheitszeichen*! Ein beliebter Anfänger-Fehler ist es, für den Gleichheitstest die BASIC-Konvention zu verfolgen und zu schreiben:

```
a = 5    anstatt   a == 5
```

Was passiert, wenn Sie sowas in einer Schleife als Kontrollbedingung stehen haben? Werfen Sie mal einen Blick auf das folgende falsche Programm:

```
main()                                  /*************************/
                                        /*  Unfreiwillige Endlos-  */
{                                       /*  schleife. NICHT AUS-   */
    int i;                              /*  PROBIEREN!             */
                                        /*************************/
    for (i = 0; i = 5; ++i)             /*  Zuweisung statt Ver-   */
    printf("\n%d",i);                   /*  gleich in for-Schleife.*/
}                                       /*************************/
```

**Prog. 4.6:** *Ein beliebter Bug*

Dieses Programm wird eine endlose Folge von Fünfen auf dem Bildschirm ausgeben, jede auf einer eigenen Zeile. In der Kontrollbedingung der *for*-Schleife findet sich eine Wertzuweisung und keinesfalls - wie vielleicht beabsichtigt - ein Vergleich. Aber C akzeptiert dies; denn die Wertzuweisung hat den Wert 5 und alles, was ungleich 0 ist, wird von C als der Wahrheitswert "wahr" aufgefaßt. Bei jedem Schleifendurchgang wird erneut 5 an *i* zugewiesen, produziert den Wert 5 und somit "wahr" und sorgt so dafür, daß die Schleife kein Ende nimmt. Daß im Reinitialisierungs-Teil die Variable *i* inkrementiert wird, hat keinerlei Auswirkung auf das Ergebnis. Denn vor jedem erneuten Eintritt in die Schleife prüft C ja die Bedingung ab, sprich: führt (fälschlich) eine Wertzuweisung durch.

### 4.2.3 Bitoperatoren

Diese Familie von Operatoren wird besonders dann häufig eingesetzt, wenn maschinennahe Programmieraufgaben anstehen. Im weiteren Verlauf dieses Buches finden sich etliche Beispiele für ihre Anwendung. Für ein Verständnis der Bitoperatoren ist es nötig, die elementaren logische Verknüpfungen UND, ODER, EXKLUSIVES ODER und NEGATION zu verstehen. Eine Übersicht hierzu finden Sie in der Abbildung 4.4.

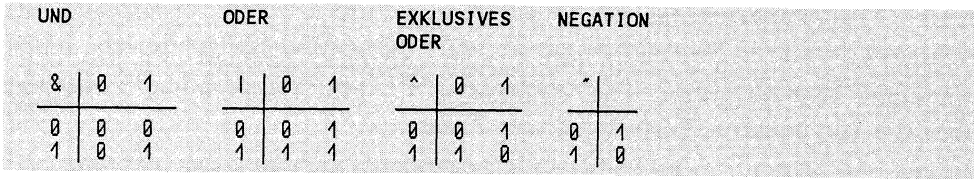



**Abb. 4.4:** *UND, ODER, EXKLUSIVES ODER und NEGATION*

Diese vier Tabellen zeigen Ihnen, wie zwei Bits mit den Operatoren UND, ODER bzw. EXKLUSIVES ODER verknüpft werden; in der linken oberen Ecke finden Sie auch das in C verwendete Zeichen für die jeweilige Operation. Die Negation ist eine einstellige Operation. Sie dreht lediglich den Bit-Wert ihres Arguments um. Die Bitoperatoren in C arbeiten jedoch nicht auf einzelnen Bits. Vielmehr werden mit ihnen stets zwei 16 Bit lange Größen (also *ints*) verknüpft, wobei die Operatoren alle 16 Bits zugleich bearbeiten. Ein Beispiel soll dies deutlich machen.

Angenommen, die Variablen *i* und *j* enthalten die binären Werte, die Sie im Speicherdiagramm der Abbildung 4.5 sehen können. Der Inhalt der Variablen *i* entspricht dem dezimalen Zahlenwert 39323 bzw 0x999b in Hexadezimalschreibweise. In *j* ist die Zahl 17524 bzw. 0x4474 gespeichert (es wurde vorausgesetzt, daß sowohl *i* als *j* vom Typ *unsigned* sind).

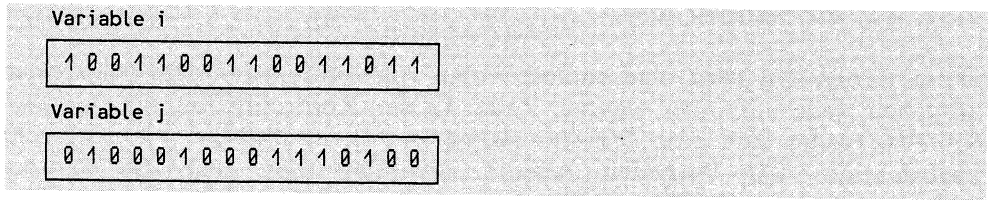



**Abb. 4.5:** *Bitweise logische Verknüpfung*

Die folgenden Operationen liefern dann das unten aufgeführte Ergebnis (jeweils in binärer, dezimaler und hexadezimaler Notation):

```
i & j = 10000 = 16 = 0x10
i | j = 1101110111111111 = 56831 = 0xDDFF
i ^ j = 1101110111101111 = 56815 = 0xDDEF
   ~i = 0110011001100100 = 26212 = 0x6664
   ~j = 1011101110001011 = 48011 = 0xBB8B
```

Es werden also immer die einzelnen Bitpositionen der Variablen *i* und *j* miteinander verknüpft. Dabei liefert die UND-Verknüpfung nur dann den Wert Eins im Ergebnis, wenn beide Bitpositionen Eins sind; ODER liefert nur dann Null, wenn beide verknüpften Bitpositionen Null sind; beim EXKLUSIVEN ODER gibt's dann eine Eins, wenn die verknüpften Positionen

unterschiedliche Werte haben. Die einstellige NEGATION dreht einfach alle Bitpositionen um: aus Eins mach Null und umgekehrt.

Obwohl bei Bitverknüpfung immer mit 16 Bit auf einmal gearbeitet wird, können Sie diese Operatoren natürlich auch zusammen mit *char*-Variablen einsetzten. Diese werden dann für die Dauer der Bitoperation zweitweise auf 16 Bit 'ausgedehnt' (mehr darüber im Kapitel 6).

Die Bitoperatoren arbeiten zwar immer mit einer ganzen 'Bitleiste'; dennoch ist es möglich, mit ihnen den Status eines einzelnen Bits zu testen. Dazu bedient man sich sogenannter "Masken", das sind Konstanten, in denen bestimmte den Programmierer interessierende Bitpositionen gesetzt sind. Möchte ich etwa herausfinden, ob in einer Variablen das zweite Bit gesetzt ist, dann werde ich zu folgendem Code greifen:

```
#define MASKE 0x0002
        .
        .
        if (i & MASKE)
            printf("\nBit 2 ist gesetzt!);
        .
        .
```

Dies funktioniert so: in der hexadezimalen Konstante 0x0002 ist nur das zweite Bit gesetzt; alle anderen haben den Wert 0. Wenn ich nun irgendeine Variable über UND mit dieser Maske verknüpfe, dann hat das Ergebnis nur dann einen Wert ungleich 0, wenn auch in dieser Variablen das zweite Bit gesetzt ist. Ein Wert ungleich Null bedeutet in C bekanntermaßen "wahr" und die Bedingung im *if* ist erfüllt.

In ähnlicher Weise ist es auch möglich, mit dem ODER-Operator ein einzelnes Bit zu setzen oder über UND-Verknüpfung und mit einer geeignete Maske das obere oder untere Byte einer Integer auszumaskieren. In späteren Kapiteln dieses Buches finden Sie mehrere Beispiele für den Nutzen dieser Techniken.

```
main()
{
    unsigned int i, j;

    i = 2; j = 3;

    printf("\nVorher: i = %d j = %d",i,j);
    i = i ^ j;
    j = i ^ j;
    i = i ^ j;
    printf("\nNachher: i = %d j = %d",i,j);
}
```

**Prog. 4.7:** *Die wundersamen Eigenschaften des EXKLUSIVEN ODER*

Noch eine verblüffende Eigenschaft der EXKLUSIV-ODER-Verknüpfung sei erwähnt. Mit ihr ist es möglich, ohne Rückgriff auf eine Hilfsvariable den Inhalt zweier Variablen auszutauschen; sehen Sie dazu im Programm 4.7 nach.

Neben den Verknüpfungsoperatoren gibt es in C noch zwei Verschiebe-Operatoren. Mit ihnen ist es möglich, die Bits eines Zeichens oder einer Zahl um eine gegebene Anzahl an Positionen nach links (mit dem Operator *<<*) oder rechts (mit *>>*) wandern zu lassen. Bezugnehmend auf die Variablen *i* und *j* der Abbildung 4.5 würden sich mit diesen Operatoren folgende Ergebnisse einstellen:

```
i >> 4 = 0000100110011001 = 2457 = 0x999
j << 2 = 0001000111010000 = 4560 = 0x11D0
```

Bei den Verschiebe-Operatoren gibt also der rechts vom Operator stehende Wert an, um wieviele Positionen die Bits nach links bzw. nach rechts wandern sollen. Rechts vom Operator kann natürlich auch eine Variable stehen. In den Beispielen wurde stillschweigend vorausgesetzt, daß durch die Verschiebung freiwerdende Bitpositionen mit Nullbits aufgefüllt werden. Dies ist jedoch nur dann garantiert, wenn es sich bei den Variablen, mit denen die Verschiebeoperatoren arbeiten, um *unsigned int*-Variablen handelt.

### 4.2.4 Logische Operatoren

Die soeben vorgestellten Bitoperatoren arbeiten auf allen Bits einer Variablen zugleich. Sie bedienen sich dabei der (logischen) Verknüpfungen UND, ODER, EXKLUSIVES ODER und NICHT, die in der Abbildung 4.4 definiert sind. Diese Verknüpfungen kann man aber nicht nur für Bitoperationen heranziehen, sondern auch zur Verknüpfung wahrheitswertiger Ausdrücke, um damit beispielsweise komplexe Kontrollbedingungen zu formulieren.

| Operator | Bedeutung |
|---|---|
| ! | Negation |
| && | UND-Verknüpfung |
| \|\| | ODER-Verknüpfung |

**Abb. 4.6:** *Logische Verknüpfungen*

Wenn als Teil einer Problemlösung Formulierungen vorkommen wie etwa "wenn Bedingung A UND Bedingung B vorliegt..." oder "solange das eingelesene Zeichen ein Kleinbuchstabe ODER eine Ziffer ist...", dann werden für die Umsetzung in C-Code logische Operatoren gebraucht. In den Bei-

spielprogrammen haben Sie schon einige kennengelernt; die Abbildung 4.6 zeigt Ihnen alle in C vorhandenen.

Der erste Operator (*!*) ist einstellig; mit ihm wird der Wahrheitswert einer Bedingung umgedreht. Sie brauchen ihn, um Formulierungen wie "wenn das eingelesene Zeichen NICHT ein Großbuchstabe ist" in C umzusetzen. Die beiden anderen Operatoren sind zweistellig und verknüpfen zwei wahrheitswertige Ausdrücke gemäß den in Abbildung 4.4 definierten Regeln.

Wie Sie sehen können, benutzt C für die Bitoperatoren und die logischen Verknüpfungsoperatoren ähnliche Zeichen. Dadurch ist gerade für Anfänger die Gefahr der Verwechslung geben. Die Bitoperatoren werden gebraucht, um Manipulationen an in binärer Form vorliegender Informationen vorzunehmen. Dies ist meist bei sehr maschinennaher Programmierung nötig, etwa dann, wenn man den Zustand eines einzelnen Bits in einer bestimmten Speicherstelle des Computers abfragen oder setzen will. Diese Speicherstellen können Ein-Ausgabeports sein, im Speicher liegende "Register" peripherer Geräte oder das Speicherabbild des Bildschirms. Die logischen Verknüpfungen sind auf einer höheren Ebene angesiedelt; sie dienen der Programmsteuerung, tauchen in *if*s, in *for*- und *while*-Schleifen auf. Da C keinerlei Typüberprüfung macht, ist es jedoch ohne weiteres möglich, anstelle einer logischen Verknüpfung einen Bitoperator zu schreiben. Der Compiler wird Sie auf diesen Lapsus nicht hinweisen (denn eventuell ist das ja auch beabsichtigt und gemäß der Philosophie von C darf der Programmierer alles machen, was er will). Aber die Wahrscheinlichkeit ist sehr groß, daß eine Schleife, deren Kontrollbedingung von einem Bitoperator regiert wird, zu der unbeliebten Gattung der Endlosschleifen zu zählen ist.

Die zweistelligen Verknüpfungsoperatoren weisen noch eine Besonderheit auf, die manchmal in Programmen vorteilhaft genutzt werden kann. Wenn Sie sich die Abbildung 4.4 noch einmal genauer ansehen, werden Sie feststellen, daß bei einer UND-Verknüpfung zweier Wahrheitswerte das Ergebnis bereist feststeht, wenn nur einer der beiden verknüpften Werte "falsch" (gleich 0) ist. In diesem Fall ist es überhaupt nicht mehr nötig, das andere Argument der Verknüpfung zu betrachten: das Ergebnis ist in jedem Fall 0.

Ähnlich bei der ODER-Verknüpfung. Ist eines der Argumente 1, dann spielt der Wert des anderen Arguments keine Rolle mehr. In jedem Fall ergibt sich "wahr" (oder 1).

Diese Beobachtungen nutzen die logischen Operatoren, um sich das Leben zu erleichtern. Die UND-Verknüpfung betrachtet zuerst ihr erstes Argument. Ist dies "falsch", dann bricht sie sofort mit dem Wert "falsch" ab, ohne ihr zweites Argument noch eines Blickes zu würdigen. Nur für den

Fall, daß das erste Argument "wahr" ist, untersucht sie auch das zweite Argument.

Bei ODER läuft es gerade umgekehrt: stellt der Operator fest, daß das erste Argument "wahr" ist, so gibt er, ohne sich um das zweite zu kümmern, einfach "wahr" aus. Die beiden Operatoren machen also keinen Handstrich zuviel.

Das setzt voraus, daß diese Operatoren ihre Argumente immer von links nach rechts untersuchen. Jeder C-Compiler garantiert Ihnen das; allerdings nur für diese beiden Operatoren. Alle anderen zweistelligen Operatoren, die Ihnen in diesem Kapitel vorgestellt werden, können sich ihren Argumenten in der Reihenfolge zuwenden, die ihnen gerade angemessen erscheint. Sie sollten sich also auf keinen Fall auf eine bestimmte Reihenfolge verlassen!

Das Auswertungsverhalten der logischen Verknüpfungen kann auch zur Falle werden. Denn in C ist es möglich, in Vergleichen Zuweisungen unterzubringen; das nächste Programm liefert ein erneutes Beispiel für diese Praxis. Ist eine Zuweisung aber zweites Argument eines logischen Operators, dann wird sie unter bestimmmten Umständen niemals ausgeführt. Hierzu zwei Beispiele:

```
if (0 && ((c = getchar()) != 'A')
.
.
.

if (1 || ((c = getchar()) != 'E')
.
.
.
```

In beiden Fällen unterbleibt die Zuweisung an die Variable *c*, ja, die Funktion *getchar* wird gar nicht erst aufgerufen. Dieser Teil des Programms wird durch das erste Argument der logischen Verknüpfung (im Falle UND der Wert "falsch", im Falle ODER "wahr") gleichsam verdeckt.

Das folgende Beispielprogramm demonstriert noch einmal die Einsatzmöglichkeiten der Verknüpfungs- und Bitoperatoren. Die Aufgabenstellung ist die folgende: eine in einer *unsigned int* gespeicherte Zahl soll in binärer Schreibweise auf den Bildschirm ausgegeben werden (Funktion *bin_dump*). Die zweite Funktion übernimmt es, eine vom Benutzer in binärer Darstellung eingegeben Zahl (also eine Folge von Nullen und Einsen) in eine *unsigned int* umzuwandeln (Funktion *get_bin*).

```
#include <stdio.h>

main()

{

   unsigned int i,j;
   unsigned int get_bin();

do
{  printf("\nBitte Binärzahl 1 eingeben: ");
   i = get_bin();
   printf("\nBitte Binärzahl 2 eingeben: ");
   j = get_bin();
   printf("\nDezimal: i = %u, j = %u",i,j);
   printf("\ni + j = "); bin_dump(i + j);
   printf("\ni & j = "); bin_dump(i & j);
   printf("\ni ^ j = "); bin_dump(i ^ j);
   printf("\i | j = "); bin_dump(i | j);
}
   while ((i != 0) && (j != 0));

}


unsigned get_bin()

{  char c;
   unsigned int i;

   i = 0;

   while (((c = getchar()) == '0') || (c == '1'))
      i = ((i << 1) | (c - '0'));

   return i;
}


bin_dump(i)
unsigned int i;

{  int j;

   for (j = 15; j >= 0; --j)
      if (i & (1 << j))
         putchar('1');
      else
         putchar('0');
}
```

```
/*************************/
/*  Ein- und Ausgabe von  */
/*  Binaerzahlen; Wirkung */
/*  von Bitoperatoren.    */
/*************************/
/* Auch Funktionen muessen */
/*  deklariert werden!    */
/*                        */
/*                        */
/* Erste Binaerzahl lesen. */
/*                        */
/* Zweite Zahl lesen.     */
/* Ausgabezeichen fuer un- */
/*  signed ist %u!        */
/* Addition.              */
/* Bitweises UND.         */
/* EXKLUSIVES ODER.       */
/* Bitweises ODER.        */
/*                        */
/* Schleife wird beendet, */
/*  wenn eine der beiden  */
/*  Zahlen gleich 0.      */
/*************************/

/*************************/
/* Binaerzahl einlesen.  */
/*************************/
/*                        */
/* Die Schleifenbedingung */
/*  liefert ein Beispiel  */
/*  fuer logische ODER-   */
/*  Verknuepfung zweier   */
/*  Bedingungen.          */
/*                        */
/*                        */
/*************************/

/*************************/
/* Unsigned int binaer   */
/*  ausgeben.            */
/*************************/
/*                        */
/* Die Schleife erzeugt   */
/*  sukzessive Masken (im */
/*  "if"), bei denen nur  */
/*  eine Bitposition ge-  */
/*  setzt ist. Dies be-   */
/*  stimmt, ob 0 oder 1   */
/*  ausgegeben wird.      */
/*************************/
```

**Prog. 4.8:** *Ein- und Ausgabe von Binärzahlen*

Die Eingabefunktion *get_bin* liest solange Zeichen von der Tastatur, bis der Benutzer ein anderes als die beiden zugelassenen Ziffernzeichen '0' und

'1' eingibt. Das eingelesene Zeichen wird an die Variable *c* zugewiesen. In bewährt knapper C-Manier sind Einlesen, Zuweisen und Test des Zeichens auf '0' oder '1' allesamt in der Kontrollbedingung der *while*-Schleife untergebracht.

Das Ergebnis wird in der mit 0 vorbesetzten (initialisierten) Variable *i* aufgebaut. Bei jedem Schleifendurchgang wird der numerische Wert der eingelesenen Ziffer (mit *c - '0'*) bestimmt, die bisher bereits aufgebaute Ziffer um eine Position nach links verschoben (*i << 1*) und über bitweises ODER der eben eingelesene Zahlenwert (eine binäre 0 oder 1) in das niedrigstwertige Bit von *i* gesetzt. Der Ergebnis wird erneut in *i* gespeichert.

Zum Ausgeben einer Integer *i* in Binärschreibweise geht man folgendermaßen vor: das Ergebnis wird mit 16 Stellen ausgegeben. Die Funktion *bin_dump* erzeugt nacheinander in einer Schleife 16 Masken, in denen zuerst das höchstwertige Bit (Bit 15), dann Bit 14, dann Bit 13 usw. bis Bit 0 gesetzt ist. Das bewirkt die Anweisung *1 << j*, wobei *j* nacheinander die Werte von 15 bis 0 durchläuft. Wie Sie sehen, kann auch eine Konstante Argument der Verschiebeoperatoren sein. Über bitweise UND-Verknüpfung wird als nächstes getestet, ob das in der Maske gesetzte Bit auch in der auszugebenden Zahl gesetzt ist und entsprechend eine '1' oder '0' ausgegeben. Das Erzeugen der Maske und Testen des Bits ist in die Bedingung der *if*-Anweisung integriert: *(i & (1 << j))*.

Die Hauptfunktion *main* des Programms 4.8 benutzt die beiden Unterfunktionen *get_bin* und *bin_dump*, um Ihnen die Wirkungsweise unterschiedlicher Operatoren auf Bitebene zu verdeutlichen. Dazu kommt eine Variante der *while*-Schleife zum Einsatz, in der die Kontrollbedingung erst am Ende der Schleife abgefragt wird. Diese sogenannte *do*-Schleife wird später in diesem Kapitel noch genauer erläutert.

Interessant ist in diesem Zusammenhang noch die Formulierung der Kontrollbedingung. Mit der vorliegenden Fassung *while ((i != 0) && (j != 0))* wird die Schleife schon abgebrochen, sobald der Benutzer nur für eine der beiden Variablen den Wert 0 eingibt. Es werden zwei Tests auf Ungleichheit über UND verknüpft. Möchte ich erreichen, daß die Schleife nur dann verlassen wird, wenn beide Variablen den Wert 0 haben, dann ist das so zu formulieren: *while (!(i == 0 && j == 0))*. Hier werden zwei Gleichheitstests mit UND verknüpft und das Verknüpfungsergebnis negiert.

Noch zwei Neuigkeiten können Sie dem Hauptprogramm *main* entnehmen: hat eine Funktion einen anderen Datentyp als *int* zum Wert, dann muß sie deklariert werden. In diesem Fall wird in *main* bekanntgemacht, daß *get_bin* eine vorzeichenlose Integer zurückgibt. Die zweite Neuigkeit betrifft die Formatsteuerung für *printf* bei der Ausgabe von vorzeichenlosen

Integers. Sie können in diesem Fall nicht mit *%d* arbeiten, sondern müssen mit *%u* signalisieren, daß das zugehörige Argument als positive Zahl zu interpretieren ist.

### 4.2.5 Zuweisungsoperatoren

Die meisten Programmiersprachen kennen nur ein Zeichen für Zuweisung; C bietet deren Elf! In den meisten Sprachen verändert die Zuweisung den Wert einer Variablen - und sonst nichts. In C sind alle elf Varianten der Zuweisung Operatoren, produzieren also einen weiterverwendbaren Wert; die Variable ändern sie sowieso! Die Zuweisungsoperatoren und ihre Eigenschaften tragen viel zu dem berühmt-berüchtigten exotischen Aussehen von C-Programmen bei. Positiv gesagt: sie erlauben Programme von hoher Integrationsdichte. Die Abbildung 4.7 versammelt alle Vertreter dieser Gattung.

| Operator | Bedeutung |
|---|---|
| = | Einfache Zuweisung |
| += | Zuweisung mit gleichzeitiger Addition |
| -= | Zuweisung mit gleichzeitiger Subtraktion |
| *= | Zuweisung mit gleichzeitiger Multiplikation |
| /= | Zuweisung mit gleichzeitiger Division |
| %= | Zuweisung mit gleichzeitiger Restbildung |
| >>= | Zuweisung mit bitweisem Verschieben nach rechts |
| <<= | Zuweisung mit bitweisem Verschieben nach links |
| &= | Zuweisung mit bitweiser UND-Verknüpfung |
| ^= | Zuweisung mit bitweiser EXKLUSIV ODER-Verknüpfung |
| \|= | Zuweisung mit bitweiser ODER-Verknüpfung |

**Abb. 4.7:** *Die Zuweisungsoperatoren*

Warum so viele? Eine Analyse herkömmlicher Programmiersprachen hat erbracht, daß sehr oft Zuweisungen auftauchen, an denen nur eine einzige Variable beteiligt ist, deren Wert geändert werden soll. Oft liest man also in BASIC

```
a = a / 2
x = x * 4
b = b + 2
```

Dafür hat der Erfinder von C eine knappere Schreibweise eingeführt. Aber nicht nur Schreibfaulheit war ein Motiv für diese Regelung. Denn in herkömmlicher Weise geschriebene Zuweisungen dieser Art werden von Compilern meist nicht optimal in Maschinencode übersetzt. Der Compiler erkennt nicht, daß zweimal dieselbe Variable beteiligt ist, er wird bei jedem

Auftauchen dieser Variablen in der Zuweisung von neuem ihren Wert bestimmen. Die in C gebräuchliche Notation stellt hingegen einen deutlichen Fingerzeig für den Compiler dar. Programme werden dadurch effizienter.

```
#include <stdio.h>

main()
{
    char c, tolower();

    while ((c = getchar()) != 'E')
        putchar(tolower(c));
}

char toupper(zei)
     char zei;

{
    if (islower(zei))
        return (zei + 'A' - 'a');
    else
        return(zei);
}

int islower(c)
     char c;

{
    return (c >= 'a' && c <= 'A');
}


char tolower(c)
     char c;

{
    return(isupper(c) ? c + 'a' - 'A' : c);
}


int isupper(c)
     char c;

{
    return (c >= 'A' && c <= 'a');
}
```

```
/***************************/
/*  Eingelesene Zeichen in */
/*  Kleinbuchstaben umwan- */
/*  deln, bis Benutzer 'E' */
/*  eingibt.               */
/*                         */
/***************************/

/***************************/
/*  Parameter in Gross-    */
/*  buchstaben wandeln     */
/***************************/
/* Falls Kleinbuchstabe:   */
/*    umwandeln            */
/* ansonsten:              */
/*    unveraendert zurueck */
/***************************/

/***************************/
/*  Ist der Parameter      */
/*  kleingeschrieben?      */
/***************************/
/*  Bereich ueberpruefen.  */
/***************************/

/***************************/
/* Zeichen in Kleinschrei- */
/*  bung umwandeln.        */
/***************************/
/* Auswahl ueber den drei- */
/*  stelligen Operator     */
/*  "? :"!                 */
/***************************/

/***************************/
/* Grossbuchstabe?         */
/***************************/
/*                         */
/* Bereich ueberpruefen.   */
/***************************/
```

**Prog. 4.9:** *Umwandlung in Kleinbuchstaben:* tolower

### 4.2.6 Der Vergleichsoperator

Ein- und zweistellige Operatoren sind in Programmiersprachen eine Alltäglichkeit. Aber C ist nicht wie andere Sprachen; es kennt auch einen dreistelligen Operator! Dieser Operator bietet eine Kurzschreibweise für

*if...else* in bestimmten Situationen. Am klarsten wird seine Verwendung an einem Beispiel, das Sie in Programm 4.9 finden. Es geht um die Funktion *tolower*, die Großbuchstaben in Kleinbuchstaben umwandeln soll. Da die Aufgabenstellung dieser Funktion sehr ähnlich zu der von *toupper* ist, das in Programm 3.8 vorgestellt wurde, ist diese Funktion mit ihrer Hilfsfunktion *islower* noch einmal mit aufgeführt.

Was in der Funktion *toupper* mit *if..else* erledigt wurde und vier Zeilen beanspruchte, das macht *tolower* in einer einzigen Zeile, allerdings mit einer geheimnisvoll anmutenden Schreibweise. Ein Ausdruck der in der Abbildung 4.8 gezeigten Form ist wie folgt zu lesen: wenn die <bedingung> erfüllt ist, dann hat das Ganze <ausdruck1> als Wert, ansonsten liefert es eben <ausdruck2>. All dies bewirkt ein dreistelliger Operator, der einen wahrheitswertigen Ausdruck und zwei beliebige andere Ausdrücke als Argument hat und je nach Lage des wahrheitswertigen Ausdrucks einen der beiden anderen auswählt. Zum Hinschreiben des Operators benötigt man zwei Zeichen: ein Fragezeichen *?*, vor das die Bedingung und hinter das der erste Ausdruck geschrieben werden muß, sowie einen Doppelpunkt *:*, der den ersten und zweiten Ausdruck trennt.


**Abb. 4.8:** *Anatomie des Vergleichsoperators*

Wenn in einem Programm einer von zwei Werten für die weitere Verarbeitung ausgewählt werden soll, dann kann man dies mit dem bereits bekannten *if...else* tun. Knapper (und effizienter) ist aber oftmals die Formulierung mit dem Vergleichsoperator. In der Beispielsfunktion *tolower* wählt er aus, was zurückgegeben werden soll: entweder das unveränderte Zeichen, wenn dies bereits kleingeschrieben ist (Test mit *islower*), oder ein nach den üblichen Methoden durch Addition eines Versatzwertes (der Zahlenbetrag, der Großbuchstaben von Kleinbuchstaben in der ASCII-Tabelle trennt) umgewandeltes Zeichen.

## 4.3 Vorrang, Assoziativität und Typumwandlung

Häuften sich in einem Programm die Operatoren, dann habe ich bisher durch den freizügigen Einsatz von Klammern versucht, klare Verhältnisse zu schaffen. Doch allzuviele Klammern erhöhen nicht unbedingt die Leserlichkeit des Programmes. Alle (konventionellen) Programmiersprachen bie-

ten deshalb die Möglichkeit, in solchen Fällen Klammern einzusparen; C ist da keine Ausnahme.

| Operator | | Assoziativität |
|---|---|---|
| ( ) | | links-rechts |
| [ ] | | |
| . | | |
| -> | | |
| ! | | rechts-links |
| ~ | | |
| ++ -- | | |
| * & | (bei Pointern und Adressen) | |
| (<typ>) | (Cast-Operator) | |
| sizeof | | |
| * / | % | links-rechts |
| + - | | links-rechts |
| >> << | | links-rechts |
| >= > | <= < | links-rechts |
| == != | | links-rechts |
| & | | links-rechts |
| ^ | | links-rechts |
| \| | | links-rechts |
| && | | links-rechts |
| \|\| | | links-rechts |
| ?: | | rechts-links |
| = += | -= *= /= | rechts-links |
| %= >>= | <<= &= = \|= | |

**Abb. 4.9:** *Vorrang und Assoziativität der C-Operatoren*

Der Trick, mit dem sich Klammern einsparen lassen, ist Ihnen noch aus der Schule vertraut; er hat mit dem Vorrang von Operatoren zu tun. Wir alle lernten im Mathematikunterricht das Sprüchlein "Punkt vor Strich" und wissen seitdem, daß in dem Ausdruck *5 + 2 * 3 - 7* die Multiplikation vor

der Addition und Subtraktion kommt und sich deshalb das Ergebnis 4 einstellt. Anders bei *(5 + 2) * 3 - 7*, denn Klammern werfen alle Vorrangregeln über den Haufen. Hier kommt 14 als Ergebnis 'raus.

Die Regel "Punkt vor Strich" bringt zum Ausdruck, daß Multiplikation und Division Vorrang vor Addition und Subtraktion haben, also in einem zusammengesetzten Ausdruck vor diesen ausgeführt werden müssen. Die Regel unterscheidet - anders ausgedrückt - zwei Vorrangstufen und ordnet die Multiplikation in die höhere, die Addition in die niedrigere der beiden Stufen ein. Die Vorrangregeln von C sind etwas komplizierter, da es hier mehr Operatoren gibt. Insgesamt werden in C 14 Vorrangstufen unterschieden, die in der Tabelle der Abbildung 4.9 dargestellt sind. Je weiter oben ein Eintrag in dieser Tabelle zu finden ist, desto größer ist sein Vorrang.

Nicht alle Operatoren dieser Tabelle wurde Ihnen bereits vorgestellt; für die fehlenden müssen erst noch die nötigen Grundlagen gelegt werden. Da jedoch die runden Klammern 'Vorfahrt' vor jeder anderen Vorrangregel bekommen müssen, finden sie sich ganz oben in der Tabelle. Was da noch so alles steht, wird Ihnen in den nächsten Kapiteln vorgestellt.

Die nächste Etage der Tabelle wird von den einstelligen Operatoren bewohnt. Wie Sie sehen, führen in C die Zeichen * und & ein Doppelleben, da sie auch als einstellige Operatoren vorkommen. Was sie da bewirken, erfahren Sie ebenfalls erst in einem der nächsten Kapitel.

Die Einordnung der zweistelligen Operatoren macht keine Schwierigkeiten ("Punkt vor Strich"), allerdings ist bei den Operatoren zur Bitverknüpfung Vorsicht geboten. Ihre Vorrangstellung ist sehr verwirrend und bei ihnen empfiehlt es sich, stets Klammern zu verwenden. Wollen Sie z.B. testen, ob in einer Variablen *i* von den zwei niedrigstwertigen Bits nur das zweite gesetzt ist, dann könnten Sie versucht sein, dies als

```
i & 3 == 2
```

zu schreiben. Aber das ist falsch, da der Vergleich Vorrang vor dem Bitoperator hat: es würde 0 (das Ergebnis des Vergleichs) mit *i* über UND-Verknüpft und alle Bits wären im Ergebnis gelöscht. Sie müssen stattdessen schreiben:

```
(i & 3) == 2
```

Der erste Eindruck von dieser Vorrangtabelle mag verwirrend sein; sie wurde jedoch entwickelt, um bei klammerfreien Ausdrücken der "natürlichen" Lesart den Vorzug zu geben. Der Ausdruck

```
a + b < c && d - e >= c
```

hat die "natürliche" Lesart: "a plus b ist kleiner als c und d minus e ist größer oder gleich c". Wird Ihnen diese Anweisung z.B. vorgelesen oder über Telefon durchgegeben, dann werden Sie sie ziemlich sicher so auffassen: die beiden Vergleiche sollen vor der UND-Verknüpfung durchgeführt werden; vor den Vergleichen soll jedoch noch addiert bzw. subtrahiert werden - und genauso wird er, dank der Vorrangregeln, vom C-Compiler aufgefaßt! Man sagt auch, daß die Vorrangregeln so gewählt wurden, um komplexe Ausdrücke den "Telefon-Test" bestehen zu lassen.

Im einzelnen liegen der natürlichen Lesart folgende Überlegungen zugrunde:

- die einstelligen Operatoren kommen vor den zweistelligen. Dies ist auch in der Schulmathematik so, wo es allerdings nur das einstellige Minus (für die Kennzeichnung negativer Zahlen) gibt.

- wegen der natürlichen Lesart wird den arithmetischen Operatoren Vorrang vor den Vergleichsoperatoren gegeben (siehe das obige Beispiel). Wie in der Schule, erhalten auch in C innerhalb der arithmetischen Operatoren die sogenannten multiplikativen Operatoren *, / und % Vorrang vor den additiven + und -.

- die Vergleichsoperatoren kommen - wieder wegen der Lesbarkeit - vor den Verknüpfungsoperatoren. Auch dies wird durch das letzte Beispiel gerechtfertigt. Daß die UND-Verknüpfung Vorrang vor der ODER-Verknüpfung hat, ist allerdings höchstens für den geschulten Logiker 'natürlich'. Deshalb: Verknüpfungsketten ("a UND b ODER c UND d") vorsichtshalber klammern!

- der dreistellige Entscheidungsoperator kommt vor der Zuweisung. denn oft hängt von seinem Ergebnis der Wert der Zuweisung ab. In der Zeile

```
i += (a > 3 ? 2 : -1)
```

  wird *i* um 2 erhöht, wenn *a* größer 2 ist, ansonsten aber um 1 erniedrigt.

- Die Zuweisung kommt ganz zuletzt. Erst nachdem alle Operatoren angewendet und deren Ergebnisse bestimmt sind, kann das Resultat zugewiesen werden. Zuweisungen, die sich in den Argumenten von Operatoren verbergen, sind deshalb entsprechend zu klammern. Wer glaubt, die folgendermaßen formulierte Schleifenbedingung

```
while (c = getchar() != 'E')
```

würde ihm in der Variablen *c* das eingelesene Zeichen liefern, der irrt: *c* enthält stets einen Wahrheitswert, "wahr" oder "falsch", je nachdem, ob das eingelesene Zeichen ein 'E' war oder nicht. Möchte man die Benutzereingabe in *c* zwischenspeichern, dann muß man klammern:

```
while ((c = getchar()) != 'E')
```

Die Vorrangstufen regeln also die Verhältnisse in klammerfreien Ausdrücken - beinahe! Denn was ist mit einem Ausdruck der Form:

```
a - b - c
```

Soll erst *b* von *a* subtrahiert werden und dann davon *c*? Ginge es nur um die Arithmetik, dann wäre das gleichgültig. Aber in den Argumenten von Operatoren können sich Zuweisungen verbergen. Betrachten Sie das nächste Beispiel:

```
b = 2;
c = 3;
(b = 1) - c - b;
```

Wenn nun die Subtraktionen von links nach rechts erfolgen, dann hat der letzte Ausdruck den Wert -3. Geht C jedoch umgekehrt vor, dann ergibt sich 0. Was macht man, wenn mehrere Operatoren gleichen Vorrangs aneinandergereiht sind? Man befolgt die Regeln der Assoziativität. Auch diese können Sie der Abbildung 4.9 entnehmen. "links-rechts"-Assoziativität bedeutet, daß gleichrangige Operatoren von links nach rechts ausgewertet werden. Für die arithmetischen Operatoren ist dies die natürliche Reihenfolge. Nicht jedoch für die Zuweisung. Bei Mehrfachzuweisungen wie etwa

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

möchte man, daß zuerst die Zuweisung an *c*, dann an *b* und dann an *a* erfolgt. Darum haben die Zuweisungsoperatoren "rechts-links"-Assoziativität. Daß auch der Wenn-dann-Operator in dieser Richtung assoziiert, ist nachrangig, da in guten (also verständlichen) C-Programmen die Schachtelung dieses Operators vermieden wird.

Das Problem der Typumwandlung betrifft Operationen, bei denen die Operanden von unterschiedlichem Datentyp sind. Bekanntermaßen ist es möglich, in C folgendes zu schreiben:

```
char c;
c = 32;
c += 4;
```

Durch die Zuweisung von 32 an *c* enthält diese Zeichenvariable ein Leerzeichen (vgl. ASCII-Tabelle). Wird *c* um 4 hochgezählt, dann stellt es das Zeichen '$' dar (Position 36 in der Tabelle). Aber, wohlgemerkt: weder 32

noch 4 sind Zeichenkonstanten (diese müssen ja bekanntlich in einfache Anführungszeichen eingeschlossen sein). Hier haben wir es also mit zwei Operationen zu tun (beide Male Zuweisungen), die mit Argumenten unterschiedlichen Typs (*char* und *int*) arbeiten.

Solche gemischte Operanden kommen auch im Zusammenhang mit der Arithmetik häufiger vor. Ein Beispiel:

```
float f;
int i;
long l;
i = 30000;
l = 3 * i;
f = i + l;
```

In der letzten Zeile sind drei Datentypen beteiligt: eine *int* und *long*-Variable werden addiert. Das Ergebnis wird einer *float*-Variablen zugewiesen. Was passiert in einem solchen Fall?

Daten unterschiedlichen Typs werden in unterschiedlichen Formaten gespeichert und sind erstmal nicht miteinander verträglich. Bei gemischten Operationen ist es daher möglich, zeitweilige Typumwandlungen vorzunehmen. Bei dieser Typumwandlung hat der ATARI seine eigenen Präferenzen, die mit der Struktur des verwendeten Prozessors, des Motorola 68000 zusammenhängen.

Dieser Prozessor arbeitet intern mit einer Datenbreite von 32 Bit, verkehrt jedoch mit der Außenwelt (und seinem Speicher) in Größen von 16 Bit (da der Datenbus nur über 16 Leitungen verfügt).

Daraus ergibt sich, daß der Prozessor eine natürliche Vorliebe für 32-Bit-Größen hat. Auch 16 Bit sind noch angenehm; am unteren Ende der Beliebtheits-Skala finden sich die 8-Bit-Zeichen. Damit haben wir die Typenhierarchie, die Sie in der Abbildung 4.10 sehen können. Die 'bevorzugten' Datentypen sind mit einem Pfeil markiert.

```
--> double
    float
    (--> unsigned long)
--> long
--> unsigned int
--> int
    (unsigned char)
    char
```

**Abb. 4.10:** *Typenhierarchie des ATARI-C*

Diese Tabelle spiegelt die Verhältnisse wieder, wie sie im C des Entwicklungssystems herrschen. Sollten Sie mit einem anderen C-Compiler arbeiten,

dann müssen Sie eventuell mit einer leicht geänderten Typenhierarchie rechnen. Denn zwischen *float* und *unsigned int* kennt C üblicherweise noch einen Datentyp, die *unsigned long*-Werte. Dieser Typ ist im ATARI-C - ebenso wie der Typ *unsigned char* - nicht implementiert (nicht vorhanden) und deshalb in der Tabelle eingeklammert.

Bei der Typumwandlung gelten folgende Regeln. Für die Verarbeitung wird jeder Wert eines nicht-bevorzugten Datentyps erstmal in den nächsthöheren Datentyp umgewandelt. Das bedeutet: wenn Sie eine *char*-Variable in einem Programm haben, dann benötigt diese Variable zwar im Arbeitsspeicher nur ein Byte Platz. Aber sobald sie mit dieser Variablen arbeiten, ihr irgendetwas antun wollen, dann wird die Variable aus dem Speicher in den Prozessor (genauer: in dessen Register) gebracht und dort zu einer 16 Bit langen Integer 'aufgeblasen'. Dann tut C damit (mit dieser Integer, die gleichsam das ursprüngliche Zeichen vertritt), was immer auch zu tun ist. Soll das Ergebnis wieder in den Speicher wandeln, dann wird es erst auf 8 Bit zurechtgestutzt und dann abgelegt.

Dadurch ist es in C ohne Probleme möglich, auf Zeichen Integerkonstanten zu addieren; denn bei der Verarbeitung hat man es durch die Typumwandlung des Zeichens ohnehin mit gleichen Datenformaten zu tun.

Sind an einer Operation zwei unterschiedliche Operanden beteiligt, dann wird der höchstrangige (falls nötig) erstmal in den nächsthöheren bevorzugten Typ umgewandelt. Der niedrigstrangige wird dann ebenfalls in ebendiesen Typ gezwungen. Sind also an einer Operation eine Zeichenvariable, eine Integer und eine Float beteiligt, dann läuft die gesamte Operation (nach Typenhierarchie) im *double*-Format ab.

Bei Zuweisungen gilt die Regelung, daß das Ergebnis rechts vom Zuweisungszeichen in den Datentyp der Variablen links vom Zuweisungszeichen umgewandelt wird.

Für das C des ATARI-Entwicklungssystems sind noch einige Anmerkungen erforderlich. Zum Zeitpunkt der Erstellung dieses Buches war im C-Compiler von Digital Research der Datentyp *double* noch nicht implementiert, so daß der "höchste" verfügbare Datentyp *float* ist. Ein weiteres Problem betrifft die Zeichenvariablen. Zwar sieht C einen Type *unsigned char* vor, aber dieser ist ebenfalls nicht implementiert. Versucht man, ein Grafikzeichen (mit einem Zeichencode größer 127) für numerische Operationen heranzuziehen, dann stellt man fest, daß der Compiler dies als negative Zahl auffaßt. Bei der Umwandlung von *char* in *int* wird das gesetzte achte Bit der Zeichenvariable als Vorzeichenbit interpretiert und die Integer wird negativ. Da hilft es auch nichts, wenn man versucht, in eine *unsigned int* umzuwandeln: dann sieht C die Variable einfach als eine große Zahl im

Bereich über 65000. Probieren Sie doch einmal das folgende Programm aus; dazu muß gesagt werden, daß man in C die Umwandlung einer Variablen in einen andern Typ erzwingen kann, indem man den gewünschten Typ in Klammern vor die Variable schreibt. Der eingeklammerte Datentyp zählt als Operator; er trägt den Namen "Cast".

```
main()

{  char c;
   int i;

   for (c = i = 125; i <= 180; ++i, ++c)
   {  printf("\nc: char %c, int %d, unsigned %u, long %ld",
              c, (int) c, (unsigned) c, (long) c);
      printf("\ni: char %c, int %d, unsigned %u, long %ld",
             (char) i, i, (unsigned) i, (long) i);
   }
}
```

**Prog. 4.10:** *Probleme mit vorzeichenbehafteten Zeichen.*

Wie das Programm zeigt, ist die umgekehrte Richtung unproblematisch: die Umwandlung eines Integer-Werts in eine Zahl geht wie gewünscht vonstatten. Sie können dem Programm auch noch einige weitere Format-Zeichen für die Ausgabe mit *printf* entnehmen: *%c* für Zeichen-Ausgabe, *%u* für Ausgabe von vorzeichenlosen Integers und *%ld* für die Ausgabe von *long*-Variablen.

Noch einige Anmerkungen zum Cast-Operator. Er wird meist dann eingesetzt, wenn man eine Variable als Argument an eine Funktion übergeben will, die Daten von einem anderen Typ erwartet. Dies soll das folgende Programmfragment illustrieren:

```
int funkt1(i)
long i;
.
.

int funkt2()
{   char c;

    funkt1((long)c);
    .
    .
}
```

Der Parameter von *funkt1* ist als *long* deklariert. In *funkt2* soll dieser Funktion aber eine Zeichenvariable übergeben werden. Dies würde normalerweise zu Problemen führen, wenn nicht durch den Cast-Operator *(long)* zuerst das Zeichen in den richtigen Datentyp umgewandelt wird. Sie sehen also: eine Typumwandlung erfolgt in C nur dann automatisch, wenn man

Operatoren mit gemischten Datentypen verwendet. Passen die Typen der Argumente einer Funktion nicht mit den Parameter-Deklarationen zusammen (wie im obigen Fall), dann müssen Sie selbst mit dem Cast-Operator für die geeigneten Umwandlungen sorgen.

## 4.4 Kontrollstrukturen

Ein C-Programm (oder eine C-Funktion) besteht aus einer Folge von Deklarationen und Anweisungen. Deklarationen versorgen den Compiler mit Informationen über die im Programm verwendeten Datenelemente (die Variablen, aber auch benötigte Funktionen). Anweisungen geben an, was mit diesen Daten zu geschehen hat. Zwar können Deklarationen und Anweisungen in C-Programmen gemischt werden. Es hat sich jedoch wegen der besseren Lesbarkeit eingebürgert, alle Deklarationen an den Anfang des Programms zu stellen und diesen die ausführbaren Anweisungen folgen zu lassen.

Eine normale ausführbare Anweisung sieht so aus:

```
<ausdruck>;
```

Sie muß also mit einem Strichpunkt abgeschlossen sein. Bei dem Ausdruck handelt es sich meist um eine Zuweisung oder einen Funktionsaufruf.

Normalerweise werden die Ausführungen in einem Programm hintereinander ausgeführt, in der Reihenfolge, in der Sie sie hingeschrieben haben. In zwei wichtigen Fällen wünscht man jedoch, daß von dieser normalen Reihenfolge der Ausführung abgewichen wird: wenn Programmteile in Abhängigkeit von einer Bedingung ausgeführt oder übersprungen werden sollen und wenn Programmteile in einer Schleife wiederholt werden sollen.

Sprachelemente, mit denen die Reihenfolge der Programmausführung beeinflußt werden kann, faßt man unter den Sammelbegriff "Kontrollstrukturen". Viele Kontrollstrukturen, wie etwas die bereits bekannten *for*- und *while*-Schleifen, regeln die Auswertung von genau einer Anweisung. Möchte man eine Folge von Anweisungen in eine *if*-Schleife packen, dann muß man aus dieser Anweisungsfolge eine einzige Anweisung machen, eine sogenannte "zusammengesetzte Anweisung". Eine zusammengesetzte Anweisung sieht so aus:

```
{   <anweisung1>
    <anweisung2>
    ...
    <anweisungn>
}
```

Sie wird also mit geschweiften Klammern umschlossen und es ist nicht nötig, sie mit einem Strichpunkt zu versehen.

Schließlich ist in manchen Fällen noch eine degenerierte Form der Anweisung nötig, die sogenannte "leere Anweisung". Ihre Gestalt:

```
;
```

Die leere Anweisung besteht also aus einem einzelnen Strichpunkt. Obwohl es sich dabei um ein Konstrukt von schlagender Sinnlosigkeit zu handeln scheint: sowas kommt in C-Programmen öfter vor, als man annehmen würde! Jetzt aber zu den ominösen Kontrollstrukturen!

### 4.4.1 Entscheidungen: if und if...else

Zum Formulieren von Entscheidungen hat man in C zwei Möglichkeiten;

```
if (<ausdruck>) <anweisung1>          oder
if (<ausdruck>) <anweisung1> else <anweisung2>
```

Der in runden Klammern stehende Ausdruck wird ausgewertet und sollte einen Wahrheitswert liefern (kein Problem in C, in dem ohnehin alles eine Zahl ist). Liefert er den Wert "wahr" (also eine Zahl ungleich 0), dann wird die <anweisung1> ausgeführt. Darin gleichen sich beide Varianten. Die Unterschiede kommen ins Spiel, wenn der <ausdruck> in Klammern falsch ist. Im ersten Fall fährt C dann mit der Anweisung hinter dem *if* fort, im zweiten Fall wird jedoch <anweisung2> ausgeführt. Dazu in Abbildung 4.11 ein kleines Beispielprogramm.

```
#define WAHR 1
#define FALSCH 0

main()

{   int i;

    for (i = FALSCH; i <= WAHR; ++i)
    {  printf("\n\ni ist %c", (i ? 'W' : 'F'));
    if (i)
        printf("\n<anweisung1> im if");
    printf("\n<anweisung> nach dem if");
    if (i)
        printf("\n<anweisung1> im if...else");
    else
        printf("\n<anweisung2> im if...else");
        printf("\n<anweisung> nach dem if...else");
    }
}
```

**Prog. 4.11:** *Varianten von* if

Beachten Sie, wie zu Beginn der *if*-Schleife das passende Zeichen für die Ausgabe mittels *printf* durch den "wenn-dann"-Operator ausgewählt wird. Das Programm erzeugt folgende Ausgabe:

```
i ist F
<anweisung> nach dem if
<anweisung2> im if...else
<anweisung> nach dem if...else
i ist W
<anweisung1> im if
<anweisung> nach dem if
<anweisung1> im if...else
<anweisung> nach dem if...else
```

Wie sie sehen können, werden die Anweisungen hinter den beiden *if*s in jedem Fall ausgeführt. Die im 'Machtbereich' von *if* stehenden Anweisungen kommen im Unterschied dazu nur bedingt zur Ausführung.

Nun ist die *if*-Anweisung - wie der Name schon sagt - selbst eine Anweisung. Daher hindert den Programmierer nichts daran, mehrere *if*s oder *if...else*-Anweisungen ineinanderzuschachteln. Um dabei nicht den Überblick zu verlieren, sollten Sie sich angewöhnen, durch Bildung von Anweisungsblocks den Geltungsbereich der einzelnen Komponenten zu kennzeichnen.

Dies macht das nächste Beispielprogramm klar, welches ein von der Tastatur eingelesenes Zeichen daraufhin untersucht, ob es sich um eine Ziffer, einen Buchstaben oder ein Sonderzeichen handelt. Bei Buchstaben unterscheidet es fernerhin zwischen Groß- und Kleinschreibung und zwischen Konsonanten und Vokalen. Das ganze geht solange, bis Sie einen Punkt eingeben.

Die allgemeine Logik des Programmes läßt sich also folgendermaßen in Pseudocode fassen:

```
while ((c = getchar()) != '.')
{   wenn (c ist eine Ziffer)
        melde "Ziffer";
    ansonsten wenn (c ist ein Kleinbuchstabe)
    {   melde "Kleinbuchstabe";
        untersuche: Vokal oder Konsonant?
    }
    ansonsten wenn (c ist ein Grossbuchstabe)
    {   melde "Grossbuchstabe";
        untersuche: Vokal oder Konsonant?
    }
    ansonsten melde "Sonderzeichen";
}
```

Die Untersuchung, ob man es mit einem Vokal oder Konsonant zu tun hat, kann man so ausformulieren:

```
wenn (c ist Vokal)
    melde "Vokal";
ansonsten
    melde "Konsonant";
```

Damit erhält man das Programm 4.12. Im Kommentar zu diesem Programm habe ich noch einmal versucht, Ihnen die Verschachtelung der einzelnen *if*s und *else*s mit einem Diagramm zu veranschaulichen. (Anmerkung: unter der mir verfügbaren Version des ATARI-C-Compilers konnte das Programm - obwohl es korrekt ist - nicht übersetzt werden. Vermutlich sind's dem Compiler, genau wie mir, zu viele Verschachtelungen!)

```
#include <stdio.h>
#define isvokal(z) (z == 'A' || z == 'E' || z == 'I' || z == 'O' || z == 'U')
#define isupper(z) (z >= 'A' && z <= 'Z')
#define islower(z) (z >= 'a' && z <= 'z')
#define isdigit(z) (z <= '0' && z <= '9')
#define toupper(z) (islower(z) ? z + 'A' - 'a' : z)
#define tolower(z) (isupper(z) ? z + 'a' - 'A' : z)


main()                                          /***************************/
                                                /* Zeichen analysieren.    */
{  char c;                                      /***************************/
                                                /*                         */
   while ((c = getchar()) != '.')               /*  Einlesen bis Punkt.    */
   {                                            /*                         */
      if (isdigit(c))                           /*  if 1 ------------|     */
         printf("\nZiffer!");                   /*                   |     */
      else if (isupper(c))                      /*  else 1 ----------|     */
      {  printf("\nGrossgeschriebener ");       /*  if 2 ------------|     */
         if (isvokal(c))                        /*     if -------|   |     */
            printf("Vokal!");                   /*               |   |     */
         else                                   /*     else -----|   |     */
            printf("Konsonant!");               /*                   |     */
      }                                         /*  else 2 ----------|     */
      else if (islower(c))                      /*  if 3 ------------|     */
      {   printf("\nKleingeschriebener ");      /*                   |     */
            if  (isvokal(toupper(c)))           /*     if -----|     |     */
              printf("Vokal!");                 /*             |     |     */
            else                                /*     else ---|     |     */
              printf("Konsonant!");             /*                   |     */
      }                                         /*                   |     */
      else printf("\nSonderzeichen!");          /*  else 3-----------|     */
   }                                            /*                         */
}                                               /***************************/
```

**Prog. 4.12:** *Analyse von Eingabezeichen*

Bei der Zuordnung von *else* zu *if* verfolgt C die Strategie, ein *else* dem letzten *else*-losen *if* im Programm zuzuordnen. Sie sollten sich auf die

Gruppierungsversuche von C allerdings nicht verlassen und stattdessen selbst durch Anweisungsblocks klare Verhältnisse schaffen.

Das Programm 4.12 hält noch eine weitere Besonderheit bereit, die nichts mit *if* zu tun hat. Es benötigt einige Funktionen, die zum Teil bereits bekannt sind, wie etwa *islower*, *isupper*, *toupper*. Aber diese Funktionen werden diesmal als Makros definiert! Denn der Makro-Präprozessor kann nicht nur einfache Textersetzung vornehmen; er kann bei dieser Ersetzung auch mit Parametern arbeiten. Wenn im Programm 4.12 die Zeile

```
if (isupper(c))
```

vorkommt, dann macht der Präprozessor daraus folgendes:

```
if ((c >= 'A' && c <= 'Z'))
```

Er ersetzt also den Makro *isupper* durch den Text, der ihn definiert, tauscht dort aber überall den Platzhalter *z* aus der Makrodefinition gegen die tatsächlich im Programm verwendete Variable *c* aus. Daß er dies tun muß, weiß der Präprozessor, weil der Makro *toupper* ein Paar runder Klammern aufweist, in denen der Platzhalter steht.

### 4.4.2 Mehrfachauswahl mit switch und case.

Sie wollen ein Programm schreiben, das Zeichen von der Tastatur liest, für jeden Vokal zählt, wie oft er vorkommt, ebenso die Anzahl der gelesenen Leerzeichen, der Satzzeichen und die Anzahl der Konsonanten vermerkt. Sie könnten dieses Programm mit *if...else*-Schachtelungen in der Manier des letzten Beispiels versuchen. Viel übersichtlicher und eleganter ist jedoch die Lösung, die Sie im Programm 4.13 sehen. Sie beenden dieses Programm durch Eingabe eines Kreuzzeichens "#".

Versuchen Sie es einmal mit diesem Satz, und Sie erhalten als Ergebnis:

```
Vokale: 24      Konsonanten 37
Leerzeichen: 11  Satzzeichen: 2
```

Ein Gespann von zwei Anweisungen: *switch* und *case* - ermöglicht hier eine sehr elegante Formulierung. Die *switch*-Anweisung berechnet den Ausdruck in Klammern, der einen Integer-Wert produzieren sollte (aber in C sind ja auch Zeichen Zahlen!). Deren Wert wird nun mit allen Fällen im *case* verglichen. Paßt eine der *case*-Konstanten, dann werden die zugehörigen Anweisungen ausgeführt. Hier weist C eine kleine Inkonsistenz auf, da es hier nicht nötig ist, die Anweisungen zu einem Block zusammenzufassen. Es können beliebig viele Anweisungen beim zugehörigen *case* stehen. Um die Ausführung dieser Anweisungen zu unterbrechen, ist es allerdings notwendig, diese mittels *break* zu verlassen.

```
#define islower(z) (z >= 'a' && z <= 'z')
#define toupper(z) (islower(z) ? z + 'A' - 'a' : z)
                                                /*************************/
main()                                          /*  Zeichentypen zaehlen.  */
                                                /*************************/
{  char c;                                      /*  Eingelesenes Zeichen.  */
   int v,                                       /*  Vokale.                */
       k,                                       /*  Konsonanten.           */
       l,                                       /*  Leerzeichen.           */
       s;                                       /*  Satzzeichen.           */
                                                /*                         */
   v = k = l = s = 0;                           /*                         */
                                                /*                         */
   while ((c = getchar()) != '#')               /*  Je nach Zeichen ver-   */
   {  switch (toupper(c))                       /*  zweigen:               */
      {  case 'A':                              /*                         */
         case 'E':                              /*  Die Vokale...          */
         case 'I':                              /*                         */
         case 'O':                              /*                         */
         case 'U': ++v;                         /*                         */
                   break;                       /*                         */
                                                /*                         */
         case ' ': ++l;                         /*  das Leerzeichen...     */
                   break;                       /*                         */
                                                /*                         */
         case ',':                              /*  Die Satzzeichen...     */
         case '.':                              /*                         */
         case ';':                              /*                         */
         case ':':                              /*                         */
         case '!':                              /*                         */
         case '?':  ++s;                        /*                         */
                                                /*  Alles andere zaehlt    */
         default:   ++k;                        /*   als Konsonant...      */
      }                                         /*                         */
   }                                            /*                         */
                                                /*                         */
   printf("\nVokale: %d\t\tKonsonanten %d",v,k); /*  Ergebnis ausgeben.     */
   printf("\nLeerzeichen: %d\t\tSatzzeichen: %d",l,s);  /*                 */
}                                               /*                         */
                                                /*************************/
```

**Prog. 4.13:** *Zeichentypen zählen mit* switch

Die einzelnen *case*-Anweisungen dienen nur als Marke, als Einstieg in die Anweisungsfolge. Wäre die *break*-Anweisung nicht vorhanden, dann würde mit der nächsten Anweisung im *switch* fortgefahren. Dies illustriert das folgende Programm-Fragment:

```
main()

{
   int a, b, c;

    for (i = 3; i; --i)
        switch(c = getchar())
            { case 'a':  ++a;
              case 'b':  ++b;
              case 'c':  ++c;
            }

    printf("\n a = %d, b = %d, c = %d", a, b, c);
```

Dieses Programm liest drei Zeichen von der Tastatur. Gibt man ihm nun nacheinander die Zeichen 'a', 'b' und 'c' (auf die Reihenfolge kommt es nicht an!), so erhält man keineswegs das Ergebnis *a = 1, b = 1, c = 1*! Vielmehr sieht das Resultat so aus:

```
a = 1, b = 2, c = 3
```

Was ist passiert? Nehmen wir an, Sie geben als erstes ein 'a' ein. C findet eine passende Marke beim ersten *case* und zählt die Variable *a* hoch. Dann fährt es mit der nächsten Anweisung fort, ud zählt *b* und anschließend *c* hoch. Denn keiner hat ihm gesagt, daß es mit den Anweisungen im *switch* hinter *++a* aufhören soll! Ähnlich verhält es sich bei Eingabe von 'b', bei dem die Variablen *b* und *c* hochgezählt werden. Nur das 'c' wirkt nur auf eine Variable, aber nur, weil es zufälligerweise am Ende der Auswahlmöglichkeiten steht.

Was beabsichtigt war, sieht so aus:

```
main()

{
   int a, b, c;

    for (i = 3; i; --i)
        switch(c = getchar())
            { case 'a':  ++a;
                         break;
              case 'b':  ++b;
                         break;
              case 'c':  ++c;
            }

    printf("\n a = %d, b = %d, c = %d", a, b, c);
```

Wie Sie sehen, ist eine *default*-Marke in einer *switch*-Anweisung nicht erforderlich. Sie dient - wenn Sie vorhanden ist - als 'Mädchen für alles': wenn keine der anderen Marken greift, dann werden die mit *default* verknüpften Anweisungen ausgeführt. So zählt im Programm 4.13 alles als

Konsonant, was kein Vokal, Leerzeichen oder Satzzeichen ist (also auch - fälschlicherweise - die Ziffern!). Das Englische *default* heißt auf Deutsch "Nichterscheinen" (z.B. bei einem Gerichtstermin); man liest es im Zusammenhang mit *switch* am besten als "in allen anderen Fällen".

Die Reihenfolge, in der Sie die einzelnen *case*-Marken hinschreiben, spielt keine Rolle. Allerdings dürfen gleiche *case*-Marken in einem *switch* nicht mehrfach vorkommen. Auch sollten Sie sich angewöhnen, in jedem *switch* eine *default*-Klausel mit einem *break* aufzunehmen. Dies mag vielleicht nicht immer nötig sein, kann sich aber bei späteren Programmänderungen - wenn zusätzliche *case*-Marken aufgenommen werden - auszahlen.

### 4.4.3 Die for-Schleife

Die letzten beiden Kontrollstrukturen (*if* und *case*) dienten der Auswahl von Anweisungen. Die folgenden dienen zur Schleifenbildung, bewirken also die wiederholte Ausführung von Anweisungen.

Die Informatiker unterteilen die Gattung der Programmschleifen in verschiedene Unterabteilungen: Zählschleifen, abweisende und nichtabweisende Schleifen.

Zählschleifen sind solche Schleifen, bei denen mitgezählt wird, wie oft eine Anweisung bzw. eine Anweisungsfolge wiederholt werden soll. Zählschleifen benutzen dazu eine Schleifen- oder Zählvariable, deren Wert bei jedem Schleifendurchgang um einen bestimmten Betrag erhöht wird, die also 'hochgezählt' wird. Ebenso ist natürlich die umgekehrte Zählrichtung möglich, also das 'herunterzählen' der Schleifenvariable.

Die Zählschleife von C wird mit *for* gebildet; *for*-Schleifen sind Ihnen bereits aus Kapitel 3 bekannt. Der Übersichtlichkeit halber will ich noch einmal aufführen, welche Bestandteile eine *for*-Schleife enthalten kann:

```
for ( <initialisierung> ; <kontrollbedingung> ; <re-initialisierung> )
      <anweisung>
```

"Enthalten kann" sage ich deswegen, weil keineswegs alle Komponenten bei einer *for*-Schleife unbedingt erforderlich sind! So dienen Schleifenkonstrukte zwar dazu, Anweisungen zu wiederholen, aber in C können diese zu wiederholenden Anweisungen fehlen! Was sich so widersinnig anhört, kann durchaus seinen praktischen Nutzen haben. Möchte man etwa in einem Programm eine Bildschirmausgabe - beispielsweise eine Copyright-Meldung - für eine bestimmte Zeit auf dem Terminal 'festfrieren', dann bietet es sich an, nach Ausgabe des Textes eine "Bremsschleife" in das Programm einzubauen, um dessen Fortgang zu verzögern. Solche Bremsschleifen erreicht man, indem man einfach eine Variable hochzählt; etwa so:

```
for (i = 0; i != 30000; ++i)
     ;
```

Alles, was hier geschehen soll, wird bereits im Schleifenkopf (in den runden Klammern hinter *for*) erledigt. Eine zu wiederholende Anweisung ist eigentlich gar nicht nötig. Nun erzwingt die Syntax von *for* jedoch hinter dem Schleifenkopf eine Anweisung, weswegen man ganz einfach zu der weiter oben vorgestellten "leeren Anweisung" greift. Diese wird unter den Schleifenkopf geschrieben und säuberlich eingerückt. Das hat nicht nur ästhetische Gründe, sondern kann auch Fehler verhindern helfen.

Ein beliebter Bug in C ist es nämlich, etwas wie das Folgende zu schreiben:

```
int i, j;
for (i = 1, j = 0; i != 11; ++i);
     j += i;
```

Man meint, damit in *j* die Summe der Zahlen von 1 bis 10 berechnet zu haben. Tatsächlich enthält *j* aber nach Beendigung der Schleife den Wert 11! Denn hinter dem Schleifenkopf, auf der selben Zeile wie die runden Klammern, verbirgt sich ein unscheinbarer Strichpunkt: dadurch ist die Anweisung *j += i* gar nicht mehr im Geltungsbereich der *for*-Schleife!

Durch korrekte Einrückung wird klar, was das Programmfragment eigentlich tut:

```
int i, j;
for (i = 1, j = 0; i != 11; ++i)
     ;
j += i;
```

Gewöhnen Sie sich also an, niemals einen Strichpunkt auf die selbe Zeile wie den Kopf einer *for*-Schleife zu setzen.

In einer *for*-Schleife kann im Extremfall jeder der drei Teile <initialisierung>, <kontrollbedingung> und <reinitialisierung> weggelassen werden (nicht aber die trennenden Strichpunkte!). Damit erhält man:

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

Das ist eine von vielen Arten, in C eine Endlosschleife zu bauen. Denn wenn die Kontrollbedingung fehlt, dann wird einfach angenommen, sie wäre dauernd "wahr"! Das folgende Programmfragment zählt denn auch munter die Variable *i* hoch - ohne damit jemals wieder aufzuhören. Also nicht ausprobieren, denn Sie müßten zum Reset-Knopf greifen!

```
for (i = 0; ; ++i)
     ;
```

Da man in *for*-Schleifen den Reinitialisierungsteil ebenfalls weglassen kann, liegen auch hierin ernstzunehmende Fehlerquellen. Die bereits mehrfach benutzte *while*-Schleife, die Zeichen einliest bis zu einem definierten Ende-Zeichen (hier den Punkt):

```
while ((c = getchar()) != '.')
      .
      .
      .
```

kann man mit *for* ja auch so ausdrücken:

```
for ( ;(c = getchar()) != '.'; )
      .
      .
      .
```

Aber Vorsicht! Wehe, man kommt mit den Strichpunkten durcheinander und schreibt:

```
for ( ;;(c = getchar()) != '.')
      .
      .
      .
```

Wieder stellt sich eine hartnäckige Endlosschleife ein.

Neben den 'abgemagerten' *for*-Schleifen, bei denen Bestandteile weggelassen wurden, gibt es auch die 'aufgeblasenen' *for*-Schleifen, bei denen eine oder mehrere Komponenten des Schleifenkopfs mehrfach vertreten sind. Daß man auf diese Art mehrere Variablen initialisieren und reinitialisieren lassen kann, wurde ja bereits wiederholt in den Programmbeispielen demonstriert. Dazu müssen die einzelnen Anweisungen lediglich durch Komma getrennt werden.

Dieses trennende Komma ist genaugenommen ein Beispiel für den Einsatz eines Operators, den ich Ihnen bisher unterschlagen habe: der "Komma"-Operator. In C ist es möglich, mehrere Ausdrücke durch Komma getrennt hinzuschreiben. C garantiert Ihnen in diesem Fall, daß diese Ausdrücke von links nach rechts berechnet werden. Der Wert des gesamten Ausdrucks ist dann der Wert des letzten durch Komma getrennten Ausdrucks. Allerdings müssen Sie dabei beachten, daß die Zuweisungsoperatoren in der Vorrangtabelle noch vor dem Komma-Operator rangieren. Betrachten Sie dazu das folgende Programm und dessen Ergebnis (Abbildung 4.11; Anmerkung: Der ATARI-C-Compiler konnte in der mir vorliegenden Version den Komma-Operator nicht richtig übersetzen und lieferte ein falsches Ergebnis). Den Ausdruck *i = j++, ++j;* faßt C also auf, als wäre er so geklammert: *(i = j++), ++j;*.

```
main()

{   int i, j;

    printf("j = %d",j = 1);
    i = j++ , ++j;
    printf("\ni = j++, ++j ergibt %d. j = %d",i,j);
    i = (j++, ++j);
    printf("\ni = (j++, ++j) ergibt %d. j = %d",i,j);
}

j = 1
i = j++, ++j ergibt 1. j = 3
i = (j++, ++j) ergibt 5. j = 5
```

**Abb. 4.11:** *Der Komma-Operator und seine Wirkung.*

Allerdings: das Komma wird auch noch in einem anderen Zusammenhang verwendet, in dem es mit dem Komma-Operator nicht verwechselt werden darf. Es trennt die Argumente einer Funktion; aber in diesem Kontext kann man sich nicht darauf verlassen, daß die Argumente von links nach rechts ausgewertet werden. Der folgende Code

```
i = 1;
printf("%d %d",++i,++i);
```

kann bei einigen Compilern das Ergebnis "2 3" haben, bei anderen (darunter auch dem Compiler im ATARI-Entwicklungssystem) jedoch "3 2". Über die Reihenfolge, in der die Argumente einer Funktion ausgewertet werden, legt der C-Standard nichts fest.

### 4.4.4 Abweisende und nichtabweisende Schleifen: while und do...while

Bei diesem Schleifentyp wird eine Anweisung bzw. Anweisungsfolge in Abhängigkeit vom Warheitswert einer Bedingung wiederholt. Die abweisende Variante setzt man immer dann ein, wenn sich ein Problem umgangssprachlich so umreißen läßt:

```
solange eine Bedingung wahr ist
     tue dieses-und-jenes
```

In C formulieren Sie dies mit der *while*-Schleife, deren Anatomie Sie in der Abbildung 4.12 finden. Dieser Schleifentyp wird "abweisend" genannt, weil er die Kontrollbedingung überprüft, ehe er sich an die Ausführung der Anweisung macht. Ist diese Bedingung schon zu Beginn falsch, dann kommt es gar nicht erst zur Ausführung der Anweisung; C fährt einfach hinter der *while*-Schleife mit der Arbeit fort.

```
while (<kontrollbedingung>)
    <anweisung>
```

**Abb. 4.12:** *Anatomie der* while-*Schleife*

Bei Problemen, die erfordern, daß die Anweisung in der Schleife mindestens einmal ausgeführt wird, benötigt man die nicht-abweisende Variante der *while*-Schleife. Ihr entspricht die umgangssprachliche Problemformulierung:

```
tue dieses-und-jenes
    solange eine Bedingung wahr ist.
```

Da die Kontrollbedingung erst am Ende der Schleife überprüft wird, wird die Anweisung in der Schleife auf keinen Fall übergangen. Wie dieser Schleifentyp in C gebildet wird, zeigt Ihnen die Abbildung 4.13.

```
do
    <anweisung>
while (<kontrollbedingung>);
```

**Abb. 4.13:** *Anatomie der* do...while-*Schleife*

Beachten Sie, daß diese Schleife mit einem Strichpunkt hinter der Kontrollbedingung abgeschlossen werden muß. Den Unterschied zwischen abweisenden und nichtabweisenden Schleifen soll das folgende kurze Beispiel zeigen:

```
#define FALSCH 0

main()

{   while (FALSCH)
        printf("\nAbweisende Schleife!");

    do
        printf("\nNichtabweisende Schleife!")
    while (FALSCH);
}
```

In beiden Schleifen ist die Kontrollbedingung stets falsch (das heißt in C: hat den numerischen Wert 0). Entsprechend wird dieses Programm nur die Meldung "Nichtabweisende Schleife!" auf den Bildschirm bringen.

Auch bei den *while*-Schleifen gibt es - wie bei den *for*-Schleifen - wieder ein C-typisches Idiom. Oft trifft man *while*-Schleifen in C-Programmen an, bei denen die Wiederholungs-Anweisung fehlt. Möchte man z.B. ein Programm schreiben, das Eingaben von der Tastatur akzeptiert, erstmal alle Leerzeichen ignoriert und von da an alle eingelesenen Zeichen zählt bis zum ersten Leerzeichen, so kann man das wie im Programm 4.14 tun.

Die *while*-Schleife zu Beginn des Programms überliest Leerzeichen; dies so - also mit einer leeren Anweisung im Schleifenkörper - zu notieren, ist gängiger C-Slang. Nach Beendigung der *while*-Schleife wurde bereits ein Nicht-Leerzeichen gelesen, das natürlich mitgezählt werden muß. Deshalb erfolgt das Mitzählen in einer *do...while*-Schleife, die auch dafür sorgt, daß das nächste, die Schleife beendende Leerzeichen nicht fälschlich mitgezählt wird.

Da die eingelesenen Zeichen im Programm in keiner Weise weiterverwendet werden, ist es unnötig, sie in einer Zeichenvariablen aufzubewahren. Die übliche "Zuweisung im Vergleich" (Muster: *((c = getchar()) == ' ')*) unterbleibt deshalb. Das von *getchar* gelieferte Zeichen (vornehmer: der Wert der Funktion *getchar*) wird für den Vergleich benötigt, danach aber sofort 'vergessen'.

```
#include "stdio.h"

main()

{   char c;
    int i;

    i = 0;

    while (getchar() == ' ')
        ;

    do
        i++;
    while (getchar() != ' ');

    printf("\n%d Zeichen.",i);
}
```

```
/*****************************/
/*  Abweisende und nicht-    */
/*    abweisende Schleife.   */
/*****************************/
/*                           */
/*                           */
/*                           */
/*                           */
/* Leerzeichen ueberlesen    */
/*                           */
/*                           */
/* Es wurde bereits ein      */
/*  Zeichen gelesen; daher   */
/*  do-Schleife. Beachte     */
/*  den abschliessenden      */
/*  Strichpunkt!             */
/*****************************/
```

**Prog. 4.14:** *Abweisende und nichtabweisende Schleife*

Noch einmal möchte ich Sie darauf hinweisen, daß die *do...while*-Schleife mit einem Strichpunkt abgeschlossen werden muß. Zwar steht auch am Ende der *while*-Schleife in diesem Programm ein Strichpunkt; dabei handelt es sich jedoch um die leere Anweisung (die von der Schleife kontrollierte Anweisung) und nicht um eine Endemarkierung für die Schleife. Da *while*-Schleifen auch Anweisungsblocks kontrollieren können und diese nicht mit einem Strichpunkt abgeschlossen werden müssen, sind durchaus *while*-Schleifen möglich, die ohne Strichpunkt enden. Nicht so bei den *do...while*-Schleifen: hier hat stets ein Strichpunkt am Ende zu stehen.

### 4.4.5 Verlassen von Schleifen: break und continue

Normalerweise kontrolliert der Schleifenkopf die Anzahl der Wiederholungen in einer Schleife; bei der *for*-Schleife durch Zählen von Variablen, bei den *while*-Schleifen durch Überprüfen von Bedingungen am Schleifenanfang oder -ende. Manchmal kann es aber erforderlich sein, aus einer Schleife 'auszubrechen', sie zu verlassen in Abhängigkeit von Bedingungen, die nicht im Schleifenkopf kontrolliert werden. Für diese Art des vorzeitigen Ausstiegs aus einer Schleife hält C die Anweisungen *break* und *continue* bereit.

Taucht *break* in einer Schleife auf, dann wird diese sofort verlassen; es erfolgt kein erneuter Durchgang durch die Schleife mehr. Alle Anweisungen, die im Schleifenkörper auf *break* folgen, werden ebenfalls ignoriert. Die *continue*-Anweisung bewirkt ebenfalls ein sofortiges Verlassen der Schleife, diesmal jedoch für einen erneuten Durchgang. Mit *continue* kann man also lediglich verhindern, daß eine oder mehrere Anweisungen im Schleifenkörper von der Ausführung ausgenommen werden. Dazu ein Beispiel; die beiden folgenden Programmausschnitte bewirken genau das gleiche:

```
/* Leerzeichen ueberlesen; Variante 1 */

while (1)
    if ((c = getchar()) != ' ')
       break;


/* Leerzeichen ueberlesen; Variante 2 */

while ((c = getchar()) == ' ')
    ;
```

Beide Schleifen überlesen Leerzeichen. In der ersten Variante wird dies jedoch in einer (potentiellen) Endlosschleife getan: die Kontrollbedingung ist eine Konstante, eine, die immer wahr ist. Um jetzt den Computer nicht 'aufzuhängen', muß diese Schleife mit Gewalt verlassen werden. Dazu dient die *break*-Anweisung, die jedoch von *if* kontrolliert wird, um nur bei einem Nicht-Leerzeichen aktiviert zu werden.

Wie in diesem Beispiel ist es in allen Fällen (theoretisch) möglich, auf *break* zu verzichten und eine alternative Problemlösung mit *while* oder *do...while* zu erreichen. Es gibt jedoch Fälle, in denen *break* und sein Verwandter *continue* eine 'natürlichere' Formulierung erlauben. Aber was als 'natürlich' zu gelten hat, ist Geschmackssache - natürlich!

Im Zusammenhang mit der Mehrfachauswahl (*switch* und *case*) ist Ihnen *break* bereits einmal begegnet. Dort diente es zum Verlassen der *switch*-Anweisung. Ist ein *switch* in eine Schleife eingebettet, dann bewirkt ein

*break* im *switch* nicht, daß damit die Schleife abgebrochen wird. Das Programm 4.13 ist dafür ein Beispiel.

In dem Beispiel für *continue* - dem zweiten Schleifenunterbrecher - wird auf eine bereits bekannte Problemstellung zurückgegriffen. In einer Schleife sollen Leerzeichen und sonstige Zeichen gezählt werden, bis ein Punkt eingegeben wird. Auch hierzu werden wieder zwei Formulierungen angeboten.

In der ersten Variante sorgt die *continue*-Anweisung dafür, daß das Hochzählen der "andere"-Variablen übersprungen wird; in der zweiten Variante wird genau derselbe Effekt mit *if...else* erreicht. Natürlich ist hier (ebenso wie beim *break*-Beispiel) unter Gesichtspunkten des Stilpurismus der zweiten Variante der Vorzug zu geben.

Schleifen können ineinander geschachtelt werden; liegt so etwas vor, dann verlassen *break* und *continue* immer nur die Schleife, in der sie unmittelbar enthalten sind.

```
.
.
.
char c;
int leer, andere;

leer = andere = 0;
while ((c = getchar()) != '.')
{   if (c == ' ')
    {   ++leer;
        continue;
    }
    ++andere;
}

leer = andere = 0;
while ((c = getchar()) != '.')
    if (c == ' ')
        ++leer;
    else
        ++andere;
```

### 4.4.6 Die unbedingte Sprunganweisung: goto

Aus der schlechten alten Zeit der unstrukturierten Programmierung hat C ein Sprachkonstrukt übernommen, das in den siebziger Jahren der Hauptfeind der theoretischen Informatik war: der unbedingte Sprung mit *goto*. Dem BASIC-Programmierer wohlvertraut, ist dies das Mittel, mit dem man am einfachsten von einem an sich trivialen Problem zu einer völlig unverständlichen Lösung desselben gelangen kann. Ein bißchen ist dem *goto* in C die Schärfe genommen, da als Sprungmarken keine nichtssagenden Zeilen-

nummern vorgeschrieben sind (wie in BASIC), sondern (im Rahmen der Namenskonventionen von C) frei wählbare Namen. Ähnlich wie die *case*-Marken bei *switch* müssen auch die Sprungmarken für *goto* mit einem nachgestellten Doppelpunkt gekennzeichnet werden.

Die *goto*-Anweisung kann stets vermieden und durch geeignete Schleifen ersetzt werden. Während sich bei den - auch schon bedenklichen - *break* und *case*-Anweisungen gelegentlich Rechtfertigungen für ihre Anwendung finden lassen, ist *goto* stets die schlechteste aller Alternativen. Deshalb sollten Sie am besten sofort wieder vergessen, daß es diese Anweisung in C gibt. Und deshalb präsentiere ich Ihnen dafür auch kein Beispiel.

### 4.4.7 Die return-Anweisung.

Auch diese Anweisung ist Ihnen bereits begegnet. Sie dient dazu, bei Funktionen einen Wert zurückzugeben. Die *return*-Anweisung muß zu den Kontrollstrukturen gezählt werden, da sie die normale Reihenfolge der Abarbeitung unterbrechen kann. Taucht sie in einer Funktion auf, dann wird die Funktion mit dem hinter *return* angegebenen Wert sofort verlassen. Eventuell nachfolgende Anweisungen in der Funktion kommen nicht mehr zur Ausführung.

# 5 Arrays und Pointer

Einzelne Zeichen und Zahlen verarbeiten: das war alles, was die bisherigen Programmbeispiele zuwege brachten. Meist ist dies zu wenig; alle Programmiersprachen bieten deshalb die Möglichkeit, zusammengesetzte Datenobjekte zu verarbeiten.

Das natürliche Beispiel für zusammengesetzte Datenobjekte sind Zeichenketten, unter Programmierern auch als "Strings" bekannt. Strings sind Ihnen vertraut, seit Sie Lesen können, denn Strings sind nichts anderes als Aneinanderreihungen von Zeichen - Buchstaben und Ziffern, Sonderzeichen und Leerzeichen. Auch dieses Buch ist nichts anderes als ein einziger, langer String.

Sieht man das ganze aus einer etwas höheren Perspektive, so ist ein String nichts anderes als eine Aneinanderreihung von Objekten der gleichen Datenklasse (in diesem Fall von Zeichen bzw. *chars*) mit einer bestimmten Länge. Solche Aneinanderreihungen kann man nicht nur mit Zeichen bilden; auch Aneinanderreihungen von Zahlen sowie von anderen, bisher noch nicht besprochenen Datentypen sind möglich. In C werden alle diese Formen der Aneinanderreihung als "Array" bezeichnet.

## 5.1 Integer-Arrays

Ein Beispiel soll dies deutlich machen. Das folgende Beispielprogramm liest - wieder einmal; aber wir kommen schon noch zu anderen Anwendungen! - Zeichen von der Tastatur und zählt, wie oft jeder Buchstabe auftaucht. Mit den bisher verfügbaren Sprachmitteln wäre diese Aufgabe nur sehr umständlich zu lösen. Man könnte sich 26 Integer-Variable definieren, eine für jeden Buchstaben des Alphabets, und dann in einem *switch* mit 26 *case*-Marken die jeweils passende Variable hochzählen. Aber es geht einfacher; wie, das steht im Programm 5.1.

Ich habe angenommen, daß die bereits bekannten Makros zur Zeichenverarbeitung (*isupper*, *islower* etc.) in der Datei *ctype.h* zu finden sind. Im Anhang finden Sie diese Makro-Datei ausgedruckt.

Als erstes begegnet Ihnen in diesem Programm eine Array-Deklaration, nämlich

```
int buchst[26];
```

Deklarationen haben - Sie erinnern sich - den Zweck, Speicherplatz zu reservieren, der ausreicht, um Daten eines bestimmten Typs aufzunehmen und diesem Platz einen Namen zu geben, unter dem er im Programm angesprochen werden kann. Diese Aufgaben besorgt auch obige Deklaration, allerdings tut sie noch etwas mehr.

Die Array-Deklaration besorgt im Speicher des Computers eine zusammenhängende Folge von Speicherzellen, "Array" genannt, die groß genug ist, um 26 Werte bzw. "Komponenten" vom Typ Integer aufzunehmen. Die Größe dieses Speicherbereichs ergibt sich also aus dem Datentyp der Komponenten (hier: *int*) und der Anzahl der gewünschten Komponenten (hier: 26), die in eckigen Klammern hinter dem Namen des Arrays zu schreiben ist. Wie jede andere Anweisung wird die Array-Deklaration mit einem Strichpunkt abgeschlossen.

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

                                        /*************************/
main()                                  /*  Zeichenstatistik     */
                                        /*************************/
{   int buchst[26];                     /* Hier werden die Buch-  */
    int position;                       /*  staben gezaehlt.      */
    int i;                              /*                        */
    char c;                             /*                        */
                                        /*                        */
                                        /*                        */
    for(i = 0; i <26; ++i)              /*  Array initialisieren. */
        buchst[i] = 0;                  /*                        */
                                        /*                        */
    while ((c = getchar()) != '#')      /*  Lesen bis '#'         */
        if (isupper(c) || islower(c))   /*  Nur Buchstaben!       */
        {   c = toupper(c);             /*  Grosschreibung machen.*/
            position = c - 'A';         /*  Position berechnen.   */
            buchst[position] += 1;      /*  Komponente hochzaehlen*/
        }                               /*                        */
                                        /*                        */
    for (i = 0; i < 26; ++i)            /*  Den Array durchgehen. */
        printf("\n%c: %dmal",           /*                        */
                (char) (i + 'A'),       /*  Zeichen berechnen.    */
                buchst[i]);             /*  Haeufigkeit.          */
}                                       /*************************/
```

**Prog. 5.1:** *Zeichen zählen im Array*

Der Array erhält auch einen Namen, in diesem Falle *buchst*. Aber genauer müßte man sagen: die Array-Deklaration vereinbart auf einmal 26 Namen, nämlich einen für jede Komponente des Arrays! Die erste Komponente hat den Namen *buchst[0]*, die zweite heißt *buchst[1]* und so weiter bis zur letzten, die den Namen *buchst[25]* trägt. Jede dieser Komponenten kann eine Integer speichern, verhält sich also genauso wie eine 'normale' Integer-

Variable. Die Verhältnisse, die nach Ausführung der Array-Deklaration im Speicher Ihres Computers herrschen könnten, stellt die Abbildung 5.1 dar.

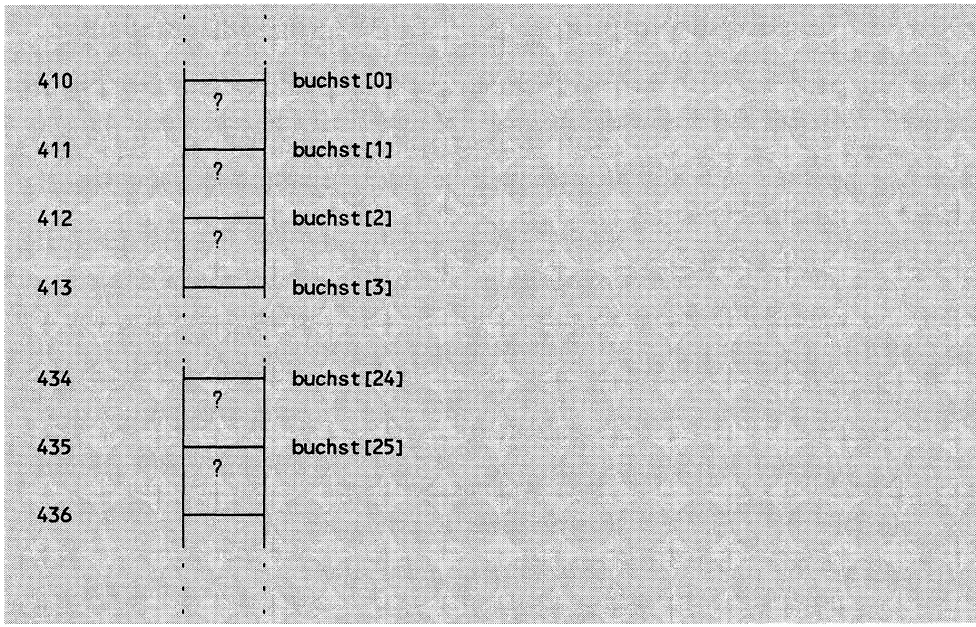



**Abb. 5.1:** *Speicherdiagramm eines Integer-Arrays*

Links von den Speicherzellen stehen die (hypothetischen) Adressen der einzelnen Komponenten; rechts davon finden Sie den Namen, den die einzelnen Komponenten durch die Array-Deklaration erhalten haben. Da die Deklaration nur Speicherplatz zuweist und benennt, aber keinen Wert in den Komponenten ablegt, enthalten die Speicherplätze ein Fragezeichen, um zu symbolisieren, daß ihr Wert noch nicht bekannt - nicht definiert - ist.

Natürlich hat der Array *buchst* im Zusammenhang mit dem Programm 5.1 auch einen Sachsinn. Die einzelnen Komponenten sollen nämlich dazu dienen, die Vorkommnisse der einzelnen Buchstaben zu zählen. In *buchst[0]* speichert das Programm die Anzahl der 'A's, in *buchst[1]*, die der 'B's und so fort bis hin zu *buchst[25]*, in dem die 'Z's gezählt werden. Beachten Sie, daß die Zählung bei Null beginnt, daß also die Komponenten eines C-Arrays, anders als in den meisten anderen Programmiersprachen, bei Null beginnend durchnumeriert sind.

Um eine Arraykomponente anzusprechen, schreibt man folglich ihre Positionsnummer in eckigen Klammern hinter den Array-Namen. Diese Posi-

tionsangabe heißt im Fachjargon "Array-Index". Ihre gar nicht zu überschätzende praktische Bedeutung gewinnen die Arrays dadurch, daß als Array-Index nicht nur Integer-Konstanten, sonder beliebige Ausdrücke mit Zahlenwert verwendet werden können. In der Regel sind dies Variablen.

Der erste ausführbare Schritt im Programm 5.1 demonstriert dies. Der Inhalt der Variablenkomponenten ist zu Beginn des Programmes undefiniert. Da in den Komponenten Werte gezählt werden sollen, ist es erforderlich, diese Komponenten mit dem Anfangswert 0 vorzubesetzen. Natürlich könnte man schreiben:

```
buchst[0] = buchst[1] = ... = buchst[24] = buchst[25] = 0;
```

Aber was wäre damit gewonnen? Der übliche Weg ist es, eine Integervariable, nennen wir Sie *i*, in einer Schleife von 0 bis 25 hochzuzählen und dann in der Schleife mit

```
buchst[i] = 0;
```

nacheinander die Zuweisung an die einzelnen Komponenten zu veranlassen. Im C-Jargon: die Variable *i* dient als Array-Index.

Jetzt dürfte sich auch schon abzeichnen, wie das Programm 5.1 die eigentliche Arbeit, nämlich das Zählen der Buchstaben, erledigt. Für jedes eingelesene Zeichen wird seine Position im Alphabet und damit im Array *buchst* bestimmt und dann die entsprechende Array-Komponente hochgezählt. Die Position von 'A' ist dabei 0, die von 'B' ist 1 usw. Da Sie schon mehrfach mit Buchstaben gerechnet haben, ist es für Sie kein Geheimnis mehr, wie man von einem Buchstaben zu seinem Positionswert gelangt: man subtrahiert einfach den Positionswert von 'A'.

Dies setzt jedoch voraus, daß man es nur mit Großbuchstaben zu tun hat; deshalb wird zuerst das eingelesene Zeichen in Großschreibung umgewandelt. Der nächste Schritt bestimmt die Positionsnummer des Zeichens und schließlich wird anhand der Positionsnummer die entsprechende Array-Komponente hochgezählt. Da Array-Komponenten ganz normale Variablen sind, kann man Ihnen auch auf die übliche Art einen Wert zuweisen, kann man insbesondere aus dem vollen Angebot der kombinierten Zuweisungsoperatoren schöpfen. Hier kommt der Operator, der gleichzeitig zuweist und hochzählt, zum Einsatz:

```
buchst[position] += 1;
```

All dies geschieht in der *while*-Schleife. Gezählt werden dürfen allerdings nur Buchstaben; für Sonderzeichen ist im Array *buchst* nichts vorgesehen.

Daher werden diese durch ein vorgeschaltetes *if* ausgefiltert; der Filter läßt nur Zeichen passieren, die entweder Groß- oder Kleinbuchstaben sind.

Ist die Zählerei zuende, dann soll die Statistik ausgegeben werden. Dazu wird wieder in einer *for*-Schleife der Array durchlaufen, wobei diesmal auf den Wert der einzelnen Komponenten zugegriffen wird. Ausgegeben werden soll das Zeichen, gefolgt von seiner Häufigkeit. Dazu ist es nötig, aus der Zeichenposition (die in der Variablen *i* gezählt wird) erstmal ein Zeichen zu machen. Dies geschieht durch Aufaddieren der Zeichenkonstante 'A'. Da diese Addition ein Ergebnis vom Typ *int* liefert (erinnern Sie sich an die Regeln der Typumwandlung), *printf* aber aufgrund der Formatanweisung *%c* ein Zeichen erwartet, muß das Ergebnis mit dem Cast-Operator *(char)* erst in den richtigen Datentyp 'umgebogen' werden.

```
#include <chartype.h>
#include <stdio.h>
#define is_char(c) (islower(c) || isupper(c))
#define pos(c) (toupper(c) - 'A')
#define zei(i) ((char) i + 'A')
                                        /*****************************/
main()                                  /*  Buchstaben-Statistik.    */
                                        /*****************************/
{   int buchst[26];                     /*                           */
    int i;                              /*                           */
    char c;                             /*                           */
                                        /*                           */
                                        /*                           */
    for(i = 0; i < 26; ++i)             /*                           */
        buchst[i] = 0;                  /*                           */
                                        /*                           */
    while ((c = getchar()) != '#')      /*  Zeichen mit Makro aus-   */
        if (is_char(c))                 /*  filtern.                 */
            ++buchst[pos(c)];           /*  Positionsbestimmung      */
                                        /*  mit Makro.               */
    for (i = 0; i < 26; ++i)            /*                           */
        if (buchst[i])                  /*                           */
        {   printf("\n%c: ", zei(i));   /*                           */
            hist(buchst[i]);            /*  Ausgabe in Histogramm-   */
        }                               /*   form.                   */
}                                       /*****************************/

hist(i)                                 /*****************************/
    int i;                              /*  Histogramm ausgeben.     */
                                        /*****************************/
{   while(i--)
        putchar('*');
}
```

**Prog. 5.2:** *Idiomatische Variante von Programm 5.1*

Aus didaktischen Gründen habe ich das Beispiel 5.1 umständlicher formuliert, als es ein versierter C-Programmierer tun würde. Die C-typische

Version reiche ich nun in Programm 5.2 nach. Die Hauptunterschiede zwischen den beiden Varianten liegen in der *while*-Schleife. Es kommen drei neue Makros zum Einsatz; das erste trägt den Namen *is_char* und soll erkennen, ob das Eingabezeichen ein Buchstabe ist. Es tut dies, indem es *isupper* und *islower* (beide ebenfalls Makros!) kombiniert. Das zweite Makro mit dem Namen *pos* erzeugt aus einem Zeichen seine Position im Array. Das Makro *zei* schließlich macht aus einer Integer-Variablen ein großgeschriebenes Zeichen. Jedesmal passiert nichts prinzipiell Neues, außer daß die zuvor 'ausbuchstabierten' Verarbeitungsschritte jetzt in einem Makro zusammengefaßt wurden.

Ich habe ja bereits erwähnt, daß als Array-Index beliebige Ausdrücke mit einem Zahlenwert herhalten können. Das Makro *pos* ist ein solcher Ausdruck (genauer: erzeugt nach Makro-Expansion einen solchen Ausdruck), kann also in den eckigen Klammern als Array-Index auftauchen. Fernerhin sind alle Array-Komponenten Variablen; zum Hochzählen von Variablen schreibt man am bequemsten *++*. Somit haben wir:

```
++buchst[pos(c)];
```

und diese eine Anweisung ersetzt drei Anweisungen im alten Programm. Da der Wert dieser Anweisung nicht weiterverwendet werden soll, ist es gleichgültig, ob man mit dem Prä- oder Postinkrement arbeitet. Auch bei der Ausgabe des Ergebnisses hat sich etwas geändert; es wird nur für solche Buchstaben eine Statistik ausgegeben, die einen Zählwert ungleich 0 haben. Und das Ergebnis wird, weil's schöner ist, graphisch - als Histogramm - dargestellt. Die eigens dazu definierte Mini-Funktion *hist* gibt soviele Sternchen aus, wie für den Buchstaben Einträge gemacht wurden und erhält als Argument daher die im *buchst*-Array gespeicherte Zahl. Ein Beispiellauf von Programm 5.2 sieht so aus:

```
ABC, die Katze liegt im Schnee#
A: **
B: *
C: **
D: *
E: *****
G: *
H: *
I: ***
K: *
L: *
M: *
N: *
S: *
T: **
Z: *
```

### 5.1.1 Arraygrenzen: Vorsicht, Falle!

Über einen Punkt müssen Sie sich bei den Arrays im Klaren sein: ihre Länge steht fest. Bei der Deklaration des Arrays geben Sie in den eckigen Klammern die 'Abmessungen' des Arrays an. Hier muß eine Konstante stehen; jeder Versuch, an diese Stelle eine Variable zu schmuggeln, wird vom Compiler unbarmherzig mit einer Fehlermeldung geahndet. Das schließt natürlich nicht mittels *#define* eingeführte symbolische Konstanten aus.

Ihre Längenangabe in der Array-Deklaration legt die Maximallänge des Arrays fest. Wenn Sie (wie in den beiden letzten Beispiel-Programmen) einen Array mit 26 Komponenten deklariert haben, dann steht es Ihnen frei, davon nur die ersten zehn Komponenten zu verwenden. Dem C-Programm macht das nichts aus, Sie haben lediglich Platz verschwendet.

Anders, wenn Sie über das Ziel hinausschießen und bei einem Array mit 25 Komponenten versuchen, einen Wert in die sechsundzwanzigste Komponente zu schreiben. Wenn Sie bisher mit BASIC oder Pascal gearbeitet haben, dann denken Sie vielleicht "Halb so tragisch; gibt's eben eine Fehlermeldung". Nicht bei C. Weder der Compiler noch Ihr Programm selbst überprüft, ob Sie auch die Indexgrenzen Ihrer Arrays eingehalten haben. All die folgenden Scheußlichkeiten sind in C also möglich:

```
int arr[10];

arr[15] = 0;
arr[127] = 0;
arr[-2] = 0;
```

Daß der Compiler sowas klaglos durchgehen läßt bedeutet keinesfalls, daß es Sinn macht! In der Regel legen Programme, die über Array-Grenzen schreiben, ein gar seltsames Verhalten an den Tag. Es kann sein, daß sie sofort nach dem Start den Rechner 'aufhängen'; dies ist sogar noch das kleinere Übel. Denn es kann auch sein, daß sie eine Zeitlang ohne Schwierigkeiten laufen. Schon freuen Sie sich, glauben, daß alles in Ordnung ist und legen des Programm als erledigt zu den Akten. Und plötzlich, meist wenn Sie schon längst vergessen haben, wie das Programm funktioniert, passieren wundersame Dinge. Das sind die heimtückischsten Fehler, da sie zu einem Zeitpunkt auftreten, da der Programmierer sich erst wieder in die Logik seines Programmes einarbeiten muß und die deshalb umso mehr Arbeit machen.

Andere Programmiersprachen machen - entweder, so weit dies möglich ist, bereits durch den Compiler, oder zur Laufzeit des Programmes - ausführliche Überprüfungen auf Einhaltung der Indexgrenzen. Ein fehlerhaftes Programm wird auch in diesen Sprachen nicht laufen, aber es verabschiedet

sich wenigstens gnädig mit einer Fehlermeldung, was das Debugging beträchtlich erleichtert.

Warum bietet C diesen Komfort nicht? Das liegt in der Philosophie der Sprache begründet. C ist eine Sprache für Systemprogrammierung, gehört also in die Hand eines Systemprogrammierers, und das sind in der Regel Leute, die schon wissen, was sie tun. Das kann so weit gehen, daß eine bewußte Überschreitung der Indexgrenzen vorliegt, d.h. daß der Programmierer einen ganz bestimmten Effekt mit voller Absicht herbeiführen will; C will ihn daran nicht hindern. Es ist eine Sprache für Erwachsene; BASIC und Pascal jedoch sind als ausgesprochene Anfänger- und Lernsprachen entwickelt worden. Die Überprüfung zur Laufzeit führt außerdem zusätzlichen Code in Ihr Programm ein, macht es dadurch umfangreicher und langsamer. C möchte Ihnen aber gerade die Möglichkeit geben, möglichst kompakte und schnelle Programme zu schreiben.

Ein weiteres Wort der Warnung ist im Zusammenhang mit den Arrays angebracht. Bei der Deklaration vereinbaren Sie eine Maximalgröße für Ihre Arrays. Wiederum hindert Sie C nicht daran, Arrays zu vereinbaren, die größer sind als der gesamte verfügbare Speicher! Im C des ATARI ist es möglich, einen Integer-Array mit einer Million Komponenten zu deklarieren (den zu speichern man runde 490K bräuchte; so viel ist bei einem 520K-System keinesfalls mehr übrig, wenn das Betriebssystem geladen ist!) und dann einen Wert in seine letzte Komponente zu schreiben. Natürlich mit dem Ergebnis, daß der Rechner hängt...

Seien Sie also nicht maßlos in Ihren Speicherplatz-Wünschen!

```
#include <stdio.h>
#define MAX 80

                                        /****************************/
main()                                  /* Eingabe umdrehen.        */
                                        /****************************/
{   char arr[MAX];                      /* Hier kommen die Zeichen */
    char c;                             /*  rein.                   */
    int i;                              /*                          */
                                        /*                          */
    i = 0;                              /*                          */
                                        /*                          */
    while (i < MAX && (c = getchar()) != '.')   /* Auf Einhaltung der In- */
        arr[i++] = c;                   /*  dexgrenzen achten!      */
                                        /*                          */
    printf ("\n");                      /* Ergebnis kommt auf neue */
                                        /*  Zeile.                  */
    do                                  /* Do-Schleife garantiert, */
        putchar(arr[--i]);              /*  dass auch arr[0] aus-  */
    while (i);                          /*  gegeben wird.          */
}                                       /****************************/
```

**Prog. 5.3:** *Umdrehen der Eingabe mit Zeichen-Array*

## 5.2 Zeichen-Arrays und Strings

Natürlich sind Sie in C nicht auf Integer-Arrays beschränkt. Jeder einfache Datentyp kann zur Bildung von Arrays herhalten. Es sind also auch Zeichen-Arrays möglich, wie das Beispiel-Programm 5.3 zeigt. Dieses liest in einen Zeichen-Array mit einer Maximallänge von 80 Komponenten (definiert als symbolische Konstante MAX) solange Zeichen, bis entweder die Arraygrenze erreicht oder ein Punkt eingegeben wurde. Anschließend werden die eingelesenen Zeichen in umgekehrter Reihenfolge wieder ausgegeben.

Beachten Sie, wie in der *while*-Schleife zum Einlesen zugleich die Einhaltung der Array-Grenzen überprüft wird. Das Hochzählen der Indexvariablen *i* erfolgt im Schleifenkörper mittels Postinkrement. Das bedeutet, daß nach Verlassen der Einlese-Schleife *i* hinter das zuletzt gespeicherte Zeichen weist (der Punkt wird nicht mit abgespeichert!). Deshalb wird in der Ausgabeschleife mit Prädekrement gearbeitet. Die *do...while*-Schleife sorgt dafür, daß auch noch das letzte Element mit dem Index 0 ausgegeben wird. Hier eine Beispiellauf für dieses Programm:

```
Dieser Satz hat weniger als achtzig Zeichen.
nehcieZ gizthca sla reginew tah ztaS reseiD
```

Von den Zeichen-Arrays zu den Strings ist es in C nur ein kleiner Schritt. Als Konstanten sind Ihnen die Strings schon begegnet: Meldungen, die Sie mit *printf* ausgeben, genauer: das erste Argument von *printf* ist ein String. Stringkonstanten erkennt man daran, daß sie in doppelte Anführungszeichen ("Gänsefüßchen") eingeschlossen sind.

Jetzt wissen Sie, wie Stringkonstanten im Programm aussehen; eine andere Frage ist, wie Sie der Computer sieht. Strings sind für ihn Zeichen-Arrays, die mit einem Null-Byte abgeschlossen sind, also einem Byte, das den (hexadezimalen, binären, oktalen, dezimalen - das ist in diesem Fall alles Eins) Zahlenwert 0 hat. Die Abbildung 5.2 zeigt Ihnen, wie der String "String" im Speicher abgelegt ist:

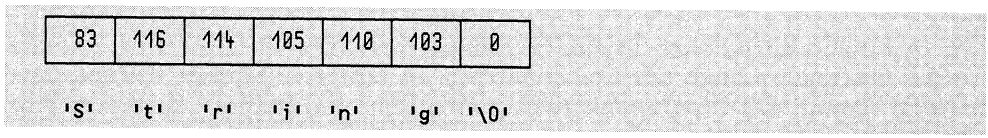



**Abb. 5.2:** *Speicherdiagramm eines Strings*

Im Unterschied zu den bisherigen Speicherdiagrammen habe ich die Speicherzellen diesmal der Übersichtlichkeit wegen horizontal angeordnet. Die Werte in den Speicherzellen sind die (dezimalen) ASCII-Werte des ge-

speicherten Zeichens. Darunter sehen Sie das jeweilige Zeichen als Zeichenkonstante; beachten Sie die Ersatzdarstellung des Null-Bytes.

Wegen dieses abschließenden Null-Bytes benötigt ein String immer ein Byte mehr an Speicherplatz, als er sichtbare Zeichen aufweist. Da Strings Zeichen-Arrays sind und da Sie für die Deklaration von Arrays selbst zuständig sind, sollten Sie dies beachten. Wenn Sie also in Ihrem Programm mit Strings arbeiten wollen, deren Maximallänge 100 Zeichen beträgt, so müssen Sie einen Zeichenarray mit mindestens 101 Komponenten deklarieren, um nicht in Schwierigkeiten zu geraten.

Das nächste kleine Beispiel (Programm 5.4) liest Zeichen von der Tastatur, speichert Sie in einem Array, macht einen String daraus und gibt diesen mit *printf* aus. Das Ende des Eingabestrings erkennt das Programm am Betätigen der *Return*-Taste.

```
#include <stdio.h>

#define ARRLEN 81
#define STRLEN ARRLEN - 1
                                        /***************************/
main()                                  /* String von Tastatur le- */
                                        /*  sen.                   */
{   char string[ARRLEN];                /***************************/
    char c;                             /*                         */
    in i;                               /*                         */
                                        /*                         */
    i = 0;                              /*                         */
    printf(">");                        /*  Prompt ausgeben.       */
                                        /*                         */
    while (i < STRLEN && (c = getchar()) != '\r') /* Zeichen lesen und weg- */
        string[i++] = c;                /*  speichern.             */
                                        /*                         */
    string[i] = '\0';                   /* So wird aus dem Array   */
                                        /*  ein String...          */
    printf(">%s",string);               /* Ausgeben; %s ist Format */
                                        /*  fuer Strings.          */
}                                       /***************************/
```

**Prog. 5.4:** *String von der Tastatur lesen.*

Die maximale Stringlänge beträgt in einem C-Programm stets eins weniger als die maximale Länge des Zeichenarrays. In bester C-Manier werden deshalb zu Beginn des Programms zwei symbolische Konstanten definiert: eine für die Arraylänge mit dem vielsagenden Namen ARRLEN und eine, die die Stringlänge angibt und STRLEN heißt.

Den Zeichenarray zur Speicherung des Strings auf den Namen *string* zu taufen: diesen billigen Kalauer konnte ich mir nicht verkneifen. Seine Länge wird durch die symbolische Konstante STRLEN (die für den Wert

81 steht) angegeben. Somit hat ein String mit maximal 80 Zeichen (Konstante STRLEN) darin Platz.

Das Programm gibt als erstes eine Eingabeaufforderung (auch "Prompt" genannt) aus und liest dann - ähnlich wie Programm 5.3 - in einer *while*-Schleife Zeichen von der Tastatur. Es hört damit auf, wenn die maximale Stringlänge erreicht wurde, oder wenn Sie die *Return*-Taste betätigen.

An dieser Stelle muß ich auf eine Besonderheit des ATARI-C eingehen. Üblicherweise wird bei Betätigen der *Return*-Taste an C das Zeichen "Newline" weitergeleitet, dessen Ersatzdarstellung in C '\n' ist. Das ATARI-C - genauer: die Funktion *getchar* - weicht von dieser Konvention ab; man erhält "Return". Der Grund liegt darin, daß in der mir vorliegenden Version des C-Compilers *getchar* umittelbar durch Aufruf einer Betriebssystems-Funktion realisiert wird, die sich nicht um die Einhaltung des in C üblichen Protokolls kümmert. Dies kann jedoch in späteren Versionen des Compilers und zu dem Zeitpunkt, zu dem Sie dieses Buch lesen, geändert sein. Sollte das Beispielprogramm bei Ihnen also nicht funktionieren, so ändern Sie den Kopf der *while*-Schleife in:

```
while (i < STRLEN && (c = getchar()) != '\n')
```

Ist das Endekriterium für die Zeicheneingabe erreicht, dann kommt eine kleine aber bedeutungsschwere Anweisung. Hinter das letzte gelesene Zeichen wird ein Null-Byte geschrieben - und so aus einem gewöhnlichen Zeichen-Array ein String gemacht.

Jetzt erst ist *string* ein passendes Argument für *printf*, das neben verschiedensten Zahlen und Zeichen auch Strings ausgeben kann. Es tut dies, wenn Sie in den Kontrollstring die Formatangabe *%s* aufnehmen. Dann erwartet *printf* als zugehöriges Argument einen String und verläßt sich darauf, daß dieser mit dem ominösen Null-Byte abgeschlossen wird. Wehe, er ist's nicht; dann bekommen Sie Ärger mit Ihrem Programm!

Der *printf*-Aufruf verdeutlicht auch, wie man einen String - oder, ganz allgemein: einen Array - an eine Funktion als Argument übergeben kann: einfach, indem man seinen Namen hinschreibt!

Jetzt fragt sich nur, wie Funktionen aussehen, die Strings als Argumente haben. Das zeigt Ihnen Beispielprogramm 5.5. Es benutzt zwei Stringfunktionen: *get_s* zum Einlesen eines Strings von der Tastatur und *str_len* zur Bestimmung der Länge eines Strings. Im Hauptprogramm werden lediglich 5 Strings eingelesen und dann zusammen mit Angaben über ihre Länge wieder ausgegeben.

```
#define MAX 1025
#define MAXSTR MAX - 1

                                                    /************************* */
main()                                              /*  Stringlaenge bestim-   */
                                                    /*  men.                   */
{  char string[MAX];                                /*************************/
   int i;                                           /*                        */
   int str_len();                                   /*                        */
                                                    /*                        */
                                                    /*                        */
   for (i = 5; i ; --i)                             /* 5 Strings einlesen (mit */
   {  putchar('>');                                 /*  Prompt), Laenge be-    */
      get_s(string);                                /*  stimmen und aus-       */
      printf("\n--> %d Zeichen\n",str_len(string)); /*  geben..               */
   }                                                /*                        */
}                                                   /*************************/

                                                    /*************************/
int str_len(s)                                      /* Stringlaenge bestimmen. */
    char s[];                                       /*************************/
                                                    /*                        */
{  int i;                                           /*                        */
                                                    /*                        */
   i = 0;                                           /* Noch kein Zeichen ge-  */
                                                    /*  funden...             */
   while (s[i])                                     /* Endekriterium ist Null- */
      ++i;                                          /*  Byte.                 */
                                                    /*                        */
   return i;                                        /*                        */
}                                                   /*************************/


                                                    /*************************/
get_s(s)                                            /* String von Tastatur    */
char s[];                                           /*  lesen.                */
                                                    /*************************/
{  char c;                                          /*                        */
   int i;                                           /*                        */
                                                    /*                        */
   for (i = 0; (c = getchar()) != '\r'; ++i)        /* Endekriterium ist RE-  */
      s[i] = c;                                     /*  TURN-Taste.           */
                                                    /*                        */
   s[i] = '\0';                                     /* Der springende Punkt... */
                                                    /*************************/
}
```

**Prog. 5.5:** *Unterfunktionen, die mit Strings arbeiten*

Das Hauptprogramm bietet keine Neuigkeiten. Es bedient sich zum Lesen des Strings der Funktion *get_s* und läßt sich die Stringlänge mit *str_len* anzeigen. Beide Funktionen erhalten Strings als Argumente; an diese Feststellung schließt sich zwanglos die Frage an, wie denn der Parameter in der Funktion zu deklarieren ist.

Im wesentlichen wie im Hauptprogramm. Wenn Sie die beiden Funktionen betrachten, werden Sie feststellen, daß in der Funktions-Deklaration lediglich die Größenangabe in den eckigen Klammern weggeblieben ist. Um dies zu erklären, muß ich Ihnen ein wenig über die Mechanismen der Parameter-Übergabe in C erzählen.

Wenn eine Funktion mit einem Parameter vom Typ *int* aufgerufen wird, dann erhält die Funktion in diesem Parameter eine Kopie des Arguments. Betrachten Sie dazu folgendes Beispiel:

```
{
    .
    .
    .
    int i;

    i = 5;

    fun(i);
    .
    .
    .
}

fun(p)
int p;

{   .
    .
    .
}
```

Beim Aufruf von *fun* hat *i* den Wert 5, d.h. an der mit *i* bezeichneten Speicherstelle ist diese Zahl abgelegt. Die Übergabe des Arguments *i* an den Parameter *p* erfolgt, indem dieser Wert in die mit *p* assoziierte Speicherstelle kopiert wird. Für *p* wird also ebensoviel Speicherplatz bereitgestellt wie für *i*. Wenn *p* den Wert empfangen hat, dann sind die Verbindungen zu *i* abgebrochen: Änderungen an *p* innerhalb von *fun* haben keinerlei Auswirkungen auf den in *i* gespeicherten Wert. Dies kann man so zusammenfassen: eine Funktion empfängt in Ihrem Parameter eine Kopie des Werts des Arguments.

Nicht so bei den Arrays und damit den Strings; denn keine Regel ohne Ausnahme. Wenn Sie an *str_len* einen String mit 50 Zeichen übergeben, dann werden keineswegs all diese Zeichen als Kopie an *str_len* übergeben. Stattdessen erhält die Funktion die Anfangsadresse des Strings (das wird im nächsten Unterkapitel noch ausführlicher erklärt!) im Parameter übergeben. Das bedeutet im Wesentlichen: Modifikationen am Parameter sind auch Modifikationen in der aufrufenden Funktion. Der String, der im aufrufen-

den Programm deklariert ist, wird in der Funktion manipuliert, aber unter einem anderen Namen (nämlich dem des Parameters) angesprochen.

Deshalb muß C in der Parameter-Deklaration über die Array-Länge nichts mitgeteilt werden. Denn es reserviert nur Speicherplatz für die Aufnahme der Array-Adresse; wieviele Komponenten dieser Array in der rufenden Funktion hat, ist uninteressant.

Die Parameter-Deklaration

```
char s[];
```

sagt C, daß ein Parameter mit Namen *s* deklariert wird und daß dieser Parameter die Anfangsadresse eines Zeichen-Arrays, also eines Strings, beim Aufruf erhalten wird. Daß es ein Array ist, erkennt der Compiler an den (leeren) eckigen Klammern; daß es ein Zeichen-Array ist, wird aus dem Datentyp *char* ersichtlich, der die Deklaration einleitet.

Auf einen so deklarierten String-Parameter kann dann in der üblichen Art durch Angabe eines Index zugegriffen werden. Ebendies machen die beiden Funktionen. Die zweite, *get_s*, ist eigentlich nur eine Kopie von Programm 5.4. Die erste demonstriert, warum ein Null-Byte als String-Ende gar nicht so unpraktisch ist.

Sie erinnern sich ja noch daran, daß in C ein Null-Wert für den Wahrheitswert "falsch" steht. Deshalb kann man in C-Schleifen den Inhalt eines Zeichenarrays zur Kontrollbedingung der Schleife machen. Das demonstriert die Funktion *strlen*. Mit der Anweisung

```
while (s[i])
    ++i;
```

wird der Index *i* solange hochgezählt, bis sich an der *i*-ten Stelle des Strings *s* der Wert 0 einstellt; das aber ist genau am Ende des Strings der Fall, da dieser - wie jeder String - durch ein Null-Byte abgeschlossen wird. Da die Zählung in Arrays bei 0 beginnt, ist garantiert, daß *i* nach verlassen der *while*-Schleife die Stringlänge enthält. Es wird als Wert zurückgegeben.

Aber für den Umgang mit Strings (und mit Arrays) gibt es in C wesentlich elegantere und einfachere Sprachmittel, als bisher vorgestellt. Diese haben mit den bereits erwähnten Array-Adressen zu tun, genauer, mit Adressen im Allgemeinen.

### 5.2.1 Pointer

Die Entscheidung, C mit einer Fülle von Operatoren und damit mit einem mächtigen Inventar an "Wert"-vollen Ausdrücken auszustatten, trägt zum großen Teil zur Ausdrucksmächtigkeit dieser Sprache bei. Mehrfach habe ich Ihnen bereits demonstriert, wie sich damit Problemlösungen in hochkonzentrierter Form darstellen lassen, was nicht nur den künstlerischen Ehrgeiz des Programmierers befriedigt, sondern in der Regel auch die Effizienz der Programme steigert.

Den zweiten großen Beitrag, dem C seine einzigartige Stellung als höhere Sprache zur Systemprogrammierung verdankt, liefern die "Pointer". Pointer haben mit Adressen zu tun, es sind Variablen, deren Werte Adressen sind.

Aber erst mal schön der Reihe nach: Sie wissen, daß eine Variable nichts anderes ist als ein selbstgewählter Name für einen Speicherplatz, an dem ein Wert abgelegt werden kann. Sie wissen ferner, daß Ihr Computer Speicherplätze identifizieren kann, weil er sie mit 'Hausnummern' versehen hat, den Adressen. Daher hat jede Variable eine Adresse.

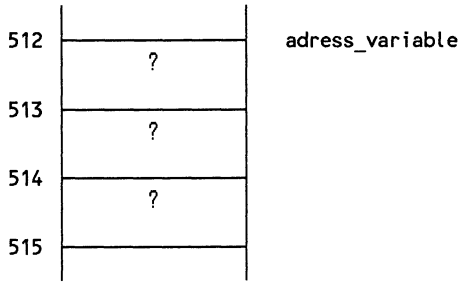
In C können Sie Variablen vereinbaren, die als Wert Adressen,die Adressen von anderen Variablen, erhalten sollen. Dazu ein Beispiel:

```
.
.
.
.
int *adress_variable;       /* 1 */
int i;                      /* 2 */

i = 5;                      /* 3 */
adress_variable = &i;       /* 4 */
*adress_variable = 6;       /* 5 */
.
.
.
.
```

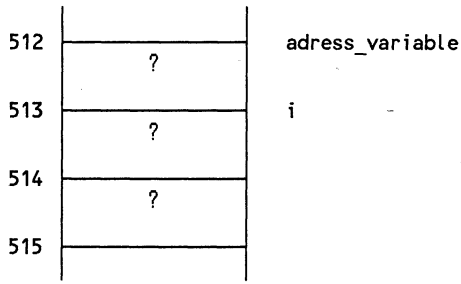
Ich habe die einzelnen Schritte dieses Programmfragments durchnumeriert, um besser erklären zu können, was hier abläuft.

1. Mit *int *adress_variable;* wird eine Variable deklariert, die den Namen *adress_variable* trägt und zur Aufnahme der Adresse einer Integer-Variable bestimmt ist. All dies erkennt der Compiler an der Deklaration; wie, das werde ich Ihnen noch genauer erklären. Nach Ausführung dieser Anweisung könnte es im Speicher Ihres Computers so aussehen:

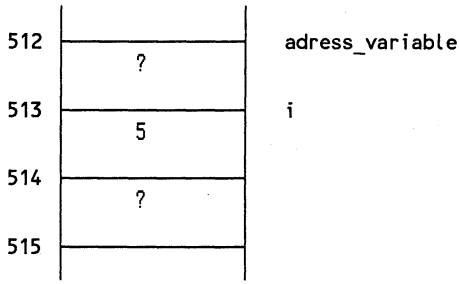



Am linken Rand stehen die (fiktiven) Adressen der Speicherzellen, die durch die üblichen Kästchen symbolisiert werden.
Durch die Deklarationsanweisung angetrieben, sucht sich der Compiler den ersten freien Speicherplatz, der zur Aufnahme einer Adresse ausreicht (hier die Speicherzelle 512) und merkt sich, daß Sie ihn auf den Namen *adress_variable* getauft haben. Da die Deklaration über den in der Variablen gespeicherten Wert noch nichts festlegt, steht in der Speicherstelle ein Fragezeichen: der Wert ist undefiniert.

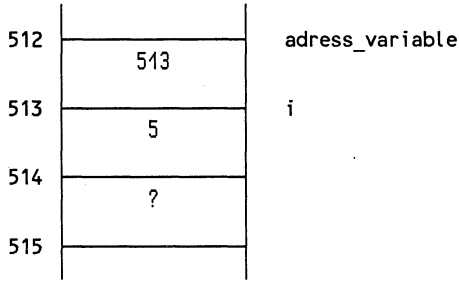
2. Was der Compiler bei der Deklaration einer Integer-Variablen macht, ist klar; deshalb hier nur das Ergebnis als Grafik:




3. Durch die Zuweisung wird *i* aus seinem undefinierten Zustand erlöst und erhält einen Wert. In Bildern:

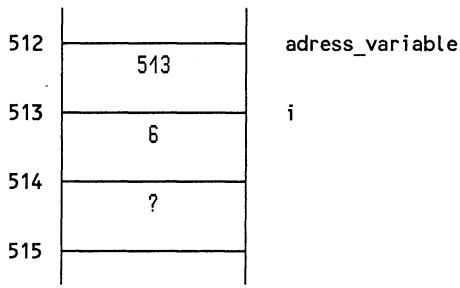


4. Jetzt wird's mysteriös! Ganz offensichtlich haben wir es hier mit einer Zuweisung zu tun: *adress_variable* steht links vom Zuweisungszeichen =. Auch die rechte Hälfte der Zuweisung wäre in Ordnung, doch *i* ist mit einem seltsamen & verunziert! Was soll's? Dazu erstmal das Speicherdiagramm:




Jetzt befindet sich in *adress_variable* also die Adresse der Integer-Variablen *i*. Dies ist das Werk des Operators &, den ich Ihnen bisher unterschlagen habe. Seine Wirkung in kurzen Worten: er wird angewandt auf eine Variable und liefert als Wert die Adresse dieser Variable. Diese Adresse - eine Zahl - kann dann im Programm weiterverwendet werden, etwa für eine Zuweisung wie im vorliegenden Fall.

5. Aber warum sollte jemand so etwas tun wollen? Der Hauptgrund für die Einführung von Variablen war ja gerade, daß sich die Programmierer nicht mehr mit den numerischen Adressen herumschlagen müssen!
Sehen Sie weiter. Der fünfte Schritt ist wieder eine Anweisung; diesmal eine, deren rechter Teil halbwegs zivilisiert aussieht. Dafür verwirrt der linke Teile mit einem vorangestellten Sternchen. Sehen Sie sich das Ergebnis an:




Der Wert von *i* hat sich geändert! Obwohl in der Anweisung *i* mit keinem Wort erwähnt wurde! Wie das? Das ist die Wirkung des zweiten Operators, den ich bisher unterschlagen habe, des *. Dieser führt ein Doppelleben: in seiner zweistelligen Existenz bewirkt er die Multiplika-

tion. Als einstelliger Operator hingegen weist er C an, die Variable, vor der er steht, als Zeiger zu interpretieren.

In der Zuweisung

```
*adress_variable = 6;
```

wird C angewiesen, nicht etwa den Wert 6 an der Adresse *adress_variable* abzulegen, sondern vielmehr nachzusehen, welcher Wert in *adress_variable* gespeichert ist, diesen Wert als Adresse aufzufassen und an dieser Adresse die 6 abzuladen. Da durch die Anweisung /* 4 */ an *adress_ variable* die Adresse von *i* zugewiesen wurde, läuft das auf eine Veränderung von *i* hinaus. Sie können das auch etwas anders formulieren: durch die Zuweisung /* 4 */ ist *adress_variable* zu einem Alias für *i* geworden. Mit dem Alias und dem Operator * ist es möglich, indirekt (ohne sie zu erwähnen, gleichsam über einen Strohmann) die Variable *i* zu manipulieren.

Daß der Operator * die indirekte Wert-Änderung erlaubt, hat ihm den englischen Namen "indirection" eingetragen. Dieser Begriff widersetzt sich bisher hartnäckig einer akzeptablen Eindeutschung, weswegen ich Sie mit einem furchtbaren Fachbegriff aus der Informatik behelligen muß. Das, was in der Anweisung /* 5 */ abgelaufen ist, nennt der Informatiker "Dereferenzierung".

Der Wert von *adress_variable* verweist auf eine andere Variable; er ist also eine "Referenz" (Verweisung). Durch * wird C angewiesen, dieser Referenz nachzugehen, sie aufzulösen oder eben: zu "dereferenzieren".

Da ist mir der Fachbegriff für den Typ der Variablen *adress_variable* schon lieber. Man sagt, durch die Deklaration /* 2 */ wird eine Variable vom Typ "Zeiger auf Integer" vereinbart, oder, im saloppen C-Jargon: "Pointer auf Integer" bzw. "Integer-Pointer".

Wenn es Integer-Pointer gibt, dann sollte es auch andere Pointer geben. In der Tat können Sie in C einen Pointer auf jeden elementaren Datentyp vereinbaren. Auch müssen die Pointer nicht ausschließlich links vom Zuweisungszeichen stehen. All dies demonstriert das Programm 5.6.

Das Programm manipuliert Pointer auf verschiedene Arten und gibt jeweils die Anweisungen und deren Ergebnis aus. Es hat rein didaktischen Wert und bringt nichts Nützliches zuwege. Sie können sich die mühselige Arbeit sparen, das Programm einzugeben, da ich Ihnen auch gleich noch das Ergebnis des Programms dazuliefere (in der Abbildung 5.3)

```
main()

{   int i, *ip, j;
    char c, *cp, d;

    i = 10;
    c = 'A';
    printf("\n\nAnweisung: i = 10; c = 'A';");
    printf("\nErgebnis: i = %d, c = %c", i, c);

    ip = &i;
    cp = &c;
    printf("\n\nAnweisung: ip = &i; cp = &c;");
    printf("\nErgebnis: i = %d, *ip = %d, c = %c, *cp = %c",
                        i,      *ip,      c,      *cp);

    *ip = 11;
    *cp = 'B';
    printf("\n\nAnweisung: ip = 11; cp = 'B';");
    printf("\nErgebnis: i = %d, *ip = %d, c = %c, *cp = %c",
                        i,      *ip,      c,      *cp);

    j = *ip;
    d = *cp;
    printf("\n\nAnweisung: j = *ip; d = *cp;");
    printf("\nErgebnis i = %d, *ip = %d, j = %d, c = %c, *cp = %c, d = %c",
                       i,       *ip,     j,      c,      *cp,     d);

    ++j;
    ++d;
    printf("\n\nAnweisung: ++j; ++d;");
    printf("\nErgebnis: i = %d, *ip = %d, j = %d, c = %c, *cp = %c, d = %c",
                        i,       *ip,     j,      c,      *cp,     d);

    *ip += 5;
    *cp += 5;
    printf("\n\nAnweisung: *ip += 5; *cp += 5;");
    printf("\nErgebnis: i = %d, *ip = %d, j = %d, c = %c, *cp = %c, d = %c",
                        i,       *ip,     j,      c,      *cp,     d);
}
```

```
/*****************************/
/*  Pointer.                 */
/*****************************/
/* Jeweils zwei Variable     */
/*  und einen Pointer de-    */
/*  klarieren.               */
/* Einfache Wertzuweisung.   */
/*****************************/

/*****************************/
/* Die Pointer werden zum    */
/*  Alias der Variablen.     */
/*****************************/

/*****************************/
/* Indirekte Wertzuweisung   */
/*****************************/

/*****************************/
/* Wertzuweisung mit Alias.  */
/*****************************/

/*****************************/
/* Das beruehrt nur "j"      */
/*  und "d"...               */
/*****************************/

/*****************************/
/* Das beruehrt auch "i"     */
/*  und "j"...               */
/*****************************/
```

**Prog. 5.6:** *Pointer*

Zu Beginn des Programmes sehen Sie, daß die Deklaration von Pointern in die Deklaration von 'normalen' Variablen eingestreut sein kann. Außerdem sagt diese erste Zeile einiges darüber aus, wie der Compiler eine Deklaration auffaßt. Alle Variablen in der ersten Programmzeile haben etwas mit dem Datentyp *int* zu tun; das erkennt er an dem einleitenden Schlüsselwort für den Datentyp. Die Variablen *i* und *j* sind einfache Integervariablen, d.h., wenn man sie im Programm verwendet, dann liefern sie einen Integerwert. Anders steht es mit *ip*. Wenn man in einem Programm **ip* hinschreibt, dann liefert dies ebenfalls einen Integerwert; dies entnimmt der Compiler der Deklaration. Deshalb muß *ip* eine Variable vom Typ "Integer-

Pointer" sein (das denkt er sich dazu). Sie sehen: die Deklaration von Pointer-Variablen erfolgt auf ebenso 'indirektem' Wege wie der Umgang mit ihnen! Genaueres über die Deklaration finden Sie im nächsten Kapitel.

```
Anweisung:  i = 10; c = 'A';
Ergebnis:   i = 10, c = A
Anweisung:  ip = &i; cp = &c;
Ergebnis:   i = 10, *ip = 10, c = A, *cp = A
Anweisung:  ip = 11; cp = 'B';
Ergebnis:   i = 11, *ip = 11, c = B, *cp = B
Anweisung:  j = *ip; d = *cp;
Ergebnis:   i = 11, *ip = 11, j = 11, c = B, *cp = B, d = B
Anweisung:  ++j; ++d;
Ergebnis:   i = 11, *ip = 11, j = 12, c = B, *cp = B, d = C
Anweisung:  *ip += 5; *cp += 5;
Ergebnis:   i = 16, *ip = 16, j = 12, c = G, *cp = G, d = C
```

**Abb. 5.3:** *Das Ergebnis von Programm 5.6*

Analog dazu werden die Zeichen-Variablen *c* und *d* vereinbart. Bei **cp* denkt sich der Compiler: "Wenn ich im Programm **cp* sehe, dann liefert das einen *char*-Wert. Also muß die Variable *cp* ein Zeichen-Pointer sein". Für *c* und *d* reserviert er genügend Platz, um ein Zeichen zu speichern. Für *cp* jedoch stellt er genügend Platz bereit, um eine Adresse zu speichern.

Der erste Satz ausführbarer Anweisungen macht einfache Wertzuweisung; *i* und *j* haben jetzt einen definierten Wert. Über die anderen Variablen im Programm ist noch nichts bekannt; man kann sich ihren Wert nicht ansehen.

Der zweite Anweisungssatz macht die Pointervariablen *ip* und *cp* zu einem Alias für die Variablen *i* und *j*. Wenn man sich jetzt mit **ip* bzw. **cp* auf indirektem Wege den Wert holt, auf den die Pointer zeigen, dann sieht man, daß dieser mit *i* und *c* geteilt wird.

Im dritten Schritt sehen Sie, daß eine Wertzuweisung an eine Variable auch ihr Alias berührt. Der vierte Schritt macht deutlich, daß Pointer auch rechts vom Zuweisungszeichen auftreten können. Nach Dereferenzierung mit dem *-Operator liefern sie ja einen Wert (wenn man zuvor dafür gesorgt hat, daß Ihnen ein "Wert"-volles Alias beigesellt wurde!). Dieser kann in Zuweisungen auftauchen.

Der nächste Schritt zeigt, daß zwischen *i* und *ip* auf der einen und *j* auf der anderen Seite kein Zusammenhang besteht (gleiches gilt für *c*, *cp* und *d*). Änderungen an *j* lassen *i* und sein Alias unberührt. Auch das Umgekehrte gilt; dies zeigt der letzte Schritt. Er zeigt auch, daß jeder Zuweisungsoperator für die indirekte Wertzuweisung über Pointer verwendet werden kann (im Beispiel: *+=*).

```
#include <stdio.h>
#define ARRAYLEN 80
#define STRLEN ARRAYLEN - 1

main()

{   char string[ARRAYLEN];
    char *cp;
    int len;

    printf(">");
    get_s(string);

    cp = string;
    len = str_len(cp);

    while (len >= 0)
        putchar(cp[len--]);


}

int str_len(s)
    char s[];

{  int i;

   i = 0;

   while (s[i])
      ++i;

   return i;
}

get_s(s)
char s[];

{  char c;
   int i;

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

   s[i] = '\0';

}
```

```
/*************************/
/* Strings sind Zeichen- */
/*  Pointer!             */
/*************************/
/* Alias fuer "string".  */
/*                       */
/*                       */
/*                       */
/*                       */
/*                       */
/* "string" ist ein Poin- */
/*  ter; "cp" ist jetzt  */
/*  sein Alias. Alles, was */
/*  mit "cp" geschieht,  */
/*  bezieht sich jetzt   */
/*  auf "string".        */
/*                       */
/*************************/
```

**Prog. 5.7:** *Arrays sind Pointer*

### 5.2.2 Pointer und Arrays

All dies mag ganz lustig sein. Aber wo liegt der Nutzen? Das soll Ihnen das nächste Programm verdeutlichen. Dies tut bereits Bekanntes: einen String einlesen und ihn rückwärts wieder ausgeben. Es verwendet dazu auch die

bereits bekannten Funktionen *str_len* und *get_s*. Aber es macht den Zusammenhang zwischen Strings und Zeichen-Pointer klar; sehen Sie selbst!

Im ersten Teil des Programmes passiert nichts Neues. In den Zeichen-Array *string* werden von *get_s* Zeichen geschrieben und mit dem unverzichtbaren Null-Byte abgeschlossen.

Doch im zweiten Teil wird die Pointer-Variable *cp* zum Alias von *string* gemacht. Anders als bisher ist dazu der &-Operator nicht nötig, denn eine Array-Variable ist in C nichts anderes als die Adresse des Arrays, genauer: die Adresse der ersten Komponente des Arrays! Als ich versuchte, Ihnen die Parameterübergabe bei den Funktionen *get_s* und *str_len* zu erklären, ist dies bereits angeklungen. Durch seine Deklaration ist *cp* als Zeiger ausgewiesen, kann also rechtmäßig per Wertzuweisung eine Adresse empfangen.

Daß *cp* und *string* jetzt ihre Daten teilen, zeigt der restliche Teil des Programms. Hier wird konsequent mit *cp* weitergearbeitet: zur Bestimmung der Stringlänge ebenso wie zur Ausgabe der Zeichen. Besonders pikant ist, daß *cp* ebenso wie ein Array mit Index angesprochen werden kann, obwohl es eigentlich gar nicht als Array deklariert ist. Das kommt, weil C zwischen Arrays und Pointern keinen Unterschied macht; in C ist beides ein Pointer. Die Array-Schreibweise mit Index benutzt man nur, um mehr traditionell gesinnten Gemütern entgegenzukommen!

Der normale Weg, um indirekt an den Wert eines Pointers heranzukommen, ist jedoch der *-Operator. Das Programm 5.8 ist eine abgeänderte Version von Programm 5.7, in der konsequent alle Pointer mit den ihnen zustehenden Operatoren traktiert werden.

Die erste Konsequenz aus der Gleichsetzung von Arrays und Pointern sehen Sie in den Parameter-Deklarationen der beiden Funktionen *get_s* und *str_len*. Beide erhalten einen String als Argument, Strings aber sind Zeichen-Pointer und genauso sind auch die Parameter deklariert. Wenn Ihnen die Geheimnisse der Deklaration in C noch nicht restlos einleuchten sollten, so trösten Sie sich. Dazu kommt noch ein eigenes Kapitel.

Der C-Programmierer liebt es, seinen Variablen kurze, aber - wie er meint - sprechende Namen zu geben. "Zeichen-Pointer" heißt auf Englisch "character pointer"; deshalb kriegen Variablen dieses Typs, wenn sie keine besonders herausragende Funktion im Programm haben, mit Vorliebe den Namen *cp* verpaßt. Integer-Pointer haben als häufigsten Namen *ip*; alles klar?

```
#include <stdio.h>
#define ARRAYLEN 80
#define STRLEN ARRAYLEN - 1

main()

{   char string[ARRAYLEN];
    char *cp;
    int len;

    printf(">");
    get_s(string);

    cp = string;
    len = str_len(cp);
    cp += len;

    while (--len >= 0)
        putchar(*--cp);


}


int str_len(cp)
    char *cp;


{  int i;

   i = 0;

   while (*cp++)
      ++i;

   return i;
}


get_s(cp)
char *cp;

{  char c;

    while ((c = getchar()) != '\r')
        *cp++ = c;

   *cp = '\0';

}
```

```
/*************************/
/* Pointer-Arithmetik.   */
/*************************/
/*                       */
/*                       */
/*                       */
/*                       */
/*                       */
/*                       */
/*                       */
/*                       */
/* Pointer auf das Ende  */
/*  des Strings setzen.  */
/*                       */
/* Erst wird dekrementiert */
/*  und dann dereferen-  */
/*  ziert.               */
/*************************/

/*************************/
/* Pointer-Version.      */
/*************************/
/*                       */
/*                       */
/*                       */
/*                       */
/*                       */
/*                       */
/* "*cp" liefert Wert;   */
/*  "++" inkrementiert den */
/*  Pointer, der so ueber */
/*  den String wandert;  */
/*  "i" zaehlt mit.      */
/*************************/

/*************************/
/*  Pointer-Version.     */
/*************************/
/*                       */
/* Zu "*cp++ = c":       */
/*  erst wird dereferen- */
/*  ziert ("*cp"), dann  */
/*  zugewiesen und danach */
/*  der Pointer inkremen- */
/*  tiert.               */
/*************************/
```

**Prog. 5.8:** *Elementare Pointer-Arithmetik*

Als erstes will ich mich der Funktion *get_s* genauer zuwenden. Sie erhält (in *main*) als Argument den String *string* , empfängt also im Parameter *cp*

die Adresse der ersten Komponente. Dazu sollten Sie sich die Abbildung 5.4 betrachten.

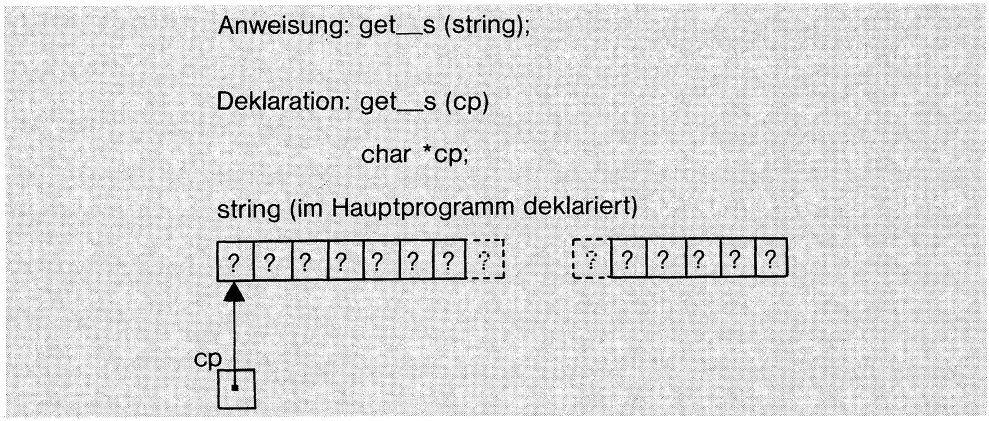



**Abb. 5.4:** *Ein Zeichen-Pointer weist auf einen String.*

Um Sie nicht länger mit numerischen Adressen zu behelligen, habe ich eine in der Informatik übliche graphische Darstellung für Pointer gewählt. Zu Beginn der Abbildung sehen Sie die Anweisung oder Anweisungsgruppe, deren Wirkung dargestellt werden soll. Dann folgt ein vereinfachtes Speicherdiagramm. Immer noch sind die Variablen kleine Kästchen, die für Speicherstellen stehen. Aber um zu symbolisieren, daß eine Pointer-Variable auf eine bestimmte Adresse verweist, schreibe ich nicht länger diese Adresse in die Variable, sondern lasse einfach einen Pfeil auf die Speicherstelle mit der betreffenden Adresse zeigen. So spart man sich die Adressen und die Sache wird übersichtlicher. Zur Übung sehen Sie in Abbildung 5.5 noch einmal die Vorgänge, die bei den Anweisungen

```
int i;        /* 1 */
int *ip;      /* 2 */
i = 5;        /* 3 */
ip = &i;      /* 4 */
```

ablaufen.

In die Begriffe dieser graphischen Darstellung übersetzt bedeutet der Ausdruck **ip* dieses: "gehe dem Pfeil nach, der von *ip* wegführt und nimm den Wert an der Stelle, zu der du so gelangst." Mit **ip* kann man Werte verändern, wenn es links von der Zuweisung steht. Steht es rechts von der Zuweisung, so produziert es den Wert, der sich durch Verfolgen des Pointers findet. Das Programm 5.6 brachte dafür ein Beispiel.

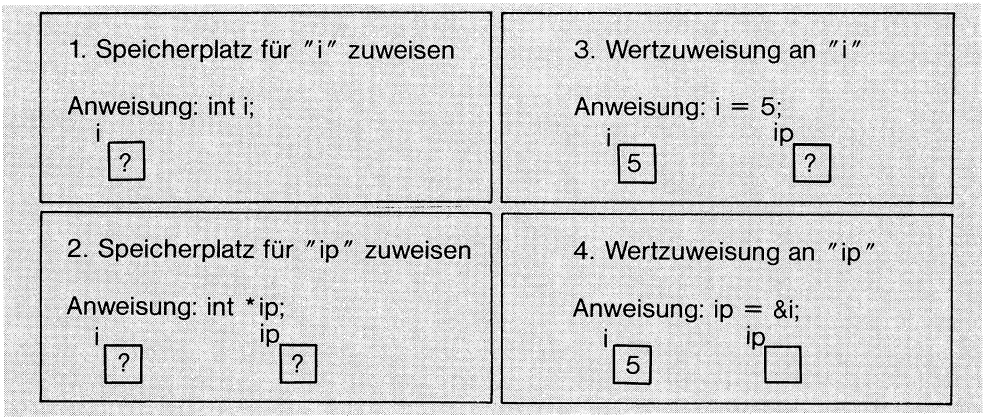



**Abb. 5.5:** *Ein Pointer wird zum Alias einer Integer-Variablen.*

Jetzt aber zurück zur Funktion *get_s*. Nach ihrem Aufruf, ehe die erste ausführbare Anweisung angegangen wird, herrschen im Computer die Verhältnisse, die die Abbildung 5.4 symbolisiert. Der Array, auf den *cp* zeigt, hat noch keine Werte erhalten, sein Inhalt ist deshalb undefiniert. Aus dem eben Gesagten geht hervor, daß man mit **cp* in die erste Komponente einen Wert schreiben kann. Der Ausdruck:

```
*cp = 'A'
```

hätte auf die Situation in der Abbildung 5.4 die Auswirkungen, die die Abbildung 5.6 wiederspiegelt. In die erste Komponente des Arrays bekommt man also einen Wert. Wie aber kommt man an die nächste Komponente heran?

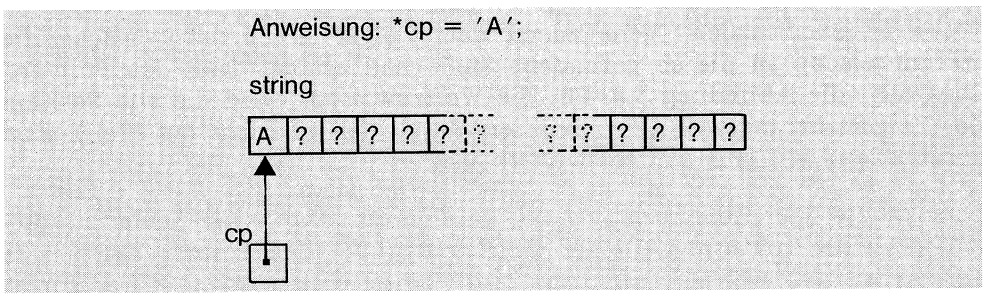



**Abb. 5.6:** *Wertzuweisung durch Dereferenzieren.*

Dazu muß man sich der Tatsache erinnern, daß Adressen Zahlen und daß Pointer Variablen sind. Ist in einer Variablen eine Zahl gespeichert, dann kann man diese Variable auch inkrementieren. Ist die Variable ein Pointer... Ja, was passiert dann? Die Abbildung 5.7 zeigt es: der Zeiger ist um

eine Position weitergewandert! Der gestrichelte Pfeil erinnert an die Verhältnisse vor Ausführung der Inkrementierung.

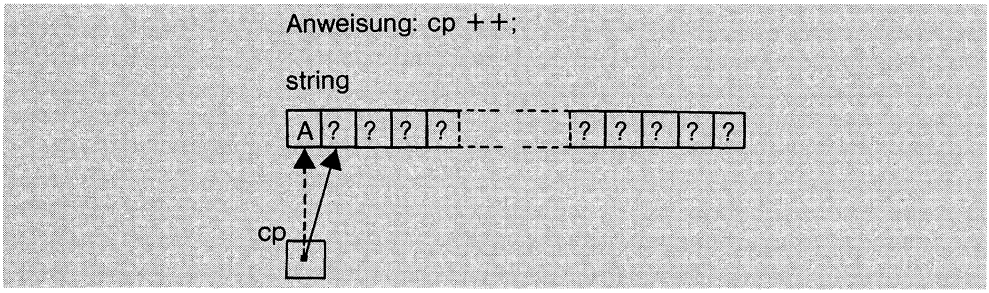



**Abb. 5.7:** *Inkrementieren eines Pointer.*

Nun fehlt nur noch ein Schritt: die Integration von Wertzuweisung und Inkrementieren. Um eine Wert - z.B. das Zeichen 'B' - indirekt (über Pointer) zu speichern und diesen Pointer dann zu erhöhen, könnte man also schreiben:

```
*cp = 'B';
cp++;
```

Aber für C ist typisch, daß es noch kürzer geht. Anstelle der obigen beiden Ausdrücke schreibt der versierte C-Programmierer nämlich einfach

```
*cp++ = 'B';
```

Wenn Sie sich die Tabelle der Bindungsstärken aus dem letzten Kapitel noch einmal ansehen, dann werden Sie feststellen, daß der Operator * höhere Präzedenz als der Operator *++* hat. Deswegen liest C den Ausdruck **cp++* folgendermaßen: "gehe zuerst dem Zeiger in *cp* nach, mache die Wertzuweisung an die so gefundene Stelle und erhöhe dann den Pointer." Dies zeigt die Abbildung 5.8. Für die Wertzuweisung wird die alte Position des Zeigers herangezogen (symbolisiert durch den gestrichelten Pfeil). Erst anschließend wird der Pointer weiterbewegt.

Nun dürfte klar sein, wie *get_s* funktioniert. Die *while*-Schleife kontrolliert eine Anweisung, in der gleichzeitig über den Pointer Zeichen in den Array geschrieben werden und der Pointer weiterbewegt wird. Dieses Weiterschieben des Pointers geschieht nach jeder Zuweisung. Ist die Schleife beendet, zeigt er folglich hinter das letzte eingelesene Zeichen. Nun muß nur noch durch einfache Dereferenzierung dorthin das Null-Byte geschrieben werden, das den String abschließt.

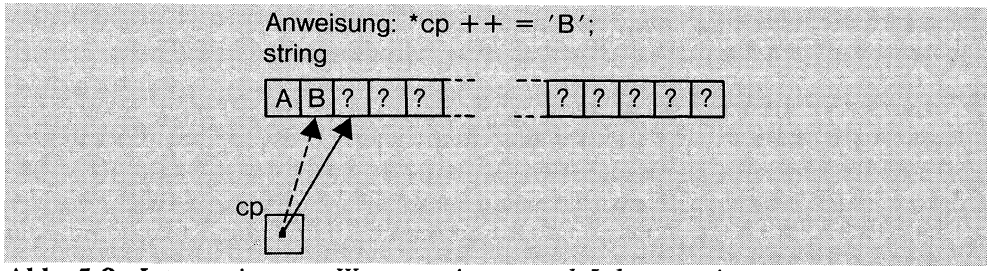



**Abb. 5.8:** *Integration von Wertzuweisung und Inkrementieren*

Auch die Mechanik von *str_len* sollte nun klar sein. Diesmal taucht das **cp++* in der Kontrollbedingung der *while*-Schleife auf. Da es hier nicht zur Zuweisung benutzt wird, liefert es nur den Wert, auf den der Pointer gerade zeigt und inkrementiert den Pointer dann. Dieser Wert wird irgendwann einmal - wenn nur *cp* weit genug verschoben worden ist - das Null-Byte sein, welches den String abschließt. Dann aber bricht die Schleife ab. In ihr wird mittels *++i* mitgezählt, wie oft dieses Verschieben möglich war. Dies ergibt genau die gesuchte String-Länge.

Bleibt nur noch ein Punkt zu klären. Zeiger können nicht nur in Einerschritten inkrementiert (und dekrementiert) werden. Auch die Erhöhung um größere Beträge ist möglich. Möchte man z.B. in einem String nur jeden zehnten, zwanzigsten, dreißigsten usw. Buchstaben sehen und zeigt *cp* auf den Anfang dieses Strings, dann kann man mit

```
cp += 10;
```

eine Verschiebung des Zeigers um 10 Positionen erreichen. Hier wird genau genommen mit Adressen gerechnet, weswegen man in diesem Zusammenhang auch von "Adressarithmetik" bzw. "Pointer-Arithmetik" spricht; darüber aber später noch mehr.

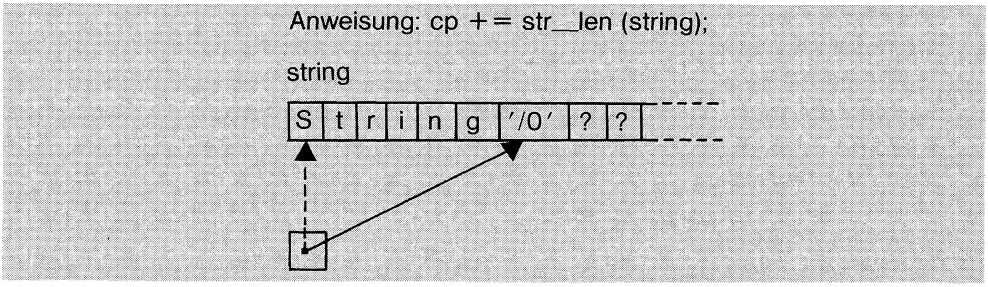



**Abb. 5.9:** *Adreßarithmetik mit Pointern*

Wenn ich nun einen Zeiger auf den Anfang eines Strings habe und ihn in einem einzigen Schritt an das Ende des Strings bewegen will, was mache

ich dann? Sehr richtig: ich besorge mir die String-Länge und addiere diesen Wert auf den Pointer! Dazu noch eine Abbildung (Abb. 5.9).

In dieser Abbildung ist in der Variablen *string* bereits ein String gespeichert (nämlich "String"; einfallslos, nicht wahr?). Dieser hat die Länge 6 und das ist auch der Wert, den der Aufruf *str_len(string)* liefert. Die Abbildung zeigt nun, was passiert, wenn man diesen Wert auf die Pointer-Variable *cp* addiert: der Zeiger weist hinter das Ende des Strings (auf das Null-Byte).

Das Programm 5.8 bewahrt sich die von *str_len* gelieferte Länge noch zusätzlich in der Variablen *len* auf. Es benötigt sie, um das Rückführen des Pointers für die umgekehrte Zeichenausgabe zu überwachen. Denn wenn man den String von hinten nach vorne durchgeht, dann kommt nicht das rettende Null-Byte, an dem man das Ende der Ausgabe (hier: den Anfang des Strings) erkennen kann.

Zuvor jedoch gilt es eins zu bedenken: der Zeiger weist hinter das Ende. Ehe man durch Dereferenzieren auf ein String-Zeichen zugreift, muß man den Pointer erst noch zurücksetzen. Da kommt folgende Anweisung gerade recht:

```
*--cp;
```

Das vorangestellte -- dekrementiert zuerst den Pointer; dann erst wird dereferenziert. Somit ist wieder alles in Ordnung. Um nicht zu weit zurückzugehen, wird auch die Zählvariable *len* mit Prädekrement herabgezählt. Hat sie den Wert 0, dann bricht die Schleife ab.

Es scheint so, als ob Sie jetzt alle Kombinationen von *, *cp*, *++* und *--* durchhaben. Aber weit gefehlt! Was könnte z.B. folgende Anweisungssequenz bewirken?

```
char c, *cp;
cp = &c;
c = 'B';
--*cp;
```

Sehen Sie sich die Abbildung 5.10 an. Deren Unterschrift sagt es schon: obwohl die Anweisung *--*cp* wie Adreßarithmetik aussieht, wird hier tatsächlich der Wert manipuliert, auf den *cp* zeigt. Denn die Anweisung ist zu lesen, als wäre sie so geklammert:

```
--(*cp);
```

Das kommt von der Assoziativität der einstelligen Operatoren (* und -- sind beides einstellige Operatoren), die von rechts nach links gruppiert

werden. Zuerst kommt die Dereferenzierung; sie liefert das Zeichen 'B'. Dieses wird dann dekrementiert, es stellt sich also das Ergebnis 'A' ein. Natürlich hätte man das auch billiger haben können; mit *--c* etwa; aber was tut man nicht alles, um C zu lernen!

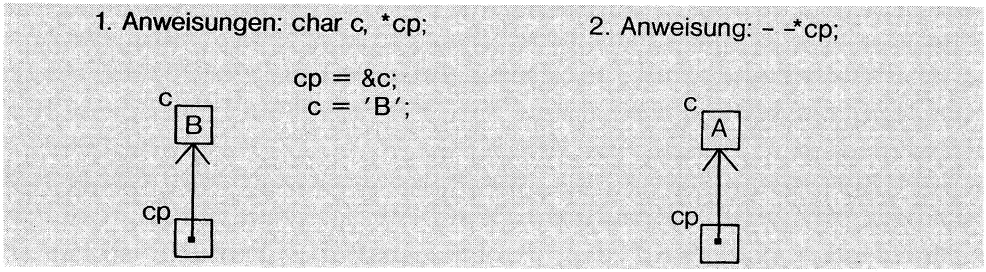



**Abb. 5.10:** *Sieht aus wie Adreßarithmetik, ist aber keine!*

## 5.3 Adreßarithmetik

An *str_len* kann man erkennen, worin der Vorteil des abschließenden Null-Bytes bei Strings liegt. Dadurch ist es möglich, die Abarbeitung des Strings in die Kontrollbedingung einer Schleife zu integrieren: man verschiebt einen Pointer so lange über die String-Zeichen, bis dieser einmal auf das Null-Byte zeigt und so (für C) den Wahrheitswert "falsch" liefert. Verwechseln Sie übrigens nicht ein Null-Byte mit der Ziffer '0'. Letztere hat in der ASCII-Tabelle die Position 48 und das ist auch der Zahlenwert, mit dem diese Ziffer in C gehandelt wird. Das Null-Byte können Sie als Zeichenkonstante nur mit dem Escape-Mechanismus notieren. Hier noch einmal beide Konstanten:

```
'\0'    /* Null-Byte fuer String-Abschluß */
'0'     /* Zeichenkonstante */
```

In jedem ernstzunehmenden C-System ist eine Funktion zur Bestimmung der Stringlänge Bestandteil der Standard-Bibliothek. Nur ist sie da vermutlich anders programmiert, als ich es Ihnen gezeigt habe. Denn obwohl die letzte Version - die Pointer-Version - schon ganz schön C-typisch ist: es geht noch um eine Spur professioneller. Die Profi-Version sehen Sie im Programmbeispiel 5.9.

Diese Version funktioniert wie folgt (vgl. Abb. 5.11): bei Eintritt in die Funktion zeigt der Parameter *s* auf den Anfang des Strings, dessen Länge zu bestimmen ist. In der Abbildung beginnt der String an der (fiktiven) Adresse 500, dies ist also auch der Wert des Parameters *s* bei Funktions-

eintritt. Eine zweite Pointer-Variable *cp* wird (in der Deklaration!) ebenfalls auf den Stringanfang gesetzt und dann solange weitergeschoben, bis das Null-Byte erreicht ist. Die Differenz zwischen den beiden pointern ist jetzt um Eins höher als die gesuchte Stringlänge!

```
strlen (s)
        char *s;

{   char *cp = s;

    while (*cp++)
        ;

    return (cp - s - 1);
}
```

**Prog. 5.9:** *Subtraktion von Pointern*

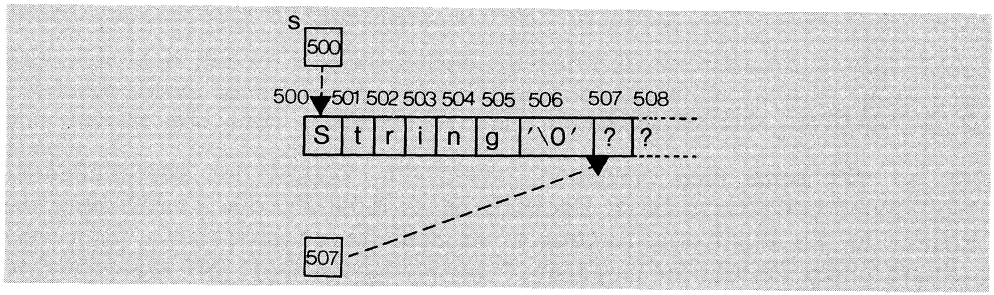



**Abb. 5.11:** *Längenberechnung von Strings durch Adreß-Subtraktion*

Warum? In der *while*-Schleife, die den Pointer verschiebt, wird zuerst das Zeichen betrachtet, auf das der Pointer verweist und anschließend dieser inkrementiert. Ist das Null-Byte erreicht, dann wird also trotzdem noch einmal verschoben; deshalb muß von der Adreßdifferenz noch 1 subtrahiert werden. Dies funktioniert allerdings nur, weil Strings Arrays sind und die Komponenten eines Arrays in aufeinanderfolgenden Speicherzellen untergebracht werden.

Es mag den Anschein haben, als könne der C-Programmierer Pointer (und damit Adressen) in beliebiger Weise manipulieren. Da jedoch nur einige Formen der Adreßarithmetik in einem Programm sinnvoll sind, läßt der Compiler nicht alles zu, was denkbar ist. Die erlaubten Möglichkeiten sind:

Addieren einer Integer-Konstante zu einem Pointer
Subtrahieren einer Integer-Konstante von einem Pointer
Subtrahieren zweier Pointer
Vergleiche zweier Pointer

Alle anderen Operationen sind untersagt. Weder dürfen zwei Pointer addiert werden, noch dürfen sie multipliziert und dividiert werden. Auch sind alle bitweisen logischen Operationen mit Pointern (jedoch nicht mit den Werten, auf die sie verweisen!) untersagt.

Arrays können natürlich nicht nur mit Integern und Zeichen gebildet werden. Jeder andere Datentyp von C ist erlaubt und somit sind auch Arrays aus Gleitpunktzahlen oder aus *long*-Integers möglich.

Nun ist die Pointer-Arithmetik eine sehr maschinennahe Angelegenheit; man könnte daher denken, daß zu ihrer Beherrschung genaue Kenntnisse über die verschiedenen Formen der Datenspeicherung im Computer nötig sind. Auf vielen Computern, so auch beim Prozessor des ATARI, benötigen die meisten elementaren C-Datentypen zu ihrer Speicherung mehr als eine Speicherzelle. Der Motorola 68000, der im ATARI die Denkarbeit erledigt, arbeitet zwar intern mit 32-Bit-Registern. Sein Verkehr mit dem Speicher erfolgt jedoch in kleineren Happen: er hat 'nur' einen 16-Bit-Datenbus, kann also auf einmal nur ein 16 Bit langes Wort in den Speicher transportieren. Dennoch wird der Speicher nicht in 16-Bit-Worten adressiert; was den Speicher betrifft, ist der ATARI eine Byte-Maschine: die Speicherzellen sind Byte-weise durchnumeriert.

Das bedeutet, daß im Arbeitsspeicher die Komponenten eines Zeichen-Arrays an aufeinanderfolgenden Adressen liegen (denn ein Zeichen benötigt ein Byte). Integers beanspruchen zwei Bytes oder ein Wort, *long*-Integers werden in einem 32 Bit großen Langwort untergebracht und somit über vier Speicheradressen verteilt. Um nun einen Pointer über einen Integer-Array zu verschieben, muß der numerische Wert der Zeiger-Variablen jedesmal um den Wert zwei verändert werden; bei einem *long*-Array entsprechend um den Wert vier.

Aber keine Angst! All dies erfolgt automatisch, ohne daß Sie sich darum zu kümmern brauchen. Der C-Compiler weiß nämlich aus der Deklaration einer Pointer-Variablen, auf welchen Datentyp sie verweist. Außerdem kennt er die Länge jedes Datentyps in Bytes; er kann also die nötigen Anpassungen vornehmen, ohne daß Sie etwas davon bemerken. Wenn Sie einen Zeichen-Pointer inkrementieren, dann wird dieser tatsächlich nur um 1 hochgezählt; inkrementieren Sie jedoch einen Pointer auf eine *long*-Integer, dann erfolgt in Wirklichkeit eine Erhöhung des Zahlenwertes um 4. Auch bei Adreß-Arithmetik wie etwa der Pointer-Subtraktion werden die entstehenden Werte vom Compiler automatisch normiert.

Lassen Sie - durch einen Fehler im Programm oder infolge eines nicht ganz jugendfreien Programmiertricks - einen Integer-Pointer auf einen String zeigen und schieben Sie ihn dann durch Inkrementieren über den String,

dann erhalten Sie folglich nur das erste, dritte, fünfte... Zeichen des Strings!

### 5.3.1 Zusammenhang von Arrays und Pointern

C bietet zwar die übliche Array-Notation (mit Index in eckigen Klammern), arbeitet intern aber ausschließlich mit Pointern. Angenommen, Sie haben einen Integer-Array *ia* und eine Integer-Variable *i* wie folgt definiert:

```
int i, ia[50];
```

Die normalen Zugriffe auf Komponenten über Index übersetzt sich der Compiler dann wie folgt in seine eigene Denkweise:

| Array-Notation | Dasselbe mit Pointer: |
|---|---|
| `i = ia[0];` | `i = *ia;` |
| `ia[0] = 5;` | `*ia = 5;` |
| `i = ia[3];` | `i = *(ia + 3);` |
| `ia[9]++;` | `*(ia + 9)++;` |

Jetzt dürfte auch klarer sein, warum C darauf besteht, seine Arrays mit 0 beginnen zu lassen: die Umrechnung von Index-Schreibweise in Pointer-Arithmetik ist dadurch unmittelbar möglich.

Beachten Sie die Klammerung im dritten und vierten Beispiel. Es wäre falsch, **ia + 3* bzw. **ia + 9* zu schreiben. Dies hätte die Wirkung, auf die erste Komponente des Arrays den Wert 3 bzw. 9 zu addieren; keinesfalls aber wird damit die vierte bzw. zehnte Komponente (wie beabsichtigt) angesprochen! Der einstellige *-Operator zum Dereferenzieren bindet eben stärker als die Addition.

Um einen Pointer auf den Anfang eines Arrays zu setzten, brauchen Sie nur den Array dem Pointer zuweisen; denn Array-Namen sind ja in C Zeiger auf das erste Array-Element. Ebenso ist es aber auch möglich, einen Pointer an eine beliebige andere Stelle im Array zeigen zu lassen:

```
int ia[50], *ip;

ip = ia;
ip = &(ia[20]);
```

Die erste Anweisung setzt den Integer-Pointer *ip* auf das erste Element des Integer-Arrays *ia*. Manchmal ist es jedoch praktisch, einen Zeiger mitten in einen Array zu setzten. Natürlich könnte man dazu wie in der erste Anweisung diesen zuerst auf den Anfang zeigen lassen und ihn dann entsprechend inkrementieren. Eine andere, direktere Möglichkeit zeigt jedoch die

zweite Anweisung. Sie setzt den Pointer *ip* auf das einundzwanzigste Array-Element: mit *ia[20]* wird dieses Element ausgewählt; der *&*-Operator besorgt dessen Adresse, die dann an den Pointer *ip* zugewiesen werden kann.

Eine Möglichkeit, die Sie von BASIC oder auch Pascal eventuell gewohnt sind, ist in C nicht gegeben: die Zuweisung an einen Array. BASIC erlaubt Ihnen etwa Folgendes:

```
10 A$ = "Das ist ein String!"
```

String-Variablen können also in BASIC auf der linken Seite einer Zuweisung auftauchen. Das geht in C nicht. Es ist also nicht erlaubt (und wird auch vom Compiler als Fehler erkannt), zu schreiben:

```
char strng[50];
.
.
.
strng = "Das ist ein String!";
```

C behandelt Arrays, wie es *strng* einer ist, in dieser Hinsicht wie Konstanten. Etwas anderes ist die Zuweisung von Einzelzeichen an Array-Komponenten. Daß

```
strng[0] = 'D';
strng[1] = 'a';
.
.
.
```

zulässig ist, haben Sie bereits gesehen. In diesem Punkt herrscht bei manchen C-Programmierern manchmal Verwirrung. Deshalb noch einmal die wichtigsten Tatsachen über Arrays in der Zusammenfassung:

1. Bei der Deklaration eines Arrays reserviert C genügend zusammenhängenden Speicherplatz, um die bei der Deklaration (in den eckigen Klammern) angegebene Anzahl an Komponenten unterzubringen.
2. Der Array-Name ist nichts anderes als ein Zeiger auf die erste Komponente. Deshalb können Arrays an Pointer vom passenden Typ zugewiesen werden.
3. Ansonsten sind Array-Namen für C Konstanten; sie dürfen niemals in einer Zuweisung als *lvalue* (also links vom Zuweisungs-Operator) auftauchen.

Einen String an einen anderen zuweisen, das ist in BASIC ganz einfach:

```
10 A$ = "Das ist ein String"
20 B$ = A$
```

Damit stehen im BASIC-String B$ die gleichen Zeichen wie im String A$. Das geht so in C nicht! Um den Inhalt eines Strings in einen anderen String zu bekommen, müssen Sie eine Funktion bemühen, die zeichenweise den Inhalt des einen Arrays in den anderen kopiert! Betrachten Sie dazu das Programm 5.10.

```
#include <stdio.h>

main()

{   char str_1[80], str_2[80];

    printf("\nBitte String 1 eingeben!\n>");
    gets(str_1);
    str_copy(str_2, str_1);
    printf("\nInhalt von String 2: %s", str_2);
}



str_copy (ziel, quelle)
        char *ziel;
        char *quelle;

{
    while (*ziel++ = *quelle++)
        ;
}
```

```
/***************************/
/* In C ist keine Zuwei-   */
/*  sung an Strings moeg-  */
/*  lich!                  */
/***************************/
/*                         */
/* Funktion aus der Stan-  */
/*  dard-Bibliothek.       */
/*                         */
/***************************/


/***************************/
/* Strings kopieren.       */
/***************************/
/*                         */
/*                         */
/* Wie ueblich: Zuweisung  */
/*  und Test integriert!   */
/*                         */
/***************************/
```

**Prog. 5.10:** *'Zuweisen' von Strings*

Das Hauptprogramm benutzt zum Einlesen des ersten Strings von der Tastatur die Funktion *gets*. Diese ist in der Standard-Bibliothek jedes besseren Compilers enthalten; sehen Sie dazu einmal in Ihrem Handbuch nach. Sollte sie fehlen, oder sollte sie nicht wie gewünscht funktionieren (wie es z.B. bei frühen Versionen des ATARI-C-Compilers der Fall ist), dann greifen Sie bitte auf die in den letzten Beispielen verwendete Funktion *get_s* zurück.

Der Zeichen-Array *str_1* wird somit über Tastatureingaben mit einem Wert versehen. Um diesen String auch an *str_2* weiterzugeben, wird eigens eine Funktion benötigt, die zeichenweise überträgt. Es hat sich eingebürgert, den Ziel-Array (in den übertragen werden soll) der Funktion als erstes Argument zu übergeben. Der Quell-Array, aus dem die Zeichen kommen, ist zweites Argument. Das Hauptprogramm überprüft dann lediglich, ob die Kopiererei auch funktioniert hat; es ist nur ein Testrahmen für die Funktion *str_copy*, um die sich eigentlich alles dreht.

Diese macht einen etwas unscheinbaren Eindruck, besteht sie doch im wesentlichen aus nur einer Zeile. Dabei demonstriert sie in augenfälliger Weise, wie klar man sich in C ausdrücken kann!

Die Funktion empfängt die Anfangsadresse der beiden Strings in ihren Parametern *quelle* und *ziel*. Die Einzelheiten des Kopiervorgangs werden wohl am schnellsten klar, wenn ich sie zuerst einmal in 'Langform' hinschreiben. Hier also eine Version von *str_copy*, die jeden Schritt einzeln aufführt:

```
                                        /****************************/
str_copy (ziel, quelle)                 /* Kopieren von Strings:    */
        char *ziel;                     /*  Langversion.            */
        char *quelle;                   /****************************/
                                        /*                          */
                                        /*                          */
{   char c;                             /*                          */
                                        /*                          */
    while (*quelle != '\0')             /* Bis zum Stringende...    */
    {   c = *quelle;                    /*  Zeichen aus Quelle ho-  */
        *ziel = c;                      /*  len und in Ziel schrei- */
        ++quelle;                       /*  ben; Zeiger weiter-     */
        ++ziel;                         /*  schieben.               */
    }                                   /*                          */
                                        /* Zuletzt Ziel-String ord- */
    *ziel = '\0';                       /*  nungsgemaess abschlies- */
                                        /*  sen.                    */
}                                       /****************************/
```

Das Vorgehen ist klar: die beiden Pointer werden synchron über die Strings verschoben. Das Zeichen, auf das der *quelle*-Zeiger zeigt, wird geholt (durch Dereferenzieren) und dann an die Stelle geschrieben, an die der *ziel*-Zeiger verweist. Dies geht so lange, bis im Quell-String das Null-Byte erreicht ist, dieser also vollständig abgearbeitet wurde. Abschließend muß die Funktion noch für korrekte Beendigung des *ziel*-Strings sorgen.

Nun kann man das Übertragen der Einzelzeichen ohne Hilfe eines Zwischenspeichers *c* bewerkstelligen, indem man einfach schreibt:

```
*ziel = *quelle;
```

Aber - Sie haben es bereits mehrfach gesehen: auch das Inkrementieren der beiden Pointer kann in diese Anweisung integriert werden. Da zuerst kopiert und dann Inkrementiert werden muß, geht das in der Form

```
*ziel++ = *quelle++;
```

Dieser Ausdruck hat einen Wert (denn es ist eine Zuweisung und Zuweisungen haben einen Wert; jaja, ich weiß, ich wiederhole mich...). Sein Wert ist das soeben übertragene Zeichen. Handelt es sich dabei um das Null-

Byte, dann ist dieser Wert für C zugleich der Wahrheitswert "falsch". Deshalb kann man den gesamten Ausdruck auch gleich als Kontrollbedingung in die *while*-Schleife stecken! Wegen der Eigenschaften des Postinkrements ist gewährleistet, daß auch noch das Null-Byte übertragen wird, ehe die Schleife abgebrochen wird. Damit gelangt man zu der hochintegrierten Version des Programms 5.10.

Auch die Funktion zum Kopieren von Strings ist Bestandteil der Standard-Bibliothek; sie heißt dort *strcopy*.

## 5.4 Mehrdimensionale Arrays

Aus anderen Sprachen kennen Sie vielleicht die Möglichkeit, in Tabellenform angeordnete Daten in mehrdimensionalen Arrays zu speichern. Trotz seiner Bevorzugung der Pointer verzichtet natürlich auch C nicht darauf, insbesondere, da für viele mathematische und naturwissenschaftliche Aufgabenstellungen zwei- und mehrdimensionale Arrays unerläßlich sind.

Um einen Integer-Array mit 5 Reihen zu jeweils 10 Spalten zu deklarieren, scheiben Sie

```
int zwei_dim [5] [10];
```

Die Dimensions-Angabe über die Reihen kommt also vor der Angabe über die Spalten. Anders als z.B. in Pascal müssen Sie für jede Angabe ein eigenes Paar eckiger Klammern schreiben. Die Wirkung dieser Deklaration: der Compiler reserviert im Speicher einen zusammenhängenden Bereich, der 50 Variable vom Typ "Integer" aufnehmen kann. Da eine Integer 2 Byte lang ist, werden dafür 100 Byte geopfert. Sich diesen Bereich als zweidimensionale Matrix vorzustellen ist jedoch nur eine Eselsbrücke für den Programmierer: im Speicher ist alles hintereinander angeordnet. Erst kommen die Speicherzellen für die erste Spalte, dann die für die zweite Spalte usw. Dies zeigt die Abbildung 5.12 für einen Array mit 3 Zeilen und 4 Spalten. In den Speicherzellen stehen in eckigen Klammern jeweils die Indizes für die einzelnen Komponenten.

Um sich auf das dritte Element in der fünften Reihe des Arrays *zwei_dim* zu beziehen und ihm den Wert 99 zuzuweisen, schreiben Sie:

```
zwei_dim [4] [2] = 99;
```

(Denken Sie daran, daß die Zählung der Array-Komponenten in C bei Null beginnt!) Nach diesem Muster sind auch Arrays mit mehr als zwei Dimensionen möglich.

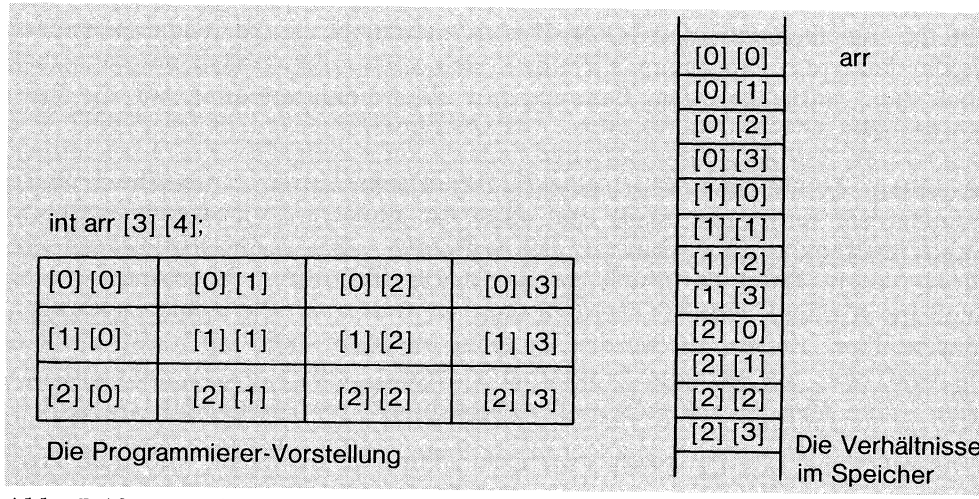



**Abb. 5.12:** *Speicherung zweidimensionaler Arrays*

Soll eine Funktion mit einem mehrdimensionalen Array als Parameter arbeiten, dann müssen Sie C Angaben über die Zeilen-Ausmaße des Arguments machen. Eine Funktion, die den Beispiels-Array *zwei_dim* als Argument verarbeiten kann, muß so deklariert werden:

```
array_fun (tab)
int tab[] [10];

    .
    .
    .
```

Der Parameter *tab* (für "Tabelle") muß also mit Angaben über die Länge einer Tabellen-Reihe deklariert werden. Dies braucht der Compiler deswegen, weil er ja für seine eigenen Zwecke Zugriffe auf Array-Komponenten über Indizes wieder in Pointer-Arithmetik umwandelt. Wegen der reihenweisen Speicherung mehrdimensionaler Arrays (vgl. Abb. 5.12) werden Angaben über die Reihen-Länge benötigt, um die korrekten Versatzwerte für die Zeiger-Manipulation zu erhalten. Die Anzahl der Reihen ist für den Compiler an dieser Stelle nicht relevant; er würde sie höchstens dann benötigen, wenn er sich um die Einhaltung von Indexgrenzen zu kümmern hätte. Das aber tut er bekanntlich nicht.

## 5.5 Pointer-Arrays

Gutgeschriebene Programme stürzen im Fehlerfall nicht sang- und klanglos ab; vor ihrem Ableben hauchen Sie vielmehr noch eine Fehlermeldung aus,

auf daß der Bediener den bösen Fehler nicht noch einmal begehe! Andere Fehler kann das Programm abfangen, sich wieder von ihnen erholen; aber auch dann sollte man dem Benutzer mit einer Fehlermeldung auf die Finger klopfen.

In diesem Zusammenhang ist eine Funktion sehr nützlich, die eine Fehlernummer als Argument erhält und dann die dieser Nummer entsprechende Fehlermeldung ausgibt. Daß mit Fehlernummern gearbeitet wird, zeugt von Profi-Arbeit. Denn der Profi rechnet damit, daß seine Programme später auch ins Ausland verkauft werden; dann muß er aber alle Fehlermeldungen umschreiben. Besser, sie werden an einer einzigen Stelle verwaltet, als daß sie weit über das gesamte Programm verstreut immer wieder mal auftauchen - und man bei späteren Änderungen immer mal wieder eine vergißt.

Eine mögliche Lösung dieses Problems könnte so aussehen, wie dies Programm 5.11 zeigt. Die Auswahl der Fehlermeldungen erfolgt in einer *switch*-Anweisung; die einzelnen Fehlermeldungen sind als Stringkonstanten in der Funktion *pr_error* festgeschrieben.

```
pr_error(n)
        int n;


{  switch (n)
    {   case 1:  printf("\nDas sollten Sie nicht tun!");
                 break;
        case 2:  printf("\nDas verstehe ich auch nicht!");
                 break;
        case 3:  printf("\nWas ist los...?!?");
                 break;
        case 4:  printf("\nJetzt aber Schluss mit dem Bloedsinn!");
                 break;
        case 5:  printf("\nDiesen Fehler gibt's gar nicht!");
                 break;
        case 6:  printf("\nFehler Nr. 99");
                 break;
        default: printf("\nFehler: unbekannter Fehler.");
                 break;
    }
}
```

**Prog. 5.11:** *Ausgabe von Fehlermeldungen*

Dies ist zwar eine Lösung des Problems; aber sie macht einen ziemlich unbeholfenen, nicht sehr eleganten Eindruck. So etwas wurmt den engagierten Programmierer!

Erinnern wir uns an die Arrays: eigentlich wäre es ja naheliegender, in Analogie dazu mit einer Tabelle von Fehlermeldungen zu arbeiten. Die

Fehlernummer *n* dient bei dieser Lösung als Index, der aus dieser Tabelle die passende Fehlernummer aussucht.

Wie jedoch erhält der Programmierer in C eine solche Tabelle mit Fehlermeldungen? Eine Möglichkeit, an die man nach der Lektüre des letzten Kapitels denken könnte, ist ein zweidimensionaler Zeichen-Array. Wenn man zehn Fehlermeldungen zu verwalten hat und die längste Meldung 79 Zeichen lang ist (mehr hat in einer Bildschirmzeile nicht Platz!), dann könnte man folgende Variable deklarieren:

```
char fehler[10] [80];
```

Die Zeilen dieser Tabelle dienen zur Aufnahme der einzelnen Fehlermeldungen als Strings. Die Zeilenlänge muß mit 80 Zeichen angegeben werden, da ja auch noch das abschließende Null-Byte Platz finden soll!

Aber diese Lösung ist auch nicht ideal. Denn in dem zweidimensionalen Array werden jetzt für jede Meldung 80 Zeichen reserviert, auch wenn sie nur 10 Zeichen lang ist. Dies ist eine unnötige Platzverschwendung, über die man aber bei der enormen Speicherkapazität des ATARI großzügig hinwegsehen könnte.

Ein anderes Problem ist jedoch hartnäckiger: wie bekommt man die Fehlertexte in diesen Array? Weil Arrays - was die Zuweisung betrifft - wie Konstanten behandelt werden, können Sie nicht einfach schreiben:

```
fehler[0] = "\nDas sollten Sie nicht tun!";
fehler[1] = "\nDas verstehe ich auch nicht!";
.
.
.
```

Das erlaubt der Compiler nicht. Die einzige Möglichkeit, den Array *fehler* mit Anfangswerten zu versorgen (zu "initialisieren", wie man auch sagt) besteht darin, wieder die Funktion *strcopy* zu bemühen. Das sieht so aus:

```
strcpy(&(fehler[0] [0]), "\nDas sollten Sie nicht tun!");
strcpy(&(fehler[1] [0]), "\nDas verstehe ich auch nicht!");
.
.
.
```

Mit *fehler[0] [0]* wird das erste Zeichen der ersten Tabellen-Zeile angesprochen. Die Funktion *strcopy* möchte jedoch Strings als Argumente, also Zeichen-Pointer. Deshalb muß man sich mit dem &-Operator die Adresse des über Indexierung angesprochenen Zeichens besorgen. Die gleiche Wirkung hat es aber auch, einfach *fehler[0]*, *fehler[1]* etc. zu schreiben. Dies ergibt sich aus dem Zusammenhang zwischen Arrays und Pointern. Um

jetzt die dritte Meldung in dem *fehler*-Array auszugeben, schreiben Sie einfach:

```
printf(fehler[2]);
```

Das Ausgeben der Fehlermeldungen ist jetzt also unproblematisch und gegenüber der *switch*-Version wesentlich vereinfacht. Aber ein großes Problem bleibt: das Initialisieren des Arrays. So, wie die Dinge jetzt stehen, müssen Sie nämlich die Initialisierung durch wiederholte *strcopy*-Aufrufe in die Funktion *pr_error* mit aufnehmen; die Abbildung 5.13 zeigt dies andeutungsweise.

```
pr_error(n)
        int n;

{   char error[10] [80];


    strcpy(fehler[0], "\nDas sollten Sie nicht tun!");
    strcpy(fehler[1], "\nDas verstehe ich auch nicht!");
    .
    .
    printf(fehler[n]);
}
```

**Abb. 5.13:** *Probleme mit der Initialisierung*

```
                                        /*************************/
char *Fehler[] =                        /* Array mit Pointern ini- */
{  "\nDas sollten Sie nicht tun!",      /*  tialisieren.           */
   "\nDas verstehe ich auch nicht!",    /* Einige durch und durch  */
   "\nWas ist los...?!?",               /*  realistische Fehler-   */
   "\nJetzt aber Schluss mit dem Bloedsinn!",  /*  meldungen...     */
   "\nDiesen Fehler gibt's gar nicht!", /*                         */
   "\nFehler Nr. 99",                   /*************************/
} ;
                                        /*************************/
                                        /* Gib alle Fehlermeldun-  */
main()                                  /*  gen aus; bestimme ih-  */
                                        /*  re Anzahl.             */
                                        /*************************/
{  char **cpp;                          /* Alias fuer den Mel-     */
   int i, nr_errors;                    /*  dungs-Array.           */
                                        /*                         */
                                        /*                         */
   nr_errors = (sizeof(Fehler) / sizeof (char *));/* Anzahl der Meldungen  */
                                        /*  berechnen.             */
   cpp = Fehler;                        /* Pointer auf Anfang des  */
                                        /*  Arrays setzen.         */
   for (i = 0; i < nr_errors; ++i)      /*                         */
      printf(*cpp++);                   /* Ausgeben.               */
                                        /*                         */
}                                       /*************************/
```

**Prog. 5.12:** *Ein Pointer-Array.*

Das bedeutet: jedesmal, wenn eine Fehlermeldung ausgegeben werden soll, wird zuerst der gesamte Arrays mit den Meldungen beschrieben und dann die geeignete ausgewählt! Dieses Verfahren ist gänzlich untragbar.

Es gibt jedoch eine Lösung, die alle Probleme beseitigt. Sie sehen sie im Programm 5.12.

In diesem Programm-Beispiel sind viele Neuigkeiten enthalten, die später noch genauer besprochen werden. Die erste Neuigkeit betrifft die Initialisierung von Variablen. Dazu sollten sie sich noch einmal einige elementare Tatsachen über Strings und Pointer ins Gedächtnis rufen.

Pointer sind Variablen; Variablen kann - im Unterschied zu Arrays - etwas zugewiesen werden. Insbesondere kann man einem Pointer einen String zuweisen. Dieser String kann ebensogut eine Konstante sein!

Man kann also schreiben:

```
.
.
.
char *cp1, *cp2, *cp3;
.
cp1 = "\nDas sollten Sie nicht tun!";
cp2 = "\nDas verstehe ich auch nicht!";
.
.
.
```

Der nächste Schritt: Arrays sind nichts anderes als zusammenhängende Folgen von Variablen. Auch Pointer sind Variablen. Also sollte es möglich sein, Arrays von Pointern zu bilden. Die Deklaration für einen solchen Array hat folgende Gestalt:

```
char *pointer_array[50];
```

Damit wird ein Array mit dem Namen *pointer_array* definiert, der 50 Komponenten vom Typ Zeichen-Pointer aufnehmen kann. Jede dieser Komponenten kann also ein String sein, wobei über die Maximallänge nichts ausgesagt ist: jeder in diesen Array 'eingehängte' String ist nur so lange, wie für seine Zeichen Platz benötigt wird. Das ist der eine große Vorteil, den diese Lösung vor einem zweidimensionalen Zeichen-Array hat.

Der andere Vorteil: dieser Array kann initialisiert werden. Dazu muß er jedoch im Programm als globale Variable deklariert sein.

Eine globale Variable ist eine Variable, die nicht innerhalb einer Funktion definiert ist (also nicht in einem Anweisungsblock), sondern außerhalb. Im Programm 5.12 ist die Variable *Fehler* eine globale Variable, denn sie wird

(als Array von Zeichenpointern) vor der Hauptfunktion *main* definiert. Zur besseren Kennzeichnung verfolge ich die Konvention, globale Variable stets mit einem Großbuchstaben beginnen zu lassen. Genaueres über diese Materie erfahren Sie im nächsten Kapitel.

Ein globaler Array (und nur dieser!) kann initialisiert werden. Das Initialisieren darf man nicht mit der Wertzuweisung verwechseln. Die Wertzuweisung wird während der Ausführung eines Programmes durchgeführt; das Initialisieren jedoch besorgt der Compiler, es wird nur einmal - zum Zeitpunkt der Übersetzung des Programmes - erfolgen. Beim Programmstart findet das C-Programm dann die Daten im initialisierten Array bereits vor. Um die Details der Initialisierung brauchen Sie sich jetzt noch nicht zu kümmern. Es genügt, sich klarzumachen, daß durch die Initialisierung im Programm die Verhältnisse herrschen, die Sie in der Abbildung 5.14 sehen können.

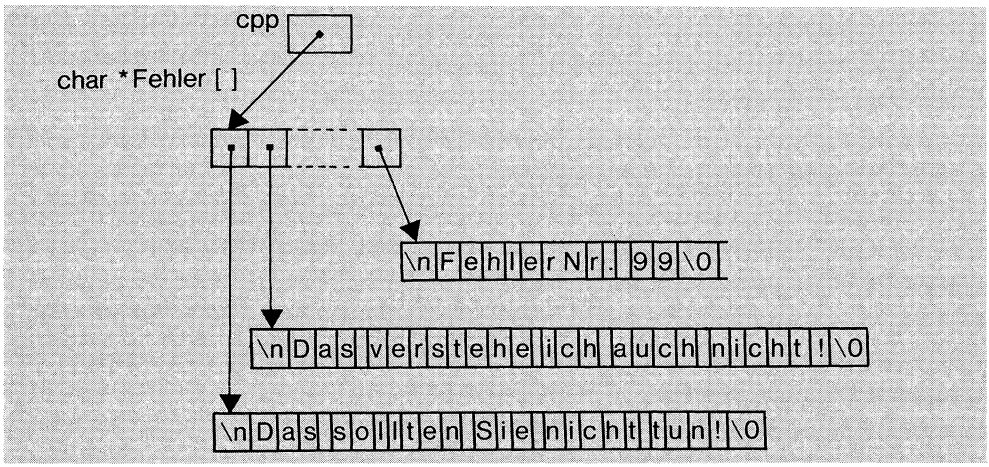



**Abb. 5.14:** *Ein initialisierter String-Array*

Wenn Sie einen Pointer-Array gleich durch den C-Compiler initialisieren lassen, dann brauchen Sie keine Angaben über die Anzahl der Komponenten zu machen. Aus den Daten, die Sie für die Initialisierung bereitstellen (im Beispiel sind das die in geschweiften Klammern stehenden Strings) errechnet sich der Compiler selbst alle nötigen Informationen. Wie üblich reserviert der Compiler für jede Stringkonstante Speicherplatz; zusätzlich werden jedoch in den Array *Fehler* Zeiger gesetzt, die auf die einzelnen Strings verweisen. Das bedeutet, daß mit *Fehler[0]* der erste String, mit *Fehler[1]* der zweite String usw. angesprochen werden kann.

Arraynamen sind Pointer. Ein Zeichen-Array ist ein Zeichen-Pointer; ein Array mit Zeichen-Pointern ist folglich ein Pointer auf einen Zeichen-Pointer. Wenn man im Programm ein Alias für *Fehler* benötigt, dann muß man dieses so deklarieren, wie es Programm 5.12 in der Funktion *main* zeigt:

```
char **cpp;
```

Damit wird ausgedrückt, daß *cpp* ein Pointer auf einen Pointer auf *char* ist. Diesem Pointer kann der Array *Fehler* demnach zugewiesen werden. Die Abbildung 5.14 zeigt bereits den Zustand nach dieser Zuweisung.

Um den ersten String aus *Fehler* zu erhalten, brauche ich jetzt nur **cpp* zu schreiben. Den nächsten String erhalte ich, indem ich zuerst *cpp* inkrementiere und den Pointer dann mit * dereferenziere. Nach diesem Verfahren druckt die *for*-Schleife im Hauptprogramm alle im *Fehler*-Array enthaltenen Meldungen aus. In typischer C-Stenografie wird wieder das Dereferenzieren und Inkrementieren in eine Anweisung integriert. Aber woher weiß die Schleife, wann sie damit aufhören soll? Woher hat das Programm Kenntnis von der Anzahl der Meldungen, mit denen *Fehler* initialisiert wurde?

Dies ist die wundersame Wirkung des *sizeof*-Operators, den ich Ihnen bisher unterschlagen hatte. Der Operator kann auf eine Variable angewendet werden und liefert als Ergebnis die Größe dieser Variablen in Byte. Ist die Variable ein Array, dann sagt mir *sizeof*, wieviele Bytes zur Speicherung des Arrays benötigt werden.

Da jedoch nur Zeichen ein Byte zur Speicherung benötigen, ist die Byteanzahl in der Regel nicht die Anzahl der Komponenten im Array. Wenn man jedoch wüßte, wieviele Bytes Speicherplatz eine einzelne Komponente des Arrays beansprucht, dann bräuchte man die beiden Zahlen nur zu dividieren und hätte die gesuchte Komponentenzahl!

Dazu können Sie die zweite Verwendungsweise von *sizeof* einsetzen: angewendet auf einen Typnamen liefert es nämlich den Speicherbedarf eines Objektes von diesem Typ. Wenn Sie *sizeof* einen Typnamen übergeben, dann sollten Sie ihn in runde Klammern einschließen. Genaueres über die Bildung von Typ-Namen steht im Kapitel 6. Probieren Sie einmal das Programm 5.13 aus!

Die erste *printf*-Anweisung gibt den Platzbedarf von Variablen aus (Werte in der Erklärung in runden Klammern). Dabei handelt es sich der Reihe nach um ein Einzelzeichen (mit 1 Byte Länge), einen zehnelementigen Integer-Array (20 Bytes), eine *long*-Integer (4 Bytes), einen Zeichen-Pointer (4

Bytes), einen Integer-Pointer (4 Bytes) und einen zehnelementigen Array mit *long*-Integern (40 Bytes). Wie Sie sehen können, beanspruchen Adressen (Pointer sind Adressen!) 4 Bytes zu ihrer Speicherung.

```
main()

{   char c;
    int i[10];
    long l;
    char *cp;
    int *ip;
    long lp[10];

    printf("\n%d %d %d %d %d %d", sizeof c, sizeof i,
           sizeof l, sizeof cp, sizeof ip, sizeof lp);

    printf("\n%d %d %d %d %d",sizeof (char) , sizeof (int *),
           sizeof (char **), sizeof (long), sizeof (float));
}
```

**Prog. 5.13:** *Der* sizeof*-Operator*

Im zweiten *printf* wird der Platzbedarf einiger Datentypen ausgegeben: eines Zeichens (*char*; 1 Byte), eines Integer-Pointers (*int* *; 4 Bytes), eines Pointers auf Zeichen-Pointer (*char* ** - von diesem Typ ist *Fehler*; 4 Bytes), einer *long*-Integer und einer Gleitpunktzahl (beide 4 Bytes).

Die Geheimnisse der Typangaben werden im nächsten Kapitel noch eingehend erläutert. Es wird jedoch deutlich, daß man die Komponentenzahl eines Arrays ermitteln kann, indem man seinen Platzbedarf durch den Platzbedarf einer Einzelkomponente dividiert. Ebendies tut das Programm 5.12, um die Anzahl der Meldungen festzustellen.

Dieses Verfahren hat enorme Vorteile: will ich eine neue Fehlermeldung aufnehmen oder eine bestehende löschen, dann brauche ich nur die Initialisierung zu ändern (den in geschweiften Klammern stehenden Teil) und das Programm neu zu übersetzen. Der *sizeof*-Operator garantiert mir, daß sich alle Funktionen, die mit *Fehler* arbeiten, über dessen aktuelle Komponentenzahl informieren können.

Es ist jetzt sehr einfach, die zu Eingang dieses Kapitels erwähnte Fehlerfunktion zu schreiben (Programm 5.14). Die Fehlermeldungen werden in einem globalen String-Array abgelegt; die Funktion überprüft lediglich, ob die Fehlernummer, die sie als Argument erhalten hat, zulässig ist und übergibt in diesem Fall die entsprechende Meldung zur Ausgabe. Unzulässige Fehlermeldungen quittiert sie mit einer eigenen Fehlermeldung.

```
#include <stdio.h>

                                                /*****************************/
char *Fehler[] =                                /* Array mit Pointern ini-   */
{  "\nDas sollten Sie nicht tun!",              /*  tialisieren.             */
   "\nDas verstehe ich auch nicht!",            /* Einige durch und durch    */
   "\nWas ist los...?!?",                       /*  realistische Fehler-     */
   "\nJetzt aber Schluss mit dem Bloedsinn!",   /*  meldungen...             */
   "\nDiesen Fehler gibt's gar nicht!",         /*                           */
   "\nFehler Nr. 99",                           /*****************************/
} ;

                                                /*****************************/
pr_error(n)                                     /* Fehlerfunktion - End-     */
        int n;                                  /*  fassung.                 */
                                                /*****************************/
{   int nr_errors;                              /* Maximale Fehlerzahl be-   */
                                                /*  stimmen.                 */
    nr_errors = (sizeof Fehler / sizeof (char *));/*                          */
                                                /*                           */
    if (n > nr_errors)                          /* Fehlermeldung nicht be-   */
        put_s("\npr_error: unbekannte Fehlermeldung!"); /*  kannt!           */
    else                                        /* Ansonsten:                */
        put_s(*(Fehler + n));                   /*  Ausgeben.                */
}                                               /*                           */
                                                /*****************************/

puts(cp)                                        /*****************************/
     char *cp;                                  /* String ausgeben.          */
                                                /*****************************/
{                                               /*                           */
    while (*cp)                                 /* "putchar" gibt Einzel-    */
        putchar(*cp++);                         /*  zeichen aus.             */
}                                               /*****************************/
```

**Prog. 5.14:** *Fehlertabelle und Fehler-Funktion*

Die Funktion empfängt in ihrem Parameter *n* eine Fehlernummer (bei 0 beginnend) und überprüft, ob diese zulässig ist. Falls nicht, quittiert *pr_error* dieses Ansinnen mit einer eigenen Fehlermeldung; ansonsten wird der geeignete String aus dem Array ausgewählt. Zwar wäre es auch möglich, dies in der üblichen Index-Schreibweise als

```
Fehler[n]
```

zu notieren; ich habe mich jedoch für die Pointer-Schreibweise entschieden, um Ihnen noch einmal dem Zusammenhang zwischen Arrays und Pointern vor Augen zu führen.

Die Ausgabe eines Strings besorgt nicht, wie bisher stets, die Funktion *printf*. Stattdessen finden Sie im Programm 5.14 eine eigene spezialisierte Ausgabefunktion für Strings mit dem Namen *put_s*. Diese sollte Bestandteil der Standard-Bibliothek sein (dort heißt sie *puts*; bitte in der Doku-

mentation nachsehen). Für alle Fälle habe ich jedoch die Funktion selbst definiert; außerdem sehen Sie daran, wie einfach das geht! *put_s* greift auf die bereits bekannte Funktion *putchar* zur Zeichen-Ausgabe zurück, die in *stdio.h* definiert wird.

# 6 Daten und Deklarationen

Die strukturierte Programmierung heißt nicht nur deshalb so, weil sie vernüftige Kontrollstrukturen wie *for* und *while* einsetzt. Damit läßt sich der Ablauf eines Programmes klar und übersichtlich - eben strukturiert - gestalten. Ebenso wichtig ist jedoch die Möglichkeit, die Daten, die das Programm verarbeitet, zu strukturieren, also sogenannte "Datenstrukturen" zu bilden.

Sie haben ja bereits gesehen, um wieviel einfacher die Lösung mancher Probleme ist, wenn man anstatt einer Ansammlung von Einzelvariablen einen Array einsetzt. Auch die Verwendung von Pointer-Arrays bringt, wie im letzten Kapitel demonstriert, Vorteile mit sich, die auf anderem Wege nicht zu erreichen wären. Das ist nur möglich, weil C dem Programmierer die Bildung von Datenstrukturen erlaubt, ihm die Möglichkeit gibt, sich die im Programm verarbeiteten Daten in einer für das zu lösende Problem möglichst angemessenen Weise zurechtzulegen.

Um diese Möglichkeiten voll ausschöpfen zu können, ist ein vertieftes Verständnis desjenigen Teils von C erforderlich, der mit Daten zu tun hat. Nur beides zusammen: Kontrollstrukturen und Datenstrukturen, ermöglicht wirklich professionelle Programmierarbeit, wie man sie in vielen guten C-Programmen zu sehen bekommt.

## 6.1 Variablen: Gültigkeit und Speicherungsklassen

Eine Variable ist dazu da, einen Wert zu speichern. Was gibt es darüber noch groß zu sagen? Warum ein eigenes Kapitel für dieses so einfache Thema?

Weil Variablen eine Speicherklasse und einen Gültigkeitsbereich haben können, weil sie einfach und zusammengesetzt sein können, weil sie initialisiert werden können, weil ihr Wert statisch oder dynamisch verwaltet werden kann und und und...

All dies sind Dinge, die in den bisherigen Ausführungen - wenn überhaupt - oftmals nur am Rande angeklungen sind. Das sechste Kapitel dient deshalb dazu, die bisher über Variablen herumschwirrenden Einzelinformationen zusammenzutragen und zu vereinheitlichen.

### 6.1.1 Von der Sichtbarkeit der Variablen: Gültigkeitsbereich

Wenn Sie bisher in BASIC gearbeitet haben, dann wissen Sie, daß in BASIC-Programmen jede Variable an jeder Stelle des Programms verfügbar ist. Man sagt dazu auch, daß in BASIC alle Variablen global sind. Globale Variablen sind ein zweischneidiges Schwert; meistens schneidet man sich als Programmierer damit ins eigene Fleisch. Der Vorteil dieses Verfahrens: daß alle in Variablen gespeicherten Informationen in allen Teilen eines Programms zugänglich sind, erleichtert dem Programmierer den Informationszugriff.

```
int Var1;                                          globale Variable

main ()

{   int x;

    Var1 = 10;

    x = fun1(2, 3);
    printf("\nVar1 = %d, x = %d", Var1, x);

    x = fun2(4, 5);
    printf("\nVar1 = %d, x = %d", Var1, x);
}

int fun1(a, b)
    int a, b;                                      lokale Variablen

{   Var1 = a * b;
    return a - b;
}

int fun2(a, b)
    int a, b;

{   int x;

    x = a - b;
    Var1 = x * 4;
    return a * b;
}
```

**Abb. 6.1:** *Lokale und globale Variablen*

Oft aber kommt es in BASIC vor, daß eine an einer Programmstelle benötigte Variable an einer anderen Stelle versehentlich verändert wird. So ist es in BASIC-Programmen ohne weiteres möglich, die in einer Zählschleife verwendete Zählvariable an anderer Stelle (z.B. in einem innerhalb der Schleife gerufenem Unterprogramm) zu verändern und so die schönsten Endlosschleifen zu kreieren, ohne daß man der Schleife selbst den Grund für den Fehler ansehen würde: der verhängnisvolle Schritt steht an ganz

anderer Stelle im Programm. Entsprechend mühsam ist dann die Fehlersuche. Dies ist die Kehrseite der Medaille "Globale Variable".

Die Computerwissenschaftler haben schon lange erkannt, daß die Nachteile der globalen Variablen ihre Vorteile bei weitem überwiegen. Deshalb wurde in moderne Programmiersprachen das Konzept der lokalen Variablen eingeführt.

Bei der Unterscheidung zwischen "lokal" und "global" geht es um die Frage, in welchen Teilen eines größeren Programmes eine Variable bekannt ist. Dies wiederum hängt davon, wo die Variable im Programm deklariert ist. Betrachten Sie dazu die Abbildung 6.1.

Da sinnvolle Anwendungen für globale Variable nur im Zusammenhang eines größeren Programmes möglich sind (oder im Zusammenhang mit Initialisierungen; darauf komme ich bald zu sprechen) muß ich Sie dafür auf das nächste Kapitel des Buches vertrösten.

Eine Variable ist lokal, wenn sie Parameter einer Funktion ist oder in einem Anweisungsblock deklariert wurde. In der Abbildung 6.1 werden sechs lokale Variable deklariert: eine in *main* mit Namen *x*; zwei in *fun1* (die beiden Parameter *a* und *b*) sowie noch einmal drei in *fun2*: die beiden Parameter *a* und *b* sowie die Integer-Variable *x*.

Da tauchen ja die Namen *a*, *b* und *x* gleich mehrfach auf! Das ist aber kein Problem, denn es handelt sich um lokale Variable und das bedeutet: sie sind nur innerhalb des Blocks, in dem sie deklariert wurden, bekannt.

Ein Parameter ist nur innerhalb der Funktion, deren Parameter er ist, bekannt und ansprechbar. Es kann anderswo, in einer anderen Funktion, eine Variable oder einen Parameter gleichen Namens geben, aber das führt zu keinem Konflikt. Jede Funktion kennt nur "ihren" Parameter. Sie bekommt gar nicht mit, daß vielleicht eine andere Funktion einen gleichnamigen Parameter besitzt und hat auch keinerlei Möglichkeit, darauf zuzugreifen.

Etwas anders ausgedrückt: *a* und *b* in *fun1* haben nicht das Geringste mit dem *a* und *b* in *fun2* zu tun. *fun2* hat keinerlei Möglichkeit, die Parameter von *fun1* zu verändern (oder wenigstens ihren Inhalt zu Gesicht zu bekommen) und umgekehrt.

Ganz anders sieht es mit *Var1* aus. Diese Variable ist außerhalb jeder Funktion deklariert. Deshalb ist es eine globale Variable. Eine globale Variable ist ab der Stelle im Programm in allen Funktionen bekannt, die danach noch definiert werden. Da *Var1* gleich zu Beginn des Beispiels in der Abbildung 6.1 definiert wurde, ist es in *main*, in *fun1* und in *fun* be-

kannt und kann von all diesen Funktionen aus modifiziert werden, ohne daß diese Funktionen die Variable erst deklarieren müßten. Darum hat das Programm aus der Abbildung das folgende Ergebnis:

```
Var1 = 6, x = -1
Var1 = -4, x = 20
```

*main* weist zwar gleich zu Beginn der Variablen den Wert 10 zu; dann aber wird *fun1* gerufen, welches ebenfalls *Var1* modifiziert (es weist ihm das Produkt seiner Parameter zu); so kommt es, daß *Var1* jetzt den Wert 6 hat. Auch der nachfolgende *fun2*-Aufruf modifiziert *Var1*. Das teuflische an der Sache ist, daß diese Modifikationen in *main* nicht zu sehen sind. Beim Aufruf von *fun1* und *fun2* in *main* deutet nichts darauf hin, daß sich diese Funktionen nicht nur brav mit ihren Parametern beschäftigen und einen Wert zurückgeben, sondern so nebenbei auch noch eine Variable nachhaltig modifizieren. Man sagt deshalb auch, daß *fun1* und *fun2* Funktionen mit einem Seiteneffekt sind.

Seiteneffekte - die stets mit globalen Variablen einhergehen - sind in größeren Programmen schwer zu verfolgen. Das macht solche Funktionen zu aussichtsreichen Kandidaten für subtile Fehler. Andererseits können globale Variable beträchtliche Arbeitsersparnis herbeiführen. Wird nämlich eine bestimmte Information in einer Vielzahl von Funktionen eines Programms benötigt, dann hat man zwei Möglichkeiten:

1. Diese Information über Parameter weiterzugeben
2. Die Information in einer globalen Variablen unterzubringen

Bei der ersten Möglichkeit muß man alle Funktionen um diesen Parameter erweitern, was manchmal zu unübersichtlichen Programmen führt. Die zweite Möglichkeit ist - bei sorgsamen Umgang mit den globalen Variablen - in diesem Falle oft die bessere.

Es spricht noch ein zweites Argument für die globalen Variablen, nämlich daß sie initialisiert werden können. Dazu wird später noch mehr zu sagen sein.

Die Sichtbarkeitsregel für globale Variable lautet: eine globale Variable ist ab der Stelle, in der sie im Programm deklariert ist, für alle nachfolgenden Funktionen sichtbar. Deshalb ist das Programm in der Abbildung 6.2 nicht korrekt.

Hier wird die Variable *Var1* in *main* verwendet, ehe sie deklariert wurde. Dies quittiert der Compiler mit einer Fehlermeldung; denn er erlaubt nur die Verwendung von Variablen, die zuvor deklariert wurden. Die Funktionen *fun1* und *fun2* können nach wie vor auf *Var1* zugreifen, denn ihre

Definition erfolgt hinter der von *Var1*. Die Regeln für die Sichtbarkeit globaler Variablen haben ihre Ursache im Vorgehen des Compilers: beim Übersetzten des Quellprogramms geht er dieses von vorne nach hinten durch und reserviert für jede deklarierte Variable Speicherplatz. Erst dadurch wird die Variable dem Compiler bekannt und erst danach darf sie in Zuweisungen auftauchen.

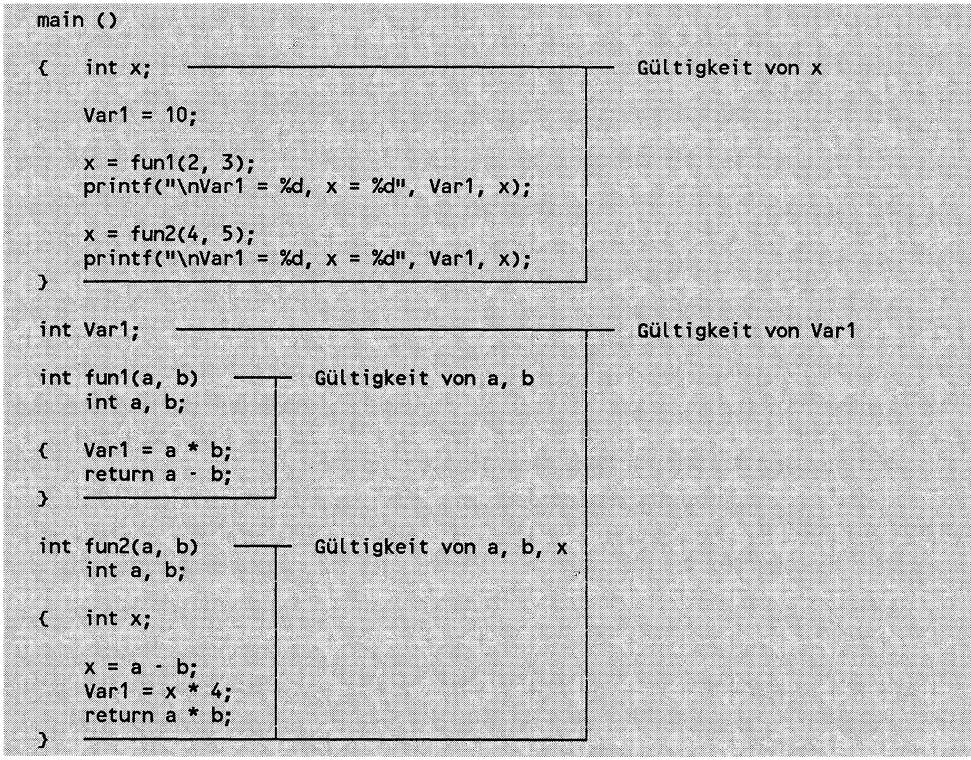



**Abb. 6.2:** *Gültigkeitsbereich globaler Variablen*

Die Sichtbarkeit von globalen Variablen ab der Stelle ihrer Deklaration im Programm ist nur einer Einschränkung unterworfen: hat eine Funktion einen Parameter oder eine lokale Variable mit dem gleichen Namen wie eine globale Variable, dann verstellt dieser Parameter bzw. diese lokale Variable der Funktion gleichsam die Sicht auf die globale Variable gleichen Namens. Aus diesem Grund habe ich mir angewöhnt, globale Variable stets mit einem Großbuchstaben beginnen zu lassen, um eine irrtümliche Namensgleichheit zwischen globalen und lokalen Variablen (die dann konsequent kleingeschrieben werden sollten) weitestgehend auszuschließen.

Dieses 'Verstellen' von Variablen durch Namensgleichheit demonstriert die Abbildung 6.3. Hier ist *var1* einmal als globale Variable deklariert, dann aber auch als lokale Variable der Funktion *fun*.

```
int var1;

main()

{   var1 = 10;
    printf ("\nvar1 vorher: %d", var1);
    fun();
    printf ("\nvar1 nachher: %d", var1);
}

fun()

{   int var1;

    var1 = 20;
}
```

**Abb. 6.3:** *Lokale Variable haben Vorrang vor globalen.*

Wie ein Programmlauf zeigt, haben in solchen Situationen, in denen ein Namenskonflikt zwischen globalen und lokalen Variablen besteht, die lokalen Variablen den Vorrang. Die Zuweisung des Werts 20 an *var1* innerhalb von *fun* hat keinerlei Auswirkung auf den Wert der globalen Variablen *foo*, die nur einmal, zu Beginn von *main*, mit einem Wert versehen wird. Hier das Ergebnis des Programms:

```
var1 vorher: 10
var1 nachher: 10
```

Um es noch einmal zusammenzufassen: der Gültigkeitsbereich einer Variablen hängt von der Stelle ab, an der sie deklariert wurde. Erfolgt die Deklaration außerhalb einer Funktion, so ist die Variable global und von dieser Stelle ab allen nachfolgend definierten Funktionen zugänglich. Erfolgt die Deklaration in der Parameterliste einer Funktion oder innerhalb ihres Anweisungsblocks, dann ist die Variable lokal und nur in der betreffenden Funktion bekannt.

### 6.1.2 Vom Leben der Variablen: Speicherungsklassen

Der Ort der Deklaration einer Variablen gibt Auskunft darüber, wo die Variable bekannt ist. Die folgenden Ausführungen befassen sich mit der Art und Weise, wie Variablen gespeichert werden, was unter anderem Auswirkungen auf die 'Lebensdauer' einer Variablen hat. Dazu kann der C-Programmierer seine Variablen neben einem Gültigkeitsbereich auch noch

mit einem Speicherungsklassen-Attribut (uff!) versehen. Zur Auswahl stehen hier *auto*, *static*, *register* und *extern*.

### 6.1.2.1 Automatische Variablen

Funktionen werden aufgerufen, tun ihre Arbeit, liefern gegebenenfalls ein Ergebnis und treten dann wieder in den Hintergrund. Dort harren sie eines neuerlichen Aufrufs, bis das Programm beendet ist. Dann 'sterben' sie. Nur eine Funktion ist nicht ganz so bescheiden: die Hauptfunktion *main*. Sie ist aktiv, solange das Programm läuft, ja sie ist das Programm. Nun enthalten die meisten Funktionen lokale Variablen, speichern darin Daten. Was ist mit diesen Daten, wenn die Funktionen ins zweite Glied zurücktreten? Verschwinden sie?

Im Normalfall schon. Der Normalfall einer lokalen Variablen in C ist die sogenannte "automatische" Variable. Wenn die Funktion aufgerufen wird, dann tauchen diese Variablen wie aus dem Nichts auf, können in der üblichen Weise manipuliert werden. Ist die Funktion mit ihrer Arbeit fertig, dann verschwindet die automatische Variable wieder. Daten, die eben noch in ihr gespeichert waren, sind dann plötzlich nicht mehr erreichbar. Wenn die Funktion erneut aufgerufen wird, dann werden die automatischen Variablen erneut (eben automatisch) ins Leben gerufen und können wieder mit Daten versorgt werden. Automatische Variablen sind eine Art Kurzzeitgedächtnis für die Funktion, das Informationen nur solange speichert, wie die Funktion aktiv ist.

Da die Parameter einer Funktion nach deren Verlassen ihren Wert verlieren, sind auch sie zu den automatischen Variablen zu rechnen. C sieht zwar das Schlüsselwort *auto* vor, um dieses Variablenattribut bei der Deklaration zu vergeben. Da der Compiler für lokale Variable aber ohnehin automatisch diese Speicherungsklasse verwendet, wenn Sie ihm nichts anderes befehlen, sieht man es in C-Programmen so gut wie nie.

Nun kann es sein, daß eine Funktion bei ihrem Aufruf wertvolle Erkenntnisse gesammelt hat, die man beim nächsten Aufruf ganz gerne noch verfügbar hätte. Man bräuchte dazu ein 'Langzeitgedächtnis', Variablen, die ihren Wert zwischen Funktionsaufrufen beibehalten. Dazu gibt es in C die statischen Variablen.

### 6.1.2.2 Statische Variablen

Den Nutzen dieser Variablen verdeutlicht das Programm 6.1. Es liest Zeichen von der Tastatur, bis es mit dem Endezeichen '#' abgebrochen wird. Die gelesenen Zeichen werden wieder ausgegeben, es sei denn, Sie geben

von der Tastatur mehrfach hintereinander dasselbe Zeichen ein. Das Programm ignoriert diese Wiederholungen: Sie sehen das Zeichen nur einmal.

Zum Einlesen der Zeichen darf nicht, wie üblich, die Funktion *getchar* benutzt werden, da diese die eingelesenen Zeichen nicht stumm an das Programm weitergibt, sondern sie auf den Bildschirm echot. Dieses Echo soll aber das Programm selbst übernehmen (und so eben Wiederholungen eines Zeichens unterdrücken), weswegen ich für diesen Zweck auf eine Funktion *Ohne_echo* zurückgreife, die sich direkt an das Betriebssystem des ATARI wendet. Sie benötigen dazu Zugang zu den TOS-Funktionen (zum Betriebssystem des ATARI). Im ATARI-C erreicht man dies mit der Funktion *gemdos* aus der Standard-Bibliothek. *Ohne_echo* ist am Programmbeginn als Makro definiert. Sollten Sie mit einem anderen C-Compiler arbeiten, dann müssen Sie hierfür diejenige Funktion aus der Standardbibliothek heranziehen, die in Ihrem System Konsoleingabe ohne Echo bewirkt.

```
#include <stdio.h>
extern long bdos()
#define Ohne_echo() gemdos(0x8)
                                        /*****************************/
main()                                  /* Tastatur 'entprellen'.    */
                                        /*****************************/
{  char c;                              /*                           */
                                        /*                           */
   while ((c = Ohne_echo()) != '#')     /* Zeichen ohne Echo ein-    */
      outchar(c);                       /*  lesen und vom Filter     */
}                                       /*  outchar ausgeben las-    */
                                        /*  sen.                     */
                                        /*****************************/

outchar(c)                              /*****************************/
char c;                                 /* Zeichen nur einmal        */
                                        /*  'durchlassen'.           */
                                        /*****************************/
{  static char last_c = '\0';           /* Das zuletzt gelesene      */
                                        /*  Zeichen.                 */
   if (c == last_c)                     /* Neues Zeichen gleich      */
      return;                           /*  letztes?                 */
                                        /*                           */
   putchar(c);                          /* Wenn nicht: ausgeben.     */
   last_c = c;                          /* Zeichen dauerhaft mer-    */
}                                       /*  ken.                     */
                                        /*****************************/
```

**Prog. 6.1:** *Statische lokale Variable*

Jetzt aber zu dem Trick, mit dem die Unterdrückung mehrfacher Zeichen gelingt. Die Filter-Funktion *outchar* arbeitet mit einer Zeichenvariable, die so deklariert ist:

```
static char last_c;
```

Da *last_c* innerhalb von *outchar* deklariert ist, ist es lokal: keine andere Funktion kann ihm etwas anhaben. Da es mit dem Attribut *static* versehen ist, ist es jedoch nicht zum Vergessen verdammt. Stattdessen behält es seinen Wert zwischen den einzelnen Aufrufen von *outchar* und eignet sich daher hervorragend, um sich das letzte gelesene Zeichen zu merken.

Die Deklaration von *c* bietet ein weiteres Beispiel für die Initialisierung, die Ihnen bereits im Programmbeispiel zu den Pointer-Arrays begegnet ist. Sieht der Compiler eine Variablendeklaration, dann reserviert er Speicherplatz für die Variable und - wie wir jetzt wissen - notiert sich, ob die Variable global oder lokal ist und ob sie zwischen Funktionsaufrufen erhalten bleiben soll (also eine statische Variable ist). Er legt jedoch noch keinerlei Wert in der Variablen ab: nach der Deklaration ist eine Variable undefiniert. In Speicherdiagrammen habe ich dies dadurch ausgedrückt, daß ich in die Variablenzellen ein Fragezeichen geschrieben habe.

Nun vergleicht *outchar* jedes eingelesene Zeichen mit *last_c*. Beim ersten Aufruf der Funktion ist aber deren Wert undefiniert und das bedeutet: *last_c* kann ebensogut zufälligerweise das gerade von der Tastatur eingelesene Zeichen enthalten, das dann unterdrückt würde. "Undefiniert" bedeutet nämlich nicht, daß die Variable überhaupt nichts enthält; es läßt sich nur nicht sagen, was sie enthält!

Um zu verhindern, daß das Programm durch einen dummen Zufall das allererste Zeichen unterdrückt, muß *last_c* mit einem harmlosen Startwert versorgt werden; hier bietet sich das bekannte Null-Byte an, da dies von der Tastatur nicht eingegeben werden kann. Allerdings: per Zuweisung geht das Versorgen mit dem Startwert nicht. Denn die Zuweisung würde ja bei jedem Aufruf ausgeführt, wodurch das (am Ende von *outchar*) gespeichert letzte Zeichen verlorenginge. Aus dieser Zwickmühle hilft nur eine Initialisierung.

Eine Initialisierung ist - es wurde bereits einmal erwähnt - eine Versorgung einer Variablen mit einem Startwert, die der Compiler vornimmt. Trifft er auf eine Deklaration mit einer Initialisierung, dann reserviert er nicht nur Speicherplatz (und merkt sich den ganzen anderen Krimskrams, der oben erwähnt wurde), sondern läßt diesen Speicherplatz nicht undefiniert: er wird mit dem in der Initialisierung angegebenen Wert besetzt. Dieser Wert wird durch nachfolgende Zuweisungen überschrieben und ist dann verloren. Mehr Informationen über die Initialisierung finden Sie im Abschnitt 6.4 dieses Kapitels.

### 6.1.2.3 Register-Variablen

Automatische Variablen können mit einem Attribut versehen werden, das zwar nichts an ihrer Sichtbarkeit (sie bleiben lokal) und ihrer Lebensdauer (ihr Wertverschwindet nach Beendigung der Funktion) ändert, dafür aber Auswirkungen auf die Geschwindigkeit des Programmes hat.

Normalerweise bleibt es dem Compiler überlassen, wo er für eine Variable Speicherplatz besorgt. Meist werden diese im Arbeitsspeicher des Computers abgelegt; automatische Variablen werden dabei mit Vorliebe auf den Stapel gedrückt. Den schnellsten Zugriff auf Variablen hat der Prozessor jedoch dann, wenn er sie in seinen Registern vorfindet. Dort werden sie letztlich ohnehin verarbeitet; jeder Zugriff auf eine Variable bedeutet nämlich, daß ihr Wert aus dem Arbeitsspeicher (vom Stapel) erstmal in ein Prozessor-Register transportiert wird, ehe er weiterverarbeitet werden kann. Ein Programm wird deshalb umso schneller, je mehr Variablen permanent in den Registern gehalten werden. Da der Prozessor nur über eine begrenzte Anzahl von Registern verfügt, können nicht alle Variablen diese Sonderbehandlung erfahren.

Deshalb ist es zumindest wünschenswert, wenn die am häufigsten benützten Variablen eines Programmes in Registern untergebracht werden. Aber woher weiß der arme Compiler, welche Variablen arg in die Pflicht genommen werden und welche nicht so hart arbeiten müssen?

Er erfährt es von Ihnen. Automatische Variablen (und nur diese!) können mit dem schmückenden Beiwort *register* versehen werden. Dies ist ein Hinweis an den Compiler, die so deklarierte Variable falls möglich in einem Register unterzubringen. Die Abbildung 6.4 zeigt eine neue, optimal getunte Variante der bereits mehrfach als Beispiel vorgekommenen Funktion *strcpy*. Die Parameter sind Register-Variable, die ganze Arbeit wird in der Kontrollbedingung der *while*-Schleife erledigt: schneller geht's nicht!

```
strcpy(ziel, quelle)
        register char *ziel;
        register char *quelle;

{   while (*ziel++ = *quelle++)
        ;
}
```

**Abb. 6.4:** *Register-Variablen*

### 6.1.2.4 Externe Variablen

C-Programme müssen nicht vollständig in einer Datei enthalten sein. Sie können auf mehrere Module verteilt werden, die separat übersetzt werden

können. Es ist Aufgabe des Linkers, sich aus diesen Modulen (die Sie ihm natürlich angeben müssen!) alle benötigten Bestandteile eines Programmes zusammenzuklauben und daraus das fertige Programm zu montieren.

In diesem Zusammenhang muß das Thema "globale Variable" wieder aufgegriffen werden. Die Bestimmung, daß eine globale Variable ab der Stelle bekannt ist, an der sie deklariert ist, muß nämlich um einen wesentlichen Punkt erweitert werden: dies bezieht sich nur auf die Datei, in der die Variable vorkommt. Es ist jedoch möglich, auf globale Variable aus einem anderen Modul zuzugreifen. Betrachten Sie dazu die Abbildung 6.5.

```
DATEI 1:                        DATEI 2:

main                            int x;
{
    extern int x;               fun2()
    .                           {
    .                               .
    .                               .
}                                   .
                                }
fun1()
{
    int x;
    .
    .
    .
}
```

**Abb. 6.5:** *Externe Variable*

Die Abbildung zeigt ein fiktives Programm-Fragment, das aus einem Hauptprogram und zwei Funktionen *fun1* und *fun2* besteht. Das Programm ist auf zwei Module verteilt. Das Hauptprogramm und *fun1* befinden sich in einem Modul (DATEI1), die Funktion *fun2* steckt in dem Modul DATEI2. Übrigens: in jedem C-Programm darf es natürlich nur eine Hauptfunktion *main* geben!

Im Modul DATEI2 ist eine Variable $x$ als globale Variable deklariert; das ist nichts Neues. Neu ist jedoch das, was in *main* zu sehen ist:

```
extern int x;
```

Damit wird $x$ als Variable definiert, die in einem anderen Modul vorhanden ist. Der Compiler weiß nun, daß es irgendwo eine Integer-Variable $x$ gibt, die das Hauptprogramm *main* anzusprechen wünscht. Er weiß auch, daß es sich dabei um eine globale Variable handeln muß. So kommt er nicht in Versuchung, das $x$ in *fun1* hierfür heranzuziehen. Dieses $x$ ist lokal. Keine Macht der Welt kann an es 'von außen' heran.

Wohlgemerkt: *x* wird in *main* nicht definiert, sondern lediglich deklariert. "Definieren" bedeutet, den Compiler zum Bereitstellen von Speicherplatz zu veranlassen. Dies geschieht für *x* in DATEI2. Eine Deklaration reserviert jedoch keinen Platz; sie teilt dem Compiler im Gegenteil mit, daß dies an anderer Stelle bereits geschehen ist. Die *extern*-Deklaration veranlaßt den Compiler zu einer Notiz an den Linker, hierfür die richtige Variable einzusetzen. Dies kann nur der Linker, denn er ist die Instanz, die alle Module zu sehen bekommt und sie zusammenmontiert.

Wäre *x* am Ende von DATEI2 deklariert, dann dürfte es in *fun2* nicht verwendet werden. In DATEI1 hat nur *main* auf dieses *x* zugriff, da es nur hier definiert ist. Es wäre jedoch möglich, *x* außerhalb jeder Funktion in DATEI1 zu deklarieren, also genauso, wie eine globale Variable deklariert wird. Dann wäre es wieder in allen Funktionen des Moduls DATEI1 bekannt. Sein Speicherplatz wird jedoch nach wie vor in DATEI2 bereitgestellt.

Gibt es keine Möglichkeit, eine globale Variable vor dieser Zugänglichkeit von außen (also von anderen Modulen) zu schützen? Die gibt es selbstverständlich schon, sonst hätte ich diese rhetorische Frage ja gar nicht erst gestellt!

Will man die Sichtbarkeit einer globalen Variablen streng auf das Modul beschränken, in dem sie deklariert ist, dann muß man sie als statische Variable deklarieren. Ändert man also die Deklaration zu Beginn des Moduls DATEI2 aus der Abbildung 6.5 so ab:

```
static int x;
```

dann steckt der Wurm in dem Programm. Denn das globale *x* ist jetzt nur mehr innerhalb von DATEI2, nicht aber innerhalb des ganzen Programms global. Das Ansinnen von *main* kann nicht befriedigt werden und Sie erhalten eine Fehlermeldung.

Aber nur vom Linker! Der Compiler, der immer nur eine Quelldatei vor Augen hat, kann diesen Umstand nicht bemerken. Erst der Linker findet für das von *main* gewünschte *x* nichts in den Modulen und reagiert sauer. In C-Programmen können also auch Fehler enthalten sein, die erst zum Zeitpunkt des Linkens ans Tageslicht kommen.

Einer dieser Fehler, der sehr beliebt ist, besteht darin, eine Funktion zu verwenden, die in keinem Modul und in keiner Bibliothek definiert ist. Sie sind jetzt in der Lage zu verstehen, was Funktionen für C sind. Sie sind zwar keine Variablen (denn sonst könnte man ihnen etwas zuweisen, und das ist sinnlos), aber sie sind allesamt global! Eine Funktion, die in einem

Modul definiert ist, kann aus anderen Modulen heraus gerufen werden; der Linker wird's schon richten!

Trifft der C-Compiler auf eine Funktion, die in der aktuellen Quelldatei nicht definiert ist, dann denkt er sich: "Wird schon anderswo zu finden sein; hinterlasse ich halt mal eine Nachricht an den Linker". Was passiert, wenn Sie sich bei einem Funktionsnamen verschreiben und - sagen wir mal - anstelle von *printf pintf* tippen? Den Compiler stört's nicht; er bemerkt die Funktion und hinterläßt eine Notiz an den Linker. Der aber müht sich vergeblich: in keiner Bibliothek kann er das Gewünschte finden. Dies bringt Ihnen eine Fehlermeldung ein; zwar reichlich spät, aber immerhin...

Zu Deklaration und Definition von Funktionen gibt es noch mehr zu sagen, was an dieser Stelle nur ablenken würde; schlagen Sie dazu den Abschnitt 6.2.2 dieses Kapitels nach.

Als Kuriosität sei's am Rande vermerkt: weil Funktionen global sind (sie werden ja nicht innerhalb eines Blocks deklariert!), kann man sie ebenso wie globale Variablen durch Zugriffe von außen schützen, indem man sie mit der Speicherungsklasse *static* versieht. Dann sind diese Funktionen nur mehr in der Quelldatei bekannt, man kann sie sich aber nicht mehr vom Linker in andere Module importieren lassen. Mir ist diese Möglichkeit immer etwas obskur vorgekommen und ich habe bisher noch kein sinnvolles Beispiel für ihren Einsatz gefunden!

## 6.2 Deklarationen

Sie haben gesehen, daß in C alles, was mit Daten zu tun hat - Variablen, aber auch Funktionen - mit zwei Attributen versehen ist: der Sichtbarkeit und der Speicherungsklasse. Damit ist aber noch nichts über die Art der Daten gesagt, mit denen es diese Variablen und Funktionen zu tun haben. Darüber muß der Compiler aber Bescheid wissen, um Speicherplatz zu reservieren und um einige Buchhaltungsaufgaben (Adressberechnung bei Array-Indizierung oder Pointer-Arithmetik) korrekt ausführen zu können. Diese Informationen werden ihm in der Deklaration mitgeteilt. Alles, was mit Daten zu tun hat, muß also deklariert werden. Mit Daten zu tun haben in C: Variablen und Funktionen. Variablen speichern Daten. Funktionen werden auf ihre Argumente, also auf Daten, angewendet. Und sie haben einen Wert, produzieren also Daten. Deshalb müssen Variable, Funktionen und deren Parameter deklariert werden.

### 6.2.1 Einfache und zusammengesetzte Datentypen

Die Daten, mit denen Variable und Funktionen zu tun haben, können entweder einfach oder zusammengesetzt sein. Einfache Datentypen sind in C:

- o Zeichen
- o Ganze Zahlen
- o Gleitpunktzahlen
- o Pointer
- o Funktionen

Diese Datentypen heißen deshalb "einfach", weil man aus ihnen die zusammengesetzten Datentypen bilden kann, wie z.B. Arrays, Zeiger auf Funktionen, Strukturen, Funktionen, die Zeiger auf Strukturen mit Arrays als Wert haben... Den Möglichkeiten zur Bildung zusammengesetzter Datentypen sind in C keine Grenzen gesetzt. Sie erfordern aber ein vertieftes Verständnis der einfachen Datentypen.

Wie Sie sehen können, dürfen die einfachen Datentypen nicht mit den elementaren Datentypen verwechselt werden, die in Kapitel 4 besprochen wurden. Da war nämlich keine Rede von Funktionen und Pointern!

Die zusamengesetzten Datentypen von C sind

- o Arrays
- o Strukturen
- o Unions

Bisher tauchten nur die Arrays im Buch auf. Die Strukturen werden Gegenstand späterer Abschnitte dieses Kapitels sein. Alles, was es über die Deklaration zusammengesetzter Datentypen zu wissen gibt, kann jedoch am Beispiel der Arrays klargemacht werden; es entgeht Ihnen also nichts!

Jede Deklaration, ob für einen einfachen oder einen zusammengesetzten Datentyp, führt in C auf einen elementaren Datentyp der Sprache zurück. Zur Deklaration der elementaren Datentypen benutzt man, wie Sie bereits wissen, die Schlüsselwörter *char*, *int*, *float* und *double*, wobei für *int* noch die Attribute *unsigned* und *long* verfügbar sind.

Die Deklaration eines einfachen Datentyps und eines Arrays gibt an, welchen elementaren Datentyp man erhält, wenn man die deklarierte Variable im Programm so hinschreibt, wie sie in der Deklaration steht. Klingt kompliziert? Dann müssen Beispiele her.

Die Abbildung 6.6 zeigt einige einfache Deklarationen (was nicht heißt, daß einfache Datentypen deklariert werden, sondern nur, daß sie einfach

zu verstehen sind). Die Bedeutung der einzelnen Deklarationen will ich nun einzeln durchgehen.

```
1.  int i;
2.  int i[10];
3.  register int i[];
4.  static int *ip;
5.  int fun();
```

**Abb. 6.6:** *Einfache Deklarationen*

1. Dies besagt ganz einfach, daß man im Programm eine Integer erhält, wenn man *i* hinschreibt. Deklarationen sind eben Aussagen darüber, auf welchen elementaren Datentyp (hier: *int*) das deklarierte Objekt (hier: die Variable *i*) führt, wenn man es im Programm hinschreibt.

2. Dies besagt, daß man im Programm eine Integer erhält, wenn man i[0], i[1], ... i[9] hinschreibt. Es besagt ferner, daß man dies auf zehnerlei Weise tun kann. Damit dies möglich ist. muß *i* ein Array sein. Deshalb wird *i* mit dieser Deklaration als Integer-Array mit 10 Komponenten definiert. Es zeichnet sich schon ab: eine C-Deklaration sagt nicht so sehr, was ein Objekt ist, sondern was man mit ihm tun kann.

3. Dies besagt, daß man eine Integer erhält, wenn man i[...] hinschreibt. Es sagt nicht aus, was alles in den eckigen Klammern stehen kann (deswegen die drei Punkte). Ansonsten aber entspricht die Deklaration der letzten, d.h. *i* wird als Integer-Array mit unbekannter Dimension deklariert. Solche Deklarationen braucht man, wenn *i* Parameter einer Funktion sein soll, oder wenn *i* eine externe Variable ist, für die in einem andern Modul Speicherplatz reserviert wurde. Beachten Sie, daß im Unterschied dazu im Beispiel 2 der Array *i* definiert (und nicht nur deklariert) wurde: da findet sich eine Größenangabe, die dazu führt, daß der Compiler Speicherplatz reserviert.
   Nun ist die Deklaration im Beispiel 3 auch noch mit dem Schlüsselwort *register* versehen. Sie wissen, daß dies nur auf automatische Variablen angewendet werden kann und diese dann in Prozessorregister legt. Deshalb kann *i* nur Parameter einer Funktion sein; und deshalb kann *i* keine externe Variable sein.

4. Dies besagt, daß man eine Integer erhält, wen man im Programm **ip* hinschreibt. Nun wissen Sie, daß der *-Operator sinnvoll nur auf einen Pointer angewendet werden kann und dann das Objekt liefert, auf das dieser Pointer zeigt. Deshalb muß *ip* ein Integer-Pointer sein. Hier wird

die indirekte Art der Deklarationen in C deutlich: *ip* liefert eine Integer, wenn man es dereferenziert. Daraus folgt: *ip* ist ein Pointer. Es wird dies jedoch nicht direkt mitgeteilt!
Außerdem findet sich in der Deklaration eine Angabe zur Speicherungsklasse. Mit *static* wird mitgeteilt, daß *ip* seinen Wert beibehalten soll, falls es eine lokale Variable ist, oder daß es für andere Module unzugänglich sein soll, falls es eine globale Variable st.

5. Dies besagt, daß man eine Integer erhält, wenn man *fun* anwendet. Daß *fun* etwas 'anwendbares' ist, erkennt man an den runden Klammern in der Deklaration (den Funktions-Klammern). Weil *fun* angewendet werden kann und dann einen Wert liefert, muß es eine Funktion sein. Denn das ist es, was Funktionen so einzigartig macht. Wieder kommt die indirekte Art von C zum Vorschein.
Keinerlei Aussagen werden in dieser Deklaration darüber gemacht, auf wieviele Dinge (Argumente) man die Funktion anwenden kann und von welcher Art diese Argumente sind. Das gehört in die Funktionsdefinition, hier jedoch hat man es mit einer Funktionsdeklaration zu tun. Auf den Unterschied zwischen Definition und Deklaration gehe ich später noch genauer ein.

Aus diesen einfachen Beispielen läßt sich schon eine bestimmte Systematik der Deklarationen erkennen:

- Jede Deklaration 'mündet' in einen elementaren Datentyp von C (*int*, *char*, *float*, *double*). Dieser Datentyp leitet die Deklaration ein. Falls das deklarierte Objekt mit einer bestimmten Speicherungsklasse versehen werden soll, dann wird das Schlüsselwort für die Speicherungsklasse (*auto*, *static*, *register*, *extern*) vor die Typangabe geschrieben.
- Deklariert wird stets ein Objekt, das einen bestimmten Namen erhält. Dieser Name steht rechts von der Typangabe und kann mit diversem schmückendem Beiwerk versehen sein, das rechts oder links davon angebracht ist.
- Bei dem 'schmückenden' Beiwerk, mit dem der Name ausstaffiert werden kann, handelt es sich um den Stern, die eckigen und geschweiften Klammern.
  Der Stern wird links vom Namen hingeschrieben und macht deutlich, daß es sich bei dem deklarierten Objekt um einen Pointer handelt. Die Klammern werden rechts vom Namen hingeschrieben und drücken aus, daß es sich bei dem deklarierten Objekt um einen Array (wenn sie eckig sind) oder um eine Funktion handelt (wenn sie rund sind).
  Bleiben die Klammern leer, dann wird der Array bzw. die Funktion nur deklariert. Steht etwas in den Klammern, dann hat man es mit einer Definition zu tun.

Aus diesen Bausteinen können komplexe Deklarationen aufgebaut werden. In einer komplexen Deklaration darf es zwar immer nur einen Typ-Bezeichner (*char*, *int*, *float*, *double*) und eine Angabe zur Speicherungsklasse (*extern*, *static*, *auto*, *register*) geben, dafür dürfen (beinahe) beliebige Kombinationen von Sternen, runden und eckigen Klammern vorkommen.

Erinnern Sie sich an das Problem mit dem Vorrang der Operatoren? Wenn Sie noch einmal die Tabelle in der Abbildung 4.9 betrachten, dann werden Sie feststellen, daß die runden Funktionsklammern und die eckigen Array-Klammern in C Operatoren sind und daß diese höchste Priorität haben. Daß * ein Operator ist, ist schon längere Zeit bekannt; der Vorrangtabelle entnehmen Sie, daß er hinter den Klammer-Operatoren rangiert. Dies ist für das Verständnis der Beispiele in Abbildung 6.7 wichtig, in der Sie einige komplexe C-Deklarationen zusammen mit einem Kommentar sehen, der sagt, was damit deklariert wird.

Einige dieser Deklarationen gehen weit über das hinaus, was in diesem Buch bisher besprochen wurde. Aber in dem Maße, in dem Ihre Fertigkeiten als C-Programmierer zunehmen, wird auch Ihr Bedürfnis für komplexe Datentypen steigen. Dann können Sie sich anhand dieser Erklärungen jeden Datentyp - und sei er noch so abartig - selbst zusammensetzen.

| Deklaration | | Bedeutung |
|---|---|---|
| 6. | `int *ipa[];` | Array von Integer-Pointern |
| 7. | `int *ipfun();` | Funktion, die Integer-Pointer zur}ckgibt |
| 8. | `int (*funp)();` | Pointer auf Funktion, die Integer zur}ckgibt |
| 9. | `int (*p_arr[])();` | Array mit Pointern auf Integer-Funktionen |
| 10. | `int *(*fp_arr[])();` | Array mit Pointern auf Funktionen, die Integer-Pointer zurückgeben. |

**Abb. 6.7:** *Komplexe Deklarationen*

Hier wieder eine ausführliche Analyse der einzelnen Deklarationen:

6. Die Array-Klammern binden stärker als der *-Operator; deshalb muß *ipa* ein Array sein. Bleibt die Frage, was er enthält. Der Rest der Deklaration besagt, daß die im Array *ipa* enthaltenen Komponenten eine Integer liefern, wenn man sie mit * dereferenziert; also sind es Pointer.

7. Die runden Funktions-Klammern binden stärker als der *-Operator. Also ist *ipfun* eine Funktion. Was gibt sie zurück? Der Rest der Deklaration sagt, daß der Wert von *ipfun* eine Integer liefert, wenn man ihn mit * dereferenziert; also ist es ein Integer-Pointer.

8. Der Vorrang der runden Funktions-Klammern vor dem * wird hier durch Klammerung aufgehoben. Wir müssen uns deshalb zuerst mit dem Teil *(*funp)* beschäftigen. Dieser besagt, daß man durch Dereferenzieren von *funp* etwas erhält; also ist *funp* ein Pointer; aber worauf? Dazu muß der Rest der Deklaration betrachtet werden. Jetzt kommen die Funktions-Klammern ins Spiel, die besagen: wenn *funp* dereferenziert wird, dann erhält man etwas, das angewendet werden kann, eine Funktion also. Folglich ist *funp* eine Pointer auf eine Funktion. Bleibt noch zu klären, was diese Funktion zurückgibt, aber das ist einfach: eine Integer.

9. Wieder wurde durch Klammerung der Vorrang der Funktions-Klammern aufgehoben. der eingeklammerte Teil sagt aus, daß *p_array* ein Array ist, aber einer, dessen Komponenten erst dereferenziert werden müssen; also sind es Pointer. Worauf? Das teilen die Funktionsklammern mit: auf Funktionen. Das *int* zu Beginn der Deklaration macht schließlich klar, daß diese Funktionen Integers zurückgeben.

10. Das krönende letzte Beispiel unterscheidet sich nur in einem Punkt von der Deklaration 9: es sagt nämlich, daß das, was die im Array *fp_arr* enthaltenen Funktionen zurückgeben, erst dereferenziert werden muß, um eine Integer zu liefern. Also ist es ein Pointer, genauer: ein Integer-Pointer.

Der Fantasie sind anscheinend keine Schranken gesetzt. Dennoch müssen Sie einige Einschränkungen beim Kreieren komplexer Datentypen beachten:

- Arrays von Funktionen sind nicht erlaubt; es können jedoch - wie Sie gesehen haben - sehr wohl Arrays mit Pointern auf Funktionen deklariert werden.
- Funktionen dürfen keine Arrays, Strukturen oder Funktionen als Wert zurückgeben. Das verbietet aber nicht die Deklaration von Funktionen, die Pointer auf diese Dinge zurückgeben!

```
int i;
register int j;
char c;
int *fip();
char *cp;
```

Dies entspricht:

```
int i, *fip();
char c, *cp;
register int j;
```

**Abb. 6.8:** *Abgekürzte Schreibweise für Deklarationen.*

Bei der Deklaration mehrer Variablen, die alle in den gleichen elementaren Datentyp münden, ist eine abgekürzte Schreibweise möglich, die bereits mehrfach in den Beispielen auftauchte. Sie können in einer einzigen Anweisung zusammengefaßt werden, wenn die Deklarationen für die einzelnen Variablen durch Komma getrennt werden (vgl. Abbildung 6.8). Alle gemeinsam deklarierten Variablen besitzen auch die gleiche Speicherungsklasse; deshalb darf die Deklaration von *j*, welches ja eine Register-Variable sein soll, nicht mit der von *i* und *fip* zusammengefaßt werden.

Noch eine weitere Möglichkeit hält C bereit, um Ihnen bei der Deklaration/Definition komplizierter Datentypen das Leben zu erleichtern. C kennt ein weiters Schlüsselwort - *typedef* -, das sich syntaktisch wie eine Angabe zur Speicherungsklasse (also wie *auto*, *register*, *static* und *extern*) verhält: es wird wie diese vor eine Typdeklaration geschrieben. Aber die Aufgabe von *typedef* ist es nicht, irgendwelchen Speicherplatz zuzuweisen, sondern sie besteht im Vereinbaren eines neuen Datentyps!

Nehmen wir an, Sie arbeiten in einem Programm häufig mit Pointern auf Funktionen, die Integer-Pointer als Wert zurückgeben. Wie Sie jetzt wissen, ist zur Deklaration einer Variable vom passenden Typ - nennen wir Sie *funct1* - folgender Aufwand nötig:

```
int * (*funct1)();
```

Brauchen Sie in Ihrem Programm nun aber nicht nur eines, sondern gleich ein ganzes Dutzend Tierchen dieser Art, dann steht Ihnen nicht nur erheblicher Schreibaufwand bevor, sondern das Programm erhält auch ein geradezu furchteinflößendes Aussehen. Denn komplexe Deklarationen sind nicht gerade besonders leicht zu lesen und verstehen; das geben sogar die Entwickler von C zu!

Man kann sich aber in C einen Datentyp "Pointer auf eine Funktion, die Integer-Pointer liefert" selbst bauen, diesen mit einem sprechenden Namen versehen und dann Deklarationen unter Verwendung dieses Eigenbau-Typs besorgen. Hier steht, wie es geht:

```
typedef int *(*PFIP)();

PFIP funct1, funct2, funct3;
```

Mit der ersten Zeile wird ein neuer Datentyp vereinbart, der den Namen PFIP trägt. Das soll schwach an "Pointer auf Funktion mit Integer Pointer als Wert" erinnern; es ist großgeschrieben, um daran zu gemahnen, daß es sich dabei um etwas Selbstdefiniertes handelt (aus ähnlichen Gründen hat sich die Großschreibung bei *define*-Konstanten eingebürgert).

Diesen neuen Datentyp kann man nun ebenso wie die "eingebauten" Typen (*int*, *char*, *float* und wie sie alle heißen) zur Definition neuer Variablen verwenden. In der nächsten Zeile wird das auch gleich gemacht: *funct1*, *func2* und *funct3* werden als "Pointer auf Funktionen, die Integer-Pointer zurückgeben" definiert - und zwar Kraft der zuvor erfolgten *typedef*-Vereinbarung.

Wenn Sie einen neuen Datentyp vereinbaren wollen, dann schreiben Sie also *typedef*, lassen darauf die Typdeklaration folgen, ganz so, als würden Sie eine Variable deklarieren, schreiben aber an die Stelle des Variablennamens den neuen, selbstvereinbarten Typnamen. Dieser unterliegt den üblichen Gesetzmäßigkeiten für C-Namen (keine Sonderzeichen im Namen, Maximallänge beachten) und sollte - wenn Sie sich dem Mainstream der C-Programmierer anschließen wollen - großgeschrieben sein.

### 6.2.2 Deklaration und Definition

Wird eine Variable in einem Programm verwendet, dann muß sie deklariert werden. Dies ist erforderlich, damit der Compiler seine Arbeit korrekt verrichten kann. Aber auch Funktionen, die einen Wert zurückgeben. müssen deklariert werden, falls dieser Wert im Programm weiterverwendet werden soll.

Diese Regel wird allerdings in C nicht sehr streng kontrolliert. Verwenden Sie eine Funktion in Ihrem Programm, ohne sie zu deklarieren, dann nimmt der Compiler einfach an, daß diese Funktion einen Integer-Wert zurückliefert. Liefert sie einen anderen Wert, dann kann die Deklaration ebenfalls unterbleiben, wenn der Wert im Programm nicht weiterverwendet wird. In Beispielprogrammen des letzten Kapitels wurden gelegentlich Stringfunktionen aus der Standard-Bibliothek (*strcopy*, *gets*) verwendet, die einen Zeichen-Pointer als Wert liefern, die also in den Programmen als

```
char *strcopy(), *gets();
```

deklariert werden müßten. Da ich jedoch den Wert niemals weiterverwendete, d.h., es kam keine Anweisung der Form

```
cp2 = gets(cp);
```

vor, führte dies zu keinen Problemen. Was aber passiert, wenn man in dieser Situation vergessen hat, *gets* ordnungsgemäß zu deklarieren? Betrachten Sie dazu die Abbildung 6.9.

In diesem Programm-Fragment verwendet die Hauptfunktion eine Funktion *fun*, die in *main* nicht deklariert ist. Darüber sieht der Compiler großzügig hinweg; er nimmt einfach an, daß *fun* eine Integer zurückgibt. Der nach-

folgenden Definition von *fun* können Sie jedoch entnehmen, daß der Wert der Funktion ein Zeichen ist. Um die Verhältnisse vollends durcheinander zu bringen, wird bei der Anwendung von *fun* in *main* dessen Wert auch noch an einen Zeichen-Pointer zugewiesen.

```
main()
{   char *cp;
    .
    .
    .
    cp = fun(...);
}

char fun(...)
{
    .
    .
    .
}
```

**Abb. 6.9:** *Typ-Probleme mit Funktionen*

Hier würde folgendes passieren: der Compiler erwartet von *fun* eine Integer, also 2 Bytes. Er bekommt aber nur ein Byte geliefert (denn *fun* gibt eine Zeichen zurück), deshalb besorgt er sich von irgendwoher das nötige zweite Byte! Welchen Wert dies hat, kann man nicht sagen; es ist undefiniert. Die Wahrscheinlichkeit ist jedenfalls sehr groß, daß dadurch 'Schrott' in das Programm eingeführt wird. Dieser Schrott wird an den Zeiger *cp* zugewiesen, für diesen Zweck also nach den Regeln der Typumwandlung auf vier Bytes 'aufgeblasen'; niemand kann jetzt mehr sagen, auf was der Zeiger *cp* eigentlich zeigt!

Soweit läßt es der Compiler in der Situation von Abbildung 6.9 zum Glück nicht kommen. Bei der Übersetzung von *main* macht er sich eine Notiz, daß *fun* vom Typ "Integer" ist (denn er findet in *main* keine anderslautende Deklaration für *fun*). Dann aber kommt er an die Definition von *fun* und findet dort eine Deklaration als Zeichen-Funktion. Dies widerspricht seinen eben gemachten Aufzeichnungen und er reagiert mit einem Fehler (der ATARI-Compiler meldet z.B. "redeclaration: fun").

Den Widerspruch kann der Compiler allerdings nur aufdecken, wenn die Definition der anstößigen Funktion in der selben Datei wie *main* (bzw. die Funktion, die sie verwendet) zu finden ist. Sind Hauptfunktion und Unterfunktion *fun* auf verschiedene Module aufgeteilt, dann bleibt der Compiler bei seiner irrigen Annahme über den Typ von *fun*. Der Fehler bleibt unentdeckt, da sich der Linker - die einzige Instanz, die alle Funktionen des

Programms zu Gesicht bekommt - um Typenüberprüfung nicht mehr kümmert. Erst wenn Ihr Programm anfängt, Seltsames zu tun, werden Sie wieder auf den Fehler gestoßen; aber finden Sie ihn dann mal!

Jetzt kann ich Ihnen auch ein Detail erklären, das Sie bisher vielleicht vor Rätsel gestellt hat. Gelegentlich mußte ich in Beispiel-Programmen auf TOS-Funktionen direkt zugreifen. Bei der dazu verwendeten Betriebssystem-Funktion *gemdos* handelt es sich um eine Funktion, die einen *long*-Wert zurückgibt. Dies muß dem Compiler mitgeteilt werden. Nun hat man dazu zwei Möglichkeiten: eine lokale und eine globale Deklaration.

Will man eine andernorts definierte Funktion, wie es *gemdos* ist, nur in einer einzigen eigenen Funktion verwenden, dann reicht es aus, sie lokal in dieser Funktion zu deklarieren. Benötigt man sie jedoch in mehreren Funktionen, dann müßte sie in jeder dieser Funktionen deklariert werden. Dies ist umständlich; stattdessen kann man zu einer globalen Deklaration greifen, die die deklarierte Funktion *gemdos* in allen Funktionen dieses Moduls bekanntmacht. Korrekterweise habe ich die Deklaration noch mit der Speicherungsklasse *extern* versehen, da *gemdos* ja in einem anderen Modul definiert ist. Deshalb wird also das Programm 6.1 mit der Deklaration

```
extern long bdos();
```

eingeleitet.

All dies zeigt nicht nur, daß man mit Deklarationen in C-Programmen sehr sorgfältig umgehen muß. Es zeigt auch, daß ein Unterschied zwischen Definition und Deklaration besteht. Die Definition ist der Ort, an dem bei einer Variablen Speicherplatz bereitgestellt und bei einer Funktion gesagt wird, was sie tut. Eine Definition geht stets auch mit einer Angabe über den Datentyp einher.

Externe Objekte - also globale Variablen und Funktionen - können jedoch an Stellen verwendet werden, an denen sie nicht definiert sind. Dann ist es aber immer noch notwendig, den Compiler über eine Deklaration mit den notwendigen Typ-Angaben zu versorgen.

Um nicht länger so theoretisch zu bleiben, will ich das mit einem kleinen Beispiel demonstrieren. Das Programm 6.2 zeigt eine weitere Stringfunktion, die hier *str_cat* heißt, die jedoch auch in der Standard-Bibliothek unter dem Namen *strcat* enthalten ist.

Die Funktion verknüpft zwei Strings (der mathematische Fachausdruck für das Verknüpfen von Strings ist "Konkatenieren"; daher hat sie ihren seltsamen Namen). Der Array für den ersten String muß dabei groß genug sein, um auch noch die Zeichen des zweiten Strings aufzunehmen. Die

Funktion gibt als Wert einen Zeiger auf den Anfang des Ergebnis-Strings zurück, der unmittelbar zur Ausgabe verwendet werden kann.

```
                                        /*************************/
main()                                  /* Definition und Deklara- */
                                        /*  tion.                  */
                                        /*************************/
{   char str1[100], str2[20];           /*  Definition.            */
    char *str_cat(), *gets();           /*  Deklaration.           */
                                        /*                         */
    puts("\nString1: ");                /*                         */
    gets(str1);                         /*                         */
    puts("\nString2: ");                /*                         */
    gets(str2);                         /*                         */
    puts(str_cat(str1, str2));          /* str_cat gibt str1 zu-   */
}                                       /*  rueck.                 */
                                        /*************************/

                                        /*************************/
char *str_cat(str1, str2)               /*  Strings verknuepfen.   */
     register char *str1, *str2;        /*************************/
                                        /*                         */
{   char *anfang;                       /* Anfang des ersten       */
                                        /*  Strings merken, da die */
    anfang = str1;                      /*  Pointer verschoben     */
                                        /*  werden.                */
    while (*str1)                       /* Ende des ersten Strings */
        ++str1;                         /*  suchen.                */
                                        /*                         */
    while (*str1++ = *str2++)           /* Wie gewohnt kopieren.   */
        ;                               /*                         */
                                        /* Zeiger auf den Anfang   */
    return anfang;                      /*  zurueckgeben.          */
}                                       /*************************/
```

**Prog. 6.2:** *Definition und Deklaration von Funktionen*

Zum Verknüpfen der Strings sucht *str_cat* zuerst das Ende des ersten Strings; dazu verschiebt es den ersten Parameter *str1* (der ja ein Pointer ist; denken Sie an den Zusammenhang zwischen Arrays und Pointern!) solange, bis das Null-Byte auftaucht. Dann wird - nach der Manier von *strcpy* - zeichenweise der zweite String einkopiert. Wert der Funktion soll der verknüpfte String sein, also ein Zeiger auf den Anfang der Zeichenfolge. *str1* kann dazu nicht mehr herhalten, den es wurde bis ans Stringende verschoben. Aber die erste Aktion der Funktion war zum Glück, sich diesen Anfangs-Zeiger in der Pointer-Variable *anfang* aufzuheben. Diese wird deshalb als Funktionswert zurückgegeben. Da die beiden Parameter in der Funktion die Hauptarbeit leisten, wurden sie als Register-Variablen deklariert.

Im Hauptprogramm sehen Sie, daß *str_cat* aufgrund seines von *int* abweichenden Typs deklariert werden muß. Der Vollständigkeit halber ist auch

die Funktion *gets* deklariert, obwohl dies hier nicht nötig ist, da ihr Wert nicht weiterverwendet wird. Wenn Sie den Funktionstyp einer Bibliotheksfunktion wissen wollen, dann müssen Sie in den Unterlagen zu Ihrem C-Compiler nachschlagen. Dort findet sich eine Beschreibung der Bibliotheksfunktionen, die den Funktionstyp und natürlich auch die erforderlichen Parameter angibt.

Im Programm 6.2 wird der Datentyp von *str_cat* zweimal erwähnt: einmal in *main*, wo *str_cat* jedoch lediglich deklariert wird. Dann findet sich jedoch noch die Definition der Funktion, also die Stelle im Programm, an der gesagt wird, was die Funktion eigentlich tut. Um es noch einmal zusammenzufassen: die Definition einer Variable besorgt Speicherplatz, die Definition einer Funktion gibt an, was die Funktion tut. Deklarationen, die nicht definieren, geben nur Typinformationen an den Compiler weiter.

## 6.3 Strukturen

Im Kapitel über die Deklarationen (6.2) wurde bereits erwähnt, daß es in C zur Bildung von zusammengesetzten Datenstrukturen neben den Arrays noch andere Möglichkeiten gibt. Von besonderer Bedeutung sind hier die Strukturen.

Arrays sind Aneinanderreihungen von Komponenten gleichen Typs. Viele Probleme erfordern es jedoch, Komponenten unterschiedlichen Typs zu Einheiten zusammenzufassen. Das naheliegendste Beispiel für derlei Probleme sind Personaldaten.

```
struct Person
{   char name[25];
    char vorname[20];
    int alter;
}
```

**Abb. 6.10:** *Strukturdeklaration*

Wer die Daten seiner Angestellten, seiner Freundinnen, seiner Schuldner oder Gläubiger oder von wem auch immer verwalten will, der muß sich zu jeder Person unterschiedliche Informationen merken. So hat jede Person einen Namen und ein Alter. Den Namen eines Menschen muß man im Computer als String speichern; das Alter sollte man jedoch als Zahl speichern, damit numerische Operationen (z.B. Altersvergleiche) damit möglich sind. Name und Alter einer Person gehören zusammen; es wäre deshalb wünschenswert, wenn sie auch im Computer gemeinsam gespeichert und als Einheit behandelt werden könnten.

Dazu gibt es in C die Strukturen. Die Abbildung 6.10 zeigt, wie eine Struktur deklariert werden kann, die alle gewünschten Informationen aufnimmt.

Diese Deklaration teilt mit, daß die Struktur *Person* drei Bestandteile hat: einen Zeichen-Array zur Aufnahme des Namens (in den maximal 24 Zeichen lange Namen passen), einen Array für den Vornamen und eine Integer für das Alter. Hinter dem C-Schlüsselwort *struct* folgt der Name, den Sie der Struktur geben wollen und dann in geschweiften Klammern eine Aufzählung der Komponenten, aus denen sich die Struktur zusammensetzen soll. Die Komponenten werden ähnlich wie Variable deklariert, sie erhalten einen Namen und eine Typangabe.

Die Deklaration in der Abbildung 6.10 reserviert jedoch noch keinerlei Speicherplatz. Sie stellt nur eine Schablone dar, die man für die Definition von passenden Variablen benutzen kann. Steht in einem Programm die Struktur-Schablone der Abbildung 6.10, dann kann man zwei Variablen mit dieser Struktur wie folgt definieren:

```
struct Person vater;
struct Person mutter;
```

Damit stehen zwei Variable *vater* und *mutter* bereit, die die Struktur aufweisen, welche in Abbildung 6.10 beschrieben wurde. Jede dieser beiden Variablen verfügt also über ihre eigene *name*-Komponente; in jeder ist Platz für eine Integer *alter* und für 20 Zeichen eines Vornamens reserviert. In diese Struktur-Variablen können jetzt geeignete Informationen eingetragen werden. Es wäre auch möglich gewesen, die Deklaration der Variablen so zu besorgen:

```
struct
{   char name[25];
    char vorname[20];
    int alter;
} vater, mutter;
```

Der Unterschied zwischen diesen beiden Varianten: in der Abbildung 6.10 wird eine Strukturschablone vereinbart und mit einem Namen (*Person*) versehen. Dieser Name kann später im Programm wiederverwendet werden, um Variable mit der gewünschten Struktur zu definieren. Die zweite Möglichkeit faßt die Strukturerklärung und das Definieren der Variablen zusammen. Diesmal unterbleibt jedoch die Vereinbarung eines Namens für die Struktur. Möchte man an späterer Stelle im Programm erneut Variablen mit dieser Struktur definieren, dann muß man dazu die gesamte Strukturschablone wiederholen.

Eine dritte Möglichkeit ergibt sich, wenn man die Vereinbarung der Strukturschablone, eines Namens dafür und der Variablen dieses Typs zusammenfaßt, also alle Zutaten in einen Topf wirft:

```
struct Person
{   char name[25];
    char vorname[20];
    int alter;
} vater, mutter;
```

Jetzt ist es möglich, an anderer Stelle im Programm weitere Variablen zu definieren, etwa mit

```
struct Person sohn, tochter;
```

Sie sehen: das Schlüsselwort *struct* wirkt zusammen mit dem selbstgewählten Strukturnamen *Person* als syntaktische Einheit, die in Definitionen ähnlich eingesetzt werden kann wie die elementaren Typbezeichnungen *int*, *char*, *float* und *double*.

Mit der Sichtbarkeit von Strukturnamen verhält es sich übrigens genauso wie mit der Sichtbarkeit von Variablen: ist die Struktur global (also nicht innerhalb eines Blocks) beschrieben und benannt, dann ist der Name von da ab in der gesamten Programmdatei bekannt. Lokal beschrieben Strukturen haben denselben Gültigkeitsbereich wie lokale Variablen.

Strukturen sind zusammengesetzte Datentypen; da ist die Frage von Bedeutung, wie man an die einzelnen Komponenten herankommen kann. Um beispielsweise die Namens-Komponente der Variablen *vater* anzusprechen, schreibt man:

```
vater.name
```

Hinter dem Namen der Struktur-Variablen folgt, durch einen Punkt abgetrennt, der Name der gewünschten Komponente: Strukturkomponenten werden über ihren Namen angesprochen. Der Punkt dient hier als Operator, der zur Auswahl einer Komponente der Struktur dient. Wenn Sie die Vorrangtabelle der Abbildung 4.9 betrachten, werden Sie feststellen, daß er höchste Priorität genießt. In einer Anweisung würde der Zugriff auf eine Strukturkomponente so aussehen:

```
strcpy(vater.name, "Aterkant");
strcpy(vater.vorname, "Hein W.");
vater.alter = 27;
```

Damit ist die Variable *vater* mit allen nötigen Daten versorgt. Da die *alter*-Komponente der *Person*-Struktur eine Integer ist, kann man ihr durch einfache Zuweisung einen Wert verpassen. Die beiden andern Komponenten

sind Strings und lieben's etwas umständlicher; sie bekommen ihren Wert mit *strcpy* einkopiert.

```
struct Person
{   char name[25];
    char vorname[20];
    int alter;
};


main()

{   struct Person vater, mutter;

    puts("\nBitte Daten zu \"vater\" eingeben:");
    fill_pers(&vater);
    puts("\nBitte Daten zu \"mutter\" eingeben:");
    fill_pers(&mutter);
    puts("\nIhre Daten:");
    show_pers(&vater); show_pers(&mutter);
}


fill_pers(pp)
        struct Person *pp;

{   char buff[50];
    int atoi();
    char *gets();

    puts("\nVorname:");
    gets(pp->vorname);
    puts("\nName:");
    gets(pp->name);
    puts("\nAlter:");
    pp->alter = atoi(gets(buff));
}


show_pers(pp)
        struct Person *pp;

{   printf("\nName: %s, Vorname: %s, Alter: %d",
            pp->name,
            pp->vorname,
            pp->alter);
}
```

```
/***************************/
/* Funktionen und Struk-   */
/*  turen.                 */
/***************************/
/*                         */
/* Funktionen duerfen nur  */
/*  Pointer auf Struktu-   */
/*  ren als Argument er-   */
/*  halten; deshalb mit    */
/*  "&" Adresse besorgen!  */
/***************************/

/***************************/
/*  Struktur fuellen.      */
/***************************/
/*                         */
/* Fuer die Zahleneingabe. */
/* Umwandeln String ->Zahl */
/*                         */
/*                         */
/*                         */
/*                         */
/*                         */
/*                         */
/* Zuweisung moeglich,     */
/*  weil "int".            */
/***************************/

/***************************/
/* Struktur zeigen.        */
/***************************/
/* So geht's am einfach-   */
/*  sten!                  */
/*                         */
/*                         */
/***************************/
```

**Prog. 6.3:** *Funktionen und Strukturen.*

Natürlich können Struktur-Komponenten nicht nur links von der Zuweisung vorkommen. Um die *alter*-Komponente der Struktur *mutter* auf den gleichen Wert wie die von *vater* zu setzten, schreibt man:

```
mutter.alter = vater.alter;
```

Um bequem mit *Person*-Strukturen arbeiten zu können, empfiehlt es sich, eine Funktion zu schreiben, die sie mit Werten versorgt. Nun ist es in C nicht möglich, einer Funktion eine Struktur als Parameter zu übergeben; dafür sind Zeiger auf Strukturen erlaubt. Die Situation entspricht der bei den Arrays: auch Arrays können nicht an Funktionen übergeben werden, sondern nur Zeiger darauf. Wie man mit Pointern auf Strukturen arbeitet, zeigt das Programm 6.3.

Das Hauptprogramm benutzt zwei selbstdefinierte Funktionen: *fill_pers*, um eine *Person*-Struktur mit Werten zu versorgen und *show_pers*, um die Werte anzuzeigen. Zwei Strukturvariable *vater* und *mutter* sind in diesem Programm definiert. Da das Strukturschema global vereinbart und benannt ist, kann man es in allen nachfolgenden Funktionen verwenden, also auch zur Definition der beiden Variablen und der Funktionsparameter.

Weil Strukturen nicht direkt an Funktionen übergeben werden können, ist es nötig, mit *&* ihre Adresse zu besorgen (die also ein Pointer auf eine *Person*-Struktur ist!). Interessant ist nun, wie die Funktionen mit diesem Struktur-Pointer umspringen.

Betrachten Sie dazu die Funktion *fill_pers*. Der Deklaration ihres Parameters *pp* können Sie unschwer entnehmen, daß es sich dabei um einen Pointer auf eine Person-Struktur handelt. Um an die Struktur heranzukommen, auf die der Pointer verweist, muß man also

```
*pp
```

schreiben. Um an die *name*-Komponente heranzukommen, wäre entsprechend

```
(*pp).name
```

erforderlich. Die Klammern sind notwendig, da der Punkt-Operator stärker bindet als der Stern für die Dereferenzierung. Da in C sehr häufig mit Pointern auf Strukturen gearbeitet wird, hat man sich dafür eine kürzere Notation einfallen lassen:

```
(*pp).name  entspricht  pp->name.
```

der Operator -> erlaubt es also, bequem die Komponenten von Strukturen anzusprechen, auf die ein Pointer verweist.

Da die ersten beiden Komponenten der *Person*-Struktur Strings sind, ist es möglich, sich ihren Wert mit *gets* zu besorgen. Beachten Sie jedoch, daß *gets* keinerlei Längenüberprüfung vornimmt. Für die *name*-Komponente sind 25 Zeichen vereinbart; *gets* wird jedoch bereitwillig mehr als 25 Zeichen von der Tastatur akzeptieren und sie irgendwohin schreiben - bloß

wohin, das kann man nicht sagen. Es besteht eine gewisse Chance, daß die Zeichen in lebenswichtige Teile Ihres Programms oder gar des Betriebssystems geschrieben werden. Das Beispielprogramm ist also alles andere als robust!

Es soll aber auch nur den Umgang mit Strukturen demonstrieren. Die *alter*-Komponente ist laut Deklaration eine Integer. Um eine Integer von der Tastatur einzugeben, sind zwei Schritte nötig: einmal, die Ziffern von der Tastatur zu lesen und zum zweiten, sie in die intern verwendete Binärdarstellung umzuwandeln. In jeder Programmiersprache - auch in BASIC - sind diese beiden Schritte nötig. BASIC versteckt mit seiner bequemen INPUT-Anweisung lediglich den nötigen Aufwand vor dem Benutzer.

Natürlich wäre es möglich, mit der *scanf*-Funktion zu arbeiten, um die Integer zu erhalten. Aber *scanf* ist eine sehr leistungsfähige und deshalb auch umfangreiche Funktion: sie macht das Programm größer als nötig. Der Trend geht in C zu maßgeschneiderten, möglichst kleinen und effizienten Funktionen.

Die Lösung hier: das Einlesen der Ziffern besorgt die bereits bekannte Funktion *gets*. Da diese einen String benötigt, in dem sie die Zeichen ablegen kann, wird in *fill_pers* ein lokaler Array *buff* für diesen Zweck bereitgestellt. Das Umwandeln in eine Zahl besorgt eine Spezialfunktion aus der Standardbibliothek: *atoi* (Abkürzung für: "ASCII to Integer") wandelt einen String von ASCII-Ziffern in eine Integer um. Da *gets* nicht nur Zeichen einliest, sondern - Sie sehen es an seiner Deklaration in *fill_pers* - auch einen Zeiger auf den Anfang des Strings zurückgibt, *atoi* aber ebendiesen Zeiger als Argument erwartet, kann man die beiden Funktionen einfach ineinanderschachteln.

### 6.3.1 Verschachteln von Strukturen

Mit Arrays und Strukturen ist die Bildung beliebig komplexer Datenstrukturen möglich. Dabei hängt es von der Problemstellung ab, für welche Möglichkeit man sich entscheidet. Denn beide haben ihre Stärken und Schwächen. In Arrays dürfen nur gleichartige Komponenten enthalten sein; das ist eine Einschränkung, die die Arrays dadurch ausgleichen, daß sie über Indizes bzw. Pointer, allgemeiner: über berechnete Ausdrücke, den Zugriff auf ihre Komponenten gestatten. Die Verarbeitung von Array-Komponenten in Schleifen ist dadurch sehr einfach. In Strukturen dürfen beliebige Komponenten miteinander gemischt werden; darin sind sie den Arrays überlegen. Der Zugriff auf ihre Komponenten erfolgt jedoch über Namen und diese können nicht berechnet werden; sie müssen im Programm explizit angegeben werden.

Oft erreicht man die beste Lösung eines Problems, indem man alle Möglichkeiten gemischt einsetzt. Daß Strukturen Arrays als Komponenten enthalten können, wurde am Beispiel der Abbildung 6.10 bereits deutlich: *name* und *vorname* sind Arrays. Es sind jedoch noch weitere Kombinationen denkbar, insbesondere: Arrays von Strukturen und Strukturen von Strukturen.

All dies ist einfacher, als es sich anhört. Die Abbildung 6.11 zeigt neben der bereits bekannten *Person*-Struktur die Beschreibung einer weiteren Struktur *Anschrift*, in der eine Adresse gespeichert werden kann. Bis auf die Postleitzahl sind alle Komponenten Strings. Es ist sinnvoll, die Postleitzahl als Integer zu speichern, da mit ihr numerische Auswertungen möglich sein sollen. Wer Anschriften in solchen Strukturen verwaltet, der könnte sie sich dann z.B. nach Postleitzahlen sortiert ausgeben lassen.

```
struct Person
{   char name[25];
    char vorname[20];
    int alter;
};

struct Anschrift
{   char strasse[40];
    int plz;
    char ort[40];
};

struct Angestellter
{   struct Person pers;
    struct Anschrift ans;
    int dienstalter;
};
```

**Abb. 6.11:** *Verschachtelte Strukturen*

Von eigentlichem Interesse ist aber die dritte Struktur in der Abbildung. Auch sie hat drei Komponenten, doch die ersten beiden sind Strukturen! Anstatt Namen und Anschrift eines Angestellten noch einmal in allen Einzelheiten aufzuführen, werden einfach die beiden bereits bekannten Strukturen *Person* und *Anschrift* hier eingebunden. Fragt sich wieder, wie man einzelne Komponenten anspricht. Dazu muß jedoch zuerst eine Variable vom passenden Typ definiert werden:

```
struct Angestellter a1;
```

Folgende Ausdrücke sprechen alle Komponenten von *a1* an:

| | |
|---|---|
| `a1.pers.name` | wählt den Namen |
| `a1.pers.vorname` | wählt den Vornamen |

| | |
|---|---|
| `a1.pers.alter` | wählt das Alter |
| `a1.ans.strasse` | wählt die Strasse |
| `a1.ans.plz` | wählt die Postleitzahl |
| `a1.ans.ort` | wählt den Ort |
| `a1.dienstalter` | wählt das Dienstalter |

Während der Zugriff auf Einzelkomponenten über einen zweistufigen Namen umständlich erscheint, hat das Verfahren den Vorteil, daß die Teilstrukturen von *Angestellter* als Einheiten behandelt werden können. Um die Informationen über Name, Vorname und Alter in die Variable *a1* einzutragen, kann man die aus dem Programm 6.3 bekannte Funktion *fill_ pers* benutzen. Der dazu nötige Aufruf sieht so aus:

```
fill_pers(&(a1.pers));
```

Mit *a1.pers* wird die gesamte Teilstruktur ausgewählt. Eine Struktur kann jedoch nicht als Argument an eine Funktion übergeben werden; erlaubt sind nur Pointer, also muß zuerst der Adreß-Operator & angewendet werden.

Sie sehen: an verschachtelten Strukturen ist nichts Geheimnisvolles.

Will ein Betrieb seine Angestellten tatsächlich mit dieser Struktur verwalten, dann müßte er für jeden Angestellten eine entsprechende Variable definieren, in der dann dessen personenbezogene Daten abgelegt werden. Bei zehn Angestellten könnte man die Variablen *a1, a2,... a10* definieren, aber das ist ein äußerst ungeschicktes Vorgehen.

Denn man hat es in diesem Fall ja mit einer Ansammlung gleichartiger Komponenten zu tun und für diesen Fall sind die Arrays da! Es bietet sich daher an, die Daten für die Angestellten in einem Array zusammenzufassen, der sich aus Strukturen vom Typ *Angestellter* zusammensetzt.

Angenommen, eine Firma hat zehn Angestellte; dann könnten deren Daten in einem Array aufbewahrt werden, der wie folgt definiert wird:

```
struct Angestellter Personal[10];
```

Dies teilt dem Compiler mit, daß *Personal* ein Array mit 10 Komponenten ist, deren jede eine Struktur vom Typ *Angestellter* ist. Jetzt ist es möglich, die Angaben zur Person jedes Angestellten in einer Schleife einzulesen:

```
.
int i;

for (i = 0; i < 10; ++i)
    fill_pers(&(Personal[i].pers));
.
```

Mit *Personal[i]* erhält man die *i*-te Komponente des Arrays; diese ist eine Struktur, von der die Teilkomponente *pers* benötigt wird (siehe Strukturbeschreibung von *Personal* in der Abbildung 6.11!). Diese Teilkomponente kann an die Funktion *fill_pers* zur Datenversorgung weitergegeben werden, indem man sich mit *&* ihre Adresse besorgt.

Erinnern Sie sich: der Name eines Arrays ist in C ein Zeiger auf den Anfang des Arrays, genauer: ein Zeiger auf die erste Komponente des Arrays. Deshalb funktioniert das Einlesen der Werte in den *Personal*-Array auch auf folgende Weise:

```
.
.
.
int i;
struct Angestellter *ap;

ap = Personal;

for (i = 0; i < 10; ++i)
    fill_pers(&ap++->pers);
```

Zugegeben: das ist schon die hohe Schule der klammerfreien Schreibweise. Aber alles ist ganz logisch, wenn man's Schritt für Schritt betrachtet. *Personal* ist eine Array mit *Angestellter*-Strukturen, ist also selbst ein Zeiger auf eine solche Struktur. Der Pointer *ap* ist vom gleichen Typ, er kann also den Array zugewiesen bekommen; dies ist der erste Schritt. In der *for*-Schleife muß nun über diesen Pointer die *Person*-Komponente der einzelnen Array-Bestandteile herausgeholt und anschließend der Pointer *ap* inkrementiert werden. Um eine Komponente einer Struktur zu erhalten, auf die ein Pointer zeigt, benutzt man den Pfeil-Operator; dies führt zu:

```
ap->pers
```

Nun nützt diese Teilstruktur der Funktion *fill_pers* nichts; sie erwartet einen Zeiger auf eine Struktur, den man mit *&* erhält. Damit hat man:

```
&ap->pers
```

Klammern sind nicht nötig, da der *->*-Operator stärker bindet als das adreßbeschaffende *&*. Nun soll jedoch anschließend daran der Pointer inkrementiert werden; eine Aufgabe, wie geschaffen für das Postinkrement:

```
&ap++->pers
```

Wieder braucht's keine Klammern, denn von allen hier vorkommenden Operatoren bindet *++* am schwächsten, kommt also als letztes zum Zuge!

```
struct Person
{   char name[25];
    char vorname[20];
    int alter;
} ;

struct Anschrift
{   char strasse[40];
    int plz;
    char ort[40];
} ;

struct Angestellter
{   struct Person pers;
    struct Anschrift ans;
    int dienstalter;
} ;

#define PERS 10

struct Angestellter Personal[PERS];

                                        /*************************/
main()                                  /*  Arrays von Strukturen. */
                                        /*************************/
{                                       /*                       */
    int i;                              /*                       */
    struct Angestellter *ap;            /*                       */
                                        /*                       */
    for (i = 0; i < PERS; ++i)          /* Zugriff auf Komponenten */
       fill_pers(&(Personal[i].pers));  /*  des Array ueber Index. */
                                        /*                       */
    ap = Personal;                      /* Pointer auf Arrayanfang */
                                        /*                       */
    for (i = 0; i < PERS; ++i)          /* Zugriff ueber Pointer.  */
       show_pers(&ap++->pers);          /*                       */
                                        /*                       */
}                                       /*************************/



                                        /*************************/
fill_pers(pp)                           /*  Struktur fuellen.    */
        struct Person *pp;              /*************************/
                                        /*                       */
{   char buff[50];                      /* Fuer die Zahleneingabe. */
    int atoi();                         /* Umwandeln String ->Zahl */
    char *gets();                       /*                       */
                                        /*                       */
    puts("\nVorname:");                 /*                       */
    gets(pp->vorname);                  /*                       */
    puts("\nName:");                    /*                       */
    gets(pp->name);                     /*                       */
    puts("\nAlter:");                   /*                       */
    pp->alter = atoi(gets(buff));       /* Zuweisung moeglich,   */
}                                       /*  weil "int".          */
                                        /*************************/
```

```
show_pers(pp)                                   /**************************/
        struct Person *pp;                      /* Struktur zeigen.       */
                                                /**************************/
{   printf("\nName: %s, Vorname: %s, Alter: %d", /* So geht's am einfach-  */
            pp->name,                           /*  sten!                 */
            pp->vorname,                        /*                        */
            pp->alter);                         /*                        */
}                                               /**************************/
```

**Prog. 6.4:** *Array mit Strukturen*

Klar, wenn Sie sowas einem Anfänger zeigen, dann fällt der in Ohnmacht! Auch ist es bestens geeignet, bereits vorhandene Vorurteile gegen C massiv zu zementieren. Doch in C werden viele clevere Programme von cleveren Leuten geschrieben; und die verwenden diese kompakte Ausdrucksweise. Es ist ja auch kein schmutziger Programmiertrick, dessen man sich schämen müßte, sondern zeigt lediglich, daß man die Sprache vollkommen beherrscht - und darauf kann man auch ein wenig stolz sein! Im Programm 6.4 sind alle in diesem Kapitel vorkommenden Einzelinformationen noch einmal zusammengetragen. Im Hauptprogramm werden zuerst Daten in den *Personal*-Array eingelesen, wobei die einzelnen Komponenten über Array-Index angesprochen sind. Anschließend können Sie die eingelesenen Daten zur Kontrolle noch einmal betrachten. Diesmal erfolgt der Zugriff über Pointer.

## 6.4 Initialisierung

Die Daten, mit denen Programme arbeiten sollen, stehen manchmal schon von Anfang an fest. Oft ist es auch wünschenswert, wenn Variablen einen bestimmten Startwert besitzen, wenn sie also bereits zu Beginn eines Programms mit einem definierten Wert versorgt sind. Ein Beispiel für diesen Fall haben Sie im Kapitel 5 gesehen. Dort ging es um eine Funktion, die Fehlermeldungen verwalten sollte. Die Fehlermeldungen waren in einem Array mit Zeichen-Pointern gespeichert. Dessen Inhalt stand schon bei Programmbeginn fest. Deshalb wäre es umständlich gewesen, ihn durch Anweisungen des Programms erst mit Daten zu füllen. Die Lösung: der Array wurde initialisiert.

Es gibt zwei grundsätzliche Methoden, mit denen Daten in Programmvariable gelangen können: per Zuweisung oder per Initialisierung. Die Zuweisung speichert die Daten zum Ausführungszeitpunkt des Programmes in den Variablen. Sollen Daten berechnet oder erst durch den Benutzer geliefert werden, dann ist die Zuweisung die Methode der Wahl.

Bei der Initialisierung besorgt der Compiler die anfängliche Versorgung der Variablen mit Daten. Wenn das Programm startet, dann sind diese in den Variablen also schon enthalten. Es werden keine zusätzlichen Programm-Anweisungen mehr benötigt, um die Daten zu erhalten; dies beschleunigt das Programm.

In C bewerkstelligt man die Initialisierung einer Variablen, indem man bei ihrer Deklaration den gewünschten Startwert angibt. Dabei muß man zwischen einfachen und zusammengesetzten Datentypen unterscheiden. Die Abbildung 6.12 stellt die Initialisierung zweier Variablen einer anfänglichen Wertzuweisung gegenüber.

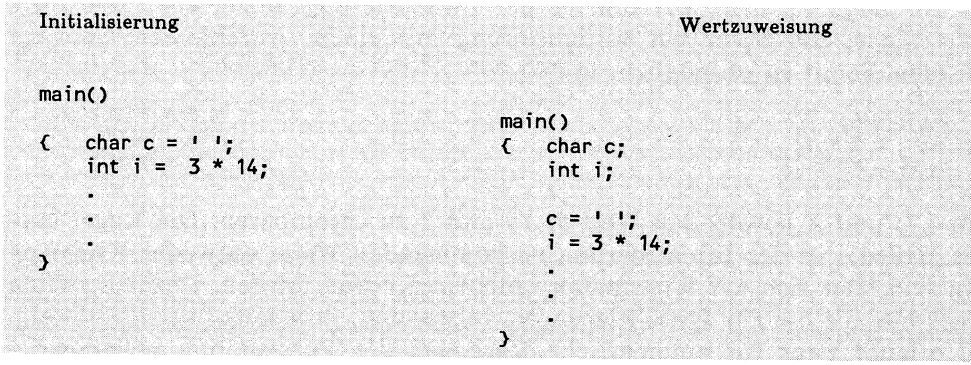



**Abb. 6.12:** *Initialisierung vs. Wertzuweisung*

Die Initialisierung sieht aus wie eine Zuweisung, die sich direkt an die Deklaration der Variable anschließt, die also noch Teil der Deklarationsanweisung ist.

Mit beiden Varianten - Initialisierung oder Zuweisung - wird derselbe Effekt erzielt: im restlichen Programm (symbolisiert durch die drei Punkte) haben die Variablen *c* und *i* einen definierten Anfangswert (das Leerzeichen bzw. die Zahl 42). Doch bei der Initialisierungs-Variante hat sich der Compiler darum gekümmert, daß in den Variablen der richtige Wert steht: er hat bei ihrer Deklaration nicht nur Speicherplatz reserviert, sondern auch gleich den gewünschten Wert in diesen Platz geschrieben. Ohne Initialisierung wäre der Wert der Variablen wie üblich undefiniert (was in den mehrfach verwendeten Speicherdiagrammen durch ein Fragezeichen symbolisiert wurde).

Bei der Zuweisungs-Variante starten die beiden Variablen mit einem undefinierten Wert. Erst durch die nachfolgende Zuweisung erhalten sie ihren Startwert. Jedesmal, wenn das Programm ausgeführt wird, müssen diese Zuweisungen erst durchlaufen werden. Selbstverständlich können Variablen,

die durch Initialisierung einen Anfangswert erhalten haben, anschließend durch Zuweisung geändert werden.

Es ist möglich, einfache Deklarationen und Initialisierungen zu mischen. Sie können also schreiben:

```
int i, j = 2, k;
```

Damit werden drei Integer-Variable deklariert, wobei der Wert von *i* und *k* undefiniert ist; *j* erhält den Startwert 2.

Bei der Initialisierung sind Sie nicht auf Konstanten als Werte beschränkt. Es ist möglich, auch Variable in der Initialisierung anzugeben, allerdings nur solche, die zuvor per Initialisierung mit einem Anfangswert versorgt wurden. Somit ist es möglich, mit

```
int i = 5;
int j = 3 * i;
int k = 32 % j;
```

für *i*, *j* und *k* jeweils den Wert 5, 15 und 2 zu vereinbaren. Die Regel lautet, daß die in der Initialisierung vorkommenden Werte entweder Konstanten sein oder sich auf Konstanten reduzieren lassen müssen. Deshalb sollten Variable, die Sie für Initialisierungen verwenden, ihrerseits initialisiert sein. (C erlaubt zwar für automatische Variablen die Verwendung beliebiger - auch noch nicht initialisierter - Variablen im Initialisierungs-Ausdruck, doch wird dadurch der Wert der initialisierten Variablen undefiniert. Sie sollten daher auf diese Möglichkeit verzichten).

Pointer-Variablen können auch auf die Adresse beliebiger zuvor deklarierter Variablen gesetzt werden. Das folgende Demonstrations-Programm druckt daher den Buchstaben 'A':

```
main()
{   char c;
    char *cp = &c;

    c = 'A';
    putchar(*cp);
}
```

Der Pointer *cp* zeigt durch die Initialisierung auf die Zeichen-Variable *c*; diese erhält ihren Wert per Zuweisung. Der wird durch Dereferenzierung über *cp* auf den Bildschirm gebracht.

Bei einfachen Datentypen gibt es in C außer den eben genannten keinerlei Einschränkungen. Ob lokal oder global, ob automatisch, statisch oder Register-Variable: alle können sie initialisiert werden. Anders ist es bei zusammengesetzen Datentypen. Für solche Variablen ist die Initialisierung nur

erlaubt, wenn sie global oder statisch sind. Dies war der Grund, warum im Programm 5.15 der Array mit den Fehlermeldungen global deklariert und initialisiert wurde (denn C erlaubt keine statischen Arrays).

Bei der Initialisierung einer zusammengesetzten Variablen (also eines Arrays oder einer Struktur) geht man folgendermaßen vor: hinter das Initialisierungs-= schreiben Sie eine öffnende geschweifte Klammer, dann für jede Komponente der Struktur einen Initialisierungswert und schließen das ganze mit einer schließenden geschweiften Klammer ab. Die einzelnen Werte für die Initialisierung im Klammerpaar müssen Sie durch Komma trennen.

Um eine Variable *p1* mit der im letzten Abschnitt vorgestellten *Person*-Struktur zu initialisieren, schreibt man also:

```
struct Person p1 =
{   "Stechlich",
    "Bernhard",
    45
} ;
```

Enthält eine zusammengesetzte Variable Komponenten, die ihrerseits zusammengesetzt sind, dann wird die obige Regel einfach wiederum auf jede zusammengesetzte Komponente angewandt. Hier ein Beispiel für die Initialisierung einer *Angestellter*-Variablen:

```
struct Angestellter a1 =
{   {   "Aterkant",
        "Hein W."
        35
    } ,
    {   "Unterm Deich 14",
        2001,
        "Ovelkamp"
    } ,
    1
}
```

Hieran können Sie nicht nur ersehen, daß Hein W. Aterkant 35 Jahre alt ist, sondern auch, wo er wohnt und daß er dem Betrieb, der ihn in dieser Struktur im Computer speichert, erst seit einem Jahr angehört.

(**Anmerkung**: Die mir vorliegende Fassung des ATARI-Compilers wurde mit der Initialisierung verschachtelter Strukturen nicht fertig. Sie können also die Beispiele in der vorliegenden Form nicht nachvollziehen, falls Sie mit derselben Version des Compilers arbeiten.)

Bei der Initialisierung von zusammengesetzten Variablen müssen Sie daran denken, daß nur globale sowie externe und statische Variablen initialisiert werden dürfen. Deshalb ist folgendes Programmfragment falsch:

```
main()

{   char array[80] = "Hallihallo!";
    .
    .
    .
}
```

Die Variable *array* ist - nomen est omen - zusammengesetzt; und sie ist eine automatische Variable. Deshalb spielt hier der Compiler nicht mit. Allerdings ist folgendes möglich:

```
main()

{   char *array = "Hallihallo!";
    .
    .
    .
}
```

Die Variable *array* ist - nix "nomen est omen" - jetzt ein einfacher Pointer. Der darf mit einer passenden Konstanten, also mit einer Stringkonstanten, initialisiert werden. Wer jetzt glaubt, ein Schlupfloch durch die Initialisierungs-Einschränkung für Strukturen gefunden zu haben, den muß ich enttäuschen: es gibt keine 'Struktur-Konstanten' in C, deshalb kann man diesen Trick auch nicht auf einen Struktur-Pointer anwenden!

Schließlich geht auch noch dieses:

```
struct Angestellter a1 =
{   {   "Aterkant",
        "Hein W.",
        35
    } ,
    {   "Unterm Deich 14",
        2001,
        "Ovelkamp"
    } ,
    1
} ;

main()

{   char *cp = a1.pers.name;
    .
    .
    .
}
```

Denn: *a1* ist initialisiert, *a1.pers.name* ist somit ein String und den kann ich per Initialisierung dem Pointer *cp* zum Drauf-Zeigen verpassen.

## 6.5 Dynamische Speicherverwaltung

Wenn Sie eine Variable definieren, dann reserviert der Compiler Platz dafür im Arbeitsspeicher des Computers. Wenn Sie die Variable auch noch initialisieren, dann schreibt er an diesen Platz zusätzlich den gewünschten Wert.

Der Platzbedarf einer Struktur-Variablen hängt ab vom Platzbedarf ihrer Einzelkomponenten plus einiger protokollarischer Gepflogenheiten, auf deren Einhaltung der Prozessor besteht. Hierzu ein Beispiel. Die Struktur

```
struct
{   char strng[3];
    int i;
} stru;
```

setzt sich aus einem dreielementigen Zeichen-Array und einer Integer zusammen. Zeichen sind ein Byte lang, Integers beanspruchen ein Wort (entspricht zwei Bytes). Also ergibt sich für die Variable *stru* ein Platzbedarf von 5 Byte. Dennoch reserviert der Compiler für *stru* 6 Bytes, denn er darf eine Wort-Größe (hier: die Integer-Komponente von *stru*) nicht an einer ungeradzahligen Adresse beginnen lassen. Dies liegt in der Adressierungs-Technik der M68000 begründet. Zwischen der letzten Komponente des Strings und der Integer wird also ein Füll-Byte eingeschoben, um die Integer auf eine gerade Adresse zu bringen.

Eine Variable vom Typ *Angestellter* wird daher im Speicher des ATARI 130 Bytes an Platz einnehmen. Nun wurde im letzten Kapitel vorgeschlagen, die Mitarbeiter eines Betriebs in einem Array solcher Strukturen zu verwalten. Bei einem Kleinbetrieb mit 10 Mitarbeitern würde dieser Array 1300 Bytes, bei 100 Mitarbeitern 13000 und bei 1000 Mitarbeitern... aber Rechnen können Sie selber!

Mit den Arrays gibt es nämlich ein Problem: man muß bei ihrer Deklaration die Maximalgröße angeben und hat danach keine Chance mehr, an diesem oberen Grenzwert etwas zu verändern. Nun vermag aber kein einigermaßen optimistisch (oder "gedämpft optimistisch", wie das neuerdings so schön heißt) in die Zukunft blickender Betriebsinhaber eine zuverlässige Prophezeiung über die Maximalzahl seiner Mitarbeiter anzugeben. Denn das Unternehmen könnte, ja sollte sogar, expandieren. Nun einfach eine unwahrscheinlich große Maximalzahl anzugeben ist auch keine Lösung des

Problems. Denn der Compiler wird Platz für alle angegebenen Komponenten reservieren und es kümmert ihn nicht, ob sie jemals wirklich benötigt werden.

Damit wird viel Speicherplatz verschwendet, der auch auf einer damit so üppig ausgestatteten Maschine wie dem ATARI für Besseres verwendet werden könnte. Es wäre deshalb wünschenswert, wenn man sich je nach Bedarf zur Laufzeit des Programmes Platz für neue Komponenten besorgen könnte.

Man kann. Es geht hier um den Unterschied zwischen statischer und dynamischer Speicherverwaltung. Die Speicherverwaltung, die der Compiler besorgt, ist statisch. Er weist Variablen einmal Platz zu - und daran ändert sich dann nichts mehr. Der Platz bleibt reserviert, unabhängig davon, ob er überhaupt noch benötigt wird (was ist zum Beispiel, wenn das Unternehmen Mitarbeiter entlassen muß?). Auch wenn der zugewiesene Platz nicht ausreicht (das Unternehmen stellt Leute ein), dann tut es dem Compiler leid: er kann nichts mehr ändern. Daher das Wörtchen 'statisch'.

Unter einem dynamischen Verwaltungs-Regime ist es möglich, sich während der Laufzeit des Programmes Speicherplatz für zusätzliche Variablen zu besorgen bzw. den Platz für nicht mehr benötigte Variablen wieder freizugeben. Die dynamische Speicherverwaltung ist auch in C möglich. Allerdings sind die hierfür einzusetzenden Sprachkonstrukte kein Teil der Programmiersprache, sondern Funktionen, die Bestandteil der mit dem Compiler ausgelieferten Standard-Bibliothek sind. Die Namen der Funktionen sind leider nicht in allen Bibliotheken einheitlich.

Die dynamische Speicherverwaltung erledigen meist zwei Funktionen im Zusammenspiel: eine allokiert (weist zu) Speicher für zusätzliche Variable; die andere gibt den allokierten Speicher wieder frei. Wahrscheinliche Namen für dieses Funktions-Gespann sind *alloc* und *free* (wer was macht, geht aus den Namen hervor!). Einige Compiler (darunter der des ATARI) unterscheiden noch zwischen einer Funktion für die Platzzuweisung von Strukturen (genannt *malloc*) und für die Platzzuweisung von Arrays (genannt *calloc*).

Ich werde in den nächsten Beispielen mit der Funktion *malloc* arbeiten. Sind Sie mit dieser erst einmal vertraut, dann bereitet Ihnen die Arbeit mit *calloc* keinerlei Probleme mehr. Sollte Ihr Compiler nur über *alloc* verfügen, dann wird dieses sich mit sehr hoher Wahrscheinlichkeit genauso wie das hier beschriebene *malloc* verhalten.

Die Funktion *malloc* benötigt eine Größenangabe über den zu besorgenden Speicherplatz. Diese Größenangabe muß in Bytes erfolgen - aber das ist

genau die Information, die der *sizeof*-Operator liefert! Als Wert liefert *malloc* einen Zeiger auf den Anfang der zugewiesenen Speicher-Region. Diesen Zeiger kann man einer Pointer-Variablen zuweisen und hat somit dauerhaften Zugriff auf den Speicherplatz. Betrachten Sie dazu das folgende Programm-Fragment:

```
main()

{   struct Angestellter *malloc();
    struct Angestellter *ap;

    ap = malloc(sizeof (struct Angestellter));
    .
    .
    .
}
```

Dies hat für den nachfolgenden Programmteil (symbolisiert durch die drei Punkte) die gleiche Wirkung, als hätte man folgendes Geschrieben:

```
main()

{   struct Angestellter a;
    struct Angestellter *ap = &a;
    .
    .
    .
}
```

In beiden Fällen zeigt *ap* auf einen Speicherblock, der groß genug ist, um alle gewünschten Informationen aufzunehmen. Es gibt jedoch einen wesentlichen Unterschied: die Funktion *malloc* kann ich an jeder beliebigen Stelle des Programms und sooft ich will aufrufen und mir damit beliebig viele zusätzliche Variablen besorgen.

Die Deklaration von *malloc* im letzten Beispiel bedarf noch der Erläuterung. Da im ATARI alle Zeiger gleich groß sind, empfiehlt es sich, *malloc* in der Funktion, die es verwendet, so zu deklarieren, als lieferte es einen Pointer auf den gewünschten Datentyp (hier: auf eine Struktur vom Typ *Angestellter*). Dies erfolgt nur aus Gründen der Dokumentation. Allerdings dürfen Sie niemals vergessen, den Typ von *malloc* zu deklarieren (und stets als Zeiger-Typ!). Denn sonst nimmt der Compiler an, daß *malloc* eine Integer zurückgibt (die auf dem ATARI halb so lange wie eine Adresse ist). Da das Ergebnis von *malloc* stets an eine Zeiger-Variable zugewiesen wird, erfolgt die Umwandlung dieser Integer in eine Adresse; was dabei herauskommt, ist mit ziemlicher Sicherheit ein Pointer in lebenswichtige Regionen des Betriebssystems!

Mit der Funktion *malloc* ist es jetzt möglich, das oben aufgeworfene Angestellten-Problem zu lösen. Statt einen Array mit - sagen wir mal - 1000 Angestellten zu definieren, reicht es jetzt aus, einen Array mit 1000 Angestellten-Pointern zu vereinbaren. Da ein Pointer 4 Byte lang ist, ist dessen Platzbedarf ungleich geringer. Jedesmal, wenn ein neuer Angestellter hinzukommt, besorgt man sich mit *malloc* Platz dafür und hängt diesen Speicherplatz in den Array ein; damit ist der Neue erfaßt.

Um das Beispiel möglichst einfach zu halten, habe ich mich auf Strukturen vom Typ *Person* beschränkt. Das Prinzip dieser Lösung wird im Programm 6.5 auch damit erkennbar.

```
#include <stdio.h>

struct Person
{   char name[25];
    char vorname[20];
    int alter;
} ;

#define MAXIMUM 10

struct Person *Personal[MAXIMUM];
                                                /*************************/
                                                /* Dynamische Speicherver- */
main()                                          /*  waltung.             */
                                                /*************************/
{   struct Person *malloc();                    /* "malloc" gibt zwar ei-  */
    int neu = 0;                                /*  nen Zeichen-Pointer    */
    int i;                                      /*  zurueck; aber dies     */
                                                /*  macht klar, wozu es    */
    do                                          /*  hier dient.            */
    {   printf("\nBitte Daten eingeben");       /*                         */
        Personal[neu] = malloc (sizeof (struct Person));  /* Neue Struktur  */
        fill_pers(Personal[neu]);               /*  in den Array einhaen-  */
        printf("\nWeiter (J/N)?");              /*  gen.                   */
    }                                           /*                         */
    while (++neu < MAXIMUM && getchar() == 'J') ; /*                       */
                                                /*                         */
    for (i = 0; i < neu; ++i)                   /* Array-Inhalt anzeigen.  */
        show_pers (Personal[i]);                /*                         */
}                                               /*************************/
```

**Prog. 6.5:** *Dynamische Speicherverwaltung mit* malloc

Im Programm wird ein Array *Personal* mit Pointern auf *Person*-Strukturen global definiert. Immer noch ist es nötig, für die Komponentenzahl des Arrays eine Abschätzung der oberen Grenze der aufzunehmenden Daten vorzunehmen (symbolische Konstante MAXIMUM). Eine Komponente im Array *Personal* benötigt jedoch nur vier Byte Speicherplatz; deshalb ist die Platzverschwendung lange nicht so groß wie bei der letzten Lösung.

Das Hauptprogramm trägt solange neue Daten in diesen Array ein, wie es der Benutzer wünscht bzw. wie es möglich ist (die Arraygröße MAXIMUM darf nicht überschritten werden); dies wird in der Kontrollbedingung der *do*-Schleife überwacht. Die Variable *neu* gibt die Array-Position an, bei der neue Daten eingehängt werden sollen. Da zu Beginn des Programmes noch keine Daten vorhanden sind, muß man mit dem Eintragen am Anfang des Arrays beginnen: *neu* erhält den Anfangswert 0 und wird bei jedem Schleifendurchgang inkrementiert.

Sollen neue Daten eingegeben werden, dann besorgt sich das Programm über *malloc* Speicherplatz von passender Größe. Diese wird mit dem *sizeof*-Operator bestimmt. Den Zeiger, der von *malloc* geliefert wird, hängt man dann durch Zuweisung in den Array ein - und das war's schon! Die bereits bekannte Funktion *fill_pers* kann jetzt diesem Zeiger nachgehen und die von der Tastatur eingegeben Daten eintragen.

Wenn der Benutzer die Dateneingabe abschließen will, oder wenn der Array *Personal* erschöpft ist, dann wird sein Inhalt ausgegeben. Dies geschieht, indem *show_pers* in einer Schleife jedem im Array enthaltenen Pointer nachgeht.

## 6.6 Rekursive Datenstrukturen

Die Themen, die in diesem Abschnitt angesprochen werden, sind in der Informatik von so großer Bedeutung, daß ganze Bände darüber verfaßt wurden. Natürlich ist es im Rahmen dieses Buchs nicht möglich, sie auch nur einigermaßen erschöpfend zu behandeln. Sie werden jedoch sehen, daß sich hinter dem Begriff "Rekursive Datenstruktur" nichts Geheimnisvolles verbirgt und daß in C das Arbeiten damit sehr einfach ist. Einer eingehenderen Beschäftigung mit dieser wichtigen Materie steht dann nichts mehr im Wege.

Ein Großteil der praktischen Bedeutung der Struturen rührt von dem Umstand her, daß Strukturen ineinander verschachtelt werden können. Sie haben in der Struktur *Angestellter* ein Beispiel dafür gesehen; sie enthält zwei Komponente, die selbst Strukturen sind, nämlich *Anschrift* und *Person*.

Wie weit kann dieses Verschachteln von Strukturen gehen? Insbesondere: kann eine Struktur in sich selbst eingeschachtelt sein? Ist also etwas wie das Folgende möglich?

```
struct Endlos
{   char dta[80];
    struct Endlos next;
}
```

Die Antwort auf diese Frage ist "Nein!". Eine Struktur, die sich selbst enthält, wäre - wie der Name des Beispiels suggeriert - tatsächlich endlos. Beim Zuweisen von Speicherplatz für eine verschachtelte Struktur sieht der Compiler bei der Beschreibung jeder Komponente nach, wie sie beschaffen ist und wieviel Platz sie daher benötigt. Wenn er dies bei der Struktur *Endlos* versucht, dann wird er in einer endlosen Schleife gefangen.

Wenn der Compiler für eine Variable dieses Typs Platz besorgen soll, dann reserviert er zuerst die 80 Bytes für den String *dta*. Dann sieht er, daß er Platz für eine weitere Struktur *Endlos* besorgen muß, die in die gerade betrachtete eingeschachtelt ist. Also sucht er die Beschreibung der Struktur *Endlos* auf und reserviert wieder 80 Bytes für den String; dann sieht er, daß er Platz für eine weitere Struktur *Endlos* besorgen muß, die in die gerade betrachtete eingeschachtelt ist (welche ihrerseits in die Ausgangsstruktur eingeschachtelt ist). Also reserviert er erst mal die 80 Bytes für den String; dann sieht er...

Sie sehen: das ginge jetzt endlos so weiter. Darum schiebt der Compiler dem einen Riegel vor: Strukturen, die sich selbst enthalten, sind verboten.

Nicht verboten ist jedoch eine Struktur, die einen Zeiger auf sich selbst enthält! Betrachten Sie einmal die Struktur in der Abbildung 6.13.

```
struct Person
{   char name[25];
    char vorname[20];
    int alter;
} ;


                                        /***************************/
                                        /* Struktur eines Listen-  */
struct Plist                            /*  eintrags.              */
{   struct Person dta;                  /* Daten-Komponente.       */
    struct Plist *next;                 /* Zeiger auf naechsten    */
} ;                                     /*  Knoten.                */
                                        /***************************/
```

**Abb. 6.13:** *Eine rekursive Datenstruktur*

Die Struktur *Plist* hat zwei Komponenten: eine *Person*-Struktur, die unter dem Namen *dta* (Abkürzung für "data" = Daten) angesprochen werden kann und einen Zeiger auf eine *Plist*-Struktur, zugänglich unter dem Namen *next*. Wird verlangt, eine Variable - nennen wir sie *start* - mit dieser Struktur zu definieren, dann besorgt der Compiler Platz für die *Person*-

Komponente und für den Zeiger; mehr ist nicht nötig. Der Zeiger zeigt - wenn er nicht initialisiert ist - erstmal ins Nirgendwo. Die Abbildung 6.14 ist eine graphische Darstellung der Verhältnisse, die nach der Definition der Variablen *start* im Speicher herrschen.

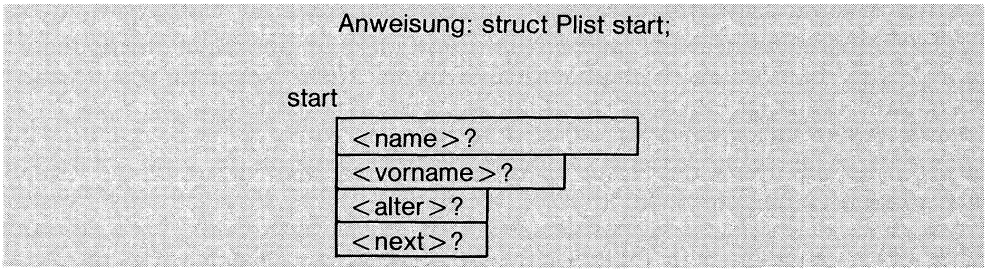



**Abb. 6.14:** *Aufbau einer Liste; Teil 1*

Die Variable *start* ist sehr schematisch dargestellt. Die Komponenten der *Person*-Teilstruktur sind mit ihren Namen beschriftet; sie enthalten jeweils ein Fragezeichen, um anzuzeigen, daß ihr Wert noch unbestimmt ist. Insbesondere hat der Zeiger *next* noch keinen sinnvollen Wert.

Das kann jedoch geändert werden. Es ist in einem Programm ja möglich, sich mit *malloc* eine weiter *Plist*-Struktur zu besorgen und diese an die Variable *start* anzuhängen. Das erledigt die Anweisung

```
start.next = malloc (sizeof (struct Plist));
```

Die Wirkung dieser Anweisung veranschaulicht die Abbildung 6.15. Durch *malloc* wurde eine neue *Plist*-Struktur ins Leben gerufen. Der Zeiger darauf, welcher Wert von *malloc* ist, wurde in der *next*-Komponente von *start* abgespeichert, so daß diese also auf die neue Struktur zeigt (dargestellt durch den Pointer-Pfeil). Beachten Sie jedoch, daß diese neue Struktur namenlos ist: man kommt nur zu ihr, indem man den *next*-Zeiger von *start* verfolgt.

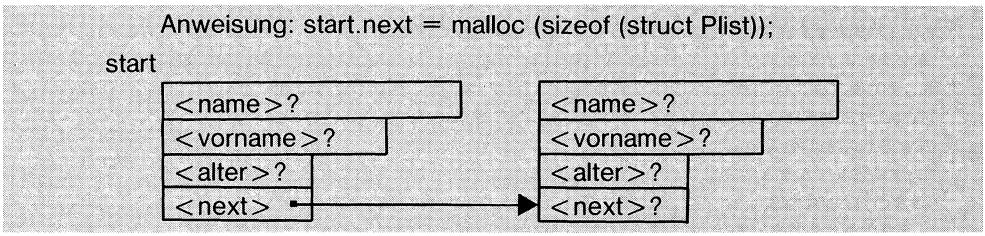



**Abb. 6.15:** *Aufbau einer Liste; Teil 2*

Mit dieser Methode können beliebig viele Strukturen aneinandergekettet werden. Man nennt über Zeiger verknüpfte Gebilde dieser Art auch Listen. Dies ist der Grund, warum ich für die Struktur den Namen *Plist* gewählt habe (Abkürzung für "Personen-Liste"). Listen sind abstrakte Datenstrukturen, die beliebig verlängert (und auch verkürzt; aber darum kümmern wir uns in diesem Kapitel nicht mehr) werden können. Das Verlängern einer Liste erreicht man, indem man an ihr Ende (also in die letzte Komponente der Liste) eine neue Komponenten-Struktur (eine sogenannten Listen-"Knoten") einhängt. In der Terminologie der Informatik besteht eine Liste aus "Knoten", die über Zeiger miteinander verkettet sind. Um einen neuen Knoten an das Ende der Liste einzuhängen, muß man dieses Ende jedoch erst einmal finden.

Dies geht am besten wie folgt. Man vereinbart einen Zeiger auf eine *Plist*-Struktur (hier *plp* genannt) und läßt diesen auf den Listenanfang *start* zeigen:

```
struct Plist *plp = &start;
```

Die Wirkung dieser Anweisung veranschaulicht die Abbildung 6.16.

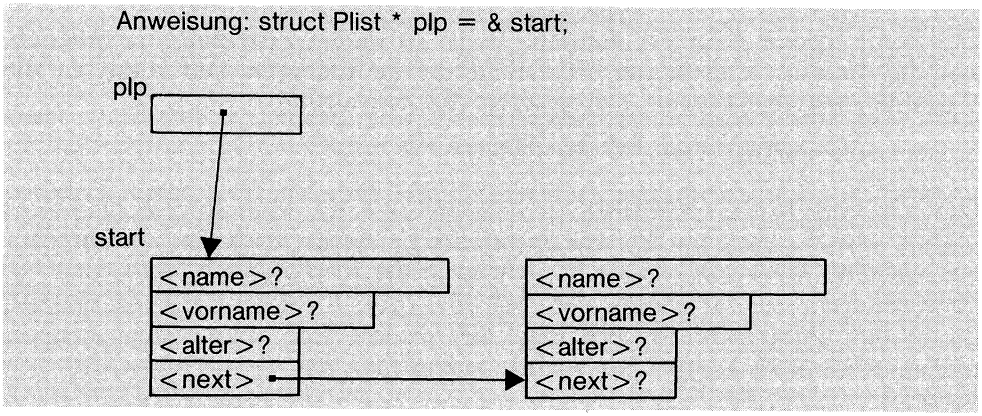



**Abb. 6.16:** *Zeiger über Liste verschieben; Teil 1*

Dann verschiebt man den Zeiger *plp* zum Nachfolger von *start*. Dies ist möglich, ohne den Namen *start* überhaupt zu erwähnen und geht mit folgender Anweisung:

```
plp = plp->next;
```

Wieder können Sie der Abbildung 6.17 entnehmen, welche Wirkung dies hat. Auf die Komponenten-Namen habe ich verzichtet, um mich nur aufs

Wesentliche zu konzentrieren. Der gestrichelte Pfeil, der von *plp* ausgeht, zeigt den Zustand vor der Zuweisung; der durchgezogene Pfeil zeigt die Wirkung der Zuweisung.

Nach dem Verschieben des Zeigers prüft man, ob dieser bereits das Ende der Liste erreicht hat. Aber woran kann man das erkennen? Dafür gibt es eine einfache Lösung: in die *next*-Komponente des letzten Listenelements schreibt man einen Zeiger-Wert, der das Listen-Ende signalisiert und der als 'echter' Zeiger-Wert niemals vorkommen kann. Dies ist der Wert 0. C garantiert dem Programmierer, daß kein vom Compiler eingeführter Zeiger - gleichgültig, ob er über statische oder dynamische Speicherverwaltung seinen Wert erhalten hat - jemals auf die Speicherstelle 0 verweist. Ein Pointer-Wert von 0 muß daher vom Programmierer gesetzt sein; man kann ihn daher verwenden, um eindeutig das Listenende zu markieren.

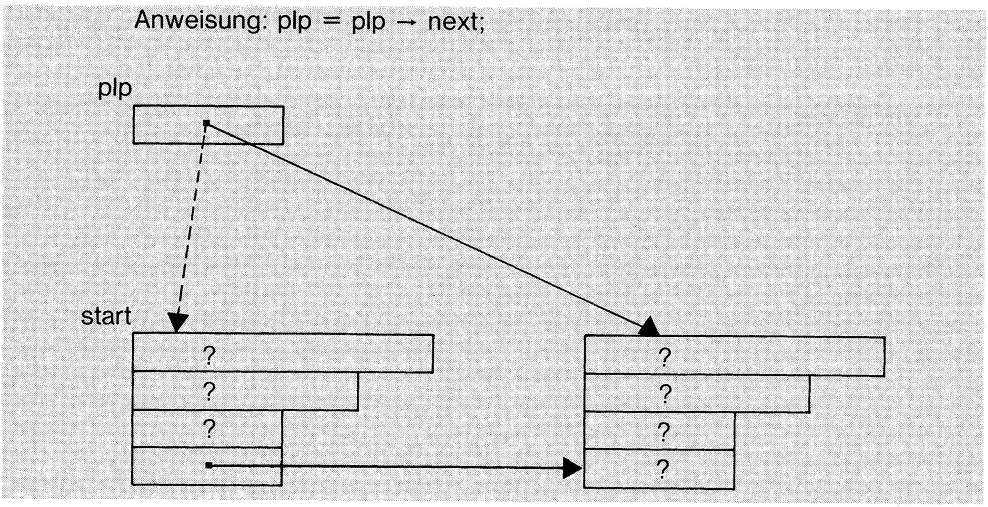



**Abb. 6.17:** *Zeiger über Liste verschieben; Teil 2*

Die Abbildung 6.18 zeigt, wie man in der Situation der Abbildung 6.17 einen neuen Knoten in die Liste einketten und diesen korrekt als Listenende markieren kann. Die Anweisungen dafür lauten:

```
plp->next = malloc (sizeof (struct Plist));
plp = plp->next;
plp->next = (struct Plist*) 0;
```

Wie Sie sehen, sind dazu drei Schritte nötig:

1. neue Komponente besorgen und einhängen,
2. Listen-Zeiger auf neue Komponente verschieben,
3. Ende der Komponente markieren.

Die Ziffern in der Abbildung 6.18 beziehen sich auf diese drei Schritte. Beachten Sie, wie im dritten Schritt mit dem *cast*-Operator aus der Null ein Zeiger vom passenden Typ gemacht wird.

Die Listenstruktur, die hier aufgebaut ist, wird auch "einfach gekettete Liste" genannt. Man kann sich nämlich nur in einer Richtung durch die Liste bewegen: von vorne nach hinten. Sie sehen das im Beispiel daran, daß man von dem neuen (dritten) Listenelement nicht mehr zum zweiten gelangen kann: es führt kein Pointer zurück. Allerdings hindert den Programmierer nichts daran, den Zeiger *plp* wieder auf den Listenanfang zu setzen (das geht, weil dieser einen Namen hat) und ihn von da aus in gewohnter Manier zu verschieben.

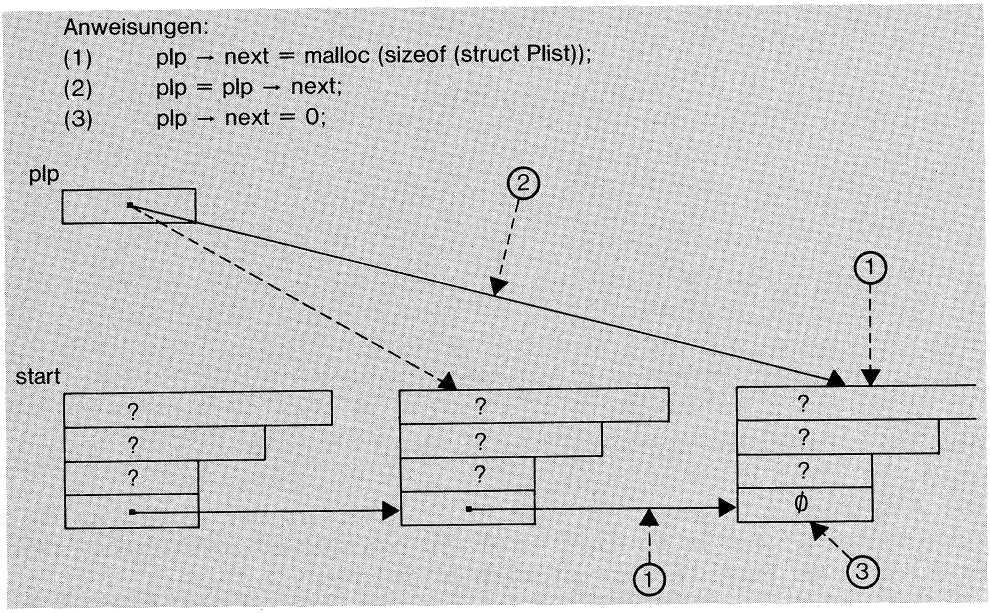



**Abb. 6.18:** *Neues Element ins Listenende einketten.*

Listen sind rekursive Datenstrukturen (daher die Überschrift dieses Kapitels), denn in der Definition eines Listen-Knotens ist ein Verweis auf ebendiesen Knoten enthalten: die Definition von *Plist* enthält einen Verweis auf (in Form eines Pointers auf) die Struktur *Plist*. Die Bedeutung der

rekursiven Datenstrukturen (neben den Listen vor allem die sogenannten "Bäume") für die Programmierung komplexer Probleme ist enorm. Im Rahmen dieses Buchs ist es unmöglich, dem Thema gerecht zu werden. Der Autor legt aber jedem ambitionierten Leser dringend ans Herz, sich mit der Materie zu beschäftigen und dazu eines der zahlreichen Bücher über "Datenstrukturen" zur Hand zu nehmen.

Im Zusammenhang mit dem Leitbeispiel dieses Kapitels - der Verwaltung von Personaldaten - stellen die Listen die Lösung des Speicherplatz-Problems dar. Denn auch in der letzten Variante, die mit einem Pointer-Array arbeitet, ist man immer noch auf die Abschätzung einer Obergrenze für die verwaltbaren Daten angewiesen. Der Pointer-Array wird statisch (vom Compiler) zugewiesen; sollte er sich später als zu klein erweisen, dann ist keine nachträgliche Änderung mehr möglich. Bei Angabe einer unrealistisch hoch erscheinenden Obergrenze hat man immer noch das Problem mit dem Überhang an Speicherplatz, der für die Pointer vom Compiler reserviert wird.

Gänzlich anders sieht die Lage aus, wenn man mit Listen arbeitet. Hier braucht nur bei Bedarf ein neuer Knoten eingehängt zu werden, es wird also immer nur genau soviel Platz vom Arbeitsspeicher abgezwackt, wie auch benötigt wird. Mit der Funktion *free* ist es auch möglich, nicht mehr benötigte Knoten aus der Liste auszuketten und die Liste ohne Speicherplatz-Verschwendung stets auf dem aktuellen Stand zu halten.

Das Beispiel-Programm 6.6 demonstriert, wie man eine Liste mit Personendaten aufbauen kann. Beim Start des Programmes ist bereits ein Knoten definiert; dieser wird *Start* genannt, da er als Listenanfang dient. *Start* ist eine globale Variable, da der Listenanfang aus allen Funktionen, die mit dr Liste zu tun haben, zugänglich sein sollte,

Eine weitere globale Variable ist der Zeiger *End*, der stets auf das aktuelle Listenende zeigt und so das schnelle Einketten neuer Knoten erlaubt. Da bei Programmbeginn erst ein Knoten vorhanden ist, fallen Anfang und Ende zusammen; dies können Sie an der Initialisierung von *End* erkennen.

Das Hauptprogramm nimmt zuerst Daten in die Liste auf. Dazu wird der bereits bestehende Knoten gefüllt und an ihn anschließend ein neuer, leerer Knoten eingehängt (Funktion *new_node*), der mit dem NULL-Pointer als Listenende markiert wird. Das Verfahren hat den Nachteil, daß immer ein Knoten mehr in der Liste hängt, als Daten in ihr enthalten sind, nämlich der letzte Knoten. Dafür ist es einfach zu programmieren! Die Funktion *new_node* besorgt nicht nur einen neuen Knoten, sondern sie verschiebt auch den *End*-Pointer so, daß er auf das neue Ende zeigt. Das funktioniert natürlich nur, weil *End* eine globale Variable ist.

```
#include <stdio.h>

#define NULL (struct Plist *) 0

struct Person

{   char name[25];
    char vorname[20];
    int alter;
} ;


struct Plist
{   struct Person dta;
    struct Plist *next;
} ;


struct Plist Start;

struct Plist *End = &Start;



main()
{
    char c;

    do
    {
        printf("\nBitte Daten eingeben:");
        fill_pers(&End->dta);
        new_node();
        printf("\nWeiter (J/N)");
    }
    while   (getchar() == 'J' );

    show_list(&Start);
}


struct Plist *new_node()

{   struct Plist *malloc();

    End->next = malloc(sizeof(struct Plist));
    End = End->next;
    End->next = NULL;
    return End;
}
```

```
/***************************/
/* Struktur eines Listen-  */
/*  eintrags.              */
/* Daten-Komponente.       */
/* Zeiger auf naechsten    */
/*  Knoten.                */
/***************************/
/*                         */
/* Listenanfang            */
/*                         */
/* Zeigt stets auf das     */
/*  letzte (ungefuellte)   */
/*  Element der Liste.     */
/***************************/


/***************************/
/* Liste fuellen un anzei- */
/*  gen.                   */
/***************************/
/*                         */
/*                         */
/*                         */
/* Daten eintragen.        */
/* Neuen Knoten besorgen.  */
/*                         */
/*                         */
/*                         */
/*                         */
/* Liste ausgeben.         */
/***************************/


/***************************/
/* Neuen Knoten besorgen.  */
/***************************/
/* Nicht vergessen!        */
/*                         */
/* Neuen Knoten einhaengen.*/
/* Ende-Zeiger weiter-     */
/*  schieben und Liste     */
/*  korrekt abschliessen.  */
/***************************/
```

```
show_list (startnode)
          struct Plist *startnode;

{   if(startnode->next == NULL )
        return;

    show_pers (&startnode->dta);

    show_list (startnode->next);
}
```

```
/*************************/
/* Liste ausgeben.       */
/*************************/
/*                       */
/* Wir sind am Ende; also */
/*  aufhoeren, denn letz- */
/*  ter Knoten enthaelt  */
/*  nichts!              */
/* Ansonsten: Daten anzei- */
/*  gen und Rekursion.   */
/*************************/
```

**Prog. 6.6:** *Rekursive Datenstruktur und rekursive Funktion*

Wenn der Benutzer signalisiert, daß er keine Daten mehr aufnehmen will, gibt das Programm die in der Liste gespeicherten Daten aus. Die hierzu vorgesehene Funktion *show_list* demonstriert eine weitere Lieblingstechnik der fortgeschrittenen Programmierkunst: eine rekursive Funktion.

Eine rekursive Funktion ist eine Funktion, die sich selbst aufruft. Das ist in C - wie in jeder ordentlichen Programiersprache; BASIC kann's nicht! - ohne weiteres möglich. Rekursive Funktionen sind meist dann am praktischsten, wenn rekursiv definierte Datenstrukturen (wie hier die Listen) verarbeitet werden sollen.

Die Funktion *show_list* hangelt sich wie an einer Strickleiter an den Knoten der Liste entlang und gibt die darin gespeicherten Daten aus. Ihr Argument ist ein Zeiger auf einen Listenknoten; ich habe ihn *startnode* genannt, weil mir nichts besseres eingefallen ist.

Eine rekursive Funktion besteht aus mindestens zwei Zutaten: der Rekursionsbasis und dem Rekursionsschritt. Die Rekursionsbasis gibt den Verfahrensschritt an, der durchgeführt werden soll, wenn das Endekriterium erreicht ist. Im Falle der Funktion *show_list* ist das Ende-Kriterium das Erreichen des Listenendes. Dies erkennt die Funktion daran, daß die *next*-Komponente des gerade betrachteten Listenknotens den 0-Pointer enthält. Da der letzte Listenknoten keine Daten mehr enthält, kann die Funktion einfach mit *return* verlassen werden.

Ist der Knoten nicht letzter in der Liste, dann werden seine Daten angezeigt; anschließend kann der nächste Knoten ausgegeben werden. Den nächsten Knoten erreicht man, indem man dem *next*-Zeiger des aktuellen Knotens nachgeht. Was aber wäre besser zur Ausgabe des nächsten Knotens geeignet als die Funktion *show_list* selbst? Dies erkennend, ruft sich die Funktion *show_list* eben selbst - und schon haben wir den Rekursionsschritt!

Sie sehen: auch dies ist keine Hexerei.

Noch eine wichtige Anmerkung: die Reihenfolge, in der in einer rekursiven Funktion der Rekursionsschritt und die Rekursionsbasis hingeschrieben werden, macht meist den Unterschied zwischen einer eleganten, effizienten Problemlösung und einem Programmabsturz aus. Versuchen Sie, sich dies mit *show_list* deutlich zu machen.

# 7 Verkehr mit der Außenwelt - Dateien

Alle bisherigen Beispiel-Programme empfingen ihre Informationen während des Programmlaufs von der Tastatur und gaben ihre Daten auf dem Bildschirm aus. Diese interaktive Programmierung ist für viele Zwecke ausreichend; doch oft möchte man Daten dauerhaft speichern und sie auf andere Art als durch Tastatureingabe in das Programm bringen. Dazu sind die Dateien da.

Es gibt jedoch noch eine andere Art, wie Daten in das Programm gelangen können: man kann dem Programm beim Aufruf bestimmte Informationen in der Kommando-Zeile mitgeben. Dieser Möglichkeit möchte ich mich zuerst zuwenden.

## 7.1 Parameter von der Kommandozeile

Um ein C-Programm zu benutzen, ruft man es auf. Das bedeutet (unter TOS), daß man den Namen des Programms tippt und anschließend die *Return*-Taste betätigt. Bei der Arbeit unter GEM wird das Bildsymbol des Programms (oder sein Name) mit der Maus angewählt und zweimal angeklickt.

Benötigt das Programm Daten, dann besorgt es sich diese; einzige Möglichkeit dazu war - wenn man einmal von der Initialisierung von Variablen absieht - bisher die Eingabe von der Tastatur. Es ist jedoch auch möglich, dem Programm bereits bei seinem Aufruf Informationen mitzugeben. Dies soll an einem kleinen Problem verdeutlicht werden.

Wie Sie in früheren Beispielen gesehen haben, ist es sehr einfach, in C Programme zu schreiben, die alle Eingabezeichen in Kleinschreibung bzw. in Großschreibung umwandeln. Beide Programme haben die gleiche Struktur: in einer *while*-Schleife werden bis zum Erreichen einer Endebedingung Zeichen eingelesen und wieder ausgegeben, nachdem sie eine der beiden Filterfunktionen (bzw. Makros) *toupper* oder *tolower* durchlaufen haben.

Wegen dieser strukturellen Ähnlichkeit bietet es sich an, beide Funktionen in einem Programm zusammenzufassen; der Benutzer muß dann lediglich angeben, welche Form der Umwandlung er wünscht. Dies kann das Programm von ihm im Dialog erfragen. Aber in der Regel ist dem Benutzer schon beim Starten des Programmes klar, welche der beiden Verwendungs-

weisen er benötigt. Es wäre deshalb gut, wenn man dem Programm diese Information auch bereits beim Start mitgeben könnte.

Man kann. Die Hauptfunktion *main* hat nämlich zwei Argumente. Diese dienen genau dem Zweck, Parameter für das Programm in der Kommandozeile zu übergeben. Angenommen, das eben angesprochene Umwandlungs-Programm trägt den Namen *change*; dazu muß bei den meisten Compilern lediglich die Programmquelle in einer Datei mit dem Namen *change.c* stehen. Assembler und Linker übernehmen dann diesen Namen und ändern nur die Extension (den Namensteil hinter dem Punkt) für ihre Zwecke. Um das Programm aufzurufen, tippen Sie seinen Namen und drücken die *Return*-Taste:

```
change <RETURN>
```

Nun ist es aber möglich, C-Programme mit Aufrufargumenten zu versehen. Es soll z.B. das Programm *change* das Argument 'K' (bzw 'k') oder 'G' (bzw. 'g') erhalten, was bedeuten möge, daß eine Umwandlung in Kleinschreibung oder Großschreibung erfolgen soll. Ein korrekter *change*-Aufruf sieht dann so aus:

```
change k <RETURN>          oder:
change K <RETURN>          oder:
change g <RETURN>          oder:
change G <RETURN>
```

Der Buchstabe hinter dem Programmnamen ist ein Argument, das in *main* als Parameter das Verhalten des Programms steuern kann. Jetzt muß man aber wissen, wie man dieses Argument in das Programm bekommt, so daß es dort ausgewertet werden kann.

Dazu müssen in der Definition von *main* zwei Parameter für die Hauptfunktion deklariert werden. Die Namen, die Sie für diese Parameter wählen, sind beliebig (wie das bei Parametern so der Brauch ist); es haben sich jedoch die Bezeichnungen aus der Abbildung 7.1 eingebürgert.

```
main(argc, argv)
     int argc;
     char *argv[];

{    .
     .
     .
```

**Abb. 7.1:** *Deklaration der Parameter für* main

Wie Sie der Abbildung entnehmen können, kann *main* mit zwei Parametern deklariert werden. Dies bedeutet jedoch nicht, daß Sie in Ihren Pro-

grammen auf maximal zwei Argumente beschränkt sind. Vielmehr steckt hinter den beiden *main*-Parametern ein sinnreicher Mechanismus, der die Übergabe (theoretisch) beliebig vieler Parameter erlaubt.

Den Deklarationen entnehmen Sie, daß der erste *main*-Parameter (*argc*) eine Integer, der zweite (*argv*) hingegen ein String-Array ist. Beim Start des Programmes enthält *argc* die Anzahl der Argumente, die Sie an Ihr Programm übergeben haben plus 1; *argc* ist denn auch die Abkürzung für "argument count" (deutsch "Argumentzähler"). Jedes Programm-Argument, das Sie in der Kommandozeile übergeben haben, ist ein Eintrag im zweiten Parameter, dem String-Array *argv* (Abkürzung für "argument vector"). Das erste Argument ist der zweite String in *argv*, das zweite Argument ist der dritte String usw.

Wenn Sie also ein Programm *prg* wie folgt aufrufen:

```
prg Eins zwei 3 VIER
```

dann werden damit folgende Werte in die Parameter *argc* und *argv* übertragen:

```
argc hat als Wert 5

argv[1] hat als Wert "Eins"
argv[2] hat als Wert "zwei"
argv[3] hat als Wert "3"
argv[4] hat als Wert "VIER"
```

Da erhebt sich die Frage: warum hat *argc* den Wert 5 und was steckt im ersten String des *argv*-Arrays? Der Wert für *argc* ist leicht erklärt: der Programmname (also die Zeichenfolge, mit der das Programm aufgerufen wird) zählt mit. Auch wenn Sie ein Programm ohne jegliche Argumente aufrufen, hat *argc* also den Wert 1.

Die Frage nach dem ersten String in *argv* ist nicht so leicht zu beantworten. Der C-Standard fordert, daß sich dort der Programmname findet. Doch dies ist unter TOS anscheinend technisch nicht machbar. Der ATARI-C-Compiler plaziert als ersten String in *argv* die Meldung "C runtime" (deutsch: "C-Laufzeitumgebung"). Sie sollten sich daher nicht darauf verlassen, daß Ihr Compiler in diesem Punkt den Standard erfüllt.

Jetzt ist das Zusammenspiel der beiden Parameter klar: *argc* gibt an, wieviele Aufruf-Argumente im String-Array *argv* enthalten sind. Der Array *argv* enthält die einzelnen Aufruf-Argumente als Strings. Argumente werden auf der Kommandozeile durch Leerzeichen getrennt; daran erkennt den C Anfang und Ende der einzelnen Parameter.

Mit dem folgenden kleinen Programm können Sie sich die Argumente der Kommandozeile anzeigen lassen:

```
main(argc, argv)
     int argc;
     char **argv;

{    while (argc--)
         puts(*argv++);
}
```

In der Deklaration von *argv* habe ich mir den Zusammenhang zwischen Arrays und Pointern zunutze gemacht. Da das Programm die einzelnen Strings nicht über Array-Index, sondern über Pointer erreicht, ist dieser Form der Deklaration der Vorzug gegeben worden. Sie sollten sich angewöhnen, Array-Parameter mit den Array-Klammern zu deklarieren, wenn Sie sie im Programm so einsetzen (also über Index auf Komponenten zugreifen) und als Pointer, wenn Sie die (in C in der Regel schnellere!) Methode mit der Dereferenzierung bevorzugen.

Die einzelnen Argumente sind als Strings in *argv* abgelegt; sie werden deshalb mit *puts* ausgegeben. Über Dereferenzierung mit * erreicht man die Einzelstrings; durch Postinkrement wird der *argv*-Pointer nach der Ausgabe weitergeschoben.

Um einen Punkt zu klären, über den bei manchen C-Programmieren Verwirrung herrscht: Sie müssen die beiden Parameter nicht *argc* und *argv* nennen. Dies sind lediglich durch die Tradition institutionalisierte Namen. Doch jeder andere zulässige Parameter-Name tut es ebenso, also auch *otto* und *emil*. Wichtig ist alleine, daß die Parameter richtig deklariert sind (der erste als Integer und der zweite als String-Array) und daß Sie sich dann im Programm an Ihre Namen auch halten. Wollen Sie ein Programm mit Parametern von GEM heraus starten, dann müssen Sie dafür sorgen, daß die Argumente auch übergeben werden. Dazu ist es nötig, mit der Option "Anwendung anmelden" aus dem "Optionen"-Menü das Programm als "TOS übernimmt Parameter" dem System bekanntzumachen.

Jetzt aber zurück zum Ausgangsproblem: Zeichenumwandlung in Abhängigkeit von einem Kommando-Zeilen-Argument. Programme, die Argumente von der Kommando-Zeile bekommen, sind vor allem unter UNIX sehr häufig anzutreffen. Da der UNIX-Stil vor allem erfahrene C-Programmierer bereits 'verdorben' hat, ist anzunehmen, daß sich diese Tradition auf dem ATARI fortsetzten wird. Zu einem guten Programm gehört eine Überprüfung der Parameter aus der Kommando-Zeile. Im Programm 7.1 ist dies ansatzweise demonstriert.

```
#include <stdio.h>                          /*************************/
                                            /* Argumente in der Komman-*/
main(argc, argv)                            /*  dozeile und Pointer    */
     int argc;                              /*  auf Funktionen.        */
     char **argv;                           /*************************/
                                            /*                        */
{    char (*chfun) ();                      /* Zeiger auf die Umwand- */
     char up(), low();                      /*  lungsfunktion.        */
     char c;                                /*                        */
                                            /*                        */
     if (argc < 2)                          /* Nicht genuegend Argu-  */
        usage(0);                           /*  mente angegeben!      */
                                            /*                        */
     switch (up(**++argv))                  /* Hier darf kein Makro   */
     {                                      /*  stehen!               */
        case 'G' : chfun = up;              /* Umwandeln in Gross-    */
                   break;                   /*  schreibung.           */
                                            /*                        */
        case 'K' : chfun = low;             /* Umwandeln in Klein-    */
                   break;                   /*  schreibung.           */
                                            /*                        */
        default  : usage(1);                /* Fehler: falsche Option.*/
     }                                      /*                        */
                                            /*                        */
     while ((c = getchar()) != '#')         /* Haupt-Schleife: Einle- */
        putchar((*chfun) (c));              /*  sen, filtern und aus- */
}                                           /*  geben.                */
                                            /*************************/

char *Error[] =
{    "Zu wenig Parameter!",
        "Unbekannter Parameter; nur 'K' und 'G' erlaubt!"
} ;
                                            /*************************/
usage (error)                               /* Fehler aus Tabelle aus-*/
       int error;                           /*  geben.                */
                                            /*************************/
{    puts("\nFehler:");                     /*                        */
     puts(Error[error]);                    /*                        */
     exit();                                /* Programm beenden.      */
                                            /*************************/


                                            /*************************/
char up(c)                                  /* Das sind die bereits   */
     char c;                                /*  bekannten Umwandlungs-*/
                                            /*  Makros im Funktions-  */
{    return (toupper(c));                   /*  gewand. Die Verklei-  */
}                                           /*  dung ist noetig, um   */
                                            /*  einen Pointer auf die */
char low(c)                                 /*  Funktionen zu erhalten.*/
     char c;                                /*  (Makros sind keine    */
                                            /*  Funktionen; auf sie   */
{    return (tolower(c));                   /*  kann man nichts zeigen*/
}                                           /*  lassen!)              */
                                            /*************************/
```

**Prog. 7.1:** *Übernahme von Argumenten aus der Kommandozeile*

Da das Programm einen Parameter benötigt, wird zuerst überprüft, ob der Benutzer überhaupt ein Argument mit übergeben hat. Im *switch* wird dann fernerhin dafür gesorgt, daß nur korrekte Parameter akzeptiert werden. Im Fehlerfall bemüht das Programm eine Fehlerfunktion *usage*, die dem Benutzer den korrekten Gebrauch (englisch *usage*) des Programmes mitteilt und anschließend die Programmausführung abbricht. Dazu wird die Funktion *exit* aus der Standard-Bibliothek bemüht.

Zur Auswertung des Arguments im *switch* braucht nur ein einzelnes Zeichen betrachtet werden: das erste (und einzige) Zeichen des ersten Argumentstrings. Den ersten Argument-String erhält man durch Inkrementieren des *argv*-Pointers (denn zu Beginn zeigt *argv* ja auf den Programmnamen (oder was immer das Betriebssystem an seiner Stelle liefert!): daher *++argv*. Den ersten String erreicht man durch Dereferenzieren des inkrementierten Pointers; so kommt's zu **++argv*. Jetzt brauchen wir nur noch das erste Zeichen im String; da ein String sich ebenso wie ein Zeichen-Pointer verhält, kommt man da durch nochmaliges Dereferenzieren 'ran. Der krönende Abschluß also: ***++argv*.

Dies ist ein weiteres Beispiel für den Zusammenhang zwischen der Deklaration einer Variablen und ihrer Verwendung. Die Deklaration

```
char **argv;
```

sagt ja aus, daß man durch zweimaliges Dereferenzieren der Variablen *argv* ein Zeichen erhält. Fernerhin gibt mir das Programm 7.1 die willkommene Gelegenheit, die im letzten Kapitel mehrfach angeklungenen ominösen "Zeiger auf Funktionen" zu erklären.

Ähnlich, wie der Name eines Arrays ein Zeiger auf das erste Element im Array ist, ist der Name einer Funktion ein Zeiger auf diese Funktion!. Ähnlich, wie ein Array-Name in C eine Konstante ist, ist auch ein Funktionsname eine Konstante: beiden kann nichts zugewiesen werden.

Aber sie können auf der rechten Seite von Zuweisungen auftreten. Beiden - Arrays und Funktionen - können passenden Pointern zugewiesen werden.

Doch warum sollte man das bei den Funktionen tun wollen? Natürlich aus Gründen der Eleganz; und um seine Programme vor den neugierigen Blikken von C-Dilettanten zu schützen. Aber Spaß beiseite: einige Probleme lassen sich wirklich mit Funktions-Pointern eleganter formulieren. Programm 7.1 ist dafür, wie ich hoffe, ein Beispiel.

Die Haupt-Schleife des Programms 7.1 kann so beschrieben werden (siehe auch den Kommentar zum Programm): lies ein Zeichen ein und gib es gefiltert wieder aus. Das Filtern übernimmt eine der beiden Funktionen *to-*

*upper* oder *tolower*, je nachdem, welchen Parameter der Benutzer gewählt hat. Natürlich könnte man in der Schleife mit einem *if* anhand des Parameters die jeweils benötigte Filterfunktion auswählen, etwa so:

```
char option;
.
.
option = toupper(**argv);

while ((c = getchar()) != '#')
        putchar ( option == 'G' ? toupper(c) : tolower(c));
.
.
.
```

Aber mit diesem Verfahren wird eine Entscheidung, die bereits zu Beginn des Programmlaufs feststeht, überflüssigerweise bei jedem Schleifendurchgang erneut getroffen. Wenn das Programm einmal bei der *while*-Schleife angelangt ist, dann steht fest, welche der beiden Filter-Funktionen der Benutzer wünscht.

Als Filter-Funktion kommt einer von zwei 'Werten' in Frage. Variablen sind Dinge, die Werte haben können. Funktions-Pointer sind Variablen, die Funktionen (genauer: Zeiger darauf) als Werte haben können. Deshalb liegt es nahe, die gewünschte Filterfunktion über einen Funktions-Pointer anzusprechen, den man beim Programmanfang auf die gewünschte Funktion zeigen läßt. Dies erledigt das Programm 7.1 in dem *switch*, der die Parameter aus der Kommandozeile auswertet. Dem Pointer *chfun* wird entweder der Filter *up* oder *low* zugewiesen. Soll die Funktion dann tatsächlich angewendet werden, dann muß erst die Variable *chfun* dereferenziert werden, indem man ihr einen Stern verpaßt und dann dahinter in die Funktionsklammern das Argument schreibt. Da die Funktionsklammern stärker binden als der *-Operator, ist ein zusätzliches Klammerpaar nötig, wodurch sich der Ausdruck

```
(*chfun) (c)
```

für die Funktions-Anwendung ergibt.

Die beiden Filterfunktionen *up* und *low* sind nichts anders, als die bereits bekannten Umwandlungs-Makros *toupper* und *tolower* im Funktionsgewand. Dies ist nötig, weil Makros keine Funktionen sind. Einige Makros werden vielleicht vom Präprozessor in Funktionsaufrufe expandiert; andere aber - und dazu zählen *toupper* und *tolower* - expandieren zu Operator-Anwendungen (erinnern Sie sich: beide sind mit dem dreistelligen *? ... : ...*-Operator definiert!). Auf einen Operator kann man jedoch keinen Funktions-Pointer zeigen lassen.

Wer jetzt einwendet, daß man dadurch zwar einen überflüssigen Test in der Schleife eingespart, dafür aber zwei neue unnötige Funktionen eingeführt hat, der hat keinen Sinn für Schönheit!

Die Filter-Funktion *up* wird übrigens bereits im *switch* eingesetzt, der die Parameter-Auswertung steuert (und dazu das Argument einheitlich in Großschreibung umwandelt). Es wäre falsch gewesen, hier den Makro *to-upper* einzusetzen. Die Gründe dafür sind subtil; doch dies ist ohnedies ein Kapitel, das subtiler Materie gewidmet ist. Sehen wir uns also einmal an, was der Makro-Präprozessor aus

```
toupper (**++argv)
```

macht. Dazu wiederhole ich noch einmal die nötigen Makro-Definitionen:

```
#define toupper(c) (islower(c) ? c + 'A' - 'a': c)
#define islower(c) (c >= 'a' && c <= 'z')
```

Als erstes wird *toupper* expandiert (zur Erinnerung: die Ersetzung eines Makros durch den Text, der ihn definiert, nennt man Makro-Expansion). Dazu muß lediglich der Makro-Parameter *c* durch das tatsächliche Makro-Argument ***++argv* ersetzt werden; man erhält:

```
(islower(**++argv) ? **++argv + 'A' - 'a': **++argv)
```

Nun ist nach Expansion von *toupper* immer noch der Makro *islower* übrig, der ebenfalls expandiert werden muß. Dies führt zu:

```
((*++argv >= 'a' && *++argv <= 'z') ? **++argv + 'A' - 'a': **++argv)
```

Dies ist der Ausdruck, den der Compiler nach vollständiger Expansion durch den Präprozessor erhält. In diesem Ausdruck wird der Pointer *argv* insgesamt dreimal (Zwischenfrage: warum nicht viermal?) inkrementiert! Wir wissen jedoch, daß *argv* im Falle des vorliegenden Programms nur zwei Strings enthält: den Programmnamen und das Programm-Argument. Nach dem zweiten Inkrementieren (es wird ja auch mit Präinkrement gearbeitet) zeigt *argv* also ins Pointer-Nirwana. Diesem Verweis geht das Programm aber freudig nach, um das Ergebnis (den gewünschten Großbuchstaben) zu produzieren; das Unheil nimmt seinen Lauf!

Moral: Makroexpansion ist etwas anderes als ein Funktionsaufruf; es ist bloße Textersetzung. Arbeitet der Ersetzungstext mit Seiteneffekten (hier: dem Inkrementieren eines Pointers), dann können sich die hübschesten Fehler einstellen. Fehler, die schwer zu finden sind, da man den Makro meist schon als erledigt und fehlerfrei abgebucht hat.

## 7.2 Umlenkung der Ein-/Ausgabe.

Es gibt noch eine andere Art von Argumenten, die Sie Ihren Programmen in der Kommandozeile mitgeben können. Zu deren Erklärung greift man am besten zu der sattsam bekannten Zeichen-Kopier-Schleife, die die Abbildung 7.2 noch einmal wiederholt. Beachten Sie jedoch, daß als Endekriterium für die Schleife diesmal das Dollarzeichen '$' gewählt wurde. In der Abbildung wird angenommen, daß das Programm den Namen *cpy* erhalten hat und bereits fertig übersetzt ist. Unter dem Quelltext des Programms sehen Sie in der Abbildung einen Aufruf des Programms.

```
#include <stdio.h>

main ()

{   char c;

    while ((c = getchar()) != '$')
        putchar(c);
}


Aufruf von cpy mit Umlenkung:


cpy >outdat.txt <RETURN>
```

**Abb. 7.2:** *Umlenkung der Ein-/Ausgabe*

Wie Sie sehen, wurde dem Programm in der Kommandozeile ein Argument mitgegeben. Dieses ist in zweifacher Hinsicht merkwürdig. Einmal ist das Argument (bei dem es sich um einen Dateinamen zu handeln scheint; dafür spricht jedenfalls die Extension ".txt") mit einer spitzen Klammer dekoriert. Dann aber sind im Programm gar keine *argc*- und *argv*-Parameter deklariert. Das bedeutet, daß das Programm die Argumente aus der Kommandozeile gar nicht auswertet!

Kann es auch nicht. Denn den Parameter *>outdat.txt* würde das Programm nie zu sehen bekommen; den schnappt sich bereits das Betriebssystem. Denn ein mit einer öffnenden spitzen Klammer eingeleiteter Parameter ist eine Anweisung an das Betriebssystem, die Ausgabe bestimmter C-Funktionen auf einen anderen Kanal als den Bildschirm umzulenken.

Das Betriebssystem ist diejenige Instanz des Computers, die sich um den Verkehr mit der Außenwelt kümmert; zu seinen vielen Aufgaben gehört unter anderem: Lesen von der Tastatur, Ausgeben auf dem Bildschirm oder Drucker, Ein-/Ausgabe auf Dateien und Verwaltung von Disketten. Zur Kommunikation mit der Außenwelt bedient sich das Betriebssystem soge-

nannter "Kanäle". Einige Kanäle führen nur in eine Richtung: von der Außenwelt in den Computer (wie z.B. die Tastatur; von der kann man nur empfangen, aber nichts an sie senden) oder vom Computer zur Außenwelt (wie z.B. der Drucker; der kann nur empfangen aber nichts an den Computer senden). Neben diesen Kommunikations-Einbahnstraßen (den "unidirektionalen Kanälen"; das für die Freunde der Fremdworte) gibt es auch Kanäle, die in beiden Richtungen (vom und zum Computer) nutzbar sind. Zu diesen "bidirektionalen Kanälen" zählen insbesondere Diskettendateien; aus ihnen können Informationen gelesen, in sie können Informationen geschrieben werden.

Neben dem Begriff des "Kanals" ist für ein Verständnis der Vorgänge, die mit der Kommandozeile in Abbildung 7.2 ausgelöst werden, noch der Begriff des "Standard-Kanals" von Bedeutung. Das Betriebssystem geht davon aus, daß in der Standard-Anwendungssituation eines Programms der Mensch mit seinem Computer interaktiv in Verbindung treten möchte. Das bedeutet für ihn: die Eingaben an das System kommen normalerweise von der Tastatur; die Ausgaben des Systems erfolgen normalerweise auf dem Bildschirm. Anders ausgedrückt: die Tastatur ist Standard-Eingabekanal, der Bildschirm ist Standard-Ausgabekanal für das Betriebssystem.

Aber es läßt sich gerne eines Besseren belehren. Denn der Anwender kann den Wunsch haben, andere Kanäle zu Standard-Ein-/Ausgabekanälen zu machen. Er kann es vorziehen, die Ausgabe von Programmen auf den Drucker, in eine Datei, auf die serielle Schnittstelle etc. zu legen. Ebenso kann er das Bedürfnis haben, die Eingabe von Programmen aus einer Datei oder von der seriellen Schnittstelle zu empfangen.

Um jetzt nicht für jede Kombination von Ein- und Ausgabekanälen eigene, spezialisierte Programme schreiben zu müssen, bedient sich das Betriebssystem des ATARI einer Technik, die erstmals mit dem Betriebssystem UNIX weite Verbreitung fand: die Standard-Kanäle können umgelenkt werden.

In bisherigen Beispielsprogrammen wurde davon ausgegangen, daß *getchar* von der Tastatur liest und *putchar* auf den Bildschirm schreibt. Doch dies entspricht nicht ganz den Tatsachen. Richtig muß es heißen: *getchar* empfängt ein Zeichen vom Standard-Eingabekanal; *putchar* schreibt ein Zeichen auf den Standard-Ausgabekanal. Was gerade als Standard-Kanal vereinbart ist, darum kümmert sich das Betriebssystem.

Normalerweise ist die Tastatur Standard-Eingabe und der Bildschirm Standard-Ausgabe. Deshalb funktionieren im Normalfall die Funktionen *getchar* und *putchar* (und alle, die sich darauf stützen) wie oben angenommen. Abweichungen vom Normalfall teilt der Benutzer dem Betriebssystem in der

Kommandozeile seines Programmes mit. Womit wir wieder bei der Abbildung wären.

Sieht das Betriebssystem in der Kommandozeile ein Argument, das mit > beginnt, dann faßt es dies als Anweisung auf, die Standard-Ausgabe umzulenken. Umgelenkt wird auf den Kanal, der hinter dem > angegeben ist; in diesem Fall ist es eine Disketten-Datei mit dem Namen "outdat.txt". Wenn Sie das Programm in der Abbildung 7.2 wie angegeben starten, dann sehen Sie seine Ausgabe also nicht auf dem Bildschirm. Stattdessen werden alle ausgegebenen Zeichen vom Betriebssystem in eine Datei Namens "outdat.txt" umgelenkt. Nach Beendigung des Programms können Sie sich die Datei dann (mit dem Betriebssystems-Kommando *type* oder unter GEM durch Anklicken mit der Maus) ansehen.

**Anmerkung:** Die Betriebssystems-Version des ATARI, die zum Zeitpunkt der Erstellung dieses Buchs im Umlauf war, konnte die Umlenkung der Ausgabe nicht richtig verarbeiten.

Ebenso wie die Standard-Eingabe kann auch die Standard-Ausgabe umgelenkt werden. Wenn Sie dies wünschen, so teilen Sie es dem Betriebssystem durch einen schließenden spitzen Pfeil gefolgt von dem Eingabe-Kanal ihrer Wahl mit. Somit ist es möglich, mit dem selbstgeschriebenem Programm *cpy* den Inhalt einer Datei (ähnlich wie mit dem eingebauten *type*-Kommando) auf dem Bildschirm anzusehen. Dazu schreiben Sie:

```
cpy <cpy.c
```

Mit dieser Programm-Zeile können Sie sich somit den Quellcode des Programms ansehen, das gerade läuft.

Doch halt! Sie sehen nicht alles! Denn Endekriterium für die *while*-Schleife ist das Dollarzeichen. An der Stelle, an der in der Datei "cpy.c" dieses Zeichen steht, bricht die Ausgabe ab - der Rest geht verloren. Jetzt verstehen Sie auch, warum ich als Endekriterium das Dollarzeichen und nicht, wie sonst üblich, den 'Gartenzaun' # einsetzte: dieser Taucht in der Quelle *cpy.c* gleich zu Beginn in der *include*-Anweisung auf. Wenn Sie dann das Beispiel wie beschrieben nachvollziehen, bekommen Sie nichts zu sehen!

Es wäre nun schon wünschenswert, die Datei bis zu ihrem Ende ansehen zu können. Dazu aber muß das Programm *cpy* das Dateiende erkennen, um die Schleife korrekt zu beenden. Woran jedoch soll man das Dateiende erkennen können?

Dazu befolgt die Funktion *getchar* eine bestimmte Konvention: wenn Sie das Dateiende erreicht hat, dann gibt sie einen eindeutigen Wert zurück, der nicht mit einem Zeichen verwechselt werden kann. Meist verwenden

die C-Compiler den Wert -1 als Signal für das Dateiende. Normalerweise brauchen Sie sich jedoch nicht darum zu kümmern, welcher Wert nun dafür herhält. In einer *include*-Datei mit dem Namen "stdio.h" ist bei allen ernstzunehmenden Compilern eine Konstante Namens EOF (Abkürzung für "End of File", also "Dateiende") als Makro vereinbart. Um also auch Dateien als Standard-Eingabe verarbeiten zu können, muß das *cpy*-Programm so umgeschrieben werden:

```
#include <stdio.h>

main ()

{   char c;

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

**Anmerkung:** Das *getchar* des ATARI-Compilers konnte in der mir verfügbaren Version das Dateiende nicht erkennen. Wenn Sie ebenfalls mit einer frühen Compiler-Version arbeiten, dann sollten Sie darauf verzichten, die Umlenkungs-Beispiele auszuprobieren.

Schließlich können beide Möglichkeiten kombiniert werden: Ein- und Ausgabe eines Programmes können beide umgelenkt werden. Mit der Kommandozeile

```
cpy <cpy.c >cpy.bat
```

werden Zeichen aus der Datei "cpy.c" gelesen und in die Datei "cpy.bat" geschrieben; es wird also eine Kopie der Datei "cpy.c" mit Namen "cpy.bat" erstellt. Das Programm *cpy* eignet sich - so unscheinbar es ist - auch zum Kopieren von Dateien; daher auch der Name (kopieren heißt "copy" auf englisch).

## 7.3 Gepufferte Ein-/Ausgabe

Die Verarbeitung von Dateien ist, wie im letzten Kapitel beschrieben, durch Umlenkung der Ein-/Ausgabe möglich. Doch geschieht dies nur zeichenweise: *getchar* und *putchar* lesen bzw. schreiben immer nur ein Zeichen vom bzw. auf den jeweiligen Kanal. Nicht für alle Anwendungen ist diese zeichenweise Verarbeitung angemessen. Auch reicht es nicht für alle Anwendungen aus, wenn lediglich zwei Dateien - eine für die Eingabe, eine für die Ausgabe - bearbeitet werden können. Schließlich kann es auch noch notwendig sein, die Namen der Dateien nicht von der Kommandozeile zu empfangen, sondern erst während des Programmlaufs zu erhalten, sei es

durch Benutzereingaben, sei es durch Berechnungen, die das Programm selbst anstellt.

Für diese Anwendungsfälle stellt C andere Möglichkeiten der Ein-/Ausgabe bereit. Dazu muß man wissen, wie in C eine Kommunikation über andere Kanäle als den Standard-Ein- und Ausgabekanal erfolgen kann.

Neben den beiden vom Betriebssystem verwalteten Kanälen kann sich ein C-Programm auch eigene Kanäle einrichten und diese mit Geräten (Bildschirm, Tastatur, Drucker, vor allem aber: Dateien) verbinden. Zu diesem Zweck muß das Programm den Kanal bzw. die Kanäle eröffnen, über die es mit der Außenwelt in Verbindung treten will.

Das Öffnen eines Kanals bedeutet, eine Verbindung zwischen dem Programm und einem externen Gerät oder einer Datei herzustellen. Außerdem wird mit dem Öffnen des Kanals dem Programm die Kanal-Identifikation mitgeteilt. Nur zum Öffnen des Kanals benötigt das Programm den Dateinamen. Von da ab werden alle Transaktionen, die den Kanal betreffen, über die Identifikation abgewickelt (wenn Ihnen das jetzt zu abstrakt erscheint, dann lesen Sie einfach weiter; später wird Ihnen dann klarwerden, warum ich so allgemein von "Kanal" und "Kanalidentifikation" gesprochen habe).

Bei den folgenden Ausführungen sollten Sie beachten, daß alles, was mit Ein- und Ausgabe zusammenhängt, strenggenommen nicht mehr zum Sprachumfang von C gehört. In der Sprache C gibt es keinerlei Anweisungen oder Operatoren, die mit Ein-/Ausgabe zu tun haben. Alle für die Ein-/Ausgabe nötigen Schritte werden vielmehr über Funktionen abgewickelt, die in C geschrieben sind. Diese Funktionen sind in der Standard-Bibliothek enthalten und weitestgehend standardisiert. Der Compiler kümmert sich also nicht im Geringsten um die Ein-/Ausgabe. Die Funktionen, die damit zusammenhängen, müssen stattdessen vom Linker aus der Bibliothek geholt werden.

Dies mag unnötig umständlich erscheinen. Tatsächlich gewinnt der Programmierer dadurch jedoch an Flexibilität: denn wenn er mit den vorgefertigten Ein-/Ausgabe-Routinen aus der Bibliothek nicht zufrieden ist, dann kann er sich eben eigene - vielleicht bessere, vielleicht spezialisiertere - Funktionen schreiben.

Die Bibliothek, in der die E-/A-Funktionen liegen, wird zwar "Standard"-Bibliothek genannt; doch als erfahrener Programmierer sollten Sie allen Behauptungen, etwas sei standardisiert, mit vorsichtigem Mißtrauen begegnen. Die folgenden Ausführungen über die Standard-Bibliothek werden mit großer Wahrscheinlichkeit auch für Ihr C-System zutreffen. Aber es ist

möglich, daß in untergeordneten Details kleine Abweichungen bestehen; wenn also etwas nicht so funktioniert, wie es sollte, ziehen Sie am besten Ihr Handbuch zu Rate.

Kanäle können in C entweder gepuffert oder ungepuffert sein. Die Bedeutung dieser Ausdrücke wird später noch klar werden. Fürs erste wenden wir uns den gepufferten Kanälen zu.

Ein gepufferter Kanal ist in C eine Struktur; diese ist in der Include-Datei "stdio.h" definiert und trägt den Namen FILE (muß großgeschrieben werden!). Der Name deutet schon darauf hin, daß die gepufferte Ein-/Ausgabe hauptsächlich für die Kommunikation mit Dateien gedacht ist ("Datei" heißt englisch "file"). Die Identifikation eines gepufferten Kanals ist ein Zeiger auf eine FILE-Struktur, unter C-Programmieren kurz und salopp "FILE-Pointer" genannt. Auch für den Ausdruck "gepufferte Ein-/Ausgabe" ist noch ein anderer Begriff im Umlauf: man nennt einen gepufferten Eingabekanal kurz einen "Eingabe-Strom" ("input stream") und einen gepufferten Ausgabekanal entsprechend eine "Ausgabe-Strom" ("output stream").

Das Öffnen einer Datei (sprich: die Verknüpfung eines Dateinamens mit einem FILE-Pointer) übernimmt die Funktion *fopen*. Die folgenden Anweisungen zeigen, wie eine Datei EINGABE.DAT geöffnet wird:

```
.
.
.
FILE *ein, *fopen();

ein = fopen("EINGABE.DAT","r");
.
.
.
```

Die Funktion *fopen* hat zwei Argumente und beide sind Strings. Das erste Argument ist der Name der zu öffnenden Datei. Das zweite Argument ist der Modus, in dem die Datei geöffnet werden soll. Das ATARI-C kennt hier drei Möglichkeiten: "r", "w" und "a". Die Bedeutung dieser Optionen:

"r": die Datei wird zum Lesen geöffnet
"w": die Datei wird zum Schreiben geöffnet
"a": die Datei wird zum Anfügen geöffnet.

Öffnen Sie eine Datei mit dem Modus "r", dann ist der zugehörige Kanal unidirektional: Sie könne nur daraus Lesen, aber nichts darauf schreiben. Die Modi "w" und "a" eröffnen bidirektionale Kanäle. Falls eine Datei des angegebenen Namens noch nicht existiert, wird sie im Modus "w" kreiert. Existiert sie bereits, dann können mit "a" neue Daten an das Ende der Datei

(also hinter bereits in der Datei enthaltene andere Daten) geschrieben werden. Im Modus "w" wird mit dem Schreiben am Dateianfang begonnen; das bedeutet, daß eventuell bereits vorhandenen Daten durch Öffnen im Modus "r" verlorengehen. (Über die Modi und ihre verschiedenen Auswirkungen herrscht bei unterschiedlichen Compilern nicht immer Einigkeit; also im Handbuch nachschlagen).

### 7.3.1 Zeichenweise gepufferte Ein-/Ausgabe: getc und putc.

Nun möchten Sie natürlich auch wissen, wie man mit einem gepufferten Kanal kommuniziert, wie man also Zeichen von ihm empfängt oder an ihn schreibt. Dazu gibt es zwei Funktionen, die Ihnen vermutlich bekannt vorkommen werden: *getc* und *putc*. Beide lesen bzw. schreiben zeichenweise von einem Kanal. Dies ist jedoch nicht einer der vom Betriebssystem verwalteten Standard-Kanäle, sondern ein im Programm eröffneter Ein-/Ausgabe-Strom. Um zu wissen, welcher Strom gemeint ist, benötigen die Funktionen also den FILE-Pointer (denn dieser identifiziert den Kanal) als zusätzliches Argument.

Das kleine Programm 7.2 kopiert die Datei EIN.DAT in die Datei AUS.DAT. Als erstes werden im Programm zwei Kanäle geöffnet: einer zur Eingabe, einer zur Ausgabe. Das geschieht mit *fopen*, welches als Wert einen FILE-Pointer zurückgibt (die Kanal-Identifikation), den sich das Programm in passenden Pointer-Variablen aufhebt, weil es mit diesen Kanälen ja noch arbeiten will.

```
                                        /***************************/
                                        /* Enthaelt Definition der */
#include <stdio.h>                      /*  FILE-Struktur.         */
                                        /*                         */
                                        /***************************/
main()                                  /*  Kopiere Datei.         */
                                        /***************************/
{   FILE *quelle, *ziel, *fopen;        /* <quelle> und <ziel> sind*/
    int c;                              /*  die Kanaele. <c> muss  */
                                        /*  "int" sein, weil EOF   */
                                        /*  kein "char" ist.       */
    quelle = fopen("EIN.DAT", "r");     /* Kanaele eroeffnen und   */
    ziel = fopen("AUS.DAT", "w");       /*  merken.                */
    while ((c = getc(quelle)) != EOF)   /* Kopierschleife.         */
        putc(c,ziel);                   /*                         */
                                        /*                         */
    fclose(quelle);                     /* Schliessen der Kanaele  */
    fclose(ziel);                       /*  nicht vergessen!       */
                                        /*                         */
}                                       /***************************/
```

**Prog. 7.2:** *Datei kopieren mit gepufferter Ein-/Ausgabe*

Sind die Kanäle offen, kann das Kopieren beginnen. *getc* liest ein Zeichen von dem Kanal, der sein Argument ist und gibt es zurück. Bei Erreichen des Datei-Endes signalisiert *gets* diese durch den (in "stdio.h" definierten) Wert EOF; da dieser Wert kein Zeichen iat, darf die Variable *c* auch nicht als *char* deklariert werden. Meist speichert man die Werte von *getc* in einer *int*-Variablen; so auch hier.

Das gelesene Zeichen wird von *putc* auf den Ausgabekanal ausgegeben. Dazu muß die Funktion neben dem Ausgabezeichen (als erstes Argument) auch noch den FILE-Pointer des Ausgabe-Stroms als zweites Argument erhalten.

Nach Beendigung der Kopierschleife dürfen Sie das Programm nicht einfach verlassen. Gepufferte Kanäle müssen nämlich - mit der Funktion *fclose* - geschlossen werden. Warum? Weil sie gepuffert sind!

Denken Sie daran, daß man gepufferte Ein-/Ausgabe meist für das Arbeiten mit Disketten (oder Festplatten, so man hat) benutzt. Die Ein-/Ausgabe von Disketten mag dem Menschen zwar schnell erscheinen; aber für den Prozessor dauert sie eine Ewigkeit. Disketten sind mechanische Speichergeräte und die Mechanik kann mit der Elektronik nicht mithalten. Denn was passiert, wenn der Computer ein Zeichen von der Diskette lesen will? Vereinfacht gesagt folgendes: die Steuer-Elektronik des Laufwerks sagt dem Lese-/Schreibkopf (den Sie sich ähnlich wie den Tonarm eines Plattenspielers vorstellen können), an welcher Position auf der Diskette das gewünschte Zeichen zu finden ist; dann wird der Tonarm dorthin bewegt, anschließend wartet die Steuerelektronik, bis das Zeichen sich unter dem Tonarm befindet (die Diskette rotiert ja), liest es und gibt es an den Prozessor weiter.

In der Zwischenzeit vergeht der Prozessor schier vor Langeweile! Seine Arbeitsgeschwindigkeit ist ungleich größer als das Tempo, mit dem die Diskette arbeitet. Würde für jedes vom Prozessor gewünschte Zeichen ein eigener Diskettenzugriff in der oben geschilderten Manier erfolgen, dann wäre der Computer zu nichts mehr zu gebrauchen.

Darum werden die Informationen gepuffert. Die Funktion zur gepufferten Ein-/Ausgabe legen zwischen den Prozessor mit seinem Informationsbedürfnis und der langsamen Diskette eine Zwischenstation zur Informationsvermittlung ein. Irgendwo im Arbeitsspeicher reservieren sie sich einen größeren Speicherbereich für eigene Zwecke. Wenn nun der Rechner ein Zeichen von der Diskette haben will, dann lesen diese Funktionen einen ganzen Schwung Daten aus der Umgebung des gewünschten Zeichens von der Diskette und legen sie zur Zwischenspeicherung in den Puffer. Darunter befindet sich natürlich auch das gewünschte Zeichen, das an den Com-

puter und somit an Ihr Programm aus dem Puffer weitergereicht wird. Beim nächsten Zeichen, das Ihr Programm wünscht, ist die Wahrscheinlichkeit sehr groß, daß es beim letzten Lesevorgang bereits mit eingelesen wurde und sich daher schon im Puffer befindet. Ein Zugriff auf die Diskette entfällt; das Zeichen muß lediglich vom Puffer an das Programm weitergeleitet werden. Solange der Computer also die Daten in der Reihenfolge wünscht, wie Sie auf der Diskette gespeichert sind, bietet die gepufferte Eingabe einen beträchtlichen Zeitgewinn gegenüber dem direkten Disketten-Zugriffe. Der Fachmann sagt: durch die Pufferung wird die Anzahl der Disketten-Zugriffe minimiert.

Die gepufferte Ausgabe funktioniert ähnlich. Zeichen, die Ihr Programm ausgibt, werden nicht sofort auf die Diskette geschrieben, sondern erst in einem Puffer aufgesammelt. Erst wenn dieser Puffer voll ist, wird er als Ganzes auf die Diskette geschrieben und ist somit für den Empfang neuer Zeichen bereit. Dieses "Auf-die-Diskette-Schreiben" (auch "rausschreiben" genannt) eines Puffers bezeichnet man mit dem englischen Fachausdruck "to flush a buffer".

Es gibt also Lese- und Schreibpuffer. Lese-Puffer werden gefüllt, wenn vom Programm ein Zeichen gewünscht wird, das sie nicht enthalten. Schreib-Puffer werden rausgeschrieben, wenn sie voll sind.

Was aber, wenn das Kopierprogramm 7.2. fertig ist, ehe sein Schreibpuffer voll ist? Die Funktion *putc*, die die Pufferverwaltung übernimmt, hat noch keinen Anlaß, den Puffer rauszuschreiben. Würden jetzt die beiden *fclose*-Anweisungen fehlen, dann würde das Programm abbrechen, ohne daß der Ausgabepuffer in die Ausgabedatei geleert wird: die Datei ist unvollständig!

Dieses Leeren der Puffer (sowie das korrekte Aktualisieren des Disketten-Inhaltsverzeichnisses) übernimmt die *fclose*-Funktion. Einige Compiler führen beim Verlassen eines Programms automatisch für jeden geöffneten Kanal ein *fclose* durch; ebenso ist es üblich, daß auch *exit* offene Kanäle schließt. Aber Sie sollten sich darauf nicht unbedingt verlassen und besser selbst dafür sorgen, daß Ihre Kanäle geschlossen werden.

Sie sehen: mit einem Kanal ist eine Vielzahl an Informationen verknüpft. Die gepufferten Ein-/Ausgabe-Funktionen müssen unter anderem wissen, wo der Puffer für die Zwischenspeicherung zu finden ist, wie groß er ist, wieviele Zeichen bereits in ihn geschrieben wurden etc. All diese Informationen sind in der FILE-Struktur enthalten. Diese können Sie sich in der Datei "stdio.h" ansehen; nach den eben erfolgten Ausführungen sollten Sie ihren Aufbau in groben Zügen verstehen können.

Beim Öffnen von Kanälen können Fehler auftreten. Eine Datei zum Lesen zu eröffenen, die nicht existiert oder eine Datei auf einer Diskette zu kreieren, auf der kein Platz ist - das sind Fehlersituationen, die ein Öffnen des Kanals unmöglich machen. Die *fopen*-Funktion muß diese dem Programm mitteilen, damit es angemessen reagieren kann. Sie tut es, indem sie anstelle eines FILE-Pointers den Wert 0 zurückgibt. Im Programm 7.3 sehen Sie eine Funktion *f_open*, die das Öffnen von Dateien mit Fehlerüberprüfung übernimmt.

```
#include <stdio.h>

                                        /***************************/
FILE *f_open(file,how)                  /* Datei oeffnen mit Feh-  */
            char *file, *how;           /*  lercheck.              */
                                        /***************************/
{  FILE *fp, *fopen();                  /*                         */
                                        /*                         */
   if (fp = fopen(file,how))            /* Versuche, die Datei zu  */
      return fp;                        /*  oeffnen; falls kein    */
   else                                 /*  Fehler, Datei-Zeiger   */
   {  puts("Fehler beim Öffnen von");   /*  zurueckgeben; sonst    */
      puts(file);                       /*  mit Fehlermeldung ab-  */
      exit();                           /*  brechen.               */
   }                                    /*                         */
                                        /***************************/
}
```

**Prog. 7.3:** *Öffnen von Kanälen mit Fehlerüberprüfung.*

Diese Funktion können Sie einsetzen, um ein Programm zu schreiben, das eine Datei kopiert und den Namen der Quell- und Zieldatei als Argument von der Kommandozeile erhält:

```
main(argc, argv)
     int argc;
     char **argv;

{    FILE *in, *out, *f_open();
     int c;

     if (argc < 3)
     {   puts("\nBitte zwei Dateien angeben!");
         exit();
     }

     in = f_open(*++argv,"r");
     out = f_open(*++argv,"w");

     while ((c = getc(in)) != EOF)
         putc(c,out);

     fclose(in);
     fclose(out);
}
```

Kanäle können nicht nur zu Dateien eröffnet werden. Auch andere Ein-/Ausgabe-Geräte sind als erster Parameter für *open* (und entsprechend für *f_open*) zulässig. Über die verwendete Bezeichnungen müssen Sie sich jedoch im Handbuch versichern (bei der Beschreibung der *fopen*-Funktion). Üblich sind folgende Namen:

| | |
|---|---|
| CON: | für den Bildschirm |
| LST: oder PRN: | für den Drucker |
| AUX: | für die serielle Schnittstelle |

Wenn Sie das letzte kleine Programm mit dem Namen *fcpy* versehen haben, dann können Sie mit dem Aufruf:

```
fcpy fcpy.c CON:
```

die Quelle des Programms am Bildschirm anzeigen lassen.

### 7.3.2 Andere gepufferte Ein-/Ausgabe-Funktionen

Nicht nur die Funktionen *getchar* und *putchar* haben in *getc* und *putc* ein gepuffertes Äquivalent. Auch andere Ein-/Ausgabe-Funktionen wirken nicht nur auf die Standard-Kanäle, sondern in einer gepufferten Version auf beliebige andere Kanäle. Da hier die Konventionen bei den Compilern unterschiedlich sind, werde ich auf die einzelnen Funktionen nicht näher eingehen, sondern lediglich ihre Namen erwähnen; Sie müssen die Einzelheiten dann im Handbuch nachschlagen.

Zum Schreiben und Lesen von Strings gibt es neben *puts* und *gets* auch die gepufferten Varianten *fgets* und *fputs*. Für die formatierte Ausgabe steht neben *printf* auch *fprintf* zur Verfügung. Auch *scanf*, die formatierte Eingabe-Funktion, hat ein gepuffertes Äquivalent in *fscanf*. All diese Funktionen arbeiten mit einem FILE-Pointer als zusätzlichem Parameter.

Will man außer Zeichen und Strings auch andere Größen in der Ein-/Ausgabe verarbeiten oder ist man an einer Transaktion mit den Kanälen interessiert, die in größeren Einheiten vor sich geht, dann benutzt man die beiden Funktionen *fread* und *fwrite*.

Für die Eingabe dient *fread*; es hat 4 Parameter:

1. einen Puffer, in dem die gelesenen Daten abgelegt werden,
2. die Größe der zu übertragenden Einheiten,
3. die Anzahl der zu übertragenden Einheiten und
4. einen FILE-Pointer, der den zu lesenden Kanal bestimmt.

Der Puffer ist ein im Programm deklarierter Puffer und darf nicht mit dem internen Puffer verwechselt werden, den alle gepufferten Ein-/Ausgabe-Funktionen benutzen und selbsttätig verwalten.

Der Puffer, den Sie als erstes Argument übergeben, kann ein Zeichen-Array sein, wenn Sie die Übertragung von Zeichen wünschen. In diesem Fall würde man die Funktion *fread* so einsetzten:

```
.
.
.
char buff[512];
file *in, *f_open();
int nread;


in = f_open("EIN.DAT","r");
nread = fread(buff, sizeof(char), sizeof(buff), in);
.
.
.
```

Damit wird mit einer einzigen Anweisung der gesamte Puffer *buff* gefüllt - vorausgesetzt, die Datei EIN.DAT enthält genügend Daten (in diesem Fall mindestens 512 Bytes). Um herauszufinden, wieviele Zeichen tatsächlich übertragen wurden, gibt *fread* die Anzahl der übertragenen Einheiten zurück (die im Programmfragment in der Variablen *nread* vermerkt wird). Enthält die Datei keine Daten mehr, dann ist der Wert von *fread* gleich 0, so daß *fread* sich gut zur Integration in die Kontrollbedingung von Schleifen eignet.

Es ist aber ebensogut möglich, einen Integer-Array zu füllen; angenommen, die Datei EIN.DAT enthält einen Array mit Integers; dann kann man diese Integers aus der Datei wie folgt in einen programminternen Array transportieren:

```
.
.
.
int iarr[512];
file *in, *f_open();
int nread;

in = f_open("EIN.DAT","r");
nread = fread(iarr, sizeof(int), sizeof(iarr), in);
.
.
.
```

Je nachdem, welche Daten eine Datei enthält, können mit *fread* nicht nur einfache Datentypen, sondern z.B. auch Strukturen übertragen werden.

Dazu muß man *fread* als ersten Parameter einen Struktur-Array übergeben, als zweiten die Größe der Strukturen im Array und als dritten die Anzahl der zu lesenden Strukturen.

Da erhebt sich die Frage, wie man diese Strukturen aus dem Programm in eine Datei schreiben kann. Dazu dient die Komplementär-Funktion von *fread*, die den Namen *fwrite* trägt. Sie hat ebenfalls vier Parameter, die dieselbe Bedeutung wie bei *fread* haben, nur daß in ihrem Fall Daten aus dem Puffer (erster Parameter) in die Datei übertragen werden, deren FILE-Pointer letzter Parameter von *fwrite* ist. Wieviele Daten zu übertragen sind, gibt wieder der dritte Parameter an.

Das Programm 7.4 zeigt eine weitere Variante der Kopier-Routine, die mit *fread* und *fwrite* arbeitet. Wie bereits erwähnt, hat *fread* den Wert 0, wenn keine Daten mehr gelesen werden konnten und wurde deswegen gleich zur Kontrollbedingung der *while*-Schleife gemacht.

```
                                        /***************************/
#include <stdio.h>                      /* Fuer Ein-/Ausgabe.      */
                                        /*                         */
#define BUFSIZE 512 * 9                 /* Groesse Kopierpuffer.   */
                                        /***************************/


                                        /***************************/
fcopy(qs,zs)                            /* Kopiere Datei <qs> nach */
      char *qs, *zs;                    /*  <zs>.                  */
                                        /***************************/
{  FILE *quelle, *ziel, *f_open();      /*                         */
   char buff[BUFSIZE];                  /* Kopier-Puffer.          */
   int nread;                           /* Gelesene Zeichen.       */
   int nwrite;                          /* Geschriebene Zeichen.   */
                                        /*                         */
                                        /*                         */
   quelle = f_open(qs,"r");             /* Dateien oeffnen.        */
   ziel = f_open(zs,"w");               /*                         */
                                        /*                         */
   while (nread = fread(buff,1,BUFSIZE,quelle))  /* Kopier-Schleife.  */
   {  nwrite = fwrite(buff,1,nread,ziel);        /*                   */
      if (nwrite != nread)              /* Ueberpruefen, ob Schrei-*/
      {   puts("Fehler beim Schreiben!");  /*  ben erfolgreich; falls */
          break;                        /*  nicht, Schleife mit    */
       }                                /*  Fehlermeldung verlas-  */
   }                                    /*  sen.                   */
                                        /*                         */
   fclose(quelle);                      /* Dateien schliessen.     */
   fclose(ziel);                        /*                         */
}                                       /*                         */
                                        /***************************/
```

**Prog. 7.4:** *Kopieren von Dateien mit* fread *und* fwrite

Da *fwrite* als Wert die Anzahl der geschriebenen Zeichen zurückgibt, ist es möglich, den Erfolg des Schreibvorgangs zu überprüfen. In der Kopierschleife wird die Anzahl gelesener mit der Anzahl geschriebener Zeichen verglichen und bei Unstimmigkeiten die Schleife mit *break* verlassen.

## 7.4 Ungepufferte Ein-/Ausgabe

Neben der gepufferten Ein-/Ausgabe gibt es in C auch die Möglichkeit, Daten ohne Zwischenschaltung eines Puffers zwischen Dateien (oder anderen Geräten) und dem Programm zu übertragen. Dazu müssen Kanäle für die ungepufferte Ein-/Ausgabe geöffnet werden. Dies besorgt die Funktion *open*. Sie erhält - ähnlich wie *fopen* - als Argumente den Dateinamen und eine Angabe über den Modus, in dem die Datei geöffnet werden soll. Als Wert gibt sie eine Kanal-Identifikation zurück, die jedoch nicht so aufwendig gestaltet ist wie bei der gepufferten Ein-/Ausgabe. Da für die ungepufferte Kommunikation - wie der Name schon sagt - keine Puffer und damit zusammenhängende Verwaltungsaufgaben benötigt werden, reicht eine einfache Kanalnummer. Wert von *open* ist daher eine Integer, die die Kanalnummer (auch "file handle" genannt) darstellt. Drei ungepufferte Kanäle sind übrigens in jedem C-Programm schon bei Programmbeginn geöffnet und mit Kanal-Nummern versehen. Es sind dies

der Standard-Eingabe-Kanal mit Nummer 0
der Standard-Ausgabe-Kanal mit Nummer 1
der Standard-Fehler-Kanal mit Nummer 2

Bei jedem besseren C-Compiler sind in "stdio.h" für diese drei Kanäle bereits Namen vereinbart, und zwar

| | |
|---|---|
| *stdin* | für Kanal 0 |
| *stdout* | für Kanal 1 |
| *stderr* | für Kanal 2 |

Die ersten beiden Kanäle sind alte Bekannte; sie sind Ihnen bereits im Kapitel über die Umlenkung der Ein-/Ausgabe begegnet. Der dritte Kanal ist neu und hat folgende Berechtigung: *stdout* kann - ebenso wie *stdin* - bekanntlich umgelenkt werden. Üblicherweise führt *stdout* zum Bildschirm, aber ebensogut kann Dateiausgabe oder Ausgabe auf einen Drucker durch Umlenkung damit bewirkt werden. Tritt nun ein Fehler auf, den Sie in Ihrem Programm mit einer Fehlermeldung abfangen und würde die Fehlermeldung über *stdout* ausgegeben, dann kann es sein, daß der Benutzer sie niemals zu Gesicht bekommt.

Stellen Sie sich folgendes Szenario vor: Ihr Programm legt sämtliche Ausgaben auf den Kanal *stdout* - also normalerweise auf den Bildschirm. Nun wird *stdout* vom Benutzer in eine Diskettendatei umgelenkt. Aber die Diskette mit dieser Datei ist voll; beim Schreiben tritt deswegen ein Fehler auf. Ihr Programm registriert diesen Fehler zwar und will ihn ausgeben - doch die Fehlerausgabe erfolgt ebenfalls in die volle Datei, was natürlich nicht geht! Die Folge: das Programm stürzt ab und Sie wissen nicht, warum.

Deshalb reserviert C noch einen dritten Kanal *stderr*, der für die Fehlerausgabe vorgesehen ist und der stets zum Bildschirm führt. *stderr* kann auch nicht umgelenkt werden, so daß garantiert ist, daß Fehlermeldungen stets zu sehen sind.

Bleibt noch zu klären, wie man ungepufferte Kanäle öffnet und wie man Daten auf ihnen liest bzw. schreibt.

Zum Öffnen gibt es eine Funktion *open* mit zwei Parametern. Der erste ist der Dateiname (ein String), der zweite ist - anders als bei *fopen* - eine Integer, die den Öffnungs-Modus angibt. Die Angabe zum Öffnungs-Modus sind sehr compilerabhängig, weswegen Sie die folgenden Ausführungen auf jeden Fall mit Ihrem Handbuch abgleichen sollten. Die hier gemachten Angaben beziehen sich wie üblich auf das ATARI-C. Die Bedeutung der Werte für den zweiten Parameter:

0 Datei zum Lesen öffnen
1 Datei zum Schreiben öffnen
2 Datei zum Anfügen öffnen.

Unglücklicherweise setzt das *open* des ATARI-C bei allen drei Modi voraus, daß die zu öffnende Datei bereits vorhanden ist. Deswegen habe ich eine eigene Funktion *d_open* definiert, die als zweiten Parameter auch den Wert 3 akzeptiert und die in diesem Fall eine Datei mit dem gewünschten Namen kreiert. Sie bedient sich dazu der Funktion *creatb* aus der Standard-Bibliothek (die in Ihrem System eventuell auch den Namen *creat* trägt; Handbuch!), die ebenso wie *open* als Wert eine Kanalnummer zurückgibt oder -1, wenn das Kreieren nicht möglich war. Zum Öffnen bereits bestehender Dateien verwende ich nicht das beschrieben *open*, sondern eine Variante mit dem Namen *openb*, die die Datei als Binär-Datei eröffnet und so auch das Kopieren von Programmen erlaubt.

Das Programm 7.5 führt eine weitere Variante der Datei-Kopier-Routine vor, die jedoch mit ungepufferter Ein-Ausgabe arbeitet. Diesem Beispiel können Sie auch entnehmen, welche Funktionen in C die Ein-/Ausgabe bei ungepufferten Kanälen übernehmen.

```
#include <stdio.h>

#define BUFSIZE 512 * 9

fcopy(qs,zs)
      char *qs, *zs;

{  int quelle, ziel, d_open();
   char buff[BUFSIZE];
   int c;
   int nread;
   int nwrite;

   quelle = d_open(qs,0);
   ziel = d_open(zs,3);

   while (nread = read(quelle,buff,BUFSIZE))
   {   nwrite = write(ziel,buff,nread);
       if (nread != nwrite)
       {   puts("Fehler beim Schreiben!");
           exit();
       }
    }

   close(quelle);
   close(ziel);
}


int d_open(file,how)
            char *file; int how;

{
   int handle;

   switch (how)
   {  case 0 :
      case 1 :
      case 2 : handle = openb(file,how);
               break;
      case 3:  handle = creatb(file,2);
               break;
      default: puts("\nUnbekannter Modus f}r d_open!");
               exit();
    }

   if (handle != -1)
      return handle;
   else
   {  puts("Fehler beim \Oeffnen von");
      puts(file);
      exit();
   }

}
```

```
/***************************/
/* Datei kopieren -- unge- */
/*  pufferte Variante.     */
/***************************/
/*                         */
/* Bis auf die unterschied-*/
/*  liche Syntax von "read"*/
/*  und "write" ist hier   */
/*  alles voellig analog   */
/*  zu der gepufferten     */
/*  Version.               */
/*                         */
/*                         */
/*                         */
/*                         */
/*                         */
/*                         */
/*                         */
/*                         */
/* Auch ungepufferte Ka-   */
/*  naele muessen ge-      */
/*  schlossen werden, um   */
/*  das Directory zu ak-   */
/*  tualisieren!           */
/***************************/

/***************************/
/* Ungepufferten Kanal     */
/*  oeffnen.               */
/***************************/
/*                         */
/*                         */
/*                         */
/* Die Modi 0, 1 und 2     */
/*  werden an "openb" wei- */
/*  tergegeben. Modus 3    */
/*  ist selbstgestrickt    */
/*  und ruft "creatb".     */
/*                         */
/*                         */
/*                         */
/*                         */
/*                         */
/*                         */
/* Ueberpruefen, ob Oeff-  */
/*  nen bzw. Kreiren er-   */
/*  folgreich war.         */
/*                         */
/*                         */
/*                         */
/*                         */
/***************************/
```

**Prog. 7.5:** *Datei kopieren mit* read *und* write

Die Transaktion mit den ungepufferten Kanälen wird mit *read* und *write* abgewickelt. Diese Funktionen übertragen stets byteweise und benötigen daher nur drei Parameter: die Kanalnummer, die Anzahl zu übertragender Bytes und den internen Programmpuffer (einen Zeichen-Array), aus dem die Daten kommen bzw. in den sie geschrieben werden sollen. Als Kanalnummer sind neben selbstgeöffneten Kanälen selbstverständlich auch 0, 1 und 2 bzw. *stdin*, *stdout* und *stderr* möglich. Bedenken Sie jedoch: bei *stdin* und *stdout* gehen Sie das Risiko einer Umlenkung durch den Benutzer ein.

Die ungepufferte Ein-Ausgabe kann durchaus gegenüber der gepufferten Ein-/Ausgabe Geschwindigkeitsvorteile bringen, wenn die Puffer-Größe geschickt gewählt wird und stets ganze Puffer übertragen werden. Die in den Beispielprogrammen vereinbarte Puffer-Größe (Konstante BUFSIZ) entspricht auf dem ATARI einem Diskettensektor und ermöglicht somit sehr effizienten Zugriff.

## 7.5 Random-Zugriff auf Dateien

Über die Techniken der Dateiverwaltung müßte eigentlich ein eigenes Buch geschrieben werden. Ähnlich wie das Thema "Datenstrukturen" ist auch das Gebiet der Dateiverwaltung schier unerschöpflich und kann in diesem Buch nur an der Oberfläche angekratzt werden. Deshalb gebe ich mehr der Vollständigkeit wegen an, wie man in C im Random-Zugriff mit Dateien arbeitet. Sinnvolle Beispiele würden den Rahmen dieses Buchs sprengen.

In C erlauben zwei Funktionen den direkten Zugriff auf jedes einzelne Byte in einer Datei: eine für gepufferte und die andere für ungepufferte Kanäle. Die Funktionen heißen *lseek* und *fseek*; letztere ist die gepufferte Version.

Beide Funktionen haben drei Parameter:

1. eine Kanalidentifikation,
2. eine Byte-Position,
3. eine Modus-Angabe.

Bei *lseek* muß die Kanal-Identifikation eine Kanalnummer ("file handle") sein; *fseek* erwartet hier einen FILE-Pointer. Die beiden restlichen Parameter haben bei beiden Funktionen gleiche Bedeutung.

Um die Funktionsweise des Random-Zugriffs zu verstehen, können Sie sich vorstellen, daß alle Ein-/Ausgabe-Funktionen von C für die Arbeit mit Disketten-Dateien einen Zeiger benutzen, der auf die Dateiposition weist, an der die nächste Operation stattfinden soll. Beim Öffnen einer

Datei wird dieser Zeiger auf den Dateianfang gesetzt, es sei denn, die Datei wird zum Anhängen von Daten (Modus "a" bei *fopen* bzw. 2 bei *open*) geöffnet: dann wird der Zeiger hinter das letzte Byte in der Datei gesetzt.

Jedesmal, wenn Daten aus einer Datei gelesen oder in sie geschrieben werden, wird dieser Zeiger um die Anzahl der gelesenen Bytes weiterbewegt. Auf diese Art realisieren die E-/A-Funktionen die sequentielle Verarbeitung von Dateien.

Mit den Funktionen *lseek* und *fseek* ist es jedoch möglich, diesen Datei-Zeiger an jede gewünschte Stelle innerhalb einer Datei zu verschieben und so Abweichungen von der normalen sequentiellen Verarbeitungsweise zu erreichen. Nachfolgende E-/A-Operationen arbeiten dann ab dieser Stelle, so daß ein Random-Zugriff auf die Dateien möglich wird.

Der zweite Parameter von *lseek* und *fseek* muß eine *long*-Integer sein und gibt die Position des gewünschten Bytes an. Der dritte Parameter bestimmt, von wo ab die Angabe im zweiten Parameter gerechnet werden soll. Für den dritten Parameter sind folgende Werte zulässig:

| **Wert** | **Bedeutung** |
|---|---|
| 0 | Positioniere vom Dateianfang |
| 1 | Positioniere ab der aktuellen Position |
| 2 | Positioniere vom Dateiende |

Hier einige Beispiele:

```
lseek(out, (long) 20, 0);
```

damit wird in der Datei, die durch die Kanalnummer *out* bezeichnet ist, der Dateizeiger auf das zwanzigste Byte gesetzt. Wird unmittelbar daran die Anweisung

```
lseek(out, (long) 25, 1);
```

durchgeführt, dann steht der Dateizeiger jetzt an Position 45 in der Datei, denn beim Modus 1 wird die Positionsangabe ab der aktuellen Position des Dateizeigers gerechnet. Um den Dateizeiger an das Ende der Datei zu bewegen, schreiben Sie

```
lseek(out, (long) 0, 2);
```

Mit der Anweisung

```
lseek(out, (long) -34, 2);
```

bekommen Sie den Dateizeiger 34 Bytes vor das Ende der Datei gesetzt; es sind also auch negative Positionier-Werte erlaubt.

Nach dem Positionieren mit *lseek* oder *fseek* können Sie mit den passenden E-/A-Funktionen (da kennen Sie ja jetzt schon eine ganze Menge: *read*, *write*, *fread*, *fwrite*, *getc*, *putc*, *gets*, *puts*) auf die Daten zugreifen. Aber Vorsicht! Jede E-/A-Operation modifiziert ihrerseits den Dateizeiger, schiebt ihn um die Anzahl der gelesenen oder geschriebenen Zeichen weiter nach hinten. Deshalb muß bei relativer Positionierung (Modus 2) mit Umsicht zu Werke gegangen werden.

Übrigens ist es auch möglich, sich die aktuelle Position des Dateizeigers anzeigen zu lassen. Mit *ftell* bzw. *tell* erhalten Sie diesen Wert (als *long*-Integer; bei der Deklaration beachten!) im ersten Fall für gepufferte, im zweiten für ungepufferte Kanäle.

## 7.6 Zum Ausklang: Kopieren mit Namensmuster

Zum Ausklang des ersten Teils und als Überleitung zum zweiten Teil möchte ich Ihnen die Programmierung einer Utility zeigen, die nicht nur nützlich für die praktische Arbeit ist, sondern auch einige Routinen von TOS (dem Betriebssystem des ATARI) benutzt.

Ich ziehe es für die Programmierung unter C vor, nicht mit der grafischen Benutzeroberfläche von GEM zu arbeiten, sondern unter TOS den gewohnten kommandoorientierten Dialog mit dem Rechner zu führen. Dies geht, indem die zum Entwicklungssystem gehörende Routine COMMAND.PRG ausgeführt wird. Dabei handelt es sich um einen Kommandointerpreter, der - jedenfalls nach Beschreibung - ähnlich wie der Kommando-Interpreter unter MS-DOS (dem Betriebssystem des IBM PC und seiner vielen nahen und fernen Verwandten) funktioniert.

Er tut dies jedoch leider nur in begrenztem Umfang. Unter MS-DOS gibt es ein Kommando COPY, mit dem eine oder mehrere Dateien kopiert werden können. Die Syntax des Kommandos ist wie folgt:

```
COPY <quelle> [<ziel>]
```

Um z.B. die Datei OTTO.TXT vom Laufwerk A auf das Laufwerk B zu kopieren, schreibt man:

```
COPY A:OTTO.TXT B:
```

Beachten Sie, daß die Laufwerks-Angabe mit einem Doppelpunkt abgeschlossen wird. Wenn Laufwerk A ohnehin das aktuelle Laufwerk ist, dann reicht auch dieses aus:

```
COPY OTTO.TXT B:
```

Nun werden Sie einwenden, daß sich dies ebenso gut, ja sogar bequemer mit dem Kopierverfahren von GEM (Datei-Symbol mit MAUS anwählen und einfach Verschieben) erledigen läßt. Doch warten Sie ab:

```
COPY OTTO.TXT B:EMIL.DAT
```

Dies kopiert die Datei OTTO.TXT nicht nur auf das Laufwerk B, sondern gibt der Kopie auch einen neuen Namen, nämlich EMIL.DAT. Das können Sie unter GEM nur in 2 Schritten bewerkstelligen: erst mit der Maus kopieren, dann die Kopie anklicken und umbenennen. Das ist schon wesentlich umständlicher, da neben der Mausoperation ja doch noch ein Tastatur-Dialog nötig ist.

Es gibt jedoch auch diese Möglichkeit:

```
COPY ??DAT.* B:
```

Damit werden alle Dateien vom Laufwerk A nach B kopiert, deren Namen auf DAT endet, eine beliebige Extension besitzt und mit zwei beliebigen Buchstaben beginnt. Wenn auf Laufwerk A die Dateien

```
ABDAT.TXT
XYDAT.C
MYDAT.ASM
A2DAT.PRG
```

enthalten sind, dann werden all diese Dateien in einem Schwung mit nur einem Kommando kopiert. Unter GEM dürften Sie da mit der Maus unter Umständen ganz schön herumfummeln; denn GEM erlaubt Ihnen zwar, Dateien, die sich grafisch nebeneinander gruppieren lassen, mit dem Gummiband-Cursor gemeinsam zu kopieren. Aber zum Zusammengruppieren gibt's unter GEM nur das Sortieren: nach Namen und nach Extension. Und da streuen die obigen Dateinamen ganz schön auseinander.

Kurz und gut: COPY mit Wildcards ist oftmals praktischer als Kopieren mit der Maus. Wildcards (oder zu deutsch: Jokerzeichen) nennt man übrigens die Zeichen *?* und *. Diese sind Platzhalter; *?* für genau ein Zeichen, * für beliebig viele Zeichen.

"Was verschwendet er denn bloß so viele Worte auf das COPY, wenn es unter TOS 'eh da ist?", werden Sie jetzt denken. Ganz einfach: weil es nicht funktioniert. Jedenfalls nicht in der Version, die ich zur Verfügung hatte. Vielleicht hat sich das ja geändert, wenn Sie dieses Buch lesen. Aber vielleicht möchten Sie gar nicht unter TOS arbeiten, die Vorteile eines solchen COPY-Kommandos aber auch unter GEM genießen. Dann brauchen Sie nur das folgende Beispielprogramm abzutippen. Geben Sie Ihm jedoch nicht den Namen COPY (weil sonst TOS durcheinanderkommt und es

mit seinem eigenen, eingebauten, aber nicht funktionierenden COPY verwechselt).

Das Kopieren von Einzeldateien ist für Sie jetzt ja kein Problem mehr. Mittlerweile haben Sie die Auswahl zwischen drei Routinen, die Dateien kopieren (zeichenweise, gepuffert und ungepuffert). Das eigentliche Problem sind die Wildcards.

Glücklicherweise ist unter TOS alles halb so schlimm. Das Betriebssystem des ATARI stellt nämlich zwei sehr komfortable Funktionen zur Verfügung, die das Auffinden von Dateien auf einer Diskette, die auf ein Namensmuster passen, sehr erleichtern. Beide Funktionen durchsuchen dazu das Inhaltsverzeichnis der Diskette nach passenden Dateien.

Die erste Funktion (hier *Fsfirst* genannt) erhält ein Namensmuster als Argument (also einen Dateinamen, der eventuell eines oder mehrere Jokerzeichen enthält) und liefert als Ergebnis den Namen der ersten Datei auf der Diskette ab, die auf dieses Muster paßt. Die nächste Funktion (*Fsnext*) benötigt kein Argument; sie wird lediglich aufgerufen und liefert dann die nächste passende Datei ab. *Fsnext* - obwohl argumentlos - weiß, nach welchen Dateien es suchen soll, weil es Informationen benutzt, die von *Fsfirst* hinterlassen wurden. Es ist also unbedingt notwendig, die beiden Funktionen in der beschriebenen Reihenfolge aufzurufen.

Der Platz, an dem *Fsfirst* seine Botschaft an *Fsnext* hinterläßt ist übrigens derselbe, an dem es auch die gefundene Datei zurückgibt. *Fsfirst* und *Fsnext* kommunizieren miteinander und mit aufrufenden Funktionen nicht über Argumente und Parameter, sondern über einen Informations-Block, einer Struktur mit Namen DTA (Abkürzung für "Disk Transfer Address"), deren Deklaration Sie zu Beginn des Programms 7.6 sehen können. Dazu müssen beide Funktionen jedoch erfahren, wo sie diesen Informationsblock finden können; für diesen Zweck braucht's eine dritte Funktion *Fsetdta*, die die Adresse der DTA-Struktur beiden Funktionen bekanntmacht.

Das Programm 7.6 realisiert eine Funktion *step*, die ein Namensmuster als erstes Argument (Parameter *muster*) erhält und bei jedem Aufruf die nächste auf das Muster passende Datei in einen Argument-Puffer (Parameter *buff*) kopiert, die auf das Muster paßt - wenn sich eine finden läßt. Wenn nicht, wird der Wert 0 zurückgegeben. Das Hauptprogramm zeigt, wie man *step* benutzen kann, um sich alle auf der Diskette enthalten Dateien anzusehen, die auf ein Namensmuster (als Argument von der Kommandozeile eingegeben) passen. Das Programm funktioniert also genauso wie das TOS-Kommando DIR!

```
                                                /*************************/
extern long gemdos();                           /* Makros, die das BS rufen*/
#define Fsetdta(ptr)  gemdos(0x1a,ptr)          /*  DTA-Adresse setzen.    */
#define Fsfirst(name) (int) gemdos(0x4e,name,0) /*  Erstmalige Suche...    */
#define Fsnext() (int) bdos(0x4f)               /*  Folgesuche im Inhalts- */
                                                /*  verzeichnis (mit Mu-   */
                                                /*  ster!)                 */
#define TRUE 1                                  /* Englisch ist modern...  */
#define FALSE 0                                 /*                         */
                                                /*                         */
                                                /*                         */
struct DTA                                      /* Diese Struktur wird von */
{  char schrott[30];                            /*  Fsfirst() und Fsnext() */
   char name[14];                               /*  benoetigt.             */
} dta;                                          /*************************/

                                                /*************************/
main(argc,argv) int argc; char **argv;          /* Zum Testen. Programm    */
                                                /*  funktioniert wie DIR.  */
{  char buff[13];                               /*************************/
                                                /*                         */
   ++argv;                                      /* Zeiger auf Namensmuster */
                                                /*                         */
   while (step(*argv,buff))                     /* Ausgeben, was step()    */
      puts(buff);                               /*  findet.                */
                                                /*                         */
}                                               /*************************/


                                                /*************************/
int step(muster,buff)                           /* Schrittweises Durch-    */
        char *muster, *buff;                    /*  suchen des Inhalts-    */
                                                /*  verzeichnisses.        */
{  static int first = TRUE;                     /*************************/
                                                /*                         */
   switch (first)                               /* Erster Versuch.         */
   {  case TRUE:  Fsetdta(&dta);                /*                         */
                  if (!Fsfirst(muster))         /*  Passende Datei gefun-  */
                  {  strcpy(buff,dta.name);     /*  den --> kopieren.      */
                     first = FALSE;             /*  Schalter fuer naechsten*/
                     return TRUE;               /*  Zugriff setzen.        */
                  }                             /*                         */
                  else return FALSE;            /*                         */
                                                /*                         */
      case FALSE: if (!Fsnext)                  /* Folgeversuch.           */
                  {  strcpy(buff,dta.name);     /*  Was gefunden --> kopie- */
                     return TRUE;               /*  ren.                   */
                  }                             /*                         */
                  else                          /* Nichts mehr da; Schal-  */
                  {  first = TRUE;              /*  ter fuer naechsten     */
                     return FALSE;              /*  Aufruf zuruecksetzen.  */
                  }                             /*                         */
      }                                         /*                         */
}                                               /*************************/
```

**Prog. 7.6:** *Durchsuchen des Inhaltsverzeichnisses einer Diskette.*

Der Datenblock für den Informationsaustausch (die DTA) ist ein 41 Byte langer Bereich, bei dem die ersten 30 Bytes für unsere Zwecke nicht interessieren (daher auch der etwas despektierliche Name für diese Komponente). Daran schließt sich ein String mit dem gefundenen Dateinamen an. *Fsfirst* und *Fsnext* sind beide so freundlich, auch das abschließende Null-Byte an den Namen anzuhängen: ganz so, wie's C mag.

Natürlich findet sich in der DTA nur dann ein Name, wenn auf der Diskette (genauer: im Inhaltsverzeichnis) auch ein Namens-Eintrag steht, der auf das Muster paßt. *Fsfirst* und *Fsnext* signalisieren ihren Erfolg, indem sie den Wert 0 zurückgeben (das ist nicht so ganz die feine C-Art); andere Werte signalisieren eine Fehlanzeige. Für das seltsame Verhalten dieser Funktionen (0 bedeutet Erfolg, Ergebnisse werden nicht als Werte zurückgegeben, sondern über globale Variable) bin ich nicht verantwortlich; sie sind Teil des Betriebssystems und das haben die Jungs von Digital Research ihrem Schöpfer gegenüber zu vertreten... Um diese Funktionen benutzen zu können, müssen Sie lediglich in Ihrer Bibliothek eine Funktion *gemdos* finden (oder eine andere, die Ihnen den Zugriff auf Betriebssystems-Funktionen erlaubt). Zu Beginn des Programms 7.6 sehen Sie die nötigen Makro-Definitionen. Da *gemdos* in der Regel eine *long*-Integer zurückgibt, muß es deklariert werden.

Die Funktion *step*, die schrittweise das Inhaltsverzeichnis nach passenden Dateinamen durchsucht, befolgt hingegen - schließlich ist sie selbstgemacht - das C-Protokoll: findet sie etwas, dann gibt sie den Wahrheitswert "wahr" (Konstante TRUE) zurück; ansonsten ist ihr Wert "falsch" (Konstante FALSE). Das, was sie findet, kopiert sie in das zweite Argument, einen String, der lange genug sein muß, um einen Dateinamen aufzunehmen.

Nun muß *step* ein anderes Verhalten an den Tag legen, je nachdem, ob es mit einem Namensmuster zum ersten Mal aufgerufen wurde, oder ob es sich um einen Folgeaufruf handelt. Dazu macht die Funktion von einem internen Schalter Gebrauch, der sie darüber informiert. Die Variable *first* ist eine statische Integer und dient dazu, *step* von einem Aufruf zum nächsten darüber zu informieren, ob es bereits einmal benutzt wurde oder ob es sich um den ersten Aufruf (daher der Name) handelt.

Beim ersten Aufruf muß die DTA-Adresse gesetzt und die Suche mit *Fsfirst* eingeleitet werden. Findet *Fsfirst* einen passenden Eintrag, dann kann die Suche weitergehen, *first* wird also auf FALSE gesetzt. Außerdem kopiert *step* den gefundenen Namen aus der DTA, wo ihn *Fsfirst* ablegt, in den Argumentstring, über den es das Ergebnis an die aufrufende Funktion weitergibt.

Bei einem Folgeaufruf von *step* wird die Suche mit *Fsnext* fortgesetzt. Im Erfolgsfall wird wie soeben beschrieben verfahren. Kann kein weiterer Eintrag mehr gefunden werden, dann muß *step* nicht nur dies an die rufende Funktion melden (indem es FALSE zurückgibt), sondern auch den internen Schalter *first* wieder auf TRUE setzen, damit es für einen erneuten Aufruf mit einem anderen Namensmuster bereit ist.

Jetzt sind alle Zutaten für das Kopier-Kommando beisammen: eine Funktion, die Dateien kopiert und eine, die bei wiederholter Anwendung alle Dateien liefert, die auf ein vorgegebenes Namensmuster passen. Bleibt nur noch, diese beiden Funktionen in geeigneter Weise zu kopieren.

Doch Halt! So einfach darf man es sich nicht machen. Denn das Programm soll wie ein Kommando des Betriebssystems eingesetzt werden, und da muß man einigermaßen verantwortungsbewußt vorgehen, sprich: Benutzerfehler so weit als möglich abfangen. Deshalb nimmt die Fehlerüberprüfung im abschließenden Programm 7.7 beinahe die Hälfte des gesamten Quelltexts ein.

Nicht alle Verwendungen des Kommandos (nennen wir es KOPY) sind sinnvoll. Für die folgenden Darstellungen gehe ich davon aus, daß A das aktuelle Laufwerk ist. Der erste und einfachste Fall einer fehlerhaften Anwendung sieht so aus:

```
KOPY
```

Hier hat der Benutzer einfach zu sagen vergessen, was wohin kopiert werden soll. In diesem Fall sollte das Programm mit einem dezenten Hinweis auf die beabsichtigte Verwendungsweise reagieren.

KOPY kann nun mit einem oder mit zwei Parametern aufgerufen werden. Eine Anwendung mit einem Parameter wäre etwa:

```
KOPY B:OTTO.DAT
```

Dies kann man am besten als den Wunsch interpretieren, die Datei OTTO.DAT vom Laufwerk B auf das aktuelle Laufwerk (welches nach Annahme Laufwerk A ist) zu kopieren. Soweit ist das Ganze in Ordnung. Nicht in Ordnung ist jedoch dieses:

```
KOPY A:OTTO.DAT
```

Da A das aktuelle Laufwerk ist, müßte das als der Versuch aufgefaßt werden, eine Datei auf sich selbst zu kopieren; das darf KOPY nicht zulassen. Das kann man auf Parameter mit Namensmuster ausdehnen. So sollte

```
KOPY B:*.C
```

als der Wunsch interpretiert werden, alle C-Dateien vom Laufwerk B auf das aktuelle zu kopieren; ebenso muß

```
KOPY A:*.C
```

als unsinnig zurückgewiesen werden, wenn A das aktuelle Laufwerk ist.

Für die folgenden Ausführung tue ich mich etwas leichter, wenn Sie mir die Einführung von zwei Fachbegriffen erlauben: ein "vollqualifizierter Dateiname" ist einer, der keinerlei Wildcard- (Joker-)-Zeichen enthält (indem also weder ein *?* noch ein * vorkommt). Dateinamen mit Wildcards nennt man hingegen "teilqualifiziert" (weil da noch was fehlt).

Wird KOPY nun mit 2 Parametern aufgerufen, dann muß man unterscheiden, ob beides vollqualifizierte Dateinamen sind oder nicht. Der Aufruf

```
KOPY A:OTTO.DAT B:EMIL.C
```

bedeutet, die Datei OTTO.DAT vom Laufwerk A nach B zu kopieren und dort mit dem neuen Namen EMIL.C zu versehen. Das ist in Ordnung; nicht in Ordnung ist jedoch:

```
KOPY OTTO.DAT OTTO.DAT
```

Wieder soll eine Datei auf sich selbst kopiert werden; das erlaubt KOPY nicht.

Dieses ist OK:

```
KOPY A:*.C B:
```

Alle C-Dateien werden von A nach B kopiert. Falsch ist aber dieser Aufruf:

```
KOPY A:*.C B:ALLE.C
```

KOPY sträubt sich dagegen, mehrere Dateien in eine zu kopieren (obwohl es so erweitert werden kann, daß es auch dies beherrscht - doch das sei dem Leser als Übung überlassen). KOPY befolgt hier folgende Regel: wenn der erste Parameter teilqualifiziert ist, dann ist als zweiter Parameter nur mehr eine Laufwerks-Angabe zulässig. Schließlich ist noch eine Verwendung von KOPY auf jeden Fall verkehrt:

```
KOPY OTTO.DAT *.*
```

Gleichgültig, ob der erste Parameter voll- oder teilqualifiziert ist, der zweite darf niemals teilqualifiziert sein.

All diese Fehler fängt das Programm 7.7 ab. Dazu bedient es sich einiger Hilfsfunktionen: die Funktion *log* bestimmt das aktuelle Laufwerk; sie benutzt dazu eine Betriebssystems-Funktion *Dgetdrv*, die zu Beginn des Programms als Makro definiert ist und die eine Laufwerks-Nummer liefert (0 bedeutet A, 1 bedeutet B und schon haben Sie den Dreh...), welche von *log* in einen Buchstaben verwandelt wird.

Eine weitere Funktion *has_drive* überprüft, ob in einem Namensmuster eine Dateiangabe enthalten ist; dies ist der Fall, wenn das erste Zeichen der Angabe ein Buchstabe ist (Makro *ischar*) und das zweite ein Doppelpunkt.

Die Funktion *drive_of* liefert die Laufwerksangabe (das erste Zeichen in Großschreibung gewandelt); sie wurde nur der besseren Lesbarkeit wegen mit aufgenommen. Wichtig ist noch die Funktion *is_wild*, die überprüft, ob eine Angabe teil- oder vollqualifiziert ist. Dazu sucht sie nach den Zeichen * oder ? mit einer Bibliotheks-Funktion *index*, die als ersten Parameter eine String und als zweiten ein darin zu suchendes Zeichen hat. Wird das Zeichen gefunden, so liefert *index* den Wert "wahr" (siehe Handbuch).

Die Hauptfunktion bestimmt zuerst das aktuelle Laufwerk und beginnt dann mit einer Fehlerüberprüfung, die von der Anzahl der Parameter abhängt; deshalb ein *switch*. Kein Parameter ist ein Fehler; dies wird gemeldet. Bei einem Parameter muß das Programm sich überzeugen, daß dessen Laufwerksangabe ungleich dem aktuellen Laufwerk ist. Ist das der Fall, wird für die nachfolgende Verarbeitung das Quell- und Ziellaufwerk vermerkt und registriert, ob es sich um Kopieren mit Wildcard handelt.

Bei zwei Parametern ist die Fehlerüberprüfung schwieriger. Zuerst wird bei jedem Parameter die Laufwerks-Angabe mit *set_drive* extrahiert. Dies ist entweder das, was der Benutzer angegeben hat, oder das aktuelle Laufwerk, wenn keine Angabe erfolgte (siehe Definition von *set_drive*!). Der erste Fehler, der abgefangen wird, ist die Angabe eines teilqualifizierten Dateinamens als zweiter Parameter. Anschließend wird bestimmt, ob Kopieren mit Wildcard gewünscht ist, da davon die weitere Fehlerbehandlung abhängt. Bei Kopieren mit Wildcard darf der zweite Parameter nur eine Laufwerksangabe sein (nächste Überprüfung) und Quell- und Ziellaufwerk müssen voneinander abweichen. Sind beide Angaben vollqualifiziert, dann dürfen sie nicht gleich sein. Dazu wird die Funktion *strcmp* bemüht,die zwei Strings vergleicht und 0 zurückgibt, wenn sie gleich sind (Handbuch!). Schließlich sind alle Fehler abgefangen, das eigentliche Kopieren kann beginnen.

Die Funktion *fcopy* erledigt die Kopierarbeit, sie benötigt zwei Dateinamen für das Kopieren. Diese werden in zwei Strings mit Namen *source* (enthält die Quell-Datei) und *dest* (enthält die Zieldatei) aufgebaut. Waren zwei

vollqualifizierte Dateinamen angegeben, dann ist dies überflüssig und man kann ganz einfach die Parameter aus der Kommandozeile übernehmen. In allen anderen Fällen jedoch muß der Name für die Quell- und die Zieldatei erst bestimmt werden. Das Quell- und Ziellaufwerk kennen wir ja bereits (es ist im Laufe der Fehleranalyse angefallen). Es kann also bereits in die beiden Strings *source* und *dest* geschrieben werden. Was folgt, ist von der gewünschten Kopierart abhängig; jedenfalls setzten wir uns schon mal zwei Pointer *sp* und *dp* an die Stelle, an die die nächsten Informationen geschrieben werden.

Für das Kopieren verbleiben jetzt drei Fälle: zwei Parameter und beide vollqualifiziert, ein Parameter ohne Wildcard und schließlich zwei Parameter und Kopieren mit Wildcard. Der erste Fall ist der einfachste: die beiden Parameter aus der Kommandozeile werden lediglich an *fcopy* übergeben; das Programm ist fertig.

Im zweiten Fall ist der Name für das Kopier-Ziel nicht gegeben; er muß erst aufgebaut werden, kann jedoch vom ersten Parameter *file_1* übernommen werden.

Im dritten Fall werden die Namen der Quell-Dateien erst von *step* in einer Schleife geliefert. Durch Kopieren erhält man auch hier den Ziel-Namen. Das Programm ist so angelegt, daß der Ziel-Name jeder kopierten Datei auf dem Bildschirm ausgegeben wird. Somit hat der Benutzer noch eine letzte Kontrolle.

Sie können das Programm in der gewohnten Weise aus TOS aufrufen, oder es unter GEM als "TOS übernimmt Parameter" installieren. Bei meiner Version des C-Compilers ergab sich leider, daß Parameter mit Wildcards nur dann korrekt an das Programm übergeben wurden, wenn sie in Anführungszeichen eingeschlossen waren. Ich mußte also schreiben:

```
KOPY "*.C" A:
```

Sollte das Programm bei Ihnen abstürzen, dann kann dieser Fehler auch noch in Ihrem System enthalten sein.

```
extern long gemdos();                                   /*************************/
#define Dgetdrv() (int) gemdos(0x19)                    /* Aktuelles Laufwerk be- */
                                                        /*  stimmen.             */
char Logged_drv;                                        /* Aktuelles Laufwerk mer- */
                                                        /*  ken.                 */
                                                        /*************************/
main(argc, argv)                                        /* Dateien kopieren mit   */
    int argc;                                           /*  Jokerzeichen (Wild-   */
    char **argv;                                        /*  cards).              */
                                                        /*************************/
                                                        /*                       */
{                                                       /*                       */
    char *file_1, *file_2;                              /* Argumente des Kommandos */
    char log();                                         /*                       */
    char drive_of();                                    /*                       */
    char *sp, *dp;                                      /* Zeiger auf Quelle bzw. */
    char source[15], dest[15];                          /*  Ziel.                */
                                                        /*                       */
    char source_drv;                                    /* Quell-Laufwerk.       */
    char target_drv;                                    /* Ziel-Laufwerk.        */
    int  wild_card;                                     /* Kopieren mit Joker.   */
                                                        /*                       */
                                                        /*                       */
    Logged_drv = log();                                 /* Bestimme aktuelles LW. */
                                                        /*                       */
    switch    (--argc)                                  /* Parameter auswerten.  */
    {   case   0 : perror(0);                           /* Fehler: nix angegeben! */
                   exit();                              /*                       */
                                                        /*                       */
        case   1 : file_1 = argv[1];                    /* Nur eine Angabe:      */
                   if  (!has_drive(file_1)              /*  Verhindere Quelle =  */
                        || (drive_of(file_1) == Logged_drv))  /*  Ziel!        */
                   {   perror(1);                       /*                       */
                       exit();                          /*                       */
                   }                                    /*                       */
                   wild_card = is_wild(file_1);         /* Parameter setzen.     */
                   target_drv = Logged_drv;             /*                       */
                   source_drv  = drive_of(file_1);      /*                       */
                   break;                               /*                       */
                                                        /*                       */
        case   2 : file_1 = argv[1];                    /* Zwei Argumente zum Kom- */
                   file_2 = argv[2];                    /*  mando angegeben.     */
                   target_drv = set_drive(file_2);      /* Laufwerke fuer Kopie-  */
                   source_drv = set_drive(file_1);      /*  ren bestimmen.       */
                   if   (is_wild(file_2))               /* Wildcard in Quelle     */
                   {    perror(2);                      /*  nicht erlaubt.       */
                        exit();                         /*                       */
                   }                                    /*                       */
                   wild_card = is_wild(file_1);         /* Kopieren mit Wildcard? */
                   if   (wild_card && strlen(file_2) != 2)  /* Falls ja,        */
                   {  perror(3);                        /* darf 2. Argument nur   */
                      exit();                           /*  Laufwerksangabe sein! */
                   }                                    /*                       */
                   if (wild_card && source_drv == target_drv)  /*  Fehler      */
                   {  perror(1);                        /*  Quelle = Ziel abfangen.*/
                      exit();                           /*                       */
                   }                                    /*                       */
```

```
                if (!wild_card && !strcmp(file_1,file_2))  /*  Fehler     */
                {  perror(1);                             /*  Quelle = Ziel abfangen.*/
                   exit();                                /*                         */
                }                                         /*                         */
    }                                                     /* Jetzt kommt die eigent- */
                                                          /*  liche Kopiererei.      */
    if(!wild_card && argc == 2)                           /* Ohne Jokerzeichen ist   */
    { puts(file_2);                                       /*  es einfach...          */
      fcopy(file_1, file_2);                              /*                         */
      exit();                                             /*                         */
    }                                                     /*                         */
                                                          /*                         */
    source[0] = source_drv;                               /* Dateinamen vorbereiten. */
    dest[0] = target_drv;                                 /*                         */
    source[1] = dest[1] = ':';                            /*                         */
    sp = &source[2]; dp = &dest[2];                       /*                         */
                                                          /*                         */
    if(!wild_card && argc == 1)                           /* Nur ein Parameter; da   */
    {   strcpy(dp,&file_1[2]);                            /*  muss erst mal der Name */
        puts(dest);                                       /*  fuer das Ziel aufgebaut*/
        fcopy(file_1,dest);                               /*  werden.                */
        exit();                                           /*                         */
    }                                                     /*                         */
                                                          /*                         */
    while (step(file_1,sp))                               /* Passende Dateien suchen */
    { strcpy(dp,sp);                                      /* Zieldatei benennen.     */
      puts(dest);                                         /* Meldung an Benutzer.    */
      fcopy(source,dest);                                 /* Kopieren.               */
    }                                                     /*                         */
}                                                         /*************************/


                                                          /*************************/
set_drive(cp)                                             /* Laufwerksangabe setzen. */
        char *cp;                                         /*************************/
                                                          /* Wenn keine angegeben,   */
{  if (!has_drive(cp))                                    /*  aktuelles Laufwerk,    */
      return Logged_drv;                                  /*  ansonsten LW-Buchstabe */
   return drive_of(cp);                                   /*  extrahieren.           */
}                                                         /*************************/

                                                          /*************************/
char *Error[] =                                           /*  Fehlermeldungen      */
{   "\nVerwendungsweise: copy <quelle> [<ziel>]",         /*                       */
    "\nFehler: Quelle gleich Ziel!",                      /*                       */
    "\nFehler: Wildcard nur in Quelle erlaubt!",          /*                       */
    "\nFehler: Nur Laufwerksangabe als zweiter Parameter erlaubt!"  /*            */
} ;                                                       /*                       */
                                                          /*                       */
                                                          /*************************/
perror(i)                                                 /* Fehler ausgeben.      */
      int i;                                              /*************************/
{                                                         /* Das kennt man ja...   */
   puts(Error[i]);                                        /*************************/
}

                                                          /*************************/
char log()                                                /* Aktuelles Laufwerk be-*/
                                                          /*  stimmen (Arbeit macht*/
{   return 'A' + Dgetdrv();                               /*  das Betriebssystem.  */
}                                                         /*************************/
```

```
has_drive(str)
        char *str;

{
   return (ischar(*str) && *++str == ':') ;
}


char drive_of(str)
             char *str;

{   return (toupper(*str));
}

is_wild(str)
       char *str;

{   return (index(str,'*') || index(str,'?'));
}
```

```
/***************************/
/* Laufwerks-Angabe im     */
/*  string enthalten?      */
/***************************/
/* Buchstabe gefolgt von   */
/*  Doppelpunkt?           */
/***************************/

/***************************/
/* Laufwerksbuchstabe ex-  */
/*  trahieren.             */
/*                         */
/***************************/

/***************************/
/* Teilqualifizierter Da-  */
/*  teiname?               */
/***************************/
/* Nach Wildcards suchen.  */
/***************************/
```

**Prog. 7.7:** *Kopieren mit Wildcard.*

# Teil II:
# TOS-Programmierung

# 8 Das Betriebssystem des Atari

Dem Titel "C-Programmierung unter TOS" wurde das Buch bisher nur zum Teil gerecht. Zwar ist viel von C-Programmierung die Rede, wenig jedoch und nur marginal haben Sie bisher über TOS erfahren. Dies will ich nun nachholen. Der zweite Teil des Buches wendet sich deshalb der TOS-Programmierung, das heißt: der Systemprogrammierung zu. Krönender Abschluß (ich weiß: Bescheidenheit ist eine Zier...) dieses Teils und des Buches ist ein Diskmonitor (oder Diskette neditor oder Diskdoktor oder wie auch immer Sie ein Programm zu nennen gewohnt sind, mit dem Sie auf jedes Byte Ihrer Disketten zugreifen und es auch ändern können).

Dieses Programm demonstriert Mehreres: zum einen ersehen Sie daran, wie einfach die Systemprogrammierung in C unter TOS geht. Kein Wunder, ist ja TOS der Sprache C geradezu auf den Leib geschneidert. Dann aber demonstriert Ihnen das Programm auch die Technik (und den Stil), mit dem ein umfangreicheres Programmierprojekt in C angegangen werden soll, die sogenannte "Top-Down-Programmierung". Das Programm ist verhältnismäßig umfangreich, da ich großen Wert auf eine vernünftige Bedienerführung und Benutzeroberfläche gelegt habe.

Apropos "Benutzeroberfläche": mit diesem neudeutschen (und m.E. nicht sonderlich glücklich gewählten) Wort sind die 'Umgangsformen' gemeint, die ein Programm seinem Benutzer gegenüber an den Tag legt; es hat dies nichts damit zu tun, ob sich der Benutzer gewaschen hat oder nicht...

Zum Verständnis des Diskmonitors und der Systemprogrammierung im Allgemeinen sind einige Kenntnisse über Aufbau und Funktionsweise des Betriebssystems erforderlich. Ich kann Ihnen diese Kenntnisse hier nicht in vollem Umfang vermitteln; dann würde nämlich dieses ohnehin schon umfangreiche Buch aufgrund seines schieren Gewichtes untragbar werden. Dafür gibt es auch Spezialbücher, die sich nur diesem Thema widmen, wie z.B. das "ATARI Systemhandbuch" von I. und P. Lüke.

Anstatt Ihnen hier eine vollständige Liste aller Systemfunktionen und -variablen mit allen Einzelheiten zu geben, beschränke ich mich darauf, im Anhang eine kurze Übersicht über die für den Diskmonitor benötigten Teile des Systems (GEMDOS und BIOS) zu geben und einzelne, für das Programm besonders wichtige Funktionen ausführlicher zu erläutern. Sinn und Zweck dieses Teils ist es, Ihnen die Benutzung der Systemfunktionen exemplarisch vorzuführen, so daß Sie zur Entwicklung eigener Programme,

die andere Funktionen nutzen, in der Lage sind. Und er soll Ihnen den letzten Schliff als C-Programmierer geben.

Ehe wir uns jedoch auf die Programmierung des Editors stürzen, sind einige grundlegende Informationen über die Anatomie des Betriebssystems nötig.

## 8.1 TOS: Die Anatomie des Betriebssystems

Dem Computer-Anfänger erscheint es selbstverständlich, daß seine Maschine auf Tastendruck reagiert, Zeichen auf dem Bildschirm ausgeben und - im Falle des ATARI - seine Maus-Bewegungen auf der Tischplatte in Cursorbewegungen auf dem Bildschirm verwandeln, Fenster öffnen und schließen, vergrößern, verkleinern und verschieben und noch vieles andere mehr kann. Er vermutet hinter dieser Fähigkeit, mit der Außenwelt in Kontakt zu treten, meist eine 'angeborene' Fähigkeit des Computers. Aber "den Computer" als geschlossene Einheit, wie er vielleicht in der Vorstellung eines naiven Neulings existieren mag, gibt es gar nicht. Es gibt den Prozessor, aber dessen Fähigkeiten haben herzlich wenig mit Außenkontakten zu tun. Der kann in erster Linie einige geradezu lächerlich einfach erscheinende logische und arithmetische Operationen mit Bits und Bytes ausführen, kann seinen Arbeitsspeicher auf vielfältige Weise adressieren, beschreiben und lesen und die Ausführung von Programmen kontrollieren. Aber er kann - im übertragenen Sinn - nicht sprechen und hören.

Dazu ist er mit speziellen Geräten verbunden - eben den peripheren Geräten - die die Schnittstelle zur Umgebung darstellen. Es muß ihm beigebogen werden, wie er sich dieser Schnittstelle bedienen kann. Das machen Programme.

Was wie ein elementarer Vorgang erscheinen mag: Sie drücken auf eine Buchstaben-Taste Ihres Computers und der betreffende Buchstabe erscheint auf dem Bildschirm - das ist in Wahrheit Ergebnis eines gar nicht so trivialen programmgesteuerten Prozesses. Das Programm muß unter anderem überprüfen, ob die Tastatur gedrückt wurde, muß herausfinden, welche Taste Sie betätigten, muß diesen Tastendruck in einen entsprechenden Code umwandeln und diesen Code schließlich an den Programmteil weitergeben, der sich um die Bildschirmausgabe kümmert. Dieser Programmteil muß nun nachsehen, welches Punktemuster dem gewünschten Buchstaben auf dem Bildschirm entspricht und muß dafür sorgen, daß das Muster an der richtigen Stelle dort aufgebaut wird.

Die Dienste, die man unter dem Stichwort "Kommunikation mit der Peripherie" zusammenfaßt, stellt das Betriebssystem bereit. Diese Sammlung von fundamentalen Software-Diensten trägt auf dem ATARI den Namen TOS. TOS überwacht die Ein- und Ausgabe von Zeichen von der Tastatur und auf den Bildschirm bzw. auf andere Ausgabekanäle (wie z.B. den Drucker oder die serielle Schnittstelle). Sie können also TOS ein einzelnes Zeichen geben und verlangen, daß es dieses Zeichen an der aktuellen Cursorposition ausgibt.

Ganz besonders bedeutsam sind die TOS-Dienste, die mit der Diskettenverwaltung zu tun haben. TOS ist diejenige Instanz des Computers, die über Dateien sowie Inhalts- und Teilverzeichnisse (in der Terminologie von GEM "Ordner" genannt) Bescheid weiß. Man kann von ihm verlangen, ein Teilverzeichnis (einen Ordner) einzurichten bzw. zu schließen, das aktuelle Verzeichnis zu wechseln (einen Ordner zu verlassen und dafür einen anderen zu öffnen) und natürlich Dateien in diesen Verzeichnissen unterzubringen. Auch das Lesen und Schreiben von Dateiinformationen wird über TOS abgewickelt.

Schließlich kümmert sich TOS noch um die Speicherverwaltung. Der ATARI ST ist ja unter anderem aufgrund des üppigen Angebots an Arbeitsspeicher in begrenztem Umfang zum Multitasking fähig, d.h. es können mehrere Programme gleichzeitig im Speicher koexistieren und die Kontrolle kann diesen Programmen abwechselnd übergeben werden. Wird dies nur schnell genug gemacht, so hat der Benutzer den Eindruck, daß diese Programme auf seinem Computer gleichzeitig ablaufen. Außerdem können auf dem ATARI Programme andere Programme aufrufen und nach deren Beendigung die Kontrolle wieder übernehmen ("Programm-Chaining"). Das Betriebssystem hat die Aufgabe, für Programme Speicherplatz zu besorgen und diesen - falls er nicht mehr benötigt wird - wieder für andere Programme freizugeben. Auch läßt es Programme laufen, übergibt ihnen also die Kontrolle über den Prozessor und sorgt dafür, daß diese die von ihnen benötigte Arbeitsumgebung ("Base Page" und "Environment") vorfindet. Auch die Beendigung von Programmen und die Kontrollübergabe an die rufende Instanz werden von TOS überwacht.

All diese Dienste werden arbeitsteilig geleistet. Denn um Betriebssysteme nicht allzu maschinenabhängig werden zu lassen (dann müßte ein Hersteller sein System für jeden neuen Computer, ja schon für jede neue Hardwareerweiterung komplett umschreiben), verfolgt man bei ihrer Entwicklung die Strategie, sie in einen allgemeinen Teil und einen gerätespezifischen Teil zu gliedern.

Der gerätespezifische Teil ist diejenige Instanz, die wirklich weiß, wie man mit den Geräten zu reden hat. Diese Instanz trägt traditionell den Namen BIOS ("Basic Input Output System", zu Deutsch "Elementares Ein-/Ausgabe-System). Auch die Betriebssysteme CP/M und MS-DOS, die für 8-Bit-Computer bzw. IBM-kompatible 16-Bitter zum Standard geworden sind, nennen ihre 'Hardwareknechte' so.

Aber der ATARI ist wesentlich besser ausgestattet als diese Maschinen; eine Maus ist bei ihm serienmäßig enthalten. Seine Bildschirmauflösung ist sensationell und mit ihr seine Grafikfähigkeiten (auch in Farbe). Ebenfalls einzigartig ist, daß er mit einer MIDI-Schnittstelle zur Synthesizer-Steuerung versehen ist. Um all diese Extras nutzen zu können, hat man den ST mit einem zusätzlichen XBIOS versehen; dies ist die Abkürzung für "eXtended BIOS", also "erweitertes BIOS" zu Deutsch.

Diese beiden Module, die mit der Hardware direkt zusammenarbeiten, werden von einem übergeordneten Modul aufgerufen, in dem die 'Logik' des Betriebssystems steckt; es trägt den Namen GEMDOS (GEM, Abkürzung für "Graphics Environment Manager", ist die vertraute Benutzeroberfläche mit Menüs, Fenstern, Bildsymbolen; "DOS" ist die Abkürzung für "Disk Operating System", was soviel wie "Diskettenbetriebssystem" bedeutet). Die Modalitäten des Verkehrs mit der Außenwelt, die Organisation der Diskette und der Dateien und die Strategien der Speicherverwaltung sind GEMDOS bekannt. Es setzt diese mehr abstrakte Denkweise um in Kommandos an die BIOS- und XBIOS-Funktionen. Wie man sich dieses Zusammenspiel zwischen Betriebssystems-Logik (in GEMDOS) und hardwarenahen Funktionen (in BIOS und XBIOS) vorzustellen hat, wird das Kapitel über den Aufbau der Diskette und von Dateien noch genauer erläutern.

Das Betriebssystem des ATARI gliedert sich also in drei Module: GEMDOS, BIOS und XBIOS, wobei das Modul GEMDOS in diesem Triumvirat den Vorsitz hat. Diese drei Module sind das Betriebssystem; alle zusammen werden Sie mit dem überbegriff TOS bezeichnet.

Ist nun das also TOS, was Sie beim Einschalten des Computers zu sehen bekommen? Mitnichten! Es ist die Benutzeroberfläche von TOS (da haben wir das Wort schon wieder!), und die ist auswechselbar.

TOS hat nämlich alle Fähigkeiten, um zwischen dem Menschen und der Maschine mit all ihren Bestandteilen zu vermitteln, aber es tut dies nicht von selbst. Dazu ist ein weiteres Programm nötig, eines, das die Wünsche des Benutzers entgegennimmt und sie in Anweisungen an TOS übersetzt. Dies wiederum bedeutet, daß aufgrund der Benutzerwünsche GEMDOS-Funktionen angestoßen werden, die wiederum BIOS- oder XBIOS-Aufrufe

nach sich ziehen, welche wiederum dafür sorgen, daß die Hardware das tut, wozu sie da ist.

Dieses Entgegennehmen von Benutzerwünschen besorgt - wie sollte es anders sein - ein Programm. Beim Entwurf dieses Programms kann der Programmierer seiner Fantasie die Zügel schießen lassen, sprich: er kann die Benutzeroberfläche ganz nach eigenem Geschmack gestalten.

Ein Forschungsteam, das in dieser Beziehung mit besonders viel Geschmack ausgestattet war, arbeitete Anfang der 70er Jahre am Palo Alto Research Center (PARC) der Firma XEROX und war da mit der Entwicklung eines Personal Computers beschäftigt - also lange bevor es die Dinger tatsächlich auf dem Markt gab. In diesem Projekt wurde alles neu entwickelt: Hardware, Software ja sogar eine eigene Programmiersprache ("Smalltalk"), die ideal für nichtprofessionelle Computerbenutzer geeignet sein sollte. Dabei machte man sich auch grundlegende Gedanken über die Beschaffenheit der Benutzeroberefläche eines solchen Systems - und erfand die Windows, Menüs, Rollbalken usw. sowie das passende Gerät zum Umgang mit diesen: die Maus. Das, was die PARC-Forschungsgruppe sich ausgedacht hat, ist als objektorientierte grafische Benutzeroberfläche bekanntgeworden und die Programmierung unter Smalltalk, der Sprache, für die diese Grafik-Oberfläche erfunden wurde, nennt man "objektorientiertes Programmieren". Die Ideen der PARC-Gruppe waren so revolutionär, daß sie erst gute zehn Jahre später ihren Niederschlag in der Computer-Industrie fanden.

Zuerst war da Apple mit seiner Lisa und dem MacIntosh; dann zog die Firma Digital Research (DR), Schöpfer des ersten Industriestandard-Betriebssystems CP/M nach und entwickelte GEM. Dies finden Sie auch auf Ihrem ATARI. Doch weder Apple noch DR sind die Erfinder der grafischen Benutzeroberfläche; diese Ehre gebührt dem PARC-Team.

GEM ist ein sehr umfangreiches und komplexes Programmsystem, das jedoch letztendlich nur die Aufgabe hat, Benutzerwünsche in TOS-Kommandos umzusetzen, also als Vermittler zwischen dem Menschen und dem Betriebssystem aufzutreten. GEM ist aber beileibe nicht die einzige Lösung dieses Vermittler-Problems. Vor der Entwicklung der objektorientierten grafischen Benutzeroberfläche verkehrte man mit seinem Computer nicht wie die alten Ägypter via Hieroglyphen (auch 'Pictogramme' genannt), sondern durch über die Tastatur eingegebene Kommandos. Dieser etwas antiquierte, aber eingefleischten Programmierern - darunter auch mir - ans Herz gewachsene kommandoorientierte Zugang zur Maschine ist auf dem ATARI auch zu haben. Dazu benötigen Sie das Programm COMMAND, das Sie aus GEM anklicken und so dieses System durch einen konventionellen Kommando-Interpreter ersetzten können.

Das Zusammenspiel aller eben erwähnten Module soll noch einmal die Abbildung 8.1 verdeutlichen. In der obersten Schicht dieser Grafik sehen Sie die Benutzeroberfläche; hier haben Sie die Wahl zwischen COMMAND, GEM oder einem selbstgeschriebenem Programm.

Darunter folgt TOS mit seinen drei Modulen GEMDOS, BIOS und XBIOS; deren hierarchische Organisation ist auch optisch kenntlich gemacht. Dann erst - ganz unten - kommt die Hardware. Oder etwa nicht?

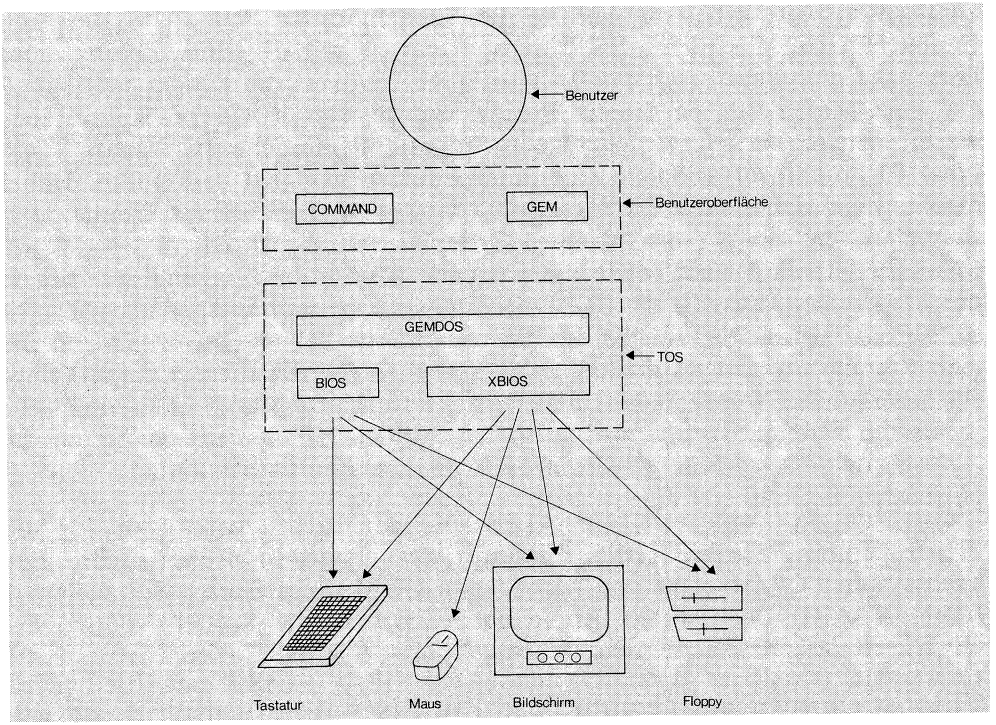



**Abb. 8.1:** *Zwischen Benutzer und Hardware: das Betriebssystem*

Die rhetorische Frage legt nahe, daß da noch etwas fehlt; es sind dies die sogenannten "Systemvariablen". Die drei Module des TOS müssen sich für ihre Arbeit bestimmte Informationen merken und benötigen Datenbereiche, über die sie miteinander kommunizieren können. Im Arbeitsspeicher des ATARI sind deshalb in einem geschützten Speicherbereich einige dedizierte Adressen vereinbart. Hierzu nur ein kleines Beispiel.

Der Arbeitsspeicher des ATARI kann erweitert werden; neben dem ST mit 512 KB gibt es ja bereits den ST+ mit einem Megabyte Arbeitsspeicher; aber theoretisch ist es denkbar, den ATARI auf bis zu 4 Megabyte aufzu-

blasen. Derjenige Teil von TOS, der sich um die Speicherverwaltung kümmert, muß natürlich wissen, wieviel Speicher gerade verfügbar ist, um korrekt arbeiten zu können. Diese Information findet er und alle Teile, die es sonst noch angeht, an der Adresse *0x42e* des ATARI. Die Entwickler des Systems garantieren, daß diese Information an genau dieser Stelle steht; deshalb habe ich gerade eben von "dedizierten" Speicherstellen gesprochen.

Die Systemvariablen und ihr Inhalt sind von vitaler Bedeutung für das Betriebssystems; deshalb muß verhindert werden, daß ihr Inhalt irrtümlich verändert wird und sich so eventuell ein fataler Programm-Absturz ereignet (z.B. einer, bei dem schnell noch die Diskette im Laufwerk gelöscht wird). Deshalb sind auf dem ATARI alle Systemvariablen in dem Speicherbereich zwischen den Adressen *0x000* bis *0x800* untergebracht, denn mit diesen Adressen hat es eine besondere Bewandtnis.

Der Prozessor kann nämlich auf diesen Speicherbereich nur zugreifen, wenn er sich in einem besonders privilegierten Modus (dem "Supervisor"-Modus) befindet. Ein normales Anwender-Programm läuft nicht unter diesem Modus, sondern im sogenannten "User"-Modus für die Unterprivilegierten. Das garantiert, daß Anwendungsprogramme nicht versehentlich (z.B. durch Verbiegen eines C-Pointers) die Systemvariablen korrumpieren können. Sie werden jedoch noch eine Methode kennenlernen, um - auf eigene Verantwortung - an die Systemvariablen heranzukommen.

## 8.2 Das Dateisystem

Auf Disketten werden Daten in Dateien gespeichert; das wissen Sie ja. Dateien haben einen Namen, über den sie angesprochen werden können. Und sie enthalten Informationen; das ist ja gerade der Witz an der Sache. Wenn Sie sich mit reiner Anwendungsprogrammierung beschäftigen wollen, dann ist das alles, was Sie über Dateien jemals wissen müssen.

Doch in Wirklichkeit sind die Verhältnisse nicht so einfach. Was dem 'naiven' Benutzer als Einheit erscheinen mag (Dateiname und Dateiinformation), wird erst durch Vermittlung des Betriebssystems und unter Anwendung einer verhältnismäßig komplexen Logik für den Benutzer so einfach zu handhaben. Die Betriebssystems-Funktionen, die mit der Diskette zu tun haben und die Verwaltungslogik, welche sie realisieren, faßt man unter dem Namen "Dateisystem" zusammen. Da das Verständnis des Dateisystems für die Programmierung eines Diskettenmonitors Voraussetzung ist, will ich mich diesem jetzt zuwenden.

### 8.2.1 Der Aufbau einer Diskette: das Grundgerüst

Auch die unterste Ebene in der Verarbeitungshierarchie kommt nicht ohne Struktur aus. Um auf einer Diskette Informationen unterbringen zu können ist es nötig, ein einfaches Grundgerüst für die Informationsspeicherung auf der Diskette festzulegen, in das dann die Informationen eingefügt werden können. Stellen Sie sich das ähnlich wie bei einer Bibliothek vor, in der Sie Ihre Bücher organisieren wollen. Sie könnten natürlich alle Bücher auf einen großen Haufen schmeißen; aber dann wäre der Zugriff auf die einzelnen Bücher sehr langsam, wenn er überhaupt möglich ist. Denn jedesmal, wenn Sie ein Buch suchen, müssen Sie den ganzen Haufen danach durchwühlen.

Das macht wohl keiner so; vielmehr überlegt man sich ein Organisationsschema, das einen schnelleren Zugriff ermöglicht. Dieses Schema erfordert eine bestimmte Struktur (z.B. die Anordnung der Bücher nach Autorennamen) und um diese Struktur effizient verwirklichen zu können sind Voraussetzungen nötig. Die Voraussetzungen für eine effiziente Bibliotheks-Organisation sind Regale.

Die Voraussetzung für effiziente Informationsspeicherung auf Diskette ist ein Disketten-Format, eine Unterteilung der Diskette in Spuren und Sektoren. Die Disketten-Sektoren entsprechen den Bücherregalen im Bibliotheks-Beispiel.

Durch das Formatieren wird diese Unterteilung auf der Diskette installiert. Es wird ihr also eine bestimmte Strukturierung aufgeprägt, die herstellerseitig noch nicht auf dem Datenträger vorhanden ist. Dies ist Aufgabe eines eigenen Formatier-Programmes; es versieht beim Formatieren die Diskette mit allen nötigen Informationen, die die Steuerelektronik für das Laufwerk benötigt, um die Untergliederung in Spuren und Sektoren wahrzunehmen.

Spuren sind konzentrische Kreise auf der Diskette; legt man über diese konzentrischen Kreise noch radiale Schnitte, vergleichbar Kuchenstücken bei einer Geburtstagstorte, so erhält man die endgültige Untergliederung (vgl. Abbildung 8.2). Die Oberfläche der Diskette wird durch diese zweidimensionale Gliederung in einzelne Abschnitte unterteilt, die man in der Terminologie von GEMDOS "logische Sektoren" oder ganz einfach "Sektoren" nennt, und die von Null beginnend durchnumeriert werden.

Die Gliederung der Diskettenoberfläche in Sektoren ist keine Erfindung von ATARI; auch andere Computer, die mit Disketten arbeiten, bedienen sich ihrer. Die Formate unterschiedlicher Computer (Anzahl der Spuren und Sektoren, Länge der Sektoren usw.) unterscheiden sich jedoch voneinander; deshalb können Disketten nicht zwischen verschiedenen Geräten frei

ausgetauscht werden. Auch bei ATARI gibt es Unterschiede, ja nachdem, ob Sie mit 40 oder 80 Spuren, mit einseitigen oder doppelseitigen Laufwerken arbeiten. Die Zahl der logischen Sektoren auf einer ATARI-Diskette kann daher unterschiedlich sein. Alle ATARI-Laufwerke bringen jedoch pro Spur der Diskette 9 logische Sektoren unter. Auch die Sektorgröße ist bei ATARI für Disketten einheitlich; Sie beträgt stets 512 Bytes.

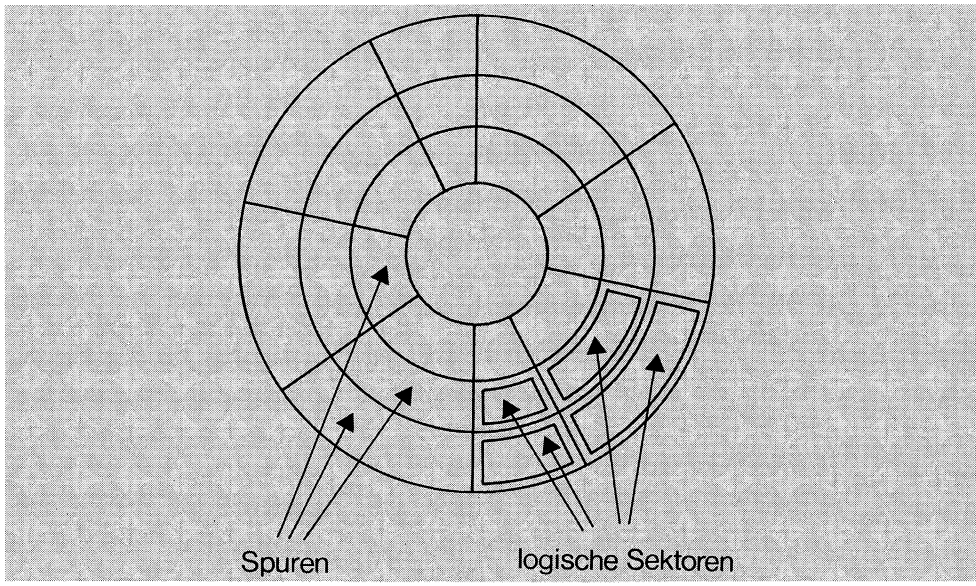



**Abb. 8.2:** *Unterteilung der Diskette in logische Sektoren.*

Das durch die logischen Sektoren vorgegebene Grundraster wird für die Zwecke des GEMDOS nun in 3 logische Bereiche unterteilt:

*1. Der Boot-Sektor*

Er ist stets der erste Sektor auf der Diskette (Nummer 0) und enthält Informationen über die Diskette (Anzahl der Sektoren, Seiten usw.). Soll beim Urstart des Computers (Einschalten oder Betätigen des Reset-Knopfs) ein bestimmtes, auf der Diskette gespeichertes Programm ausgeführt werden, so ist auch dies im Boot-Sektor vermerkt.

*2. Der Verwaltungsbereich*

Dieser Bereich schließt sich an den Boot-Sektor an und wird von GEMDOS dazu benötigt, gewisse buchhalterische Informationen über die Dateien auf der Diskette zu verwalten. Seinem Aufbau ist das folgende Unterkapitel

gewidmet. Auf einer typischen Diskette belegt der Verwaltungsbereich die Sektoren 1 mit 17.

*3. Der Datenbereich*

Hier sind die eigentlichen Informationen untergebracht, also die Daten, die Sie auf der Diskette gespeichert haben. Der Datenbereich nimmt auf der Diskette den weitaus größten Platz ein. Auf einer einseitigen Diskette belegt er die Sektoren 18 bis 720, auf einer zweiseitigen Diskette die Nummern 18 bis 1440.

### 8.2.2 Der Aufbau des Verwaltungsteils

Eine Datei hat zwei Komponenten: einen Namen, unter dem sie angesprochen werden kann und Daten, die in ihr gespeichert sind. Im Verwaltungsbereich führt GEMDOS Buch über die Dateinamen und darüber, wo im Datenbereich die zur Datei gehörigen Informationen zu finden sind. Um bei dem Bibliotheksbeispiel zu bleiben: der Verwaltungsbereich ist so etwas wie ein Bibliothekskatalog. Er sagt, welche Bücher vorhanden sind (entspricht dem Verzeichnis der Dateinamen) und wo man sie finden kann (entspricht dem Verweis auf den Datenbereich).

Wird eine Datei neu eingerichtet und mit Daten beschrieben, dann legt GEMDOS in einem speziellen Bereich des Verwaltungsteils - das sog. "Inhaltsverzeichnis" oder "Directory" - Informationen darüber ab, wie die Datei heißt (Name und Extension). Außerdem besorgt es im Datenbereich Platz für die Informationen in der Datei.

Dieses Verfügbarmachen von Platz aus dem Datenbereich nennt man fachmännisch auch "allokieren"; dazu sind einige Erklärungen nötig. Angenommen, Sie haben eine leere Diskette und wollen jetzt Informationen in einer neuen Datei namens DATEI1 speichern. Dann besorgt GEMDOS den nächsten freien Platz im Datenbereich und weist ihn der Datei zu. Wird jetzt diese Datei geschlossen und eine andere - nennen wir sie DATEI2 - geöffnet und beschrieben, dann wird wiederum der nächste freie Platz im Datenbereich für DATEI2 allokiert. Erneutes Beschreiben von DATEI1 führt nun dazu, daß auf der Diskette die Daten für DATEI2 von den Daten für DATEI1 'umrahmt' sind, oder - anders ausgedrückt: die logisch zusamengehörigen Informationen zu DATEI1 sind physikalisch auf der Diskette verstreut (vgl. Abbildung 8.3).

Dagegen läßt sich nichts machen, da GEMDOS ja nicht von vorneherein den maximalen Platzbedarf jeder einzelnen Datei kennt. Um dennoch die Dateiinhalte richtig zusammenklauben zu können, muß sich GEMDOS irgendwie merken, was wo steht.

Hierbei kommt es wesentlich darauf an, in welchen Einheiten (Größenordnungen) Platz im Datenbereich allokiert wird. Theoretisch könnte man, da Dateien byteweise beschrieben werden, immer nur ein Byte auf einmal allokieren. Im schlimmsten Fall würde das aber dazu führen, daß lauter ein Byte lange Fragmente der Datei über die Diskette verstreut sind und man sich den Ort all dieser Bytes merken muß. In diesem Fall ist der Verwaltungsaufwand enorm, ja, man würde für die Verwaltung der Informationen ebensoviel Platz benötigen wie für ihre Speicherung (ein Zustand, den manche real existierende Verwaltungen angeblich bereits erreicht haben)!

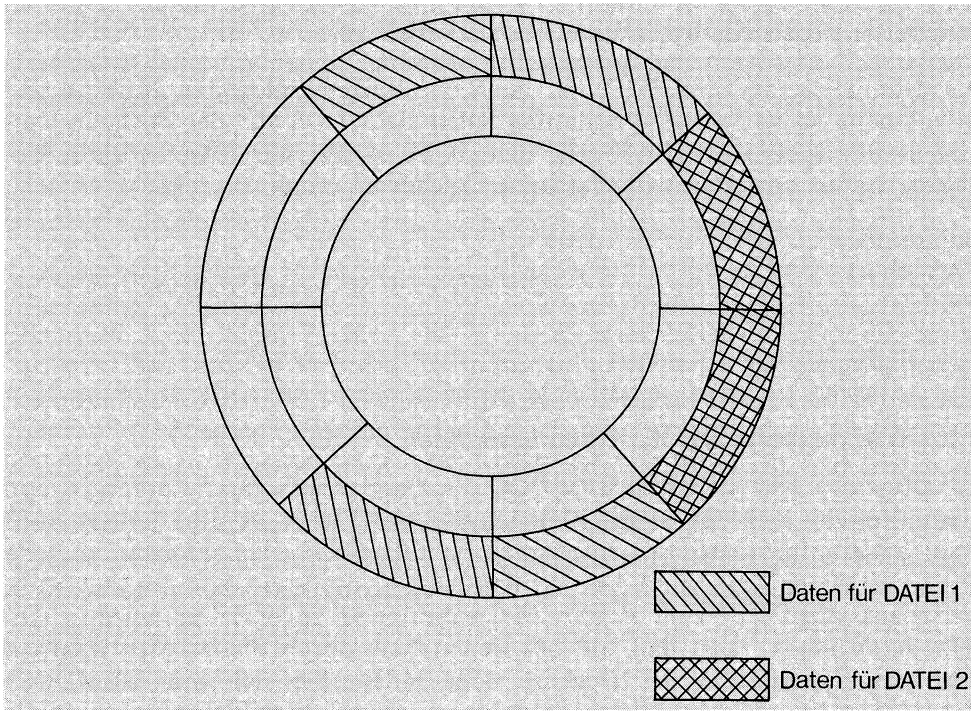



**Abb. 8.3:** *Gestreute Speicherung logisch zusammengehöriger Informationen*

Deshalb haben die Entwickler von GEMDOS einen anderen Weg beschritten. Der Datenbereich wird in Zuweisungseinheiten von jeweils 2 Sektoren verwaltet. Diese Zuweisungseinheiten nennt man auch "Cluster". Jedesmal, wenn für Informationsspeicherung in einer Datei Platz benötigt wird, wird der Datei ein Cluster (also 1024 Byte = 1K) zugewiesen. Dies geschieht auch dann, wenn die Datei tatsächlich nur 100 Byte an Informationen enthalten soll. Die restlichen 924 Byte sind dann verschwendeter Platz, sogenannter "Verschnitt".

GEMDOS merkt sich beim Einrichten der Datei den Anfangscluster und füllt ihn beim Beschreiben solange mit Informationen auf, bis er erschöpft ist. Dann wird der Datei irgendwo im Datenbereich ein neuer Cluster zugewiesen und es stehen wieder 1024 Byte zur Verfügung. Natürlich muß sich GEMDOS aber auch merken, wo dieser zweite und alle folgenden Datencluster der Datei zu finden ist.

Dafür gibt es die FAT. Die FAT ("file allocation table") ist eine Liste, die darüber Auskunft gibt, welche Datencluster den einzelnen Dateien zugewiesen sind. Den ersten Cluster einer Datei findet man im Directory. Sucht man die Folgecluster, so muß man in der FAT nachsehen, die also eine Art Wegweiser durch den verstreuten Datenbereich ist.

All dies, Directory und FAT, ist im Verwaltungsbereich der Diskette untergebracht. Wegen seiner Bedeutung führt GEMDOS auf der Diskette stets eine zweite Kopie der FAT mit, um im Fehlerfall mit dieser weiterzuarbeiten. Damit ergibt sich die folgende Gliederung einer Diskette:

Bootsektor
FAT Nr. 1
FAT Nr. 2
Directory
Datenbereich

### 8.2.3 Das Directory aus der Nähe betrachtet.

Im Directory hält sich das GEMDOS die Dateinamen samt Extension und einen Verweis auf den ersten Datencluster der Datei. Es gibt jedoch auch noch andere Informationen im Directory. GEMDOS versieht nämlich jede Datei mit einer Zeit- und Datumsmarkierung, die Auskunft darüber gibt, wann auf die Datei zum letzten Mal schreibend zugegriffen wurde. Dies setzt natürlich voraus, daß Sie bei der Arbeit durch Stellen der internen Uhr (im Kontrollfeld des Desktop oder indem Sie sich mit GEMDOS-Funktionen ein eigenes Programm dafür schreiben) dem Computer stets die richtige Vorstellung von Zeit und Datum beigebracht haben. Außerdem ist es möglich, eine Datei mit bestimmten Attributen zu versehen. Man kann sie als schreibgeschützt kennzeichnen, kann sie von der normalen Anzeige des Inhaltsverzeichnisses (beim Öffnen einer Diskette im GEM oder mit dem DIR-Befehl unter COMMAND) ausschließen (und somit 'verstecken') und noch einiges mehr.

Und dann gibt es da noch zwei besondere Arten von Dateien. Beim Formatieren einer Diskette können Sie eine (maximal 11 Zeichen lange) Diskettenkennung vergeben. Diese Diskettenkennung speichert GEMDOS im Directory. Da aber mit einer Kennung keine verwendbaren Informationen

verknüpft sind, muß man den Verzeichnis-Eintrag für die Kennung gesondert markieren.

Die zweite Kategorie der Einträge wird durch die hierarchische Dateistruktur von GEMDOS eingeführt. Sie können nämlich im Desktop nicht nur Dateien einrichten, sondern auch Unterverzeichnisse, die in der GEM-Terminologie "Ordner" genannt werden. Aus der Sicht des Betriebssystems sind Ordner nichts anderes als Dateien, die jedoch nicht mit Daten beginnen, sondern an deren Anfang ein eigenes Directory für das Unterverzeichnis steht. Diese Unterverzeichnisse können wiederum auf weitere Unterverzeichnisse verweisen usw. Ein Verweis auf ein Unterverzeichnis steht - ähnlich wie ein normaler Datei-Eintrag - in einem Directory, ist jedoch gesondert gekennzeichnet, damit er nicht mit einem normalen Dateieintrag verwechselt werden kann.

Da Unterverzeichnisse auch Dateien sind, kann es vorkommen, daß im Datenbereich einer Diskette Records zu finden sind, deren Aufbau einem Directory entspricht.

### 8.2.4 Der Aufbau eines Directory-Eintrags

Wer sich schon einmal näher mit MS-DOS beschäftigt hat, dem Betriebssystems-Standard für Personal Computer, die auf den Prozessoren von Intel basieren, dem wird das soeben Gesagte über den Aufbau des Directory sehr vertraut vorkommen. Tatsächlich entspricht die Struktur des Directory und der FAT einer Diskette beim Atari völlig der von MS-DOS. Das geht sogar so weit, daß die Directory-Einträge auf der Diskette in der für die Intel-Prozessoren typischen Weise gespeichert werden: bei einem 16-Bit-Wort kommt zuerst das niedrigstwertige und dann erst das höchstwertige Byte. Eine "long integer" müßte daher - auf die Verhältnisse auf dem Motorola 68000 übertragen - 'von hinten' gelesen werden, um richtig interpretiert zu werden.

Daher sind für die Ausgabe eines Directory-Eintrags durch ein C-Programm eine Menge an maschinennahen Byte-Fummeleien erforderlich, um alles wieder wie gewohnt hinzubiegen. C kann hier in seiner Eigenschaft als Quasi-Assembler im vollen Lichte erstrahlen!

Jeder Directory-Eintrag ist 32 Byte lang; in einem ATARI-Sektor haben somit 32 Directory-Einträge Platz. Der Directory-Eintrag gibt, wie bereits erwähnt, Auskunft darüber:

- welchen Namen und welche Extension die Datei hat
- wie groß sie ist

- ob es sich um eine normale Datei, ein Teilverzeichnis (Ordner) oder die Diskettenkennung handelt
- wo auf der Diskette die zugehörigen Daten zu finden sind (die Anfangsclusternummer)
- welche Attribute gesetzt sind (z.B. "nur Lesezugriff")
- welche Datums- und Zeitinformation der Datei zugeordnet sind

All dies ist in folgender Datenstruktur untergebracht:

*Adresse 0 bis 7:*

Der Dateiname; auf normalem Weg (von GEM oder COMMAND) angelegte Dateinamen sind maximal 8 Zeichen lang und setzen sich ausschließlich aus Großbuchstaben zusammen. Eine Ausnahme bilden Diskettenkennungen, die auch Null-Bytes enthalten. Mit dem Diskmonitor ist es natürlich möglich, 'seltsame' Dateinamen zu erzeugen, die z.B. Kleinbuchstaben oder Grafikzeichen enthalten. Wozu das gut sein soll, müssen Sie allerdings schon selber wissen.

Das erste Byte im Directory-Eintrag hat jedoch noch eine spezielle Bedeutung. Es kann die Werte 0x00, 0x2e und 0xe5 annehmen. Die Bedeutung dieser Werte:

*Byte 0 ist 0x00*: Es handelt sich um einen unbenutzten (und völlig leeren) Directory-Eintrag. Dies macht GEMDOS, um die Suche im Directory nach freien Einträgen zu beschleunigen.

*Byte 0 ist 0xe5*: Die Datei ist gelöscht. Daran kann man ersehen, daß "Löschen von Daten" auch unter GEMDOS erstmal nur ein logisches Löschen ist. Das Betriebssystem markiert lediglich das erste Byte im Namen der Datei, tut aber sonst noch nichts mit den Daten.

*Byte 0 ist 0x2e* (also die Zeichenkonstante '.'): Der vorliegende Record ist ein Teilverzeichnis. Teilverzeichnisse beginnen stets mit den beiden Namenseinträgen "." und ".."; weitere Informationen finden Sie in den Erläuterungen zur Clusternummer (Adressen 0x1a und 0x1b).

Teilverzeichnisse sind ja nichts anderes als Dateien, allerdings solche, die nicht mit Daten anfangen, sondern erstmal mit weiteren Inhaltsverzeichnis-Einträgen. Die Clusternummer (s.u.) im Verzeichniseintrag sagt Ihnen, wo auf der Diskette das Teilverzeichnis zu finden ist. Die ersten beiden Einträge in einem Teilverzeichnis sind üblicherweise wieder Teilverzeichnis-Records. Einer zeigt auf das Teilverzeichnis selbst, der zweite (bei dem Byte 0 und Byte 1 den Wert 0x2e haben) zeigt auf das Elternverzeichnis des aktuellen Teilverzeichnisses.

*Adresse 8 - 0xa*

Extension der Datei; entweder lauter Leerzeichen oder Großbuchstaben.

*Adresse 0xb:*

Dateiattribut. Hierbei handelt es sich um einen Bitvektor; die einzelnen Bitpositionen sind wie folgt zu interpretieren:

*Bit 0 gesetzt*: Auf die Datei darf nur schreibend zugegriffen werden.

*Bit 1 gesetzt*: Die Datei ist unsichtbar (wird im Inhaltsverzeichnis nicht angezeigt)

*Bit 2 gesetzt*: Systemdatei; ebenfalls keine Anzeige im Inhaltsverzeichnis

*Bit 3 gesetzt*: Der Eintrag ist die Diskettenkennung; außer dem Namen und dem Attribut enthält dieser Eintrag keine sinnvolle Information mehr

*Bit 4 gesetzt*: Der Eintrag ist für ein Teilverzeichnis

*Bit 5 gesetzt*: Das sog. "Archiv"-Bit. GEMDOS nutzt diese Möglichkeit nicht, Sie können aber in diesem Bit kennzeichnen, ob die Datei beschrieben wurde und dies zum Sichern von Dateien benutzen.

*Adresse 0xc - 0x15*

Reserviert; wird von GEMDOS augenscheinlich nicht benutzt.

*Adresse 0x16 - 0x17*

Datumsinformation. Es handelt sich hierbei um ein 16 Bit langes Bitfeld, dessen einzelne Bitpositionen wie folgt zu interpretieren sind:

```
Bit 15                    Bit 0
H H H H H M M M M M M S S S S S
|         |
Adresse 0x17    Adresse 0x16
```

Die Bits 15 bis 11 sind als die Binärzahl für die Stunde (0 - 23) zu interpretieren; die Bits 10 bis 5 liefern, als Binärzahl gelesen, die Minuten (0 - 59) und die Bits 4 bis 0 enthalten die Sekundeninformationen (in 2-Sekunden-Schritten).

*Adresse 0x18 - 0x19*

Zeitinformation; auch dies ist ein Bitfeld ähnlich dem obigen. Es ist wie folgt zu interpretieren:

```
Bit 15                    Bit 0
J J J J J J J M M M M D D D D D
|              |
Adresse 0x19   Adresse 0x18
```

*Bit 15 bis 9*: das Jahr, beginnend mit 1980 gezählt
*Bit 8 bis 5*: der Monat (1 - 12)
*Bit 4 bis 0*: der Tag (1 - 31)

*Adresse 0x1a - 0x1b*

Anfangscluster der Dateiinformation; aufgrund der Struktur der FAT hat der erste Datencluster die Nummer 2.

Die Clusternummer eines Teilverzeichnisses verweist auf den Anfang desselben im Datenbereich. Dies beginnt erstmal selbst mit einem Directory mit dem hier beschriebenen Aufbau, dessen erste beide Einträge stets auch Verweise auf Directories sind, die die Namen "." und ".." (Sie haben schon richtig gelesen!) tragen.

. verweist in der Cluster-Nummer auf sich selbst (warum?)
.. verweist in der Cluster-Nummer auf das Elternverzeichnis (steht hier die Clusternummer 00, so ist das Elternverzeichnis die Wurzel des Teilverzeichnis-Baumes). Mit dem Eintrag "." ist es z.B. dem DIR-Kommando möglich, den Anfang für die Anzeige zu finden; mit ".." kann man aus einem Teilverzeichnis um eine Stufe 'nach oben' steigen.

Hinter diesen beiden Einträgen mit Sonderfunktion kommen bei den Teilverzeichnissen die Einträge für Dateien und eventuelle weitere Unterverzeichnisse, die ganz normal aufgebaut sind.

*Adresse 0x1c - 0x1f*

Dateigröße in Bytes.

Sie sehen: in einem Directory-Sektor stecken eine ganze Menge Informationen, die nicht auf dem üblichen Weg interpretiert werden können. Deshalb wird der Diskmonitor für die Ausgabe eines Directory-Eintrags für jedes Teilfeld eine spezialisierte Sub-Funktion anstoßen, die weiß, wie das entsprechende Feld aufzufassen ist. Aber ehe Sie sich dem Listing des Diskmonitors zuwenden, sind noch einige Präliminarien nötig.

## 8.3 Zugriff auf Systemvariablen

Einige Seiten zuvor habe ich Ihnen eine Methode angekündigt, um auf die geschützten Systemvariablen zugreifen zu können. Dieses Versprechen will ich nun einlösen. Im Diskmonitor habe ich für den Zugriff auf Systemvariable keine Verwendung; deshalb steht das folgende Beispielprogramm alleine da - gleichsam zum Warmwerden.

Von den vielen Systemvariablen des ATARI habe ich mir für das Beispiel eine ausgesucht, die Ihnen vielleicht nicht sonderlich nützlich erscheinen mag, die mir jedoch gut in mein didaktisches Konzept paßt. Zeigt sie doch, daß rekursive Strukturen, speziell über Zeiger verknüpfte Listenstrukturen, keinesfalls eine exotische Angelegenheit für ein paar Spinnerte sind, sondern daß sie aus der modernen Informatik nicht mehr wegzudenken sind.

Beim Studium der Systemunterlagen des ATARI stellt man nämlich fest, daß TOS verhältnismäßig oft von Listenstrukturen Gebrauch macht. Eine dieser Strukturen, der sogenannte "Buffer Control Block" oder BCB, wird zur Pufferung von Ein-/Ausgabedaten benutzt. Das folgende Beispielprogramm geht ihr nach.

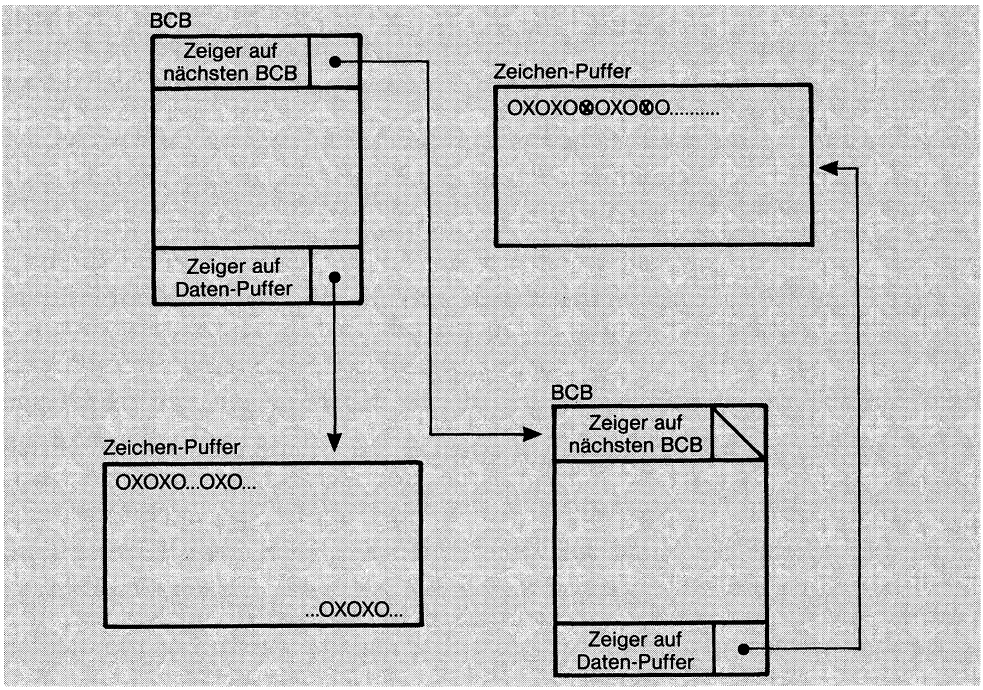



**Abb. 8.4:** *Struktur der Diskettenpuffer*

Macht der ATARI Disketten-Ein-/Ausgaben, dann werden die zu übertragenden Daten im Speicher gepuffert, um die Anzahl der Disketten-Zugriffe gering zu halten. Die BCB's sagen dem System unter anderem, an welchen Stellen im Speicher es die zugehörigen Puffer finden kann und welche Daten darin gespeichert sind. In der Abbildung 8.4 finden Sie eine Grafik der Struktur, die die BCB's haben.

An den Systemadressen 0x4B2 und 0x4B6 finden sich zwei Zeiger auf solche BCB-Strukturen. Jeder BCB zeigt unter anderem auf seinen Nachfolger und auf einen Puffer, der die Daten enthält. Diese Puffer als Liste zu organisieren hat unter anderem den Vorteil, daß das System sich nach Bedarf weitere Puffer (mit zugehörigen BCB's) irgendwo im Speicher besorgen kann, wo es gerade Platz findet.

Im Beispielprogramm 8.1 sehen Sie, wie man in C absolute Adressen der Maschine manipulieren kann. Mit der Deklaration

```
struct BCB *zeiger;
```

wird zu Beginn von *main* ein Pointer auf eine BCB-Struktur vereinbart. Diesen Pointer setzt das Programm gleich zu Beginn auf die Adresse 0x4B2, indem ihm dieser konstante Wert zugewiesen wird. Der Wert an der Adresse 0x4B2 ist seinerseits ein Pointer, der auf den Anfang der ersten BCB-Liste zeigt. Wenn man ihn dereferenziert, kann man also die in der Struktur gespeicherten Daten ansehen.

Nun ist das Dereferenzieren des Pointers an der Adresse 0x4B2 nicht ganz so einfach, weil er sich nämlich im geschützten Teil des Speichers befindet. Die Anweisungsfolge:

```
zeiger = 0x4B2;
zeiger = *zeiger;
```

würde zu einem Systemfehler (einem sogenannten Bus-Fehler) führen, weil mit **zeiger* einer geschützten Systemadresse nachgegangen wird. Ehe man das tun kann, muß man also erst den Supervisor-Modus betreten, wozu es eine eigene Funktion im BDOS gibt.

Im Hauptprogramm wird daher der Anfang der ersten BCB-Liste im Supervisor-Modus bestimmt. Ich gehe davon aus, daß diese Adresse (also die BCB-Liste selbst und die Puffer, auf die die BCB's verweisen) nicht im geschützten Adressbereich liegt, so daß es anschließend möglich ist, den Supervisor-Modus wieder zu verlassen. Bei meinen Testläufen mit diesem Programm hat diese Annahme bisher nie zu Problemen geführt. Sollte aber das Programm bei Ihnen abstürzen (und dabei 2 kleine Bömbchen auf den Bildschirm zeichnen; dies ist das Kennzeichen für einen Bus-Fehler), so

müssen Sie eventuell die gesamte *main*-Funktion im Supervisor-Modus ablaufen lassen.

Das Programm gibt jeden in der Liste eingehängten Datenpuffer in hexadezimaler Darstellung auf dem Bildschirm aus. Es tut dies unter Zuhilfenahme einer Technik, von der auch der Diskmonitor ausgiebigen Gebrauch macht: mit direkter Cursor-Steuerung.

Der Bildschirm des ATARI ist nämlich in begrenztem Umfang intelligent, d.h. er kann eine Anzahl von Befehlen ausführen. Befehle, die der Bildschirm versteht, sind zum Beispiel "Bildschirm löschen", "Zeile ab der aktuellen Cursor-Position bis zum Ende löschen", "Zeile einfügen" oder "Cursor an eine bestimmte Stelle des Bildschirms setzten". Das ist ganz praktisch und in C sehr leicht zu haben. Man erteilt die Befehle an den Bildschirm nämlich durch die übliche Zeichen-Ausgabe, z.B. mit *putchar*. Allerdings muß man kenntlichmachen, daß es sich bei den ausgegebenen Zeichen um Kommandos an den Bildschirm handelt. Dies besorgt der bereits von *printf* geläufige Escape-Mechanismus, nur daß das Escape-Zeichen für den Bildschirm nicht der umgekehrte Schrägstrich von C ist, sondern das ASCII-Zeichen mit dem Hexadezimalcode 0x1B (das sinnigerweise den Namen *Escape* trägt!). Es ist somit sehr einfach, sich die Bildschirm-Befehle als C-Makros zu definieren. das Programm 8.1 zeigt zu Beginn, wie dies gemacht wird.

Da der Inhalt der Datenpuffer nicht unbedingt lesbar sein muß, habe ich mich für hexadezimale Ausgabe des Puffers entschieden. Um einen Datenpuffer hexadezimal auszugeben, positioniert die Funktion *hex_out* den Cursor erst einmal in Zeile 4 und Spalte 8 des Bildschirms (mit dem Bildschirm-Kommando GOTO, das als C-Makro definiert ist) und beginnt dann, einzelne Zeichen des Puffers hexadezimal darzustellen. Um Ihnen das Zurückblättern zu ersparen, habe ich die Funktion zur hexadezimalen Ausgabe eines Zeichens hier noch einmal wiederholt.

Der Anfang des Beispielprogramms zeigt auch, wie die Einbindung von Betriebssystems-Routinen im ATARI-C über die Bibliotheks-Funktion *gemdos* erfolgt. Im nächsten Kapitel finden Sie zu diesem Thema genauere Informationen.

```
                                                /*************************/
extern long gemdos();                           /* Aufruf von GEMDOS-Funk- */
#define Super(stack) gemdos(0x20, stack)        /*  tion und Definition    */
                                                /*  von Escape-Sequenzen   */
#define ESC 0x1b                                /*  fuer Bildschirmsteue-  */
#define SCRN(a) putchar(ESC); putchar(a)        /*  rung.                  */
#define GOTO(x,y) SCRN('Y');putchar((char) x + 32); putchar((char) y + 32)
#define CLRSCRN SCRN('E')                       /*************************/

struct BCB
{  struct BCB *naechster;  /* Zeiger auf naechsten BCB                   */
   int drive;              /* Laufwerks-Nummer oder -1                   */
   int typ;                /* BCB-Typ (Daten oder Directory/FAT)         */
   int recno;              /* Record-Nummer auf Drive                    */
   int dirty;              /* Puffer veraendert?                         */
   long DMDptr;            /* Zeiger auf Media Deskriptor                */
   char *puffer;           /* Zeiger auf Puffer mit Daten                */
}

                                                /*************************/
info(ptr)                                       /* Informationen im Puf-  */
     struct BCB *ptr;                           /*  fer ausgeben.         */
                                                /*************************/
{                                               /*  Ueberschrift drucken. */
   printf("\nLaufwerk: %d\tTyp: %d\tRecord: %d\tDirty? %d\n",  /*          */
           ptr->drive, ptr->typ, ptr->recno, ptr->dirty);  /*             */
                                                /*                        */
   if (ptr->puffer)                             /* Puffer hexadezimal aus- */
      hex_out(ptr->puffer);                     /*  geben.                */
}                                               /*************************/

                                                /*************************/
                                                /* Fuer Ueberschrift: wird */
header(txt)                                     /*  erste oder zweite BCB- */
      char *txt;                                /*  Liste gerade ange-    */
                                                /*  zeigt?                */
{  printf("\n\t\t\t%s BCB-Liste", txt);         /*                        */
}                                               /*************************/

#define INFOZAHL 32                             /*************************/
#define ANFZEILE 4                              /* Bildschirmpositionen   */
#define LINKER_RAND 8                           /*  fuer hexadezimale     */
#define ZEILZAHL 16                             /*  Ausgabe.              */
#define ENDZEILE 20                             /*************************/

                                                /*************************/
hex_out(record)                                 /* Puffer mit absoluter   */
        register char *record;                  /*  Cursor-Positionierung */
                                                /*  hexadezimal ausgeben. */
{                                               /*************************/
   register int  zeile, zlr;                    /*                        */
                                                /*                        */
   for (zeile = ANFZEILE; zeile < ENDZEILE; ++zeile)  /*  Cursor zeilenwei- */
   {  GOTO(zeile,LINKER_RAND);                  /*  se weiterbewegen.     */
      for (zlr = INFOZAHL; zlr; --zlr)          /*                        */
         hex_dump(*record++);                   /*                        */
   }                                            /*                        */
}                                               /*************************/
```

```
#define MASKE 0x0f

hex_dump(h)
        register char h;

{
   register unsigned int byte;

   byte = (h >> 4) & MASKE;
   putchar(byte + (byte > 9 ? 'W' : '0'));
   byte = h & MASKE;
   putchar(byte + (byte > 9 ? 'W' : '0'));
}
                                        /*************************/
main()                                  /* Systemvariablen nach- */
                                        /*  spueren.             */
{  struct BCB  *zeiger;                 /*************************/
   long save_ssp;                       /*                       */
                                        /*                       */
                                        /* Supervisor-Modus betre- */
   save_ssp =  Super((long) 0);         /*  ten; alten Stack auf- */
                                        /*  heben.               */
   zeiger =  0x4b2;                     /* An dieser Adresse ist */
                                        /*  der erste Pointer;   */
   zeiger = *zeiger;                    /*  durch dereferenzieren */
                                        /*  gelangt man an den   */
                                        /*  Anfang der BCB-Liste. */
   Super(save_ssp);                     /* Supervisormodus wieder */
                                        /*  verlassen.           */
   do                                   /*                       */
   {  CLRSCRN;                          /* Erste BCB-Liste ausge- */
      header("Erste");                  /*  ben. Zuvor Bildschirm */
      info(zeiger);                     /*  loeschen (CLRSCRN).  */
      printf("\nWeiter mit beliebiger Taste...");  /*           */
      getchar();                        /*                       */
   }                                    /*                       */
   while (zeiger = zeiger->naechster);  /*                       */
                                        /*                       */
   save_ssp = Super((long) 0);          /* Zeiger auf Anfang der */
   zeiger = 0x4b6;                      /*  zweiten BCB-Liste    */
   zeiger = *zeiger;                    /*  setzen (im Supervi-  */
   Super(save_ssp);                     /*  sormodus).           */
                                        /*                       */
   do                                   /*                       */
   {  CLRSCRN;                          /* Zweite Liste ausgeben. */
      header("Zweite");                 /*                       */
      info(zeiger);                     /*                       */
      printf("\nWeiter mit beliebiger Taste...");  /*           */
      getchar();                        /*                       */
   }                                    /*                       */
   while (zeiger = zeiger->naechster);  /*                       */
                                        /*                       */
   CLRSCRN;                             /* Bildschirm loeschen und */
                                        /*  fertig.              */
}                                       /*************************/
```

**Prog. 8.1:** *Zugriff auf Systemvariable*

# 9 Der Diskmonitor

## 9.1 Die Bedienung des Programms

Um zu verstehen, wie ein Programm funktioniert, ist es wichtig zu wissen, wie es sich verhält. Das Verhalten des Programms wird nachhaltig von seinem Zweck diktiert. Der Zweck eines Diskettenmonitors ist schnell gesagt: mit ihm soll es möglich sein, jeden Sektor auf einer Diskette anzusehen und - falls gewünscht - zu verändern.

Schon die Beschreibung des Zweckes macht klar, daß sich diese Aufgabe nicht ausschließlich mit GEMDOS-Funktionen erledigen läßt; denn GEMDOS denkt in Dateien. Diskettensektoren, die keiner Datei angehören (entweder, weil sie noch unbenutzt sind oder - was interessanter ist - weil sie einer gelöschten und damit aus dem Bewußtsein von GEMDOS gestrichenen Datei gehören) sind auf diesem Wege völlig unzugänglich. Das Kernstück des Editors wird deshalb vom BIOS zu erledigen sein; aber sehen wir weiter.

Aus der Aufgabenstellung folgt, daß ein Diskettenmonitor dem Benutzer erlauben muß, den Inhalt eines Sektors zu analysieren und ihn zu ändern. Da kommt schon das erste Problem ins Spiel. Diskettensektoren enthalten Bytes. Diese Bytes können - wenn der Sektor nicht leer ist - Teile einer Textdatei sein, sie können jedoch auch einem Programm angehören. Textdateien enthalten fast ausschließlich abdruckbare Zeichen, also Klartext, weswegen es möglich sein sollte, einen Textsektor auch im Klartext zu betrachten. Mit anderen Worten: der Editor muß Sektoren als ASCII-Text ausgeben und bearbeiten können.

Die Bytes, aus denen die Programme bestehen, beschränken sich nicht auf die abdruckbaren Zeichen (siehe die ASCII-Tabelle im Anhang). Vielmehr ist jedes der 255 rechnerisch möglichen Bitmuster in einem Byte möglich; da hat eine ASCII-Anzeige nicht viel Sinn. Deswegen muß der Editor auch in der Lage sein, einen Programmsektor in hexadezimaler Darstellung anzuzeigen.

Wie Sie jetzt wissen, gibt es eine weitere Art von Sektoren, die über eine bestimmte Struktur verfügen: die Directory-Einträge. Da das Speicherungsformat von Directory-Einträgen etwas seltsam ist, sollte der Monitor noch über ein drittes Anzeigeformat für Directory-Einträge verfügen. Theoretisch könnte sich der Benutzer des Monitors diese Informationen auch aus

der hexadezimalen Darstellung zusammenklauben, doch dies ist schwierig. Nur Name und Extension einer Datei werden im Directory als Klartext gespeichert; alle anderen Informationen müssen interpretiert werden, wobei das besonders bei den Bitfeldern für Datum und Uhrzeit etwas mühselig ist. Hinzu kommt, daß wegen des im letzten Kapitel bereits erwähnten Intel-Speicherformats alle über Bytelänge hinausgehenden Informationen erst umgedreht werden müssen; das soll besser der Computer besorgen!

Dieses Directory-Format ist dasjenige, in dem sich der Monitor beim Aufruf meldet (vgl. Abbildung 9.1). Er zeigt dem Benutzer stets als erstes das Directory der Diskette an, von der er geladen wurde. In der Abbildung sehen Sie übrigens eine Diskette, auf der neben dem Monitor-Programm (DISKEDT.PRG) noch die Include-Dateien und C-Quellen enthalten sind, die den Monitor ausmachen.

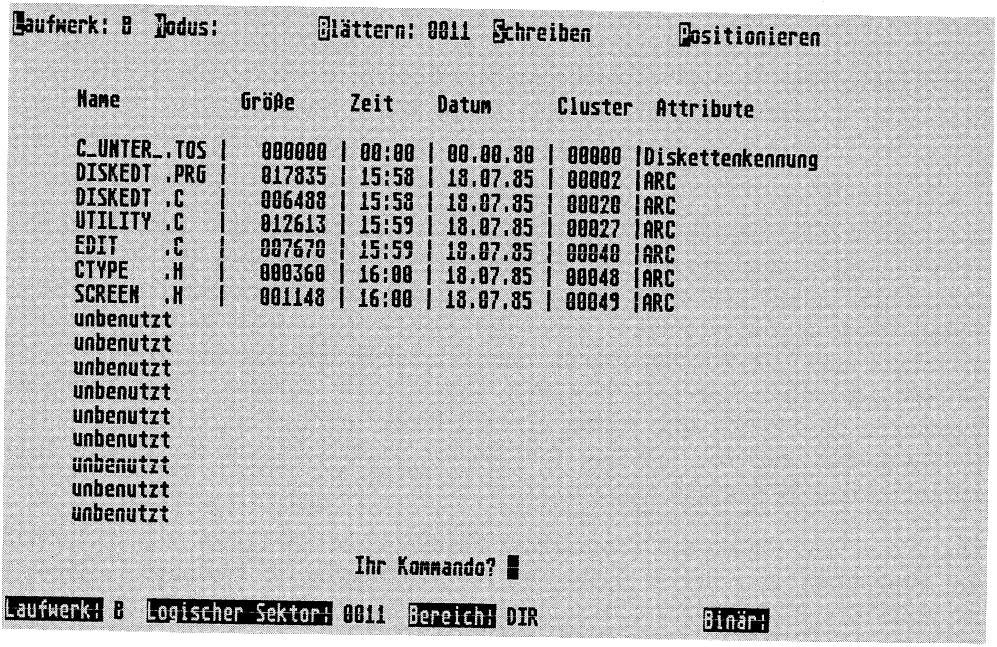



**Abb. 9.1:** *So meldet sich der Monitor*

In der Mitte des Bildschirms sieht man das Anzeige-Fenster, in dem die Sektor-Information im gewählten Format dargestellt wird. Die oberste Bildschirmzeile ist eine Menüzeile; sie sagt dem Benutzer, welche Kommandos er zur Verfügung hat. Er kann ein Kommando auswählen, indem er den (in der Menüzeile revers dargestellten) Anfangsbuchstaben des Kommandos tippt. Dazu muß er jedoch im Kommandomodus sein; dies er-

kennt er an der vorletzten Zeile des Bildschirms, die ihn mit der Meldung "Ihr Kommando?" zur Eingabe eines Kommando-Buchstabens auffordert. Drückt der Benutzer übrigens in dieser Situation auf die *Esc*-Taste, dann wird das Monitor-Programm beendet.

Die letzte Zeile des Bildschirms ist eine Statuszeile. Sie gibt dem Bediener unter anderem darüber Aufschluß, von welchem Laufwerk der angezeigte Sektor stammt, welche logische Nummer er hat und in welchen Bereich der Diskette er fällt. Weitere Informationen in der Statuszeile werden erst beim Edieren im ASCII- oder Hexadezimalmodus angezeigt.

Ein Beispiel für das Arbeiten im ASCII-Modus finden Sie in der Abbildung 9.2. Der in der Abbildung bearbeitet Sektor ist übrigens der Anfang der Quelldatei für den Diskmonitor - der Monitor betrachtet sich selbst! Wie Sie sehen können, sind jetzt in der Statuszeile zwei Einträge hinzugekommen; sie betreffen die sogenannte "Alternativanzeige".

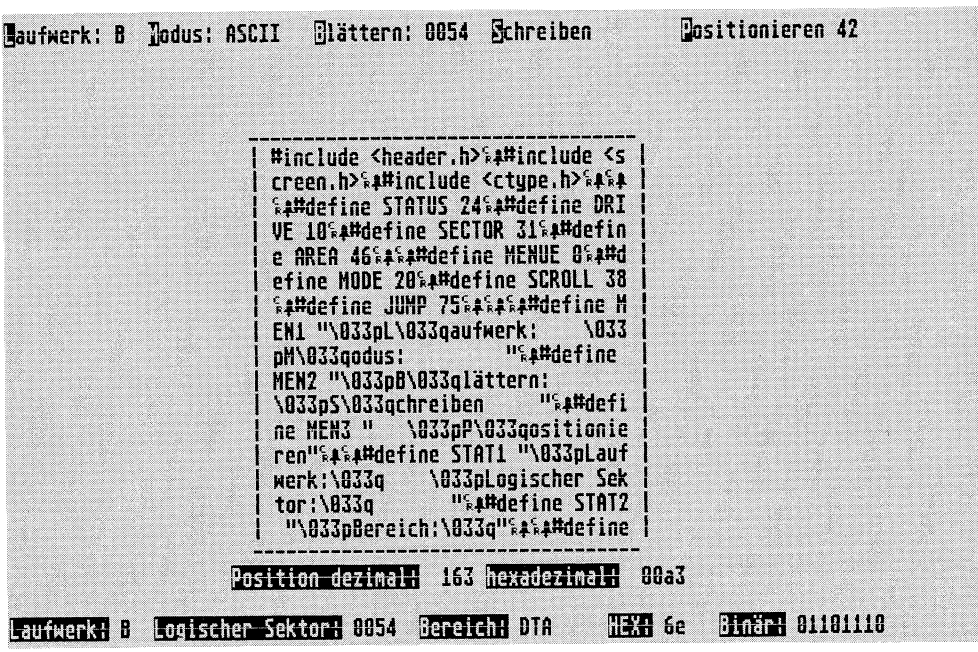



**Abb. 9.2:** *Sektor im ASCII-Format edieren*

Neben der ASCII- und der Hexadezimaldarstellung spielen in der EDV ja noch die Binärdarstellung und manchmal auch die Dezimaldarstellung eine Rolle. Gerade wenn man sich einen Sektor hexadezimal anzeigen läßt und man nicht so besonders fit im Hexadezimalsystem ist, kann es nützlich sein,

zusätzlich noch den Binär- und Dezimalwert eines Zeichens zu erfahren. Deshalb weist der Monitor folgende Bequemlichkeit auf: ein Sektor wird in einem gewählten Anzeigeformat auf dem Bildschirm dargestellt (im Directory-Format, hexadezimal oder im ASCII-Format); zusätzlich dazu wird das Zeichen, auf dem sich der Cursor gerade befindet, in der Statuszeile in Binärdarstellung angezeigt. Außerdem gibt es zu den Anzeigemodi "Hexadezimal" und "ASCII" jeweils einen Alternativmodus (für ASCII ist dies hexadezimal, für hexadezimal ist es dezimal) in dem das Zeichen unter dem Cursor ebenfalls in der Statuszeile angezeigt wird.

In der Abbildung 9.2 arbeitet der Editor im ASCII-Modus; die Alternativanzeige ist in diesem Modus die Hexadezimal-Darstellung. Daneben sehen Sie in der Statuszeile auch noch den Binärwert des aktuellen Zeichens. Welches das aktuelle Zeichen ist, können Sie der Abbildung ebenfalls entnehmen: der Cursor (der in der Abbildung leider nicht sichtbar ist), befindet sich auf dem 163ten Byte des Sektors; wer es ganz genau wissen will, der erfährt dies auch noch hexadezimal (vorletzte Bildschirmzeile).

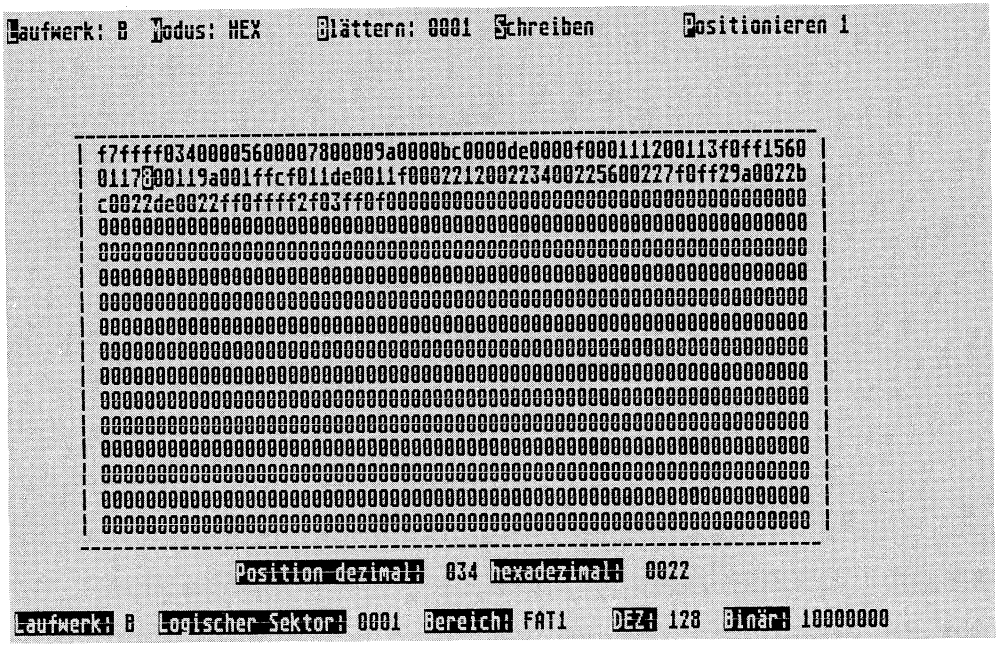



**Abb. 9.3:** *Sektor im Hex-Format edieren*

Diese Positionsangaben sind besonders im Hex-Modus nützlich, für den die Abbildung 9.3 ein Beispiel bietet. Wie Sie sehen können, ist die Alternativdarstellung in diesem Modus die dezimale Schreibweise. Der (in dieser Ab-

bildung sichtbare; der Kerl blinkt eben!) Cursor befindet sich im Byte Nr. 34 des Sektors. Bei der Arbeit im Hex-Modus ist noch zu beachten, daß ein Byte in Hexadezimal-Schreibweise zwei Ziffern benötigt.

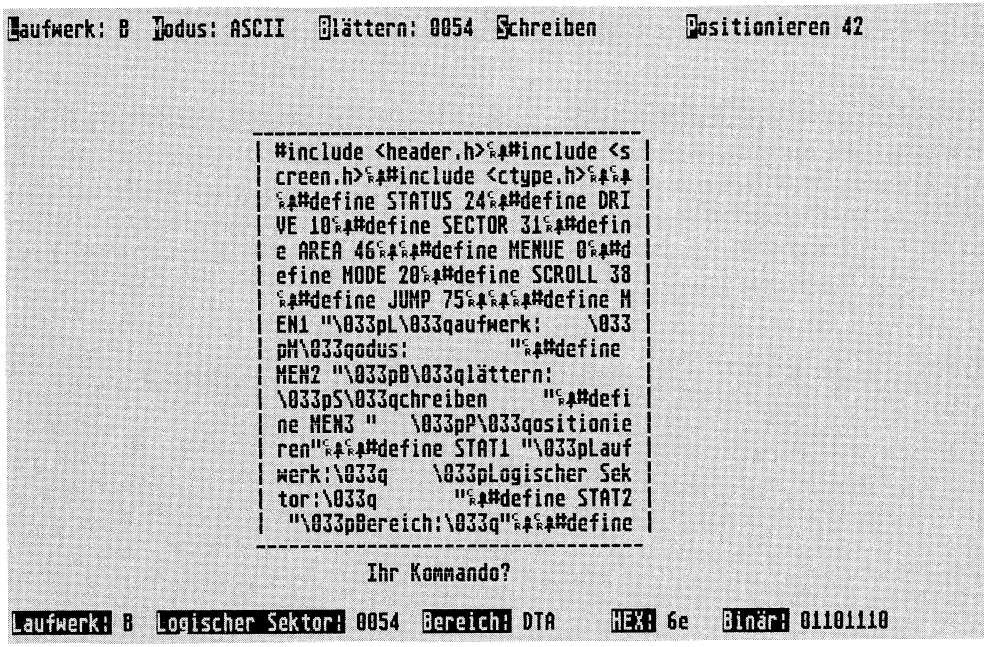



**Abb. 9.4:** *Der Editor im Kommando-Modus*

Wenn Sie einen Sektor bearbeiten, dann können Sie den Cursor mit den Pfeiltasten der ATARI-Tastatur beliebig herumbewegen (probieren Sie mal aus, was am Zeilen-Anfang und -Ende passiert), Sie können ihn mit der *Home*-Taste in die linke obere Ecke bewegen und Sie können den Sektor-Inhalt verändern, indem Sie den Cursor an die gewünschte Stelle bewegen und dort die entsprechenden Zeichen tippen. Allerdings können Sie im Hex-Modus nur Hexadezimal-Ziffern (Ziffern 0 bis 9 und Buchstaben 'A' mit 'F') eingeben; andere Zeichen akzeptiert der Monitor in diesem Modus nicht. Im ASCII-Modus können alle auf der Tastatur vorhandenen Zeichen eingegeben werden. Der Directory-Modus erlaubt keine Änderungen; man kann ihn nur zum Analysieren eines Sektors benutzen. Haben Sie einen Sektor geändert und wollen wieder den Urzustand vor der Änderung herstellen, dann können Sie dies durch Betätigen der Taste *Undo* erreichen. Schließlich erlaubt Ihnen die Taste *Help*, den Sektor zu löschen, ihn mit binären Nullen vollzuschreiben.

Um einen anderen Sektor zu bearbeiten oder den Anzeige-Modus bzw. das Laufwerk zu wechseln, müssen Sie erst einmal in den Kommando-Modus kommen. Dies geschieht ganz einfach durch Drücken der Taste *Esc*. Daraufhin verschwindet die Positionsanzeige aus der vorletzten Bildschirmzeile und es erscheint stattdessen eine Eingabeanforderung. Die Abbildung 9.4 zeigt dies für den Fall, daß sich der Editor gerade im ASCII-Modus befindet.

Kommandos werden ausgewählt, indem man den Anfangsbuchstaben des Kommandos eingibt; welche Kommandos verfügbar sind, kann der Benutzer aus der Menüzeile erfahren. Um etwa auf eine beliebige Stelle auf der Diskette zu springen, benutzt man das Positioniere-Kommando. Die Abbildung 9.5 zeigt, wie der Monitor auf die Auswahl dieses Kommandos reagiert: mit einem Hilfstext.


**Abb. 9.5:** *Das Positioniere-Kommando*

Der Cursor steht in dem Eingabefeld hinter dem Kommando in der Menüzeile und der Benutzer kann eine Sektornummer eingeben. Natürlich muß der Monitor verhindern, daß der Benutzer auf einen nicht existierenden Sektor zu positionieren versucht. Will man sich lediglich um einen Sektor auf der Diskette vorwärts- oder zurückbewegen, dann ist es einfacher, mit dem Blättere-Kommando zu arbeiten. Hier kann man sich einfach durch

Betätigen der "+" oder "-"-Taste in Einerschritten durch die Sektoren bewegen. Auch das Vergrößern der Schrittweite ist möglich: dazu muß der Benutzer lediglich die *Shift*-Taste zusammen mit der "+"- oder "-"-Taste betätigen. Die Schrittweite ändert sich zur nächsten Zweierpotenz hin (also 1, 2, 4, 8, 16 usw.), nimmt aber niemals einen Wert an, mit dem man über den Bereich der vorhandenen Sektoren hinausblättern könnte.

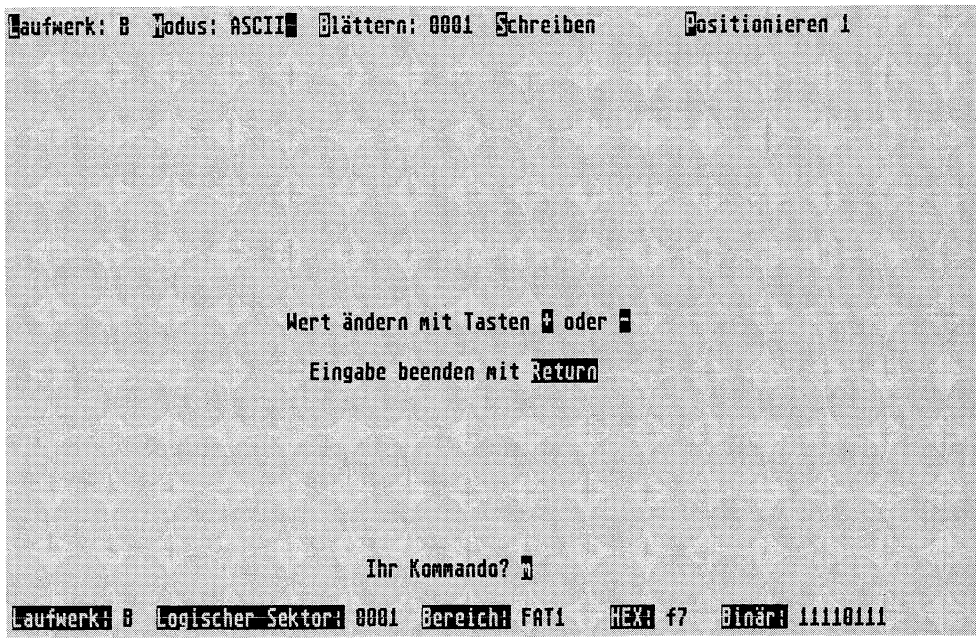



**Abb. 9.6:** *Anzeige-Modus ändern*

Nach dem gleichen Schema funktioniert die Änderung des aktuellen Laufwerks und des Anzeigemodus. Für die Änderung des Anzeigemodus finden Sie ein Beispiel in der Abbildung 9.6.

Der Hauptnutzen des Monitors besteht jedoch darin, beliebige Änderungen des Disketten-Inhalts vornehmen zu können. Dazu müssen Sie in einem Sektor durch Überschreiben den Inhalt modifizieren und dann im Kommando-Modus (den Sie mit *Esc* erreichen) das Schreibe-Kommando anwählen. Die Funktion bestätigt Ihnen, daß sie etwas getan hat, indem sie die Nummer des geschriebenen Sektors anzeigt. Dies sehen Sie in der Abbildung 9.7.

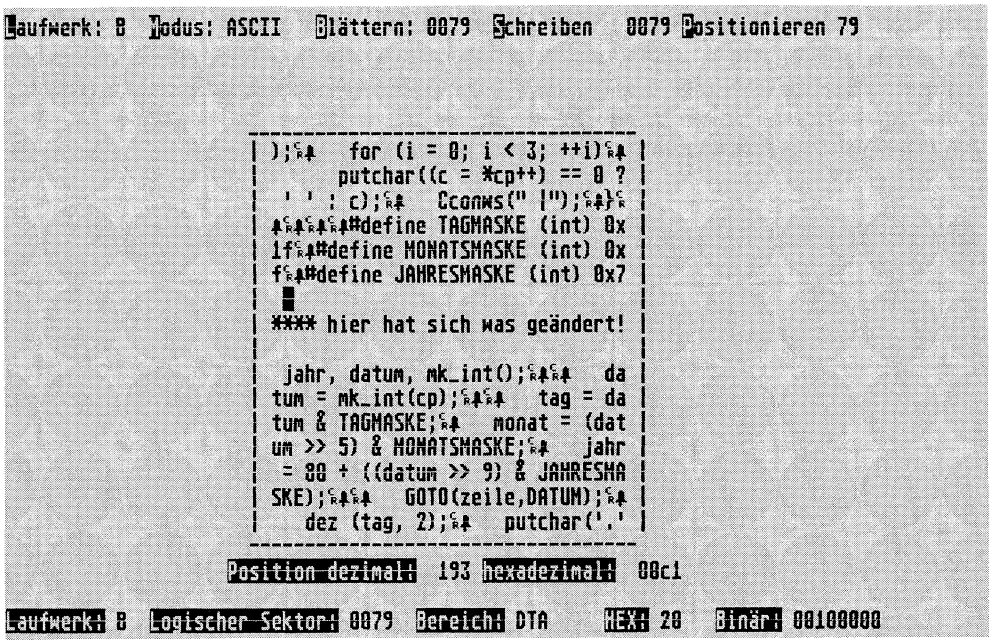



**Abb. 9.7:** *Geänderten Sektor auf Diskette schreiben*

## 9.2 Das Programm

Das Programm setzt sich aus drei großen Modulen zusammen. Das Hauptmodul DISKEDT realisiert die Benutzeroberfläche: es ist für die Interpretation und Ausführung der Benutzer-Kommandos verantwortlich, wobei die meisten Kommandos interne Parameter setzen (Z.B. die Nummer des angezeigten Sektors, den Anzeigemodus, das aktuelle Laufwerk usw.).

Das zweite Modul trägt den Namen RECEDIT und enthält denjenigen Teil des Programms, der die Bearbeitung eines Sektors im ASCII-Modus oder im Hex-Modus auf dem Bildschirm erlaubt.

Im dritten Modul mit dem Namen UTILITY sind einige Hilfsfunktionen enthalten, die von den anderen Moduln benötigt werden. Darunter finden sich auch alle Funktionen, die zur Ausgabe eines Sektors im Directory-Format (vgl. Abbildung 9.1) benötigt werden. Ich habe versucht, den Diskmonitor so weit wie möglich unabhängig von der Standard-Bibliothek zu machen: alle verwendeten Funktionen werden selbst definiert oder werden auf Leistungen des Betriebssystems zurückgeführt. Wenn Ihr Compiler also einigermaßen standardisiert ist und Ihnen Zugriff auf GEMDOS und BIOS erlaubt, dann sollten Sie mit dem Programm keine Schwierigkeiten haben.

Alle drei Module sind ausführlich kommentiert, so daß auf Details des Programms nicht mehr eingegangen werden muß. Ich werde mich daher lediglich auf die Darstellung der allgemeinen Programmlogik und die wichtigsten Fein- und Besonderheiten beschränken.

Das Modul DISKEDT ist dafür zuständig, die Kommandos auszuführen, die der Benutzer in der Menüzeile vorfindet. Außerdem muß es den Bildschirm aufbauen, d.h. Menü- und Statuszeile (soweit zu Programmbeginn möglich) auf einen zuvor gelöschten und somit gesäuberten Bildschirm schreiben. Das Modul arbeitet mit fünf globalen Variablen, in denen es sich den aktuellen Editor-Modus (ASCII, Hex oder Directory) merkt, die Nummer des Sektors, den der Benutzer bearbeiten will, die Nummer des vom Benutzer gewählten Laufwerks und die Nummer des Disketten-Bereichs, in dem sich der aktuelle Sektor befindet. Eine weitere globale Variable enthält den gerade bearbeiteten Disketten-Sektor. Wie dieser von der Diskette ins Programm kommt, werden Sie gleich erfahren.

Die Hauptfunktion *main* zaubert zuerst alle nötigen Beschriftungen auf den Bildschirm und initialisiert dann die Diskette. Dazu wird die Funktion *drv_get* aus dem Modul UTILITY benutzt, die als Wert die Nummer des aktuellen Laufwerks zurückgibt, welche auch gleich (mit *drv_out*) angezeigt wird.

Der Monitor startet mit der Anzeige des Inhaltsverzeichnisses der aktuellen Diskette. Deshalb erfrägt das Programm die Sektornummer des Directory über eine UTILITY-Funktion und - jetzt kommt's: liest den betreffenden Diskettensektor in den internen Sektor-Puffer.

Wer jetzt glaubt, daß dazu jede Menge Programmieraufwand nötig wäre, der sieht sich angenehm überrascht: die BIOS-Funktion *Rwabs* (vgl. den Anhang) erledigt nämlich alle nötigen Aufgaben. Mir ist kein anderes Betriebssystem bekannt, in dem der direkte Zugriff auf die Diskette ähnlich einfach zu erreichen wäre!

Nachdem der Directory-Sektor eingelesen ist, wird er in lesbarer Form angezeigt; dies übernimmt die UTILITY-Funktion *dir_out*, die als Argument den Puffer mit den Daten und die Bildschirmzeile erhält, an der sie mit der Anzeige beginnen soll.

Dann beginnt etwas, was Ihnen in diesem Programm noch öfter begegnen wird: eine Endlosschleife (FOREVER), hinter der sich ein Kommandointerpreter verbirgt. In der Endlosschleife wird die Funktion *interpret* gerufen, die Benutzerkommandos empfängt und ihre Ausführung veranlaßt. Sie teilt der rufenden Stelle - also *main* lediglich mit, ob der Benutzer das Programm verlassen oder den Modus ändern will.

Die interessanten Einzelheiten passieren also in *interpret*, das deshalb genauer unter die Lupe genommen werden soll. Auch diese Funktion besteht im wesentlichen aus einer Endlosschleife, die folgende Struktur hat: Zeichen von der Konsole lesen; überprüfen, ob es sich um einen zulässigen Kommando-Buchstaben handelt und falls ja, das Kommando ausführen; ansonsten erneuter Schleifendurchlauf.

Von den verfügbaren Kommandos sind drei Positionier-Kommandos: *L* ändert das Laufwerk, *B* blättert um einen oder mehrere Sektoren weiter nach vorne oder hinten und *P* erlaubt das direkte Positionieren auf der Diskette. Diese Kommandos werden in *interpret* durch Aufruf einer eigenen Funktion pro Kommando ausgeführt. Meist zieht dies die Änderung interner Parameter nach sich. Nach Ausführung des Kommandos muß stets der Sektor-Puffer auf den neuesten Stand gebracht werden, es muß also wieder mit *Rwabs* absolut von der Diskette gelesen werden.

Die Funktion *Rwabs* kann aber auch Schreiben; dies demonstriert das Kommando *S*. Alle zu seiner Ausführung nötigen Schritte sind in der Funktion *interpret* enthalten: zuerst wird der aktuelle Inhalt des Puffers *Sectbuff* geschrieben (kenntlich an der Eins als erstem Parameter), dann wird an der entsprechenden Position der Menüzeile die Nummer des geschriebenen Sektors angezeigt (*dez_dump* stammt aus UTILITY) und

schließlich wird der rufenden Stelle (also *main*) mitgeteilt, daß sie weitermachen - und das bedeutet, erneut den Kommandointerpreter aufrufen - soll.

Etwas komplizierter ist die Verarbeitung der *B*-Option. Das eigentliche Blättern besorgt die Funktion *ch_sector* (Abkürzung für *change sector*), die für das Setzen der internen Parameter zuständig ist und der ich mich gleich zuwenden werde. Nach Ausführung von *ch_sector* steht jedenfalls fest, welchen Disk-Sektor der Benutzer bearbeiten möchte; dieser wird mit *Rwabs* in den internen Puffer gebracht, wo er dann - im zuletzt gewählten Modus, also ASCII, Hex oder Directory - bearbeitet werden kann.

Nun jedoch zu *ch_sector* selbst. Hier ist - hauptsächlich aus Gründen der Verspieltheit des Programmierers - ein zusätzliches Feature eingebaut. Durch Betätigen der "+"- oder "-"-Taste kann man auf der Diskette vorwärts- bzw. rückwärtsblättern. Drückt man jedoch zugleich die *Shift*-Taste, dann ändert sich dadurch die Schrittweite, mit der geblättert wird. Mal sehen, wie man das programmtechnisch realisiert.

Die Höflichkeit gebietet es, dem Benutzer erst mal einen passenden Hilfstext zu präsentieren (Funktion *help_2*). Dann aber kommt schon wieder eine Endlosschleife: das hat den Effekt, daß der Benutzer für schnelles Blättern auch den Finger einfach auf der "+"- oder "-"-Taste liegen lassen kann. Nach jedem Blättern wird die Anzeige in der Statuszeile aktualisiert; dies besorgt der Code zu Beginn der Endlosschleife. Er schaltet auch den Cursor ein (dies ist eine reine Vorsichtsmaßnahme; vermutlich ist er bei Betreten der Funktion ohnehin schon eingeschaltet), damit der Benutzer eine optische Kontrolle hat, welche Funktion (Blättern) er gerade benutzt. Als nächstes wird eine Zeichen von der Tastatur ohne Echo gelesen; dies besorgt die GEMDOS-Funktion *Crawcin*. Nun kann es sein, daß der Benutzer auch eine Shift-Taste gedrückt hat. Mit der BIOS-Funktion *Getshift* kann dies ganz leicht überprüft werden, da diese einen Bit-Vektor liefert, in dem die ersten beiden Bits gesetzt sind, wenn eine der beiden Shift-Tasten betätigt worden war. Dies läßt sich ganz einfach durch Ausmaskieren feststellen.

Bei gedrückter Shift-Taste wird die Schrittweite für das Blättern (gespeichert in der Variablen *increment*) um die nächste Zweierpotenz verändert, wenn nicht dadurch über das physikalische Diskettenende hinausgeblättert wird. Das Diskende liefert die UTILITY-Funktion *data_end*; die nächste Zweierpotenz erreicht man im Binärsystem durch Shiften (ebenso wie man im Zehnersystem die nächste Zehnerpotenz durch Anhängen einer Null erreicht!).

Der Rest der Endlosschleife ist einfach: je nach gedrückter Taste wird der aktuelle Sektor hoch-oder niedergezählt. Gerät man dabei über den zulässigen Wertebereich, so wird einfach auf den Diskettenanfang oder das Ende positioniert. Drücken der Return-Taste durch den Benutzer schließt die Funktion ab.

Nach dem eben Gesagten sollte es Ihnen möglich sein, die anderen Kommandos von *interpret* und die damit assoziierten Funktionen zu verstehen. Ich kann mich also dem zweiten großen Modul zuwenden, das einen Sektor auf dem Bildschirm ediert.

Auch *edit*, die Hauptfunktion dieses Moduls, ist im wesentlichen wieder ein Kommando-Interpreter (und wenn Sie jetzt "Endlosschleife" denken, dann liegen Sie ganz richtig!) Er reagiert auf Tastendruck, um den Cursor über den Bildschirm wandern zu lassen oder Änderungen in einem Sektor entgegenzunehmen. Etwas kompliziert wird die Angelegenheit jedoch dadurch, daß der Editor zwei Modi - Hexadezimal und ASCII - beherrschen muß und daß ich es mir in den Kopf gesetzt habe, jedes Zeichen auch noch in einem Alternativ-Format in der Statuszeile anzuzeigen. Aber dadurch kommt die Würze in das Programm.

Und die besteht in den ominösen Funktions-Pointern, von denen es in *edit* gleich ganze vier gibt, wobei einer sogar noch global deklariert ist. Warum der Aufwand?

Nun, je nach Modus muß *edit* den Sektor in einem anderen Format anzeigen (braucht also unterschiedliche Funktionen zur Bildschirmdarstellung des Sektors), muß eine andere Alternativ-Anzeige benutzen, darf nur bestimmte Zeichen einfügen (im Hex-Modus nur die Hexadezimalziffern, im ASCII-Modus hingegen jedes von der Tastatur erzeugbare Zeichen) und muß sich beim Einfügen der Zeichen anders verhalten.

Ohne die Möglichkeit, mit Funktions-Pointern zu arbeiten, hätte ich vermutlich zwei Edierfunktionen geschrieben, eine für jeden Modus. Die hätten sich allerdings in weiten Strecken geglichen und das beleidigt das Schönheitsempfinden des Programmierers. Mit Funktions-Pointern aber ist es ganz einfach. Alle vom Edier-Modus abhängigen Eigenschaften werden von unterschiedlichen Funktionen realisiert; *edit* ruft jedoch keine dieser Funktionen direkt, sondern benutzt Funktions-Pointer, denen als Wert je nach Modus dann die passenden Funktionen zugewiesen werden. Dazu ein einfaches Beispiel: im ASCII-Modus soll die Alternativdarstellung hexadezimal, im Hex-Modus dezimal erfolgen. Deshalb benutzt *edit* einen Funktions-Pointer *Alternate* (Großschreibung, weil dies eine globale Variable ist) und weist zu Beginn an *Alternate* entweder *hex_d* zur hexadezimalen Ausgabe eines Bytes zu, oder *dez_d*, um das Byte in Dezimalschreibweise

darzustellen. Ebenso verhält es sich mit *show_fun*, der Funktion, die den zu bearbeitenden Sektor auf dem Bildschirm anzeigt. Im Hex-Modus ist dies *hex_out*, im ASCII-Modus ist es *asc_out*; beide Funktionen finden Sie im Modul UTILITY.

Im Hex-Modus darf der Benutzer nur Hexadezimalziffern eingeben; der Editor braucht also eine Funktion, die überprüft, ob Eingabezeichen legal sind; diese ist Wert des Funktionspointers *legal*, der im Hex-Modus die Funktion *is_hex* zugewiesen bekommt. Im ASCII-Modus sind alle Zeichen legal; deshalb hat *legal* in diesem Modus als Wert eine Funktion *true*, die stets den Wahrheitswert "wahr" zurückliefert.

Der Editor soll den Cursor über den Bildschirm wandern lassen, aber auch legale Eingabezeichen auf dem Bildschirm darstellen und sie in den Sektor-Puffer einfügen. Im ASCII-Modus ist dies kein Problem; im Hex-Modus bereitet es jedoch einige Schwierigkeiten, auf die ich noch zu sprechen kommen werde. Aus der Sicht von *edit* ist jedenfalls alles halb so schlimm: um das Einfügen von Zeichen in den Puffer und deren Darstellung am Bildschirm kümmert sich eine Funktion, die an den Pointer *insert* gebunden ist. Zu Beginn von *edit* wird in Abhängigkeit vom Modus eben einfach die passende Funktion hier eingehängt.

Was aber - außer Initialisieren - macht die Funktion *edit* sonst noch; was geschieht in der ominösen Endlosschleife? Diese ist wieder ein Kommandointerpreter nach bewährtem Strickmuster. Diesmal reicht es jedoch nicht aus, ein Zeichen von der Tastatur zu lesen. Der Benutzer kann nämlich auch Funktionstasten betätigen (die Pfeiltasten sowie *Undo* und *Home*) und diese liefern kein Zeichen. Wie kann man dann im Programm erkennen, ob sie gedrückt wurden?

Dies geht mit Hilfe des sogenannten Scancodes. Jeder Taste der ATARI-Tastatur ist eine eindeutige Nummer zugeordnet - eben der "Scancode" dieser Taste - und es gibt eine TOS-Funktion (*Bconin* aus dem BIOS mit Parameter 2), die bei Betätigen der Taste nicht nur das zugehörige Zeichen, sondern auch den Scancode zurückliefert. Die UTILITY-Funktion *scan* greift auf diese Betriebssystems-Funktion zurück, liefert als Wert das gelesene Zeichen (oder 0, falls eine Funktionstaste betätigt wurde) und schreibt in den Integer-Pointer, der ihr Argument ist, den Scancode der Taste. Nach Aufruf von *scan* steht also in *character* das Zeichen, falls der Benutzer eine Buchstaben-Taste betätigte, auf jeden Fall aber in der Integer *scancode* der Scancode der Taste. Die Scancodes der Funktionstasten sind als symbolische Konstanten zu Beginn des Programmes vereinbart. Somit ist es *edit* jetzt möglich, je nach gedrückter Taste die entsprechenden Kommandos zu geben.

Hat der Benutzer ein Zeichen eingegeben, so kann es entweder *Escape* sein, welches ein Verlassen des Editors bewirkt, oder es ist irgend ein darstellbares Zeichen. Ist es ein im aktuellen Modus legales Eingabezeichen, dann wird es mit der aktuellen Einfügefunktion in den Sektorpuffer eingefügt und am Bildschirm angezeigt. Sehen Sie sich einmal an, wie das - zu Beginn der FOREVER-Schleife - in *edit* gemacht wird: jetzt können Sie sehen, welche enormen Möglichkeiten im Konzept der Funktions-Pointer verborgen sind!

Hat der Benutzer kein Zeichen eingegeben, dann wird er wohl eine Funktionstaste betätigt haben; im folgenden *switch* werden deshalb die Scancodes ausgewertet. Bei den Funktionen zur Cursorbewegung besteht der gesamte Aufwand darin, die entsprechende Bewegungs-Funktion aufzurufen und dann einen erneuten Schleifendurchgang anzustoßen.

Bei der Funktion, mit der der alte Pufferinhalt wiederhergestellt werden kann (*case*-Marke *UNDO*), wird in *edit* etwas mehr Aufwand betrieben. Beim ersten Aufruf hat sich *edit* wohlweislich den Anfangszustand des Puffers an einem sicheren Ort (dem Array *savebuf*) gemerkt. Dieser Anfangszustand wird nun durch Kopieren wieder in den Arbeitspuffer übertragen, der neue Pufferinhalt mit der aktuellen Anzeigefunktion (Hex oder ASCII) angezeigt und dann der Cursor an den Sektor-Anfang gesetzt.

Vielleicht haben Sie sich schon gefragt, wie das Programm das Zusammenspiel zwischen der am Bildschirm angezeigten Information und dem Inhalt des Sektorpuffers organisiert. Denn alles, was der Benutzer auf dem Bildschirm sieht und alle Veränderungen, die er dort durch Bewegen des Cursors oder durch Überschreiben von alter Information vornimmt, muß seinen Niederschlag ja auch in den im Programm gespeicherten Daten finden.

Zu diesem Zweck unterhält der Editor einen Zeiger in den Sektor-Puffer, einen *char*-Pointer, der bezeichnenderweise den Namen *Cursor* trägt und der immer genau auf das Zeichen im Puffer zeigt, über dem sich der auf dem Bildschirm sichtbare blinkende Cursor gerade befindet. Jede Funktion, die den Bildschirm-Cursor verändert, ist dafür verantwortlich, daß auch der interne Puffer-Zeiger wieder auf den richtigen Stand gebracht wird.

Im ASCII-Modus ist dies kein großes Problem, da die Entsprechung zwischen Bildschirm-Anzeige und Pufferinhalt eindeutig ist: ein Zeichen auf dem Bildschirm entspricht in der ASCII-Anzeige einem Byte im Sektor-Puffer. Bewegt sich der Bildschirm-Cursor nach rechts, so reicht es aus, dem Sektor-Pointer mit *++Cursor* nachzuziehen.

Anders liegen die Verhältnisse im ASCII-Modus. Ein Byte des Puffers entspricht hier zwei Zeichen in der Bildschirmanzeige. Bildschirm-Cursor und

Sektor-Pointer können nun nicht mehr synchron bewegt werden. Um nur ein Beispiel zu geben: Bewegt sich der Bildschirm-Cursor nach rechts, so darf der Sektor-Pointer nur inkrementiert werden, wenn sich der Cursor im niederwertigen Halbbyte eines angezeigten Bytes befand. Bei den anderen Bewegungsvorgängen sind die Vorgänge ebenso kompliziert. Genaueres erfahren Sie, wenn Sie die Bewegungsfunktionen *up*, *left*, *right* und wie sie alle heißen eingehender studieren.

Auch das Eingeben von Zeichen zum Zweck der Veränderung eines Sektors hat so seine Tücken. Nicht im ASCII-Modus: da braucht man lediglich das eingelesene Zeichen an die Stelle im Sektorpuffer zu schreiben, auf die der Pointer *Cursor* zeigt. Was aber ist im Hex-Modus? Hier ändert ein Eingabezeichen ja stets nur ein Halbbyte und im Sektor-Puffer ist dazu einige Bitfummelei nötig, die auch noch unterschiedlich ausfällt, je nachdem, ob das höher- oder niederwertige Halbbyte geändert werden soll. Sehen Sie sich dazu die Funktion *hex_in* an.

Ein Wort noch zu den Bewegungsfunktionen. Sie sind für einen automatischen Zeilensprung ausgelegt. Ein Beispiel: will der Benutzer nach rechts (Funktion *right*), steht der Cursor jedoch bereits am Ende der Zeile, dann springt er in den Anfang der nächsten Zeile. Steht der Cursor am Ende des Sektors, dann springt er ganz an den Anfang des Sektors. Ähnlich verhalten sich die anderen Bewegungsfunktionen.

Das dritte große Modul trägt den Namen UTILITY und enthält vermischte Funktionen, die für andere Funktionen Hilfsdienste verrichten, aber auch den Teil des Editors, der die Ausgabe eines Directory-Sektors übernimmt. Um diesen Teil zu verstehen, müssen Sie sich nur die Darstellungen im Kapitel 8 vor Augen halten.

Noch eine Anmerkung zu den Listings, ehe Sie sich ins Vergnügen stürzen. Alle drei Module werden mit der Zeile *#include <header.h>* eingeleitet. In der Datei *header.h* stehen lediglich weitere, systemspezifische *include*-Direktiven für den Makro-Präprozessor, mit denen Dateien, die der Compiler benötigen, wie z.B. *stdio.h* mit in das Programm gezogen werden. Welche Dateien Sie mit *header* einschließen lassen, ist Systemabhängig; es empfiehlt sich jedoch, eine Datei mit aufzunehmen, die die C-Namen der Betriebssystems-Funktionen enthält, welche in der ATARI-Dokumentation und in diesem Buch verwendet werden (siehe die Anhänge zu GEMDOS und BIOS). Im Unterkapitel 9.5 sehen Sie, wie eine solche *include*-Datei (die bei mir den Namen *cnames.h* trägt) für das ATARI-C aussieht.

Die Datei *screen.h*, die vom UTILITY-Modul gebraucht wird, enthält C-Makros zur Bildschirmsteuerung. Ihren Inhalt finden Sie im Unterkapitel 9.6. In der Datei *ctype.h* befinden sich die üblichen, mehrfach im Buch

verwendeten Makros zur Klassifikation von Zeichen. Sie ist im Unterkapitel 9.7 noch einmal aufgeführt. Sollten Sie Fragen zu den GEMDOS- und BIOS-Funktionen haben, die über die im Anhang gegebenen Informationen hinausgehen, so empfehle ich Ihnen, dazu ein spezielles Systemhandbuch zu Rate zu ziehen.

Und jetzt bleibt mir nur noch, Ihnen viel Spaß mit den Listings zu wünschen!

## 9.3 Listing des Moduls DISKEDT

```
#include <header.h>
#include <screen.h>
#include <ctype.h>

/*                    MODUL: DISKEDT                                        */


/*****************************************************************************/
/*  Treiber für den Disketten-Editor. Dieses Modul realisiert hauptsächlich  */
/*  die Benutzeroberfläche des Editors. Es interpretiert Kommandos des Benut-*/
/*  zers, mit denen der Ediermodus geändert, das Laufwerk gewechselt und auf */
/*  beliebige Sektors der Diskette positioniert werden kann. Alle Kommandos  */
/*  können aus einem Menü ausgewählt werden; mit Ausnahme der direkten Posi- */
/*  tionierung werden Wertveränderungen durch Betätigen der Tasten "+" und   */
/*  "-" vorgenommen, wodurch eine Fehlbedienung ausgeschlossen ist.  Neben   */
/*  dem Interpretieren der Benutzerkommandos übernimmt das Modul auch die An-*/
/*  zeige diverser Statusinformationen, sofern diese nicht vom Sektor-Editor */
/*  (Moduls RECEDIT) geleistet wird.                                         */
/*****************************************************************************/

                                        /***************************/
#define STATUS 24                       /* Bildschirmpositionen    */
#define DRIVE 10                        /*  der Felder in der Menü-*/
#define SECTOR 31                       /*  und Statuszeile. Im    */
#define AREA 46                         /*  Feld AREA wird ange-   */
                                        /*  zeigt, in welchem Be-  */
#define MENUE 0                         /*  reich der Diskette     */
#define MODE 20                         /*  (System, Directory...) */
#define SCROLL 38                       /*  wir uns gerade befin-  */
#define JUMP 75                         /*  den.                   */
                                        /***************************/


/*****************************************************************************/
/*  Beschriftungen der Menüzeile (Zeile 1; vgl. Konstante MENUE) und der     */
/*  Statuszeile (Konstante STATUS). In die Beschriftungen sind die Escape-   */
/*  Sequenzen für reverse Darstellung einzelner Zeichen bereits eingearbei-  */
/*  tet: daher ihr seltsames Aussehen!.                                      */
/*****************************************************************************/

#define MEN1 "\033pL\033qaufwerk:    \033pM\033qodus:        "
#define MEN2 "\033pB\033qlättern:       \033pS\033qchreiben     "
#define MEN3 "   \033pP\033qositionieren"

#define STAT1 "\033pLaufwerk:\033q    \033pLogischer Sektor:\033q      "
#define STAT2 "\033pBereich:\033q"
                                        /***************************/
#define SYS  0                          /* 'Botschaften' von der   */
#define FAT1 1                          /*  Funktion "where_is",   */
#define FAT2 2                          /*  die anzeigen, in wel-  */
#define DIR  3                          /*  chem Bereich der Dis-  */
#define DTA  4                          /*  kette wir uns gerade   */
                                        /*  befinden.              */
                                        /***************************/
```

```
                                                /*************************/
                                                /* 'Botschaften' des Kom- */
#define END -1                                  /*  mandointerpreters, die */
#define CONTINUE 3                              /*  dem Treiber sagen, wel-*/
#define HEX 1                                   /*  chen Modus des Editors */
#define ASCII 0                                 /*  er wählen soll (oder ob*/
#define DIRECTORY 2                             /*  er aufhören soll).     */
                                                /*************************/

/*****************************************************************************/
/*  Globale Variablen. Einige Statusinformationen sowie der Puffer, der den  */
/*  gerade vom Editor bearbeiteten Disketten-Sektor enthält, werden zwecks   */
/*  leichterem Informationsaustausch zwischen Funktionen in globalen Variab- */
/*  len gehalten.                                                            */
/*************************************************                           */
                                                /*                       */
int Mode;                                       /* Statusparameter in glo- */
int Sector;                                     /*  balen Variablen: Edi-  */
int Drive;                                      /*  tormodus, aktueller    */
int Area;                                       /*  Sektor und Bereich.    */
                                                /* Puffer für den aktuellen*/
char Sectbuff[512];                             /*  Sektor.                */
                                                /*************************/


/*****************************************************************************/
/*  Das Hauptprogramm zeichnet die Beschriftungen auf den Bildschirm, posi-  */
/*  tioniert in das Directory des aktuellen Laufwerks, zeigt dieses an und   */
/*  ruft dann in einer Endlosschleife (FOREVER) den Kommandointerpreter, der */
/*  dafür zuständig ist, daß die internen Parameter den Wünschen des Benut-  */
/*  zers gemäß gesetzt werden. Das Hauptprogramm muß lediglich den Editor    */
/*  im richtigen Modus aufrufen oder die Anzeige eines Sektors im Directory- */
/*  Format veranlassen (Beachte: in diesem Format kann NICHT ediert werden,  */
/*  oder das Programm beenden. Was es zu tun hat, erfährt es über 'Botschaf- */
/*  ten vom Kommandointerpreter "interpret", der somit die Hauptarbeit über- */
/*  nimmt.                                                                   */
/*************************************************                           */
                                                /*                       */
main()                                          /*                       */
{                                               /*                       */
   int command;                                 /* Die 'Botschaft' des In- */
                                                /*  terpreters.          */
                                                /*                       */
   CLRSCRN;                                     /* Bildschirm löschen,    */
   menout();                                    /*  Beschriftung in Menü-  */
   statout();                                   /*  und Statuszeile anzei- */
                                                /*  gen.                 */
   drv_out(Drive = drv_get());                  /* Aktuelles Laufwerk an-  */
                                                /*  zeigen und zugleich    */
   Sector = directory();                        /*  initialisieren, Posi-  */
                                                /*  tion des Directory ge- */
   Rwabs (0, Sectbuff, 1, Sector, Drive);       /*  ben lassen und dieses  */
                                                /*  in den Editor-Puffer   */
   Area = DIR;                                  /*  lesen.                 */
   Mode = DIRECTORY;                            /* Anfangsparameter setzen.*/
                                                /*                       */
   update();                                    /* Bildschirmanzeige aktu- */
                                                /*  alisieren.           */
```

```
    dir_out (Sectbuff, 5);                          /* Sektor im Directory-For-*/
                                                    /*  mat anzeigen.          */
    FOREVER                                         /* Interpreterschleife.    */
    {                                               /*                         */
       command = interpret();                       /* Botschaft besorgen und  */
       switch (command)                             /*  darauf reagieren.      */
       {  case END:                                 /* Der Benutzer hat genug, */
             REV_OFF;                               /*  also Bildschirm be-    */
             CRSR_ON;                               /*  reinigen, alles in den */
             CLRSCRN;                               /*  Grundzustand versetzen */
             return;                                /*  (reine Vorsichtsmaß-   */
                                                    /*  nahme) und Ex und Hopp!*/
          case HEX:                                 /* Benutzer will Edit-Mo   */
          case ASCII:                               /*  dus ändern; das kann   */
             edit (Sectbuff, command);              /*  er haben ("command"    */
             continue;                              /*  enthält bereits rich-  */
                                                    /*  tigen Modus).          */
          case DIRECTORY:                           /* Benutzer will Anzeige   */
             dir_out (Sectbuff, 5);                 /*  im Directory-Format.   */
             continue;                              /*                         */
                                                    /*                         */
          case CONTINUE:                            /* Benutzer will einfach   */
          default:                                  /*  weitermachen...        */
             continue;                              /*                         */
       }                                            /*                         */
    }                                               /*                         */
}                                                   /*************************/

/*************************************************************************/
/*  Der Kommandointerpreter besorgt sich durch Anzeigen einer freundlichen  */
/*  Meldung ein Benutzerkommando (wobei nur im Menü enthaltene Kommandos ak- */
/*  zeptiert werden, sorgt dafür, daß die Kommandos ausgeführt werden, in-  */
/*  dem er an die entsprechenden Subfunktionen deligiert und teilt dann sei- */
/*  nem 'Vorgesetzten' -- dem Hauptprogramm -- mit, ob eine Modusänderung   */
/*  nötig ist oder ob der Benutzer die Lust verloren hat und aufhören will. */
/*************************************************/                        */
                                                  /*                       */
int interpret()                                   /*                       */
                                                  /*                       */
{  char comm_char,                                /* Der Kennbuchstabe des  */
   get_command();                                 /*  gewählten Kommandos   */
                                                  /*  und die Funktion, die */
                                                  /*  ihn liefert.          */
   FOREVER                                        /* Ausführung des Komman-  */
   {  comm_char = get_command();                  /*  dos veranlassen; falls */
                                                  /*  kein legaler Kommando- */
      if (comm_char == ESC)                       /*  buchstabe, fängt das   */
         return END;                              /*  ganze von Vorne an; da-*/
                                                  /*  her die Endlosschleife.*/
      clear_junk();                               /* Anzeigefläche säubern.  */
                                                  /*                         */
      switch (toupper(comm_char))                 /* Kommando-Buchstaben     */
      {                                           /*  normieren.             */
         case 'L':                                /* Aktuelles Laufwerk än-  */
            ch_drive();                           /*  dern, dort Sektor le-  */
            Rwabs (0, Sectbuff, 1, Sector, Drive);/*  sen und im aktuellen   */
            clear_junk();                         /*  Modus edieren.         */
            return Mode;                          /*                         */
```

```
                                                /*                          */
        case 'M':                               /* Editor-Modus ändern.     */
           ch_mode();                           /*                          */
           clear_junk();                        /*                          */
           return Mode;                         /*                          */
                                                /*                          */
        case 'B':                               /* 'Blättern' auf der Dis-  */
           ch_sector();                         /*  kette.                  */
           Rwabs (0, Sectbuff, 1, Sector, Drive);/*  Neuen Sektor lesen und */
           clear_junk();                        /*  edieren, nachdem An-    */
           return Mode;                         /*  zeige bereinigt wur-    */
                                                /*  de.                     */
        case 'S':                               /* Edierten Sektor auf      */
           Rwabs (1, Sectbuff, 1, Sector, Drive);/*  Diskette schreiben;    */
           GOTO (MENUE, 56);                    /*  zugleich die Nummer     */
           dez_dump ((long) Sector, 4);         /*  des geschriebenen Sek-  */
           return CONTINUE;                     /*  tors anzeigen.          */
                                                /*                          */
        case 'P':                               /* Auf neuen Sektor posi-   */
           set_sector();                        /*  tionieren, bestimmen,   */
           Area = where_is(Sector);             /*  in welchem Bereich der  */
           update();                            /*  Disk er sich befinden,  */
           Rwabs (0, Sectbuff, 1, Sector, Drive);/*  den Sektor lesen und   */
           clear_junk();                        /*  die Anzeige aktuali-    */
           return Mode;                         /*  sieren. Sektor im ak-   */
                                                /*  tuellen Modus edieren.  */
        default:                                /* Unbekanntes Kommando.    */
           continue;                            /*                          */
     }                                          /*                          */
  }                                             /*                          */
}                                               /**************************/


/*************************************************************************/
/*  Es folgen die eigentlichen Kommandos, die im Dialog mit dem Benutzer und */
/*  durch Anzeigen eines Hilfstextes interne Parameter setzten, wobei darauf */
/*  geachtet wird, daß keine unzulässigen Werte entstehen können.        */
/*************************************************************************/

                                                /***************************/
char *Modi[] =                                  /* Strings für die Anzei-  */
{  "ASCII",                                     /*  ge des aktuellen Edi-  */
   "HEX  ",                                     /*  tormodus.              */
   "DIR  ",                                     /*                         */
} ;                                             /***************************/

                                                /***************************/
ch_mode()                                       /* Editor-Modus ändern.    */
                                                /***************************/
{  register char command;                       /*                         */
                                                /* Hilfstext anzeigen; Be- */
   help_1();                                    /*  nutzer kann Modus      */
                                                /*  mit + oder - ändern    */
   FOREVER                                      /*  und mit Return bestä-  */
   {                                            /*  tigen.                 */
      GOTO (MENUE, MODE);                       /* Aktuellen Modus anzei-  */
      Cconws(Modi[Mode]);                       /*  gen.                   */
      CRSR_ON;                                  /*                         */
```

```
      command = Crawcin();
      switch (command)
      {  case '\r':
            return;

         case '+':
            if (Mode == 2)
               Mode = 0;
            else ++Mode;
            break;

         case '-':
            if (Mode == 0)
               Mode = 2;
            else --Mode;
            break;

         default:
            break;
      }
   }
}


ch_drive()

{  register char command;

   help_1();

   FOREVER
   {
      drv_out (Drive);
      CRSR_ON;
      command = Crawcin();
      switch (command)
      {  case '\r':
            drv_out(Drive);
            return;

         case '+':
            if (drv_set (Drive + 1))
               drv_out(++Drive);
            break;

         case '-':
            if (drv_set (Drive - 1))
               drv_out(--Drive);
            break;

         default:
            break;
      }
   }
}
```

```
/* Benutzerkommando ohne   */
/*  Echo lesen.            */
/* Benutzer will Auswahl   */
/*  beenden.               */
/*                         */
/* Benutzer geht 'nach     */
/*  oben'; also auf Über-  */
/*  lauf der Optionen      */
/*  prüfen.                */
/*                         */
/*                         */
/* Benutzer geht nach un-  */
/*  ten; auf Unterlauf der */
/*  Optionen prüfen.       */
/*                         */
/*                         */
/*                         */
/* Kommando nicht bekannt; */
/*  ignorieren.            */
/*                         */
/*                         */
/***************************/

/***************************/
/* Laufwerk wechseln.      */
/***************************/
/*                         */
/* Ändern geht mit +/-     */
/*                         */
/*                         */
/*                         */
/*                         */
/* Aktuelles Laufwerk an-  */
/*  zeigen und Benutzer-   */
/*  kommando holen.        */
/*                         */
/* Benutzer ist glücklich  */
/*  mit seiner Wahl...     */
/*                         */
/*                         */
/* Benutzer will nach oben.*/
/*  Geht das überhaupt?    */
/*  Falls ja, soll er      */
/*  seinen Willen haben,   */
/*  ansonsten geht's weiter*/
/*  in der Schleife.       */
/* Nach unten funktioniert */
/*  es genauso.            */
/*                         */
/*                         */
/* Unbekanntes Kommando    */
/*  ignorieren.            */
/*                         */
/*                         */
/***************************/
```

```
#define SHIFT (long) 0x03

ch_sector()

{  register char command;
   register int increment = 1;


   help_2();

   FOREVER
   {  Area = where_is (Sector);
      update();
      CRSR_ON;
      command = Crawcin();
      if (Getshift() & SHIFT)
      {  if (increment > data_end())
            increment = 1;
         else increment <<= 1;
      }
      switch (command)
      {  case '\r':
            return;

         case '+':
            if (Sector + increment > data_end())
               Sector = 0;
            else
               Sector += increment;
            break;

         case '-':
            if (Sector - increment < 0)
               Sector = data_end();
            else
               Sector -= increment;
            break;

         default:
            break;
      }
   }
}


set_sector()

{  char buff[10];
   char *Gets();
   register int i;

   help_3();
   CRSR_ON;

   FOREVER
   {  GOTO (MENUE, JUMP);
      DEL_EOL;
      i = atoi(Gets(buff));
```

```
/**************************/
/* 'Blättern' auf der Dis- */
/*  kette.                */
/**************************/
/* Der Beutzer kann sich  */
/*  mit den Tasten + und - */
/*  durch die Disk bewegen.*/
/*  Drückt er gleichzei-  */
/*  tig die Shift-Taste,  */
/*  so wird die Schritt-  */
/*  weite für das Blättern */
/*  verändert (Zweierpo-  */
/*  tenzen).              */
/*  Beim Blättern wird    */
/*  stets mit angezeigt,  */
/*  wo man sich gerade be- */
/*  findet (Sektornr. und */
/*  Bereich). Außerdem    */
/*  sorgt die Funktion da- */
/*  für, daß die Schritt- */
/*  weite nicht zu groß   */
/*  wird und man so nicht */
/*  über die Disk hinaus- */
/*  blättern kann.        */
/*  Ob die Shift-Taste be- */
/*  tätigt wurde, kann man */
/*  mit der Funktion "Get- */
/*  shift" des BIOS fest- */
/*  stellen.              */
/*  Versucht der Benutzer, */
/*  über den zulässigen   */
/*  Sektorenbereich hi-   */
/*  nauszublättern, dann  */
/*  wird einfach an den   */
/*  Anfang bzw. das Ende  */
/*  der Diskette gegangen. */
/*                        */
/*                        */
/*                        */
/*                        */
/*                        */
/**************************/

/**************************/
/* Sektor absolut (durch  */
/*  Eingabe einer Zahl)   */
/*  setzen.               */
/**************************/
/* "Gets" ist Ersatz für  */
/*  defektes "gets" aus der*/
/*  Library.              */
/* Benutzer zur Eingabe ei-*/
/*  ner Zahl auffordern.  */
/*                        */
/* Zahl lesen und ignorie- */
/*  ren, falls ungültig.  */
/*                        */
```

```
        if (i > data_end() || i < 0)
           continue;
        else
        {  Sector = i;
           break;
        }
     }
}


char get_command()

{  register char comm_char;

        GOTO (22,0);
        DEL_EOL;
        GOTO (22, 32);
        Cconws("Ihr Kommando? ");
        CRSR_ON;
        comm_char = Crawcin();
        CRSR_OFF;
        GOTO (22, 46);
        REV_ON;
        putchar(comm_char);
        REV_OFF;
        return comm_char;
}


char *Areas[] =
{  "SYS ",
   "FAT1",
   "FAT2",
   "DIR ",
   "DTA ",
} ;


update()

{  GOTO (STATUS, AREA);
   Cconws(Areas[Area]);
   GOTO (STATUS, SECTOR);
   dez_dump ((long) Sector, 4);
   GOTO (MENUE, SCROLL);
   dez_dump ((long) Sector, 4);
}



drv_out(drv)
        register int drv;

{  GOTO(STATUS, DRIVE);
   putchar ('A' + drv);
   GOTO(MENUE, DRIVE);
   putchar ('A' + drv);
}
```

```
/*                          */
/*                          */
/* Ansonsten neuen Sektor   */
/*  setzen.                 */
/*                          */
/*                          */
/*                          */
/**************************/

/**************************/
/* Kommando vom Benutzer    */
/*  holen.                  */
/**************************/
/*                          */
/* Benutzer-Prompt anzei-   */
/*  gen.                    */
/*                          */
/*                          */
/*                          */
/* Zeichen ohne Echo lesen */
/*                          */
/*                          */
/*                          */
/* und revers anzeigen.     */
/*                          */
/*                          */
/**************************/

/**************************/
/* Strings für Anzeige      */
/*  des Diskettenbereichs; */
/*  werden von "update"     */
/*  gebraucht.              */
/*                          */
/*                          */
/**************************/

/**************************/
/* Veränderungen am Bild-   */
/*  schirm anzeigen.        */
/* Aktuellen Bereich an-    */
/*  zeigen.                 */
/* Aktuellen Sektor anzei-  */
/*  gen (in Statuszeile     */
/*  und in Menüzeile (OK,   */
/*  ich weiß, dass das      */
/*  zuviel ist...))         */
/**************************/

/**************************/
/* Aktuelles Laufwerk in    */
/*  Statuszeile und         */
/*  Menüzeile anzeigen.     */
/*  (Ich hab's eben gern    */
/*  redundant!)             */
/*                          */
/*                          */
/**************************/
```

```
/*****************************************************************************/
/*  Funktionen zum Zeichnen der Bildschirm-Beschriftung (Menü- und Status-   */
/*  zeile.                                                                   */
/*****************************************************************************/

menout()

{  GOTO(MENUE, 0); Cconws(MEN1); Cconws(MEN2); Cconws(MEN3);
}


statout()

{  GOTO(STATUS, 0); Cconws(STAT1); Cconws(STAT2);
   GOTO(STATUS, 64); REV_ON; Cconws("Binär:"); REV_OFF;
}

                                                /*************************/
clear_junk()                                    /* Anzeigeberich des Bild- */
                                                /*  schirms von Schrott    */
{  register int i;                              /*  säubern, damit nach-   */
                                                /*  folgende Funktionen    */
   for (i = 2; i < 22; ++i)                     /*  eine sauber Grundlage  */
   {  GOTO (i, 0);                              /*  vorfinden.             */
      DEL_EOL;                                  /*                         */
   }                                            /*                         */
}                                               /*************************/

/*****************************************************************************/
/*  Diverse Hilfstexte auf den Bildschirm schreiben.                         */
/*****************************************************************************/

help_3()

{  GOTO (12, 22);
   Cconws ("Bitte Zahl eingeben (max. \033p4\033q Stellen)");
   accept();
}

help_2()

{  GOTO (13, 26);
   Cconws ("Schrittweite ändern mit \033pShift\033q");
   help_1();
}

help_1()

{  GOTO (12, 25);
   Cconws ("Wert ändern mit Tasten \033p+\033q oder \033p-\033q");
   accept();
}

accept()

{  GOTO (14, 27);
   Cconws ("Eingabe beenden mit \033pReturn\033q");
}
```

## 9.4 Listing des Moduls RECEDIT

```
#include <header.h>
#include <screen.h>
#include <ctype.h>

/*                      MODUL: RECEDIT                                    */

/***************************************************************************/
/*      Das folgende Modul enthält einen Editor für Disketten-Records (512  */
/*      Bytes), der es erlaubt, Records wahlweise im ASCII-Format oder he-  */
/*      xadezimal am Bildschirm zu bearbeiten und der über ein minimales Set */
/*      an Cursor-Steuerkommandos verfügt. Aufgerufen wird der Editor wie   */
/*      folgt:                                                              */
/*              edit (record, modus)                                        */
/*      wobei <record> ein Zeiger auf den zu edierenden Disketten-Record    */
/*      ist und <modus> entweder den Wert 0 oder 1 hat, was bedeutet, daß   */
/*      im ASCII-Modus (0) oder im Hex-Modus (1) gearbeitet werden soll.    */
/***************************************************************************/

                                                /*************************/
                                                /* Symbolische Konstanten */
#define ESC 0x1b                                /*  und Makros.          */
                                                /*************************/
#define UNDO 97                                 /* Scancodes der Funk-    */
#define HOME 71                                 /*  tionstasten, die vom  */
#define CLEAR 82                                /*  Editor interpretiert  */
#define UP 72                                   /*  werden und die ent-   */
#define LEFT 75                                 /*  sprechenden Funktionen */
#define DOWN 80                                 /*  auslösen.             */
#define RIGHT 77                                /*                        */
                                                /* Bildschirmpositionen:  */
#define ALT_STRNG  54                           /* Label und Feld für al- */
#define ALT_DISP   59                           /*  ternative Darstellung. */
#define BINARY 71                               /* Pos. f. Binärdarstellung*/
#define DEZ 60                                  /* Pos. f. Dezimaldarst.  */
#define DEZPOS 39                               /* Für Positionsanzeige   */
#define HEXPOS 57                               /*  des Cursors am BS.    */
                                                /*                        */
#define ASCII 0                                 /* Modi des Editors.      */
#define HEX 1                                   /*                        */
                                                /* Für HEX-Modus: ist BS- */
                                                /*  Cursor gerade im nie- */
#define Lo_byte() (! (Ypos % 2))                /*  derwertigen Nibble?   */
                                                /*************************/


                                                /*************************/
                                                /* Globale Variablen.     */
                                                /*************************/
int X1 = 5,                                     /* Anfangszeile BS-Fenster */
    Y1,                                         /*  dto, Y-Koordinate.    */
    Xoff = 15,                                  /* Ausdehnung senkrecht   */
    Yoff,                                       /*  dto, waagerecht.      */
    Xpos = 0,                                   /* Aktuelle Koordinaten des*/
    Ypos = 0,                                   /*  Bildschirm-Cursors.   */
    Modus;                                      /* Modus (Hex oder ASCII). */
                                                /*                        */
                                                /*                        */
```

```
char *Start,                                    /* Record-Anfang.          */
     *Cursor;                                   /* Aktuelle Position im    */
                                                /*  Record.                */
char Dez[] = "\033pDEZ:\033q    ";              /* Labels für Alternativ-  */
char Hex[] = "\033pHEX:\033q    ";              /*  anzeige.               */
                                                /*                         */
                                                /*                         */
int (*Alternate) ();                            /* Alternative Anzeigefunk-*/
char *Altstrng;                                 /*  tion und Label.        */
                                                /*************************/

                                                /*************************/
                                                /* Ediere den <record> im  */
                                                /*  <modus> ASCII oder HEX.*/
edit (record, mode)                             /*************************/
     char *record;                              /*                         */
     int mode;                                  /*                         */
                                                /*                         */
{  char savebuf[512];                           /* Alter Record-Inhalt.    */
   int character;                               /* Zeichen von der Tastat. */
   int scancode;                                /* Scancode desselben.     */
   int (*show_fun)();                           /* Anzeigefunktion.        */
   int (*legal)();                              /* Eingabezeichen OK?      */
   int (*insert)();                             /* Zeichen in Record einfü-*/
                                                /*  gen.                   */
   extern int asc_out(),                        /* Diverse Funktionen, die */
             hex_out(),                         /*  je nach Modus an die   */
             asc_in(),                          /*  obigen 3 Funktions-    */
             true(),                            /*  Zeiger zugewiesen      */
             hex_d(),                           /*  werden.                */
             hex_in(),                          /*=========================*/
             is_hex(),                          /* Je nach Modus werden    */
             dez_out();                         /*  nun verschiedene Para- */
                                                /*  meter gesetzt, die     */
   switch (mode)                                /*  das Bildschirmfenster, */
   {  case ASCII:                               /*  Ausgabefunktionen und  */
         show_fun = asc_out;                    /*  Eingabefunktionen be-  */
         insert = asc_in;                       /*  treffen.               */
         legal = true;                          /* Anzeige-, Einfüge-,     */
         Alternate = hex_d;                     /*  Test- und Alternativ-  */
         Altstrng = Hex;                        /*  anzeigefunktion setzen.*/
         Y1 = 24;                               /*                         */
         Yoff = 31;                             /* ASCII-Anzeige ab Spalte */
         break;                                 /*  24 mit 32 Zeichen pro  */
                                                /*  Zeile.                 */
      case HEX:                                 /*                         */
         show_fun = hex_out;                    /*                         */
         insert = hex_in;                       /*                         */
         legal = is_hex;                        /*                         */
         Alternate = dez_out;                   /*                         */
         Altstrng = Dez;                        /*                         */
         Y1 = 8;                                /* HEX-Anzeige ab Spalte 8 */
         Yoff = 63;                             /*  mit 64 Zeichen/Zeile.  */
         break;                                 /*                         */
   }                                            /*=========================*/
```

```
Modus = mode;                          /* Modus auch für andere   */
cpy (savebuf, record);                 /*  Funktionen setzen.     */
Start = Cursor = record;               /* Record aufheben.        */
Xpos = Ypos = 0;                       /* Zeiger initialisieren.  */
CRSR_OFF;                              /*                         */
(*show_fun) (record, X1);              /* Mit ausgeschaltetem     */
frame();                               /*  Cursor Record in ge-   */
move();                                /*  wünschtem Modus anzei- */
                                       /*  gen, dann Rahmen zeich-*/
                                       /*  nen und BS-Cursor nach-*/
                                       /*  ziehen.                */
                                       /*=========================*/
FOREVER                                /* Kommando-'Interpreter'. */
{  character = scan (&scancode);       /* Zeichen und Scancode    */
   CRSR_ON;                            /*  lesen.                 */
   if (character)                      /* Zeichen gelesen:        */
   {  if (character == ESC)            /* Editor verlassen.       */
         return;                       /*                         */
      if ((*legal)(character))         /* Zeichen ist OK, also in */
         (*insert) (character);        /*  Record eintragen,      */
      continue;                        /* ansonsten übergehen.    */
   }                                   /*                         */
   else switch (scancode)              /* Funktionstaste gelesen: */
   {  case HOME:                       /*  Je nach gedrückter     */
         home();                       /*  Funktionstaste die     */
         continue;                     /*  entsprechende Funktion */
                                       /*  aufrufen.              */
      case UP:                         /* Cursor nach oben.       */
         up();                         /*                         */
         continue;                     /*                         */
                                       /*                         */
      case LEFT:                       /* Cursor nach links.      */
         left();                       /*                         */
         continue;                     /*                         */
                                       /*                         */
      case RIGHT:                      /* Cursor nach rechts.     */
         right();                      /*                         */
         continue;                     /*                         */
                                       /*                         */
      case DOWN:                       /* Cursor nach unten.      */
         down();                       /*                         */
         continue;                     /*                         */
                                       /*                         */
      case CLEAR:                      /* Record löschen (mit bi- */
         clear (record, '\0');         /*  nären Nullen auffül-   */
         (*show_fun) (record, X1);     /*  len), und leeren Record*/
         Cursor = record;              /*  anzeigen.              */
         home();                       /*                         */
         continue;                     /*                         */
                                       /*                         */
      case UNDO:                       /* Alten Record-Inhalt wie-*/
         cpy(record, savebuf);         /*  derherstellen.         */
         CRSR_OFF;                     /*                         */
         (*show_fun)(record,  X1);     /* Record anzeigen.        */
         CRSR_ON;                      /*                         */
         Cursor = record;              /*                         */
         home();                       /*                         */
         continue;                     /*                         */
```

```
        default:
            continue;
      }
   }
}


cpy (d, s)
     register char *d, *s;

{  register int i;

   for (i = 512; i--; *d++ = *s++)
      ;
}


asc_in(c)
       register int c;

{  *Cursor = (char) c;
   Bconout(5, c);
   right();
}


hex_in(c)
      register int c;

{  c = toupper(c);
   putchar(c);

   if (Lo_byte())
      *Cursor = (*Cursor
                 & (char) 0x0f)
                 | (hextoi(c) << 4);
   else
      *Cursor = (*Cursor
                 & (char) 0xf0)
                 | (hextoi(c)
                    & (int) 0x0f);

   right();
}


int hextoi (c)
    register int c;

{  if (isdigit(c))
      return (c - '0');
   else
      return (10 + (c - 'A'));
}
```

```
/*                          */
/*                          */
/*                          */
/*                          */
/**************************/

/**************************/
/* Record kopieren von <s> */
/*  nach <d>.              */
/**************************/
/*                          */
/*                          */
/*                          */
/* 512 Bytes kopieren.      */
/*                          */
/**************************/

/**************************/
/* Zeichen im ASCII-Modus  */
/*  einfügen.              */
/**************************/
/* Altes Zeichen über-      */
/*  schreiben, echo und     */
/*  BS-Cursor nachziehen.   */
/**************************/

/**************************/
/* Zeichen im HEX-Modus     */
/*  in Record einfügen.     */
/**************************/
/*                          */
/* Zeichen großmachen und   */
/*  am BS anzeigen.         */
/*                          */
/* BS-Cursor war in Lo-Byte*/
/*  Zeichen als höherwer-   */
/*  tiges Nibble in altes   */
/*  Zeichen einblenden.     */
/*                          */
/* BS-Cursor war in Hi-Byte*/
/*  Zeichen als niederwer-  */
/*  tiges Nibble in altes   */
/*  Zeichen einblenden.     */
/*                          */
/* Cursor nachziehen.       */
/**************************/


/**************************/
/* Hex-Ziffer in Zahl wan- */
/*  deln.                   */
/**************************/
/* Ziffern sind leicht;     */
/*  Buchstaben müssen erst */
/*  transformiert werden.   */
/*                          */
/**************************/
```

```
                                                /*************************/
int true(c)                                     /* Ist Zeichen legales AS- */
        int c;                                  /*  CII-Zeichen?           */
                                                /*************************/
{  return TRUE;                                 /* Jedes Zeichen ist er-   */
}                                               /*  laubt, also immer      */
                                                /*  TRUE zurückgeben...    */
                                                /*************************/


                                                /*************************/
int is_hex(c)                                   /* Ist Zeichen Hex-Zeichen?*/
        int c;                                  /*************************/
                                                /* Nur Ziffern und Buchsta-*/
{  c = (char) toupper(c);                       /*  ben 'A' bis 'F' sind   */
   return (isdigit (c) || (c >= 'A' && c <= 'F'));/*  erlaubt.              */
}                                               /*************************/


                                                /*************************/
home()                                          /* Cursor in linke obere   */
                                                /*  Ecke / an Recordanfang.*/
{  Xpos = Ypos = 0;                             /*************************/
   Cursor = Start;                              /* Record-Zeiger un BS-Cur-*/
   move();                                      /*  sor zurücksetzen.      */
}                                               /*************************/


                                                /*************************/
left()                                          /* Cursor nach links.      */
                                                /*************************/
{  if (Ypos == 0)                               /* Wir sind am linken Rand */
   {  if (Xpos == 0)                            /*  und in der 1. Spalte;  */
      {  Xpos = Xoff;                           /*  also ans Fenster-Ende  */
         Ypos = Yoff;                           /*  gehen und ans Record-  */
         Cursor = Start + 511;                  /*  ende positionieren u.  */
         move();                                /*  Veränderung anzeigen.  */
         return;                                /*                         */
      }                                         /*                         */
      else                                      /* Falls am linken Rand,   */
      {  --Xpos;                                /*  ans Ende der vorigen   */
         Ypos = Yoff;                           /*  Zeile gehen.           */
      }                                         /*                         */
   }                                            /*                         */
   else --Ypos;                                 /* Ansonsten einfach nach  */
                                                /*  links.                 */
   if (Modus == ASCII                           /* Record-Zeiger zurück,   */
       || (Modus == HEX                         /*  falls ASCII oder im    */
           && !Lo_byte()))                      /*  Hex-Modus Cursor nicht */
      --Cursor;                                 /*  auf Lo-Byte.           */
   move();                                      /* BS-Cursor nachziehen.   */
}                                               /*************************/


                                                /*************************/
right()                                         /* Cursor nach rechts.     */
                                                /*************************/
{  if (Ypos == Yoff)                            /* Wir sind am rechten     */
   {  if (Xpos == Xoff)                         /*  Rand und in der letz-  */
      {  home();                                /*  ten Zeile; also Cursor */
         return;                                /*  in linke obere Ecke.   */
      }                                         /*                         */
```

```
      else                                          /* Falls nur am rechten    */
      {  ++Xpos;                                    /*  Rand, eine Zeile zu-   */
         Ypos = 0;                                  /*  rück und an den Anfang.*/
      }                                             /*                         */
   }                                                /*                         */
   else ++Ypos;                                     /* Ansosnten einfach nach  */
                                                    /*  rechts gehen.          */
   if (Modus == ASCII                               /* Zeiger in Textpuffer    */
       || (Modus == HEX                             /*  nur nachziehen, wenn   */
           && Lo_byte()))                           /*  ASCII-Modus oder wenn  */
       ++Cursor;                                    /*  in Lo-Nibble bei HEX-  */
   move();                                          /*  Modus.                 */
}                                                   /*************************/

                                                    /*************************/
up()                                                /* Cursor eine Zeile nach  */
                                                    /*  oben.                  */
{  if (Xpos == 0)                                   /*************************/
   {  Xpos = Xoff;                                  /* Falls in erster Zeile,  */
      Cursor = Start                                /*  in letzte Zeile sprin- */
               + 480                                /*  gen und Textzeiger     */
               + (Modus == HEX ? Ypos / 2 : Ypos);/*  setzen.                */
   }                                                /*                         */
   else                                             /* Ansonsten eine Zeile    */
   {  --Xpos;                                       /*  zurück.                */
      Cursor -= 32;                                 /*                         */
   }                                                /*                         */
                                                    /* Veränderung am Bild-    */
   move();                                          /*  schirm anzeigen.       */
                                                    /*                         */
}                                                   /*************************/

                                                    /*************************/
down()                                              /* Cursor um eine Zeile    */
                                                    /*  nach unten.            */
{  if (Xpos == Xoff)                                /*************************/
   {  Xpos = 0;                                     /* Falls in letzter Zeile, */
      Cursor = Start                                /*  in erste Zeile sprin-  */
               + (Modus == HEX ? Ypos / 2 : Ypos);/*  gen und Textzeiger     */
   }                                                /*  richtig setzen.        */
   else                                             /*                         */
   {  ++Xpos;                                       /* Ansonten eine Zeile     */
      Cursor += 32;                                 /*  nach unten.            */
   }                                                /*                         */
                                                    /* Veränderung am Bild-    */
   move();                                          /*  schirm anzeigen.       */
}                                                   /*************************/

/*****************************************************************************/
/*  Das Programm verwaltet die alternative Anzeigefunktion in einem Funk-    */
/*  tions-Pointer "alternate". Da angenommen wird, daß die Funktionen ein-   */
/*  stellig sind, begegnen uns hier zwei alte Bekannte ("dez_dump" und       */
/*  "hex_dump" in neuem Gewand -- eben als einstellige Varianten. Bei        */
/*  "dez_out" muß dafür gesorgt werden, daß Grafik-Zeichen (8. Bit gesetzt)  */
/*  richtig behandelt werden und nicht durch Vorzeichen-Erweiterung als ne-  */
/*  gative Werte auf dem Bildschirm landen.                                  */
/*****************************************************************************/
```

```
                                                /*************************/
dez_out (c)                                     /* Dezimalwert eines ASCII-*/
        char c;                                 /*  Zeichens ausgeben.     */
                                                /*************************/
{  dez_dump ((long) unsign(c), 3);              /* Kann auch Grafik-Zeichen*/
}                                               /*  richtig verarbeiten.   */
                                                /*************************/


                                                /*************************/
hex_d (c)                                       /* Einstelliges Alias für  */
      char c;                                   /*  einen alten Bekannten. */
                                                /*                         */
{  hex_dump ((long) c, sizeof (char));          /*                         */
}                                               /*************************/


                                                /*************************/
move()                                          /* Bildschirm aktualisie-  */
                                                /*  ren.                   */
{  CRSR_OFF;                                    /*************************/
   GOTO (22, DEZPOS);                           /* Aktuelle Position des   */
   dez_dump ((long) Cursor - Start, 3);         /*  Cursors im Record an-  */
   GOTO (22, HEXPOS);                           /*  zeigen: dezimal        */
   hex_dump ((long) Cursor - Start, sizeof (int));/*  ...und hexadezimal.    */
   GOTO(24, ALT_STRNG);                         /* Aktuelles Byte in der   */
   Cconws(Altstrng);                            /*  Alternativdarstellung  */
   GOTO(24, ALT_DISP);                          /*  anzeigen               */
   (*Alternate) (*Cursor);                      /*                         */
   GOTO (24, BINARY);                           /*  ...und auch noch binär */
   bin_dump ((long) *Cursor, sizeof (char));    /*  ausgeben.              */
   GOTO (Xpos + X1, Ypos + Y1);                 /*                         */
   CRSR_ON;                                     /* Das wars...             */
}                                               /*************************/


                                                /*************************/
clear (record, byte)                            /* <record> mit <byte> fül-*/
      register char *record, byte;              /*  len.                   */
                                                /*************************/
{  register int i;                              /* Dient im Programm zum   */
                                                /*  Löschen des Records    */
   for (i = 512; i; --i)                        /*  (<byte> = '\0').       */
     *record++ = byte;                          /*                         */
}                                               /*************************/


                                                /*************************/
frame()                                         /* Rahmen um Bildschirm-   */
                                                /*  fenster zeichnen.      */
{  register int i, anfz, anfsp, end;            /*************************/
                                                /* Größenparameter in Ab-  */
   anfsp = Y1 - 2;                              /*  hängigkeit vom Modus   */
   anfz = X1 - 1;                               /*  setzen.                */
   end = Y1 + Yoff + 2;                         /*                         */
                                                /*                         */
   GOTO(anfz, anfsp);                           /* Kopfzeile zeichnen.     */
   for (i = Y1; i < end + 2;  ++i)              /*                         */
     putchar('_');                              /*                         */
```

```
    while (++ anfz <= X1 + Xoff)                    /* Randeinfassung zeich-   */
    {  GOTO (anfz, anfsp);                          /*  nen.                   */
       putchar('|');                                /*                         */
       GOTO (anfz, end);                            /*                         */
       putchar('|');                                /*                         */
    }                                               /*                         */
    GOTO(anfz, anfsp);                              /* Fußzeile zeichnen.      */
    for (i = Y1; i < end + 2; ++i)                  /*                         */
       putchar('-');                                /*                         */
}                                                   /*************************/
```

## 9.5 Listing des Moduls UTILITY

```
#include <header.h>
#include <screen.h>
#include <ctype.h>

/*                    MODUL: UTILITY                                    */


                                                  /***************************/
char *Gets(line)                                  /* String von der Tastatur */
          char *line;                             /*  lesen (Ersatz für feh- */
{  register char *cp;                             /*  lerhafte Library-Funk- */
   register char i;                               /*  tion).                 */
                                                  /***************************/
   cp = line;                                     /*                         */
   while ((i = Cconin()) != '\r')                 /* Zeichen mit Echo von    */
      *cp++ = i;                                  /*  Konsole holen bis      */
   *cp = 0;                                       /*  <RETURN> gedrückt.     */
   if (cp == line)                                /* String abschliessen.    */
      return NULL;                                /* Nichts gelesen?         */
   return line;                                   /*                         */
}                                                 /*                         */
                                                  /***************************/


                                                  /***************************/
asc_out(record, anfz)                             /* Record (512 Bytes) zen- */
        register char *record;                    /*  triert ab Zeile <anfz> */
        register int anfz;                        /*  als ASCII-Text ausgeben*/
                                                  /***************************/
{                                                 /*                         */
   register int  zeile, zlr;                      /* Ausgegeben werden 16    */
                                                  /*  Zeilen mit 32 Zeichen. */
   for (zeile = anfz ; zeile < (anfz + 16) ;  ++zeile)  /*  Positionieren    */
   {  GOTO(zeile, 24);                            /*  über Escape-Sequenz.   */
      for (zlr = 32; zlr; --zlr)                  /*                         */
        Bconout(5,*record++);                     /* Ausgaben über BIOS-Funk-*/
   }                                              /*  tion, um Grafikzeichen */
}                                                 /*  zu bekommen.           */
                                                  /***************************/


                                                  /***************************/
hex_out(record, anfz)                             /* Record zentriert ab Zei-*/
        register char *record;                    /*  le <anfz> hexadezimal  */
        register int anfz;                        /*  ausgeben.              */
                                                  /***************************/
{                                                 /*                         */
   register int  zeile, zlr;                      /*                         */
                                                  /*                         */
   for (zeile = anfz;  zeile < (anfz + 16); ++zeile)  /* Ausgegeben werden   */
   {  GOTO(zeile, 8);                             /*  16 Zeilen mit 64       */
      for (zlr = 32; zlr; --zlr)                  /*  Spalten.               */
        hex_dump((long) *record++, sizeof (char));  /*  Siehe selbstge-      */
   }                                              /*  schnitze Funktion.     */
}                                                 /*                         */
                                                  /***************************/
```

```
/*****************************************************************************/
/*      Anmerkung: die folgenden Ausgabefunktionen für dezimale, hexadezim-  */
/*      male, oktale und binäre Ausgabe erwarten, daß ihr erstes Argument    */
/*      eine "long" ist ("cast"-Operator verwenden!) und daß das zweite Ar-  */
/*      gument Auskunft über die tatsächliche Größe des Ausgabewertes gibt   */
/*      bzw. (bei "dez_dump") über die Größe des Ausgabefeldes. Ist also     */
/*      die "char"-Variable "c" hexadezimal auszugeben, dann lautet der kor- */
/*      rekte Aufruf:                                                        */
/*                  hex_dump ((long) c, sizeof (char));                      */
/*****************************************************************************/


                                                    /***************************/
dez_dump(l, format)                                 /* Dezimalzahl in einem    */
        register long int l;                        /*  Feld der Breite <for-  */
        register int format;                        /*  mat> ausgeben.         */
                                                    /***************************/
{  if (!format)                                     /*  Rekursionsbasis.       */
      return;                                       /*                         */
   dez_dump((long) (l / 10), --format);             /*  Erst die höherwerti-   */
   putchar('0' + (int)(l % 10));                    /*  gen Stellen ausgeben,  */
}                                                   /*  dann die aktuelle.     */
                                                    /*                         */
                                                    /***************************/



                                                    /***************************/
hex_dump(value, format)                             /* Hexadezimale Ausgabe.   */
     register long value;                           /*  <value>: Ausgabewert   */
     int format;                                    /*  <format>: Byte-Größe   */
                                                    /*    von <value>          */
                                                    /***************************/
{                                                   /* Es werden der Reihe nach*/
   register unsigned int byte;                      /*  die einzelnen Halbbytes*/
   register int shift;                              /*  von <value> ausgegeben.*/
                                                    /*  Die Schleife macht sie */
   for (shift = (format * 2) - 1; shift >= 0; --shift)  /* durch Shift und   */
   {   byte = (value >> (shift * 4)) & (long) 0x0f;  /* Ausmaskieren der un- */
       putchar( byte + (byte > 9 ? 'W' : '0'));     /*  gewuneschten Stellen   */
   }                                                /*  verfuegbar.            */
}                                                   /***************************/

                                                    /***************************/
okt_dump(value, format)                             /* Oktale Ausgabe:         */
     register long value;                           /*  <value>: Ausgabewert   */
     int format;                                    /*  <format>: Datentyp des */
                                                    /*    Ausgabewerts.        */
                                                    /***************************/
{                                                   /*                         */
   register unsigned int byte;                      /* In Dreiergruppen den    */
   register int shift;                              /*  Ausgabewert verschieben*/
                                                    /*  und das entsprechende  */
   for (shift = ((format * 8) / 3) * 3; shift >= 0; shift -= 3)  /*  Zeichen */
   {   byte = (value >> shift) & (long) 0x07;       /*  ausgeben.              */
       putchar( byte + '0');                        /*                         */
   }                                                /*                         */
}                                                   /***************************/
```

```
                                                /*************************/
bin_dump(value, format)                         /* Binaere Ausgabe.      */
     register long value;                       /*  <value>: Ausgabewert */
     register int format;                       /*  <format>: Byte-Größe */
                                                /*************************/
{  register int j;                              /* Es werden der Reihe nach*/
                                                /*  alle Bits von <value> */
   for (j = (format * 8) - 1; j >= 0; --j)      /*  ausmaskiert und ange- */
      if (value & (1 << j))                     /*  zeigt. <format> be-   */
         putchar('1');                          /*  stimmt, ab wo mit dem */
      else                                      /*  Ausmaskieren zu begin-*/
         putchar('0');                          /*  nen ist.              */
}                                               /*************************/

                                                /*************************/
char scan(i)                                    /* Zeichen ohne Echo von */
     int *i;                                    /*  Tastatur lesen und in*/
                                                /*  <i> Scancode zurück- */
{  long l;                                      /*  geben.               */
                                                /*************************/
   l = Bconin(2);                               /* Bconin liefert "long" */
   *i = (int) (l >> 16);                        /*  mit Scancode in High */
   return (char) l;                             /*  Word und ASCII-Zeichen*/
}                                               /*  in Low Word.         */
                                                /*************************/

/*****************************************************************************/
/*      Modul zur lesbaren Ausgabe eines Records mit Inhaltsverzeichnis-     */
/*      Einträgen. Dem Modul fehlt einiges an der sonst üblichen Eleganz von */
/*      C-Programmen, was von der (vorsichtig gesprochen:) fragwürdigen Ent- */
/*      scheidung von Atari herrührt, für Directory-Records das Aufzeich-    */
/*      nungsformat von MS-DOS zu übernehmen, und das bedeutet: alles, was   */
/*      größer als 1 Byte ist, wird in der Reihenfolge Low - High aufge-     */
/*      zeichnet, also gerade umgekehrt, als dies der M68000 macht. Deswegen */
/*      müssen Integers und Longs erst 'umgedreht' werden, ehe sie von C ver-*/
/*      standen werden; und deswegen hat es auch gar keinen Sinn, mit Struc- */
/*      tures zu arbeiten. Einige der Routinen in diesem Modul sind daher al-*/
/*      les andere als lupenrein -- andererseits sieht man an ihnen auch,    */
/*      was in C alles geht!.                                                */
/*****************************************************************************/



#define dez(n,s)  dez_dump (abs((long) n), s)

#define GROESSE 23                              /* Konstanten, die die Po- */
#define ZEIT 32                                 /*  sition der einzelnen   */
#define DATUM 40                                /*  Felder am Bildschirm   */
#define CLUSTER 51                              /*  bestimmen.             */
#define ATTRIBUTE 58
#define UEB "Name          Größe     Zeit    Datum      Cluster  Attribute"


                                                /*************************/
dir_out(record, anfz)                           /* Record mit Directory- */
        register char *record;                  /*  Einträgen zentriert  */
        register int anfz;                      /*  und interpretiert am */
                                                /*  Bildschirm ausgeben. */
```

```
{
   register int  zeile;

   GOTO(3, 6);
   Cconws(UEB);
   for (zeile = anfz;  zeile < (anfz + 16);  ++zeile)
   {  dir_dump(record,zeile);
      record += 32;
   }
}


dir_dump(dir_entry,zeile)
        register char *dir_entry;
        register int zeile;

{
   unsigned int i, unsign();


   i = unsign(*dir_entry);



   switch(i)
   {  case 0x0:   GOTO(zeile, 6);
                  Cconws("unbenutzt");
                  return;

      case 0xe5:  GOTO(zeile, ATTRIBUTE);
                  Cconws("gelöscht!");
                  dir_entry[11] = '\0';
                  break;

      default:    break;
   }

   name_dump(dir_entry, zeile);
   size_dump(dir_entry + 0x1c, zeile);
   cl_dump(dir_entry + 0x1a, zeile);
   time_dump(dir_entry + 0x16, zeile);
   date_dump(dir_entry + 0x18, zeile);
   attr_dump(unsign(dir_entry[0xb]), zeile);
}



name_dump(cp, zeile)
     char *cp;
     int zeile;

{  int i;
   char c;

   GOTO(zeile, 6);

   for (i = 0; i < 8; ++i)
      putchar((c = *cp++) == 0 ? ' ' : c);
```

```
                                /*************************/
                                /*                       */
                                /*                       */
                                /* Überschrift ausgeben; */
                                /*  "puts" geht nicht, da */
                                /*  es einen Zeilen-     */
                                /*  vorschub erzeugt!    */
                                /* Directory-Eintrag ist */
                                /*  32 Bytes lang.       */
                                /*************************/

                                /*************************/
                                /* Einzelnen Directory-  */
                                /*  Eintrag ausgeben.    */
                                /*************************/
                                /*                       */
                                /* Diese Routine ist nur */
                                /*  ein 'Manager', der zu */
                                /*  den einzelnen Arbeits-*/
                                /*  knechten verzweigt.  */
                                /* Das erste Byte des Ein-*/
                                /*  trags hat besondere  */
                                /*  Bedeutung; um es für */
                                /*  "switch" heranziehen */
                                /*  zu können, wird daraus*/
                                /*  eine "unsigned" gemacht*/
                                /*  (Compilerfehler!)    */
                                /*                       */
                                /*                       */
                                /*                       */
                                /* Die Attribute eines ge-*/
                                /*  löschten Eintrags zu-*/
                                /*  rücksetzen (allerdings*/
                                /*  können so keine Sub- */
                                /*  Directories mehr er- */
                                /*  kannt werden!).      */
                                /*                       */
                                /* Zu den einzelnen Routi-*/
                                /*  nen verzweigen. Da   */
                                /*  Structures aus den o.a.*/
                                /*  Gründen nicht möglich*/
                                /*  sind, werden die Daten*/
                                /*  durch Offset-Adressie-*/
                                /*  rung bestimmt!       */
                                /*************************/

                                /*************************/
                                /* Namenskomponente aus- */
                                /*  geben.               */
                                /*************************/
                                /*                       */
                                /*                       */
                                /*                       */
                                /*                       */
                                /* Ausgabe ab Spalte 6.  */
                                /* Name ist 8 Zeichen lang;*/
                                /*  der Aufwand ist not- */
                                /*  wendig, da Kennsatz- */
```

```
    putchar('.');                                   /*  Einträge Nullbytes     */
    for (i = 0; i < 3; ++i)                         /*  enthalten können!      */
       putchar((c = *cp++) == 0 ? ' ' : c);         /* Extension ist 3 Zeichen */
    Cconws(" |");                                   /*  lang.                  */
}                                                   /*************************/



#define TAGMASKE (int) 0x1f
#define MONATSMASKE (int) 0xf
#define JAHRESMASKE (int) 0x7f
                                                    /*************************/
date_dump(cp,zeile)                                 /* Datumskomponente aus- */
     char *cp;                                      /*  geben.               */
     int zeile;                                     /*************************/
                                                    /*                       */
{                                                   /*                       */
    unsigned int tag, monat, jahr, datum, mk_int();/*                       */
                                                    /* Erst mal aus der IBM-In-*/
    datum = mk_int(cp);                             /*  teger eine 68K-Integer */
                                                    /*  machen (grrr...).    */
    tag = datum & TAGMASKE;                         /* Signifikante Bits her- */
    monat = (datum >> 5) & MONATSMASKE;             /*  shiften und ausmas-  */
    jahr = 80 + ((datum >> 9) & JAHRESMASKE);       /*  kieren (siehe erläu- */
                                                    /*  ternden Text).       */
    GOTO(zeile,DATUM);                              /*                       */
    dez (tag, 2);                                   /* Tag, Monat und Jahr wer-*/
    putchar('.'); dez (monat, 2);                   /*  den jeweils zweistel- */
    putchar('.'); dez (jahr, 2);                    /*  lig ausgegeben.      */
    Cconws(" |");                                   /*                       */
}                                                   /*************************/


                                                    /*************************/
unsigned mk_int(cp)                                 /* Aus 8086-Integer eine */
         char *cp;                                  /*  8080-Integer machen. */
                                                    /*************************/
{  static char i[2];                                /* (Der Herr möge mir ver- */
   unsigned *pui;                                   /*  zeihen...)           */
                                                    /* Byte-Reihenfolge umdre- */
   i[1] = *cp++;  i[0] = *cp;                       /*  hen, passenden Zeiger */
   pui = (unsigned *) i;                            /*  mit Gewalt auf die   */
   return *pui;                                     /*  Bytefolge biegen und */
}                                                   /*  dereferenzieren; uff! */
                                                    /*************************/



#define TMASKE1 (int) 0x1f
#define TMASKE2 (int) 0x3f
                                                    /*************************/
time_dump(cp,zeile)                                 /* Zeitkomponente ausgeben.*/
     char *cp;                                      /*************************/
     int zeile;                                     /*                       */
                                                    /*                       */
{                                                   /*                       */
    unsigned int minuten, stunden, zeit, mk_int(); /*                       */
                                                    /*                       */
    zeit = mk_int(cp);                              /* Signifikante Bits her- */
```

```
   minuten = (zeit >> 5) & TMASKE2;
   stunden = (zeit >> 11) & TMASKE1;

   GOTO(zeile,ZEIT);
   dez (stunden, 2);
   putchar (':'); dez (minuten, 2);
   Cconws (" |");
}


size_dump(cp,zeile)
     char *cp;
     int zeile;

{  char sz[4];
   int i;
   long l, *lp;

   GOTO(zeile,GROESSE);
   for (i = 3; i >= 0; --i)
      sz[i] = *cp++;
   lp = (long *) sz;
   l = *lp;
   dez (l , 6);
   Cconws (" |");
}



cl_dump(cp,zeile)
     char *cp;
     int zeile;

{  unsigned i, mk_int();

   GOTO(zeile,CLUSTER);
   i = mk_int(cp);

   dez (i,  5);
   Cconws (" |");
}


#define RO 0x1
#define UNS 0x2
#define SYS 0x4
#define VOL 0x8
#define DIR 0x10
#define ARC 0x20

attr_dump(i,zeile)
     unsigned int i;
     int zeile;

{

   GOTO(zeile,ATTRIBUTE);
```

```
/*  shiften und ausmaskie- */
/*  ren.                   */
/*                         */
/* Stunden und Minuten wer-*/
/*  den zweistellig ange-  */
/*  zeigt, getrennt durch  */
/*  Doppelpunkt.           */
/*************************/

/*************************/
/* Dateigröße ausgeben.   */
/*************************/
/* Die Größe ist im Di-   */
/*  rectory als "long" ge- */
/*  speichert, aber natür- */
/*  lich wieder im Intel-  */
/*  Format (Seufz!). Also  */
/*  müssen erst wieder die */
/*  Bytes umgedreht werden */
/*  und man muß dem Com-   */
/*  piler weismachen, daß  */
/*  er es mit einer "long" */
/*  zu tun hat, ehe die    */
/*  Zahl dann sechsstellig */
/*  ausgegeben werden kann.*/
/*                         */
/*************************/

/*************************/
/* Anfangscluster des Ein- */
/*  trags ausgeben.       */
/*************************/
/*                         */
/* Das ist einfach, weil  */
/*  die Clusternummer ei-  */
/*  ne Integer ist; also   */
/*  nix Shiften und Maskie-*/
/*  ren...                 */
/*                         */
/* Fünfstellige Ausgabe.  */
/*************************/

/*************************/
/* Die Attribute sind in  */
/*  zwei Byte codiert; die */
/*  defines zeigen, daß    */
/*  für jedes Attribut ein */
/*  Bit vorgesehen ist.    */
/*                         */
/*************************/
/* Attribute des Eintrags  */
/*  anzeigen.              */
/*************************/
/*                         */
/*                         */
/*                         */
/*                         */
/*                         */
```

```
   if (i == 0)                                  /* Keine Attribute.       */
   /* Keine Attribute */                        /*                        */
      return;                                   /*                        */
   if (i & DIR)                                 /* Die folgenden beiden   */
   {  Cconws("Unterverzeichnis");               /*  Attribute schließen   */
      return;                                   /*  sich gegenseitig aus; */
   }                                            /*  alle anderen sind     */
   if (i & VOL)                                 /*  kumulativ (bedeutet:  */
   {  Cconws("Diskettenkennung");               /*  keine "return"s mehr). */
      return;                                   /*                        */
   }                                            /*                        */
   if (i & RO)                                  /* Attribut: nur lesen.   */
      Cconws("R/O  ");                          /*                        */
   if (i & UNS)                                 /* Attribut: unsichtbare  */
      Cconws("UNS  ");                          /*  Datei                 */
   if (i & SYS)                                 /* Attribut: Systemdatei  */
      Cconws("SYS  ");                          /*  (unsichtbarer als un- */
   if (i & ARC)                                 /*  sichtbar!)            */
      Cconws("ARC  ");                          /* Attribut: Archiv-Bit   */
}                                               /*  gesetzt.              */
                                                /************************/

                                                /************************/
unsigned int unsign(c) register char c;         /* Aus einem Zeichen eine */
                                                /*  "unsigned" machen.    */
{                                               /************************/
   return((int) c & 0x00ff);                    /* Vorzeichenbits ausmas- */
}                                               /*  kieren!               */
                                                /************************/

/*************************************************************************/
/*  Die folgenden Funktionen realisieren ein Modul zum Initialisieren von */
/*  Disketten. Das Modul ist als Datenkapsel angelegt, d.h. es kommuniziert */
/*  mit der Aussenwelt durch den Austausch von Botschaften, erlaubt aber den */
/*  rufenden Funktionen des Diskmonitors keinerlei Zugriff auf seine inter- */
/*  nen Daten.  Beim Wechsel des Laufwerks stellt das Modul fest, ob es sich */
/*  um ein zulässiges Laufwerk handelt und besorgt sich dann die nötigen  */
/*  Laufwerksparameter (Anfang und Größe von Directory, FAT, Datenbereich  */
/*  usw.). Auf Wunsch (durch Aufruf von Anfragefunktionen) teilt es diese  */
/*  Werte nach Außen mit.                                                  */
/*************************************************************************/

#define Getbpb(a) (struct bpb *) bios(7,a)
#define Drvmap()  bios(10)

                                                /************************/
struct bpb                                      /* BIOS-Parameter-Block */
{                                               /*  für Diskette.       */
   WORD  recsiz,                                /************************/
         clsiz,                                 /*  Wird von BIOS-Funktion */
         clsizb,                                /*  7 geliefert; EInzelhei- */
         rdlen,                                 /*  ten finden Sie bei der */
         fsiz,                                  /*  Beschreibung der BIOS- */
         fatrec,                                /*  Funktionen im Anhang.  */
         datrec,                                /*                         */
         numcl,                                 /*                         */
         bflags;                                /*                         */
};                                              /************************/
```

```
/*****************************************************************************/
/*  Diskettenparameter in globalen statischen Variablen; auf Verlangen teilt */
/*  das Modul diese Werte nach außen mit.                                    */
/*************************************************                           */
                                                  /* Anzahl der angeschlos-  */
static int Ndrives = 0;                           /*  senen Laufwerke.       */
static int Drive = -1;                            /* Nr. des aktuellen LWs.  */
static int Fat_1,                                 /* Anfangssektor FAT1      */
           Fat_2,                                 /* Anfangssektor FAT2      */
           Directory,                             /* Anfangssektor Directory */
           Data_start,                            /* Anfangssektor Daten     */
           Data_end,                              /* Letzter Diskettensektor */
           Bps,                                   /* Bytes pro Sektor        */
           Spc;                                   /* Sektoren pro Cluster.   */
                                                  /*************************/

int fat_1()                                       /*************************/
                                                  /* Mit diesen 'Auskunfts-  */
{  return Fat_1;                                  /*  funktionen' können an- */
}                                                 /*  dere Funktionen die    */
                                                  /*  in diesem Modul intern */
int fat_2()                                       /*  gespeicherten Werte    */
{  return Fat_2;                                  /*  abfragen. Die Methode  */
}                                                 /*  stellt sicher, daß     */
int directory()                                   /*  die Werte von außen    */
{  return Directory;                              /*  auf keinen Fall geän-  */
}                                                 /*  dert werden können; Sie*/
int data_start()                                  /*  wurde hauptsächlich    */
{  return Data_start;                             /*  aus didaktischen Grün- */
}                                                 /*  den gewählt.           */
int data_end()                                    /*                         */
{  return Data_end;                               /*                         */
}                                                 /*************************/

/*****************************************************************************/
/*  Aktuelles Laufwerk besorgen. Diese Funktion initialisiert zugleich die   */
/*  Diskette im aktuellen Laufwerk, falls sie noch nie zuvor gerufen wurde.  */
/*  Dies ist beim Programmstart des Diskmonitors der Fall; "drv_get" erkennt */
/*  diese Situation an einem negativen Wert in der Variablen "Drive" (siehe  */
/*  Initialisierung.                                                         */
/************************************************                            */
                                                  /*                         */
int drv_get()                                     /*                         */
                                                  /* Erster Aufruf der Funk- */
{  if (Drive < 0)                                 /*  tion?                  */
      drv_init(Dgetdrv());                        /* Ja: alles initialisie-  */
   return Drive;                                  /*  ren.                   */
}                                                 /*************************/

/*****************************************************************************/
/*  Diese Funktion wechselt das aktuelle Laufwerk und initialisiert es,      */
/*  falls ein LAufwerk mit der gewünschten Nummer angeschlossen ist. Falls   */
/*  nicht, signalisiert es dies an die rufende Funktion mit einem negativen  */
/*  Fehlerwert.                                                              */
/************************************************                            */
                                                  /*                         */
int drv_set(drv)                                  /* Aktuelles LW wechseln.  */
    int drv;                                      /*                         */
```

```
{                                                   /*                          */
   if (drv > Ndrives || drv < 0)                    /* Laufwerksnummer legal?   */
      return FALSE;                                 /*  Wenn nicht, Fehler.     */
   drv_init(drv);                                   /* Ansonsten initialisie-   */
   return TRUE;                                     /*  ren und Erfolg melden.  */
}                                                   /**************************/


/*************************************************************************/
/*  Laufwerk initialisieren. Um auch den erstmaligen Aufruf richtig zu be-  */
/*  handeln, bestimmt die Funktion die Anzahl der angeschlossenen Laufwerke, */
/*  leitet illegale Initialisierungswünsche auf Laufwerk 0 um und bestimmt  */
/*  dann die Disketten-Parameter aus dem BIOS-Parameter-Block des gewählten  */
/*  Laufwerks.                                                             */
/*************************************************                        */
                                                    /*                      */
int drv_init(drv)                                   /* Laufwerk <drv> initia-  */
    int drv;                                        /*  lisieren.           */
                                                    /*                      */
{  long drvs;                                       /*                      */
   struct bpb *d_p;                                 /*                      */
                                                    /*                      */
   drvs =  Drvmap();                                /* Bitvektor der Laufwerke */
   while (drvs >>= 1)                               /*  vom BIOS besorgen und  */
      ++Ndrives;                                    /*  anzahl bestimmen.   */
                                                    /*                      */
   if (drv > Ndrives || drv < 0)                    /* <drv> ist illegale Zahl, */
      drv = 0;                                      /*  also auf LW "A" umbie- */
                                                    /*  gen.                */
   Dsetdrv(drv);                                    /* In gewähltes Laufwerk  */
                                                    /*  einloggen.          */
   Drive = drv;                                     /* Aktuelles LW vermerken. */
                                                    /*                      */
   d_p = Getbpb(drv);                               /* BPB für aktuelles LW be-*/
                                                    /*  sorgen.             */
   Fat_1 = d_p->datrec -                            /* Anfang von FAT1 vom An- */
           d_p->rdlen -                             /*  fang des Datenbereichs */
           (2 * d_p->fsiz);                         /*  her zurückrechnen.  */
                                                    /*                      */
   Fat_2 = Fat_1 + d_p->fsiz;                       /* Die Bestimmung von FAT2 */
                                                    /*  ist jetzt einfach...  */
   Directory = Fat_2 + d_p->fsiz;                   /* Ebenso der Anfang des  */
                                                    /*  Directory.          */
   Data_start = d_p->datrec;                        /*                      */
                                                    /*                      */
   Data_end = Data_start +                          /* Letzten Sektor vom An- */
              (d_p->numcl * d_p->clsiz);            /*  fang des Datenbereichs */
                                                    /*  hochrechnen.        */
   Bps = d_p->recsiz;                               /*                      */
   Spc = d_p->clsiz;                                /*                      */
}                                                   /**************************/
```

```
/*****************************************************************************/
/*  Diese Funktion erhält eine Sektornummer und teilt dann mit, in welchem   */
/*  Diskettenbereich sich der entsprechende Sektor befindet (System, FAT,    */
/*  Directory oder Daten).                                                   */
/*****************************************************************************/

int where_is(sect)
    int sect;

{  if (sect < Fat_1)
      return 0;
   if (sect >= Fat_1 && sect < Fat_2)
      return  1;
   if (sect >= Fat_2 && sect < Directory)
      return 2;
   if (sect >= Directory && sect < Data_start)
      return 3;
   if (sect >= Data_start && sect <= Data_end)
      return 4;
   return -1;
}
```

## 9.6 Die Datei screen.h

```
/*****************************************************************************
*                                                                            *
*        Bildschirm-Steuercodes für ATARI ST                                 *
*                                                                            *
*****************************************************************************/

#define ESC 0x1b
#define SCRN(a) putchar(ESC); putchar(a)
                                                /*                          */
#define CRSR_UP SCRN('A')                       /* C. eine Zeile nach oben  */
#define CRSR_DOWN SCRN('B')                     /* ... nach unten           */
#define CRSR_RT SCRN('C')                       /* C. nach rechts           */
#define CRSR_LT SCRN('D')                       /* C. nach links            */
#define CLRSCRN SCRN('E')                       /* BS loeschen              */
#define HOME SCRN('H')                          /* C. in linke obere Ecke   */
#define DEL_EOP SCRN('J')                       /* bis BS-Ende loeschen     */
#define DEL_EOL SCRN('K')                       /* bis Zeilenende loeschen  */
#define INSRT_LIN SCRN('L')                     /* Zeile einfuegen          */
#define DEL_LIN SCRN('M')                       /* Zeile loeschen           */
#define CRSR_OFF SCRN('f')                      /* C. ausschalten           */
#define CRSR_ON SCRN('e')                       /* C. einschalten           */
#define DEL_BOS SCRN('d')                       /* von BS-Anfang loeschen   */
#define STORE_CRSR SCRN('j')                    /* C-Position merken        */
#define RESTORE_CRSR SCRN('k')                  /* an gemerkte Pos. gehen   */
#define DEL_LIN SCRN('l')                       /* Zeile loeschen           */
#define DEL_BOL SCRN('o')                       /* ab Zeilenanfang loeschen*/
#define REV_ON SCRN('p')                        /* Reverse Darstellung ein  */
#define REV_OFF SCRN('q')                       /*  ... und aus             */
                                                /* Es folgt die Funktion    */
                                                /*  zur absoluten Cursor-   */
                                                /*  positionierung:         */
#define GOTO(x,y) SCRN('Y');putchar((char) x + 32); putchar((char) y + 32)
```

## 9.7 Die Datei cnames.h

```
extern long bios();
extern long gemdos();
/*
    BIOS-Funktionen
*/
#define Getmbp()          bios(0)
#define Bconstat(a)       bios(1,a)
#define Bconin(a)         bios(2,a)
#define Bconout(a,b)      bios(3,a,b)
#define Rwabs(a,b,c,d,e)  bios(4,a,b,c,d,e)
#define Setexc(a,b)       bios(5,a,b)
#define Tickcal           bios(6)
#define Getbpb(a)         bios(7,a)
#define Bcostat(a)        bios(8,a)
#define Mediach(a)        bios(9,a)
#define Drvmap()          bios(10)
#define Getshift()        bios(11)
/*
   GEMDOS-Funktionen
*/
#define Pterm0()          gemdos(0x0)
#define Cconin()          gemdos(0x1)
#define Cconout(a)        gemdos(0x2,a)
#define Cauxin()          gemdos(0x3)
#define Cauxout(a)        gemdos(0x4,a)
#define Cprnout(a)        gemdos(0x5,a)
#define Crawio(a)         gemdos(0x6,a)
#define Crawcin()         gemdos(0x7)
#define Cnecin()          gemdos(0x8)
#define Cconws(a)         gemdos(0x9,a)
#define Cconrs(a)         gemdos(0x0a,a)
#define Cconis()          (int)gemdos(0x0b)
#define Dsetdrv(a)        gemdos(0x0e,a)
#define Cconos()          gemdos(0x10)
#define Cprnos()          gemdos(0x11)
#define Cauxis()          gemdos(0x12)
#define Cauxos()          gemdos(0x13)
#define Dgetdrv()         (int)gemdos(0x19)
#define Fsetdta(a)        gemdos(0x1a,a)
#define Super(a)          gemdos(0x20,a)
#define Tgetdate()        (int)gemdos(0x2a)
#define Tsetdate(a)       gemdos(0x2b,a)
#define Tgettime()        (int)gemdos(0x2c)
#define Tsettime(a)       gemdos(0x2d,a)
#define Fgetdta()         gemdos(0x2f)
#define Sversion()        (int)gemdos(0x30)
#define Ptermres(a,b)     gemdos(0x31,a,b)
#define Dfree(a,b)        gemdos(0x36,a,b)
#define Dcreate(a)        gemdos(0x39,a)
#define Ddelete(a)        gemdos(0x3a,a)
#define Dsetpath(a)       gemdos(0x3b,a)
#define Fcreate(a,b)      gemdos(0x3c,a,b)
#define Fopen(a,b)        gemdos(0x3d,a,b)
#define Fclose(a)         gemdos(0x3e,a)
#define Fread(a,b,c)      gemdos(0x3f,a,b,c)
#define Fwrite(a,b,c)     gemdos(0x40,a,b,c)
```

```
#define Fdelete(a)        gemdos(0x41,a)
#define Fseek(a,b,c)      gemdos(0x42,a,b,c)
#define Fattrib(a,b,c)    gemdos(0x43,a,b,c)
#define Fdup(a)           gemdos(0x45,a)
#define Fforce(a,b)       gemdos(0x46,a,b)
#define Dgetpath(a,b)     gemdos(0x47,a,b)
#define Malloc(a)         gemdos(0x48,a)
#define Mfree(a)          gemdos(0x49,a)
#define Mshrink(a,b)      gemdos(0x4a,0,a,b)
#define Pexec(a,b,c,d)    gemdos(0x4b,a,b,c,d)
#define Pterm(a)          gemdos(0x4c,a)
#define Fsfirst(a,b)      (int)gemdos(0x4e,a,b)
#define Fsnext()          (int)gemdos(0x4f)
#define Frename(a,b,c)    gemdos(0x56,a,b,c)
#define Fdatime(a,b,c)    gemdos(0x57,a,b,c)
```

## 9.8 Die Datei ctype.h

```
#define isupper(c) (c >= 'A' && c <= 'Z')
#define islower(c) (c >= 'a' && c <= 'z')
#define isdigit(c) (c >= '0' && c <= '9')
#define isvokal(c) (c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U')
#define toupper(c) (islower(c) ? c + 'A' - 'a' : c)
#define tolower(c) (isupper(c) ? c + 'a' - 'A' : c)
#define ischar(c) (isupper(c) || islower(c))
```

# Anhang A: Die GEMDOS-Funktionen

00 `Pterm0()`
Prozeß beenden; gibt Kontrolle an rufendes Programm zurück (mit Return-Code 0)

01 `Cconin()`
Liefert Zeichen vom Standard-Eingabekanal; gelesenes Zeichen wird an Standard-Ausgabekanal geechot.

Wert von *Cconin* ist eine *long int*, bei der im unteren Wort das Zeichen, im oberen Wort der Scan-Code der betätigten Taste steht.

02 `Cconout(chr)`
`char chr;`
Schreibt ein Zeichen auf den Standard-Ausgabekanal

03 `Cauxin()`
Holt ein Zeichen von der seriellen Schnittstelle (RS232-Schnittstelle)

04 `Cauxout(chr)`
`char chr;`
Gibt ein Zeichen an der seriellen Schnittstelle aus.

05 `Cprnout(chr)`
`char chr;`
Schickt ein Zeichen an den Drucker (Parallelschnittstelle). Kann das Zeichen nicht gesendet werden, so hat die Funktion den Wert 0.

06 `Crawio(wrd)`
`int wrd;`
Allgemeine zeichenweise Ein-/Ausgabe-Funktion für die Standard-Kanäle. Falls *wrd* den Wert *0x00ff* hat, wird ein Zeichen (ohne Echo) vom Standard-Eingabekanal gelesen und als Wert zurückgegeben; ist kein Zeichen verfügbar, dann liefert die Funktion den Wert 0.

Hat *wrd* einen anderen Wert als *0x00ff*, dann wird dieser Wert als ASCII-Zeichen interpretiert, das am Standard-Ausgabekanal ausgegeben wird.

07 `Crawcin()`
Zeichen ohne Echo vom Standard-Eingabekanal lesen. Steuerzeichen wie z.B. Control-C oder Control-S werden nicht interpretiert.

08 `Cnecin()`
Zeichen ohne Echo vom Standard-Eingabekanal lesen. Steuerzeichen wie z.B. Control-C oder Control-S werden im Unterschied zu *Crawcin* interpretiert.

09 `Cconws(str)`
`char *str;`
Schreibt einen String auf den Standard-Ausgabekanal; der String muß C-Konventionen genügen, also mit einem Null-Byte abgeschlossen sein.

0a `Cconrs(buf)`
`char *buf;`
Liest eine Zeile vom Standard-Eingabekanal. Beim Aufruf der Funktion gibt **buf* die Länge des Puffers abzüglich 1 ab; nach Verlassen der Funktion enthält **(buf+ 1)* die Anzahl der gelesenen Zeichen; die Daten beginnen an der Position **(buf + 2)*.

Die Funktion erlaubt das Edieren der Eingabezeichen (z.B. Zeichen löschen mit Backspace).

0b `Cconis()`
Überprüft den Status des Standard-Eingabekanals; hat den Wert 0, falls kein Zeichen verfügbar ist und einen von Null verschiedenen Wert sonst.

0e `Dsetdrv(drv)`
`int drv;`
Aktuelles Laufwerk setzen; *drv* ist die Laufwerksnummer (0 = A, 1 = B, ...). Wert von *Dsetdrv* ist ein Bitvektor der dem System bekannten Laufwerke (Bit 0 = A, Bit 1 = B, ...).

10 `Cconos()`
Überprüft den Ausgabestatus der Konsole; hat den Wert 0, wenn die Konsole kein Zeichen empfangen kann und einen von 0 verschiedenen Wert sonst.

11 `Cprnos()`
Überprüft den Ausgabestatus des Druckers; hat den Wert 0, wenn der Drucker kein Zeichen empfangen kann und einen von 0 verschiedenen Wert sonst.

12 `Cauxis()`
Überprüft den Eingabestatus der seriellen Schnittstelle; hat den Wert 0, wenn kein Zeichen an der RS232-Schnittstelle verfügbar ist und einen von 0 verschiedenen Wert sonst.

13 `Cauxos()`
Überprüft den Ausgabestatus der seriellen Schnittstelle; hat den Wert 0, wenn die RS232-Schnittstelle kein Zeichen empfangen kann und einen von 0 verschiedenen Wert sonst.

19 `Dgetdrv()`
Liefert die Laufwerksnummer des aktuellen Laufwerks.

1a `Fsetdta(ptr)`
`long ptr;`
Setzt die DTA (Disk-Transfer-Adresse) für nachfolgende Aufrufe von *Fsfirst* und *Fsnext*.

20 `Super(stack)`
`long stack;`
Versetzt den Prozessor in User- bzw. Supervisor-Modus. Wenn sich der Prozessor beim Aufruf von *Super* im User-Modus befindet, dann wird der Supervisor-Modus eingeschaltet, wobei *stack* als Supervisor-Stack benutzt wird. Hat *stack* den Wert 0, dann wird der aktuelle User-Stack als Supervisor-Stack benutzt.

Befindet sich der Prozessor beim Aufruf von *Super* im Supervisor-Modus, dann kehrt er in den User-Modus zurück. Wert von *Super* ist der alte Wert des Supervisor-Stacks.

2a `Tgetdate()`
Liefert das Systemdatum; Wert von *Tgetdate* ist eine Bitvektor, der wie folgt interpretiert werden muß:

```
Bits
0..4    Tag 1..31
5..8    Monat 1..12
9..15   Jahr 0..119 (gerechnet ab 1980!)
```

2b `Tsetdate(date)`
`int date;`
Setzt das Systemdatum; *date* muß in dem bei *Tgetdate* beschriebenen Format sein.

2c `Tgettime()`
Liefert die Systemzeit als Bitvektor, der wie folgt interpretiert werden muß:

```
Bits
0..4    Sekunden 0..59
5..10   Minuten 0..59
11..15  Stunden 0..23
```

2d `Tsettime(time)`
`int time;`
Setzt die Systemzeit; *time* muß in dem bei *Tgettime* beschriebenen Format sein.

2f `Fgetdta()`
Liefert die Adresse der aktuellen DTA (siehe Funktion *Fsetdta*, *Fsnext* und *Fsfirst*).

30 `Sversion()`
Liefert die Versionsnummer von GEMDOS.

31 `Ptermres(keep, ret)`
`long keep;`
`int ret;`
Beendet den aktuellen Prozeß, ohne ihn aus dem Speicher zu entfernen (hält ihn also resident). *ret* ist der Return-Code, der an den rufenden Prozeß übergeben wird; *keep* ist die Anzahl der Bytes, die im Speicher behalten werden sollen.

36 `Dfree(buf, drv)`
`struct info *ip;`
`int drv;`
Liefert Informationen über das Laufwerk, dessen Nummer in *drv* übergeben wird. Die Information wird in einer Struktur übergeben, auf die *ip* zeigt. Die *info*-Struktur ist wie folgt aufgebaut:

```
struct info
{  long b_free;    /* Anzahl freier Cluster auf dem Laufwerk */
   long b_total;   /* Gesamtzahl der Cluster */
   long b_secsiz;  /* Anzahl Bytes pro Sektor */
   long b_clsiz;   /* Anzahl Sektoren pro Cluster */
}
```

39 `Dcreate(path)`
`char *path;`
Richtet ein Teilverzeichnis ein; *path* enthält den (MS-DOS-kompatiblen) Zugriffspfad auf das einzurichtende Teilverzeichnis. *path* ist ein String, der der C-Konvention gehorcht.

3a `Ddelete(path)`
`char *path;`
Löscht ein Teilverzeichnis; zur Beschreibung von *path* siehe *Dcreate.*

3b `Dsetpath(path)`
`char *path;`
Wechselt das aktuelle Teilverzeichnis; zur Beschreibung von *path* siehe *Dcreate.*

3c `Fcreate(name, attr)`
`char *name;`
`int attr;`
Richtet eine Datei ein; *name* ist der Dateiname, der entweder ohne Zugriffspfad angegeben werden kann, womit die Datei im aktuellen (Unter)-Verzeichnis eingerichtet wird, oder mit einem Pfad versehen ist, wobei das angegebene Unterverzeichnis jedoch bereits auf der Diskette existieren muß.

Wert von *Fcreate* ist entweder eine Kanalnummer oder eine negative Fehlernummer.

Im Parameter *attr* - ein Bitvektor - können Datei-Attribute gesetzt werden; die einzelnen Bits haben folgende Bedeutung:

```
01    nur lesender Zugriff auf Datei erlaubt
02    versteckte Datei; wird nicht angezeigt
04    Systemdatei; ist ebenfalls versteckt
08    Diskettenkennung (11 Zeichen lang)
```

3d `Fopen(name, mode)`
`char *name;;`
`int mode;`
Öffnet eine Datei mit Namen *name*; *mode* hat den Wert 0, 1 oder 2, je nachdem, ob lesender, schreibender oder uneingeschränkter Zugriff verlangt wird. Wert der Funktion ist eine Dateinummer ("File Handle"), wobei mit 6 begonnen wird, da die ersten 5 Nummern bereits für die Standard-Kanäle vergeben sind, nämlich

```
0  für Standard-Eingabe (Tastatur)
1  für Standard-Ausgabe (Bildschirm)
2  für Standard-Fehlerkanal (Bildschirm)
3  für die serielle Schnittstelle
4  für den Drucker
```

3e `Fclose(handle)`
`int handle;`
Schließt die Datei mit der Nummer *handle.*

3f `Fread(handle, count, buf)`
```
int  handle;
long count;
char *buf;
```
Überträgt *count* Bytes aus der Datei mit der Kanalnummer *handle* in den Puffer *buf*. Wert ist die Anzahl der tatsächlich gelesenen Bytes oder eine negative Fehlernummer (vgl. dazu *read* aus der Standard-Bibliothek und das Kapitel 7!).

40 `Fwrite(handle, count, buf)`
```
int  handle;
long count;
char *buf;
```
Schreibt *count* Bytes in die Datei mit der Kanalnummer *handle* aus dem Puffer *buf*. Wert ist die Anzahl der tatsächlich geschriebenen Bytes oder eine negative Fehlernummer (vgl. dazu *write* aus der Standard-Bibliothek und das Kapitel 7!).

41 `Fdelete(name)`
```
char *name;
```
Löscht die Datei mit dem angegebenen Namen.

42 `Fseek(offset, handle, mode)`
```
long offset;
int  handle;
int  mode;
```
Setz den Dateizeiger für Lese-/Schreibzugriffe in der Datei mit Kanalnummer *handle*. *offset* ist der Versatzwert, der je nach dem Wert von *mode* unterschiedlich interpretiert wird. Zulässige Werte für *mode* sind:

```
0       Positionieren ab Dateianfang
1       Positionieren ab aktueller Position
2       Positionieren ab Dateiende
```

(Vergleiche Kapitel 7).

43 `Fattrib(path, mode, attr)`
```
char *path;
int mode;
int attr;
```
Falls *mode* gleich 0, liefert die Funktion die Attribute der mit *path* bezeichneten Datei. Hat *mode* den Wert 1, so werden die Attribute auf den Wert von *attr* gesetzt. Attribute sind in einem Bit-Vektor gespeichert, der wie folgt zu interpretieren ist:

```
01    nur Lesezugriff erlaubt
02    versteckte Datei
04    Systemdatei (ganz besonders versteckt!)
08    Diskettenkennung
10    Teilverzeichnis
20    Archiv-Bit: Datei wurde beschrieben
```

45 `Fdup(stdhandle)`
```
int  stdhandle;
```
Liefert als Wert eine Kanalnummer, die einen Standard-Kanal (den Sie mit *stdhandle* auswählen; siehe die Funktion *Fopen*) bezeichnet; man hat somit zwei Nummern für dieselbe Datei zur Verfügung. Wird mit *Fseek* der Dateizeiger für den einen Kanal verändert, so ändert sich damit auch die Position im anderen Kanal.

46 `Fforce(stdhandle, nonstdhandle) [6]`
```
int  stdhandle;
int  nonstdhandle;
```
Erzwingt, daß *stdhandle* denselben Kanal wie *nonstdhandle* bezeichnet. Damit können die Standard-Kanäle umgelenkt werden. Ist *nonstdhandle* z.B. die Nummer einer Programmdatei, in die ausgegeben wird und *stdhandle* hat den Wert 1 (Bildschirmausgabe), dann werden die Bildschirm-Ausgaben in die datei gelenkt.

47 `Dgetpath(pathbuf, drv)`
```
char *pathbuf;
int  drv;
```
Schreibt den aktuellen Zugriffspfad für das Laufwerk *drv* (Laufwerksnummer angeben!) in den Puffer *pathbuf*. Der Puffer muß mindestens 64 Bytes lang sein.

48 `Malloc(amount)`
```
long amount;
```
Allociert Speicherplatz; *amount* gibt die Anzahl der zuzuweisenden Bytes an. Wert von *Malloc* ist ein Pointer auf den Anfang des allokierten Speicherplatzes oder 0, falls ein Fehler auftrat.

49 `Mfree(addr)`
```
char *addr;
```
Gibt den Speicherblock, auf den *addr* zeigt, wieder frei (siehe *Malloc*).

4a `Mshrink(zero, mem, size) [12]`
```
int  zero;
char *mem;
long size;
```
Verkürzt einen bereits zugewiesenen Speicherblock (siehe *Malloc*). Der Parameter *zero* muß den Wert 0 haben, *mem* ist ein Pointer auf den Anfang des Blocks und *size* gibt die Anzahl Bytes an, die im Block gehalten werden sollen. Ein von Null verschiedener Funktionswert zeigt einen Fehler an.

4b `Pexec(mode, path, commandline, environment)[16]`
```
int  mode;
char *path;
char *commandline;
char *environment;
```
Mit dieser Funktion kann das Laden und Nachladen von Programmen bewirkt werden. *path* enthält den Pfadnamen des Programms, *commandline* eine Kommandozeile, die an dieses Programm übergeben werden soll, *environment* ist der aktuelle (MS-DOS-kompatible) Umgebungsblock; hat er den Wert 0, so erbt das gerufene Programm die Umgebung des rufenden Prozesses. *mode* gibt Auskunft über den Lade-Modus; zulässige Werte sind:

```
0       Programm laden und Starten
3       Programm nur laden
4       Base-Page für Programm einrichtenbasepage
5       Programm starten
```

Im Modus 0 hat die Funktion als Wert den Return-Code des gerufenen Programms (siehe *Pterm0*, *Pterm* und *Ptermres*); ein negativer Funktionswert signalisiert in allen Modi einen Fehler.

4c `Pterm(code)`
```
int  code;
```
Aktuellen Prozeß beenden und Return-Code *code* an rufenden Prozeß zurückgeben.

4e `Fsfirst(spec, attr) [8]`

```
char *spec;
int  attr;
```

Directory durchsuchen nach einer Datei, deren Name auf das in *spec* gespeicherte Muster paßt. Mit *attr* kann die Suche auf Dateien mit bestimmten Attribut-Kombinationen beschränkt werden. *Fsfirst* liefert sein Ergebnis in der aktuellen DTA ab. (Genaueres finden Sie im Kapitel 7).

4f `Fsnext()`

Directory nach nächstem passenden Eintrag durchsuchen (s. Kapitel 7).

56 `Frename(zero, old, new) [12]`

```
int  zero;
char *old;
char *new;
```

Datei umbenennen. *zero* muß immer 0 sein; *old* enthält den alten, *new* den neuen Dateinamen.

57 `Fdatime(handle, buf, set) [10]`

```
int  handle;
char *buf;
int  set;
```

Liefert oder setzt das Datei-Datum für die Datei mit der Kanalnummer *handle* im Directory. Falls *set* den Wert 0 hat, liefert die Funktion Datum und Zeit im Puffer *buf* ab. Hat *set* den Wert 1, dann wird der Directory-Eintrag mit dem in *buf* gespeicherten Wert markiert.

# Anhang B: Die BIOS-Funktionen

(0) Getmpb

```
Getmpb(p_mpb)
long p_mpb;
```

*p_mpb* ist die Adresse eines Speicherblocks von der Größe *sizeof(MPB)*, der zur Aufnahme des anfänglichen Speicher-Parameterblocks ("MPB" ist die Abkürzung für "Memory Parameter Block") des Systems bestimmt ist und der von der Funktion mit Informationen gefüllt wird. Der Parameter-Block hat folgende Struktur:

```
define MPB      struct mpb
define MD       struct md
define PD       struct pd

MPB
{  MD *mp_mfl;          /* Freispeicher-Liste    */
   MD *mp_mal;          /* Liste zugewiesener Speicherbereiche */
   MD *mp_rover;        /* roving ptr */
};
MD
{  MD *m_link;          /* naechster MD (oder NULL) */
   long m_start;        /* Startaddresse des Blocks */
   long m_length;       /* Anzahl der Bytes im Block */
   PD *m_own;           /* Prozeß-Deskriptor */
};
```

(1) Bconstat

```
int  Bconstat(dev)
int  dev;
```

Liefert den Eingabestatus eines zeichenorientierten Geräts. Wert ist entweder 0, wenn kein Zeichen verfügbar ist, oder -1 (0xfff), wenn am Gerät (mindestens) ein Zeichen ansteht.

Zulässige Werte für den Parameter *dev* sind:

```
0   PRT: (Drucker; Parallelport)
1   AUX: (serielle RS232-Schnittstelle)
2   CON: (Bildschirm)
3   MIDI port
4   Keyboard port (intelligente Tastatur des Atari)
```

Die folgende Übersicht gibt an, welche Operationen mit den einzelnen Geräten erlaubt sind.

| Operation | (0) PRT | (1) AUX | (2) CON | (3) MIDI | (4) KBD |
|---|---|---|---|---|---|
| bconstat() | nein | ja | ja | ja | nein |
| bconin() | ja | ja | ja | ja | nein |
| bconout() | ja | ja | ja | ja | ja |
| bcostat() | ja | ja | ja | ja | ja |

(2) Bconin

```
long Bconin(dev)
int  dev;
```

Der Parameter *dev* gibt die Gerätenummer gemäß obiger Tabelle (Funktion *bconstat*).

Die Funktion liefert im niedrigstwertigen Byte des Ergebnisses (Byte 0) ein Zeichen vom angegebenen Gerät; sie wartet auf das Zeichen - eine gute Möglichkeit, das Programm aufzuhängen, wenn der betreffende Port nicht bereit ist!

Hat der Parameter *dev* den Wert 2 (Konsol-Eingabe), dann wird im niedrigwertigen Byte des Ergebnisses (Byte 0) der Tastatur-Scancode geliefert; das niedrigstwertige Byte des höherwertigen Worts (uff; Byte 2 eben!) enthält das ASCII-Zeichen.

Ist das dritte Bit der Systemvariable *conterm* gesetzt, dann enthält Byte 3 des Ergebnisses den Wert der Systemvariablen *kbshif2*.

(3) Bconout

```
Bconout(dev, c)
int  dev, c;
```

*dev* ist die Gerätenummer entsprechend der Tabelle unter Funktion 1. Das Zeichen *c* wird an das gewählte Gerät ausgegeben. Die Funktion wartet so lange, bis das Zeichen erfolgreich geschrieben werden konnte.

(4) Rwabs

```
long Rwabs(rwflag, buf, count, recno, dev)
int  rwflag;
long buf;
int  count, recno, dev;
```

Liest oder schreibt logische Sektoren auf der Diskette oder Harddisk. Zulässige Werte für *rwflag* sind:

```
0  Lesen
1  Schreiben
2  Lesen, Diskettenwechsel nicht beachten
3  Schreiben, Diskettenwechsel nicht beachten
```

*buf* zeigt auf einen Puffer, aus dem gelsen bzw. in den geschrieben werden soll.

*count* ist die Anzahl Sektoren, die übertragen werden sollen.

*recno* ist die logische Sektoren-Nummer, bei der die Übertragung begonnen werden soll.

*dev* ist die Gerätenummer; zulässige werte sind hier:

```
0   Floppy-Laufwerk A:
1   Floppy-Laufwerk B:
2   (und grösser: Harddisk, RAM-Disk etc)
```

Hat die Funktion den Wert 0, dann war die Übertragung erfolgreich. Ein negativer Wert signalisiert einen Fehler.

(5) Setexc

```
long Setexc(vecnum, vec)
int  vecnum;
long vec;
```

*vecnum* ist die Nummer des Vektors, der geholt oder gesetzt werden soll. *vec* ist die Vektor-Adresse, die den alten Vektor ersetzen soll. Falls hier der Wert *(long) -1* steht, nimmt die Funktion keine Veränderungen vor, sondern gibt lediglich den alten Wert des Vektors zurück.

(6) Tickcal

```
long Tickcal()
```

Liefert die zwischen zwei Aufrufen des System-Timers verstrichene Zeit, aufgerundet auf Millisekunden.

(7) Getbpb

```
BPB *Getbpb(dev)
int  dev;
```

*dev* ist eine Gerätenummer (0 für Laufwerk A, 1 für Laufwerk B etc). Die Funktion liefert einen Zeiger auf einen BIOS-Parameter-Block für das gewählte Gerät zurück.

(8) Bcostat

```
long Bcostat(dev)
```

*dev* ist die Nummer eines zeichenorientierten Geräts (vgl. Funktion 1); die Funktion liefert den Ausgabe-Status des Geräts zurück. Dieser ist wie folgt zu interpretieren:

```
-1   Gerät ist ausgabebereit.
 0   Gerät ist nicht bereit.
```

(9) Mediach

```
long Mediach(dev)
int  dev;
```

*dev* ist eine Laufwerks-Nummer. Die Funktion überprüft (soweit dies möglich ist) ob in dem angegebene Laufwerk die Diskette gewechselt wurde. Mögliche Werte sind:

```
0   Mit Sicherheit kein Wechsel.
1   Wechsel ist eventuell erfolgt.
2   Diskettenwechsel konnte mit Sicherheit festgestellt werden
```

(10) Drvmap

```
long Drvmap()
```

Liefert einen Bitvektor, in dem ein gesetztes Bit signalisiert, daß das entsprechende Gerät angeschlossen ist. Bit 0 steht für Laufwerk 0, Bit 1 für Laufwerk 1 etc.

(11) Kbshift

```
long Kbshift(mode)
int  mode;
```

Falls *mode* nicht negativ ist, werden die Shift-Bits der Tastatur wie angegeben gesetzt. Bei einem negativen *mode*-Wert wird der Zustand dieser Bits als Bitvektor zurückgegeben. Die einzelnene Bitpositionen im Vektor haben folgende Bedeutung:

```
0    Rechte Shift-Taste
1    Linke Shift-Taste
2    Control-Taste
3    ALT-Taste
4    CapsLock-Taste
5    Rechter Mausknopf (CLR/HOME)
6    Linker Mausknopf (INSERT)
7    (momentan nicht benutzt; auf 0 setzen)
```

# Anhang C: ASCII-Tabelle

| Dezimal | Hexadezimal | Oktal | Binär | ASCII |
|---|---|---|---|---|
| 000 | 00 | 000 | 00000000 | |
| 001 | 01 | 001 | 00000001 | ☺ |
| 002 | 02 | 002 | 00000010 | ☻ |
| 003 | 03 | 003 | 00000011 | ♥ |
| 004 | 04 | 004 | 00000100 | ♦ |
| 005 | 05 | 005 | 00000101 | ♣ |
| 006 | 06 | 006 | 00000110 | ♠ |
| 007 | 07 | 007 | 00000111 | • |
| 008 | 08 | 010 | 00001000 | ◘ |
| 009 | 09 | 011 | 00001001 | ○ |
| 010 | 0a | 012 | 00001010 | ◙ |
| 011 | 0b | 013 | 00001011 | ♂ |
| 012 | 0c | 014 | 00001100 | ♀ |
| 013 | 0d | 015 | 00001101 | ♪ |
| 014 | 0e | 016 | 00001110 | ♫ |
| 015 | 0f | 017 | 00001111 | ☼ |
| 016 | 10 | 020 | 00010000 | ► |
| 017 | 11 | 021 | 00010001 | ◄ |
| 018 | 12 | 022 | 00010010 | ↕ |
| 019 | 13 | 023 | 00010011 | ‼ |
| 020 | 14 | 024 | 00010100 | ¶ |
| 021 | 15 | 025 | 00010101 | § |
| 022 | 16 | 026 | 00010110 | ▬ |
| 023 | 17 | 027 | 00010111 | ↨ |
| 024 | 18 | 030 | 00011000 | ↑ |
| 025 | 19 | 031 | 00011001 | ↓ |
| 026 | 1a | 032 | 00011010 | → |
| 027 | 1b | 033 | 00011011 | ← |
| 028 | 1c | 034 | 00011100 | ∟ |
| 029 | 1d | 035 | 00011101 | ↔ |
| 030 | 1e | 036 | 00011110 | ▲ |
| 031 | 1f | 037 | 00011111 | ▼ |

| Dezimal | Hexadezimal | Oktal | Binär | ASCII |
|---|---|---|---|---|
| 032 | 20 | 040 | 00100000 | |
| 033 | 21 | 041 | 00100001 | ! |
| 034 | 22 | 042 | 00100010 | " |
| 035 | 23 | 043 | 00100011 | # |
| 036 | 24 | 044 | 00100100 | $ |
| 037 | 25 | 045 | 00100101 | % |
| 038 | 26 | 046 | 00100110 | & |
| 039 | 27 | 047 | 00100111 | ' |
| 040 | 28 | 050 | 00101000 | ( |
| 041 | 29 | 051 | 00101001 | ) |
| 042 | 2a | 052 | 00101010 | * |
| 043 | 2b | 053 | 00101011 | + |
| 044 | 2c | 054 | 00101100 | , |
| 045 | 2d | 055 | 00101101 | - |
| 046 | 2e | 056 | 00101110 | . |
| 047 | 2f | 057 | 00101111 | / |
| 048 | 30 | 060 | 00110000 | 0 |
| 049 | 31 | 061 | 00110001 | 1 |
| 050 | 32 | 062 | 00110010 | 2 |
| 051 | 33 | 063 | 00110011 | 3 |
| 052 | 34 | 064 | 00110100 | 4 |
| 053 | 35 | 065 | 00110101 | 5 |
| 054 | 36 | 066 | 00110110 | 6 |
| 055 | 37 | 067 | 00110111 | 7 |
| 056 | 38 | 070 | 00111000 | 8 |
| 057 | 39 | 071 | 00111001 | 9 |
| 058 | 3a | 072 | 00111010 | : |
| 059 | 3b | 073 | 00111011 | ; |
| 060 | 3c | 074 | 00111100 | < |
| 061 | 3d | 075 | 00111101 | = |
| 062 | 3e | 076 | 00111110 | > |
| 063 | 3f | 077 | 00111111 | ? |

| Dezimal | Hexadezimal | Oktal | Binär | ASCII |
|---|---|---|---|---|
| 064 | 40 | 100 | 01000000 | @ |
| 065 | 41 | 101 | 01000001 | A |
| 066 | 42 | 102 | 01000010 | B |
| 067 | 43 | 103 | 01000011 | C |
| 068 | 44 | 104 | 01000100 | D |
| 069 | 45 | 105 | 01000101 | E |
| 070 | 46 | 106 | 01000110 | F |
| 071 | 47 | 107 | 01000111 | G |
| 072 | 48 | 110 | 01001000 | H |
| 073 | 49 | 111 | 01001001 | I |
| 074 | 4a | 112 | 01001010 | J |
| 075 | 4b | 113 | 01001011 | K |
| 076 | 4c | 114 | 01001100 | L |
| 077 | 4d | 115 | 01001101 | M |
| 078 | 4e | 116 | 01001110 | N |
| 079 | 4f | 117 | 01001111 | O |
| 080 | 50 | 120 | 01010000 | P |
| 081 | 51 | 121 | 01010001 | Q |
| 082 | 52 | 122 | 01010010 | R |
| 083 | 53 | 123 | 01010011 | S |
| 084 | 54 | 124 | 01010100 | T |
| 085 | 55 | 125 | 01010101 | U |
| 086 | 56 | 126 | 01010110 | V |
| 087 | 57 | 127 | 01010111 | W |
| 088 | 58 | 130 | 01011000 | X |
| 089 | 59 | 131 | 01011001 | Y |
| 090 | 5a | 132 | 01011010 | Z |
| 091 | 5b | 133 | 01011011 | [ |
| 092 | 5c | 134 | 01011100 | \ |
| 093 | 5d | 135 | 01011101 | ] |
| 094 | 5e | 136 | 01011110 | ^ |
| 095 | 5f | 137 | 01011111 | _ |

| Dezimal | Hexadezimal | Oktal | Binär | ASCII |
|---|---|---|---|---|
| 096 | 60 | 140 | 01100000 | ` |
| 097 | 61 | 141 | 01100001 | a |
| 098 | 62 | 142 | 01100010 | b |
| 099 | 63 | 143 | 01100011 | c |
| 100 | 64 | 144 | 01100100 | d |
| 101 | 65 | 145 | 01100101 | e |
| 102 | 66 | 146 | 01100110 | f |
| 103 | 67 | 147 | 01100111 | g |
| 104 | 68 | 150 | 01101000 | h |
| 105 | 69 | 151 | 01101001 | i |
| 106 | 6a | 152 | 01101010 | j |
| 107 | 6b | 153 | 01101011 | k |
| 108 | 6c | 154 | 01101100 | l |
| 109 | 6d | 155 | 01101101 | m |
| 110 | 6e | 156 | 01101110 | n |
| 111 | 6f | 157 | 01101111 | o |
| 112 | 70 | 160 | 01110000 | p |
| 113 | 71 | 161 | 01110001 | q |
| 114 | 72 | 162 | 01110010 | r |
| 115 | 73 | 163 | 01110011 | s |
| 116 | 74 | 164 | 01110100 | t |
| 117 | 75 | 165 | 01110101 | u |
| 118 | 76 | 166 | 01110110 | v |
| 119 | 77 | 167 | 01110111 | w |
| 120 | 78 | 170 | 01111000 | x |
| 121 | 79 | 171 | 01111001 | y |
| 122 | 7a | 172 | 01111010 | z |
| 123 | 7b | 173 | 01111011 | { |
| 124 | 7c | 174 | 01111100 | \| |
| 125 | 7d | 175 | 01111101 | } |
| 126 | 7e | 176 | 01111110 | ~ |
| 127 | 7f | 177 | 01111111 | Δ |

| Dezimal | Hexadezimal | Oktal | Binär | ASCII |
|---|---|---|---|---|
| 128 | 80 | 200 | 10000000 | Ç |
| 129 | 81 | 201 | 10000001 | ü |
| 130 | 82 | 202 | 10000010 | é |
| 131 | 83 | 203 | 10000011 | â |
| 132 | 84 | 204 | 10000100 | ä |
| 133 | 85 | 205 | 10000101 | à |
| 134 | 86 | 206 | 10000110 | å |
| 135 | 87 | 207 | 10000111 | ç |
| 136 | 88 | 210 | 10001000 | ê |
| 137 | 89 | 211 | 10001001 | ë |
| 138 | 8a | 212 | 10001010 | è |
| 139 | 8b | 213 | 10001011 | ï |
| 140 | 8c | 214 | 10001100 | î |
| 141 | 8d | 215 | 10001101 | ì |
| 142 | 8e | 216 | 10001110 | Ä |
| 143 | 8f | 217 | 10001111 | Å |
| 144 | 90 | 220 | 10010000 | É |
| 145 | 91 | 221 | 10010001 | æ |
| 146 | 92 | 222 | 10010010 | Æ |
| 147 | 93 | 223 | 10010011 | ô |
| 148 | 94 | 224 | 10010100 | ö |
| 149 | 95 | 225 | 10010101 | ò |
| 150 | 96 | 226 | 10010110 | û |
| 151 | 97 | 227 | 10010111 | ù |
| 152 | 98 | 230 | 10011000 | ÿ |
| 153 | 99 | 231 | 10011001 | Ö |
| 154 | 9a | 232 | 10011010 | Ü |
| 155 | 9b | 233 | 10011011 | ¢ |
| 156 | 9c | 234 | 10011100 | £ |
| 157 | 9d | 235 | 10011101 | ¥ |
| 158 | 9e | 236 | 10011110 | ß |
| 159 | 9f | 237 | 10011111 | ƒ |

| Dezimal | Hexadezimal | Oktal | Binär | ASCII |
|---|---|---|---|---|
| 160 | a0 | 240 | 10100000 | á |
| 161 | a1 | 241 | 10100001 | í |
| 162 | a2 | 242 | 10100010 | ó |
| 163 | a3 | 243 | 10100011 | ú |
| 164 | a4 | 244 | 10100100 | ñ |
| 165 | a5 | 245 | 10100101 | Ñ |
| 166 | a6 | 246 | 10100110 | ª |
| 167 | a7 | 247 | 10100111 | º |
| 168 | a8 | 250 | 10101000 | ¿ |
| 169 | a9 | 251 | 10101001 | ⌐ |
| 170 | aa | 252 | 10101010 | ¬ |
| 171 | ab | 253 | 10101011 | ½ |
| 172 | ac | 254 | 10101100 | ¼ |
| 173 | ad | 255 | 10101101 | ¡ |
| 174 | ae | 256 | 10101110 | « |
| 175 | af | 257 | 10101111 | » |
| 176 | b0 | 260 | 10110000 | ã |
| 177 | b1 | 261 | 10110001 | õ |
| 178 | b2 | 262 | 10110010 | Ø |
| 179 | b3 | 263 | 10110011 | ø |
| 180 | b4 | 264 | 10110100 | œ |
| 181 | b5 | 265 | 10110101 | Œ |
| 182 | b6 | 266 | 10110110 | À |
| 183 | b7 | 267 | 10110111 | Ã |
| 184 | b8 | 270 | 10111000 | Õ |
| 185 | b9 | 271 | 10111001 | ¨ |
| 186 | ba | 272 | 10111010 | ´ |
| 187 | bb | 273 | 10111011 | † |
| 188 | bc | 274 | 10111100 | ¶ |
| 189 | bd | 275 | 10111101 | © |
| 190 | be | 276 | 10111110 | ® |
| 191 | bf | 277 | 10111111 | ™ |

| Dezimal | Hexadezimal | Oktal | Binär | ASCII |
|---|---|---|---|---|
| 192 | c0 | 300 | 11000000 | ﷐ |
| 193 | c1 | 301 | 11000001 | ﷑ |
| 194 | c2 | 302 | 11000010 | ﷒ |
| 195 | c3 | 303 | 11000011 | ﷓ |
| 196 | c4 | 304 | 11000100 | ﷔ |
| 197 | c5 | 305 | 11000101 | ﷕ |
| 198 | c6 | 306 | 11000110 | ﷖ |
| 199 | c7 | 307 | 11000111 | ﷗ |
| 200 | c8 | 310 | 11001000 | ﷘ |
| 201 | c9 | 311 | 11001001 | ﷙ |
| 202 | ca | 312 | 11001010 | ﷚ |
| 203 | cb | 313 | 11001011 | ﷛ |
| 204 | cc | 314 | 11001100 | ﷜ |
| 205 | cd | 315 | 11001101 | ﷝ |
| 206 | ce | 316 | 11001110 | ﷞ |
| 207 | cf | 317 | 11001111 | ﷟ |
| 208 | d0 | 320 | 11010000 | ﷠ |
| 209 | d1 | 321 | 11010001 | ﷡ |
| 210 | d2 | 322 | 11010010 | ﷢ |
| 211 | d3 | 323 | 11010011 | ﷣ |
| 212 | d4 | 324 | 11010100 | ﷤ |
| 213 | d5 | 325 | 11010101 | ﷥ |
| 214 | d6 | 326 | 11010110 | ﷦ |
| 215 | d7 | 327 | 11010111 | ﷧ |
| 216 | d8 | 330 | 11011000 | ﷨ |
| 217 | d9 | 331 | 11011001 | ﷩ |
| 218 | da | 332 | 11011010 | ﷪ |
| 219 | db | 333 | 11011011 | ﷫ |
| 220 | dc | 334 | 11011100 | ﷬ |
| 221 | dd | 335 | 11011101 | ﷭ |
| 222 | de | 336 | 11011110 | ﷮ |
| 223 | df | 337 | 11011111 | ﷯ |

| Dezimal | Hexadezimal | Oktal | Binär | ASCII |
|---|---|---|---|---|
| 224 | e0 | 340 | 11100000 | α |
| 225 | e1 | 341 | 11100001 | ß |
| 226 | e2 | 342 | 11100010 | Γ |
| 227 | e3 | 343 | 11100011 | π |
| 228 | e4 | 344 | 11100100 | Σ |
| 229 | e5 | 345 | 11100101 | σ |
| 230 | e6 | 346 | 11100110 | µ |
| 231 | e7 | 347 | 11100111 | τ |
| 232 | e8 | 350 | 11101000 | Φ |
| 233 | e9 | 351 | 11101001 | Θ |
| 234 | ea | 352 | 11101010 | Ω |
| 235 | eb | 353 | 11101011 | δ |
| 236 | ec | 354 | 11101100 | ∞ |
| 237 | ed | 355 | 11101101 | φ |
| 238 | ee | 356 | 11101110 | ε |
| 239 | ef | 357 | 11101111 | ∩ |
| 240 | f0 | 360 | 11110000 | ≡ |
| 241 | f1 | 361 | 11110001 | ± |
| 242 | f2 | 362 | 11110010 | ≥ |
| 243 | f3 | 363 | 11110011 | ≤ |
| 244 | f4 | 364 | 11110100 | ⌠ |
| 245 | f5 | 365 | 11110101 | ⌡ |
| 246 | f6 | 366 | 11110110 | ÷ |
| 247 | f7 | 367 | 11110111 | ≈ |
| 248 | f8 | 370 | 11111000 | ° |
| 249 | f9 | 371 | 11111001 | ∙ |
| 250 | fa | 372 | 11111010 | · |
| 251 | fb | 373 | 11111011 | √ |
| 252 | fc | 374 | 11111100 | n |
| 253 | fd | 375 | 11111101 | 2 |
| 254 | fe | 376 | 11111110 | 3 |
| 255 | ff | 377 | 11111111 | |

## Stichwortverzeichnis

# Spitzen-Software für ATARI-ST-Computer

## WordStar 3.0 mit MailMerge

Der Bestseller unter den Textverarbeitungsprogrammen bietet Ihnen bildschirmorientierte Formatierung, deutschen Zeichensatz und DIN-Tastatur sowie integrierte Hilfstexte. Mit MailMerge können Sie Serienbriefe mit persönlicher Anrede an eine beliebige Anzahl von Adressen schreiben und auch die Adreßaufkleber drucken.

**Jetzt gibt es WordStar/MailMerge für den ATARI ST! Damit eröffnen sich Ihnen alle Möglichkeiten, Ihren ATARI ST für professionelle Textverarbeitung einzusetzen. Zum Superpreis!**

WordStar für die ATARI-ST-Computer wird auf zwei 3½-Zoll-Disketten geliefert. Sie beinhalten:
- CP/M-Z80-Emulator
- WordStar/MailMerge-Dateien

Hardware-Anforderungen: ATARI-ST-Computer, 80-Zeichen-Monitor, ein 3½-Zoll-Diskettenlaufwerk, beliebiger Drucker mit Centronics-Schnittstelle.
WordStar ist an den ATARI ST bereits fertig angepaßt und läßt sich bequem über Funktionstasten steuern.

**Bestell-Nr. MS 106**
**Für sagenhafte DM 199,– ***
**(sFr. 178,–/öS 1890,–*)**
*** inkl. MwSt. Unverbindliche Preisempfehlung**

Markt&Technik
ATARI ST-Software

MicroPro
WordStar 3.0
mit MailMerge für die ATARI ST-Computer

3½"-Format

## Und dazu die weiterführende Literatur:

**WordStar für ATARI-ST**

Mit diesem Buch haben Sie eine wertvolle Ergänzung zum WordStar-Handbuch: Anhand vieler Beispiele steigen Sie mühelos in die Praxis der Textverarbeitung mit **WordStar** ein. Angefangen beim einfachen Brief bis hin zur umfangreichen Manuskripterstellung zeigt Ihnen dieses Buch auch, wie Sie mit Hilfe von MailMerge Serienbriefe an eine beliebige Anzahl von Adressen mit persönlicher Anrede senden können.

**Best.-Nr. MT 90208**
**ISBN 3-89090-208-1**

**DM 49,–**

**Erhältlich bei Ihrem Buchhändler.**

---

**Markt&Technik-Softwareprodukte erhalten Sie in den Computer-Abteilungen der Kaufhäuser und im Computershop.**

Markt&Technik

**Unternehmensbereich Buchverlag**
**Hans-Pinsel-Straße 2, 8013 Haar bei München**

# Bücher für ATARI-ST-Computer

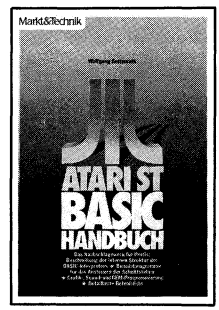

P. Rosenbeck

**C-Programmierung unter GEM/ATARI ST**

**3. Quartal 1986, ca. 300 Seiten**

GEM, die Benutzeroberfläche der ATARI-ST-Computer, gilt als außerordentlich bedienerfreundlich. Sie vereinigt herausragende grafische Darstellung und selbsterklärende, symbolische Benutzerführung auf einem Computer, der nach dem herkömmlichen Schema dem Homecomputer-Bereich zuzuordnen ist. Erst eine mächtige Programmiersprache wie C, die offizielle Entwicklungssprache von ATARI, ist GEM »gewachsen«.

**Best.-Nr. MT 90203**
**ISBN 3-89090-203-0**
**DM 58,–/sFr. 53,40/öS 452,40**



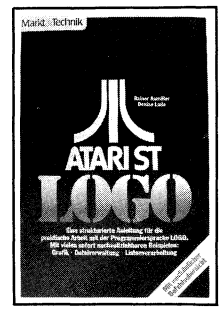

W. F. Fastenrath

**ATARI-ST-BASIC-Handbuch**

**März 1986, 264 Seiten**

Das BASIC für die ATARI-ST-Computer ist außerordentlich umfangreich und mächtig. Über 130 Befehle stehen bereit, um auch komplexere Aufgaben mit diesen Computern zu bewältigen.
Dieses Buch beabsichtigt nicht eine allgemeine Einführung in die Techniken der BASIC-Programmierung. Es stellt vielmehr eine Anleitung zur Anwendung von BASIC auf die Erfordernisse und Möglichkeiten dieses speziellen Systems dar. Eine übersichtliche Zusammenstellung des gesamten Befehlsvorrats macht dieses Buch zu einem Hilfsbuch bei der täglichen Programmierarbeit.

**Best.-Nr. MT 90205**
**ISBN 3-89090-205-7**
**DM 52,–/sFr. 47,80/öS 405,60**




R. Aumiller/D. Luda

**ATARI-ST-LOGO**

**März 1986, 236 Seiten**

LOGO – einfach wie BASIC, jedoch so leistungsstark wie Pascal oder FORTH. LOGO vereinigt viele Vorteile anderer Programmiersprachen in sich. Es ist interaktiv, listen- und prozedurorientiert, erweiterbar, einfach zu erlernen und doch komplexen Problemen gewachsen.
Dieses Buch ist für Anfänger und Fortgeschrittene gleichermaßen geeignet. Bildschirmfotos, viele ausführliche Beispiele – teilweise mit Übungsaufgaben zur Vertiefung des Gelesenen – tragen zu einer guten Verständlichkeit und einem sicheren Lernerfolg bei.

**Best.-Nr. MT 90223**
**ISBN 3-89090-223-5**
**DM 49,–/sFr. 45,10/öS 382,20**

---

**Markt & Technik-Fachbücher erhalten Sie bei Ihrem Buchhändler**

Markt&Technik

**Unternehmensbereich Buchverlag**

**Hans-Pinsel-Straße 2, 8013 Haar bei München**

# Bücher für ATARI-ST-Computer


I. Lüke/P. Lüke

**Das Systemhandbuch zum ATARI ST**

**3. Quartal 1986, ca. 300 Seiten**

Zwei Themen bilden die Schwerpunkte des vorliegenden Buches: Die Struktur der 68000-CPU und der ATARI 520/260 ST. Die ausführliche Beschreibung der Architektur der 68000-Familie (68000, 68008, 68010, 68020) und ihres Befehlssatzes wird ergänzt durch einen Nachschlageteil mit zwei- bis dreizeiligen Beispielsequenzen. Auf dieser theoretischen Basis wird die Programmierumgebung des ATARI 520/260ST anhand vieler Beispielprogramme dargestellt. Die Besprechung eines 68000-Assemblers und einige gerätespezifische Maschinensprachmodule runden das Buch ab.

**Best.-Nr. MT 90216**
**ISBN 3-89090-216-2**
**DM 52,–/sFr. 47,80/öS 405,60**


I. Lüke/P. Lüke

**Der ATARI 520 ST**

**2. überarbeitete und erweiterte Auflage 1986, 198 Seiten**

Dieses Buch enthält alle Informationen, die für Interessierte und für alle stolzen Besitzer eines gerade erworbenen ATARI 520/260 ST wichtig sind. Die jetzt vorliegende überarbeitete und erweiterte Auflage trägt den neuesten Entwicklungen bei ATARI Rechnung. Unter anderem wurden das inzwischen deutschsprachige Betriebssystem und einige geänderte Systemausstattungsmerkmale berücksichtigt. Das Buch ist somit nicht nur eine Rechnerbeschreibung mit hohem Informationswert, es leistet auch als Nachschlagewerk wertvolle Dienste.

**Best.-Nr. MT 90229**
**ISBN 3-89090-229-4**
**DM 49,–/sFr. 45,10/öS 382,20**


A. Steiner/G. Steiner

**GEM für den ATARI 520 ST**

**2. überarbeitete und erweiterte Auflage 1986, 334 Seiten**

Die Benutzeroberfläche des neuen ATARI ST – GEM genannt – erhebt den Anspruch, die Bedienung des Computers zum Kinderspiel zu machen. Dennoch: Wenn Sie die bisher übliche kommandoorientierte Umgangsweise mit Ihrem Computer pflegten, so werden Sie eine Einführung in die Bedienung von Maus, Bildsymbolen und Fenster, wie sie dieses Buch liefert, zu schätzen wissen. Besonders interessant für den erfahrenen Anwender sind die Kapitel über den internen Aufbau von GEM mit seinen Pull-Down-Menüs, Fenstern und Symbolen.

**Best.-Nr. MT 90230**
**ISBN 3-89090-230-8**
**DM 52,–/sFr. 47,80/öS 405,60**

---

**Markt & Technik-Fachbücher erhalten Sie bei Ihrem Buchhändler**

Markt&Technik

**Unternehmensbereich Buchverlag**

**Hans-Pinsel-Straße 2, 8013 Haar bei München**

# Bücher für ATARI-ST-Computer

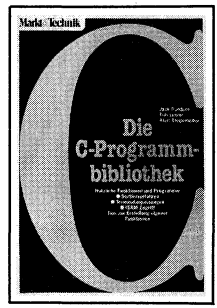

P. Rosenbeck

**C-Programmierung unter TOS/ATARI ST**

**März 1986, 376 Seiten**

Erst durch das Programmieren in C kann der stolze Besitzer alle Fähigkeiten seines ATARI ST ausnutzen. Für Leser mit elementaren EDV-Vorkenntnissen gibt der Autor in diesem Buch eine gründliche und leicht lesbare Einführung in das Programmieren mit dieser wichtigen und vielseitigen Sprache. An aussagekräftigen und in allen Einzelheiten erklärten Beispielen werden auch die fortgeschrittenen Aspekte der Sprache besprochen.

**Best.-Nr. MT 90226**
**ISBN 3-89090-226-X**
**DM 52,–/sFr.47,80/öS 405,60**



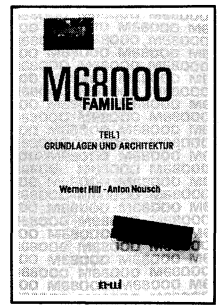

J. Purdum/T. Leslie

**Die C-Programmbibliothek**

**Februar 1986, 361 Seiten**

Dieses Buch erspart dem C-Programmierer Stunden mühseliger Kleinarbeit und hilft, effizientere Programme zu schreiben. Es ist in zwei Teile gegliedert. Der erste Teil zeigt, wie man zu universellen Bibliotheksfunktionen kommt, und gibt Tips, wie C noch wirkungsvoller eingesetzt werden kann. Der zweite Teil enthält eine Reihe ausführlich erklärter C-Funktionen als wertvolle Ergänzung Ihrer Programmbibliothek. Dazu gehören unter anderem ein Terminalinstallationsprogramm, mehrere Sortier-Algorithmen und ein Satz ISAM-Funktionen.

**Best.-Nr. MT 90133**
**ISBN 3-89090-133-6**
**DM 69,–/sFr. 63,50/öS 538,20**




W. Hilf/A. Nausch

**M68000-Familie: Teil 1**

**1984, 568 Seiten**

Informative Einführung in die Geschichte und die Entwicklungsphilosophie einer detaillierten Darstellung der Hardware sowie ausführliche Erläuterung der komfortablen Adressierungsarten.

**Best.-Nr. PW 80316**
**ISBN 3-921803-16-0**
**DM 79,–/sFr. 72,80/öS 616,20**

**M68000-Familie: Teil 2**

**1985, 400 Seiten**

Teil 2 des umfassenden Lehr- und Nachschlagewerks zum M68000 beschäftigt sich mit Anwendungen und weiteren Mitgliedern der M68000-Familie.

**Best.-Nr. PW 80330**
**ISBN 3-921803-30-6**
**DM 69,–/sFr. 63,50/öS 538,20**

---

**Markt & Technik-Fachbücher erhalten Sie bei Ihrem Buchhändler**

Markt&Technik

**Unternehmensbereich Buchverlag**

**Hans-Pinsel-Straße 2, 8013 Haar bei München**

# C-Programmierung unter TOS ATARI ST

Mit den Computern der ST-Reihe von ATARI eröffnen sich für Homecomputer neue Dimensionen. Die Grenze zu den Personal Computern – in der letzten Zeit ohnehin im Wanken begriffen – wurde endgültig niedergerissen, der Leistungsabstand zu den Mini-Computern wurde geringer; kurz: die Systeme wurden professioneller. Dies bringt jedoch auch eine wachsende Komplexität mit sich. An der Oberfläche merkt der Benutzer davon nichts: GEM ist die Freundlichkeit in Person.

Doch wer sich an die Programmierung dieses Computers heranwagt und einen Blick hinter die Kulissen wirft, der wird bald feststellen, daß dieses System hohe Anforderungen an ihn stellt. Der erstaunliche Leistungsumfang der ST-Computer läßt sich nämlich nur mit einer mächtigen Programmiersprache wie C in den Griff bekommen.

C ist deshalb auch offizielle Entwicklungssprache von ATARI. Damit ist dieser Computer der erste, auf dem einer auch im professionellen Bereich weitverbreiteten Sprache der Vorzug vor der Einsteiger- und Hobbyistensprache BASIC gegeben wurde. Dies ist eine Herausforderung für den Hobbyisten, bringt ihn mit seinen Fähigkeiten jedoch auch näher an den Arbeitsmarkt, auf dem C viel stärker gefragt ist als BASIC.

Für Leser mit elementaren EDV-Vorkenntnissen stellt dieses Buch eine gründliche und leicht lesbare Einführung in das Programmieren mit dieser wichtigen und vielseitigen Sprache dar. An aussagekräftigen und in allen Einzelheiten erklärten Beispielen werden auch die fortgeschrittenen Aspekte der Sprache (Dateiverwaltung, Structures, dynamische Speicherverwaltung, Rekursion) ebenso ausführlich wie die Grundlagen des Arbeitens in C besprochen, so daß man auch nach längerer Beschäftigung mit dieser faszinierenden Sprache immer wieder wertvolle Anregungen für die praktische Arbeit finden wird. Der Autor versäumt es nicht, den Leser in den typischen Jargon der C-Programmierer einzuweihen und ihm wichtige Maßregeln für einen – in C besonders wichtigen – guten Programmierstil mit auf den Weg zu geben.

Besonderes Gewicht wurde dem Programmieren auf Systemebene eingeräumt (Schnittstelle zum Betriebssystem TOS, Benutzung von GEMDOS, BIOS und XBIOS), so daß der Leser in der Lage ist, auch systemnahe Programme auf seinem ATARI ST zu erarbeiten.

Die Technik des systemnahen, aber auch des professionellen strukturierten Programmierens wird dem Leser an einem umfangreichen und ausführlich kommentierten Beispiel – einem Diskettenmonitor – vorgeführt. Eine nicht nur lehrreiche, sondern auch für die praktische Arbeit nützliche Anwendung.

Hard- und Software-Anforderungen:

- ATARI-ST-Computer
- C-Compiler

ISB N 3-89090-226-X

Markt&Technik

4 001057 902268

DM 52,-
sFr. 47,80
öS 405,60