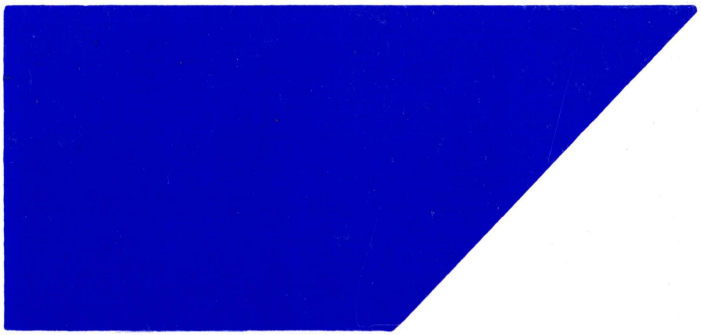
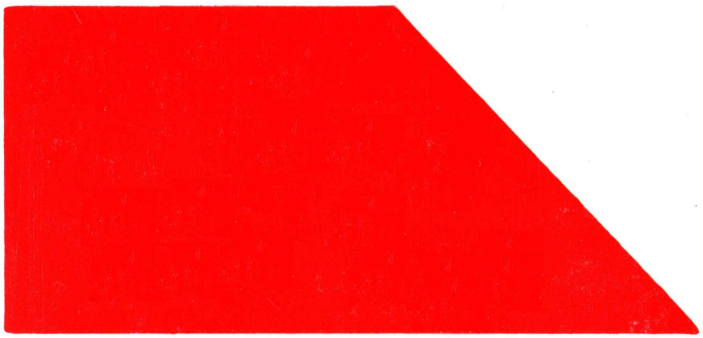
David Lawrence



Commodore Sachbuchreihe Band 9



PROGRAMMIERTECHNIKEN FÜR FORTGESCHRITTENE AUF DEM COMMODORE 64

Wertvolle Ideen und Anwendungen anwendbar auch auf dem Commodore 128

David Lawrence

PROGRAMMIERTECHNIKEN FÜR FORTGESCHRITTENE AUF DEM COMMODORE 64

Commodore Sachbuchreihe Band 9

David Lawrence

PROGRAMMIERTECHNIKEN FÜR FORTGESCHRITTENE AUF DEM COMMODORE 64

Wertvolle Ideen und Anwendungen anwendbar auch auf dem Commodore 128

Commodore

Titel der Originalausgabe: Advanced programming techniques on the Commodore 64

Sunshine Books (an imprint of Scot Press Ltd.)
12–13 Little Newport Street, London WC2R 3LD

Aus dem Englischen übertragen von Brigitte Pohl.

Artikel-Nr. 556 435/4.85 Änderungen vorbehalten. ISBN-NR. 3-89133-009-X

INHALT

Steuerzeichen in Programmlistings

Die in den Programmzeilen verwendeten Steuerzeichen haben folgende Bedeutung (vgl. auch Anhang F des Commodore-64-Bedienungshandbuchs):

BEZEICHNUNG	SYMBOL	TASTEN	ODER	CHR$(XXX)
[SCHIRM NEU]	"□"	= <SHIFT>+<CLR>		CHR$(147)
[CRSR HOCH]	"□"	= <SHIFT>+<CRSR>		CHR$(145)
[CRSR RUNTER]	"□"	= <CRSR>		CHR$(17)
[CRSR LINKS]	"□"	= <SHIFT>+<CRSR>		CHR$(157)
[CRSR RECHTS]	"□"	= <CRSR>		CHR$(29)
[SCHWARZ]	"□"	= <CTRL>+<1>		CHR$(144)
[WEISS]	"□"	= <CTRL>+<2>		CHR$(5)
[ROT]	"□"	= <CTRL>+<3>		CHR$(28)
[CYAN]	"□"	= <CTRL>+<4>		CHR$(159)
[PURPUR]	"□"	= <CTRL>+<5>		CHR$(156)
[GRUEN]	"□"	= <CTRL>+<6>		CHR$(30)
[BLAU]	"□"	= <CTRL>+<7>		CHR$(31)
[GELB]	"□"	= <CTRL>+<8>		CHR$(158)
[REVERS EIN]	"□"	= <CTRL>+<9>		CHR$(18)
[REVERS AUS]	"□"	= <CTRL>+<0>		CHR$(146)
[ORANGE]	"□"	= <C=>+<1>		CHR$(129)
[BRAUN]	"□"	= <C=>+<2>		CHR$(149)
[HELLROT]	"□"	= <C=>+<3>		CHR$(150)
[GRAU 1]	"□"	= <C=>+<4>		CHR$(151)
[GRAU 2]	"□"	= <C=>+<5>		CHR$(152)
[HELLGRUEN]	"□"	= <C=>+<6>		CHR$(153)
[HELLBLAU]	"□"	= <C=>+<7>		CHR$(154)
[GRAU 3]	"□"	= <C=>+<8>		CHR$(155)

<CTRL> BEDEUTET 'CONTROL-TASTE'.
<C=> BEDEUTET 'COMMODORE-TASTE'.

EINLEITUNG

Dieses Buch soll sich von allen anderen in Ihrem Bücherregal unterscheiden. Es stellt keine Programmsammlung dar, keine Einführung in die verfügbaren BASIC-Befehle und auch keine Sammlung üblicher Routinen, wie z. B. ein zweizeiliges Unterprogramm zur Umrechnung von Fahrenheit in Celsius.

Es ist vielmehr den Lesern gewidmet, die ihren Mikrocomputer für nützliche Aufgaben einsetzen wollen – die Art von Aufgaben, die man normalerweise 'Anwendungsprogramme' nennt. Die Zeiten, in denen man sich damit begnügte, die Leistungsfähigkeit eines Mikrocomputers nur für Spiele zu nutzen, sind hoffentlich endgültig vorbei. Die Leute *wollen* mit den Mikrocomputern arbeiten. Die Frage ist nur: Wie schreibt man ein geeignetes Programm?

Die Legenden, die sich um anwendungsbezogenes Programmieren gebildet haben, sind meistens Unsinn. Einfach gesagt, ist ein Anwendungsprogramm etwas, das die Eingabe von Informationen ermöglicht, sie speichert, verarbeitet und zu irgendeinem Zweck wieder ausgibt. Die Information kann aus Namen und Adressen bestehen, Waren und Preisen, Finanzunterlagen etc. – die Liste ist endlos. Ebenso unbegrenzt sind die Möglichkeiten der Informationsverarbeitung: Manches muß nur gespeichert, anderes nach bestimmten Kriterien sortiert werden, wieder anderes wird komplizierten mathematischen Verfahren unterworfen. Kein Buch kann den Anspruch haben, Anleitungen für alle möglichen Verarbeitungsweisen von Daten zu geben, die ein Mikrocomputer speichern kann. Die Art der Verarbeitung hängt von der Information und dem Zweck ab, zu dem sie gespeichert wird.

Einige Richtlinien für Verfahrensweisen im Rahmen eines Kernprogramms müssen hier genügen: wie man ein Programm so schreibt, daß es reibungslos funktioniert; wie man es austestet; wie man Daten möglichst ökonomisch und schnell speichert; wie man sortiert; wie man die Ausgabe so formatiert, daß sie klar und verständlich ist. Solche Dinge werden den Löwenanteil der Programmierarbeit für jeden ernsthaften Anwendungszweck ausmachen, und dieses Buch soll erklären, wie man sie mit sparsamen Mitteln erfolgreich bewältigt.

Die Komplexität der behandelten Themen ist äußerst unterschiedlich. In den folgenden Kapiteln werden Sie viele Techniken vorfinden, die nur eine einzige Programmzeile erfordern, daneben zwei- oder dreizeilige Routinen für einfache Zwecke wie die Ausrichtung von Dezimalstellen bei einer Zahlenkolonne, aber auch längere und komplexere Routinen, mit denen man Daten sehr schnell in große Felder schreiben kann. Alle sind aufgeführt, weil sie nützlich sind, denn das vorliegende Buch ist kein theoretisches Werk. Ich habe versucht, sowohl eigene Techniken als auch die Arbeit anderer auf gemeinsame Aufgabenstellungen und Methoden hin zu untersuchen. Das Buch enthält nichts, das bei der Lösung eines bestimmten Problems nicht schon erprobt worden wäre.

Es befaßt sich nicht mit dem Einsatz von Mathematik beim Programmieren. Das ist ein Spezialgebiet, das mindestens ein eigenes Buch wert ist. Selbstverständlich enthält auch dieses Buch hier und da ein bißchen Mathematik, aber nur so viel, wie für ein betriebsfähiges Programm unbedingt nötig ist. Wenn Sie sich gründlicher mit dem mathematischen Bereich beschäftigen wollen, sollten Sie sich Czes Kosniowskis neues Buch für den C 64, *'Mathematik mit dem Commodore 64'*, (Commodore Sachbuchreihe, Band 8) anschaffen. Grafische Techniken läßt das vorliegende Buch ebenfalls außer acht, abgesehen von dem, was man für eine klare und übersichtliche Datengestaltung braucht. Dies ist ebenso ein eigenes Spezialthema.
Mein Dank geht an alle Leser, die mich zum Schreiben eines derartigen Buches ermutigt haben. Ich hoffe, es entspricht wenigstens zum Teil ihren Erwartungen. Dank gilt auch Mark England, Koautor von *'Programmierung in Maschinensprache auf dem Commodore 64'* (Commodore Sachbuchreihe, Band 12 und 13), der geduldig die Vollendung dieses Buches abgewartet hat, bevor wir unser nächstes gemeinsames angehen können. Schließlich bedanke ich mich ganz nachdrücklich bei Barny und Tom, weil sie Verständnis dafür hatten, daß es manchmal wichtiger sein kann, am Computer zu sitzen als zu spielen, und bei meiner Frau Jane, weil sie in dieser Zeit alle Sorgen allein zu tragen hatte.
Ich hoffe, das Resultat rechtfertigt diesen Aufwand. Ich hoffe, dies ist ein Buch, auf das Sie immer wieder zurückgreifen werden, weil es etwas Licht in die unvermeidlichen Probleme bringt, die sich beim Programmieren ergeben. Am meisten hoffe ich, daß es Sie zu neuen Ideen anregt. Es gibt Ihnen die Hilfsmittel zum Programmieren an die Hand, aber es hat seine Aufgabe noch nicht erfüllt, wenn Sie diese Hilfsmittel nur verstehen, sondern erst, wenn Sie sie für neue Zwecke einsetzen.

David Lawrence

Nach dem großen Erfolg, den David Lawrence mit seinem Buch in England gehabt hat, bringen wir auch für den deutschen Leser dieses Standardwerk heraus, das für das Programmieren nicht nur am Commodore 64 sondern ebenso auch am 'größeren Bruder', dem neuen Commodore 128, seine Aktualität behalten wird.

Commodore Verlagsabteilung
Sachbuchredaktion

KAPITEL 1

MODULARES PROGRAMMIEREN

WIE MAN EIN LAUFFÄHIGES PROGRAMM SCHREIBT

Es scheint vielleicht ungewöhnlich, ein Buch über BASIC-Programmiertechniken mit einem Kapitel anzufangen, das sich weniger mit BASIC als mit Problemen von Stil und Aufbau eines Programms beschäftigt. Es gibt jedoch einen einfachen Grund für dieses Kapitel: Da ich es mit Fragen und Problemen der Mikrocomputer-Besitzer zu tun habe, die ihre Programme selbst schreiben, sehe ich immer deutlicher, daß ihr größtes Problem oft nicht in mangelnder Kenntnis der Funktion von BASIC liegt, sondern in der unsystematischen Art, wie sie an die eigentliche Aufgabe herangehen: nämlich BASIC in einem Programm effizient einzusetzen.
Die Programmiertechnik, die ich selbst sowohl in Büchern als auch in eigenen Programmen benutze, heißt 'modulares Programmieren'. Zunächst bedeutet das nur, Programme zu schreiben, die aus selbständigen Bausteinen zusammengesetzt sind. Die meisten praktischen Beispiele im Buch sind in dieser Form geschrieben. Sie können direkt in jedes beliebige eigene Programm, wo nötig, eingebunden werden. Modulares Programmieren geht jedoch noch viel weiter. Es enthält die ganze Philosophie eines Programmierstils, und wenn das zu großartig klingt: Es soll weiter nichts heißen, als daß man seinen gesunden Menschenverstand beim Schreiben von Computerprogrammen einsetzt.
In diesem Kapitel werden wir einige Schritte erörtern, die zum Schreiben eines erfolgreichen Programms gehören und schon notwendig sind – oder sein sollten –, lange bevor Sie die Tastatur Ihres 64 zum ersten Mal berühren.

ABGRENZEN DES PROGRAMMS

Die schlechtesten Programme, die überhaupt geschrieben werden, sind die, von denen man schon vorher begeistert ist. Man hat eine Idee, stürzt sich voller Zuversicht auf die Tastatur und versucht, seine Vision in die Praxis umzusetzen. Wenn man sich auskennt, kann man in kurzer Zeit ein betriebsfähiges Kernprogramm erstellen. Dann aber merkt man, daß es z. B. keine Möglichkeit gibt, Daten ordentlich einzugeben, und so schiebt man das in irgendeine Lücke des Programms ein oder hängt zu diesem Zweck eine Routine an den Schluß. Dann fügt man noch etwas an, damit fehlerhafte Elemente im Bedarfsfall entfernt werden, und noch etwas zur Berücksichtigung ungültiger Eingaben, die das Programm zum Absturz

bringen würden (das passiert immer, wenn ein anderer es in die Finger bekommt). Zusätzlich braucht man natürlich noch ein paar Zeilen zum Speichern von Daten auf Band. Dann fällt einem ein, daß sonst niemand weiß, wie man mit dem Programm umgeht, also muß es um ein besseres 'Menü' und vielleicht ein paar Anweisungen erweitert werden . . .

Zum guten Schluß ist aus dem Programm ein Mischmasch aus ungeordneten Zeilennummern und verwickelten, in alle Richtungen weisenden GOTOs und GOSUBs geworden. Trifft man dann auf einen unvermeidlichen Fehler, ist die Art des Fehlers weniger problematisch herauszufinden als sein Ort. Das kann Tage oder gar Wochen dauern.

Die besten Programme entstehen dann, wenn man zwar weiß, daß man sie schreiben muß, aber eigentlich keine Lust dazu hat oder nicht genau weiß, wie es gemacht wird. Das hat einen einfachen Grund: Wenn man nicht sicher ist, ob man das gewünschte Programm schreiben kann, setzt man sich erst einmal hin und denkt über das Vorhaben nach. In diesem Moment hat man die erste und in vieler Hinsicht wichtigste Hürde beim Entwerfen eines erfolgreichen Programms genommen.

Ein Programm besteht nicht nur in dem Verfahren, mit dem seine zentrale Aufgabe gelöst wird, wie z. B. die Berechnung von Steuern oder das Speichern von Namens- und Adreßdateien. Es ist vielmehr ein ganzer Komplex von Funktionen: Es muß die Eingabe einer Information und ihre korrekte Wiedergabe bewältigen, klare Angaben über die Art seiner Bedienung machen, sich mit Fehlern befassen und Korrekturen zulassen, die nach der Eingabe fehlerhafter Informationen nötig sind. Diese und viele andere Aufgaben sind für ein brauchbares Programm genauso wesentlich wie die Routine im Programmkern.

Selbst wenn Sie sich nur auf die zentrale Aufgabe und die dafür notwendigen Zeilen beschränken, ist die erste Idee selten die beste. Sie wollen also Ihre Steuer ausrechnen; aber was soll alles in dieser Aufgabenstellung inbegriffen sein? Interessieren Sie die Ergebnisse bei jeweils unterschiedlichen Steuersätzen? Soll der Abzug von Unkosten berücksichtigt werden? Welchen Zeitraum wollen Sie abdecken, und, falls er über ein Jahr hinausgeht: Was passiert, wenn die Steuersätze sich von einem Jahr zum anderen (oder sogar innerhalb eines Jahres) ändern? Wollen Sie das Programm so anlegen, daß es 'Was, wenn'-Fragen bearbeitet (z. B. „Was würde passieren, wenn mein Einkommen monatlich um DM 78,70 stiege?"), ohne bereits eingegebene Daten zu beeinflussen? Möchten Sie die Informationen speichern oder jedesmal neu eingeben, wenn Sie das Programm laufen lassen? Soll das Gehalt des Ehepartners mit einbezogen werden? Alle diese Fragen fallen mir ganz spontan ein. Nach längerem Nachdenken könnte man eine ganze Seite mit alledem füllen, was man wissen muß, bevor man endlich mit dem Programmieren einer zunächst scheinbar völlig eindeutigen Aufgabe anfangen kann. Möglicherweise werden Sie auch dann ein funktionierendes Programm

zustandebringen, wenn Sie vorher die Anwendungsmöglichkeiten nicht gründlich durchdacht haben – aber es gibt überall schon mehr als genug überflüssige Kassetten mit funktionierenden Programmen, die im Grunde unbrauchbar sind.
Die erste Aufgabe beim Schreiben eines Programms besteht also darin, sich hinzusetzen und nachzudenken und in klaren Worten aufzuschreiben, was das Programm Ihrer Vorstellung gemäß leisten soll. Wenn das erledigt ist, lassen Sie es eine Zeitlang liegen, bevor Sie sich wieder damit beschäftigen. Bei nochmaliger Prüfung wird es selten so perfekt aussehen wie vorher, und Sie werden es um einige Funktionen erweitern wollen. Wird ein Programm von anderen benutzt (und sei es noch so selten), müssen Sie deren Erwartungen berücksichtigen. Es ist absolut zwecklos, Ihre Kinder davon überzeugen zu wollen, wie spannend und unterhaltsam das ausgezeichnete neue Lernprogramm ist, das Sie geschrieben haben – solange es nicht das leistet, was Ihre Kinder gern hätten. Entweder tut das Programm, was die Benutzer möchten und erwarten, oder nicht. Wenn nicht, haben Sie mit dem Schreiben eine Menge Zeit verschwendet.
Daraus folgt, daß Sie bei der Funktionsbestimmung Ihres Programms nie zu weit gehen können. Sie sollten bei der Planung weit mehr als die bloßen Notwendigkeiten berücksichtigen. Wenn Sie mit dem ausführlichen Planen und Schreiben beginnen, müssen Sie vielleicht einige Ideen wieder aufgeben, weil die Speicherkapazität oder aber Ihre Kenntnisse dazu nicht ausreichen. Viel öfter werden Sie sich jedoch in der Lage sehen, ein Programm tatsächlich nach Ihren Vorstellungen schreiben zu können. Gerade auf die Programme, die ein Gebiet vollständig abdecken, werden Sie immer wieder zurückgreifen.

PLANEN DES PROGRAMMS

Planen? Hatten wir das nicht schon? Eben nicht, denn bis jetzt haben Sie nicht einmal über ein Computerprogramm nachgedacht. Sie haben lediglich den idealen Sklaven beschrieben, der eine Arbeit genauso erledigt, wie Sie es sich vorstellen. Wenn Sie die Aufgabe richtig angefaßt haben, dann haben Sie bis jetzt keine Rücksicht darauf genommen, ob Sie sie mit dem C 64 realisieren können, oder was man überhaupt damit machen kann. Ihre Aufgabe ist es jetzt, die ideale Funktionsbeschreibung in Bausteine zu zerlegen, die der C 64 verarbeiten kann, und die Sie programmieren können.
Dabei geht man am besten in zwei Stufen vor. Im ersten Stadium werden allgemeine Programmbereiche wie Eingabe, Ausgabe, Datenverarbeitung, Speicher u. a. bestimmt. Wenn diese ziemlich einfache Angelegenheit erledigt ist, kommt der komplizierte Teil, in dem die eigentlichen Bausteine oder Module bestimmt werden, aus denen das Programm aufgebaut ist.
In der Regel unterteilt man dazu die Programmfunktionen so weit wie möglich in

getrennte Einheiten, selbst wenn man dabei Funktionen teilen muß, die sonst scheinbar immer zusammengehören. Ein streng in Funktionseinheiten gegliedertes Programm ist immer am leichtesten zu schreiben und am unkompliziertesten, wenn die unvermeidliche Suche nach Fehlern beginnt. Außerdem sind Programme, die in klare Funktionseinheiten eingeteilt sind, paradoxerweise oft kürzer als solche, in denen alle Programmfunktionen eng gebündelt sind. Der Grund dafür ist folgender: Wenn Sie ein Programm konsequent zerlegen, stellen Sie fest, daß bestimmte Funktionen in verschiedenen Teilen und an verschiedenen Stellen des Programms immer wieder gebraucht werden.

Nehmen wir als Beispiel unser hypothetisches Programm zur Berechnung von Steuern: Die Anzeige der Daten auf dem Bildschirm erfordert eine kurze Routine, damit Zahlen mit zwei Dezimalstellen und in einer Kolonne mit untereinander ausgerichteten Dezimalstellen erscheinen (siehe Kapitel 13). Sie könnten diese drei- bis vierzeiligen Routinen einfach an die Ausgabeeinheit des Programms hängen. Aber dann kommen Sie beim Löschen einzelner Elemente darauf, daß sie in demselben Format angezeigt werden sollten, damit der Benutzer genau verfolgen kann, was gelöscht wird. Dasselbe gilt für die Eingabe, die Sie gern formatiert auf dem Bildschirm sähen. Da Sie aber die kurze Formatierroutine in einem anderen Programmteil untergebracht haben, ist es unmöglich, von einer anderen Stelle aus darauf zuzugreifen, und daher müssen Sie die Routine bei Bedarf jedesmal neu schreiben.

Das klingt vielleicht lächerlich, aber werfen Sie nur einen Blick auf die Programme, die im Anhang von Computerzeitschriften veröffentlicht werden. Überall werden Sie solche Wiederholungen entdecken. In einem ordentlich entworfenen Programm kann auf jede Funktion von jeder Stelle aus zugegriffen werden. Sehr oft werden Sie sogar entdecken, daß ein sorgfältig geschriebenes Programm um zusätzliche Funktionen erweitert werden kann, indem man eine Anzahl von Funktionen einfach in einer anderen Reihenfolge aufruft.

Auf lange Sicht ist es am wichtigsten, daß das Schreiben weiterer Programme enorm erleichtert wird, wenn Sie erst einmal über drei oder vier Programme mit klar abgegrenzten Funktionen verfügen. Es gibt in Wirklichkeit nur sehr wenige Techniken, die für ernsthaftes Programmieren wichtig sind. Wenn Sie diese einmal in einem Programm angewendet haben, werden Sie schnell dazu übergehen, sie nach dem Baukastenprinzip für die Erstellung anderer Programme auszuwählen und zu benutzen.

Die grundlegende Technik für ein sorgfältig strukturiertes Programm besteht darin, sich jeden Aspekt des Programms zu beschreiben:

"In diesem Abschnitt möchte ich dem Benutzer erläutern, wie Daten einzugeben sind, dann die Daten annehmen, sie auf bestimmte Fehler hin überprüfen, den Benutzer auffordern, die Richtigkeit der Daten zu bestätigen, und schließlich die Daten an den richtigen Platz speichern."

In diesem einfachen und relativ untechnischen Satz haben Sie bereits mindestens fünf Funktionen unterschieden, die Sie sofort notieren sollten. Arbeiten Sie sich auf die gleiche Weise durch alle allgemeinen Programmteile. Als Resultat ergibt sich eine Liste von Funktionen, die offenbar für ein Programm ausreichen.
Jetzt geht man daran, die weiter zerlegbaren Funktionen zu bestimmen. Wir haben z. B. oben erwähnt, daß die Eingabe noch einmal auf dem Bildschirm angezeigt werden soll, etwa damit der Benutzer die Korrektheit der Eingabe bestätigen kann. Zum Formatieren der Anzeige braucht man wie gesagt eine Formatierroutine. Ein weiteres Beispiel wäre die letztgenannte Funktion: das Speichern der Daten an den richtigen Platz. Wenn Sie den Inhalt alphabetisch oder chronologisch geordnet haben, müßte beim Einfügen eines neuen Datenelements zuerst die richtige Stelle gesucht und dann Platz dafür gemacht werden. Daß es sich dabei um zwei verschiedene Prozesse handelt, merken Sie spätestens dann, wenn Sie dem Benutzer die Möglichkeit geben wollen, ein bestimmtes Element zu benennen, das aufgerufen und angezeigt werden soll. Dann werden Sie nämlich feststellen, daß Ihre Suchroutine und die Zeilen zum Einschieben eines Elements aneinandergekoppelt sind.
Wie bei den allgemeinen Programmteilen gehen Sie bei jeder einzelnen Funktion des Programms vor und beschreiben sie. Sollten Sie bei einer Funktion auf zwei oder drei verschiedene Schritte kommen, so ist sie wahrscheinlich zur nochmaligen Zerlegung geeignet.
Am Ende der Prozedur haben Sie eine Liste von Funktionen, die der C 64 nach Ihrer Ansicht zur Ausführung des geplanten Programms benötigt. Sie können sich ein Bild von dem gesamten Programm, der ungefähren Länge, der Art der Struktur und den möglicherweise problematischen Bereichen machen. Gehen Sie am Schluß zur Probe ein paar typische Operationen durch, die Sie mit Ihrem Programm ausführen möchten, und benutzen Sie dabei Ihre Notizen statt den C 64:

"Anschalten, RUN (aha, ein Menü zur Funktionsbeschreibung fehlt), anzeigen, daß ich Daten eingeben will, Eingabe bestätigen, das nächste Element eingeben . . ."

Wenn alles in Ordnung ist, sollten Ihre Notizen fast genauso gut funktionieren wie später der C 64.

SCHREIBEN DER MODULE

Inzwischen können Sie der Versuchung kaum noch widerstehen, sich endlich an den C 64 zu setzen und loszulegen, aber es ist noch immer nicht soweit. Ein Programm schreibt man am besten überhaupt nicht am Computer, sondern auf einem Blatt Papier. Damit will ich nicht sagen, daß Sie erst jede Zeile des BASIC-Programms genau aufschreiben sollen, bevor Sie mit der Eingabe beginnen kön-

nen. Vielmehr sollen Sie in diesem Stadium von jeder einzelnen Programmfunktion ein Arbeitsmodell entwerfen. An diese Modelle, die sicher nicht sehr detailliert sind, können Sie sich halten, wenn die einzelnen Funktionen schließlich als Programmmodule eingegeben werden.
Das bekannteste Verfahren zur schrittweisen Entwicklung von Programmen ist wohl das Flußdiagramm. Das Problem dabei ist allerdings, daß die Besitzer von Mikrocomputern sie kaum einsetzen, auch wenn Computerprofis ihren Gebrauch sehr empfehlen. Es ist wohl realistischer, die sogenannte 'Programm-Entwicklungssprache' (PDL; program development language) anzuwenden. Das klingt vielleicht etwas entmutigend, und tatsächlich hat sich PDL in den Händen professioneller Programmierer zu einem technischen Instrument entwickelt, das die meisten BASIC-Programmierer kaum verstehen, geschweige denn benutzen. Auf so komplizierte Dinge will ich aber hier nicht hinaus, sondern ich meine mit PDL eine einfache Mischung aus BASIC und Umgangssprache, die dem fertigen Modul zwar ähnelt, aber viel schneller zu schreiben ist.
Einen Eingabemodul könnte man dann etwa so schreiben:

```
## PRINT [TITLE]

   PRINT [VERFUEGBARE KOMMANDOS] NEU/ENDE

%% INPUT"WERT";TA

   IF TA=-9999 THEN RETURN

   GOSUB ERROR CHECK

   IF ERROR THEN %%

   INPUT "BESCHREIBUNG";TD$

   GOSUB FORMAT

   KLAR AB MENUE

   PRINT TD$;":";FRMAT$

   INPUT"KORREKT";Q$:"N"THEN100

   GOSUB PLATZSUCHE
```

```
DATENEINGABE

GOTO ##

[VARIABLEN ERFORDERLICH: KEINE]
```

VERWENDETE VARIABLEN:
TA = Zwischenspeicher für den Wert
TD$ = Zwischenspeicher für die Beschreibung
Q$ = Eingabevariable
FRMAT$ = durch Formatierroutine formatierter Wert

Wenn das Modul in dieser Form aufgeschrieben ist, sehe ich genau, welche Bestimmung es haben soll. Ich kann mir genau vorstellen, wie ich es programmieren werden. Wie Sie sehen, habe ich einzelne Anweisungen ausgeschrieben; denn das ist genauso kurz wie die verbale Umschreibung. Andere Funktionen, insbesondere im Zusammenhang mit der Bildschirmanzeige, werden zunächst nur erwähnt; es ist einfacher, erst bei der endgültigen Eingabe des Moduls zu entscheiden, wie viele Bildschirmzeilen für die Neueingabe eines Elements freigemacht werden müssen, oder in welcher Farbe Titel und Menü gedruckt werden sollen. Manchmal ist es günstig, ein Verfahren mit einer Zeile zu beschreiben, wenn man sich im Moment über die genaue Form noch nicht im klaren ist, aber weiß, daß das Problem mit etwas Nachdenken und Probieren gelöst werden kann. In diesen Fällen versieht man die entsprechende Zeile mit einer Bemerkung und schreibt später ein Ergänzungsblatt.
Die Zeilen sind nicht numeriert – teils, weil ich darauf verzichtet habe, jede Zeile in der endgültigen Form auszuschreiben, und teils, weil ich dadurch beim Schreiben aufgehalten werden würde. Ein oder zwei Markierungen am Zeilenanfang wie ## und %% sind ausreichend. Wenn ich das Modul später in den C 64 eingebe, schreibe ich GOTO***** für Vorwärtssprünge und aktualisiere die Zeile, sobald ich die Nummer der Zieladresse kenne. Rückwärtssprünge sind problemlos, weil die Sprungadresse des Ziels schon bekannt ist. Wie Sie sehen, sind auch die GOSUB-Adressen noch nicht bestimmt, sondern tragen nur die Namen anderer, vielleicht noch gar nicht geschriebener Unterprogramme. Bei der späteren Programmeingabe weise ich zunächst jedem Modul einen Zeilenblock von jeweils mindestens 1000 Zeilen zu; die Adresse vermerke ich oben auf den einzelnen Seiten mit den Beschreibungen der Programmfunktionen.
Am Ende des Listings sehen Sie eine Liste der verwendeten Variablen. Manche davon dienen mir als Hinweis auf die Variablen, die vor Aufruf des Moduls definiert werden müssen. Sie werden auch für eine vollständige Aufstellung der Variablennamen benötigt, anhand derer ich mich vergewissern kann, daß kein Name zweimal

vergeben wird. Hilfsvariablen, die nur für die Dauer des Moduls gebraucht werden, und deren Inhalt entweder vergessen oder zur Dauerspeicherung an eine andere Variable übergeben wird, dürfen Namen haben, die auch in anderen Modulen vorkommen. Andere würden Verwirrung stiften, wenn man sie versehentlich an verschiedenen Stellen für verschiedene Zwecke benutzte.
Natürlich kommt man mit dieser schlichten Methode nicht bei jedem Modul weiter. Einige Module enthalten vielleicht zwei bis drei Zeilen mit mathematischen Berechnungen; diese Zeilen würde ich schon explizit aufschreiben. Ihren Bedürfnissen frei angepaßt, verhilft Ihnen diese Methode aber im allgemeinen dazu, sich ein klares Bild des Programms zu machen und es dann viel schneller eingeben zu können, als wenn Sie es direkt an der Tastatur versucht hätten. Sie sollen nicht etwa streng nach dem Muster des o. g. Beispiels vorgehen, sondern daraus nur erkennen, daß ein Programm schnell und verständlich geschrieben werden *kann,* wenn Sie sich eine solche Methode zu eigen machen.

EINGEBEN DES PROGRAMMS

Jetzt können Sie sich endlich dem C 64 in der Gewißheit zuwenden, daß Sie etwas in der Hand haben, das Sie eingeben können und das einem betriebsfähigen Programm sehr nahe kommt. Als Rohmaterial dient Ihnen ein Stapel von Notizen mit den verschiedenen Programmfunktionen. Sie sollten jedoch keinesfalls das Programm einfach von Anfang bis Ende eingeben.
Modulares Programmieren eignet sich vorzüglich zur Fehlerbeseitigung während der Programmeingabe. Es ist viel leichter, ein Programm zu korrigieren, während Sie die einzelnen Module eingeben, als später Fehler aufzuspüren, wenn das ganze Programm schon läuft. In diesem Stadium kann die Fehlerbeseitigung zwar noch nicht perfekt sein; denn viele Fehler werden erst dann sichtbar, wenn alle Module zusammen arbeiten, aber Sie ersparen sich damit trotzdem viel Kopfzerbrechen. Um ein Programm laufend zu korrigieren, müssen Sie die Module bestimmen, die am häufigsten gebraucht werden (wahrscheinlich für triviale Zwecke), und diese zuerst eingeben.
Das oben abgedruckte Eingabemodul könnte man z. B. auch ohne die PLATZ-SUCHE oder DATENEINGABE-Routinen austesten. Diese ließen sich einfach durch zwei RETURN-Befehle in den passenden Anfangszeilen derjenigen Blöcke ersetzen, in denen sie später stehen sollen. Das Eigabemodul können Sie dagegen nicht austesten, solange die ERROR CHECK- und FORMAT-Routinen noch nicht eingegeben sind. Wenn Sie das getan haben, können Sie beliebig viele Daten eingeben. Erst einmal wird damit nichts weiter gemacht, aber Sie haben jetzt die Möglichkeit zu prüfen, ob die Bildschirmdarstellung übersichtlich ist, und ob Eingaben angenommen und überprüft werden.

Es ist nie möglich, die Reihenfolge für die Eingabe der Module optimal zu treffen. Es wird oft passieren, daß Sie den Wert einer oder zwei Variablen im Direktmodus zuweisen müssen (z. B. LET A$='STEUERABZUG' ohne Zeilennummer) und dann mit GOTO zum Anfang der Routine springen (RUN würde die gerade eingegebene Variable wieder löschen). Manchmal werden Sie zwei oder drei Module zusammen eingeben müssen, wie etwa PLATZSUCHE und DATENEINGABE, weil sie gewöhnlich zusammen arbeiten. Trotz dieser Ausnahmen von der Regel sollten Sie immer alles so früh wie möglich nach der Eingabe auszutesten versuchen, denn Sie können davon ausgehen, daß jeder Fehler, den sie entdecken, relativ einfach aufzuspüren und zu korrigieren ist, weil die Chancen zwanzig zu eins stehen, daß er sich in dem (den) zuletzt eingegebenen Modul(en) befindet.
All das wird Ihnen, wie gesagt, noch kein fehlerfreies Programm bescheren, aber das Programm wird viel weniger Fehler enthalten, als wenn Sie es von Anfang bis Ende einfach eingetippt hätten. Zumindest haben Sie am Schluß ein ganzes Programm ohne einen einzigen SYNTAX ERROR, da vor Beendigung der Programmeingabe jede einzelne Zeile schon einmal durchlaufen ist.

HINWEISE UND TIPS

Im folgenden finden Sie einige Punkte, auf die Sie achten müssen, wenn Sie ein Programm modular schreiben. Die Liste ist nicht vollständig, gibt jedoch einen Überblick über die Punkte, die allzuoft beim Programmieren vernachlässigt werden:

1. Programme sind übersichtlicher, wenn alle Module eine kommentierende Überschrift haben. Ich selbst benutze immer dieses Format:

```
1000 REM******************
1001 REM NAME DES MODULS
1002 REM******************
```

Damit ist das Modul im Programm deutlich markiert, wovon Sie erheblich profitieren, wenn Sie nach einiger Zeit auf das Listing zurückkommen.

2. Nachdem Sie dem Modul eine Überschrift gegeben haben, sollten Sie bei allen auf die Routine weisenden GOTOs und GOSUBs die Zeilennummer der Überschrift (1000) angeben, und zwar aus folgendem Grund: Falls Sie dem Arbeitsteil des Moduls eine neue erste Zeile geben oder die bestehende erste Zeile löschen wollen, werden bei jedem Aufrufen des Moduls UNDEF'STATEMENT-Fehler auftauchen. Da die Position der Überschrift unverändert bleibt, sind Veränderungen an der eigentlichen Routine problemlos.

3. Seien Sie bitte großzügig mit Kommentaren in REM-Zeilen. Wenn Sie drei Monate später das Programm ändern wollen, werden Sie heilfroh über das bißchen Extraarbeit sein, das Sie sich gemacht haben. Ohne Kommentare würde es später Stunden dauern, bis Sie auch nur herausgefunden hätten, worum es geht.

4. Jedes Programm, besonders ein umfangreiches, wird durch anschauliche Variablennamen leichter verständlich. Solche Namen können jedoch zu interessanten Fehlern führen, wenn sie Buchstabenkombinationen enthalten, die mit dem Anfang von BASIC-Schlüsselwörtern übereinstimmen. Wie Sie sehen, fehlt im oben gezeigten Listing das 'O' in FRMAT$, da das Wort mit 'FOR' am Anfang nicht angenommen wird. Es gibt noch ein weiteres Problem bei beschreibenden Namen, und zwar kann es beim relevanten Teil des Namens (die ersten zwei Buchstaben) leicht zu unbemerkten Doppelanwendungen kommen. PAYMENT und PARAMETER sind z. B. in Wirklichkeit ein und dieselbe Variable. Hier leistet Ihnen die während des Schreibens erstellte Variablenliste unschätzbare Dienste.

5. Die meisten Variablen in einem modularen Programm sind Hilfsvariablen (oder sollten es sein). Eine Eingabe, die Sie direkt in das Feld oder in die Variable mit den permanent gespeicherten Daten übernehmen, ist viel schwerer zu korrigieren, wenn sie sich als fehlerhaft erweist. Eingaben sollten einer Hilfsvariablen zugewiesen werden, wobei es überflüssig ist, ihnen in jedem Modul einen anderen Namen zu geben; normalerweise werden ein oder zwei wie T$, Q$, T und Q ausreichen, ohne Verwirrung zu stiften, weil sie am Ende des Moduls vergessen werden.

6. Einige Hilfsvariablen sind weniger temporär als andere. Im oben aufgelisteten Eingabemodul sind die Daten in zwei Variablen (TA und TD$) gespeichert, die danach oben noch an zwei andere Module übergeben werden sollen, bevor sie in die Hauptdatenfelder des Programms übernommen werden. In diesem Fall ist es sinnvoll, den Variablennamen mit T anzufangen, damit deutlich wird, daß es sich um eine Hilfsvariable handelt, aber fügen Sie noch einen oder mehrere Buchstaben als Hinweis auf die Funktion der Variablen an, und behalten Sie dabei die Gefahr einer Namensduplikation im Auge.

7. Bestimmen Sie sorgfältig die Reihenfolge, in der die Module im Programm angelegt werden sollen. Hierbei ist zweierlei zu beachten:

a) Oft gebrauchte Module werden etwas schneller ausgeführt, wenn sie am Programmanfang stehen.

b) Andererseits ist nichts schwerer zu durchschauen als ein Programm, in dem alle Module offenbar ohne logische Reihenfolge durcheinanderstehen.

Wenn Ihr Vorhaben nicht übermäßig kompliziert ist und keinen riesigen Aufwand an Zeit und Berechnungen erfordert, ist es sicher besser, das Programm so anzulegen, daß die Module logische Gruppen bilden; dabei sollten die kleineren Module, die in verschiedenen Bereichen des Programmes gebraucht werden (wie die FORMAT-Routine im o. g. Beispiel), am Ende stehen. Sie werden höchstwahrscheinlich keinen Unterschied in der Geschwindigkeit feststellen.

8. Die Programminitialisierung, d. h. die Definition der verschiedenen Felder und Variablen, ist eine eigene Funktion und sollte ein eigenes Modul haben, das bei Bedarf aufgerufen werden kann. Das gibt Ihnen die Möglichkeit, das Programm beim ersten Durchlauf einzurichten und, wenn es danach wieder aufgerufen wird, Daten beliebig zu löschen. Programme können so eingerichtet werden, daß sie sich selbst initialisieren. Sie definieren Felder dann, wenn sie gebraucht werden, vermeiden es aber, den Speicher zu löschen, falls er schon brauchbare Daten enthält. Dazu stellen Sie den Programmabschnitt, der den Speicher löscht, ganz an den Kopf des Programms. An den Anfang des Moduls setzen Sie eine Zeile, die eine wichtige Variable auf Null testet. Die gewählte Variable sollte immer einen anderen Wert als Null haben, wenn das Programm Daten im Speicher hat. Die Auto-Initialisierungszeile überspringt nun die Initialisierungsroutine, wenn die Variable *nicht* gleich Null ist. Nehmen Sie dazu folgendes Beispiel:

```
1000 REM*****************
1001 REM INITIALISIERUNG
1002 REM*****************
1010 IFIT<>0THEN1500
1020 CLR
1030 DIM A$(10),A%(500),B%(100),C%(100)
```

Hier dient IT dazu, die Anzahl der vom Programm gespeicherten Daten aufzuzeichnen. Wenn Sie Daten eingeben, das Programm anhalten und mit GOTO 1000 neu starten wollen, überspringt Zeile 1010 ohne Datenverlust die Zeilen, die den Speicher löschen und die Felder neu belegen. Wollen Sie dagegen die vorhandenen Daten löschen, müssen Sie das Programm nur mit RUN starten. Damit machen Sie den Speicher frei und setzen alle Variablen auf Null, IT eingeschlossen.

9. Ein Programm verdient eigentlich erst den Namen Programm, wenn es ein Menü besitzt, das zumindest einen Überblick darüber gibt, welche Funktionen das Programm hat und wie Sie zwischen den verschiedenen Funktionen wählen können. Abgesehen von diesem Hauptmenü könnten viele der einzelnen Module, wenn sie den Zugriff auf mehr als eine Funktion erlauben, eventuell durch ein eigenes kleines Menü verbessert werden.

10. Modulares Programmieren läßt sich durch den Einsatz von 'Flags' erleichtern. Das sind Variablen, mit denen man einem Modul anzeigt, daß in einem anderen etwas (nicht) passiert ist. Das klassische Beispiel dafür ist der Fehler-Flag. Modulares Programmieren bietet sich für den Einbau einer gesonderten Routine zur Meldung verschiedener Fehlerarten an, aber wie soll diese Routine heißen, und wie können Sie verhindern, daß ein Fehler das Programm abstürzen läßt? Die Lösung besteht im Gebrauch eines oder mehrerer Flags.

Nehmen Sie z. B. an, Sie sind entlang einer GOSUB-Kette vier Unterprogramme hinabgestiegen, d. h. das erste Programm hat das zweite aufgerufen, das zweite das dritte und das dritte das vierte. An diesem Punkt findet sich ein Fehler in den Daten, der vorher nicht aufgespürt werden konnte; sagen wir, Sie möchten eine Information einfügen, und es stellt sich heraus, daß im Feld kein Platz mehr verfügbar ist. Jetzt wollen Sie zweierlei: erstens den Benutzer von dem Problem in Kenntnis setzen, und zweitens sicherstellen, daß es keine Katastrophe gibt, wenn Sie das laufende Unterprogramm mit RETURN verlassen. Das wird gewöhnlich mit Hilfe eines Fehler-Flags erreicht, z. B. einer mit ERR (ERRor) bezeichneten Variablen. Diese würde im Normalfall auf Null gesetzt, erhält aber jetzt einen dem Fehlertyp entsprechenden Wert. Alle Module enthalten eine oder mehrere Zeilen, die herausfinden, ob ERR noch gleich Null ist. Wenn nicht, geben Sie lediglich mit RETURN die Ausführung an das in der Kette vorhergehende Modul weiter. Am Anfang der Kette steht dann ein Modul, das entdeckt, daß ERR gesetzt ist und das ERROR MESSAGE-Modul aufruft, um eine Fehlermeldung mit der Nummer ERR auszugeben. Dies ist nur ein Beispiel für den Gebrauch von Flags. Sie werden merken, wie unentbehrlich sie sind, wann immer eine Meldung zum Stand der Dinge von einem Modul an ein anderes übergeben werden muß (s. auch Erläuterungen zur Verwendung von Fehler-Flags in Kapitel 5).

11. Streuen Sie die Nummern der Anfangszeilen Ihrer Module großzügig. Sie werden das Programm sicherlich später weiter ausbauen wollen, und dann ist es sehr ärgerlich, wenn Sie die saubere Struktur verderben müssen, indem Sie ein neues Modul mit jeweils um 1 aufsteigenden Zeilennummern dazwischenklemmen oder weitere Module an den Schluß hängen, die eigentlich zu einer Gruppe in der Mitte des Programms gehören. Eine Streuung von 2000 pro Modul dürfte bei vielen Programmen für den Anfang nicht zuviel sein.

12. Unter Umständen kann die Ausführung eines Programms vereinfacht werden, wenn man RETURN-Anweisungen durch GOTOs ersetzt, z. B.: Modul 1 ruft Modul 2 auf. Ist eine gewisse Bedingung erfüllt, muß Modul 3 aufgerufen werden, statt mit RETURN zu Modul 1 zurückzuspringen. Es besteht die Möglichkeit, Modul 3 mit GOSUB aufzurufen, mit RETURN zu Modul 2 zurückzuspringen und von da aus mit

RETURN zu Modul 1. Man kann aber auch einfach mit GOTO Modul 3 aufrufen, an dessen Ende die RETURN-Anweisung die Ausführung an Modul 1 zurückgibt, ohne vorher erst mit RETURN Modul 2 aufzusuchen. Ein Beispiel für den Gebrauch dieser Technik wäre die oben beschriebene Fehlerkontrolle. Wenn wir jedes Modul, das direkt vom Hauptmenü des Programms aufgerufen wird, als Modul 'zweiter Ebene' betrachten, dann könnte jedes Modul zweiter Ebene eine Fehlersuchzeile haben, die mit GOTO das Fehlermeldungsmodul aufsuchen und von da mit RETURN zum Menü zurückspringen würde. Bei diesem Verfahren ist jedoch Sorgfalt geboten, denn wenn man sich bei der Anzahl der beteiligten GOSUBs und RETURNs verzählt, kann das entweder zu einem Programmstop durch OUT OF MEMORY ERROR führen, weil nicht durch RETURNs gelöschte GOSUBs die Speicherkapazität überlasten, oder zur Ausführung eines unerwarteten Unterprogramms.

13. Denken Sie daran, daß ein modular geschriebenes Programm leicht zu ändern ist. Halten Sie Ausschau nach besseren Techniken für die Bewältigung einer bestimmten Aufgabe. Wenn Sie in einem Buch oder einer Zeitschrift eine Verbesserung finden, werfen Sie das ursprüngliche Modul heraus, und ersetzen Sie es durch ein neues mit dem besseren Verfahren. Bringen Sie Ihre Programme auf den neuesten Stand, sonst verschenken Sie einen der größten Vorteile, die dieser Programmierstil bietet.

14. Module, die wichtige Techniken enthalten, speichert man am besten sowohl einzeln als auch innerhalb des Programms. Sie können sie dann leicht in spätere Programme übernehmen, indem Sie entweder ein 'Misch'-Programm benutzen, wie in *'Der Commodore 64 in der Praxis'*, Band 11 der Sachbuchreihe, beschrieben, oder einfach die Routine vom Band oder von der Diskette laden, und den Cursor mit der RETURN-Taste die Bildschirmzeilen hinunterfahren. Damit fügen Sie natürlich die Zeilen auf dem Bildschirm (aber nur diese!) in das neue Programm ein. Beachten Sie dabei, daß Sie eventuell die Zeilennummern im Modul ändern müssen, bevor Sie es einfügen.

15. Zum Schluß vergessen Sie nicht, daß fast alles modular geschrieben werden kann. Gute, professionelle Programme enthalten oft sehr viele einzelne Unterprogramme mit nur zwei oder drei Zeilen. Man kann dieses Verfahren zwar auch zu weit treiben, aber so leicht passiert das nicht.

SCHLUSS

Von allen Kapiteln kann man dieses zweifellos am ehesten überspringen. Es enthält sehr wenige praktische Beispiele und keine Arbeits-BASIC. Nur wenn Sie die hier dargestellten Prinzipien in die Praxis umsetzen, erkennen Sie, daß – auf den gesamten Inhalt des Buches bezogen – gerade dieses Kapitel entscheidenden Einfluß haben wird auf Ihre Versuche, erfolgreiche, brauchbare und verständliche Programme zu schreiben.

KAPITEL 2

FEHLERBESEITIGUNG

Eine englische Computerzeitschrift brachte kürzlich eine Artikelserie, in der die vielen verschiedenen Computersprachen erklärt wurden, die es heute gibt. Nach Abschluß der Serie wurde in einem Leserbrief darauf hingewiesen, daß man gerade die Sprache, die allen Computer-Besitzern – gleich welchen Typs – geläufig ist, vergessen hatte: Fluchen.

Auch wenn Sie Ihr Programm eingegeben, es dabei laufend getestet, es vielleicht sogar schon ein- oder zweimal ohne sichtliche Schwierigkeiten haben durchlaufen lassen, wird es trotzdem irgendwann dazu kommen, daß es im Chaos zusammenbricht. Schlimm genug, wenn das mit einem Ihrer eigenen Programme passiert, dessen Arbeitsweise Sie verstehen. Aber noch schlimmer wird es, wenn Sie ein fremdes Programm eingeben, das möglicherweise aus einem Buch oder einer Zeitschrift stammt. Sie können beim besten Willen kein instinktives Gefühl dafür haben, was alles wo vor sich geht. Meines Wissens haben auch manche meiner Leser schon Wochen oder Monate mit dem Versuch verbracht, ein Programm zu korrigieren, bei dessen Eingabe sie einen Fehler gemacht hatten. Wenn sie mich dann endlich ansprachen, waren sie oft schon der Verzweiflung nahe und davon überzeugt, daß ihr C 64 nicht richtig funktioniert. Tatsächlich hätten sie ihre Probleme innerhalb von Minuten selbst lösen können, wenn sie nur einige einfache Techniken gekannt und angewendet hätten.

INFORMATION – DER SCHLÜSSEL ZUR FEHLERBESEITIGUNG

Wenn Sie einen Fehler in Ihrem Programm finden, sollten Sie zuallererst daran denken, daß sich alle Informationen, die Sie zum Aufspüren des Fehlers brauchen, sicher in Ihrem C 64 befinden – und diese unschätzbaren Inhalte wollen Sie auf keinen Fall auslöschen. Ignorieren Sie deshalb einen entdeckten Fehler niemals, und lassen Sie das Programm nie nochmals laufen in der Hoffnung, daß der Fehler beim nächsten Mal nicht mehr auftritt. Sie könnten damit Erfolg haben – das Programm würde scheinbar reibungslos ablaufen –, aber Sie hätten die Chance vertan, einen Fehler zu entfernen, der später unweigerlich wieder auftreten würde und dann wahrscheinlich zu einem Zeitpunkt, zu dem sich schon wichtige Daten im Speicher befinden. Also ist das erste Gebot der Fehlerbeseitigung, jedem Fehler nachzugehen, sobald man ihn bemerkt.

Vorausgesetzt, Sie haben das einmal beschlossen: Auf welche Weise können Sie

die verfügbaren Informationen am besten nutzen? Um das zu entscheiden, müssen Sie die Information genau beurteilen:

1. Der Fehler, der das Programm gestoppt hat, ist möglicherweise von der Art, daß er das Problem auf eine bestimmte Zeile begrenzt. In dem Fall hat der C 64 die Art des gefundenen Fehlers und die Stelle im Programm, an der er vorkommt, identifiziert. Die Fehlermeldung wird so aussehen:

```
? [FEHLER-MELDUNG] IN ZEILE XX
```

2. Manche Fehler können das Programm stoppen, obwohl die Zeile, in der es 'hängenbleibt', völlig korrekt aussieht. Daneben gibt es die Fehler, die das Programm veranlassen, ein sinnloses Ergebnis zu liefern, ohne es zu stoppen. Die Ursache solcher Fehler ist äußerst schwer zurückzuverfolgen, zumal es in vielen Fällen keinen Hinweis auf den Ort der Panne gibt.

ZEILENSPEZIFISCHE FEHLERMELDUNGEN

In den meisten Fällen, in denen Sie eine Fehlermeldung bekommen, sind Ihre Probleme schon halb gelöst, weil der C 64 die zu überprüfende Programmzeile bereits für Sie lokalisiert hat. Es kann sein, daß die eigentliche Korrektur des Programms an einer oder mehreren anderen Zeilen vorgenommen werden muß, aber der Schlüssel für die Art des Fehlers findet sich immer in der Zeile, auf die die Fehlermeldung hinweist. In vielen Fällen müssen Sie nicht weiter suchen als bis zu der in der Meldung erwähnten Zeile, worüber sich viele Programmierer nicht im klaren sind.
Nehmen sie z. B. den geläufigsten aller Fehler, den SYNTAX ERROR. Wann immer Sie diese Meldung bekommen, wissen Sie, daß Sie beim Eingeben der genannten Zeile IN LINE XX einen Fehler gemacht haben *müssen.* Es hat keinen Zweck, die fragliche Zeile durchzulesen, zu dem Schluß zu kommen, sie sei in Ordnung, und dann das Programm versuchsweise noch einmal durchlaufen zu lassen oder es zum Funktionieren bringen zu wollen, indem man andere Zeilen ändert. Die angegebene Zeile enthält etwas, das der C 64 nicht als BASIC erkennt, und bevor diese Zeile nicht korrigiert worden ist, wird das Programm nicht laufen.
Fehlermeldungen, die normalerweise die genaue Stelle des Fehlers angeben, sind: SYNTAX ERROR, FILE NOT FOUND, FORMULA TOO COMPLEX, REDIMMED ARRAY, TYPE MISMATCH, UNDEFINED FUNCTION, UNDEFINED STATEMENT. Die Vorgehensweise bei diesen 'zeilenspezifischen' Fehlern ist wie folgt:

1. Listen Sie den Programmbereich auf, der sich vor der Zeile befindet, damit Sie die Zeile im Zusammenhang sehen.

2. Listen Sie die Zeile selbst nochmals separat auf, wobei Sie sie durch eine oder zwei Leerzeilen von den bereits aufgelisteten Zeilen trennen. Das sollen Sie deswegen tun, weil man auf dem C 64 sehr leicht zwei Zeilen so eingeben kann, daß sie als eine einzige Zeile angenommen werden. Es ist fast unmöglich, solche Fehler zu entdecken, wenn man eine Zeile nicht separat auflistet.
Untersuchen Sie diese Zeilen:

```
10 FOR I=1 TO 10 : LET X=I*100-50/(I*2)
20 NEXT I
```

Wie groß wird die Enttäuschung sein, wenn der Durchlauf zu nichts führt als zum Aufblinken von ?SYNTAX ERROR IN LINE 10 auf dem Bildschirm. Die Suche nach dem Fehler kann Stunden oder Tage dauern. In Wirklichkeit ist weiter nichts passiert, als daß am Ende von Zeile 10 statt RETURN die Leertaste gedrückt wurde, worauf der Cursor an den Anfang der nächsten Zeile sprang und Zeile 20 eingegeben wurde. Für den C 64 sieht die vollständige Version von Zeile 10 so aus:

```
10 FOR I=1 TO 10 : LET X=I*100-50/(I*2)20
NEXT I
```

Es ist nicht weiter verwunderlich, daß dies für den C 64 etwas schwer zu verstehen ist.

3. Wenn die Zeile aufgelistet ist und keine groben Fehler enthält, muß sie Befehl für Befehl und Zeichen für Zeichen untersucht werden. Um zu gewährleisten, daß Sie die Zeile nicht zu hastig überfliegen, können Sie den Cursor an den Zeilenanfang bringen und ihn mit der Steuertaste langsam nach rechts die Zeile entlangführen, während Sie jedes Zeichen und den Befehl, in dem es enthalten ist, überprüfen. Die meisten Syntax Errors (und die anderen Fehler, die sich ergeben, wenn der C 64 eine Zeile nicht wie von Ihnen beabsichtigt versteht) entstehen durch Weglassen von Zeichen (vor allem Klammern), Schreibfehler in Schlüsselwörtern oder versehentliche Umstellung von Zeichen. Auf fehlende Kommas zwischen Befehlen, auf Verwechslungen der Zahl 1 mit dem Buchstaben 'l' und der Null mit dem Buchstaben 'O' sollte man besonders achten. Zum Beispiel:

```
GOTO I000  anstatt  GOTO 1000 ergaebe
UNDEF' STATEMENT ERROR
```

4. Auch bei sorgfältigem Lesen der Zeile werden Sie den Fehler oft nicht ausfindig machen. Dann hängt der nächste Schritt davon ab, ob Sie den Variablenwert im Speicher erhalten müssen oder nicht. Im Fall eines Syntax Error ist der Variablen-

wert nicht relevant, so daß Sie in der Zeile ruhig der Fehlersuche dienende Änderungen vornehmen können, auch wenn damit der Variablenbereich gelöscht wird. Wenn das Problem den Wert einer oder mehrerer Variablen mitbetrifft, schlagen Sie den Abschnitt mit der Überschrift 'Nicht lokalisierte Fehlermeldungen' weiter hinten in diesem Kapitel auf.

5. Vorausgesetzt, die Variablen sind entbehrlich, setzen Sie eine STOP-Anweisung in eine neue Zeile direkt hinter derjenigen, die laut Hinweis den Fehler enthält. Nun fangen Sie mit der letzten Anweisung in der fehlerhaften Zeile an und fügen ein REM ganz am Anfang der Anweisung ein. Lassen Sie die Zeile nochmals durchlaufen (in Einzelfällen werden Sie das ganze Programm wieder bis zu dieser Stelle laufen lassen müssen). Wenn der Syntax Error jetzt verschwunden ist, befindet sich der Fehler in der letzten Anweisung, da diese durch REM aus der Zeile entfernt worden ist. Besteht der Syntax Error weiterhin, löschen Sie das REM und fügen es am Anfang der vorhergehenden Anweisung derselben Zeile ein. Wenn Sie bei der ersten Anweisung der Zeile angekommen sind, haben Sie festgestellt, welche Anweisung den Syntax Error enthält.

6. Sollten Sie den Fehler noch immer nicht gefunden haben, fangen Sie an, die Variablennamen zu ändern, und versuchen Sie es wieder. Bedenken Sie, daß der tatsächliche Variblenwert für einige Fehler nicht die geringste Rolle spielt. Die *Form* der Zeile ist falsch. Die Änderung von Variablennamen hat den Zweck herauszufinden, ob Sie ungültige Namen, vielleicht mit denselben Anfangsbuchstaben wie ein BASIC-Schlüsselwort, eingegeben haben.

NICHT LOKALISIERTE FEHLERMELDUNGEN

Wie schon erwähnt, bezeichnen manche Fehlermeldungen nicht die genaue Stelle des Fehlers im Programm; sie verraten nur, wo der Schlüssel zum Fehler zu finden ist. Diese Unterscheidung gilt nicht uneingeschränkt. Wenn Sie eine Zeile eingeben wie:

```
10 A=10/0
```

werden Sie eine Fehlermeldung DIVISION BY ZERO bekommen, und der Grund dafür ist leicht zu finden. Im ganzen werden jedoch BAD DATA, BAD SUBSCRIPT, DIVISION BY ZERO, FILE NOT OPEN, FILE OPEN, ILLEGAL QUANTITY, NEXT WITHOUT FOR, NOT INPUT FILE, NOT OUTPUT FILE, OUT OF DATA, OUT OF MEMORY, OVERFLOW, UNDEFINED FUNCTION und STRING TOO LONG meist

dann auftauchen, wenn der wirkliche Fehler sich nicht in der angegebenen Programmzeile befindet, sondern weiter vorn in der Ausführung des Programms.
Das Verfahren ist in diesen Fällen nicht so klar wie bei der Korrektur einer einzelnen Zeile. Meist besteht der erste Schritt darin, den Wert jeder einzelnen in der Zeile enthaltenen Variablen auszudrucken. Wenn die Zeile also hieße

```
100 A=X*Y/(T1*T2)
```

würden Sie eingeben

```
?X

?Y

?T1

?T2
```

und sich die Werte notieren, die dabei herauskämen. Jetzt gehen Sie die Zeile in Gedanken oder auf dem Papier durch, um festzustellen, warum gerade diese Werte zu der Fehlermeldung geführt haben, die das Programm gestoppt hat. Bevor Sie den Grund nicht gefunden haben, können Sie die Fehlersuche nicht fortsetzen. In der Regel stellt das kein Problem dar, aber falls die Zeile so kompliziert ist, daß sie nicht erkennen können, wie die Variablen zusammenarbeiten, müssen Sie die in der Zeile enthaltenen Befehle neu eingeben – diesmal auf zwei oder drei kürzere Zeilen verteilt –, bevor Sie das Programm wieder laufen lassen. Das sollten Sie jedoch nur im Notfall tun, denn wenn Sie nicht in der Lage sind, die genaue Reihenfolge der Ereignisse, die zu dem Fehler geführt haben, zu rekonstruieren, könnte der Fehler beim nächstenmal ausbleiben und irgendwann plötzlich wieder auftauchen.
Nachdem man die fehlerhafte Variable identifiziert hat, bleibt einem nichts übrig, als die Ausführung des Programms in Gedanken bis zu der Stelle zurückzuverfolgen, wo die Variable den unrichtigen Wert angenommen hat. Leider ist das in einem komplexen Programm oft unmöglich. In einer solchen Lage besteht die Lösung in einer Programmänderung, durch die während des Programmablaufs eine regelmäßige Überprüfung des Variablenwerts ermöglicht wird. Zu diesem Zweck schiebt man hinter den Programmabschnitten, wo die Variable geändert werden könnte, neue Zeilen ein, die jeweils nur aus STOP bestehen; der Nachteil dabei ist, daß durch Einfügung neuer Zeilen der Speicher gelöscht wird, so daß Sie das Programm ganz neu bestimmen müssen.
Nach der Eingabe der vorläufigen STOP-Zeilen läßt man das Programm wieder

laufen, um eine Wiederholung des Fehlers zu veranlassen. Bei jedem Programm-STOP können Sie den Wert der fehlerhaften Variablen ausdrucken und dann mit CONT das Programm fortsetzen, solange Sie den Fehler noch nicht gefunden haben. Falls Sie auf den Bereich stoßen, in dem das Problem anscheinend entstanden ist, die Programmzeilen jedoch korrekt aussehen, sollten Sie u. a. nach möglichen Doppelanwendungen von Variablennamen Ausschau halten. Die Zeilen, die eine wichtige Variable enthalten, können tatsächlich fehlerlos sein, und doch liefert das Programm Unsinn, weil es demselben Variablennamen irgendwann eine andere Funktion zuweist; denken Sie daran, daß bei allen Variablen nur die ersten beiden Buchstaben relevant sind.
Sie können sich die Arbeit grundsätzlich erleichtern, wenn Sie diese beiden Regeln beachten:

1. Bei der Verarbeitung komplexer Datensätze sollten Sie als eines der ersten Programm-Module die Dateiroutine zum Speichern der Informationen auf Band oder Diskette eingeben. Speichern Sie die zum Austesten eingegebenen Daten regelmäßig ab; falls das Programm abstürzt, können Sie so den letzten Datensatz vom Band neu laden und brauchen ihn nicht von Anfang an neu einzutippen.

2. Die meisten Programmfehler zeigen sich bei einer geringen Anzahl von Daten genauso wie bei einer größeren Datenmenge. Anstatt riesige Mengen direkt einzugeben, geben Sie zunächst nur drei oder vier Daten ein und gehen Sie damit alle Programmfunktionen durch. Sollte ein Fehler auftreten, ist es bei nur vier eingegebenen Daten leicht, die Sequenz erneut zu simulieren, vor allem bei stilisierten Elementen oder Werten wie AAAA, BBBB, CCCC, 1111, 2222 und 3333.

Die Suche nach einem Fehler, der an einer unbestimmten Stelle des Programms steckt, kann als Testaufgabe angesehen werden. Sie kann nur dann erfolgreich bewältigt werden, wenn man gründlich vorgeht und der Ausführung des Programms im einzelnen folgt, und wenn Ihnen die Aufgabe jedes Programmabschnitts völlig klar ist. Sobald solche Fehler auftauchen, werden Sie es besonders zu schätzen wissen, daß Sie den in Kapitel 1 gegebenen Rat befolgt und Ihr Programm in streng funktionale Module eingeteilt haben, denn diese Programmstruktur macht das Aufspüren von Fehlern erheblich leichter.

FEHLERMELDUNGEN UND IHRE BEDEUTUNG

Es gibt so viele verschiedene Programmfehler wie Programmierer, weil sie ganz einfach von Programmierern begangen werden. Deshalb ist es ganz unmöglich, eine vollständige Aufstellung der Bedeutungen aller möglichen Fehlermeldungen

zu machen. Im folgenden finden Sie eine Liste der geläufigen Meldungen und ihrer wahrscheinlichen Ursachen:

BAD DATA: Der übliche Grund: Die Struktur der Sicherheitsroutine wurde in der entsprechenden Laderoutine nicht genau nachgebildet. Als weitere Ursache kommt in Frage, daß die gespeicherten Daten nicht korrekt mit CHR$(13) nach jedem Element getrennt wurden. Beim Gebrauch einer Variablen als Trennzeichen zwischen den Elementen (z. B. PRINT#1,A$,R$,B$,R$,A,R$,B) vergewissern Sie sich, ob Sie den Separator tatsächlich definiert haben. Das Modul kann zwar äußerlich in Ordnung sein, aber wenn R$ nicht als CHR$(13) definiert wurde, werden die Daten auf dem Band oder der Diskette verschmolzen, und im Lademodul wird dann die Reihenfolge gegenüber der gespeicherten verschoben sein.

BAD SUBSCRIPT: Sehen Sie sich die Werte in Klammern hinter dem Feldnamen an, weil einer davon größer ist als der entsprechende Bereich, den Sie dem Feld bei der DIMensionierung zugewiesen haben. Falls der Fehler nicht sofort klar ist, prüfen Sie nach, ob Sie das Feld auch wirklich dimensioniert haben, denn das Programm nimmt Verweise auf die Elemente 0 bis 9 eines eindimensionalen Feldes an, auch wenn es nicht DIMensioniert wurde, aber es bricht ab, wenn Sie Element 10 anzusprechen versuchen.

DEVICE NOT PRESENT: Entweder haben Sie bei einer Anweisung, die sich auf eine Datei bezieht (z. B. OPEN 1,7,1), die falsche Nummer angegeben, oder Sie haben vergessen, die Floppy oder den Kassettenrecorder anzuschließen, auf die das Programm zuzugreifen versucht. Leider kann dieser Fehler auch auftauchen, wenn andere Fehler bei der Dateiverwaltung vorkommen und das INPUT/OUTPUT-System durcheinandergerät. In manchen Fällen kann man das beheben, indem man das betreffende Gerät ausschaltet, in anderen muß der C 64 aus- und wieder eingeschaltet werden, was den Verlust von Programm und Daten zur Folge hat.

DIVISION BY ZERO: Da Sie wahrscheinlich nicht /0 in eine Zeile geschrieben haben, liegt die Fehlerursache vermutlich darin, daß eine Variable vom Programm nicht richtig verarbeitet wurde oder ein falscher Variablenname benutzt wurde.

EXTRA IGNORED: Eine Input-Abfrage erwartet eine bestimmte Anzahl von Daten, z. B. ist nach INPUT A,B die Eingabe von zwei Zahlen erforderlich. Wenn Sie mit einer größeren Anzahl von Daten antworten, wie z. B. 10,12,14, dann teilt Ihnen das Programm mit, daß ein Element eingegeben wurde, das nicht berücksichtigt wird. Diese Meldung erfolgt auch dann, wenn String-Eingaben selbst Kommas enthalten; denn sie werden als Separatoren zwischen Daten gedeutet. In solchen Fällen kann man sich nur mit einer GET-Operation behelfen.

FILE NOT FOUND: Entweder haben Sie den falschen Dateinamen angegeben, oder Sie benutzen die falsche Kassette/Diskette.

FILE NOT OPEN: Sie haben eine Operation mit einer Dateinummer auszuführen versucht, für die der C 64 entweder keine OPEN-Anweisung findet, oder die bereits mit CLOSE abgeschlossen wurde.

FILE OPEN: Die Umkehrung des vorigen Fehlers. Der häufigste Grund ist Vergessen der CLOSE-Anweisung in einer früheren Routine mit derselben Dateinummer. Wenn Sie also Daten in ein Programm laden und dabei ein File mit der Nummer '1' benutzen, nach dem Laden aber vergessen, CLOSE 1 einzugeben, werden Sie in Zukunft keine Daten mehr unter der Dateinummer 1 speichern können.

FORMULA TOO COMPLEX: Die einfache Lösung könnte darin bestehen, den Ausdruck der betreffenden Zeile in zwei Ausdrücke in verschiedenen Zeilen aufzuteilen. Leider kommt der Fehler auch in einer Reihe von Situationen vor, in denen das Betriebssystem durcheinandergerät; für diese Fälle gibt es keinen verläßlichen Hinweis auf die mögliche Bedeutung der Meldung.

ILLEGAL QUANTITY: Eine der Variablen, die auf ein Feld zugreifen, könnte negativ sein; oder Sie haben vielleicht versucht, eine außerhalb von −32768 bis +32767 liegende Zahl in ein Integerfeld einzusetzen. Es kann auch sein, daß Sie eine Ein-Byte-Funktion auf eine Zahl außerhalb von 0−225 anzuwenden versuchten. Im Zweifel geben Sie die einzelnen Anweisungen der Zeile im Direktmodus (ohne Zeilennummern) ein und lassen sich vom C 64 zeigen, welche er nicht akzeptiert.

NEXT WITHOUT FOR: Das Programm erkennt den Anfang der Schleife nicht, auf die sich die Anweisung bezieht. Sie haben entweder die FOR-Anweisung ausgelassen oder sind in die Schleife gesprungen, weshalb das FOR nicht ausgeführt wurde.

NOT INPUT FILE: Sie haben fälschlich eine Ausgabedatei angesprochen, anstatt eine Datei für die Eingabe von Daten zu öffnen.

NOT OUTPUT FILE: Umkehrung des o. g. Fehlers.

OUT OF DATA: Sie wollten mehr Daten mit READ einlesen als in den DATA-Anweisungen des Programms enthalten sind. Wie man dieses Problem löst, lesen Sie bitte im Abschnitt über DATA-Anweisungen in Kapitel 12 nach.

OUT OF MEMORY: Dafür gibt es vier mögliche Ursachen:

a) Sie haben Felder dimensioniert, die für die verfügbare Speicherkapazität zu groß sind. Bei Programmen, die mit großen Datenmengen arbeiten, ist es ratsam, den verfügbaren Speicher mit FRE zu prüfen, bevor Sie die Größe der Felder endgültig bestimmen.

b) Sie haben zu viele GOSUBs, die gleichzeitig laufen und dabei den Stack, d. h. den Speicherbereich, der sich ihre Rückkehradressen merken muß, überfüllen.

c) Sie haben zu viele FOR-Schleifen, die gleichzeitig laufen und ebenfalls den Stack in Unordnung bringen, der sich den aktuellen Stand jeder Schleife merken muß. Dies kann in Verbindung mit b) vorkommen, wenn Sie eine lange Kette von GOSUBs *und* eine große Zahl von Schleifen ineinander verschachtelt haben.

d) Unbestimmte Ursache. Leider ist auch dies ein Fehler, der auftreten kann, wenn der C 64 durcheinandergerät.

OVERFLOW: Das Problem entsteht meist durch eine falsch definierte Variable, so daß z. B. eine große Zahl durch einen winzigen Bruch dividiert wird.

REDIM'D ARRAY: Sie haben versucht, in einer DIM-Anweisung denselben Feldnamen zu benutzen wie in einem bereits DIMensionierten anderen Feld. Wenn Sie ein Feld neu dimensionieren wollen, müssen Sie zuallererst den Speicher löschen.

REDO FROM START: Eine Input-Anweisung, die eine Zahl erwartete, ist mit einem String beantwortet worden. Beachten Sie, daß ein INPUT, der einen String verlangt, ohne Anzeichen für ein Versehen bereitwillig auch eine Zahl annimmt.

RETURN WITHOUT GOSUB: Sie haben ohne Verwendung von GOSUB ein Unterprogramm eingegeben. Die häufigste Ursache sind Unterprogramme, die an das Programmende angehängt werden, ohne daß eine STOP-Anweisung an das Ende des Programmhauptteils gestellt worden ist.

STRING TOO LONG: Bei der Addition oder der Eingabe von Strings oder beim Einlesen von Strings aus einer Datei haben Sie versucht, einen String zu bilden, dessen Länge die allgemeine Höchstmenge von 255 Zeichen oder das Maximum von 80 Zeichen für String-Eingaben überschreitet. Der Fehler tritt auch beim Löschen von Daten auf, die beim Speichern nicht ordnungsgemäß durch CHR$(13) getrennt wurden.

?SYNTAX ERROR: Der C 64 versteht die Zeile einfach nicht, die er auszuführen versucht. Lesen Sie im ersten Teil des Kapitels nach, wie man unerkannte Fehler aufspürt.

TYPE MISMATCH: Manchmal schreibt der Zusammenhang des Programms die Bestimmung einer Zahl vor, während es tatsächlich einen String vorfindet (oder umgekehrt). Somit würde A=A$+B$ zu diesem Fehler führen. Häufigste Ursache ist versehentliches Auslassen von $-Symbolen in Zeilen, die Strings verarbeiten.

UNDEF'D FUNCTION: Sie wollen eine vom Benutzer definierte Funktion verwenden, aber das Programm erinnert sich nicht daran, daß diese Funktion definiert worden ist.

UNDEF'D STATEMENT: Die Zeile, die Sie mit GOTO oder GOSUB erreichen wollen, existiert nicht. Wenn die Meldung auf den ersten Blick keinen Sinn zu ergeben scheint, untersuchen Sie die auf GOTO oder GOSUB folgende Zahl, um sicherzugehen, daß Sie nicht aus Versehen 'I' für '1' oder 'O' für '0' geschrieben haben. Ist das nicht der Fall, dann haben Sie vielleicht, wie im ersten Teil dieses Kapitels beschrieben, eine Zeile an das Ende der vorhergehenden angehängt. Der zweite Teil der Zeile ist scheinbar vorhanden, aber er ist keine eigenständige Zeile.

SCHLUSS

Zur erfolgreichen Fehlerbeseitigung gelangt man durch Erfahrung, viel Nachdenken und harte Arbeit. Bei aller Mühe wird es doch immer ein paar Fehler geben, die Sie aus dem Konzept bringen, und die Sie erst Tage oder Wochen später finden. Wenn es soweit ist, ist die beste Unterstützung bei der Fehlerbeseitigung eine andere Person, die sich mit dem Problem befaßt; es kommt nämlich oft vor, daß ein anderer Programmierer etwas sofort entdeckt, das Sie wegen Ihrer allzu großen Vertrautheit mit dem Programm nicht sehen konnten. Dennoch sollten Sie erst dann jemand anderen rufen, wenn Sie das Problem genauer beschreiben können: die betreffenden Variablenwerte und den vermutlichen Bereich, in dem das Problem entsteht. Nur zu wissen, daß eine bestimmte Zahl einen bestimmten Fehler erzeugt, ist nicht genug; das ist erst der Anfang.

KAPITEL 3

STRINGS

Strings sind eine leichte und flexible Methode zum Speichern von Daten in den C 64. Wo man sie verwendet, können Informationen auch bei komplexen Operationen fast sofort gelöscht, eingefügt oder geändert werden. Das Herumexperimentieren mit Strings kann eine Menge zum Erfolg eines Programmierers beitragen. Dieser Umstand scheint in weiten Kreisen unbekannt zu sein – nicht etwa, weil der Umgang mit Stringfunktionen an sich schwierig ist, sondern weil er oft kniffelig ist und Zeilen mit Stringfunktionen auf den ersten Blick kompliziert aussehen.
Der C 64 bietet dem Anwender drei Stringfunktionen: LEFT$, RIGHT$ und MID$. Ihre Funktion ist den meisten völlig klar; ich gebe hier eine kurze Zusammenfassung für diejenigen, die sich nicht genau erinnern:

1) LEFT$(A$,10) bedeutet die *ersten* 10 Zeichen von A$.
2) RIGHT$(A$,10) bedeutet die *letzten* 10 Zeichen von A$.
3) MID$(A$,10) benennt den Teil von A$, der mit Zeichen Nr. 10 *beginnt,* bis zum Stringende.
4) MID$(A$,10,5) benennt den Teil von A$, der vom 10. Zeichen an 5 Zeichen umfaßt.

Für einen konkreten String A$, in diesem Fall das komplette Alphabet, ergeben sich:

1) ABCDEFGHIJ
2) QRSTUVWXYZ
3) JKLMNOPQRSTUVWXYZ
4) JKLMN

Wie die Beispiele zeigen, sind die Befehle eigentlich ganz unkompliziert. Für viele Leute fängt es offenbar dann an schwierig zu werden, wenn die Befehle kombiniert werden sollen. Dann werden manche Zeilen mit Klammern derart überladen, daß ihre Funktion sehr komplex und kaum mehr durchschaubar erscheint. Mit dieser Schwierigkeit werden Sie am besten fertig, indem Sie bei dem Ausdruck mit den meisten Klammern anfangen, die Zeile zu übersetzen, um sie Schritt für Schritt zu vereinfachen. Nehmen Sie z. B. den Ausdruck

```
MID$(LEFT$(RIGHT$(A$,10),5),3)
```

Was läßt sich damit machen? Wir setzen voraus, daß A$ hier wiederum das Alphabet ist. Wir fangen mit dem Stringausdruck in der innersten Klammer an, d. h. RIGHT$(A$,10), weil wir nichts sonst übersetzen müssen, um an das Ergebnis zu kommen. RIGHT$(A$,10) bezeichnet 'QRSTUVWXYZ'. Wir erhalten also:

```
MID$(LEFT$("QRSTUVWXYZ",5),3)
```

Nach derselben Methode ergibt sich für den Teil LEFT$ die Bedeutung QRSTU, und wir erhalten:

```
MID$("QRSTU",3)
```

oder STU. Arbeiten Sie sich wie bei jeder anderen Art von Ausdrücken auch bei Stringausdrücken von innen nach außen, dann wird sich das Problem von selbst lösen.
Im nächsten Kapitel werden wir Methoden untersuchen, wie Stringfunktionen sowohl miteinander als auch mit anderen BASIC-Befehlen kombiniert werden können, um eine Vielfalt interessanter und nützlicher Programmiermöglichkeiten zu schaffen. Einige der nachfolgenden Kapitel werden die hier beschriebenen Techniken auf ganz unterschiedliche Arten anwenden, deshalb sollten Sie erst dann weiterblättern, wenn Sie sicher sind, daß Sie die angegebenen Beispiele verstanden haben.

VERKETTUNG ODER STRING-ADDITION

Eine der einfachsten Operationen für Strings ist die Addition:

```
100 A$=B$+C$+D$
```

würde einen neuen String ergeben, der aus den aneinandergereihten drei Strings auf der rechten Seite der Gleichung besteht. Dieses einfache Verfahren dient oft zur Bildung von sinnvollen Strings, die aus kleineren Teilinformationen aufgebaut sind. Ein einfaches Beispiel dafür wäre:

```
1000 REM*****************
1001 REM STRING-ADDITION
1002 REM*****************
1010 INPUT"NACHNAME:          ";NN$
1020 INPUT"VORNAME:           ";VN$
1030 INPUT"GESCHLECHT (M/F): ";G$
```

```
1040 N$="HERR":IF G$="F"THEN N$="FRAU"
1050 NAME$=N$+" "+VN$+" "+NN$
1060 PRINT NAME$
```

Nicht immer muß diese Technik auf so triviale Art angewendet werden. Einzelne Ausdrücke können, jeweils durch eine Markierung wie '*' voneinander getrennt, zu einem einzigen String zusammengefaßt werden, um Speicherplatz zu sparen; denn jeder einzeln gespeicherte String hat einen zusätzlichen Verwaltungsbedarf von drei Bytes im Speicher. Solche 'komprimierten' Eingaben können mit Hilfe einer weiter unten beschriebenen String-Suchroutine zerlegt werden; in den Kapiteln über Datenstrukturen werden kompliziertere und flexiblere Methoden erklärt.

STRING-SUBTRAKTION

Genauso wie Strings durch Addition zusammengesetzt werden können, kann man auch Teile von einem einzelnen String subtrahieren. Das ist nicht so einfach wie die Addition, denn für den C 64 ergäbe A$=B$−C$ keinen Sinn. Einen String von einem anderen zu subtrahieren bzw. zu entfernen, bedeutet nichts anderes als den ursprünglichen String neu zu definieren, so daß die zu subtrahierenden Zeichen ausgeschlossen werden. Das genaue Verfahren hängt von der genauen Stellung der zu entfernenden Zeichen im Hauptstring ab:

1. Um LL Zeichen vom linken Ende von A$ zu entfernen, muß A$ neu definiert werden als der Abschnitt von A$, der auf die ersten LL Zeichen folgt:

```
100 A$=MID$(A$,LL+1)
```

2. Um LL Zeichen vom Ende von A$ zu entfernen, muß A$ neu definiert werden als die Gesamtheit der Zeichen bis einschließlich demjenigen, das vor dem ersten zu entfernenden Zeichen steht. Zu diesem Zweck stellt man mit LEN die aktuelle Länge des Strings fest und gibt dann an, daß er jetzt diese Länge abzüglich der Anzahl der zu löschenden Zeichen haben soll:

```
100 A$=LEFT$(A$,LEN(A$)-LL)
```

3. Um LL Buchstaben aus der Mitte eines Strings zu entfernen, muß man außer der Länge des zu löschenden Abschnitts nur die Startposition (SP) kennen. Wenn das bekannt ist, muß der String neu definiert werden als die Verknüpfung der Abschnitte *vor* und *hinter* den zu löschenden Zeichen:

```
100 A$=LEFT$(A$,SP-1)+MID$(A$,SP+LL)
```

Diese Zeile hat den Sinn: Wenn die zu löschende Zeichengruppe bei SP beginnt, sollen alle Zeichen bis einschließlich SP−1 erhalten bleiben. Die eigentliche Buchstabengruppe umfaßt LL Zeichen und endet also in Position SP (erstes Zeichen) +LL (Länge) minus Eins. Demzufolge beginnt die zweite zu erhaltende Zeichengruppe im String bei LL+SP und erstreckt sich bis zum Stringende. Das läßt sich illustrieren am Beispiel des Strings ABCDEFG, vom dem CDE subtrahiert wird. Die Anfangsspalte von CDE ist das dritte Zeichen, die Länge beträgt drei Zeichen. Es blieben also die Stringabschnitte bis SP−1 (d. h. AB) und ab SP+LL nach der zu löschenden Zeichengruppe (d. h. FG) erhalten, woraus sich der Ausdruck ABFG ergäbe.

EINFÜGEN VON ZEICHEN IN STRINGS

Nach dem, was wir über String-Addition und -Subtraktion bisher gesagt haben, fällt Ihnen vielleicht auf, daß wir zwar eine Zeichengruppe aus einem vorhandenen String entfernen können, aber noch nicht untersucht haben, wie man Zeichen in einen String einschiebt. Wir haben beim Entfernen von Elementen aus einem String eine Methode untersucht, mit der die beiden für das Ergebnis zu speichernden Strings identifiziert werden können. Beim Einschieben einer neuen Zeichengruppe wird in etwa dieselbe Methode angewendet. Im folgenden Beispiel soll ein neuer String – B$ – in A$ eingefügt werden, wobei das erste Zeichen von B$ zum Zeichen PP des erweiterten A$ werden soll:

```
100 A$=LEFT$(A$,SP-1)+B$+MID$(A$,PP)
```

UMSTELLUNG VON ZEICHEN IN STRINGS

Nachdem wir untersucht haben, wie man einem String Zeichen anfügt oder sie daraus entfernt, sind wir nun in der Lage, durch Kombinieren beider Methoden Zeichen innerhalb eines Strings umzustellen. Im wesentlichen erfordert die Umstellung von Zeichen in einem String zwei Operationen: Die zu verschiebende Zeichengruppe muß vom String subtrahiert werden und dann an anderer Stelle addiert werden. Zur Illustration des Verfahrens gehen wir von einem String A$ aus, der eine Gruppe von LL Zeichen enthält, die in der Zeichenposition SP des Strings beginnt. Die Aufgabe besteht darin, die Gruppe an eine neue Stelle zu verschieben, die in der Position FP beginnt. Als Beispiel könnte man etwa den String ABGHICDEFJKL

neu ordnen, so daß man durch Umstellung von GHI (in Position drei beginnend) an einen anderen Ort zu dem Ergebnis ABCDEFGHIJKL gelangt. Im Ergebnisstring ist die neue Anfangsposition der Gruppe, also die Zeichenspalte, sieben.
Es ist nicht ganz unkompliziert, die richtige Position zu bestimmen, an die der String wieder eingeschoben werden soll. Als erstes muß man von der Tatsache absehen, daß die Gruppe in ihrer ursprünglichen Form gelöscht wird, und einfach die neue Anfangsposition im bestehenden String festliegt. Im Fall des o. g. Musterstrings soll die verschobene Gruppe dort beginnen, wo sich zur Zeit das 'J' befindet, d. h. in Zeichenposition 10. Nun gibt es zwei Alternativen:

a) Befindet sich die neue Anfangsposition der Zeichengruppe *vor* ihrem gegenwärtigen Anfang, so muß die Nummer der Zielposition nicht korrigiert werden.

b) Befindet sich die neue Anfangsposition *hinter* dem derzeitigen Ende der Zeichengruppe, so muß die Länge der Gruppe von der Nummer der Zielposition subtrahiert werden.

Im obigen Beispiel endet die Gruppe zunächst in Position fünf, und sie soll in Position zehn wieder eingefügt werden. Daher müssen wir die Länge der Gruppe (3) subtrahieren, um die Zielposition zu erreichen, die nach den vorhergehenden Überlegungen 7 sein muß.
Auf das o. g. Beispiel angewendet, würde alles zusammen etwa zu folgendem Ergebnis führen:

```
50 A$="ABGHICDEFJKL"
60 FP=10
70 SP=3
80 LL=3
2000 REM**********************
2001 REM BEWEGEN VON ZEICHEN
2002 REM**********************
2010 IFFP>(SP+LL-1)THENFP=FP-LL
2020 TT$=MID$(A$,SP,LL)
2030 A$=LEFT$(A$,SP-1)+MID$(A$,SP+LL)
2040 A$=LEFT$(A$,FP-1)+TT$+MID$(A$,FP)
2050 PRINTA$
```

VARIABLEN:
A$ Ausgangsstring
FP Anfangsposition der neuen Zeichengruppe, wenn sie ohne vorherige

Löschung wieder eingefügt würde
LL Länge der zu verschiebenden Zeichengruppe
SP Derzeitige Anfangsposition der zu verschiebenden Zeichengruppe
TT$ Zwischenspeicher der zu verschiebenden Zeichengruppe

Mit den Zeilen ab 2000 kann man beliebige Gruppen innerhalb des Ausgangsstrings umstellen, vorausgesetzt, die Anfangsposition (SP), Zielposition (FP) und Länge (LL) sind bekannt.

SUCHEN IN STRINGS

Bei allem, was bisher über die Handhabung von Strings gesagt wurde, sind wir davon ausgegangen, daß der Programmierer alle nötigen Informationen für die Bestimmung der Anfangs- und Endstellen aller Stringabschnitte hat, auf die er zugreifen will. Das ist in der Regel insofern ganz anders, als nicht der Programmierer, sondern das Programm selbst entscheidet, wo entsprechend seiner vorgegebenen Angaben Änderungen vorgenommen werden. Sehr oft ist schon durch den Inhalt des Strings selbst festgelegt, wie das Programm ihn bearbeitet.
Weiter oben in diesem Kapitel haben wir das Beispiel einer String-Addition mit Teilen vollständiger Namen untersucht. Zu Anfang wurde jeder Teil als ein eigener String gespeichert und dann zu einem längeren String der Form 'FRAU EVA SCHMIDT' zusammengefaßt. So herum ist es weiter nicht schwierig, aber wie macht man es umgekehrt, d. h. wie entfernt man Teile aus dem Ganzen? Bei FRAU oder HERR wäre es noch einfach, da sie gleich lang sind, aber es könnten auch Titel wie DR. vorkommen, deren Länge von der Norm abweichen. Selbst wenn wir den Titel leicht aus dem Namen herausholen könnten, wäre die Länge des Vornamens noch nicht vorhersehbar. Wie kann man also den Namen zergliedern?
Die Antwort lautet: Alle zur Zerlegung des Namens nötigen Informationen sind im Namen enthalten, und zwar in Form der Leerstellen zwischen den drei Datenelementen. Dieselben Zwischenräume, die wir beim Lesen des Namens in Gedanken machen, können vom Programm genutzt werden, sofern es über eine Methode verfügt, den ganzen String nach der Zeichengruppe abzusuchen, die bearbeitet werden soll. Hier ist eine einfache Methode, einen String A$ nach einer einzelnen Zeichenkombination, TARGET$, zu durchsuchen:

```
100 FOR I=1 TO LEN(A$)-LEN(TRGT$)+1
110 IF MID$(A$,I,LEN(TRGT$))=TRGT$ THEN
SP=I:GOTO 150
120 NEXT I
```

Damit wird die Routine die Anfangsposition des ersten TRGT$ (gesuchte Zeichengruppe) innerhalb des Hauptstrings identifizieren und sie in die Variable SP ablegen. Mit dieser Technik können wir schnell eine Routine zur Zerlegung eines weiteren Strings mit mehreren Informationseinheiten entwerfen – im folgenden Beispiel die drei Teile eines Namens im Format 'HERR MICHAEL MAY':

```
50 NAME$="HERR MICHAEL MAY"
60 TRGT$=" "
70 DIM N2$(50)
3000 REM**************************
3001 REM STRING SUCHE UND AUSWAHL
3002 REM**************************
3010 TT=LEN(NAME$)-LEN(TRGT$)+1
3020 S1=1
3030 IT=0
3040 FORJ=1TOTT
3050 IF MID$(NAME$,J,LEN(TRGT$))<>TRGT$T
HEN3070
3060 N2$(IT)=MID$(NAME$,S1,J-S1):S1=J+LE
N(TRGT$):IT=IT+1
3070 NEXTJ
3080 N2$(IT)=MID$(NAME$,S1):IT=IT+1
3090 FORI=0TOIT-1
3100 PRINTN2$(I)
3110 NEXTI
```

VARIABLEN:
IT Anzahl der in NA$ gefundenen Daten
N2$ Für die in NA$ gefundenen Daten bestimmtes Feld
NA(ME)$ Zu durchsuchender Hauptstring (NAME$ wird nur als NA$ registriert)
S1 Beginn des zu durchsuchenden Abschnitts von NA$
TR(GT)$ Trennzeichen zwischen den Datenelementen
TT Letztes Zeichen in NA$, das mit TR$ verglichen zu werden lohnt, ohne NA$ ganz zu durchlaufen

Bei diesem Verfahren beginnt die Suche nach dem Zielstring beim ersten Zeichen des Hauptstrings; jedesmal, wenn der Zielstring gefunden wird, wird der Teil des Hauptstrings ab S1 bis zum Zielstring in das Feld N2$ übergeben. Die Suche wird dann von dem Zeichen hinter dem Zielstring an wieder aufgenommen. Die Routine geht davon aus, daß der Hauptstring nicht mit dem Zielstring endet, und geht daher

nach Ausführung der Schleife unmittelbar an den Teil des Hauptstrings, der hinter dem zuletzt gefundenen Teilstring steht.
Für solche Suchtechniken sind eine ganze Reihe von Anwendungen möglich. Wie im Kapitel über String-Addition beschrieben, können Informationen in Strings zusammengefaßt werden, und mit Hilfe einer Suchroutine die verschiedenen Teile wieder herausgeholt werden können. In 'intelligenten' Programmen, die den Wortlaut eingegebener Befehle analysieren, kann die Routine zur Abfrage dienen, ob bestimmte Verben oder Substantive vorkommen – diese Technik wird z. B. häufig bei Abenteuerspielen eingesetzt. Daneben kann sie in Dateiprogrammen zum Auffinden von Eintragungen in der Datei dienen, die ein bestimmtes Wort enthalten.

REGELMÄSSIGE STRINGSTRUKTUREN

Da Strings mit einfachen Anweisungen gehandhabt werden können, die bestimmte Abschnitte einfügen oder entfernen, ohne daß man sich um die Umstellung des vorhandenen Inhalts kümmern muß, sind sie der ideale Ort zum Speichern von Daten regelmäßiger Länge, in denen regelmäßig etwas eingefügt oder gelöscht werden muß. In einem einzelnen String mit einer maximalen Länge von 255 Zeichen können 63 Datenelemente mit je vier Zeichen, 50 mit je fünf Zeichen etc. untergebracht werden.
Wenn alle Daten einen identifizierbaren Platz bekommen sollen, muß der String zuerst in der für die Aufnahme aller Elemente nötigen Länge eingerichtet werden. Dazu benutzt man am einfachsten eine Schleife, um den String Zeichen für Zeichen aufzubauen. Die nächste Routine richtet einen String für die Speicherung von Elementen mit je vier Zeichen ein, deren Adresse der Anwender bestimmen kann:

```
4000 REM******************************
4001 REM SETZEN VON GLEICHL. STRINGS
4002 REM******************************
4010 A$="":FORI=1TO252:A$=A$+" ":NEXT
4020 INPUT"VIERSTELLIGES ZEICHEN:    ";T$

4030 INPUT"ZEICHEN POSITION (1-63): ";T
4040 A$=LEFT$(A$,4*(T-1))+T$+MID$(A$,T*4
+1)
4050 PRINT"␣";A$
4060 GOTO4020
```

Ebenso leicht können Daten aus einem numerierten Platz entfernt werden. Ergänzen Sie die Routine um die folgenden Zeilen, geben Sie einige Daten ein und

probieren Sie dann, wenn die Stringeingabe erwartet wird, den zweiten Teil aus, indem Sie "****" eingeben:

```
4025 IFT$="****"THEN4070
4070 INPUT"NUMMER DES ZU PRUEFENDEN ZEIC
HENS: ";NN
4080 PRINTMID$(A$,NN*4-3,4)
4090 GOTO4020
```

Man kann Elemente aus dem Feld löschen, indem man einfach ihre Positionen neu definiert als vier Leerstellen. Diese Routinen sind nicht sehr stabil, d. h. sie stürzen leicht ab, wenn man Daten eingibt, deren Länge nicht vier Zeichen beträgt. Dem kann man jedoch mit ein paar einfachen Fehlertests abhelfen, wie im nächsten Kapitel gezeigt wird.

MEHRELEMENTIGE STRINGFELDER

Wenn man Strings zum Speichern von Daten regelmäßiger Länge benutzt, besteht eine Schwierigkeit in der maximalen Länge des einzelnen Strings. Im o. g. Beispiel galt die Annahme, daß die Anzahl der abzulegenden Elemente höchstens 63 betragen darf. Wenn das auch für eine ganze Reihe von Anwendungen ausreichen dürfte, wäre für viele andere eine größere Kapazität wünschenswert. Diese kann man relativ leicht schaffen, indem man ein Stringfeld definiert und die jeweilige Position eines Elements nicht nur in bezug auf seinen Ort im String, sondern auch auf seine genaue Adresse innerhalb des ganzen Feldes berechnet. Unten sehen Sie eine Bearbeitung der vorhergehenden Routine, die mit einem 20elementigen Feld arbeitet und somit die Aufnahme von 1260 Elementen mit vier Zeichen und den Zugriff darauf ermöglicht.

```
50 DIMA$(19)
60 FORI=1TO252:A$(0)=A$(0)+"  ":NEXT
70 FORI=1TO19:A$(I)=A$(0):NEXT
5000 REM*****************************
5001 REM MEHRELEMENTIGE STRINGFELDER
5002 REM*****************************
5010 INPUT"VIERSTELLIGES ZEICHEN:
    ";T$
5020 IFT$="****"THEN5070
5030 INPUT"ZEICHENPOSITION (1-1260):
    ";T
```

```
5040 LL=INT((T-1)/63):T=T-63*LL
5050 A$(LL)=LEFT$(A$(LL),4*(T-1))+T$+MID
$(A$(LL),T*4+1)
5060 GOTO5010
5070 INPUT"NUMMER DES ZU PRUEF. ZEICHEN
S: ";NN
5080 LL=INT((NN-1)/63):NN=NN-63*LL
5090 PRINT MID$(A$(LL),NN*4-3,4)
5100 GOTO5010
```

VARIABLEN:
LL Nummer der Zeile, in die das neue Element eingesetzt wird; ergibt sich aus der Division der gewünschten Position durch die Länge der Zeilen im Feld
T/NN Ursprüngliche Nummer des einzusetzenden oder zu untersuchenden Elements, umgewandelt in die Position des Elements in Zeile LL

DATEN IN STRINGS VARIABLER LÄNGE

In den letzten beiden Abschnitten wurde von einer feststehenden Länge der bearbeiteten Strings ausgegangen. Das hat den Vorteil, daß die Position der Stringelemente ebenfalls festgelegt ist. Es gibt immer 63 Elemente, und das Element in Position 23 wird immer dort bleiben, egal was man mit den anderen Positionen anfängt. Nicht alle Anwendungen erfordern jedoch den String in voller Länge im Speicher. Häufig wird nur eine Liste von Elementen (mit oder ohne bestimmte Reihenfolge) verlangt, die man einfach nacheinander durchgehen und durch Einfügungen oder Streichungen verändern kann. Mit der nächsten Routine kann man Elemente mit vier Zeichen an den Anfang einer Liste von bis zu 1240 Elementen anfügen, die gesucht oder gelöscht werden können:

```
6000 REM*********************************
6001 REM DATENSTRINGS VARIABLER LAENGE
6002 REM*********************************
6100 DIMA$(19)
6120 INPUT"1=EINGABE/2=SUCHEN/3=LOESCHE
N/4=ENDE";FF
6130 ONFFGOSUB6200,6300,6400,6150
6140 GOTO6120
6150 END
```

```
6200 REM*********
6201 REM EINGABE
6202 REM*********
6210 PRINT:INPUT"VIERSTELLIGES ZEICHEN:
";IN$
6220 IFIT=1240THENPRINT"KEIN PLATZ:FORI=
1TO2000:NEXT:GOTO6290
6230 A$(0)=IN$+A$(0):IT=IT+1
6240 IFLEN(A$(0))<252THEN6290
6250 FORI=0TO18
6260 TT$=RIGHT$(A$(I),4):A$(I)=LEFT$(A$(
I),LEN(A$(I))-4)
6270 A$(I+1)=TT$+A$(I+1):IFLEN(A$(I+1)<2
52THEN6290
6280 NEXTI
6290 RETURN
6300 REM*******
6301 REM SUCHE
6302 REM*******
6310 PRINT:INPUT"ZU SUCHENDES ZEICHEN: "
;IN$
6320 FORI=1TOIT
6330 LL=INT((I-1)/62):PP=4*(I-LL*62)-3
6340 IFMID$(A$(LL),PP,4)=IN$THEN6370
6350 NEXTI
6360 PRINT"NICHT VORHANDEN":FORJ=1TO2000
:NEXT:GOTO6390
6370 PRINT"ZEICHEN HAT NUMMER: ";I:PRINT
:INPUT"WEITER SUCHEN";Q$
6380 IFLEFT$(Q$,1)="J"THEN6350
6390 RETURN
6400 REM**********
6401 REM LOESCHEN
6402 REM**********
6410 PRINT:INPUT"ZU LOESCHENDES ZEICHEN:
 ";IN$
6420 GOSUB6320:IFI>ITTHEN6500
6430 A$(LL)=LEFT$(A$(LL),PP-1)+MID$(A$(L
L),PP+4)
6440 IT=IT-1
```

```
6450 PRINTIN$;" GELOESCHT":FORI=1TO2000:
NEXT
6460 FORI=1TO18:IFLEN(A$(I))=248 OR A$(I
)=""THEN6490
6470 LN=248-LEN(A$(I)):A$(I)=A$(I)+LEFT$
(A$(I+1),LN)
6480 A$(I+1)=MID$(A$(I+1),LN+1)
6490 NEXTI
6500 RETURN
```

VERWENDETE VARIABLEN:
FF Nummer der vom Benutzer gewählten Funktion
IT Anzahl der Elemente mit vier Zeichen im Feld
TT$ Hilfsvariable, mit der ein Element in die nächste Zeile verlegt wird, wenn die Länge einer Zeile im Feld 252 erreicht, d. h. wenn kein neues Element angefügt werden könnte, ohne die maximale Länge von 255 zu überschreiten
LL derzeit bearbeitete Zeile innerhalb des Feldes
PP Position eines Elements in Zeile LL des Feldes
LN Länge des Elements, das beim Löschen innerhalb des Feldes nach unten verschoben wird

Sehen Sie sich die Routine in aller Ruhe an, und lassen Sie sich nicht einschüchtern; denn sie enthält wenig, das wirklich neu ist. Es sind hier lediglich einige der in diesem Kapitel beschriebenen Methoden angewendet worden. Das einzig Neue ist die Art und Weise, wie Elemente von einem String zum anderen verlegt werden, sobald die Länge eines Strings 252 erreicht. Dies geschieht, damit man ein neues Element ohne Risiko an den Anfang des Feldes anfügen kann, anstatt erst auszuprobieren, ob die Addition des neuen Elements zu einer unerlaubten Stringlänge führen würde. Beim Löschen kehrt man das Verfahren um.

GARBAGE COLLECTION

Bei komplizierten Stringroutinen, mit denen Strings innerhalb von Feldern häufig umgestellt oder im Speicher neu definiert werden, wird Ihnen gelegentlich auffallen, daß der C 64 die Verarbeitung scheinbar unterbricht. Der Schein trügt nicht, sondern das ist tatsächlich so. Der C 64 – und jedes andere Gerät, das auf Commodore BASIC 2 läuft – ordnet im Fall der Neudefinierung eines Strings den Speicher nicht so, daß der vorhandene Platz optimal genutzt wird. Strings werden nur dann im Speicher umgestellt, wenn Platz für etwas gemacht werden muß, das

an Länge zugenommen hat. Das bedeutet, daß ein durch Neudefinition verkürzter String keinen zusätzlichen Speicherplatz freimacht. Dadurch wird der Speicher nach und nach aufgefüllt, wenn große Mengen von Strings bearbeitet werden. Nach einiger Zeit wird das zum Problem, und der C 64 schafft mit der sogenannten 'Garbage Collection' Abhilfe, d. h. er ordnet den Speicher so, daß die Strings keinen überflüssigen Platz mehr belegen. Leider erfolgt die Garbage Collection nicht immer rechtzeitig genug, um den Programmstop und einen OUT OF MEMORY Error zu verhindern.
Sollten Sie bei Ihren Programmen auf dieses Problem stoßen, hilft Ihnen die FRE-Funktion des C 64 weiter, mit der Sie die Garbage Collection erzwingen können, um sich einen genauen Überblick über den verfügbaren Platz im Arbeitsspeicher zu verschaffen. Auf den Befehl

```
PRINT FRE(0)
```

(der Wert in Klammern ist ohne Bedeutung) wird die Anzahl der freien Bytes im Speicher ausgedruckt. Eine Komplikation ergibt sich daraus, daß FRE nur die Zahlen von −32768 bis 32767 verarbeiten kann. Was über 32767 hinausgeht, wird als die Anzahl freier Bytes minus 65536 ausgedrückt. Probieren Sie das nächste Programm aus, und geben Sie es genauso ein wie unten angegeben, d. h. mit der Leertaste nach PRINT in Zeile 15 und ohne Leertaste nach PRINT in Zeile 20:

```
10 DIMA(1218)
15 PRINT "␣"
20 PRINT FRE(0)
30 LIST
```

Lassen Sie das Programm laufen; jetzt sollte der Wert 32767 oben auf dem Bildschirm erscheinen, darunter das Listing. Gehen Sie zum Listing, und entfernen Sie die Leerstelle nach PRINT in Zeile 15. Da Sie ein Byte aus dem Programm genommen haben, müßte der Wert des verfügbaren Speicherplatzes jetzt mit 32768 Bytes angegeben werden, aber der erneute Programmdurchlauf gibt den Wert −32768 aus. Die Ursache dafür hängt damit zusammen, daß die FRE-Funktion mit 16-Bit-Integerarithmetik arbeitet. Streng genommen hieße das, jede Zahl von 0 bis 63535 kann ausgedrückt werden. Tatsächlich aber ist das höchste Bit einer binären Zahl reserviert für die Anzeige, ob eine Zahl negativ ist; daher wird jede Zahl über 32767, bei der alle 16 Bits gebraucht werden, als negativ gedeutet. Das Problem wird gelöst, wenn man eine logische Bedingung (Erklärung s. gesondertes Kapitel) zur Umwandlung negativer Zahlen in die korrekten positiven verwendet. Die folgende Zeile führt immer zum korrekten Wert:

```
MM=FRE(0):PRINT MM-65536*(MM<0)
```

Dies nur am Rande. Hauptsächlich sollte FRE hier als eine Möglichkeit vorgestellt werden, während des Programms die 'Garbage Collection' zu erzwingen und so einem falschen OUT OF MEMORY Error vorzubeugen. Fügen Sie einfach eine Zeile wie

```
100 T=FRE(0)
```

irgendwo ein, wo sie dann planmäßig ausgeführt wird. Da die Garbage Collection Zeit braucht und den Programmablauf etwas verlangsamen wird, sollte sie nur verwendet werden, wo sie erfahrungsgemäß notwendig ist.

SCHLUSS

Sobald Ihnen die hier beschriebenen Techniken einmal geläufig sind, brauchen Sie bei Programmen, die ausgiebigen Gebrauch von Stringfunktionen machen und voller LEFT$, MID$ und RIGHT$-Funktionen in Verbindung mit einer Fülle von Variablen stecken, nicht mehr den Mut zu verlieren. Ein Verfahren, mit dem man Strings verarbeitet, besteht im wesentlichen daraus, einen Teilstring zu identifizieren, und durch umsichtigen Einsatz von Variablen läßt sich so gut wie alles mit einem String machen. Dies eröffnet nicht nur breite Möglichkeiten für eine effektivere Datenspeicherung. Es erlaubt dem Programmierer auch, viel ausgefeiltere Programme zu schreiben, mit denen der Benutzer Strings in übersichtlicherer Form eingeben kann. Dabei bleibt dem Programm selbst die Aufgabe überlassen, die wesentlichen Teile der Eingabe zu identifizieren. Wie schon am Anfang des Kapitels erwähnt, wird auf die hier beschriebenen Techniken im folgenden häufig Bezug genommen. Überspringen Sie hier nichts, denn die Anstrengung, alles genau zu verstehen, wird sich mehr als lohnen.

KAPITEL 4

DATENEINGABE

Einer der wichtigsten Unterschiede zwischen den modernen Mikrocomputern und den leistungsfähigen Großrechnern der Vergangenheit besteht darin, daß der Mikrocomputer interaktiv ist, d. h. er steht in direktem Dialog mit dem Benutzer und erlaubt ihm den Eingriff in ein Programm, während es läuft. Vor der Erfindung des Mikrocomputers waren Computer meist als Geräte bekannt, die zuerst sowohl Programme als auch alle nötigen Daten brauchten, bevor man das Programm ohne jede Überprüfung der eingegebenen Daten durchlaufen ließ. Oft erhielt man das Ergebnis des Programmdurchlaufs erst am nächsten Tag; Fehler im Programm konnten bedeuten, daß der Vorgang viele Male wiederholt werden mußte, bevor der Benutzer auch nur eine annähernde Vorstellung davon hatte, wie das Programm nach Beseitigung aller Fehler arbeiten würde.
Infolgedessen mußte der Programmierer, wenigstens der erfolgreiche, alles vorhersehen, was im Laufe der Ausführung geschehen würde. Sollten im Programmablauf mehrere Operationen durchgeführt werden, mußten sie alle in der richtigen Reihenfolge und mit allen notwendigen Daten eingebaut werden, bevor das Programm betriebsfähig war. Sämtliche während des Programmablaufs zu treffenden Entscheidungen mußten vorher eingeplant werden; denn es gab keine Möglichkeit, ein Programm zu entwerfen, das Entscheidungen beim Benutzer erfragte. Wurde eine wichtige Entscheidung übersehen, so mußte das Programm am nächsten Tag nochmals gestartet werden.
Der moderne Mikrocomputer hat die Situation verändert. Manche neuartigen Anwendungen von Computern, wie z. B. Spiele, wären mit Geräten ohne Kommunikationsmöglichkeit zwischen Benutzer und laufendem Programm einfach nicht zu realisieren gewesen. Noch entscheidender ist, daß es für die Benutzer ernsthafter Anwendungsprogramme heute selbstverständlich ist, während der Ausführung Informationen in ein Programm einzugeben, Entscheidungen über die anfallenden Aufgaben zu treffen und dabei das Programm unter ständiger Kontrolle zu haben.
Diese Freiheit ist jedoch auch problematisch. Obwohl das nicht-interaktive Programmieren oft mühselig war, garantierte es den vorsichtigen Umgang mit Programmen und die sorgfältige Auswahl der Daten, die sie verarbeiteten. Die direkte interaktive Datenverarbeitung verführt zur Leichtsinnigkeit. Da uns die Ergebnisse des Programms fast sofort zur Verfügung stehen, kann etwas, das falsch gelaufen ist, schnell korrigiert und der Programmdurchlauf wiederholt werden.
Heute ist ein Programm nicht schon deshalb gut, weil jeder Datenausdruck so formuliert ist, daß er das Programm nicht zum Stolpern bringt, und jede mögliche wichtige Entscheidung vorausgeplant ist. Gut ist ein Programm, wenn es dem

Anwender in möglichst flexibler Weise erlaubt, Informationen einzugeben, und wenn es ihm wichtige Entscheidungen über die Arbeitsweise des Programms überträgt. Wenn man bedenkt, wie leicht während eines solchen Verfahrens Fehler gemacht werden können, beweist sich ein gutes Programm auch darin, daß der Benutzer nicht versehentlich Eingaben machen kann, die das Programm zum Absturz bringen oder zur Verstümmelung der eingespeicherten Informationen führen.
In diesem Kapitel beschäftigen wir uns mit Möglichkeiten, wie Programme Informationen annehmen können. Im nächsten Kapitel untersuchen wir einige der Methoden zum Schutz des Programms gegen mögliche Fehler, die in diesen Informationen enthalten sind.

EINGABE VON INFORMATIONEN: INPUT

Der Commodore 64 läßt zwei Arten der Dateneingabe während des Programmablaufs zu: INPUT- und GET-Anweisungen. Beide haben ihre Vorteile, obwohl sich die meisten selbstgeschriebenen Programme fast ausschließlich auf INPUT verlassen, selbst in Fällen, in denen der Gebrauch von GET weitaus angebrachter wäre.
Der wichtigste Vorzug von INPUT ist sein klarer Ablauf. Eine Abfrage erscheint auf dem Bildschirm, worauf Buchstaben oder Zahlen ein- und ausgegeben werden können. Fast ebenso wichtig ist, daß die Eingaben mit Hilfe der Cursor-Pfeile in Verbindung mit den Tasten Insert und Delete editiert werden können. Nachdem der Anwender sich davon überzeugt hat, daß das Bild genau seinen Wünschen entspricht, beendet er den INPUT durch Drücken von RETURN.
INPUT hat jedoch auch Nachteile. Erstens begrenzt es die erlaubte Menge der einzugebenden Informationen auf höchstens 80 Zeichen (selbst um 80 Zeichen einzugeben, müssen Sie den Cursor an den Anfang der Zeile *nach* der INPUT-Abfrage setzen, da nur der Inhalt zweier aufeinanderfolgender Bildschirmzeilen angenommen werden kann). Zweitens gibt es mehrere Zeichen, die von INPUT nicht korrekt verarbeitet werden, beispielsweise das Komma. Drittens bedeutet die Notwendigkeit, jede Eingabe mit RETURN abzuschließen, unter Umständen eine wirkliche Behinderung bei Programmen, die eine große Anzahl von Antworten vom Benutzer erwarten, oder wenn die Programmausführung ohne vorherige Eingabe einer Antwort durch den Benutzer fortgesetzt werden soll. Trotz der Nachteile ist die INPUT-Anweisung immer noch die Stütze fast aller Programme, die während des Ablaufs Informationen annehmen.

EINFACHE EINGABEN MIT INPUT

1. Eingabe eines einzelnen Strings:

```
10 INPUT"GIB EINE ZAHL EIN ";A$
```

2. Eingabe einer einzelnen Zahl:

```
10 INPUT"GIB EINEN STRING EIN ";A$
```

3. Eingabe mehrerer Strings:

```
10 INPUT"NAME, VORNAME, GESCHLECHT (MIT
KOMMAS EINGEBEN) ";N$,VN$,G$
```

Beachten Sie, wie in den hier eingegebenen Daten die Kommas zur Bestimmung der drei getrennten Strings dienen, die von der INPUT-Anweisung erwartet werden. Wenn der Anwender jeden Ausdruck mit RETURN statt mit Kommas abschließen würde, sähe die Anzeige so aus:

```
Bildschirm:

NAME, VORNAME, GESCHLECHT (MIT KOMMAS
EINGEBEN) ? LAWRENCE

?? DAVID

??? MAENNLICH
```

4. Eingabe mehrerer Zahlen:

```
10 INPUT"NUMERISCHE EINGABEN 1-3:";A,B,C
```

Die Wirkung ist dieselbe wie bei der Stringeingabe unter 3. In einer Eingabezeile können Zahlen und Strings gemischt vorkommen.

INPUT MEHRERER ELEMENTE IN DIESELBE BILDSCHIRM-ZEILE

Durch die flexible Handhabung des 64er-Bildschirms in Verbindung mit der INPUT-Anweisung kann das oft störende Durcheinander auf dem Bildschirm bereinigt werden. Wenn z. B. mehrere Inputs nacheinander gemacht werden müssen und der Benutzer nicht darauf angewiesen ist, sie alle zugleich vor sich zu sehen, kann man auf recht einfache Art dafür sorgen, daß alle Eingaben in dieselbe Zeile erfolgen, die dabei jeweils überschrieben wird.

1. Überschreiben von INPUTs:

```
10 PRINT"■■■■■■■■■■■■■■"
20 INPUT"□WORT 1";A1$
30 INPUT"□WORT 2";A2$
40 INPUT"□WORT 3";A3$
```

Hier ist die PRINT-Ausweisung in Zeile 10 einfach ein Beispiel dafür, wie man den Cursor an die gewünschte Bildschirmposition bringt, am besten *eine Zeile unter* der Stelle, wo die INPUT-Abfragen erscheinen sollen. Von da an reicht es, ein 'Cursor aufwärts'-Symbol an den Anfang jeder Abfrage zu setzen, damit alle Eingaben in dieselbe Zeile gemacht werden. Bei verschieden langen Eingaben ergibt sich nur das Problem, daß die frühere von der folgenden möglicherweise nicht vollständig gelöscht wird. In diesem Fall wäre jede INPUT-Zeile etwa so aufgebaut:

```
20 PRINTO$: INPUT"□WORT 1";A1$
```

O$ ist ein String mit 39 Leerstellen, denen ein 'Cursor aufwärts'-Symbol vorgeht. Bei dieser Methode löscht O$ die Zeile bis zum Ende, so daß die Eingaben immer in eine freie Zeile gedruckt werden.

2. Eingaben können natürlich auch genausogut *nebeneinander* statt untereinander auf dem Bildschirm angeordnet werden, vorausgesetzt, Sie sind sicher, daß sie nicht über die Zeile hinausgehen und so das Bildschirmformat verderben:

```
10 INPUT"WORT 1";A$
20 PRINT"■■■■■■■■■■■■■■";:INPUT"WORT 2";B$
30 PRINT"■■■■■■■■■■■■■■■■■■■■■■■■■■";:INPUT"W
ORT 3";C$
```

INPUT IN BILDSCHIRMFENSTER

Selten wirken Eingaben so professionell wie auf einem Bildschirm mit invertierten Fenstern. Obwohl man für diesen Zweck besser mit GET arbeitet, kann man auch INPUT verwenden, wenn man sicher ist, daß die einzelnen Eingaben nicht länger sind als die ihnen zugewiesenen Fenster. Das Verfahren ist in folgenden Zeilen dargestellt:

```
10 PRINT"                [REVERS EIN]
"
20 INPUT"□ WORT 1[REVERS EIN]";A$
```

Zeile 10 bringt ein 10 Spalten langes invertiertes Fenster auf den Bildschirm, vor dem genügend Platz für die geplante, nicht-invers gedruckte Abfrage bleibt. Die INPUT-Abfrage beginnt mit einem 'Cursor aufwärts'-Zeichen zwecks Rückkehr in die richtige Zeile, und [RVS] am Ende der Abfrage bewirkt die inverse Darstellung der Eingabe. RETURN am Ende der Eingabe schaltet den RVS-Modus aus, und alles folgende wird im Normal-Modus gedruckt.
Es lohnt die Mühe, mit den vielen Möglichkeiten der Gestaltung von INPUTs auf dem Bildschirm zu experimentieren. Ein übersichtlich eingeteilter Bildschirm macht die Bedienung jedes Programms wesentlich einfacher und sorgt dafür, daß bei Eingaben weniger Flüchtigkeitsfehler gemacht werden.

GET

So nützlich INPUT auch sein mag, seine Einschränkungen sind oft hinderlich. Um dem abzuhelfen, bietet 64er-BASIC mit GET einen zusätzlichen Befehl. GET ist schwieriger zu handhaben als INPUT; denn seine einzige Funktion besteht darin, die Tastatur zu lesen, d. h. festzustellen, ob eine Taste gedrückt wurde oder nicht. GET erwartet nicht RETURN, ehe es eine Eingabe als solche annimmt; jedes GET liest die Tastatur und nimmt entweder das zuerst registrierte Zeichen an oder stellt fest, daß keine Taste angeschlagen wurde. GET empfiehlt sich besonders, wenn Sie die Länge einer Eingabe sofort überprüfen möchten, wenn Einzeltastenbefehle (ohne Gebrauch von RETURN) zweckmäßig sind, oder wenn das Programm nicht auf eine Eingabe warten soll, falls der Benutzer keine Taste drückt.

WARTESTATUS MIT GET

Die klassische Aufgabe von GET besteht darin, einen Wartestatus bis zum Drücken einer einzelnen Taste herzustellen:

```
10 GETA$:IFA$=""THEN10
```

Auf diese Zeile hin führt das Programm die GET-Anweisung aus und stellt fest, ob eine Taste gedrückt wird. Falls nicht, wird dem String A$ der Wert Null zugeordnet, und die IF-Anweisung im zweiten Teil der Zeile bewirkt die nochmalige Ausführung der Zeile. Das Programm wartet also unbegrenzt, bis eine Taste gedrückt wird. Sobald dies geschieht, übernimmt A$ den Wert des Zeichens, das der Taste entspricht, und das Programm geht zur nächsten Zeile über.
Ausgeklügelte Anwendungsprogramme, die dem Benutzer in verschiedenen Stadien der Ausführung Optionen anbieten, machen von GET in dieser Form sehr häufig Gebrauch. Wenn verschiedene Möglichkeiten zur Auswahl stehen, ist es viel leichter, eine einzelne Taste zu drücken, als jede Eingabe mit RETURN abschließen zu müssen. Unten sehen Sie ein typisches Programm-Menü, das mit GET arbeitet:

```
1000 PRINT"␣0 = PROGRAMMENDE"
1010 PRINT"1 = EINGABE DATEN"
1020 PRINT"2 = LOESCHEN DATEN"
1030 PRINT"3 = AENDERN DATEN"
1040 PRINT"4 = ANZEIGE DATEN"
1050 PRINT"■BITTE WAEHLEN"
1060 GETIN$:IFIN$=""THEN1060
1070 TT=VAL(IN$)+1
1080 ON TT GOSUB10000,2000,3000,4000,500
0
1090 GOTO1000
```

Der Benutzer muß lediglich 0, 1, 2, 3 oder 4 eingeben, um den entsprechenden Programmteil aufzurufen.

CURSORSTEUERUNG MIT GET

Auch Spielprogramme machen ausgiebigen Gebrauch von GET, vor allem, um Gegenstände auf dem Bildschirm hin- und herzubewegen:

```
1000 REM*****************
1001 REM BEWEGTER CURSOR
1002 REM*****************
1010 GETT$
1020 PRINT"*■";
1030 FORI=1TO50:NEXT
1040 PRINT" ■";
```

```
1050 FORI=1TO50:NEXT
1060 IFT$=""THEN1000
1070 IFT$="□"ORT$="■"ORT$="■"ORT$="■"THE
NPRINTT$;
1080 GOTO1000
```

Diese Routine läßt einen blinkenden '*'-Cursor auf dem Bildschirm erscheinen, bis eine Taste gedrückt wird. Die beiden Schleifen erzeugen eine kurze Pause, damit der Cursor nicht flackert, sondern blinkt. Um den Cursor zu bewegen, müssen Sie nur das mit GET-Anweisung in T$ abgelegte Zeichen überprüfen und, falls es eines der Cursor-Steuerzeichen ist, drucken (PRINT). Die Printposition verändert sich, und danach können Sie die Routine für den blinkenden Cursor wieder ausführen.

GET UND BEFRISTETE ANTWORTZEITEN

Bei Spielen und manchmal auch bei ernsthafteren Anwendungen kann es günstiger sein, dem Benutzer eine bestimmte Zeitspanne für eine Antwort zur Verfügung zu stellen, als das Programm auf unbegrenzte Zeit zu stoppen. Mit GET und einer angemessenen Zeitschleife ist das ein einfacher Vorgang:

```
10 FORI=1TO1000
20 GETT$:IFT$<>""THENI=1000
30 NEXTI
40 IFT$<>""THENGOSUB500
```

Hierbei hat der Benutzer zum Antworten soviel Zeit, wie die Schleife für 1000 Ausführungen braucht. Nach erfolgter Antwort kann eine bestimmte Operation wie in Zeile 40 ausgeführt werden. Wurde während der 1000 Wiederholungen der Schleife keine Taste gedrückt, läuft das Programm von der auf Zeile 40 folgenden Zeile an weiter.

GET UND INVERTIERTE FENSTER

Wie schon erwähnt, kann man die Dateneingabe wirkungsvoll gestalten, indem man einen besonderen Raum wie z. B. ein invertiertes Fenster dafür reserviert. GET spielt bei dieser Anwendungsmethode eine wesentliche Rolle, weil es eingesetzt werden kann, um die Länge einer Eingabe während des Eingabevorgangs festzustellen. Die nächste Routine nimmt jede Eingabe bis zu einer Länge von 10 Zeichen an:

```
2000 REM**************************
2001 REM EINGABE IN REVERSES FELD
2002 REM**************************
2010 IN$=""
2020 PRINT"■        [REVERS EIN]
 ":REM 7 LEER REV.ON 10 LEER
2030 PRINT"■WORT 1:[REVERS EIN]";
2040 GETT$:IFT$=""THEN2040
2050 IFT$=CHR$(13)THEN2110
2060 IFT$=CHR$(20)ANDLEN(IN$)=0THEN2040
2070 IFT$=CHR$(20)THENIN$=LEFT$(IN$,LEN(
IN$)-1):PRINT"■■";:GOTO2040
2080 PRINTT$;
2090 IN$=IN$+T$
2100 IFLEN(IN$)<10THEN2040
2110 STOP
```

Das Fenster wurde gedruckt und die Abfrage direkt davor gestellt, wobei die Printposition am Anfang des Fensters stehenbleibt. In das Fenster können dann Buchstaben geschrieben werden. Drücken der Taste DEL (CHR$(20)) löscht den letzten Buchstaben der Eingabe sowohl auf dem Bildschirm als auch im Speicher, sofern das Fenster schon etwas enthält; bei dieser Routine können Sie die Cursor-Steuertasten nicht benutzen, nur DEL. Die Eingabe ist beendet, wenn entweder RETURN (CHR$(13)) gedrückt wird, oder wenn sie eine Länge von 10 Zeichen erreicht hat.
Mit einem solchen Programm können Sie verhindern, daß eine Eingabe länger wird als Sie möchten; damit wird sowohl eine Abweichung vom geplanten Bildschirmaufbau als auch eine Verstümmelung der Struktur der Felder ausgeschlossen.

BILDSCHIRMEDITIERUNG MIT GET

GET bietet weiterhin die Möglichkeit, die bei INPUT vorgegebene Höchstlänge von Strings zu überschreiten und vorhandene Strings zu editieren, was mit INPUT sehr schwierig ist. Die Methode, auf dem Bildschirm erscheinende Informationen aufzubereiten und diese Modifikationen aufzuzeichnen, nennt man Bildschirmeditierung. Programm-Listings können beim C 64 editiert werden, indem man mit dem Cursor vorhandene Zeilen entlangfährt und etwas hinzufügt oder wegnimmt. Das Verfahren kann auch bei Strings auf dem Bildschirm benutzt werden, obwohl solche Anwendungen meist schwierig sind.

Die unten abgedruckte Routine ist ziemlich komplex, hat aber einen einfachen Zweck. Sie erlaubt dem Benutzer die Eingabe und gleichzeitige Editierung eines Strings (genannt A$). Wäre der Anfang von A$ mit einem schon im Speicher befindlichen String identisch, könnte die Routine auch dazu dienen, ihn auf den Bildschirm auszugeben und zu verändern. Die einzige Beschränkung besteht in der absoluten Höchstgrenze von 255 Zeichen pro String.

```
3000 REM**********************
3001 REM BILDSCHIRMEDITIERUNG
3002 REM**********************
3010 DEF FNA(P)=1024+LL*40+P
3020 A$="  "
3030 P=0:LL=17
3040 PRINT"
3050 PRINTA$
3060 CH=PEEK(FNA(P)):POKE54272+FNA(P),14
:POKE FNA(P),160
3070 FORTT=1TO5:NEXTTT:POKEFNA(P),CH
3080 GETT$:IFT$=""THEN3040
3090 IFT$=CHR$(13)ORLEN(A$)=255THENSTOP
3100 IFT$=CHR$(95)ANDP<>0THEN3030
3110 IFT$=CHR$(95)ANDP=0THENP=LEN(A$)-1:
GOTO3040
3120 IFP>0ANDT$=CHR$(20)THENA$=LEFT$(A$,
P-1)+MID$(A$,P+1):P=P-1
3130 IFT$=CHR$(20)THEN3040
3140 IFT$="□"ORT$="■"THEN3040
3150 IFT$<>"■"ANDT$<>"■"THENA$=LEFT$(A$,
P)+T$+MID$(A$,P+1):P=P+1
3160 IFT$="■"ANDP>0THENP=P-1
3170 IFT$="■"ANDP<LEN(A$)-1THENP=P+1
3180 GOTO3040
```

Im einzelnen enthält die Routine folgendes:

3000 Dies ist eine für die Bildschirmeditierung wichtige Funktion. Sie PEEKt eine einzelne Zeichenposition auf dem Bildschirm. Wir brauchen diese Funktion, um später zu ermitteln, was sich schon auf dem Bildschirm befindet, und um es gegebenenfalls in dieselbe Position wieder einsetzen zu können. Vom Anwender definierbare Funktionen werden in Kapitel 12 ausführlicher beschrieben.

3020 Um überhaupt mit dem Editieren eines Strings beginnen zu können, brauchen wir zunächst einmal einen String. Diese Zeile geht davon aus, daß wir ganz von vorn anfangen, und beginnt den String als einzelne Leerstelle. Wäre hier die Editierung eines schon existierenden Strings beabsichtigt, so würde er vorläufig in A$ umbenannt werden, und diese Zeile würde dazu dienen, eine Leerstelle an das Ende des Strings anzuhängen.

3030 Die Variable P wird in der Routine gebraucht, um die Cursorposition im String aufzuzeichnen, wobei die Numerierung bei Null beginnt.

3040—3050 Diese Zeilen drucken den gewünschten String in eine Zeile, die durch die Anzahl der 'Cursor abwärts' bestimmt ist.

3060 Diese Zeile ermittelt mit Hilfe der definierten Funktion den Bildschirmcode-Wert des Zeichens in Position P in Zeile LL und speichert ihn in die Variable CH ab. An seine Stelle wird eine inverse Leerstelle gePOKEt.

3070 Während der Ausführung der kleinen Schleife bleibt die inverse Leerstelle für einen Sekundenbruchteil auf dem Bildschirm sichtbar, dann wird das ursprüngliche Zeichen wieder eingesetzt.

3080 Diese Zeile läßt den Cursor einmal blinken und prüft, ob eine Taste gedrückt wird. Wenn nicht, blinkt der Cursor nochmals.

3090 Um hierher zu gelangen, muß der Benutzer eine Taste gedrückt haben. Die Zeile prüft, ob die gedrückte Taste RETURN war, oder ob der zu editierende String schon 255 Zeichen umfaßt. In beiden Fällen wird die Routine beendet. Normalerweise benutzt man hier nicht STOP, sondern springt mit GOTO in einen anderen Programmteil, der den neugeschaffenen String verarbeitet. Vor Gebrauch des neuen Strings müßte man das letzte Zeichen – die am Anfang eingefügte Leerstelle – entfernen.

3100 Handelt es sich bei dem eingegebenen Zeichen um den 'Pfeil nach links' links oben auf der Tastatur, geht die Printposition (P) an den Anfang der Zeile zurück, falls sie sich nicht schon dort befindet.

3110 Wenn bei Eingabe des 'Pfeils nach links' die Printposition schon am Zeilenanfang steht, springt sie zum Zeilenende. Diese beiden Zeilen sind nur aufgenommen worden, um die Bewegungen innerhalb des Strings zu erleichtern.

3120–3130 Sofern die Printposition nicht am Zeilenanfang steht, löscht die Delete-Taste das Zeichen links vom blinkenden Cursor. Das Programm springt dann zur Routine für den blinkenden Cursor zurück.

3140–3150 Das eingegebene Zeichen wird – wenn es sich nicht um eine der Cursor-Steuertasten handelt – an der Cursorposition in den String eingesetzt, wobei sich alle Zeichen von der Cursorposition an um eine Stelle nach rechts verschieben. Dann wird der String wieder gedruckt.

3160–3180 Diese Zeilen ermitteln, ob das eingegebene Zeichen ein 'Cursor nach links(rechts)'–Pfeil war. Wenn ja, ist es überflüssig, den String wieder zu drucken. Geändert wird lediglich die Position des blinkenden Cursors.

Mit dem vorgestellten Programm und mit etwas Phantasie werden Sie in der Lage sein, Daten an beliebiger Stelle auf den Bildschirm zu schreiben, sie mit Hilfe der 'links-' und 'rechts'-Cursorpfeile zu editieren, Zeichen einzufügen oder zu entfernen und die editierten Daten wieder abzuspeichern. Wenn Sie es wünschen, können Sie auch eine weitere Variable hinzufügen, um die Position des Cursors sowohl vertikal als auch horizontal aufzuzeichnen. Das gibt Ihnen die Möglichkeit, sich in den verschiedenen Textzeilen auf- und abwärts zu bewegen, während das Programm laufend registriert, welche der Zeilen gerade editiert wird. Für den Benutzer ist es viel leichter, so vorzugehen, als einen String zur Überprüfung aufrufen und in geänderter Form vollständig neu eingeben zu müssen. Bei Programmen, die ständige Aktualisierung und Änderung von Daten erfordern, kann eine solche Routine vieles erleichtern.

EINFACHE BILDSCHIRMEDITIERUNG MIT INPUT

Bei aller Flexibilität einer Routine der oben beschriebenen Art ist es auch möglich, die Bildschirmeditierung mit INPUT vorzunehmen, obwohl dabei nicht mehr als 80 Zeichen eingegeben werden können und jeweils nur mit einem String gearbeitet werden kann. Geben Sie dazu einfach den zu editierenden String bzw. die Zahl auf den Bildschirm aus, und lassen Sie davor genügend Platz für die INPUT-Abfrage:

```
100 A$="ABCDEFG"
110 PRINT TAB(9);A$
120 INPUT"□WORT 1 ";A$
```

Mit dieser Methode können Sie von allen automatischen Funktionen des Bildschirmeditors, die das Betriebssystem des C 64 anbietet, Gebrauch machen; wenn

Sie RETURN drücken, nimmt der C 64 dann alles, was auf die Abfrage folgt, als neue Form von A$ an.

SCHLUSS

Die Dateneingabe kann ein ansonsten ausgezeichnetes Programm leicht ruinieren. Werden Daten auf einem vollgestopften Bildschirm mit unklarer Bedeutung und ohne erkennbare Reihenfolge abgefragt, dann wird das Eingeben zu einer fehlerträchtigen und sicherlich mühsamen Angelegenheit. Hier ist sorgfältiges Formatieren und die zusätzliche Verwendung von Farben ausschlaggebend, indem z. B. aufeinanderfolgenden Abfragen verschiedene Farben zugeteilt werden und jede Abfrage mit einem schwarzen Steuerzeichen abgeschlossen wird, das die Antworten von den Abfragen selbst auf dem Bildschirm abhebt.
Wählen Sie die Methode und das Format nach Ihrem persönlichen Geschmack. Jetzt, da Ihnen die oben skizzierten Techniken zur Verfügung stehen, gibt es aber keinen Grund mehr, warum sich Ihr Programm weiterhin in langweiligen Listen einfarbiger Abfragen auf dem Bildschirm erschöpfen sollte. Später werden wir sehen, wie Sie zusätzlich noch kontrollieren können, ob die auf Abfragen erhaltenen Daten korrekt sind.

KAPITEL 5

FEHLERKONTROLLE

So etwas wie ein idiotensicheres Programm gibt es nicht. Im günstigsten Fall kann man von einem Programm sagen, daß es bisher noch keinen Dummkopf gab, der phantasievoll genug gewesen wäre, es abstürzen zu lassen. Kennen Sie den Ärger, wenn ein schönes, brauchbares Programm im entscheidenden Moment abstürzt, weil der Benutzer etwas eingibt, das nicht wie üblich bearbeitet werden kann? In diesem Kapitel werden wir uns mit einigen Möglichkeiten befassen, wie ein Programm stabiler, d. h. weniger anfällig gegen die Eingabe unerwarteter oder abwegiger Daten gemacht werden kann. Das Kapitel enthält sehr wenige komplexe Techniken, denn Fehlerkontrolle ist größtenteils eine Sache des gesunden Menschenverstandes: ein Versuch, Fehler zu erkennen, die von Benutzern wahrscheinlich gemacht werden, und ihnen zuvorzukommen.

FEHLERVORSORGE – VERNÜNFTIGE VORSICHTSMASSREGELN

Warum müssen die Leute unbedingt Fehler machen, wenn sie mit Ihrem besten Programm arbeiten? Darauf gibt es viele Antworten, aber meist läuft es darauf hinaus, daß Ihr Lieblingsprogramm doch nicht ganz so gut ist, wie Sie meinen. Die meisten Eingabefehler entstehen, weil das Programm einfach nicht deutlich genug zeigt, in welcher Weise die verlangten Informationen vom Benutzer einzugeben sind. Vielleicht sind die Abfragen in den INPUT-Anweisungen zu knapp, oder der Bildschirm ist zu voll, um sich auf jede Abfrage richtig konzentrieren zu können. Vielleicht ändern Sie auch mitten im Programm die festgelegten Arbeitsbedingungen und erwarten plötzlich unausgesprochen eine alphabetische Eingabe, nachdem Sie vorher die Antworten immer in numerischer Form verlangt hatten. Was auch der Grund sein mag, es ist sinnlos, sich über die Dummheit der Leute zu beschweren, die Ihre Programme abstürzen lassen, denn Programme sind für den Gebrauch da, und deshalb sollten sie so entworfen sein, daß der Benutzer sich jederzeit über die laufenden Vorgänge im klaren ist.
Wenn wir das im Auge behalten, können wir mit ein paar vernünftigen Grundsätzen anfangen, die mögliche Fehler zum größten Teil ausschließen werden:

1. Machen Sie sich die Mühe, den Bildschirm mit Hilfe der im vorigen Kapitel beschriebenen Methoden sorgfältig zu formatieren. Die Bildschirmgestaltung sollte die Aufmerksamkeit des Benutzers jeweils auf die richtige Abfrage lenken und von allem Überflüssigen ablenken. Vermeiden Sie Unordnung auf dem Bildschirm.

2. Lassen Sie möglichst jemand anderen einen Blick auf Ihre Abfragen werfen, bevor Sie das Programm für fertig erklären. Nachdem Sie mit der Entwicklung des Programms Tage oder gar Wochen verbracht haben, kennen Sie den Zweck jeder eingegebenen Information zweifellos im Schlaf. Andere Benutzer jedoch haben nur eine vage Idee vom Inhalt des Programms und können sich überhaupt nicht vorstellen, was die einzelnen Eingaben bezwecken, wenn sie nicht klar zu entziffern sind. Das gilt selbst für den Fall, daß Sie nicht beabsichtigen, Ihr Programm anderen zugänglich zu machen. Wenn Sie es eine Zeitlang beiseitegelegt haben und sich dann wieder damit beschäftigen, sind Sie darüber möglicherweise genauso verwirrt wie jeder andere.

3. Geben Sie das Format der Eingabe genau an, wo immer es nicht eindeutig ist: Sollen Zahlen oder Buchstaben eingegeben werden? Gibt es einen maximalen Wert, ist die Anzahl der Zeichen begrenzt? Sollen die Elemente durch Kommas voneinander getrennt werden? . . . Nehmen wir als Beispiel ein Menü, das auf dem Bildschirm erscheint:

```
0 - PROGRAMM ENDE
1 - EINGABE NEUES WORT
2 - LOESCHEN WORT
3 - SUCHEN WORT
3 - ANZEIGE DATEN
```

Das sieht eigentlich klar aus, aber Sie können davon ausgehen, daß mancher daraufhin die Buchstaben 'Display Data' eingeben und sich dann fragen wird, warum nichts passiert, oder warum das Programm anhält. Die Abfrage hätte richtig heißen müssen 'Welche Zahl wählen Sie?'. In anderen Worten: Wenn das Programm die Eingabe in einer bestimmten Weise erwartet, sollte es den Benutzer zuerst darüber informieren.

4. Machen Sie keine zu langen oder komplizierten Input-Kombinationen. Wenn Sie wirklich 10 Elemente hintereinander stellen müssen, zerlegen Sie sie auf dem Bildschirm sichtbar in logische Gruppen, und lassen Sie evtl. Freiräume dazwischen. Paradoxerweise unterlaufen dem Benutzer bei komplexen Eingaben um so mehr Fehler, je mehr er sich an das Programm gewöhnt, weil er dann die Abfragen weniger sorgfältig lesen wird. Daß jemand bei den ersten Malen ein Programm richtig bedient hat, schließt nicht aus, daß er nach der Eingewöhnungszeit anfängt, geistesabwesend die falschen Abfragen zu beantworten.

5. Halten Sie auf jeden Fall während des gesamten Programms dieselben Konventionen ein. Wenn Eingeben von '0' normalerweise eine laufende Funktion ausschal-

tet, an anderer Stelle aber ein Datenelement löscht, brauchen Sie sich nicht zu wundern, wenn Sie sich bei jemandem unbeliebt machen, dem wichtige Daten verlorengegangen sind, obwohl er doch eigentlich nur zum Menü zurückspringen wollte.

6. Seien Sie sehr vorsichtig mit Inputs, deren Reihenfolge sich manchmal ändert. Wenn Sie in einem Dateiprogramm gewöhnlich nach Namen, Adresse und Alter fragen, aber bei einer Altersangabe von über 65 eine weitere Information brauchen, schreiben Sie die Zusatzabfrage nicht einfach ohne Vorwarnung auf den Bildschirm. Der Benutzer hat sich daran gewöhnt, drei Eingaben zu machen und wird aller Wahrscheinlichkeit nach wieder RETURN drücken, falls die Daten nach dem dritten Input nicht gleich vom Bildschirm verschwinden. Wird die dem Benutzer geläufige Reihenfolge verändert, lassen Sie den Bildschirm kurz andersfarbig aufleuchten, oder geben Sie ein akustisches Signal als Hinweis auf den geänderten Ablauf.

7. Gehen Sie besser davon aus, daß trotz Ihrer Bemühungen immer Fehler aus Müdigkeit, Langeweile oder reiner Sorglosigkeit gemacht werden. Solchen Fehlern kann man jedoch vorbeugen, indem man die soeben eingegebene Information in einem anderen Format neu ausdruckt und die Bestätigung der Richtigkeit verlangt. Bei dieser Gelegenheit könnte dem Benutzer beispielsweise auffallen, daß Alter und Telefonnummer vertauscht worden sind. Die Wiederholung einer Eingabe auf dem Bildschirm bedeutet auch: Es geschieht nichts damit, bevor der Benutzer die Korrektheit der Information bestätigt hat. Wird eine Eingabe direkt in das Hauptdatenfeld übergeben, ist es sehr schwer, die Situation noch zu retten, falls der Benutzer darin einen Fehler entdeckt.

FEHLERKONTROLLE – EINFACHE PROGRAMMIERTECHNIKEN

Mit einfachen Methoden wie den oben beschriebenen läßt sich die Fehlerquote in einem Ausmaß senken, das den Aufwand weit übertrifft. Da jedoch immer Fehler gemacht werden, prüfen wir jetzt einige Verfahren, mit denen man das Programm durch eingebaute Kontrollen dagegen absichert.

BESTIMMUNG VON GRENZWERTEN

Eine der bekanntesten Ursachen für Programmunterbrechungen besteht darin, daß in ein Programm, das planmäßig mit Daten innerhalb festgelegter Grenzen arbeiten soll, ein außerhalb dieses Bereichs liegender Ausdruck eingegeben wird. Wenn Sie

z. B. ein Programm schreiben, in das zwei Zahlen eingegeben werden, worauf die erste durch die zweite dividiert wird, bricht es mit einer Fehlermeldung ab, falls die zweite Zahl Null ist. Natürlich hätte die Abfrage den Benutzer darauf hingewiesen müssen, daß die zweite Zahl größer als Null sein muß, aber selbst dann noch muß man immer auf einen Tippfehler oder besondere Begriffsstutzigkeit gefaßt sein. Man tut gut daran, in ein Programm, das nur innerhalb bestimmter Grenzen funktioniert, ein paar einfache Sicherungen einzubauen, damit keine außerhalb dieser Schranken liegenden Daten eingegeben werden können. Problematisch sind z. B. häufig der Wert einer Zahl und die Länge eines Strings.

1. Der Wert einer Zahl

Zuallererst sollten Sie beim Entwerfen und Austesten Ihres Programms den zulässigen Wertebereich für numerische Eingaben festlegen. Im o. g. Beispiel ist Null ein klarer Fall einer ungültigen Eingabe. Sind solche Einschränkungen einmal bekannt, kann man sie leicht in eine Fehlertestzeile einbauen. Angenommen, es wird z. B. eine Zahl von eins bis zehn als Eingabe verlangt:

```
1000 INPUT"ZAHL ZWISCHEN 1 UND 10 EINGEB
EN";NN
1010 IFNN>=1ANDNN<=10THEN1050
1020 PRINT"ZAHL NICHT ZWISCHEN 1 UND 10!
"
1030 FORI=1TO2000:NEXTI:PRINT"□□";O$:PRI
NTO$:PRINT"□□";
1400 GOTO1000
```

Diese Routine kontrolliert lediglich, ob die Eingabe innerhalb der bestimmten Grenzen liegt und meldet anderenfalls für die Dauer der Schleife einen Fehler. Durch zweimaliges Drucken von O$, einem Leerstring von 39 Zeichen, werden dann Fehlermeldung und ursprüngliche Eingabe gelöscht, worauf der Benutzer einen neuen Versuch machen kann. Es ist nicht möglich, einen langen String über beide Zeilen zu drucken, weil der C 64 sich das Überschreiben des Zeilenendes merkt und den nächsten INPUT so behandelt, als hätte er zwei Zeilen erhalten. Indem er die Printposition zu weit nach unten verschiebt, zerstört er das Format. Obwohl die Meldung an den Benutzer und das Löschen der ursprünglichen Eingabe mehr Zeilen in Anspruch nehmen als die einfache Rückkehr zur ursprünglichen INPUT-Anweisung, lohnt es sich, weil der Benutzer so erfährt, *warum* die Eingabe nicht angenommen wurde. Die Zurückweisung einer Eingabe aus unbekannten Gründen ist für den Benutzer sehr frustrierend.

2. Die Länge eines Strings

Wie der Wert einer Zahl vom zulässigen Bereich abweichen und damit das Programm unterbrechen kann, so vertragen es manche Programme nicht, wenn sie einen String bestimmter Länge erwarten und nicht bekommen. Die Absicherung dagegen unterscheidet sich kaum von der bei Zahlen verwendeten Methode:

```
1000 INPUT"EINGABE (LAENGE 1 BIS 10)";A$

1010 IFLEN(A$)>=1ANDLEN(A$)<=10THEN1050
1020 PRINT"FALSCHE LAENGE!"
1030 FORI=1TO2000:NEXTI:PRINT"□□";0$:PRI
NTO$:PRINT"□□";
1040 GOTO1000
```

Hier wird kontrolliert, ob die Länge des Strings im Bereich 1–10 liegt, wobei die Zeilen genau denen der o. g. Zahlenprüfmethode entsprechen. Für Stringeingaben können Sie mit einem zusätzlichen Befehl vor der INPUT-Anweisung noch eine weitere Sicherung einbauen:

Dadurch wird berücksichtigt, daß A$ wie vor dem INPUT bleibt, wenn RETURN ohne vorherige Eingabe gedrückt wird. Hätte also A$ schon den Wert 'SMITH', würde ein versehentliches RETURN daran nichts ändern, und der Längentest würde umgangen.

VERSTÜMMELTE EINGABEN

Der Benutzer wird öfter – sei es durch Tippfehler oder falsches Lesen von Abfragen – etwas eingeben, das weniger die Beschränkungen verletzt als unverständlich für das Programm ist. Bei numerischen Eingaben geschieht das gewöhnlich, wenn man versehentlich einen Buchstaben statt einer Zahl schreibt. Bei Strings kann es vorkommen, daß der eingegebene String keinem der im Programm vorgesehenen und z. B. als Befehl verstandenen Strings entspricht.

1. Ungültige Zahlenformate

Dieser Fehlertyp tritt auf, wenn der Benutzer etwa eine Abfrage bezüglich 'Alter' irrtümlich mit einer Stringeingabe beantwortet, z. B. mit dem Teil einer Adresse. Das Ergebnis ist natürlich Unsinn, da diese Eingabe für den C 64 normalerweise den

Wert Null hätte (es sei denn, sie beginnt mit einer oder mehreren Zahlen). Die einzige Möglichkeit, sich gegen derartige Fehler wirksam zu schützen, besteht darin, Zahlen als Strings einzugeben und dann jedes Zeichen daraufhin zu prüfen, ob es eine gültige Ziffer ist. Nachstehend ein typisches Programmbeispiel:

```
1000 NN$=""
1010 INPUT"EINGABE ZAHL ";NN$
1020 IFNN$=""THENNN=0:GOTO1080
1030 FORI=1TOLEN(NN$)
1040 IFMID$(NN$,I,1)>="0"ANDMID$(NN$,I,1
)<="9"THEN1070
1050 PRINT"DIESE ZAHL IST NICHT GUELTIG"
:FORJ=1TO2000:NEXTJ
1060 PRINT"□□";0$:PRINTO$:PRINT"□□";:GOT
O1010
1070 NEXTI
```

Das Verfahren verzögert die Verarbeitung einer Eingabe nicht wesentlich und ermittelt Fehler, die dem Benutzer sonst nicht gemeldet würden. Falsche Zahlenformate führen nicht zur Unterbrechung des Programms, sie haben nur zur Folge, daß aus den Eingaben unbeabsichtigte Werte berechnet werden.
Bedenken Sie, daß die hier aufgeführte Routine keine Sicherung gegen einfaches Drücken von RETURN enthält; eine solche Eingabe wird als Null interpretiert. Diesem Fall können Sie durch die Aufnahme einer zusätzlichen Meldung wie KEIN INPUT vorbeugen.

2. Unbekannte Strings

Wenn ein Programm so angelegt ist, daß es nur einen Befehl annimmt, der aus einem von mehreren möglichen Strings besteht, kann durch Vertippen beim Eingeben des Befehls Verwirrung entstehen. Als Beispiel nehmen wir ein Programm, das die Eingabe von Daten für jeden Monat des Jahres erlaubt. Man könnte den Benutzer einfach die jeweilige Monatszahl eingeben lassen, aber das wäre sehr fehlerträchtig. Deshalb ist zu entscheiden, ob nicht der Name des Monats, für den die Eingabe gilt, ausgeschrieben werden sollte. In diesem Fall wäre es notwendig zu kontrollieren, ob die Eingabe sinnvoll war:

```
1 DIMMO$(12)
1000 MM$=""
1010 INPUT"MONATSNAME ";MM$
1020 FORI=0TO11
1030 IFMM$=MO$(I)THEN1070
```

```
1040 NEXTI
1050 PRINT"FALSCHER MONATSNAME":FORP=1TO
2000:NEXT
1060 PRINT"□□";O$:PRINTO$:PRINT"□□";:GOT
O1010
```

Voraussetzung ist hier, daß ganz vorn im Programm die Monatsnamen in den Elementen 0–11 des Feldes MO$(I) gespeichert wurden. Die Routine vergleicht die eingegebenen Daten mit den aufgezeichneten Monatsnamen und druckt eine Fehlermeldung, wenn sie kein Gegenstück vorfindet.

VERNÜNFTIGE FEHLERKONTROLLE: 'DER ZWEITE BLICK'

Wir haben schon im Abschnitt über vernünftige Vorsichtsmaßregeln gesehen, daß eine der wirksamsten Methoden zur Senkung der Fehlerquote bei Eingaben darauf beruht, den Benutzer zu zwingen, einen zweiten Blick auf seine Eingabe zu werfen und ihre Richtigkeit zu bestätigen. Soll das während des Programmlaufs regelmäßig geschehen, können Sie diese Aufgabe einem Unterprogramm übertragen; das spart Platz und garantiert gleichzeitig eine einheitliche Darstellung für das ganze Programm. Das Beispiel für ein solches Unterprogramm lautet:

```
1000 PRINTLEFT$(CC$,LL+1);
1010 FORI=0TONQ-1
1020 PRINT"[GRUEN]";PP$(I);":";:INPUT"[S
CHWARZ]";QQ$(I)
1030 NEXTI
1040 GOSUB1120
1050 PRINTLEFT$(CC$,LL+1);
1060 FORI=0TONQ-1
1070 PRINT"[GELB]";PP$(I)":[REVERS EIN]"
;TAB(20);QQ$(I)
1080 NEXTI
1090 INPUT"■RICHTIG (J/N)";TT$
1100 GOSUB1120:IFLEFT$(TT$,1)<>"J"THEN10
00
1110 RETURN
1120 PRINTLEFT$(CC$,LL+1);
1130 FORI=1TONQ+2:PRINTO$:NEXT
1140 RETURN
```

VERWENDETE VARIABLEN:
LL Zeilenposition der ersten Abfrage
NQ Anzahl der Fragen
PP$ Feld, das die Abfragen enthält

Die Routine stellt eine Reihe von Fragen in Abhängigkeit von der Variablen NQ. Die Fragen selbst werden vor Aufruf des Unterprogramms in das Feld PP$ abgelegt. Antworten werden in QQ$ gespeichert. Nachdem alle Fragen gestellt worden sind, werden sie zusammen mit den invers gedruckten Antworten wieder ausgegeben. Antwortet der Benutzer nicht mit J auf die Frage RICHTIG?, werden Fragen und Antworten gelöscht und nochmals gestellt.
Die Bildschirmformatierung geschieht mit Hilfe eines String CC$, der so aufgebaut ist:

```
CC$="
```

und der betreffende Bildschirmbereich wird mit dem üblichen, aus 39 Leerstellen bestehenden String O$ gelöscht.
Das eigentliche Format der Abfragen und die Art, wie sie erneut vorgelegt werden, ist Geschmacksache; wichtig ist nur, daß eine veränderte Gestaltung es dem Benutzer erleichtert, sich ein zweitesmal darauf zu konzentrieren. Wenn Sie die Informationen am Schluß der Routine nicht auf dem Bildschirm stehen lassen möchten, können Sie zum Löschen der entsprechenden Zeilen das Unterprogramm in Zeile 1200 aufrufen.
Für den Aufruf der Routine brauchen Sie in Ihrem Hauptprogramm folgende oder ähnliche Zeilen:

```
500 LL=10:NQ=3
510 PP$(0)="NAME"
520 PP$(1)="ADRESSE"
530 PP$(2)="TELEFON"
540 GOSUB1000
550 FORI=1TONQ:AA$(I)=QQ$(I):NEXT
```

Nachdem das o. g. Abfrageprogramm eingegeben ist, legen diese kurzen Zeilen die Position der ersten Abfrage auf dem Bildschirm (in Zeilen) und die Anzahl der zu stellenden Fragen fest und formulieren dann die Abfragen. Danach formatiert das Unterprogramm die Abfragen und fordert den Benutzer auf, die Richtigkeit der Antworten zu bestätigen. Die Abfragen werden solange wiederholt, bis die korrekten Antworten gegeben werden. Beim Verlassen des Unterprogramms werden die

Antworten im Feld QQ$ festgehalten und können an jeden beliebigen Ort übergeben und auf Dauer gespeichert werden.
Natürlich hätte es wenig Sinn, eine Routine wie diese in einem kurzen Programm oder in einem, das überhaupt nur ein oder zwei Abfragen enthält, zu verwenden. Wenn ein Programm jedoch eine ganze Reihe von Fragen stellt, kann das Unterprogramm für exaktere Antworten sorgen; gleichzeitig kann es der Notwendigkeit abhelfen, für jede vorkommende Abfrage eigens die Testroutine einzubauen.

SELBSTENTWORFENE FEHLERMELDUNGEN: DIE ELEGANTE LÖSUNG

Bis jetzt haben wir nur Routinen berücksichtigt, die sich zum Einbau in diverse Programmteile eignen, um dort verschiedene Fehlerkontrollen – hauptsächlich im Hinblick auf falsche Eingaben – durchzuführen. Das ganze Verfahren setzt voraus, daß man schon bei der Eingabe erkennen kann, ob sie falsch ist. Dies ist nicht immer möglich. Manchmal gibt man etwas ein, das weder zu lang ist noch einen zu niedrigen Wert hat, nicht leicht auf Schreibfehler hin zu überprüfen ist, und das trotzdem zu Schwierigkeiten im Programm führen kann. Das Problem dabei ist, daß die tatsächlich entstandene Schwierigkeit nicht erkennbar ist, bevor die eingegebenen Daten nicht wenigstens zum Teil vom Programm verarbeitet worden sind. Bei einiger Voraussicht könnte man wohl überblicken, wo solche Fehler auftreten können, und die entsprechenden Stellen mit Tests und Fehlermeldungen ausstatten.
Damit sind jedoch nicht alle Probleme gelöst. Man kann sich z. B. ein Programm vorstellen, das nach der Annahme einer Eingabe die Ausführung an fünf aufeinanderfolgende Unterprogramme übergibt, die jeweils verschiedene Aufgaben abarbeiten, und das zum Schluß die verarbeiteten Daten permanent speichert oder das bisherige Gesamtergebnis des Programms verändert. Nun stellt sich im vierten oder fünften Unterprogramm heraus, daß mit den eingegebenen Daten etwas nicht stimmt; der Fehler führt zwar nicht zur Unterbrechung des Programms, aber zu einem unsinnigen Endresultat der verarbeiteten Daten. Was ist dagegen zu tun?
Sicher könnte man in das vierte Unterprogramm einen Fehlertest einbauen und die Ausgabe der passenden Fehlermeldung veranlassen. Außerdem gibt es aber noch das Problem, wie man die Kette der fünf Unterprogramme verläßt. Es reicht nicht, einfach das vierte Unterprogramm zu beenden, weil danach automatisch das fünfte ausgeführt werden würde, und wie wir gesehen haben, hätte das verhängnisvolle Folgen. Richtig geht man vor, indem man das Auffinden eines Fehlers vermerkt und mit Hilfe dieser Aufzeichnung die weitere Bearbeitung der betreffenden Daten verhindert. D. h. man baut beim Verlassen des vierten Unterprogramms eine besondere Kontrolle in das Programm ein, die die Ausführung des fünften Unterpro-

gramms aufhält, wenn dieser spezielle Fehler entdeckt wird. Das wäre eine Lösung; aber da in einem etwas komplexeren Programm an allen möglichen Stellen Fehler auftreten können, ergäbe sich schließlich eine ganze Reihe von Tests, die jeweils auf einen bestimmten Fehler spezialisiert wären.
Das Problem wird elegant gelöst, indem die Aufzeichnung aller während des Programms gefundenen Fehler an eine einzige, regelmäßig zu überprüfende Variable übergeben und einem Unterprogramm die Erledigung aller erforderlichen Fehlermeldungen übertragen wird. Dieses Verfahren erleichtert es einerseits, neue Fehlertests einzufügen und neue Fehler zu definieren, andererseits erledigt es das Problem, wie bei auftretenden Schwierigkeiten die Ausführung bestimmter Programmteile verhindert werden kann.
Die Methode wird hier an einem Programm veranschaulicht, das für drei bekannte Fehler anfällig ist:

1. Im Verlauf einer Berechnung entsteht eine Zahl, die für eine sinnvolle Verarbeitung durch das Programm zu groß ist.
2. Es wird ein Datenname vergeben, der schon im Programmspeicher enthalten ist.
3. Daten müssen in relativ komplexem Format eingegeben werden, und es kann erst nach teilweiser Verarbeitung der Daten überprüft werden, ob dabei Fehler unterlaufen sind.

Sie fangen an, indem Sie beim ersten Programmdurchlauf ein Feld ERR$(3) definieren und die Elemente Eins bis Drei des Feldes auf

a) ZAHL ZU GROSS
b) DATENNAME BEREITS VERGEBEN
c) FORMATFEHLER BEI DER EINGABE

setzen. Außerdem wird eine Variable ERR bestimmt und auf Null gesetzt.
Im Programm selbst werden die Tests für bestimmte Fehler mit den zugehörigen Fehlermeldungen durch solche ersetzt, die lediglich den Wert von ERR ändern, wenn ein Fehler auftritt. Z. B. wird

```
1520 IFNN>65535 THEN PRINT "ZAHL
ZU GROSS
```

ersetzt durch

```
IFNN>65535 THEN ERR=1
```

In jedes entsprechende Unterprogramm wird nun eine neue Zeile eingesetzt, die die Ausführung des Unterprogramms verhindert, wenn irgendein Fehler entdeckt worden ist. Die Eingangszeile eines Unterprogramms könnte so aussehen:

```
IFERR<>0THEN RETURN
```

Das heißt, wenn ERR einen anderen Wert als Null hat, wird das Unterprogramm nicht ausgeführt. Bei Unterprogrammen, die weitere Unterprogramme aufrufen, können die Zieladressen geändert werden, damit nach der Rückkehr aus einem Unterprogramm, in dem ein Fehler gemeldet wurde, die Daten vom Programm nicht weiterverarbeitet werden:

```
1770 GOSUB 2500:IF ERR<>0THENRETURN
```

Am Schluß springt das Programm in das Hauptmodul zurück, das bestimmt, in welcher Reihenfolge die Unterprogramme aufgerufen werden. Das Modul wird wie folgt um eine Zeile erweitert:

```
1120 IF ERR THEN GOSUB 3000
```

Diese Zeile ruft ein einfaches Unterprogramm auf, das so aufgebaut ist:

```
3000 REM********************
3001 REM FEHLER MITTEILUNG
3002 REM********************
3010 PRINTERR$(ERR)
3020 ERR=0
3030 RETURN
```

Wenn Sie möglichen neuen Fehlern während der Programmentwicklung vorbeugen wollen, stellt dieses System insofern eine schöne Lösung dar, als Sie nur eine neue Fehlermeldung in ERR$ aufzunehmen und einen Test einzufügen brauchen, der ERR auf den passenden Wert setzt; die eingebauten Zeilen, die ERR auf Null testen, gewährleisten den Schutz des Programms gegen den von Ihnen ermittelten möglichen Fehler. Sie können in dieser Weise nach Belieben neue Fehler mit geringem Aufwand identifizieren.

SCHLUSS

Mit den in diesem Kapitel umrissenen Methoden können Sie Programme erstellen, die fast jede schlechte Behandlung durch Benutzer überleben. Wie weit Sie die Schutzmaßnahmen ausbauen, hängt teils davon ab, wie kompliziert das Programm ist, und teils davon, wie wichtig die Daten sind, die es bearbeitet. Selbst ein einfaches Programm kann mit Daten laufen, deren Eingabe vielleicht viel Zeit kostet, so daß ein Absturz nach einer halben Stunde Arbeit äußerst unangenehm wäre. Der wichtigste Aspekt ist vielleicht das Erfolgserlebnis, ein Programm geschrieben zu haben, das offenbar die Situation beherrscht – und nicht eines, das man aus Furcht vor Komplikationen nur vorsichtig behandelt.

Seien Sie trotzdem gewarnt! Geben Sie nie damit an, wie robust Ihr Lieblingsprogramm sei. Derjenige, dem Sie das erzählen, wird unweigerlich die einzige Taste drücken, mit der Sie nicht gerechnet haben – und das könnte sich vernichtend auf Ihren guten Ruf auswirken.

KAPITEL 6

SPEICHERN UND LADEN

Eines Tages wird es eine Computergeneration mit genügend großen Speichern geben, um alle Daten und Programme aufzunehmen, die wir benutzen möchten. Und nicht nur das: Daten und Programme werden sofort nach dem Einschalten des Computers als permanenter Speicherinhalt jederzeit verfügbar sein. Solche Geräte bedeuten einen genauso großen Schritt nach vorn wie die Heimcomputer in den letzten Jahren. Bis dahin müssen die Mikrocomputer-Besitzer jedoch lernen, mit Geräten auszukommen, die nur einen Teil der gesamten, ständig gebrauchten Datenmenge aufnehmen können.

Da ein Gerät wie der C 64 nicht mehr als ein Programm und die dazugehörigen Daten zur gleichen Zeit fassen kann, bietet Commodore eine Erweiterung für den Speicher des C 64 an. Der C 1530-Kassettenrekorder und das 1541-Diskettenlaufwerk sind im Grunde nichts weiter als zusätzliche Speicher. Verglichen mit der Geschwindigkeit, mit der Daten aus den RAM-Chips des C 64 geholt werden, arbeitet der Kassettenrekorder und sogar das Diskettenlaufwerk unbestreitbar im Schneckentempo. Effizientes Programmieren, vor allem was Programme für ernsthafte Anwendungen betrifft, kann sich jedoch die zusätzliche Kapazität des C 1530 und der 1541 zunutze machen, um sowohl die Beschränkung des Speichers im C 64 zu überwinden als auch den Umstand, daß die heutige Generation von Mikrocomputern nicht in der Lage ist, Daten zu behalten, während das Gerät ausgeschaltet ist.

Obwohl die meisten Mikrocomputer-Besitzer willens sind, viel Geld für Zubehör auszugeben, das die Speicherkapazität ihrer Rechner vergrößert (oft nur unwesentlich), nutzen nur wenige mit ihren selbstgeschriebenen Programmen die Kapazität ihrer Floppy voll aus, nicht einmal die des bescheideren Rekorders. In diesem Kapitel wollen wir uns die nötigen Techniken zur besseren Ausnutzung des C 1530 und der 1541 aneignen. Das Thema wird nicht erschöpfend behandelt, vor allem in bezug auf den Gebrauch des Diskettenlaufwerks; über den Einsatz der Floppy könnte man leicht ein ganzes Buch schreiben *('Floppy-Programmierung mit dem Commodore 64'*, Band 16 der Sachbuchreihe). Trotzdem werden die hier erläuterten Techniken Sie befähigen, Daten sicherer und in größerer Menge abzuspeichern, Daten mit den auf Band oder Diskette gespeicherten auszutauschen und den verfügbaren Speicherplatz optimal auszunutzen.

SPEICHERN VON PROGRAMMEN

Der wichtigste und augenfälligste Zweck der externen Speicherung ist die sichere Aufbewahrung Ihrer Programme zum weiteren Gebrauch. Es überrascht oft, wie sorglos die meisten dabei vorgehen und versäumen, während der Programmentwicklung die Aktualisierungen regelmäßig zu speichern und sich von der korrekten Aufzeichnung des Programms zu überzeugen, dabei wichtige Programme nur einmal speichern und Bänder und Disketten überall herumliegen lassen. Ein paar vernünftige Regeln sollte man beim Speichern von Programmen auf jeden Fall beachten:

1. Speichern (SAVE) Sie neue Programme während des Schreibens regelmäßig. Wie jeder andere Mikrocomputer kann auch der C 64 Programme verlieren, falls Stromschwankungen auftreten, jemand an den Stecker stößt oder Sie selbst den C 64 beim Programmieren aus dem Gleichgewicht bringen. Wieviel Arbeit dabei verlorengeht, hängt davon ab, wann Sie Ihr Programm zuletzt gespeichert haben. Wenn ein Programm schnell eingetippt wird, ist es nicht ratsam, mehr als 15 Minuten lang Zeilen einzugeben, ohne das Programm erneut zu speichern. Im Verlauf des Austestens, bei dem weniger Änderungen anfallen, kann man die Zeitspanne auf eine halbe Stunde ausdehnen. Das richtet sich im Grunde danach, wieviel Sie zu verlieren bereit sind. Wenn Sie Ihre Programme nicht regelmäßig speichern, werden Sie sicherlich früher oder später ein wichtiges verlieren, dessen Eingabe viel Zeit gekostet hat.

2. Um den Speichervorgang zu vereinfachen und mich selbst zu motivieren, stelle ich an den Programmanfang vier Zeilen, die es mir ersparen, bei jedem Abspeichern den Programmnamen auszuschreiben:

```
1 GOTO4
2 SAVE"PROGRAMM NAME":INPUT"ZURUECKSPULE
N - TASTE FUER 'VERIFY'";Q$
3 VERIFY"PROGRAMM NAME":STOP
4 REM
```

Innerhalb eines Programms hat diese Routine den Vorteil, ein Abspeichern unter falschem Namen infolge eines Tippfehlers auszuschließen, weil zum Speichern nur 'GOTO 2' eingegeben werden muß; außerdem können Sie damit alle Programme einheitlich mit 'GOTO 1' starten, falls Sie nicht RUN benutzen und schon gespeicherte Variablen löschen möchten. VERIFY kann nach Belieben verwendet werden und wird von vielen überhaupt nicht gebraucht. Bevor Sie jedoch einen Entschluß fassen, beachten Sie den nächsten Unterpunkt. Sollten Sie sich entscheiden, auf

VERIFY zu verzichten, brauchen Sie Zeile 3 natürlich nicht, wohl aber das STOP, da ansonsten das Programm bei jedem SAVE mit der Ausführung neu beginnen würde.
Wenn Sie ein Diskettenlaufwerk benutzen, können Sie eine ähnliche Routine wie oben einfügen:

```
1 GOTO3
2 SAVE"@0:PROGRAMM NAME",8:VERIFY"PROGRA
MM NAME",8:STOP
3 REM
```

Floppy-Besitzer seien hier gewarnt: Das Laufwerk 1541 verstümmelt manchmal infolge eines Schwachpunkts im Disketten-Betriebssystem bei fast vollen Disketten die eigene Dateiverwaltung, wenn dem Dateinamen '@0' vorangestellt wird (damit wird die Speicherung des Programms besorgt, selbst wenn schon eine Datei mit demselben Namen existiert). Das zuletzt gespeicherte Programm wird nicht beschädigt, aber es könnte schwierig werden, ein anderes Programm von der Diskette zu laden. Sollte das zu Problemen führen, kann man entweder die vorhandene Programmdatei löschen und dann die neue Version abermals speichern oder eine Zahl an das Ende des Programmnamens in Zeile 2 anhängen, die bei jedem Speichern des Programms geändert wird, oder man läßt den Namen stehen und bestätigt die Gültigkeit der Diskette nach dem Speichern.
Für Floppy-Besitzer gibt es keine Entschuldigung, wenn sie versäumen, die Aufzeichnung eines Programms mit VERIFY zu überprüfen. Wenn Sie das unterlassen, haben Sie verdient, was Ihnen bestimmt passiert.

3. Der C 1530-Kassettenrekorder des C 64 ist eines der sichersten Geräte zum Laden und Speichern, die auf dem Markt zu haben sind. Daher benutzen viele VERIFY zu Anfang jedes Mal, wenn sie ein Programm abspeichern, und später nie mehr, weil sie nie Probleme hatten. Sind Sie von Natur aus ein vorsichtiger Mensch, werden Sie das, was Sie gespeichert haben, sicher jedesmal überprüfen wollen. Wenn Sie eher lässig vorgehen, wenden Sie VERIFY bei Ihren Programmen sicher niemals an. Der goldene Mittelweg liegt zwischen diesen beiden Extremen.
Beim Eingeben langer Programme kann VERIFY allerhand Zeit in Anspruch nehmen. Noch mehr Zeit kann aber der Verlust des Programms kosten, was doch gelegentlich vorkommt. Ab und zu geht ein Programm wegen gewisser Beschränkungen im SAVE-Programm des C 64 verloren, häufiger jedoch macht die Qualität des benutzten Bandes Schwierigkeiten. Diese Probleme treten zwar selten auf, dann aber können sie sehr ärgerlich sein.
Der beste Weg ist ein Kompromiß. Wenn Sie der Qualität Ihres Bandes trauen, dann entwickeln Sie ein Programm z. B. auf einer C 60-Kassette, nehmen Sie jede neue

Version nach der ursprünglichen auf, und drehen Sie die Kassette dann um oder spulen Sie sie zurück. Auf diese Weise müßten Sie erst das Programm im Computer verlieren *und* eine fehlerhafte Aufzeichnung gemacht haben, bevor Sie mehr als den letzten Stand des Programms verlieren. Soll jedoch die endgültige Version abgespeichert werden, prüfen Sie das Band immer mit VERIFY.

4. Während lange Bänder beim Entwerfen von Programmen ausgesprochen gute Dienste leisten, ist für die Langzeitspeicherung auf Band der Gebrauch spezieller Computerkassetten zweifellos vorzuziehen, wobei jedem Programm eine eigene Kassette vorbehalten sein sollte. Das ist zwar etwas teurer, aber so können Sie auf jedes Programm direkt zugreifen, ohne auf langen Bändern nach dem richtigen Anfang suchen zu müssen. Außerdem besteht so weniger Gefahr, ein vorhandenes Programm versehentlich zu überspielen.

5. Halten Sie Aufnahme- und Wiedergabekopf des Rekorders sauber. Reinigungszubehör ist billig und einfach anzuwenden, dagegen ist es äußerst frustrierend, ein Programm zu verlieren, weil die Bänder auf den Köpfen eine Schmutzschicht hinterlassen haben.

6. Behalten Sie von Ihren Programmen mehr als eine Kopie, und bewahren Sie die zweite an einem ganz anderen Platz auf als die Bänder oder Disketten, mit denen Sie gewöhnlich arbeiten. Es besteht immer die Gefahr, daß Ihre Arbeitskopie auf irgendeine Weise beschädigt wird – sei es durch übermäßige Hitze, oder gar weil ein Kind mit einem Magneten spielt. Wenn Sie nicht jedes Programm auf einem eigenen Band duplizieren wollen, können Sie Ihre Sicherungskopien auf längere Bänder abspeichern und später auf kürzere Kassetten überspielen, falls sie gebraucht werden sollten.
Für Floppy-Besitzer sind Sicherungskopien besonders wichtig. Nicht selten stößt man beim Laden oder Speichern auf Schwierigkeiten, oder eine Diskette wird zufällig beschädigt. Andererseits sind Kopien viel einfacher herzustellen, denn das Abspeichern von Programmen auf zwei Disketten macht kaum zusätzliche Mühe. Auch wenn Sie ein Diskettenlaufwerk haben, sollten Sie die relative Sicherheit und Verläßlichkeit von Bandgeräten nicht außer acht lassen, wenn von wichtigem Material Sicherungskopien erstellt werden sollen. Ein ernsthafter Defekt am Diskettenlaufwerk kann viel Ärger für den Fall bedeuten, daß die Kopien des benötigten Programms nur auf Diskette erhalten geblieben sind.

SPEICHERN UND LADEN VON DATEN

Die meisten wirklich benutzten Programme brauchen als Arbeitsunterlage eine gewisse Menge von Daten. Bei nicht konstanten Daten, die also direkt in das Programm geschrieben werden können, haben Sie zwei Möglichkeiten: Entweder geben Sie die Daten jedesmal neu ein, oder Sie gehen das Problem an, wie Daten auf Band oder Diskette gespeichert und später wieder in das Programm geladen werden.
Das erste Problem, mit dem Sie beim Speichern auf Band oder Diskette konfrontiert werden, ist die genaue Bestimmung dessen, was Sie speichern möchten. Es ist unsinnig, den Inhalt eines Feldes abzuspeichern, wenn Sie nicht daran denken, auch die Variable zu speichern, die vielleicht zu dem Feld gehörte und die Anzahl der darin enthaltenen Elemente aufzeichnete. Machen Sie also als erstes eine vollständige Liste der für Ihr Programm wichtigen Variablen. Das sind keineswegs alle vorkommenden Variablen, da vielen erst beim Programmdurchlauf ein Wert zugeordnet wird. Speichern müssen Sie nur diejenigen, die nötig sind, um das Programm beim nächsten Gebrauch zum Laufen zu bringen. In diesem Zusammenhang ist darauf zu achten, daß von Feldern nur die maßgebenden Bereiche gespeichert werden. Sie haben möglicherweise ein Stringfeld von 500 Zeilen dimensioniert, in dem Sie allmählich einen Datenvorrat anlegen. Wenn Sie erst 170 Zeilen des Feldes belegt haben, sollten Sie unbedingt eine Variable haben, die die Anzahl der belegten Zeilen aufzeichnet, und nur diese Zeilen abspeichern. Das ist deswegen sinnvoll, weil die Übertragung von Daten Byte für Byte auf Band oder Diskette länger dauert als das Laden und Speichern eines Programms, und man sollte nicht noch mehr Zeit als nötig mit unnötigen Elementen verschwenden.

SPEICHERN AUF BAND

Nachdem Sie Variablen und Teilfelder bestimmt haben, müssen Sie sie in einem Modul unterbringen, das sie sicher auf Band speichert. Zu diesem Zweck öffnet man zuerst eine Datei mit einer Zeile wie:

```
1710 OPEN1,1,1,"FILE NAME"
```

Bei einem Mikrocomputer wie dem C 64 ist die 'Datei' kein statischer Ort, in den Daten abgelegt werden, sondern eine Kommunikationsleitung. Die drei Zahlen in der Musterzeile bedeuten, daß die geöffnete Datei jedes Mal, wenn etwas zu speichern ist, durch Datei 1 oder kurz '#1' angesprochen wird, daß die Zeile eine Verbindung zu 'Gerät Nummer 1' herstellt, und daß die Kommunikation von C 64 zum Peripheriegerät erfolgt statt umgekehrt. Während die Datei arbeitet, können

andere geöffnet werden. Es können bis zu 10 Dateien gleichzeitig offen sein, obwohl sehr selten mehr als eine oder zwei zusammen gebraucht werden.
Beim Öffnen einer Datei zum Abspeichern auf Band kann zwischen zwei möglichen Sekundäradressen gewählt werden, nämlich Eins und Zwei. Sekundäradresse Eins benennt eine Ausgabedatei, die alle vorgefundenen Daten annimmt. Sekundäradresse Zwei bezeichnet ebenfalls eine Ausgabedatei, in der zusätzlich ein besonderes Dateiende-Zeichen den Abschluß der Datei markiert. Beim zweiten Dateityp kann man Bytes aus der Datei lesen, ohne ihre genaue Anzahl zu kennen, wobei eine Prüfzeile das Dateiende ermittelt und eine Eingabe über das Dateiende hinaus verhindert, was einen Fehler erzeugen würde:

```
1710 NN=0:OPEN1,1,0,"FILENAME"
1720 INPUT#1,T
1730 IFST=64THENCLOSE1:RETURN
1740 A(NN)=T
1750 NN=NN+1:GOTO1720
```

Diese Routine nimmt solange Daten vom Band an, bis sie auf das Dateiende-Zeichen trifft. Dieses Zeichen ändert den Wert der Systemvariablen ST (STATUS), worauf die Programmausführung fortgesetzt wird. Vorsichtshalber weise ich darauf hin, daß die Verwendung der Sekundäradresse Zwei manchmal seltsame Auswirkungen auf das hat, was hinter der fraglichen Datei auf dasselbe Band abgespeichert wird. Meistens ist das EOF (End of File)-Zeichen nicht nötig, denn ein ordentliches Programm müßte sich merken können, wie viele Datenelemente es speichert und folglich auch, wie viele noch gespeichert werden müssen. Wenn das bekannt ist, kann man an den Anfang jedes Datenblocks Variablen für die Datenmenge stellen und das Programm so anlegen, daß es diese bestimmte Anzahl von Elementen wieder lädt, ohne dabei das EOF-Zeichen zu verwenden. In Klartext: Gebrauchen Sie Sekundäradresse Zwei nur dann, wenn es unbedingt nötig ist.
Das übliche Format zum Öffnen einer Datei sieht etwa so aus:

```
OPEN [FILE NUMMER],[GERAET NUMMER],[ART
DES FILES (SEKUNDAERADRESSE)],"FILE NAME"
```

Jeder Versuch der Datenübertragung in eine Datei, die nicht geöffnet wurde, führt zu einer Fehlermeldung.
Der betreffende Dateiname muß nicht in der Programmzeile selbst ausgeschrieben werden, er kann auch als Antwort auf eine INPUT-Anweisung vom Benutzer angegeben werden:

```
1710 INPUT"NAME DES FILES ";FF$
1720 OPEN 1,1,1,FF$
```

Damit kann ein einzelnes Programm Dateien mit ganz verschiedenen Namen erzeugen. Diese Technik kann bei der Dateieingabe dazu dienen, verschiedene Eingabedateien zu bezeichnen, damit das Programm je nach Zweck zwischen den verschiedenen Dateien wechseln kann.

AUSGABE AUF EINE DATEI

Nachdem die Datei geöffnet ist, müssen nun Daten in sie geschrieben werden. Das geschieht mit der PRINT#-Anweisung. Alles auf PRINT# und die passende Dateinummer Folgende wird genauso in die Datei übertragen, wie es mit PRINT auf den Bildschirm gedruckt würde (Drucken auf den Bildschirm ist bei Öffnen einer Datei auf Gerät Nr. 3 sogar möglich), aber es gibt doch einige Unterschiede:

1. Anders als viele andere Mikrocomputer markiert der C 64 nicht automatisch die Trennung zwischen Variablen, die mit einem Befehl wie

```
PRINT#1, A$,B$,C$
```

durch dieselbe Zeile mit PRINT abgelegt werden. Beim Einlesen der Daten vom Band würde diese Zeile als ein String ermittelt, der aus A$, B$ und C$ zusammen besteht. Will man Ausdrücke getrennt speichern, gibt es zwei Möglichkeiten: Entweder druckt man jedes Datenelement mit einer eigenen PRINT#-Anweisung oder man fügt zwischen den Ausdrücken Trennzeichen ein.
Bei Feldern verwendet man zur Trennung von Ausdrücken eine Schleife, die ein Element nach dem anderen speichert, z. B.:

```
1710 FOR I=1TOITEMS
1720 PRINT#1,A$(I)
1730 NEXT I
```

Sollen einzelne Ausdrücke in eine Datei geschrieben werden, muß ihnen ein 'Return'-Zeichen (CHR$(13)) folgen. In der Regel definiert man zu diesem Zweck bei der Initialisierung des Programms einen String, z. B. R$, als CHR$(13) und trennt dann mit R$ die einzelnen Ausdrücke, anstatt jedes Mal CHR$(13) zu tippen:

```
1740 PRINT#1,TT$ R$ CD$ R$ IT R$ NN R$ Q
Q$
```

Beachten Sie, daß die Ausdrücke nicht durch Interpunktion gegliedert sind. Sie können zwar Kommas oder Semikolons einfügen, diese werden aber vom C 64 ignoriert.

2. Der C 64 kann keine Zeichen speichern oder ausgeben, die nicht zu den normal druckbaren Zeichen gehören. Wenn Sie Strings speichern wollen, die außer den normalen Bestandteilen von Strings wie Cursor- oder Farbsteuerung noch andere Steuerzeichen enthalten, müssen Sie wenigstens einen Teil des Strings Zeichen für Zeichen in Zahlen übersetzen und diese Zahlen einzeln abspeichern.

```
1710 PRINT#1,LEN(A$)
1720 FOR I=1TOLEN(A$)
1730 PRINT#1, ASC(MID$(A$,I,1))
```

Denken Sie daran, die Stringlänge abzuspeichern, damit das Programm beim Laden weiß, wo die übersetzten Stringzeichen zu Ende sind.

3. Sie können keine leeren Strings abspeichern. Das kann bei Stringfeldern, die oft leere Elemente enthalten, Probleme ergeben. Wenn nämlich das erste Element eines Feldes leer ist, wird es nicht gespeichert. Auf dem Band rückt das zweite Element an die Stelle des ersten und zerstört damit beim Laden die Reihenfolge des Feldes. Dies kann man am einfachsten lösen, indem man jedes Element des Feldes mit einem führenden Zeichen 'polstert', bevor man es speichert:

```
1710 FOR I=1TOITEMS
1720 T$="*"A$(I):PRINT#1,T$
1730 NEXT
```

Natürlich dürfen Sie nicht vergessen, diese Zeichen beim Laden wieder zu entfernen, aber es dauert nicht länger, alle Elemente aufzufüllen, als bei jedem zu untersuchen, ob es ein Leerstring ist, und dann nur die Leerstrings zu polstern.

4. Die Reihenfolge, in der Sie die Daten speichern, kann für den späteren Zugriff darauf ausschlaggebend sein. Erinnern Sie sich an ein früheres Beispiel, das von einem Stringfeld mit 500 Elementen und einer Variablen ausging, die sich die Anzahl der "ITEMS" genannten, derzeit gebrauchten Elemente merkte. Den Inhalt dieses Feldes speichern Sie mit einer Schleife ab:

```
1710 FOR I=0TOITEMS
```

Folglich braucht das Programm beim Laden einen Wert für ITEMS, bevor es die Daten wieder vom Band holen kann. Daraus ergibt sich die einfache Regel: Werden irgendwelche Variablen aus dem Programm für die Steuerung der Datenspeicherung gebraucht, sollten sie *vor* diesen Daten abgelegt werden.

5. Nachdem alle Daten gespeichert sind, muß die Datei geschlossen werden. Geschieht das nicht, führt jeder weiterer Versuch, eine Datei mit derselben Nummer zu öffnen, zur Unterbrechung des Programms mit einer Fehlermeldung. Das Schließen einer Datei geschieht folgendermaßen:

```
CLOSE[FILE NUMMER]
```

Man sollte jedoch vor Abschluß einer Datei sichergehen, ob sich nichts mehr im 'Speicherpuffer' befindet, wo Daten während ihrer Ausgabe auf Band abgelegt werden, da ansonsten das bzw. die letzte(n) Datenelement(e) unter Umständen nicht richtig abgespeichert werden. Dazu müßte zwar CLOSE genügen, aber erfahrungsgemäß ist es sicherer, wenn man den zusätzlichen Befehl PRINT#⟨file number⟩ ohne Angabe bestimmter Daten verwendet. Daraus ergibt sich für das Schließen einer Datei folgendes Format:

```
PRINT#[FILE NUMMER]:CLOSE[FILE NUMMER]
```

LADEN VOM BAND

Der Vorgang spiegelt in vieler Hinsicht das Speichern auf Band wider. Die wichtigsten Unterschiede sind:

1. Die Art der geöffneten Datei ist anders, d. h.:

```
OPEN1,1,0,"FILE NAME"
```

Die Null zeigt an, daß der C 64 diese Datei benutzt, um Daten aus einem Peripheriegerät zu empfangen.

2. Die wichtigsten Befehle zum Einlesen von Daten vom Band sind INPUT# und GET#. Sie werden später ausführlicher behandelt.

3. Wenn die Datenelemente beim Speichern ordnungsgemäß getrennt worden sind, ist eine Ermittlung der Trennzeichen nicht erforderlich:

```
1840 INPUT#1,TT$,CD$,IT,NN,QQ$
```

würde ausreichen, um die Daten, die im obigen Beispiel zum Gebrauch von R$ als Trennzeichen gespeichert wurden, vom Band zu holen.

4. Führende Zeichen, die in Strings aufgenommen wurden, müssen wieder entfernt werden:

```
1810 FOR I=0TOITEMS
1820 INPUT#1,T$:A$(I)=MID$(T$,2)
1830 NEXT I
```

5. Strings, die in Form von numerischen Entsprechungen ihrer Zeichen abgelegt wurden, müssen in Strings zurückverwandelt werden:

```
1810 A$="":INPUT#1,LS
1820 FOR I=1TOLS
1830 INPUT#1,T:A$=A$+CHR$(T)
1840 NEXT I
```

6. Da INPUT# im Unterschied zum einfachen INPUT nur Eingaben von maximal 80 Zeichen verarbeiten kann, muß man diese Beschränkung manchmal umgehen. In solchen Fällen stellt man mit GET#, das ein Zeichen nach dem anderen vom Band holt, den ursprünglichen String zeichenweise wieder her.

```
1810 A$=""
1820 GET#1,T$:IFT$<>CHR$(13)THENA$=A$+T$
:GOTO1820
```

Hier liest GET# solange Zeichen ein und übergibt sie an A$, bis das Stringende-Zeichen ermittelt wird.

7. Im allgemeinen stellt man das Programm-Modul zum Laden von Daten am besten direkt hinter das Speichermodul. Der sicherste Weg, eine Laderoutine zu schreiben, die genau die Reihenfolge des vorherigen Speicherns widerspiegelt, besteht nämlich darin, die Zeilennummern der Speicherroutine zu editieren und PRINT#-Befehle gegen INPUT# auszutauschen. Trotzdem kann das Programm die beiden Routinen auf mehrere verschiedene Arten aufrufen.
Die Speicherroutine sollte eine normale, vom Menü aufrufbare Programmfunktion sein und vielleicht vor Programmabschluß (der besser durch Menü-Option als durch

einfaches Drücken von 'STOP' erfolgen sollte) den Benutzer darauf hinweisen, daß er gut daran tut, die Daten zu speichern, bevor sie verlorengehen. Beim Eingeben großer Datenmengen in das Programm kann der Benutzer sie so durch regelmäßiges Speichern gegen mögliche Probleme mit dem C 64 oder dem Programm absichern.
Ganz anders kann die Laderoutine mit einer Selbstladefunktion am Programmanfang aufgerufen werden, die in mancher Hinsicht der in Kapitel 1 beschriebenen Selbstinitialisierung ähnelt. Bei der Selbstinitialisierung hat der Benutzer die Wahl, entweder das Programm mit RUN zu starten – wobei die Variablen auf Null gesetzt werden – oder mit GOTO, das die Ausführung des Initialisierungsmoduls verhindert, wenn das Programm schon Daten enthält.
Als weitere Möglichkeit läßt die Selbstladefunktion den Benutzer bestimmen, ob neue Daten über die Tastatur eingegeben oder zuerst auf Band vorhandene Daten geladen werden sollen, wenn das Programm keine Daten enthält. Entscheidet er sich, Daten vom Band zu laden, wird die Initialisierungsroutine teilweise ausgeführt, und zwar derjenige Teil, der die Felder für die Aufnahme von Daten einrichtet: die Variablen werden jedoch nicht auf ihren Anfangswert gesetzt, da sie ohnehin vom Band geladen werden. Es folgt ein Beispiel für ein solches Modul:

```
1000 IFA$(0)THEN1500
1010 DIM A$(100),B$(25),A%(100),B%(25):L
IMIT=100:R$=CHR(13)
1020 INPUT"SOLL VON KASSETTE GELADEN WER
DEN (J/N) ";Q$
1030 IF LEFT$(Q$,1)="J"THEN GOSUB LADER:
GOTO 1500
1040 ITEMS=0:NN=0:CT=12:BASE=2
```

Hier enthält Zeile 1000 die Selbstinitialisierung. Die zweite Zeile dimensioniert vier Felder und definiert eine Variable, deren Wert während des Programmablaufs unverändert bleibt – in diesem Fall die Höchstzahl der zugelassenen Elemente. Danach kann der Benutzer angeben, ob Daten vom Band geladen werden sollen. Wenn ja, denken Sie daran, daß die Daten alle Variablen enthalten müssen, die an den nicht ausgeführten Teil des Initialisierungsmoduls übergeben werden. Deshalb ist es hier noch wichtiger als sonst, als Gedächtnisstütze beim Schreiben der Speicher- und Lademodule alle Variablen aufzuschreiben, selbst die mit einem Anfangswert von Null.

SPEICHER- UND LADEROUTINEN: EIN ARBEITSBEISPIEL

Weiter unten finden Sie eine Speicher-/Laderoutine, die aus einem meiner eigenen Programme stammt. Sie brauchen die Funktionen der einzelnen Variablen nicht zu verstehen, da an dem Beispiel nur die Verfahrensweise bei größeren Datenvolumen gezeigt werden soll:

```
20000 REM*************
20001 REM DATEN FILES
20002 REM*************
20025 R$=CHR$(13)
20030 INPUT"KASSETTE RICHTIG EINGELEGT -
 [REVERS EIN]RETURN";Q$
20040 IFNN$=""THEN20140
20080 OPEN1,1,1,"FILE NAME":PRINT#1,NN$,
R$,QQ$,R$,CU,R$,ITEMS
20090 IFCU=0THEN20110
20100 FORI=0TOCU-1:PRINT#1,T$(I,0),R$,T$
(I,0),R$,T(I):NEXT
20110 IFIT=0THEN20130
20120 FORI=0TOIT-1:PRINT#1,A$(I,0),R$,A$
(I,1),R$,C(I):NEXT
20130 PRINT#1:CLOSE1:RETURN
20140 OPEN1,1,0,"FILE NAME":INPUT#1,NN$,
QQ$,CU,ITEMS
20150 IFCU=0THEN20170
20160 FORII=0TOCU-1:INPUT#1,T$(I,0),T$(I
,1),T(I):NEXT
20170 IFIT=0THEN20190
20180 FORI=0TOIT-1:INPUT#1,A$(I,0),A$(I,
1),C(I):NEXT
20190 CLOSE1:RETURN
```

Der Benutzer hat hier die Möglichkeit, das Band in Position zu bringen. Danach ermittelt das Modul automatisch, ob Daten gespeichert oder geladen werden sollen, je nachdem, ob der String NN$ leer ist. Das entspricht insofern der Selbstinitialisierung, als für diesen Test unbedingt eine Variable gewählt werden muß, die nie leer ist, solange das Programm Daten enthält. Diese 'Selbstladezeile' wäre durch ein einfaches zweizeiliges Menü ersetzbar, indem der Benutzer sich für Laden oder Speichern entscheiden kann. Danach werden zwei Felder gespeichert oder gela-

den, wobei die Anzahl der zu verarbeitenden Daten vom Inhalt der beiden Variablen CU und IT abhängt. Zum Schluß wird die Datei abgemeldet. Wie Sie bemerken, ist das Format der beiden Modulhälften genau gleich. Tatsächlich diente die Speicherroutine als Vorlage für die Laderoutine, indem die Zeilennummern geändert und die Zeilen editiert wurden: Damit ist gewährleistet, daß die Elemente in derselben Reihenfolge abgerufen werden, in der sie gespeichert wurden.

BESONDERHEITEN BEI DISKETTEN

Die Arbeit mit Dateien ist für Floppy-Besitzer wesentlich angenehmer, weil Daten einfach schneller geladen und gespeichert werden. Das liegt nicht nur an der hohen Arbeitsgeschwindigkeit der Floppy, sondern auch daran, daß das Laufwerk eine Ausgabedatei auf der Diskette automatisch positioniert oder die gewünschte Eingabedatei auffindet. Die Methoden zum Speichern von Daten, zur Trennung von Elementen usw. sind dieselben, jedoch sind zusätzliche Vorkehrungen für die Angabe des gewünschten Dateityps und zum Überschreiben einer Datei nötig, wenn unter demselben Dateinamen bereits andere Daten abgelegt wurden.

1. Das Diskettenlaufwerk verlangt neben der Bestimmung des Dateityps durch die Zahlen hinter dem OPEN-Befehl und dem Dateinamen noch eine besondere Angabe. Der von uns gebrauchte Dateityp heißt 'sequentielle' Datei, d. h. sie speichert Dateielemente in der Reihenfolge der Eingabe und kann sie in derselben Reihenfolge wieder ausgeben. Dieser Dateityp wird mit einer OPEN-Anweisung wie folgt eingerichtet:

```
OPEN1,8,2,"FILE NAME,S,W"
```

8 ist die Gerätenummer des Diskettenlaufwerks. S bezeichnet eine sequentielle Datei, und W bedeutet, daß es sich um eine 'Schreibdatei' ('write file') handelt, d. h. daß Daten darauf ausgegeben werden. Normalerweise werden Diskettendateien mit Sekundäradresse Zwei angesprochen, die beim Rekorder erwähnten Nachteile gelten hier nicht.
Das Einlesen von der Diskette erfolgt durch eine OPEN-Anweisung des Formats

```
OPEN1,8,0,"FILE NAME,S,R"
```

wobei R anzeigt, daß es sich um eine 'Lesedatei' ('read file') handelt, bzw. daß Daten daraus in den C 64 zurückgeholt werden. Wird der Zusatz W/R ausgelassen, betrachtet das Diskettenlaufwerk die fragliche Datei als Lesedatei.

2. Die zweite Schwierigkeit im Umgang mit einem Diskettenlaufwerk ergibt sich im Grunde aus seiner intelligenten Eigenschaft, die auf der benutzten Diskette enthaltenen Daten genau zu kennen. Wenn man wie in dem o. g. Speicher/Lademodul mit einer Kassette arbeitet, kann der C 1530-Rekorder nicht ermitteln, ob und welche neuen Daten überschrieben werden. Der Benutzer bestimmt durch Bandpositionierung genau, wo Daten gespeichert werden sollen.
Die Floppy enthält eine Sicherung gegen versehentliches Löschen einer vorhandenen Datei bei dem Versuch, etwas unter demselben Namen zu speichern. Das ist zwar eine wertvolle Schutzmaßnahme, aber manche Dateien werden öfter von der Diskette gelesen, erweitert oder berichtigt und dann wieder gespeichert, wie es beim fortgesetzten Gebrauch eines Programms häufig geschieht. Zu diesem Zweck kann man einen besonderen Ausdruck vor den Dateinamen stellen, damit eine evtl. existierende Datei desselben Namen und Typs überschrieben wird. Das Format dafür ist:

```
OPEN1,8,2,"@:FILE NAME,S,W"
```

Dies kann natürlich mit der Möglichkeit kombiniert werden, den Dateinamen zuzuordnen, oder auch mit der Wahlfreiheit des Benutzers, ob eine Datei überschrieben werden soll oder nicht:

```
1710 INPUT"FILE NAME ";FF$:FF$=FF$+"S,W"

1720 Q$="N":INPUT"BESTEHENDES FILE UEBER
SCHREIBEN (J/N) ";Q$
1730 IFQ$="J"THENFF$="@0:"+FF$
1740 OPEN2,8,2,FF$
```

SCHLUSS

Auch wenn das Speichern und Laden von Daten im Grunde eine einfache Angelegenheit ist, kann in einem komplexen Programm der richtige Aufbau des Dateimoduls eine knifflige Arbeit sein. Das sollte aber nicht von der Tatsache ablenken, daß Programme zur Verarbeitung großer Datenvolumen einen externen Speicher brauchen. Noch sollten wir uns vom Computerzeitalter verleiten lassen, die Geschwindigkeit, mit der ein Diskettenlaufwerk oder auch ein Rekorder große Datenmengen in den Arbeitsspeicher des C 64 laden können, geringzuschätzen. Floppys und Rekorder sind verläßliche und sogar relativ schnelle Geräte zum Speichern nützlicher Daten. Ihre Verwendung ist trotz aller Einschränkungen sicherlich der Arbeit mit Programmen vorzuziehen, die entweder nur über eine eingebaute Datengruppe oder nur über soviele Daten verfügen, wie in einem Arbeitsgang an der Tastatur eingegeben werden können. Wenn eine Datengruppe die Verarbeitung lohnt, ist sie höchstwahrscheinlich auch die Mühe wert, sie für den späteren Gebrauch sicher abzuspeichern.

KAPITEL 7

LOGISCHE BEDINGUNGEN

Es gibt nur wenige Techniken, die zur Verkürzung von Programmzeilen oder überhaupt zur eleganten Form eines Programms mehr beitragen als der richtige Gebrauch der internen Logik des C 64 in Verbindung mit gewöhnlichem Programmieren in BASIC. Dieses Kapitel führt uns in die gelegentlich schwer verständliche Welt der logischen Bedingung IF und der logischen Operatoren AND und OR. Viele Mikrocomputer-Besitzer wenden sie regelmäßig in ihren Programmen an, ohne sich jemals über die Gesamtheit ihrer Möglichkeiten im klaren zu sein.

DAS BESCHEIDENE IF

Jeder BASIC-Programmierer benutzt IF – eines der wichtigsten Programmierwerkzeuge. Trotzdem wird IF nicht immer so direkt eingesetzt wie es aussieht, d. h. viele Ergebnisse lassen sich mit IF-Anweisungen direkter erzielen, als manchem Mikrocomputer-Besitzer bekannt zu sein scheint.
Das Wesentliche bei IF ist, daß damit eine Operation ausgeführt oder unterlassen werden kann, je nachdem, ob eine vom Programmierer gesetzte Bedingung erfüllt ist. Demnach wird in einer Zeile wie

```
IF A>10 THEN K=K+1
```

der Ausdruck im zweiten Teil der Zeile nur dann ausgeführt, wenn A größer als 10 ist. Das gilt für alles, was in einer Zeile auf eine IF-Anweisung folgt. Innerhalb einer einzelnen auf 80 Zeichen begrenzten Programmzeile kann in dieser Weise eine Reihe von Befehlen ausgeführt oder ignoriert werden, z. B.:

```
100 IF A>10 THEN K=K+1:X=X-10:Y=Y*100:GO
TO 200
```

Wenn auf eine einzelne Bedingung mehr Befehle folgen als in einer Zeile Platz haben, ist es am einfachsten, mit Hilfe der entgegengesetzten Bedingung einen Programmabschnitt zu überspringen:

```
100 IFA<10THENGOTO120
110 PRINT"DER WERT VON A IST GROESSER AL
S 10":K=K+1:X=X-10:Y=Y*100:GOTO200
```

SICHERUNG GEGEN UNGÜLTIGE WERTE

Den Effekt von IF, den darauffolgenden Teil der Zeile zu isolieren, kann man u. a. zum Schutz gegen mögliche falsche Variablenwerte nutzen, die das Programm unterbrechen würden. Im nächsten Beispiel erzeugt Zeile 20 einen BAD SUBSCRIPT ERROR:

```
10 DIM A(20)
15 SS=25
20 IF A(SS)>10THEN50
```

Das Problem wird umgangen, wenn man das ursprüngliche IF in Zeile 20 selbst einem IF unterordnet:

```
20 IF SS<=20 THEN IF A(SS)>10 THEN 50
```

Die mögliche Zahl der so verschachtelten IFs ist nur durch die Zeilenlänge begrenzt.

DURCH IF HERVORGERUFENE FEHLER

Ein weit verbreiteter Programmierfehler ist auf die oft vergessene Tatsache zurückzuführen, daß IF eine Zeile teilweise unberücksichtigt lassen kann. So könnte z. B. ein Programmierer in der Absicht, ein Feld abzusuchen und 10 von allen Zahlen über 100 zu subtrahieren, folgende Zeile eingeben:

```
100 FORI=0TO99:IFA(I)>100THENA(I)=A(I)-1
0:NEXT I
```

Die gewünschte Korrektur würde jedoch nur dann ausgeführt, wenn alle vorgefundenen Elemente größer als 100 wären. Beim ersten Wert unter 100 würde NEXT I ignoriert und die Schleife beendet.

IF . . .THEN . . .ELSE

Die IF-Anweisung auf dem C 64 ist in einer Beziehung unzulänglich, denn ihr fehlt eine Einrichtung, die sich als Bestandteil des BASIC vieler anderer Heimcomputer bewährt hat: die IF . . .THEN . . .ELSE-Anweisung. Bei diesem Format darf der

Benutzer zwei Operationen nach dem IF angeben: Eine wird ausgeführt, wenn die Bedingung wahr ist, die andere, wenn sie falsch ist. Eine Zeile wie

```
100 IF A>10 THEN K=K+1 ELSE K=K-1
```

würde 1 zu K addieren, wenn A größer als 10 wäre, und 1 subtrahieren, wenn nicht.– *Nur: Auf dem C 64 funktioniert das nicht!* Dieser Befehl ist in den vielen Fällen hilfreich, in denen der Programmierer entweder/oder-Entscheidungen im Programm treffen muß; auf dem C 64 muß das Problem anders gelöst werden. Der einfachste Weg sieht vor, eine Anweisung vor das IF zu stellen und die zweite dahinter. Also kann

```
100 IF A>10 THEN K=K+1 ELSE K=0
```

nachgeahmt werden durch:

```
100 TT=K:K=0:IFA>10 THEN K=TT+1
```

Wenn die auszuführenden Operationen länger als eine Zeile sind, kann man sie auf zwei Zeilen mit den jeweils entgegengesetzten Bedingungen verteilen:

```
100 IFA>10THENPRINT"MEHR ALS ZEHN EINHEI
TEN":K=K+1:GOTO200
110 IFA<10THENPRINT"NOCH KEINE ZEHN EINH
EITEN EINGEGEBEN":K=0:GOTO250
```

IF mit >, < und =

Wie Sie sehen, kommen in den angeführten Beispielen Kombinationen mit >= und <= vor. Wenn man in einer Zeile Bedingungen verwendet, muß man zwischen Einfachheit und leichter Lesbarkeit des Programms wählen. Soll z. B. eine Operation ausgeführt werden, wenn A gleich oder kleiner als 10 ist, könnte man die Bedingung so formulieren:

```
IF A<10THEN.....
```

oder sie könnte in den meisten Fällen auch lauten:

```
IF A<=10THEN.....
```

Der zweite Ausdruck ist kürzer, aber da der relevante Wert 10 ist, kann die 11 beim Lesen des Programms irritieren. Außerdem ist zu berücksichtigen, daß bei Verwendung von nicht-ganzzahligen Werten, d. h. von Zahlen, die keine ganzen Zahlen sein müssen, <11 *nicht* dasselbe ist wie <10, denn <11 kann jeder Wert zwischen 10 und 11 sein.

IF MIT DEN OPERATOREN AND UND OR

Der Wirkungsbereich der IF-Anweisung wird bedeutend durch die in BASIC enthaltenen Operatoren AND, OR und NOT erweitert. Mit den ersten beiden kann der Programmierer verschiedene IF-Anweisungen in einer Zeile kombinieren. In den Zeilen

```
100 IFA>10THEN120
110 GOTO 130
120 IFB<=100THENK=K+1
```

müssen beide gesetzten Bedingungen erfüllt sein, sonst wird der zweite Teil von Zeile 120 nicht ausgeführt. Mit AND können die Zeilen zu einer einzigen zusammengefaßt werden:

```
100 IFA>10 AND B<=100 THEN K=K+1
```

Ein anderes Problem ergibt sich bei den Zeilen

```
100 IFA>10 THEN 130
100 IFA<=100THEN 130
120 GOTO 140
130 K=K+1
```

wo die Erfüllung *einer* von beiden Bedingungen ausreicht, damit Zeile 130 ausgeführt wird. Hier ist der Gebrauch von OR angezeigt:

```
100 IFA>10 ORB<=100 THEN K=K+1
```

KOMBINIEREN VON AND UND OR

AND und OR können in derselben Zeile zur Herstellung komplexer Bedingungsgefüge eingesetzt werden, wenn man die Reihenfolge berücksichtigt, in der sie vom C 64 ausgewertet werden. Geben Sie folgende Zeilen ein:

```
100 A=10
110 B=100
120 C=50
130 PRINT"␣"
140 IFA=10 AND B=100 OR C=10 THEN PRINT"OK"
```

Das Programm gibt am Ende des Durchlaufs ein OK aus, woraus man schließen kann, daß die ersten beiden durch AND verbundenen Bedingungen ausreichen, die angegebene Operation auszuführen, obwohl C nicht gleich 10 ist. Jetzt ändern Sie Zeile 140 in:

```
140 IFA=10 AND B=150 OR C=10 THEN PRINT"
OK"
```

worauf Zeile 140 kein OK ausgibt. Wenn Sie Zeile 140 noch einmal ändern:

```
140 IFA=10 AND B=150 OR C=50 THEN PRINT"
OK"
```

ist das Ergebnis wieder OK – aber was heißt das? Es gibt zwei mögliche Erklärungen:

1. Die wahre Bedingung für C ist an die Stelle der falschen für B getreten, so daß die Zeile als IF A=10 (B=150 OR C=50) gelesen wird.

2. Die wahre Bedingung für C tritt an die Stelle beider durch AND verbundenen Bedingungen, so daß die Zeile als IF (A=10 AND B=150) OR C=50 gelesen wird. Das läßt sich nur entscheiden, indem man an Zeile 140 eine weitere Änderung vornimmt:

```
140 IFA=20 AND B=150 OR C=50 THEN PRINT"
OK"
```

Wir sehen nun, daß die Bedingung hinter OR die beiden durch AND verbundenen Bedingungen wie eine einzige behandelt hat. Daraus folgt, daß alle durch AND

verknüpften Bedingungen so erfüllt werden müssen, als wären sie eine einzige Bedingung. Andererseits ist jede Bedingung vor OR eigenständig.
Die strenge Hierarchie von AND und OR sorgt manchmal für Überraschungen, wenn scheinbar folgerichtige Zeilen nicht zum erwarteten Ergebnis führen. Sollten Sie sich also über die Verknüpfung einer Bedingungsfolge nicht ganz im klaren sein, so verwenden Sie besser Klammern. Lesen Sie diese Zeile:

```
140 IF(A=10 OR B=100) AND (C=50 OR D=20)
THEN PRINT "OK"
```

Die Klammern trennen die zweite und dritte Bedingung von AND und erzwingen die Auswertung der ersten beiden und letzten beiden Bedingungen, bevor AND berücksichtigt wird. Wenn Ihnen das noch immer unklar ist, fügen sie der o. g. kurzen Routine versuchsweise eine neue Zeile an:

```
125 D=20
```

Passen Sie Zeile 140 der letztgenannten Version an. Wie zu erwarten, gibt das Programm zum Schluß OK aus, da jede Bedingung der Zeile erfüllt ist. Jetzt ändern Sie Zeile 140 nochmals:

```
140 IF(A=10 OR B=200) AND (C=30 OR D=20)
THEN PRINT "OK"
```

und das Resultat ist immer noch OK. Da A gleich 10 ist, ist das Bedingungspaar in Klammern erfüllt. Da D gleich 20 ist, ist auch das zweite Bedingungspaar erfüllt. AND ist jetzt von zwei wahren Bedingungen umgeben, folglich kann die Zeile verarbeitet werden.

ENTSCHACHTELN KOMPLEXER BEDINGUNGEN

Wenn Sie große Probleme mit komplexen Bedingungsblöcken haben – und so ergeht es vielen –, dann zerlegen Sie die betreffende Zeile in eine Reihe von 'wahren' und 'falschen' Bedingungen, und vereinfachen Sie diese den folgenden Regeln entsprechend:

```
1 ) WAHR UND WAHR = WAHR
2 ) WAHR UND UNWAHR = UNWAHR
3 ) UNWAHR UND UNWAHR = UNWAHR
4 ) WAHR ODER WAHR = WAHR
```

```
5 ) WAHR ODER UNWAHR = WAHR
6 ) UNWAHR ODER UNWAHR = UNWAHR
```

Nehmen Sie z. B. an, im o. g. Programm sei A=10, B=100, C=50 und D=20; die Bedingungen der Zeile

```
IFA=10 AND B=200 OR C=50 THEN.....
```

lauten übersetzt:

```
WAHR UND UNWAHR ODER WAHR
```

was nach den oben angeführten Regeln zunächst zu

```
UNWAHR ODER WAHR
```

vereinfacht werden kann, und schließlich zu

```
WAHR
```

```
IF( A=10ORB=200 )AND( C=30ORD=10 )THEN...
```

wird zu

```
( WAHR ODER UNWAHR )UND( UNWAHR ODER WAHR )
```

```
WAHR UND UNWAHR
```

```
UNWAHR
```

BESTIMMUNG VON GRENZWERTEN

Bedingungen dienen oft dazu, einen Bereich für eine Variable zu bestimmen, innerhalb dessen eine Operation ausgeführt wird. Das Format dieser Anweisungen ist:

1. Bei einer Operation, die ausgeführt werden soll, wenn eine Variable im Bereich 11–20 liegt, und nicht ausgeführt werden soll, wenn sie außerhalb dieses Bereiches liegt:

```
100 IF A=>11 AND A<=20 THEN...
```

2. Bei einer Operation, die *nicht* ausgeführt werden soll, wenn eine Variable im Bereich 11–20 liegt, und die ausgeführt werden soll, wenn sie außerhalb des Bereiches liegt:

```
100 IF A<11 AND A>20 THEN...
```

Manchmal ist die Bestimmung zweier möglicher Bereiche notwendig. Das geschieht oft beim Überprüfen der Eingabe eines Zeichens in ein Programm, wenn z. B. die zulässigen Zeichen entweder die Zahlen 0 bis 9 oder die Buchstaben A bis Z sind. Dazu brauchen wir zwei Bedingungspaare:

```
100 IF (IN$>="0"AND IN$<="9") OR (IN$>=
"A"AND IN$<="Z") THEN GOTO200
```

IF MIT NOT

NOT ist vielen gar nicht als Programmelement geläufig, und es leistet auch tatsächlich nur wenig, das nicht mit anderen Mitteln erreichbar wäre. NOT hat die Funktion, die Wirkung einer Bedingung umzukehren, und kann die Lesbarkeit mancher Zeilen erleichtern. Bei der Nachahmung von IF . . .THEN . . .ELSE sind z. B. in zwei aufeinanderfolgenden Zeilen entgegengesetzte Bedingungen enthalten. Das hätte man sicher übersichtlicher ausdrücken können:

```
100 IF A>10 THEN PRINT"MEHR ALS 10 EINGA
BEN":K=K+1:GOTO 200
110 IF NOT A>=10 THEN PRINT"10 EINGABEN
NOCH NICHT ERREICHT":K=0:GOTO 250
```

ANWENDUNG LOGISCHER BEDINGUNGEN

Im Kapitel über Stringfunktionen kam eine einfache Zeile vor, die einen mit FRE ermittelten, unechten Wert in den richtigen verwandelte. Wenn der von FRE angegebene Wert negativ war, mußte 65536 dazu addiert werden. Das hätte mit einer Zeile wie

```
K=FRE(0):IFK<0THENK=K+65536
```

erreicht werden können. Tatsächlich sah die Zeile aber so aus:

```
K=FRE(0):K=K-65536*(K<0)
```

Wenn wir davon ausgehen, daß die beiden Anweisungen genau dieselbe Bedeutung haben, können wir daraus etwas über die zweite Version erfahren. Offenbar wird darin 'IF K<0') irgendwie durch (K<0) ersetzt. Da wir je nach Situation manchmal 65536 addieren möchten und manchmal nichts, muß (K<0) in der zweiten Fassung außerdem seltsamerweise einmal für den Wert Null und einmal für den Wert −1 stehen, sonst ergäbe die Zahl −65536 keinen Sinn. Wie ist das zu erklären?

WERT EINER BEDINGUNG

Bei der Auswertung einer Bedingung in einem Programm setzt der C 64 wie alle Mikrocomputer voraus, daß etwas entweder wahr oder falsch ist. Der C 64 merkt sich seine Entscheidung, indem er den Ausdruck hinter IF überprüft und ihm einen von zwei Werten gibt: −1, wenn die Bedingung wahr ist, und 0, wenn sie falsch ist. Nehmen Sie z. B. ein Programm mit den zwei Variablen X und Y, wobei X gleich 7 und Y gleich 5 ist. Betrachten Sie jetzt folgende Zeilen, und zwar als Aussagen von der Art 'John ist größer als Bill':

```
A) X=Y
B) X>Y
B) Y<=X
D) Y>=X
```

Ganz offensichtlich sind die Aussagen a) und d) falsch, während b) und c) richtig sind.
Im Zusammenhang mit einer IF-Anweisung würde a) und d) der Wert Null, b) und c) den Wert −1 zugewiesen. Wo solche Bedingungen wie oben vorkommen, wird die in der IF-Anweisung angegebene Operation ausgeführt, wenn die Bedingung den Wert −1 bekommen hat. Grundsätzlich kann man alles hinter die IF-Anweisung stellen, was einen Wert erhalten kann. Die Zeile

```
100 IF TT THEN 120
```

würde den Sprung der Zeile 120 nicht nur veranlassen, wenn der Wert von TT −1 wäre, sondern überhaupt ungleich Null. Auch mit Strings kann so verfahren werden:

```
100 IF A$ THEN 120
```

würde ausgeführt, wenn A$ etwas anderes als ein leerer String wäre. Diese Methode kann angewendet werden, wann immer es von Bedeutung ist, daß ein Wert Null ermittelt wird. Eine mögliche Anwendung, die häufig bei der Suche nach Programmierfehlern zum Einsatz kommt, ist im Kapitel über Methoden der Fehlersuche beschrieben.

ANWENDUNG VON BEDINGUNGEN ALS WERTE

Neben der Möglichkeit, eine IF-Anweisung mit Hilfe einer einzelnen Variablen zur Ausführung zu bringen, ergibt sich aus der Art, wie Bedingungen mit Null oder −1 bewertet werden, eine weitere interessante Perspektive: Bedingungen können nämlich überall im Programm für die Zuordnung von Werten benutzt werden, nicht nur in einer IF-Anweisung. Auf dieser Tatsache beruht die o. g. Zeile, mit der FRE umgewandelt wird. Das Programm ordnet jeder Bedingung, auf die es während der Ausführung trifft, einen Wert zu, und mit diesem Wert kann das Programm genauso gesteuert werden wie mit jeder anderen Variablen. Bei der o. g. FRE-Funktion wird 65536 zum Ergebnis von FRE(0) nur dann addiert, wenn die Bedingung (K<0) wahr ist. Solche Bedingungen kann man elegant miteinander kombinieren, um eine Anhäufung von IF-Anweisungen zu vermeiden. Die Zeilen:

```
100 K=100
110 IF TT>120 THEN K=K+50
120 IF XX<50  THEN K=K-25
130 IF Y$="MINUS"THEN K=K+100
140 IF FF=0 THEN K=K*ZZ
```

und viele andere dieser Art könnten etwa so ersetzt werden:

```
100 K=(100-50*(TT>120) + 25*(XX<50) - 100*
(Y$="MINUS")) * (ZZ+(ZZ-1)*(FF<>0))
```

PLUS ODER MINUS?

Das Verwirrende dabei ist die scheinbare Vertauschung von Plus- und Minuszeichen. Die Notwendigkeit dafür leuchtet aber ein, wenn man bedenkt, daß das Ergebnis einer wahren Bedingung nicht 1, sondern −1 ist. Falls Sie 100 zu K

addieren wollen, wenn etwas wahr ist, müssen Sie das Hundertfache des Werts der erfüllten Bedingung (also −1) subtrahieren, damit das Ergebnis K−(−100) bzw. K+100 ist.

MULTIPLIKATION UND DIVISION

Beim Multiplizieren (oder Dividieren) als Ergebnis einer Bedingung ist außerdem folgendes zu beachten: Die direkte Methode besteht darin, von der Zahl, mit der Sie multiplizieren wollen, zunächst ihren Wert −1 mal den Wert der Bedingung, die der gewählten entgegengesetzt ist, zu subtrahieren. Falls Sie also mit 1000 multiplizieren wollen, wenn X gleich Null ist, multiplizieren Sie in Wirklichkeit mit 1000+999*(X<>0). Wenn X *nicht* gleich Null *ist,* multiplizieren Sie mit 1000+(−999) bzw. 1.

VERMEIDEN EINER ISOLIERUNG DURCH IF

Wie Sie sich erinnern, können Bedingungen dazu dienen, die durch IF bewirkte Isolierung alles Nachstehenden bei nicht erfüllter Bedingung zu umgehen. Weiter oben wurde auf die Gefahr hingewiesen, IF in derselben Zeile zu benutzen wie NEXT, das eine Schleife beendet, wenn die Ausführung der ganzen Schleife beabsichtigt ist. Dieses Problem ist jedoch mit einer Zeile wie

```
100 FORI=0TO99:A(I)=A(I)+10*(A(I)>100):N
EXT
```

zu lösen. Vergessen Sie nicht, daß es in einigen Fällen nicht angebracht ist, IF-Anweisungen durch Ausdrücke zu ersetzen, die auf dem Wert von Bedingungen beruhen. Wo man zwei IF-Anweisungen einsetzt, um den Zugriff auf eine Variable mit ungültigem Wert zu verhindern, haben Bedingungen diese Schutzfunktion nicht. In einem früheren Abschnitt hatten wir die Zeile:

```
100 IFSS<=20 THEN IFA(SS)>10 THEN K=K+1
```

die das Programm vor dem Absturz bewahrt, wenn der Wert von SS größer als die Höchstzahl der Elemente im Feld ist. Eine Zeile wie

```
100 K=K+(SS<=20)*(A(SS)>10)
```

würde dem Programm den Zugriff auf den ungültigen Wert von SS erlauben und damit zum Absturz führen. Außer in diesen Fällen, wo das Programm geschützt

werden soll, kann eine Reihe von IFs immer durch AND ersetzt werden. Entweder so:

```
100 IF A=10 AND B=20 AND C=30 AND D=40 T
HEN K=K+10
```

oder auch:

```
100 K=K+10*(A=10)*(B=20)*(C=30)*(D=40)
```

Hier ist zweierlei bemerkenswert: Erstens ist die Platzersparnis im Programm viel geringer, und zweitens wurde offenbar die Regel in bezug auf das Vorzeichen der zu addierenden Zahl nicht eingehalten. Das liegt daran, daß die Anzahl der miteinander zu multiplizierenden Bedingungen eine Rolle spielt. Hier gilt die Regel: Ist die Anzahl der Bedingungen *gerade,* so wird das Ergebnis addiert, ist sie *ungerade,* so wird das Ergebnis subtrahiert.
Auf Bedingungen basierende Zeilen sind nicht immer leicht zu lesen, aber sie zu schreiben ist viel einfacher, als es den Anschein hat. Die Schwierigkeiten werden in vieler Hinsicht dadurch aufgewogen, daß mit ihrer Hilfe sehr kompakte Programme geschrieben werden können.

AND UND OR MIT ZAHLEN

AND und OR besitzen eine weitere, sehr nützliche Eigenschaft im Zusammenhang mit Zahlen. Mit ihrer Hilfe können arithmetische Ergebnisse erzielt werden, die auf andere Art nur äußerst schwer herauszubekommen wären. So angewendet, üben AND und OR ihre besondere Wirkung auf die einzelnen Bits binärer Zahlen zu einer Länge von 15 Bits (0–32767) aus. Werden zwei Zahlen mit AND verknüpft, so ergibt sich eine Zahl, die aus den Binärstellen besteht, die in beiden ursprünglichen Zahlen auf 1 gesetzt waren. Werden zwei Zahlen mit OR verknüpft, ergibt sich eine Zahl, in der jede Binärstelle auf 1 gesetzt wird, die in *einer* der Ursprungszahlen auf 1 gesetzt war, also:

```
231 (BINAER 11100111) AND 126 (BINAER 01
111110)=102 (BINAER 01100110)
```

und

```
231 OR 126 (BINAER 11111111)
```

Es gibt eine ganze Reihe von Möglichkeiten, wie man sich diese scheinbar abwegige Eigenschaft im Programm zunutze machen kann.

POKE MIT AND UND OR

Auf viele Funktionen des C 64, wie Klangeffekte und Sprites, kann man nur mit POKE-Anweisungen zugreifen. Häufig wird POKE benutzt, um ein einzelnes BIT innerhalb eines Bytes im Speicher zu verändern, ohne Einfluß auf die übrigen Bits. Hier leisten AND und OR gute Dienste. Um ein Bit im Byte der Adresse ADD zu setzen, braucht man in der Regel eine Zeile wie POKE ADD, PEEK(ADD) OR 2^BIT. Soll z. B. sichergestellt werden, daß Bit Null (das in einer Zahl die 1 repräsentiert) in der Adresse 53287 auf 1 geschaltet oder 'gesetzt' wird, müßte die Zeile lauten:

```
210 POKE 53287, PEEK (53287) OR2↑0
```

Statt 2^0 könnte man einfacher 1 schreiben, aber das gewählte Format garantiert, daß Sie das zu ändernde Bit nicht verwechseln.
Um ein Bit auf Null zu setzen oder rückzusetzen, ist die Funktion AND erforderlich. Dazu lautet die Regel: Man verknüpft den Inhalt der betreffenden Adresse und 255−2^BIT wie folgt durch AND:

```
210 POKE53287,PEEK(53287)AND(255-2↑0)
```

SPEICHERN MIT AND/OR

Mit Hilfe von AND und OR kann eine einzelne Variable sehr effektiv zum Speichern von bis zu 15 'ja/nein'-Datenelementen benutzt werden. Wird die Variable zunächst auf Null gesetzt, so kann jedes der 15 Bits mit einer ähnlichen Methode wie beim Poken gesetzt werden. Mit dieser Zeile würde Bit 7 gesetzt:

```
210 A=A OR 2↑7
```

Dasselbe Bit würde rückgesetzt mit

```
210 A=A AND (255-2↑7)
```

Die Variable kann dann mit einer einfachen Schleife gelesen werden:

```
100 FOR I=0 TO14:IFA AND 2↑I THEN K=I:
GOSUB 1000
110NEXT I
```

Ein einzelnes Bit wird überprüft mit einer Anweisung wie z. B.

```
100 IF A AND 2↑X THEN....
```

Ich habe ein eigenes Programm zur Aufzeichnung von Kontobewegungen, in dem jeweils eine einzelne Zwei-Byte-Variable pro Ausgabe den Monat der Abwicklung speichert.

GERADE ODER UNGERADE?

Oft ist es in Programmen notwendig zu prüfen, ob eine Variable gerade oder ungerade ist. Auch das geschieht durch einen einfachen Bit-Test, in diesem Fall für Bit Null der Variablen, wie z. B.:

```
100 IF A AND 2↑0 THEN....
```

SCHLUSS

Das letzte Kapitel ist voller Bits und Schnipsel, die scheinbar in viele verschiedene Richtungen führen. Trotzdem hoffe ich, daß Sie in Ihren Programmen viele Gelegenheiten entdecken, umständliche und ungeschickte Codes durch kürzere, klarere und elegantere Befehle zu ersetzen. Auf jeden Fall werden Ihnen die in Zeitschriften und Büchern abgedruckten Programme nun weniger Kopfzerbrechen machen; denn wenn die Verfasser ihr Handwerk verstehen, wird das, was hier dargestellt wurde, immer eine wichtige Rolle spielen.

KAPITEL 8

SORTIEREN

In diesem Kapitel soll erklärt werden, was mich dazu brachte, Bücher zu schreiben. Vor einigen Jahren veröffentlichte eine englische Computerzeitschrift ein Programm, mit dem der Benutzer Namen und Adressen von Bekannten speichern konnte. Nach Eingabe aller Namen sollten sie auf Knopfdruck vom Programm zum späteren Gebrauch alphabetisch sortiert werden. Es wurde vermutlich wegen seiner Klarheit und relativen Kürze veröffentlicht, aber ich hatte vom ersten Moment an Zweifel an der benutzten Methode.

Anstatt die erlaubten 100 Namen einzutippen, ging ich daran, eine einfache Routine zur Erzeugung von 100 unsinnigen Namen zu schreiben. Damit ließ ich das Programm arbeiten. *Eine halbe Stunde später* war der Sortiervorgang beendet. Hätte ich nur 90 Namen eingegeben und später noch einen oder zwei ergänzt, dann hätte es beinahe noch eine weitere halbe Stunde gedauert, bis diese beiden zusätzlichen Namen an den richtigen Platz sortiert gewesen wären: ein an sich nützliches Programm, das sich durch die Wahl einer ungeeigneten Methode für alle praktischen Anwendungen als unbrauchbar erwies.

Auf der Grundlage von Notizen aus einem Computerlehrgang am College schrieb ich einen kurzen Artikel, in dem ich zeigte, wie das betreffende Programm um etwa 40 Prozent hätte beschleunigt werden können – das war der erste ausführliche Artikel von mir, der veröffentlicht wurde. Seit damals bin ich immer wieder erstaunt, wie viele Anwendungsprogramme darunter leiden, daß ihre Verfasser nichts davon wissen, daß es viele verschiedene Sortiermethoden mit sehr unterschiedlichem Zeitaufwand gibt. Zweifellos war das erwähnte Programm aus der Zeitschrift für eines der langsamsten Geräte geschrieben, die es je auf dem Heimcomputermarkt gab. Bei einem so hochentwickelten Gerät wie dem Commodore 64 müßten Sie sich sehr anstrengen, um einen Sortierprozeß dermaßen zu verlangsamen, daß er für nur 100 Elemente eine halbe Stunde braucht. Dennoch gilt die Regel: Eine angemessene Sortiermethode entscheidet darüber, ob ein Programm langsam und mühselig arbeitet oder schnell genug, um es einigermaßen nutzbringend einsetzen zu können.

Wir werden in diesem Kapitel drei Sortiermethoden behandeln und zeigen, wie und warum sie funktionieren und wieso der Zeitaufwand so unterschiedlich sein kann. Bevor wir uns dem Sortieren mit dem Computer zuwenden, werden wir erst einmal genauer untersuchen, wie ein Mensch Dinge sortiert. Dabei werden wir hoffentlich eine klarere Vorstellung von dem bekommen, was wir vorhaben, wenn wir an der Tastatur sitzen.

SORTIEREN: WARUM UND WOZU?

Wir beginnen mit einem kleinen Experiment, zu dem wir 10 Zettel brauchen. Ideal wären Karteikarten, aber wenn Sie keine haben, zerschneiden Sie einfach ein Blatt Papier in 10 Quadrate von 7–8 cm Länge. Beschriften Sie die Zettel mit den Zahlen 0 bis 9, wobei Sie bitte die 6 und die 9 unterstreichen, damit sie nicht verwechselt werden können, falls sie zufällig umgekehrt zu liegen kommen. Machen Sie jetzt etwas Platz auf dem Tisch (möglichst nicht im Durchzug) und legen Sie die Zettel nebeneinander in dieser Reihenfolge aus:

7 3 1 5 6 9 4 8 0 2

(Die Reihenfolge ist an sich ohne Bedeutung; aber wenn Sie eine andere nehmen, ergeben die nachfolgenden Kommentare keinen Sinn.)
Zweck der Übung ist das Sortieren der Zettel in aufsteigender Folge von 0 bis 9, links beginnend. Die einzige Einschränkung besteht darin, daß Ihnen nur 11 Stellen zur Verfügung stehen: 10 sind durch die Zettel besetzt und eine ist frei. Das bedeutet, Sie können einen Zettel erst dann von einem Platz der Reihe auf einen anderen legen, wenn Sie vorher einen aus der Folge herausgenommen und auf dem elften Platz untergebracht haben. Damit erhalten Sie einen Platz in der Reihe, auf den Sie einen Zettel legen können und einen Zettel, der aus der Reihe entfernt und auf den Extraplatz gelegt wurde. Am Schluß sollen alle Zettel in der richtigen Reihenfolge liegen, und der Extraplatz soll leer sein. Probieren Sie es aus und versuchen Sie dabei, Ihr Vorgehen zu analysieren.
Wenn Sie sich bis hierher an den beschriebenen Weg gehalten haben, haben Sie wahrscheinlich folgendes getan:

1. Die Folge überflogen, um entweder den höchsten oder niedrigsten Wert zu finden.

2. Den Zettel auf der höchsten oder niedrigsten Stelle auf den Extraplatz gelegt und den richtigen Zettel auf die jetzt freie Stelle in der Folge gelegt.

3. Danach hatten Sie die Wahl, entweder den Zettel auf dem Extraplatz in die Lücke der Folge zu legen und bei 1. mit dem zweithöchsten bzw. -niedrigsten Wert weiterzumachen, oder die korrekte Position für den Zettel auf dem Extraplatz auszumachen und ihn dort unterzubringen, nachdem Sie den vorhandenen Zettel weggenommen haben. Letzteres verringert die Anzahl der notwendigen Vertauschungen.

Vielleicht sind Sie ganz anders vorgegangen. Aber wenn Sie Verstand und Augen richtig benutzt haben, wird folgendes für Ihre Sortiermethode zutreffen:

1. Da die Zettel am Schluß die ihren Zahlen entsprechenden Positionen in der Reihenfolge einnehmen sollten, konnten Sie jederzeit schnell erkennen, wohin jeder einzelne Zettel gehört.

2. Da die Anzahl der Zettel relativ klein war, konnten Sie die absolut höchste und niedrigste Zahl der Folge leicht feststellen und entsprechend vorgehen.

3. Sie konnten die gesamte Folge fast auf einen Blick übersehen und Ihre Maßnahmen daran orientieren.

Jetzt versetzen Sie sich in die Lage eines Mikrocomputers, der einen Sortiervorgang beginnt. Stellen Sie sich vor, Sie hätten 100 Karteikarten vor sich liegen, auf denen z. B. jeweils ein anderer Personenname steht:

1. Sie könnten sich nach keiner Reihenfolge richten, an der sofort erkennbar wäre, wohin jede einzelne Karte gehört. Es mag vielleicht eine Reihenfolge geben, aber es wäre Ihnen nicht möglich, sie zu bestimmen. Natürlich könnte man ein sehr schnelles Sortierprogramm schreiben, wenn man davon ausgeht, daß bei 100 Elementen die Abstände ihrer Werte absolut regelmäßig sind, so daß ein Blick genügte, um sofort zu entscheiden, welche Position eine Karte schließlich einnehmen wird. Dieses Verfahren hat einen Nachteil: Es funktioniert eben nur bei einer regelmäßigen Liste. Ein normales Sortierprogramm muß mit Daten aller Art fertigwerden und kann nicht – im Gegensatz zum menschlichen Gehirn – sagen: "Ach, dies ist eine regelmäßige Zahlenfolge, bei der ich jedem Element sofort den richtigen Platz geben kann."

2. Da die Daten keine bestimmte Regelmäßigkeit haben, könnten Sie nicht ganz so leicht das höchste oder niedrigste Element der Folge, dann das zweithöchste etc., bestimmen. Um die Karte mit dem höchsten Wert zu finden, müßten Sie jede einzelne Karte untersuchen und könnten erst ganz am Schluß feststellen, welche davon die höchste war.

3. Da Sie immer nur eine Aufgabe zur gleichen Zeit erfüllen und nur einen Tatbestand überprüfen können, kämen Sie nie in die Lage, sich ein Bild von der gesamten Liste zu machen. Sie müßten jeweils hier und da ein Element vergleichen, ohne überblicken zu können, was im Ganzen vor sich geht. Zeigen Sie einem Menschen eine Liste in der Reihenfolge:

0 1 2 3 4 5 6 7 8 9

mit dem Auftrag, sie von 0 bis 9 zu sortieren. Sie werden sofort die Antwort

bekommen: "Das ist schon sortiert." Der Mikrocomputer könnte diese Antwort niemals geben, ohne vorher jedes Element der Liste zu untersuchen.

Unter Berücksichtigung dieser wichtigen Unterschiede zwischen den Fähigkeiten des Menschen und des Computers könnten wir fast eine Klassifizierung von Sortiermethoden vornehmen: am einen Ende die Methoden, bei denen die Beschränkungen des Computers sämtlich akzeptiert werden, am anderen Ende diejenigen, die einige der Abkürzungen zu imitieren versuchen, die der menschliche Verstand nehmen würde. Die einfachste aller bekannten Methoden, deren Arbeitsweise zugleich sehr starr 'computergemäß' ist, heißt Bubble-Sort.

DER BUBBLE-SORT

Der Bubble-Sort gründet sich im wesentlichen auf die Fähigkeit des Mikrocomputers, zwei benachbarte Elemente vergleichen und entscheiden zu können, welches größer ist. Der Name rührt von der Art her, wie im Verlauf des Sortierens die höheren Werte die Liste gleichsam 'aufschäumen' (bubble up), wie die Blasenbildung am Rand eines Glases mit Sprudelwasser.
Zur Veranschaulichung nehmen wir wieder unsere Zettel und legen sie in der bekannten Reihenfolge aus:

7 3 1 5 6 9 4 8 0 2

Halten Sie die genannten Regeln ein, d. h. es steht Ihnen ein Extraplatz zur Verfügung, der am Schluß frei sein muß, und gehen Sie wie folgt vor:

1. Beginnen Sie mit dem ersten Zettel, der 7 links außen, vergleichen Sie ihn mit dem Zettel daneben, in diesem Fall der 3. Da 7 größer als 3 ist, nehmen Sie die 3 und legen Sie sie auf den Extraplatz. Bringen Sie die 7 auf dem Platz unter, wo vorher die 3 lag, und holen Sie nun die 3 aus dem Extraplatz auf die vorher durch die 7 belegte Stelle. Damit haben Sie die 7 in der Folge um eine Stelle nach oben verschoben.

2. Lassen Sie die 7 in der Reihenfolge weiter nach oben rücken und gehen Sie dabei jedesmal wie beschrieben vor, wenn Sie rechts daneben eine kleinere Zahl finden. Zuletzt liegt die 7 an fünfter Stelle, rechts daneben die 9.

3. Da Sie auf eine Zahl gestoßen sind, die größer als 7 ist, nämlich die 9, wenden Sie sich nun dieser neuen Zahl zu und behandeln sie genauso wie vorher die 7, d. h. Sie vertauschen sie mit der Zahl rechts daneben, falls diese kleiner ist. Da 9 die

größte Zahl der Folge ist, können Sie solange damit fortfahren, bis sie am Ende der Folge liegt.

4. Fangen Sie am Anfang der Reihe mit der 3 wieder an. Ist die Zahl rechts daneben kleiner, tauschen Sie beide aus; wenn nicht, lassen Sie die 3 liegen und machen Sie mit der größeren Zahl weiter. Zum Schluß liegt die 8 an neunter Stelle links von der 9. Da Sie beim Überfliegen der Reihe die 9 bereits als größte Zahl erkannt haben, ist der Vergleich mit der letzten Zahl der Reihe eigentlich überflüssig.

5. Gehen Sie zum Anfang zurück und nehmen Sie die 1. Sie muß sofort mit der 3 vertauscht werden. Fahren Sie weiter oben in der Reihe fort und vergessen Sie nicht, daß Sie die letzten Elemente nicht zu beachten brauchen, weil sie vorher schon korrekt eingegliedert wurden.

6. Wiederholen Sie den Vorgang, bis Sie die ganze Reihe einmal durchgehen können, ohne etwas auszutauschen. Bedenken Sie, daß Sie als Computer ohne diesen Durchgang nicht wissen, ob die Reihenfolge stimmt; denn Sie können die Reihe als Ganzes nicht sehen.

Wenn Sie genau nach Vorschrift verfahren sind, sollten die Zettel nach jedem Durchgang jeweils so ausliegen:

```
7 3 1 5 6 9 4 8 0 2  :  START POSITION
3 1 5 6 7 4 8 0 2 9
1 3 5 6 4 7 0 2 8 9
1 3 5 4 6 0 2 7 8 9
1 3 4 5 0 2 6 7 8 9
1 3 4 0 2 5 6 7 8 9
1 3 0 2 4 5 6 7 8 9
1 0 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

Nachdem Sie den Vorgang einmal durchgespielt und zuletzt eine richtig sortierte Reihenfolge herausbekommen haben, sollten Sie es mit beliebigen anderen Folgen nochmals ausprobieren, um sich mit der Methode vertraut zu machen. Es gibt jedoch noch mehr, was für die Beurteilung dieser und jeder anderen Sortiermethode ganz wesentlich ist.
Bauen Sie die Ausgangsfolge wieder auf und fangen Sie an, nach der Bubble-Sort-Methode zu sortieren. Notieren Sie sich diesmal in getrennten Spalten jeden Vergleich zwischen zwei Elementen (ohne Berücksichtigung der jeweiligen Größe der verglichenen Zahlen) und jeden Austausch von zwei Zetteln, den Sie vorneh-

men. Nach meiner Rechnung erhalte ich folgendes Resultat für die 8 Durchgänge, bei denen etwas verändert wird, und für den letzten Durchgang, der nur die richtige Reihenfolge bestätigt:

1)	9	VERGLEICHE	8	VERTAUSCHUNGEN
2)	8	"	4	"
3)	7	"	3	"
4)	6	"	3	"
5)	5	"	2	"
6)	4	"	2	"
7)	3	"	2	"
8)	2	"	1	"
9)	1	"	0	"
SUMME:	45	VERGLEICHE	25	VERTAUSCHUNGEN

Damit haben Sie etwas erfahren, das für das Verständnis der Wirkungsweise aller Sortiermethoden von entscheidender Bedeutung ist. Sortieren heißt vergleichen und vertauschen; es gibt genauso viele Unterschiede zwischen Methoden wie Methoden selbst, wobei Vertauschungen immer viel mehr Zeit erfordern als Vergleiche. Es ist möglich, Sortierverfahren ihrer erwarteten Leistung entsprechend mathematisch einzuschätzen. Wir haben das an dieser Stelle nicht wirklich vor, aber dabe ergäben sich für die Bubble-Sort-Methode im besten Fall (d. h. bei einer schon geordneten Reihenfolge) zum Sortieren einer Reihe mit N Elementen 0 erforderliche Vertauschungen und N−1 Vergleiche. Im schlechtesten Fall (bei einer Reihe von Elementen in umgekehrter Reihenfolge) brauchte die Bubble-Sort-Methode 0.5*N*(N−1) Vertauschungen und genauso viele Vergleiche.
In der Praxis bedeutet die kurze Formel: Der Bubble-Sort benötigt im ungünstigsten Fall

für 100 Elemente: 4950 Vertauschungen, 4950 Vergleiche,
für 1000 Elemente: 499 500 Vertauschungen, 499 500 Vergleiche.

Daraus können Sie ersehen, daß die Bubble-Sort-Methode in bezug auf die Anzahl der erforderlichen Operationen ungeheuer aufwendig wird, wenn die Anzahl der zu sortierenden Elemente groß wird.
Natürlich entspricht die tatsächliche Anzahl der Vertauschungen beim Sortieren einer bestimmten Liste sehr selten dem besten oder schlechtesten Fall. Bei unserer eigenen kurzen Folge hatten wir die größtmögliche Zahl von Vergleichen [0.5*10*(10−1)=45], und das wird auch bei anderen Listen häufig der Fall sein. Was die Vertauschungen betrifft, hatten wir jedoch nur etwas über die Hälfte der theoretisch größten Zahl. Bei Listen ist das unterschiedlich, aber im allgemeinen tut

sich der Bubble-Sort mit dem schnellen Sortieren um so schwerer, je länger die Liste ist. Bei kurzen Listen machen sich Unterschiede im Zeitaufwand kaum bemerkbar. Hier kommt es eher auf möglichst einfache Programmierbarkeit des Sortierprogramms an.

PROGRAMMIEREN DES BUBBLE-SORT

Nachdem wir uns in Ruhe einen Einblick in die Bedeutung des Sortierens allgemein und die Arbeitsweise des Bubble-Sort im besonderen verschafft haben, wollen wir nun erklären, wie das im Rahmen eines Programms zu formulieren ist (falls Sie jetzt noch Wert darauf legen). Weiter unten ist ein einfaches Programm skizziert, mit dem wir drei verschiedene Sortiermethoden testen werden. Das Programm besteht aus:

1. einem Zufallswortgenerator zur Erstellung einer Liste von 100 unsinnigen Buchstabenkombinationen, der in A$ abgelegt wird;

2. einer Kopierroutine, die die Ausgangsliste in ein zweites Feld kopiert, damit wir später dieselbe Liste für alle anderen Sortiermethoden benutzen und ihre Ausführungen vergleichen können;

3. einer Prüfroutine, die eine Liste während der Bearbeitung durch eines der Sortierverfahren auf ihre richtige Reihenfolge hin untersucht;

4. einer Routine zum Ausdruck der Liste, damit Sie die Reihenfolge selbst überprüfen können. Dieses Unterprogramm werden Sie womöglich gar nicht aufrufen, solange alles problemlos abläuft; denn es verhindert den gleichzeitigen Bildschirmausdruck der Timings für alle drei Sortierprogramme, die eingegeben werden;

5. einer Routine, die die ungeordnete Liste nach dem Bubble-Sort-Verfahren in alphabetischer Reihenfolge sortiert.

In diesem Kapitel beschränken wir uns auf das Sortieren von Strings. Um Zahlen zu sortieren, muß man lediglich die Namen der Felder in numerische Felder abändern. Wie Sie sehen werden, arbeiten die Sortierprogramme dabei schneller, vor allem solche, die eine große Anzahl von Vertauschungen erfordern. Denn ständiges Vertauschen von Strings führt beim C 64 dazu, daß er in ziemlich regelmäßigen Abständen Pausen zum Aufräumen des Speichers macht. Es gibt einige Methoden, die – wenigstens auf dem C 64 – bei Zahlen sehr schnell sind, aber wegen der vielen Vertauschungen für Strings nicht zu empfehlen sind.
In den angeführten Sortierbeispielen werden unsere Listen immer in alphabetische

Reihenfolge gebracht. Wenn Sie eine Liste absteigend ordnen wollen (d. h. von Z bis A), müssen Sie nur die Zeilen ändern, die Bedingungen mit > und < enthalten, um die Bedingungen umzukehren.

```
1000 REM*****************
1001 REM KONTROL ROUTINE
1002 REM*****************
1010 GOSUB5000
1020 GOSUB4000:TI$="000000":GOSUB8000:PR
INTTI$:GOSUB6000:GOSUB7000
1100 PRINT"LISTE SORTIERT"
1999 STOP
4000 REM***************
4001 REM KOPIERROUTINE
4002 REM***************
4010 FORI=0TO99
4020 B$(I)=A$(I)
4030 NEXTI
4040 RETURN
5000 REM*****************
5001 REM ERZEUGERROUTINE
5002 REM*****************
5010 IT=100:DIMA$(IT-1),B$(IT-1)
5020 FORI=0TOIT-1
5030 T$=""
5040 FORJ=1TO4+INT(9*RND(0))
5050 T$=T$+CHR$(65+INT(26*RND(0)))
5060 NEXTJ
5070 A$(I)=T$
5080 NEXTI
5090 RETURN
6000 REM*****************
6001 REM AUSDRUCKROUTINE
6002 REM*****************
6010 FORI=0TOIT-1
6020 PRINTB$(I)
6030 IFI/10=INT(I/10)THEN GET T$:IFT$=""
THEN6030:REM WEITER MIT TASTE
6040 NEXTI
6050 RETURN
```

```
7000 REM**************
7001 REM PRUEFROUTINE
7002 REM**************
7010 FORI=0TOIT-2
7020 IFB$(I)>B$(I+1)THENPRINT"FALSCH AB
STELLE ";I
7030 NEXTI
7040 RETURN
8000 REM***********
8001 REM SORTIEREN
8002 REM***********
8010 FORI=(IT-1)TO1STEP-1
8020 FORJ=0TOI-1
8030 IFB$(J)<=B$(J+1)THEN8070
8040 T$=B$(J)
8050 B$(J)=B$(J+1)
8060 B$(J+1)=T$
8070 NEXTJ
8080 NEXTI
8090 RETURN
```

Wenn Sie genauso vorgegangen sind wie bisher beschrieben, sollten Sie mit diesen Zeilen keine Schwierigkeiten haben; denn sie halten sich genau an die Methode, die Sie schon per Hand ausprobiert haben.
Das aufgelistete Sortierprogramm kann ohne weiteres aus dem Programm herausgenommen und in Ihren eigenen Anwendungen dort untergebracht werden, wo kleinere Datenmengen zu sortieren sind. Sie müssen nur die Zeilennummern Ihrem Programm anpassen und die Namen der Felder Ihren Erfordernissen gemäß ändern.
Starten Sie das Programm einfach mit RUN, und warten Sie auf die Bestätigung, daß die erzeugte Zufallsliste korrekt sortiert wurde. Sie werden über die Zeitspanne informiert, die das Sortieren der kopierten Zufallsliste vom Beginn bis zum Abschluß benötigte. Sie könnten das Programm auch für die Bearbeitung längerer Listen einrichten, indem Sie den Wert von IT in Zeile 5010 ändern – sofern es Ihnen nichts ausmacht, Däumchen zu drehen, während das Sortierprogramm vor sich hinläuft. Falls Sie sich doch zur Arbeit mit Listen entschließen, die wesentlich mehr als 100 Elemente umfassen, sollten Sie irgendwo eine zusätzliche Zeile mit FRE(0) einfügen. Diese sorgt beim C 64 dafür, daß die Garbage Collection regelmäßig ausgeführt wird, und der Rechner nicht von den gewaltigen Anforderungen, die das Jonglieren mit hunderten verschieden langer Strings mit sich bringt, so überstrapaziert wird, daß das Programm mit einem OUT OF MEMORY ERROR abbricht.

DER DELAYED-REPLACEMENT-SORT: EINE EINFACHE ABKÜRZUNG

Denken Sie einmal an die Zettel zurück, mit denen wir zu bestimmen versuchten, wie ein Mensch eine Liste sortiert. Ich habe bei der Gelegenheit bemerkt, daß eine vermutlich von den meisten benutzte Methode darin besteht, den höchsten oder niedrigsten Wert der Folge festzustellen und ihn sofort auf den richtigen Platz zu legen. Sie hätten auch anfangen können, indem Sie den dritten Zettel auf den richtigen Platz legten, dann den siebten, dann den ersten – im Grunde hätten Sie jede beliebige Reihenfolge wählen können. Dies wäre deshalb möglich gewesen, weil die Liste aus Werten mit regelmäßigen Lücken bestand und Sie die richtige Position der Zettel schon an der Nummer erkennen konnten. Wären die Intervalle unregelmäßig gewesen, dann wäre es Ihnen sehr viel schwerer gefallen, den Zettel zu finden, der an die dritte Stelle gehörte. Die meisten Listen haben Elemente mit unregelmäßigen Intervallen, und wenn Sie die Position eines Elementes zweifelsfrei bestimmen wollen, ist es am einfachsten, zuerst das größte oder kleinste, dann das zweitgrößte oder -kleinste etc. herauszufinden.
Genau so ging der Bubble-Sort vor. Bei jedem Durchgang suchte er sich den höchsten Wert, der noch nicht richtig plaziert war, und brachte ihn an die passende Stelle. Gleichzeitig nahm er Veränderungen an der restlichen Reihe vor, aber die meisten Vertauschungen hatten den Zweck, ein Element auf dem richtigen Platz unterzubringen. Es stellt sich die Frage, ob all diese Vertauschungen notwendig sind. Die Antwort heißt nein.
Wir wollen anhand der bekannten Zettel untersuchen, auf wie viele Vertauschungen man hätte verzichten können. Legen Sie zunächst die Zettel in derselben Reihenfolge aus, mit der wir die Bubble-Sort-Methode geprüft haben:

7 3 1 5 6 9 4 8 0 2

Nehmen Sie nun eine kleine Münze, die Sie auf den Zettel mit der 7 links außen legen. Die Münze bezeichnet die Position des Zettels mit dem höchsten Wert, den Sie gefunden haben. Das weitere Vorgehen:

1. Vergleichen Sie den Wert des Zettels mit der Münze und den des Zettels rechts daneben. Der nächste Zettel, die 3, hat einen kleineren Wert, deshalb bleibt die Münze auf der 7 liegen.

2. Setzen Sie den Vergleich weiter rechts fort, also bei der 1 und so weiter, solange die Zettel, die Sie vergleichen, im Wert niedriger sind als der Zettel mit der Münze.

3. Wenn Sie auf einen Zettel treffen, der einen höheren Wert hat als der mit der

Münze, in diesem Fall die 9, legen Sie die Münze auf diesen Zettel. Dann vergleichen Sie den Zettel mit dem neuen höchsten Wert mit denen rechts daneben: sind sie kleiner, bleibt die Münze liegen, sind sie größer, wird sie verschoben. In unserem Beispiel bleibt die Münze bis zum Schluß des Durchgangs auf der 9 liegen.

4. Legen Sie den Zettel, mit dem Sie den Durchgang beendet haben, auf den Extraplatz. An die freie Stelle kommt jetzt der Zettel mit der Münze. Dann holen Sie den Zettel aus dem Extraplatz nach links an die Stelle, wo vorher der Zettel mit der Münze lag.

5. Legen Sie die Münze auf den Zettel links außen, und wiederholen Sie den ganzen Vorgang, aber wie beim Bubble-Sort mit der Einschränkung, daß Sie bei jedem Durchgang weniger Vergleiche vornehmen, da der ungeordnete Bereich der Folge bei den vorhergehenden Durchgängen um jeweils eine Stelle verringert wurde.

Wenn Sie sich an die beschriebene Methode halten, müßten die Reihen bei jedem Durchgang wie folgt ausliegen:

```
7 3 1 5 6 9 4 8 0 2  :  START POSITION
7 3 1 5 6 2 4 8 0 9
7 3 1 5 6 2 4 0 8 9
0 3 1 5 6 2 4 7 8 9
0 3 1 5 4 2 6 7 8 9
0 3 1 2 4 5 6 7 8 9
0 3 1 2 4 5 6 7 8 9
0 2 1 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

Wenn Sie Ihrer Ansicht nach die Methode begriffen haben, erwartet Sie nun die eigentliche Lektion über dieses Sortierverfahren; denn jetzt werden wir die Vertauschungen und Vergleiche genauso abzählen wie beim Bubble-Sort. Um den Bubble-Sort nicht zu benachteiligen, zählen wir auch jedes Verschieben der Münze mit, und das erste Mal, als sie zu Anfang auf den linken äußeren Zettel gelegt wurde. Nach meiner Rechnung ergeben sich folgende Zahlen:

```
1) 9 VERGLEICHE    1 VERTAUSCHUNGEN    2 VERSCHIEBUNGEN
2) 8      "        1       "           2       "
3) 7      "        1       "           1       "
4) 6      "        1       "           4       "
```

5) 5	"	1	"	3	"
6) 4	"	0	"	3	"
7) 3	"	1	"	2	"
8) 2	"	1	"	2	"
9) 1	"	0	"	2	"

SUMME:
45 VERGLEICHE 7 VERTAUSCHUNGEN 21 VERSCHIEBUNGEN

Offenbar haben wir einen größeren Handel zwischen Vertauschungen und Verschiebungen der Münze durchgeführt. Selbst bei manueller Simulierung ist es viel einfacher, die Münze zu bewegen als zwei Zettel auszutauschen. Wird die Methode vom Computer gesteuert, bedeutet die einfache Wertänderung einer Variablen (die zur Aufzeichnung des höchsten zuletzt gefundenen Wertes dient) im Vergleich zu der dreiteiligen Operation beim Vertauschen von zwei Strings eine enorme Ersparnis.
Das Ergebnis für den ungünstigsten Fall bei der Delayed-Replacement-Sortiermethode macht die Situation noch deutlicher. Im denkbar schlechtesten Fall braucht dieses Verfahren wie der Bubble-Sort 0.5*N*(N−1) mindestens Vergleiche. Dagegen beträgt die größtmögliche Anzahl von Vertauschungen nur N−1, d. h. in unserem Beispiel neun. Die meisten Verschiebungen werden vorgenommen, wenn die Liste völlig geordnet ist. Beispielsweise ergibt ein Durchgang der vollständig sortierten Liste neun Verschiebungen des Zeigers bis zum höchsten Wert. Wenn wir uns die Anzahl der Verschiebungen pro Durchgang und die der Vergleiche im selben Durchgang merken, können wir sicher schließen, daß der erste Teil der Folge richtig geordnet ist, sobald die beiden Werte übereinstimmen. Zum Sortieren unserer Zettel in der Reihenfolge:

1 2 3 4 5 6 7 8 9 0

sind 45 Verschiebungen notwendig, was verdächtig an die bekannte Formel 0.5*N*(N−1) erinnert. Wir folgern daraus, daß der Delayed-Replacement-Sort im Höchstfall

für 100 Elemente: 4950 Vergleiche, 99 Vertauschungen und 4950 Verschiebungen,
für 1000 Elemente: 499 500 Vergleiche, 999 Vertauschungen und 499 500 Verschiebungen

erfordert.

Wenn es stimmt, daß Verschiebungen wesentlich schneller sind als Vertauschun-

gen, muß der Wegfall von etwa 500 000 Vertauschungen im zweiten Fall zu einer merklichen Zeitersparnis führen. Das ist nur durch einen Vergleich beider Sortiermethoden in der Praxis zu beweisen. Zu diesem Zweck muß nur die unten abgedruckte, in das schon vorgestellte Programm passende Routine eingegeben werden:

```
1020 GOSUB4000:TI$="000000":GOSUB9000:PR
INTTI$:GOSUB6000:GOSUB7000
1030 GOSUB4000:TI$="000000":GOSUB9000:PR
INTTI$:GOSUB6000:GOSUB7000
9000 REM****************
9001 REM SORTIERROUTINE
9002 REM****************
9010 FORI=IT-1TO1STEP-1
9020 NN=0
9030 FORJ=1TOI
9040 IFB$(J)>B$(NN)THENNN=J
9050 NEXTJ
9060 T$=B$(I)
9070 B$(I)=B$(NN)
9080 B$(NN)=T$
9090 NEXTI
9100 RETURN
```

Nach wiederholtem Programmdurchlauf erweist sich bei dieser Methode gegenüber dem einfachen Bubble-Sort für 100 Elemente eine Ersparnis von 50–60 Prozent. Gleichzeitig wird klar, daß wir damit das Bubble-Sort-Verfahren durch Entfernen überflüssiger Vertauschungen lediglich ausgebessert haben.

DER SHELL-METZNER-SORT: DIE WIRKUNG VON ZWEIERPOTENZEN

Zum wirklich effizienten Sortieren müssen wir den relativ sicheren Weg wiederholter, systematischer Datendurchgänge auf der Suche nach größten und kleinsten Werten verlassen. Wie bei der Binärrecherche müssen wir uns im Vertrauen auf die Fähigkeit binärer Methoden, Ordnung in das mutmaßliche Chaos zu bringen, auf eine scheinbar unsystematische Methode einlassen. Obwohl bei dieser Sortiermethode in den Anfangsstadien offenbar verrückte Vertauschungen stattfinden, ordnet sich das Durcheinander umfangreicher Listen weit schneller als bei den anderen bisher verwendeten Methoden.

Der Ablauf dieses Verfahrens ist nicht leicht zu begreifen, aber wir werden versuchen, es wieder mit unseren ungeordneten Zetteln nachzuvollziehen. Hierfür brauchen Sie jedoch außer den Zetteln vier Münzen mit unterschiedlichen Werten und ein Stück Papier. Wir nennen die vier Münzen in aufsteigender Wertfolge A, B, C und D. Auf dem Papier notieren wir die veränderlichen Werte einer Variablen, der wir den Namen GAP (Lücke) geben.

Die Methode arbeitet nach dem Prinzip, Elemente auszutauschen, deren Abstand voneinander die höchstmögliche Zweierpotenz beträgt, die in der Liste der zu sortierenden Daten vorkommen kann. Nachdem wir alle entsprechenden Vertauschungen ausgeführt haben, machen wir mit den halb so weit voneinander entfernten Elementen weiter etc. In unserem Fall beginnen wir mit den Vertauschungen bei einem Ausgangsabstand von 8; also schreiben Sie auf Ihr Blatt: GAP=8. Jetzt legen Sie die Münzen A und B auf den linken äußeren Zettel (7). Im Verlauf des Sortierens werden wir Münze A (mit dem geringsten Wert) zur Bezeichnung des linken der beiden zu vertauschenden Zettel benutzen. Münze B brauchen wir zur Markierung der Stelle innerhalb der Reihe, die wir bei der Suche nach dem jeweiligen Wert von GAP erreicht haben. Münze C wird jetzt auf den äußersten rechten Zettel gelegt, von dem aus noch mit GAP umgestellt werden kann. Im Beispiel ist das Position 2, da 2 plus der Wert von GAP an das äußere Ende der Reihe führt. Münze D wird während des Sortierens umhergeschoben und markiert die Position des Zettels, den GAP rechts von dem Zettel mit Münze A unterbringt.
Spielen Sie jetzt die unten beschriebenen Schritte nach:

1. Legen Sie die Zettel wie vorher in der Reihenfolge 7 3 1 5 6 9 4 8 0 2 aus.

2. A befindet sich in Position 1; also kommt D in Position 1+GAP (=9). Die entsprechenden Zettel sind 7 und 0, werden also ausgetauscht. Die Münzen bleiben noch liegen.

3. Auch wenn es zu diesem Zeitpunkt noch unsinnig erscheint, versuchen wir jetzt, A um soviele Stellen nach links zu verschieben, wie der Wert von GAP beträgt. Den Grund werde ich später nennen. Im Augenblick reicht es zu wissen, daß wir den Schritt in Erwägung ziehen, ihn nicht durchführen können und deshalb B um eins nach rechts verschieben.

4. Immer wenn B um eins nach rechts verschoben wird, folgt A automatisch an dieselbe Stelle. Legen Sie also A zu B auf Position 2. C befindet sich ebenfalls dort.

5. D bekommt jetzt den Platz A+GAP, d. h. 10. Die entsprechenden Zettel sind 3 und 2, werden also ausgetauscht.

6. Wir prüfen erneut die Möglichkeit, A um GAP Stellen nach links zu schieben. Da das auch jetzt nicht möglich ist, schieben wir B um eins nach rechts.

7. Wie Sie jetzt sehen, ist C und B überholt worden. Das bedeutet, der erste Datendurchgang ist beendet. Nach jedem beendeten Durchgang müssen wir den Wert von GAP halbieren und alle Münzen neu auslegen. Streichen Sie die 8 auf dem Blatt durch, und schreiben Sie statt dessen 4 hin. B kommt wieder links außen zu liegen, gefolgt von A. C gehört auf die Position 10−GAP, das ist zur Zeit 6, und steht für die äußerste rechte Stelle, von der aus ein Austausch mit GAP4 noch stattfinden könnte.

8. Jetzt können wir erneut versuchen, Elemente auszutauschen. Bringen Sie D auf A+GAP, also auf Position 5. Die dazugehörigen Zettel sind 0 und 6, folglich können sie nicht vertauscht werden. Kann ein Austausch nicht stattfinden, wird B um eins nach rechts verschoben, und A folgt auf dieselbe neue Position. D kommt auf Position 6 (A+GAP).

9. Die Verschiebung von B, A und D wird solange wiederholt, bis B und A sich in Position 6 befinden. Bis zu diesem Punkt konnten keine Vertauschungen mit GAP4 ausgeführt werden.

10. B und A liegen an sechster Stelle, und die Werte der beiden von A und D markierten Zettel sind 9 und 3. Sie können also ausgetauscht werden.

11. Da ein Austausch stattgefunden hat, überlegen wir wieder, ob A um GAP Stellen nach links verschoben werden kann. Diesmal ist es möglich, also bringen wir A auf die neue Position 2 und danach D auf A+GAP, d. h. 6. Die beiden Zettel 2 und 3 können nicht ausgetauscht werden. Wir haben das aus folgendem Grund versucht: Jedesmal, wenn ein Zettel weiter unten in der Reihenfolge zu liegen kommt, müssen wir prüfen, ob er hätte ausgetauscht werden können, wenn er von Anfang an auf dem neuen Platz gelegen hätte. Wäre die Vertauschung möglich gewesen, hätten wir versucht, A wieder um GAP Stellen nach links zu verschieben. B bleibt währenddessen auf seinem Platz.

12. Jedesmal, wenn ein Austausch nicht stattfinden kann oder A nicht um GAP Stellen nach links verschoben werden kann, wird B verschoben, gefolgt von A. Das bedeutet, B hat jetzt C überholt, und damit ist der Durchgang abgeschlossen.

13. Verringern Sie den Wert von GAP um die Hälfte auf 2. Bringen Sie A und B auf Position 1 zurück. Legen Sie C auf Position 10−GAP (8) und D auf Position A+GAP (3).

14. Nach vergeblich versuchten Vertauschungen erreichen B und A Position 4, D Position 6. Tauschen Sie die 5 und die 3 aus. A wandert auf Position 2 zurück. Da kein weiterer Austausch möglich ist, wird B um eins nach rechts gerückt.

15. B und A befinden sich jetzt auf Position 5, und Sie können die 6 und die 4 austauschen. Die Verschiebung von A nach links ermöglicht keine weiteren Vertauschungen, also wird B verschoben.

16. Da in Position 6, 7 und 8 nichts ausgetauscht werden kann, endet der Durchgang ohne weitere Vertauschungen.

17. GAP wird auf 1 verringert, A und B werden wieder auf Position 1 gebracht. C kommt auf Position 10−GAP (9) und D auf Position A+GAP (2).

18. Die Verschiebung der Zeiger bei GAP1 bewirkt einen Durchgang in derselben Art wie beim Bubble-Sort. Sie sollten jetzt in der Lage sein, die Münzen ohne weitere Anleitung zu verschieben.

19. Wenn alles geklappt hat, ist die Liste jetzt sortiert. Das bedeutet, daß GAP beim nächsten Durchgang kleiner als 1 wäre.

20. Um zu veranschaulichen, warum der Zeiger A nach jeder Vertauschung nach links verschoben wird, können Sie diesen letzten, dem Bubble-Sort ähnlichen Durchgang versuchsweise bei 5 Zetteln der Reihenfolge 0 1 3 4 2 ausprobieren. Gehen Sie mit den Münzen wie oben vor, und setzen Sie GAP auf 1. Sie stellen fest, daß kein Austausch möglich ist, bevor B auf Position 4 gelangt. Außerdem kann die Liste so nicht vollständig sortiert werden. Erst die Verschiebung von A um GAP Stellen nach links (auf Position 3) ermöglicht den zusätzlichen Austausch, mit dem 2 an die korrekte Stelle gebracht werden kann.
Wir werden bei der Shell-Metzner-Sortiermethode nicht versuchen, die Zahl der Vertauschungen, Vergleiche und Verschiebungen zu analysieren – teils, weil es den ganzen Tag dauern könnte, und teils, weil es bei dieser geringen Anzahl von Elementen wenig aussagen würde. Denn die Vorteile dieser Methode werden erst beim Sortieren größerer Datenmengen deutlich. Das heißt nicht, daß das Shell-Metzner-Verfahren nicht auch bei kleineren Datenvolumen angewendet werden kann. Ob eine kaum merkbare Beschleunigung die erforderliche zusätzliche Programmierarbeit aufwiegt, ist reine Ansichtssache.
Wenn Sie mit der folgenden Routine den Shell-Metzner-Sort in das Sortiertestprogramm eingeben, werden Sie bei größeren Datenmengen eine enorme Zeitersparnis feststellen. Grundsätzlich variiert die Geschwindigkeit des Bubble-Sort etwa entsprechend der Formel $N^2/2$, d. h. Erhöhungen von N werden miteinander

multipliziert. Bei der Shell-Metzner-Methode ändert sich der Zeitaufwand etwa nach der Regel:

```
1.6*N*(LOG N/LOG 2).
```

Das bedeutet, wenn im ungünstigsten Fall das Sortieren von 10 Elementen mit dem Bubble-Sort eine Sekunde dauerte, würde es bei 100 Elementen 100 Sekunden und bei 1000 Elementen 10 000 Sekunden dauern.
Wenn wir auch beim Shell-Metzner-Verfahren wiederum vom ungünstigsten Fall ausgehen und annehmen, daß das Sortieren von 10 Elementen eine Sekunde dauert, dann würde es bei 100 Elementen 20 Sekunden und bei 1000 Elementen 3000 Sekunden dauern.
Dies sind keine präzisen Zahlenangaben zur Geschwindigkeit von Sortierprogrammen auf dem C 64, aber sie veranschaulichen die ungeheuren Unterschiede beim Sortieren wachsender Datenvolumen. Sie können das für sich nachprüfen, indem Sie im Testprogramm mit verschieden langen Listen experimentieren.

```
1040 GOSUB4000:TI$="000000":GOSUB10000:P
RINTTI$:GOSUB6000:GOSUB7000
10000 REM*************************
10001 REM SHELL-METZNER VERFAHREN
10002 REM*************************
10010 GAP=2↑(INT(LOG(IT-1)/LOG(2))+1)
10020 GAP=GAP/2
10030 C=IT-GAP-1:B=0:IFGAP<1THENRETURN
10040 A=B
10050 D=A+GAP:IFB$(A)>B$(D)THEN10080
10060 B=B+1:IFB>CTHEN10020
10070 GOTO10040
10080 T$=B$(A)
10090 B$(A)=B$(D)
10100 B$(D)=T$
10110 A=A-GAP:IFA<0THEN10060
10120 GOTO10050
```

In der Praxis erreichen Sie bei der Anwendung der Shell-Metzner-Methode auf eine Liste von 100 Elementen eine durchschnittliche Einsparung von nur 60 Prozent gegenüber dem Bubble-Sort. Wird der Umfang der Liste jedoch vergrößert, so kommt man zu weitaus beeindruckenderen Ergebnissen. Für eine Liste mit 200 Elementen braucht das Shell-Metzner-Verfahren nur etwa 15 Prozent der Zeit, die der Bubble-Sort in Anspruch nähme.

SCHLUSS

Das Thema Sortieren haben wir nicht einmal annähernd erschöpfend behandelt. Es gibt andere Sortiermethoden, die zwar keine astronomische, aber eine wesentliche Zeitersparnis gegenüber dem Shell-Metzner-Verfahren bringen. Das Problem dabei ist, daß sie fast ausnahmslos die Bereitstellung zusätzlicher Speicherkapazität erfordern, damit Daten zwischen die Hauptliste und eine oder mehrere Unterlisten eingeschoben werden können. Für die Praxis heißt das, ihr Tempo im Vergleich mit Shell-Metzner zeigt sich erst bei großen Datenmengen, die für die praktische Anwendung auf einem Heimcomputer zu speicherextensiv sind. Mit den Sortiermethoden in diesem Kapitel können Sie von der kleinsten bis zur größten fast jede beliebige Datenmenge verarbeiten, die Methoden dem Schwierigkeitsgrad der Aufgabe anpassen und in vielen Fällen Ihre Programme enorm beschleunigen.

KAPITEL 9

DATENSTRUKTUREN I

Dieses Buch geht u. a. von dem Grundsatz aus, daß die meisten nützlichen Programme Daten speichern. Daten zu speichern und zu verarbeiten ist das, was Mikrocomputer am besten können; auf diesem Gebiet sind sie den Möglichkeiten jeder anderen Methode weit überlegen. Gelegentlich kann jedoch gerade die Leichtigkeit, mit der Mikrocomputer mit Informationen umgehen, als Ausrede für eine nachlässige und unüberlegte Datenspeicherung dienen. Erlaubt ist anscheinend alles, solange das Programm die nötigen Daten im Gedächtnis behalten kann. Der Nachteil bei dieser Vorgehensweise ist, daß die Leistungsfähigkeit eines Programms durch eine falsche Datenstruktur außerordentlich beschnitten werden kann: Das Programm wird verlangsamt, das zu verarbeitende Material wird quantitativ eingeschränkt, oder die Struktur des Arbeitsprogramms wird unnötig verkompliziert. In diesem Kapitel wollen wir einige der zahlreichen Möglichkeiten untersuchen, Daten so zu strukturieren, daß sie den Erfordernissen des Programms angepaßt sind und Geschwindigkeit und Speicherplatz maximieren.

EINFACHE DATENSTRUKTUREN: DAS MASSGESCHNEIDERTE FELD

Die bei weitem einfachste Struktur zum Speichern jeder Art von Informationen ist ein Feld, dessen Dimensionen genau darauf zugeschnitten sind, wie die Information selbst gegliedert ist. Ein Beispiel wäre etwa ein Programm zur Aufzeichnung von Umsatz, Profit, Steuern und Investitionen mehrerer Unternehmen. Hier könnte man ein numerisches Feld ARRAY (X,3) definieren, wobei X die Anzahl der Unternehmen ist, und Daten könnte man z. B. mit folgender Routine eingeben:

```
100 INPUT"FIRMENNUMMER ";CY
110 INPUT"UMSATZ ";ARRAY(CY,0)
120 INPUT"GEWINN ";ARRAY(CY,1)
130 INPUT"STEUER ";ARRAY(CY,2)
140 INPUT"INVESTITIONEN ";ARRAY(CY,3)
```

In einem Dateiprogramm zum Speichern von Namen, Adressen und Telefonnummern könnte man ein Feld ARRAY$(500,2) definieren und Eingaben mit einer ähnlichen Routine wie der obigen machen.
Solche Strukturen dürfen so komplex sein, wie es das Programm erfordert. Zum Beispiel hätte es bei dem oben erwähnten Programm für die Unternehmen nötig

sein können, alle Daten über einen Zeitraum von fünf Jahren aufzuzeichnen. In diesem Fall würde das Feld als ARRAY(X,3,4) definiert, und jede der verschiedenen Kategorien bekäme eine Schleife für fünf Eingaben, so daß aus Zeile 110 eine eigene Routine würde:

```
110 FORI=0TO4
112 PRINT"UMSATZ IN";1980+I;" ";:INPUTAR
RAY(CY,0,I)
114 NEXT
```

Solche maßgeschneiderten Felder haben den Vorteil, das Speichern und den Zugriff auf Daten auf direktem Wege zu erlauben. Alles hat seinen eindeutigen Platz, und zum Auffinden eines Datenelements braucht man nur die Nummer des Unternehmens, die Kategorie der Information – z. B. Steuern – und das Jahr zu kennen. Daten in solchen Feldern lassen sich auch auf andere Weisen einfach abrufen. Will der Benutzer z. B. wissen, welchen Profit jedes Unternehmen 1978 gemacht hat, würde eine einfache Schleife

```
100 FOR I=0 TO ITEMS:PRINT ARRAY(I,2,1):
NEXT
```

die Information problemlos bereitstellen.
Ein weiterer Vorteil maßgeschneiderter Felder besteht in der Art, in der ganze Gruppen von Feldern verglichen werden können, die verschiedene, aber parallele Reihen von Informationen enthalten. In unserem Beispiel wäre es einfach, ein weiteres Feld mit den Namen der beteiligten Unternehmen zu definieren, und auf diese Namen könnte man mit genau derselben Variablen zugreifen, die bestimmt, welcher Datensatz ausgedruckt werden soll. In komplexen Datenverarbeitungsprogrammen ist eine Reihe verschiedener Felder, die alle parallele Informationen enthalten, nichts Ungewöhnliches. Nehmen wir den Fall, daß vielfältige Informationen gespeichert werden sollen, die großenteils ähnlich strukturiert sind, sich z. B. auf verschiedene Monate des Jahres beziehen. Dann sind maßgeschneiderte Felder oft der einzige praktische Weg, die Zahl der Variablen zu kontrollieren, die zur Beschaffung von Informationen von vielen verschiedenen Stellen nötig sind. Wenn alle Daten sich auf Monate des Jahres bezögen, ließen sich vielleicht manchmal wichtige Informationen aus jedem Feld mit einer einzigen Variablen gewinnen, die für den betreffenden Monat steht.
Maßgeschneiderte Felder haben jedoch ihre Nachteile, unter anderem den Speicherplatz, den selbst ein verhältnismäßig harmloses Feld verbraucht. Ein numerisches Feld mit den Dimensionen (100,10,10) sieht wohl nicht übermäßig groß aus, aber es benötigt 50 000 Bytes Speicherplatz – viel mehr, als in BASIC auf dem

C 64 verfügbar ist. Um den für ein gewünschtes Feld notwendigen Speicherplatz zu berechnen, multiplizieren Sie einfach alle bei der Definition des Feldes in den Klammern vorkommenden Zahlen und multiplizieren das Ergebnis dann mit

2 bei einem Integerfeld,
3 bei einem Stringfeld,
5 bei einem Gleitkommafeld;

wobei Sie berücksichtigen müssen, daß die für ein Stringfeld erhaltene Zahl nur den Verwaltungsbedarf enthält, und daß alles im Feld Abgelegte zum benötigten Speicherplatz hinzuaddiert werden muß. Bei numerischen Feldern ist allen Elementen der Wert 0 zugewiesen, so daß sich der erforderliche Speicherplatz nicht erhöht, wenn Elemente geändert werden.
Beim Definieren großer und komplexer Felder ist es deshalb unerläßlich zu bestimmen, ob der Platz im Feld voll genutzt wird. Häufig müssen Dateien mit einer großen Anzahl von Kategorien definiert werden, ohne daß jede Eintragung Informationen in jeder Kategorie hat. Solche Felder heißen 'schwachbesetzt' (sparse), und nur sehr wenige Anwendungen sind mit dem unbenutzten Speicherplatz, den ein komplexes, aber schwachbesetztes Feld erzeugen kann, nicht überfordert.
Zusammenfassend möchte ich sagen: Bemühen Sie sich, eine Ihren Informationen genau angepaßte Datenstruktur zu planen, die alle Daten aufnehmen kann, die Sie voraussichtlich zu diesem Thema abspeichern wollen. Denn das wird Ihr späteres Programm zweifellos erheblich vereinfachen. Früher oder später werden Sie allerdings mit Anwendungen konfrontiert sein, wo schon die Menge der Elemente oder die Komplexität der Struktur ein maßgeschneidertes Feld von vornherein als zu aufwendig erscheinen läßt. Dann werden Sie sich mit den anderen Speicherarten in diesem Kapitel näher beschäftigten müssen.

DATENSTRUKTUREN FÜR ZAHLEN

BYTEZAHLEN IN INTEGERFELDERN

Der C 64 besitzt eine sehr sparsame Speichermethode für den Zahlenbereich, der in Datenverarbeitungsprogrammen meistens benutzt wird. Die meisten Werte in diesen Programmen – nicht die zum Speichern, sondern die zur Programmsteuerung vorgesehenen Werte – sind ganze Zahlen (ohne Nachkommastellen) in einem Bereich, der durch die Grenzen z. B. von Feldern und Stringlängen bestimmt ist. Fast immer ist es reine Verschwendung, solche Werte in normalen numerischen Feldern unterzubringen, da jedes Element eines Gleitkommafeldes fünf Bytes im Speicher belegt, während ein Integerfeld nur zwei benötigt.

Integerfelder können jedoch dazu dienen, den für kleine numerische Werte nötigen Platz noch weiter zu reduzieren, indem in jedes Feldelement mehr als ein Wert abgelegt wird. Das ist möglich, weil in einem Integerfeld jedes Element eine Zahl von −32768 bis +32767 aufnehmen kann; der Bereich umfaßt also 65536 Zahlen. Der verfügbare Bereich stellt einfach dar, was in einer 16-Bit-Zahl abgelegt werden kann, die das werthöchste der 16 Bits zur Aufzeichnung des positiven oder negativen Vorzeichens der Zahl benutzt; darauf wird im Abschnitt über die Funktion FRE im Kapitel 'Strings' näher eingegangen. Für uns ist wichtig, daß in einem Integerfeld ein Element eine Zahl bis 256*256−1 aufnehmen kann (wenn der negative Bereich zugänglich ist). Wenn wir also zwei Zahlen im Bereich 0−255 nehmen, eine davon mit 256 multiplizieren und dann die zweite addieren, werden effektiv zwei Zahlen in einer größeren abgespeichert. Beispielsweise könnte man die Zahlen 237 und 76 in dieser Reihenfolge mit Hilfe einer BASIC-Zeile speichern:

```
100 NN%=256*237+76:REM ANTWORT = 60748
```

Da der zulässige Zahlenbereich nicht 0−65535, sondern −32768 bis +32767 umfaßt, muß jede Zahl über 32767, die in das Feld abgelegt werden soll, durch Subtrahieren von 65536 geändert werden. Daher müßte die BASIC-Zeile heißen:

```
100 NN%=256*237+76:IF NN>32767THENNN=NN-
65536
```

Zur Entschlüsselung zweier Zahlen, die so in ein Element eines Integerfeldes abgespeichert werden, genügt eine weitere einfache BASIC-Zeile:

```
100 NN=A%(X)-65536*(A%(X)<0):N1=INT(NN/2
56):N2=NN-256*N1
```

Der Vorteil bei dieser Art der Zahlendarstellung liegt darin, daß man damit fast genauso viel Speicherplatz sparen kann wie beim Ablegen in einen String, d. h. ein Byte pro Zeichen plus geringer Verwaltungsbedarf für das Feld selbst. Außerdem können die Zahlen direkt auf Band oder Diskette gespeichert und geladen werden, ohne vorheriges Übersetzen von Zeichen in Zahlen und umgekehrt. Eher nachteilig dabei ist es, daß diese Methode bei normalem Gebrauch tatsächlich doppelt so langsam ist wie die Übersetzung von Zeichenwerten. Das wird Ihnen unter normalen Umständen nicht auffallen, aber Sie müssen sich schon je nach Ausmaß der beabsichtigten Lade- und Speichervorgänge zwischen den beiden Methoden entscheiden.
Zum Abspeichern einer Reihe von Werten im Bereich 0−255 ist folgende Routine geeignet:

```
100 DIMA%(99)
110 INPUT"POSITION ";PP
120 INPUT"WERT ";NN
130 TT=A%(PP/2)-65536*(A%(PP/2)<0)
140 IF PP AND 1 THENTT=256*INT(TT/256)+N
N
150 IF NOT PP AND 1 THENTT=(TT+255)+256*
NN
160 A%(PP/2)=TT+65536*(TT>32767)
170 N1=INT(A%(PP/2)/256):N2=A%(PP/2)-256
*N1
180 PRINTN1,N2
190 GOTO110
```

Zeile 130 berechnet den aktuellen Wert des richtigen Elements von A% und befaßt sich mit dem Problem negativer Zahlen. Die richtige Position in A% ist durch PP/2 festgelegt, so daß die ersten beiden Werte in Element Null, die nächsten beiden in Element Eins etc. abgelegt werden. Beachten Sie, daß Zahlen nicht abgerundet werden müssen, wenn die Division von PP durch 2 z. B. 2.5 ergibt; denn der C 64 liest A%(2.5) als A%(2). In Zeile 140 entdeckt 'IF PP AND 1' Werte mit ungerader Position; denn der logische Ausdruck PP AND 1 ist nur für ungerade Werte von PP wahr. Ist der Wert von PP ungerade, dann ist der abzulegende Wert der niedrigere der beiden Werte, die in das betreffende Element des Feldes abgelegt werden. Wir erhalten also den oberen Wert in der Form 256*INT(NN/256) und setzen auf diese Weise denjenigen Teil der Zahl auf Null, der weniger als ein ganzes Vielfaches von 256 beträgt. Dann addieren wir die zu speichernde Zahl, die so zum unteren der beiden in diesem Element abgelegten Werte wird. In Zeile 150 ermittelt 'IF NOT PP AND 1' Werte mit geradem Index, die in die obere Position des betreffenden Elements im Feld abgelegt werden sollen. Das wird durch Verknüpfung von TT und 255 mit AND erreicht. Dadurch bleiben alle Bits in NN enthalten, die Zahlen unter 128 darstellen, und alle anderen werden eliminiert. Zum Ergebnis wird jetzt 256 mal der zu speichernde Wert addiert; er wird so zum oberen der beiden in dem betreffenden Element abgelegten Werte. Zeile 160 paßt TT wieder dem Bereich −32768 bis +32767 an und legt es wieder in A% ab.
Eine solche Struktur ist leichter zu zerlegen als zu erzeugen:

```
200 FORI=0TO99
210 NN=A%(I/2)-65536*(A%(I/2)<0)
220 IF NOT I AND 1 THEN NN =INT(NN/256)
225 IF I AND 1 THEN NN =NN-256*INT(NN/25
6)
```

```
230 PRINTNN
240 IFI/10=INT(I/10)THEN GET T$:IFT$=""T
HEN240:REM WEITER MIT TASTE
250 NEXTI
```

Der Vorteil dieser Methode zur Einsparung von Speicherplatz liegt auf der Hand. Es sollte allerdings betont werden, daß sie mit größerem Zeitaufwand beim Speichern und Entschlüsseln von Werten bezahlt werden muß, wenn ständig vom Programm auf diese Werte zugegriffen wird. Einen Zeitgewinn bringt das Verfahren dagegen beim Laden und Speichern auf Band oder Diskette, da nur halb soviele Elemente verarbeitet werden müssen.

ABLEGEN IM FREIEN SPEICHER

Wer den freien Speicherplatz restlos für sein Programm nutzen möchte oder muß, sollte nicht vergessen, daß der C 64 noch 4096 Bytes in Reserve hat, die für das Erzeugen und Speichern von Variablen in normalem BASIC nicht zugänglich sind. Jedoch kann dieser Bereich, der bei der Speicheradresse 49152 beginnt, mit Hilfe von POKE- und PEEK-Befehlen genutzt werden. Fast jede ordentliche Struktur von numerischen Daten oder sogar Stringdaten kann dort abgelegt werden, wenn Sie bereit sind, einige Sorgfalt auf das Ausarbeiten der Struktur und der für den Zugriff auf die Daten benutzten Variablen aufzuwenden.
Am leichtesten lassen sich in einem solchen freien Speicherbereich Daten in Form von Byte-Werten ablegen, d. h. ganze Zahlen von 0 bis 255. Um z. B. ein Feld mit den Dimensionen 50 mal 50 zu simulieren, können Werte mit einem Befehl folgender Art gespeichert werden:

```
100 POKE 49152+50*X+Y,NN
```

Dabei ist X die Zeilennummer, von 0 an gezählt. Y ist die Spaltennummer, ebenfalls von 0 an gezählt, und NN die zu speichernde Zahl. Einlesen kann man die Werte mit dem Gegenstück dieses Befehls, diesmal mit PEEK:

```
100 PEEK(49152+50*X+Y)
```

Komplexer strukturierte Felder mit mehr als zwei Dimensionen lassen sich unter Berücksichtigung einiger Regeln erzeugen:

1. Bestimmen Sie, wie Ihr Feld aussehen würde, wenn es in BASIC dimensioniert wäre, z. B. A(20,10,5).

2. Berechnen Sie die Anzahl der Elemente in jeder Einheit je Dimension, angefangen von rechts. Im obigen Beispiel hätte die Dimension ganz rechts in jeder Einheit ein Element, die zweite von rechts in jeder Einheit hätte fünf Elemente und die linke je 50 Elemente.

3. Um eine Position in dem hypothetischen Feld festzulegen, beginnen Sie mit der Anfangsadresse des Speicherbereichs, den Sie benutzen wollen. Dann multiplizieren Sie den Index für jede Dimension mit der eben berechneten Anzahl der Elemente und addieren alles zur Anfangsadresse. Auf diese Weise bekäme das Element 14,7,3 aus dem obigen Beispiel die Adresse 49152+50*14+5*7+3.

Für Zahlen, die zwei oder mehr Bytes in Anspruch nehmen, lassen sich komplexere Strukturen erzeugen, aber Sie werden immer auf die im vorigen Kapitel beschriebenen Methoden zurückgreifen müssen, um Zahlen in einzelne Bytes aufzuteilen. Bei Zwei-Byte-Zahlen würde die Regel lauten: Man multipliziert die aus den Dimensionen berechnete Adresse mit zwei und POKEt dann die zwei Bytes, aus denen die abzulegende Zahl besteht, in die errechnete und die darauffolgende Adresse. Innerhalb eines wie im o. g. Beispiel dimensionierten Feldes aus Zwei-Byte-Elementen kann man eine Zahl von 0–65535 mit dieser Routine in Position 14,7,3 ablegen:

```
100 NN=65
110 A$=CHR$(NN)
120 PRINT A$
```

ZAHLEN IN STRINGS

Wir haben auch in Kapitel 8 gesehen, wie schnell und auch ökonomisch es sein kann, Daten in Strings variabler Länge abzulegen. Bei diesem Verfahren können neue Datenelemente an den Anfang, das Ende oder die Mitte eines Datenblocks eingefügt werden, wobei alle anderen bereits gespeicherten Elemente automatisch neu positioniert werden. Darüber hinaus ist Ihnen bekannt, daß für diese Daten die maximale Länge von 255 Bytes pro Einzelstring nicht gelten muß, da mehrere Elemente eines Stringfeldes in Anspruch genommen werden können.
Diese Erkenntnis ist nicht nur für das Problem des Speicherns kleiner Stringdaten-Elemente von Bedeutung, Strings können häufig sehr sinnvoll beim Speichern numerischer Werte eingesetzt werden. Das ist möglich, weil im Zeichensatz des C 64 jedes Zeichen seinen eigenen Wert – genannt 'ASCII-Wert' – hat: eine Zahl von 0–255. In dieser numerischen Form speichert der C 64 die Zeichen. BASIC

liefert dem Programmierer die beiden Funktionen ASC und CHR$, die Zahlen in Zeichen umwandeln und umgekehrt, so daß Zahlen in Zeichen übersetzt, als String gespeichert und später in Zahlen zurückverwandelt werden können.
Folgende Zeilen demonstrieren die Umwandlung einer Zahl in ein Zeichen:

```
100 N1=INT(NN/256):N2=NN-256*N1
110 PP=49152+2*(50*14+5*7+3)
120 POKEPP,N1:POKEPP+1,N2
```

Wenn Sie diese Zeilen durchlaufen lassen, erscheint der Buchstabe A, dessen Zeichenwert genau 65 ist. Mit dieser Zeile machen Sie die Übersetzung rückgängig:

```
130 PRINT ASC(A$)
```

Daraus ersehen wir, daß CHR$ die Funktion hat, einen bestimmten Code-Wert in ein Zeichen umzuwandeln, während ASC den Code-Wert eines bestimmten Stringzeichens ergibt. Wir können beide Funktionen kombinieren, um das Speichern kleiner numerischer Werte ökonomisch und flexibel zu gestalten:

```
100 A$=""
110 INPUT NN
120 IFNN>=0THENA$=A$+CHR$(NN):GOTO110
130 FORI=1TOLEN(A$)
140 PRINTASC(MID$(A$,I))
150 NEXTI
```

Diese kurze Routine erlaubt die Eingabe von bis zu 255 Zahlen zwischen 0 und 255. Wenn Sie eine negative Zahl eingeben, wird die zweite Hälfte der Routine aktiv und gibt die Daten in der Reihenfolge ihrer Eingabe aus. Beachten Sie, daß die Funktion MID$ in Zeile 140 nicht wirklich festlegt, daß der Teilstring, aus dem ein ASC-Wert abgeleitet werden soll, nur ein Zeichen umfaßt. MID$(A$,I) bezeichnet in Wirklichkeit alles von A$ ab Zeichenposition I. Das liegt an der Eigenschaft der ASC-Funktion, nur jeweils auf das erste Zeichen des vorgefundenen Strings einzuwirken. Dieses Verfahren ist insofern nützlich, als es den Umgang mit Strings nicht mehr auf einfaches Anfügen von Zeichen an einen vorhandenen String beschränkt. Zeile 120 hätte ebensogut heißen können:

```
120 IF NN>=0THENA$=CHR$(NN)+A$:GOTO110
```

so daß die Routine die Zahlen in umgekehrter Reihenfolge abgelegt hätte, was in einem normalen Feld kaum möglich wäre, ohne die vorhandenen Daten ständig weiterzuschieben.
Genauso können wir uns der Methoden aus Kapitel 3 bedienen, um Zeichen irgendwo in den gegebenen String einzufügen:

```
100 A$=""
110 INPUT"EINZUFUEGENDE ZAHL";NN
120 PRINT"POSITION (1TO';LEN(A$)+1;')";:
INPUT PP
130 IFNN>=0THENA$=LEFT$(A$,PP-1)+CHR$(NN
)+MID$(A$,PP):GOTO110
140 FORI=1TOLEN(A$):PRINTASC(MID$(A$,I))
:NEXT
```

Bei Werten über 255 spart die Anwendung von Strings zum Speichern von Zahlenwerten zwar keinen Speicherplatz, sie bietet aber die sehr hilfreiche Flexibilität, Werte beliebig zwischen vorhandene Daten einzuschieben, ohne alles andere nach hinten verschieben zu müssen. Die nachstehende Routine fügt eine Zwei-Byte-Zahl (0–65535) irgendwo in einen einzelnen String ein:

```
100 A$=""
110 INPUT"EINZUFUEGENDE ZAHL";NN
115 N1=INT(NN/256):N2=NN-256*N1
120 PRINT"POSITION (1TO';LEN(A$)/2+1;')"
;:INPUT PP:PP=PP*2
130 IFNN>=0THENA$=LEFT$(A$,PP-2)+CHR$(N1
)+CHR$(N2)+MID$(A$,PP-1):GOTO110
140 FORI=1TOLEN(A$)STEP2
150 PRINT256*ASC(MID$(A$,I))+ASC(MID$(A$
,I+1))
160 NEXTI
```

Löschungen können problemlos mit Hilfe der in Kapitel 3 erklärten Techniken vorgenommen werden, und die Methoden für Stringfelder im selben Kapitel können zur Kapazitätserweiterung eines einzelnen Strings eingesetzt werden. Das wird weiter unten in diesem Kapitel im Abschnitt über Zeigerfelder erläutert.
Für Programme, in denen ständig Werte eingefügt oder entfernt werden müssen, ist das Speichern von Zahlen in Strings also eine wirkliche Alternative. Der größte

Nachteil dabei ist, daß viele Zeichen, auch wenn sie im Speicher als Stringzeichen abgelegt werden können, nicht druckbar sind, d. h. nicht zu den Zeichen gehören, die der C 64 normal ausdrucken kann. Das ist für die Darstellung auf dem Bildschirm kaum von Belang, da die Zeichen selbst ohnehin bedeutungslos wären – nur die ASCII-Werte sind für uns von Interesse. Beim Laden und Speichern, wo auch PRINT in Form von PRINT# verwendet werden muß, macht es dagegen doch etwas aus. Leider kann der C 64 keine nicht druckbaren Zeichen als solche laden und speichern; sie müssen in Zahlen umgewandelt und in dieser Form gespeichert werden. Für einen Zeichenstring A$, in den numerische Werte eingelesen werden, müßte die Speicherroutine also folgendes enthalten:

```
100 PRINT#1,LEN(A$)
110 FORI=1TOLEN(A$)
120 PRINT#1,ASC(MID$(A$,I))
130 NEXTI
```

und beim Laden müßte der String wiederhergestellt werden:

```
100 INPUT#1,LL
110 FORI=1TOLL
120 INPUT#1,TT:A$=A$+CHR$(LL)
130 NEXTI
```

Da dieses Vorgehen die zum Laden und Speichern nötige Zeit erheblich verlängert, ist zu überlegen, ob der Lade- und Speicheraufwand trotz der Vorteile während des Programmlaufs das Ablegen im Stringformat lohnend erscheinen läßt.

STACKS

Die Methode, Zahlen in Strings abzulegen, wird auch für die Erzeugung von Stacks angewendet. Der Stack funktioniert nach demselben Prinzip wie eine (heute aussterbende) Briefablage im Büro: Briefe werden in der Reihenfolge ihres Eingangs auf einen Dorn gespießt und zu einem passenden Zeitpunkt wieder abgenommen und bearbeitet. Durch dieses Prinzip der Ablage wird immer der zuletzt eingegangene Brief als erster entnommen ('last in – first out').
Stacks sind für die Datenverarbeitung sehr wichtig, da viele Operationen, die ein Computer ausführt, von den Informationen bestimmt werden, die in dem 'Stack' genannten Speicherbereich abgelegt sind. Zum Beispiel GOSUBs: In einer beliebigen Kette von GOSUBs ist die Rücksprungadresse, die das Programm mit RETURN

anspringt, immer das *letzte* GOSUB. Die Adresse jedes GOSUB wird im Stack obenauf abgelegt, und bei jeder RETURN-Anweisung wird die oberste Adresse im Stack abgearbeitet.
In BASIC-Programmen ist der Stack verwendbar, wenn verschiedene Elemente gleichzeitig aus einer Datenstruktur entfernt und verändert werden müssen. Jedes Element kann dabei in ein anderes Feld abgelegt und seine Adresse im Hauptdatenfeld in einen Stack mit Zwei-Byte-Zahlen gespeichert werden. Wenn die Elemente wieder in das Hauptfeld eingelesen werden, können ihre richtigen Adressen vom Anfang des Stack an entnommen werden.
Ein Stack wird jedoch häufiger gebraucht, um die Adressen einer Gruppe von Elementen mit gleichen Eigenschaften festzuhalten, vor allem Gruppen, bei denen viele Additionen und Subtraktionen vorgenommen werden. Weiter unten in diesem Kapitel werden wir sehen, wie man nach demselben Prinzip Anzahl und Ort von Leerstellen innerhalb eines Feldes festhalten kann. Jedesmal, wenn ein Element aus dem Feld entfernt wird, wird seine Adresse in einen Stack mit Zwei-Byte-Werten abgelegt:

```
100 P1=INT(PP/256):P2=PP-256*P1

110 SK$=CHR$(P1)+CHR$(P2)+SK$
```

wobei PP die Adresse des entfernten Elements ist. Die Adresse einer Leerstelle im Feld erhält man mit:

```
100 PP=256*ASC(SK$)+ASC(MID$(SK$,2)):SK$
=MID$(SK$,2))
```

Auf diese Weise kann man etwas in ein Feld einfügen, ohne alle anderen Werte zu verschieben, damit die Leerstellen ans Ende kommen. Wenn SK$ auf einen leeren String reduziert ist, weist dies darauf hin, daß keine Leerstellen mehr verfügbar sind.

STRINGDATEN-STRUKTUREN

VERDICHTETE STRINGS

Im Kapitel über Stringverarbeitung haben wir schon einige einfachere Möglichkeiten untersucht, wie Strings zum Speichern von Informationen benutzt werden können, und wie mehr auf kleinerem Raum unterzubringen ist. Ein Beispiel dafür war der einfache verdichtete String, in dem ein Trennzeichen zur Identifizierung der einzel-

nen Teile verwendet wird, so daß eine komplette Eintragung in einer Namens- und Adreßdatei etwa in folgender Form gespeichert wird:

```
SCHMID*KARL OTTO*HAUPTSTR.11*MUSTERSTADT
*MUSTERLAND*PF 123*TEL 0909-1122*MAENNL.
```

Eine solche Struktur hat den Vorteil, daß jeder Ausdruck nicht wie ein vollständiges Element eines Stringfeldes drei Bytes zur Verwaltung benötigt, sondern nur ein Byte, nämlich das mit dem *. Damit scheint auf den ersten Blick nicht übermäßig viel eingespart zu sein. Aber für eine Datei, deren Eintragungen wie im obigen Beispiel aus je acht getrennten Ausdrücken bestehen, bedeutet es eine Einsparung von

8*3 (für einen einzelnen Ausdruck jeder Eintragung) −3 + 7 (drei Bytes für den einzelnen String plus sieben Trennzeichen)

oder 14 Bytes bei einer einzigen Eintragung. Im Fall einer Datei mit 500 Eintragungen wäre das eine Einsparung von 7000 Bytes, und das lohnt sich, wenn der gesamte Speicherplatz für Programm und Daten weniger als 39.000 Bytes beträgt. Ein Nachteil dieses Verfahrens der Stringverdichtung ist die Zeit, die für das Suchen eines einzelnen Ausdrucks benötigt wird. Um das obige Beispiel wieder zu zerlegen, brauchen wir eine Suchroutine zum Auffinden der Trennzeichen, wie im Kapitel über Stringverarbeitung beschrieben. Eine Datei mit 500 Eintragungen nach einem bestimmten Ausdruck abzusuchen, kann unangenehm lange dauern, besonders wenn er nahe am Ende eines Strings steht. Ein möglicher Ausweg besteht darin, die Informationen über den Aufbau des Strings in einem Format abzulegen, das leichter zugänglich ist, anstatt nach Sternchen in den Strings zu suchen. Das erreicht man durch den Gebrauch sogenannter 'Zeiger'. Später werden wir untersuchen, wie man Zeiger auf weitaus komplexere und effizientere Weise zum Strukturieren einer Datei einsetzen kann. Hier jedoch besteht die Technik einfach darin, dem String Zahlen vorauszuschicken, die anzeigen, wo die Zeichenkette getrennt werden soll:

```
100 PTR$="":IN$=""
110 FORI=1TO8
120  INPUTTT$
130  IN$=IN$+TT$
140 PTR$=PTR$+CHR$(LEN(IN$))
150 NEXTI
160  IN$=PTR$+IN$
```

Das Programm berechnet lediglich bei der Eingabe jedes der acht Strings die Länge von IN$, also der gesamten Eingabe, und hängt dann an den Zeigerstring PTR$ den ASCII-Werten dieser Länge an. Am Ende der Schleife sind alle Strings in IN$ abgelegt, ohne daß gekennzeichnet ist, wo der eine aufhört und der nächste beginnt, während in PTR$ acht Zeichen abgelegt sind, deren ASCII-Werte anzeigen, wo die einzelnen Datenelemente enden. Beide zusammen ergeben den String, wie er im Speicher abgelegt wird.
Sind die Datenelemente so gespeichert, genügt eine einfache Routine, um sie wieder zu zerlegen:

```
200 P1=9
210 FORI=1TO8
220 P2=ASC(MID$(IN$,I))+8
230 PRINTMID$(IN$,P1,P2-P1+1)
240 P1=P2+1
250 NEXTI
```

Dabei werden die Werte der Zeigerzeichen am Anfang des Strings dazu benutzt, beim Lesen des Strings Anfang und Ende jedes Datenelements zu bestimmen. Zu Beginn der Routine braucht man dazu nur die Anzahl der Zeigerzeichen zu kennen, um die Position des ersten Zeichens des ersten Datenelements (also des ersten Zeichens hinter dem Zeiger) festlegen zu können. Von da an sagt uns jedes Zeigerzeichen, wo ein Datenelement aufhört, und wir wissen, daß das nächste Datenelement ein Zeichen weiter beginnt. Die hier angegebene Routine setzt eine regelmäßige Struktur von acht Datenelementen voraus, aber das ist nicht unbedingt nötig. Man könnte auch eine beliebige Anzahl von Datenelementen bis zur maximalen Stringlänge eingeben und am Anfang von PTR$ ein zusätzliches Zeichen einfügen, das die Anzahl der Datenelemente festhält. Die Schleife, die die Datenelemente wieder isoliert, würde dann mit diesem Zeichen die Anzahl der erwarteten Zeigerzeichen bestimmen.
Dieses Verfahren ist wesentlich schneller, als wenn das Programm den String Zeichen für Zeichen nach Trennzeichen durchsuchen müßte. Der Hauptnachteil liegt darin, daß es die Länge einer einzelnen Eintragung auf 255 Zeichen minus der für die Zeiger benötigten Anzahl von Zeichen beschränkt. Für die meisten Dateien sind jedoch 255 Zeichen mehr als genug; wenn nicht, nimmt man einfach zwei oder mehr verdichtete Strings für jede Eintragung.

VERDICHTETE STRINGS MIT NUMERISCHEN FELDZEIGERN

Die Verwendung verdichteter Strings in der oben beschriebenen Weise ist insofern von Nachteil, als die Berechnung des ASCII-Wertes der einzelnen Zeichen zeitraubend sein kann, wenn viele Datenelemente verarbeitet werden. Noch störender ist, daß die meisten im Zeigerabschnitt des Feldes erzeugten Zeichen nicht druckbar sind. Das ist ohne Belang, wenn sie im Arbeitsspeicher abgelegt sind, aber der C 64 kann solche Zeichen nicht ohne weiteres auf Band oder Diskette abspeichern. Um einen einzigen verdichteten String im obigen Format auf Kassette aufzuzeichnen, brauchte man eine Routine wie folgt:

```
100 FORI=1TO8:PRINT#1,ASC(MID$(IN$,I):NE
XT I
110 PRINT#1,MID$(IN$,9)
```

und um die Daten zurückzubekommen, müßte man diese oder ähnliche Zeilen eingeben:

```
100 IN$="":FORI=1TO8:INPUT#1,TT:IN$=IN$+CH
R$(TT):NEXT I
110 INPUT#1,TT$:IN$=IN$+TT$
```

Das ganze Übersetzen zwischen Zahlen und Zeichen beim Laden und Speichern braucht Zeit, die sich verkürzen läßt, wenn man die Zeiger in einem numerischen Feld ablegt, wie es im Abschnitt über das Ablegen von Zahlen in Strings beschrieben wurde. Das geht zwar nur, wenn die Anzahl der Datenelemente in jedem verdichteten String bekannt und regelmäßig ist, vereinfacht jedoch das Speichern und Laden beträchtlich. Wenn man Zeiger in einem Integerfeld ablegen will, muß man zunächst bestimmen, wie viele Datenelemente man in jedem String unterbringen will, und dann ein Integerfeld dimensionieren, das in jeder Zeile halb soviele Elemente hat (plus eins, wenn die Zahl ungerade ist). Um also eine Datei mit Datensätzen zu je 9 Elementen zu verarbeiten, wäre ein Feld der Form ARRAY%(500,4) zu dimensionieren und die folgende Eingaberoutine zu verwenden:

```
100 DIM ARRAY%(500,4)
105 INPUT X
110 FORI=0TO9
120 INPUT TT$
130 IN$=IN$+TT$
```

```
140 NN=ARRAY%(X,I/2)-65536*(ARRAY%(X,I/2
)<0)
150 IF I AND I THEN NN =256*INT(NN/256)+
LEN(IN$)
160 IF NOT I AND 1 THEN NN=NN-256*INT(NN
/256)+256*LEN(IN$)
170 ARRAY%(X,I/2)=NN+65536*(NN>32767)
180 NEXTI
```

Diese Zeilen sehen eher abschreckend aus, enthalten aber nichts, was uns nicht schon bekannt wäre. Die Routine ist tatsächlich das Gegenstück zur Eingaberoutine aus dem Abschnitt über das Speichern von Zahlen in ein Integerfeld.
Eine solche Struktur zu zerlegen, ist wiederum umständlicher, als einfach Zeichen im String als Zeiger zu benutzen:

```
200 P1=1
210 FORI=0TO9
215 NN=AR%(X,I/2)-65536*(AR%(X,I/2)<0)
220 IF NOT I AND 1 THEN P2=INT(NN/256)
225 IF I AND 1 THEN P2=NN-256*INT(NN/256
)
230 PRINTMID$(IN$,P1,P2-P1+1)
240 P1=P2+1
250 NEXTI
```

Man kann beide Routinen zusammen durchlaufen lassen, um Datenelemente einzufügen und aus einem String abzurufen, weil der Wert von 'X' nicht definiert, also Null ist. Gewöhnlich schreibt ein anderer Teil des Programms die Stelle vor, auf die das neue Datenelement abzulegen ist.
Ob Sie dieses Verfahren überhaupt anwenden wollen, hängt vom Umfang der notwendigen Lade- und Speicherarbeit im Verhältnis zum Umfang der Operationen ab, denen die einzelnen Datenelemente unterworfen werden sollen. Tatsächlich dauert es doppelt so lange, für ein so gespeichertes Datenelement die Zeiger herauszusuchen oder einzusetzen, wie bei dem Verfahren, das in einem String dargestellte Werte verwendet. Es ist trotzdem eine nützliche Bereicherung Ihres Methodensortiments und merkenswert – wenigstens aus Gründen der Abwechslung. Im nächsten Kapitel untersuchen wir weitere Datenstrukturen, die erheblich schwieriger zu programmieren sind, die aber auch sehr viel Zeit und Speicherplatz sparen helfen.

KAPITEL 10

DATENSTRUKTUREN II

Im letzten Kapitel haben wir uns einige nette und einfache Methoden der Datenspeicherung angesehen, mit denen sich verschiedene Probleme lösen lassen. In diesem Kapitel wenden wir uns Datenstrukturen zu, die ebenfalls Schwierigkeiten aus dem Weg schaffen können, aber bei weitem aufwendiger zu programmieren sind.

VERKETTETE LISTEN

Sie haben sicher bemerkt, daß wir bisher bei der Diskussion von Datenstrukturen für Zahlen sehr oft Techniken eingesetzt haben, die das Verschieben von Daten zum Teil einschränken. Um einen Bytewert in die erste Position eines numerischen Feldes einzusetzen, müssen alle vorhandenen Daten im Feld um eine Stelle nach hinten verschoben werden. Wie wir gesehen haben, wird beim Ablegen von Werten in einen String automatisch alles verschoben, wenn ein Element am Anfang hinzugefügt werden soll. Manchmal allerdings können sowohl bei Zahlen als auch bei Strings die Daten praktisch nur in einem Feld wie etwa A$(500) abgelegt werden, wobei jede Eintragung eine Zeile des Feldes benötigt. Vorausgesetzt, die Daten beginnen in Zeile Null des Feldes, macht das Einsetzen eines neuen Wertes nahe dem Anfang das Verschieben großer Datenmengen erforderlich, um den nötigen Platz zu schaffen. Das kann sehr zeitaufwendig sein und außerdem Probleme der Garbage Collection aufwerfen, die wir in Kapitel 3 angesprochen haben. Die Lösung eines solchen Problems besteht oft darin, die Daten in der Reihenfolge der Eingabe zu speichern und getrennt aufzuzeichnen, wo jedes Element sich befände, wenn das Feld die gewünschte Ordnung hätte, z. B. die alphabetische Ordnung bei Strings. Bei einer 'verketteten Liste' ist an jedes Element der Liste die Adresse desjenigen Elements angehängt, das in der gewünschten Ordnung das nächste ist. Nehmen wir zum Beispiel eine Menge von drei Stringelementen: AAA, CCC und DDD. Wenn wir ein neues Element BBB an den alphabetisch richtigen Ort einfügen wollen, brauchen wir nicht CCC und DDD zu verschieben, um Platz dafür zu machen. Es reicht, dem Element AAA irgendwie einen Zeiger anzuhängen, der dem Programm sagt, daß das nächste Element in alphabetischer Reihenfolge nicht Element 2, sondern Element 4 ist. Ist Element 4 gefunden, so muß diesem ein Zeiger angehängt sein, der dem Programm sagt, daß das nächste Element in Position 2 zu finden ist. Wenn jedes Element den richtigen Zeiger besitzt, arbeitet das Programm die Elemente in der Reihenfolge 1, 4, 2, 3 ab.

Um die Methode zu veranschaulichen, die wir verwenden wollen, schneiden Sie sechs kleine Quadrate von etwa 5 cm Seitenlänger aus Papier aus und ziehen Sie quer über jedes eine Linie. Die obere Hälfte jedes Quadrats soll den Zeiger auf das nächste Element der Liste enthalten, die untere jeweils das zu speichernde Element. Schreiben Sie auf einen der Zettel '1' auf die obere Hälfte (ziemlich klein, weil die Zahl noch geändert werden soll) und 'START' auf die untere. Auf einen anderen Zettel schreiben Sie oben '65535' und unten 'STOP'. Legen Sie diese Zettel nebeneinander und lassen Sie daneben Platz, um die vier anderen Zettel in einer Reihe abzulegen. Von jetzt an fügen wir in alphabetischer Reihenfolge Zettel ein, wobei START jedem neuen Zettel im Alphabet vorangehen und STOP ihm folgen soll.

Nehmen Sie jetzt einen anderen Zettel und schreiben Sie 'BBB' auf die untere Hälfte. Nach der obigen Regel kommt BBB nach START, also sehen Sie sich die obere Hälfte von START an und gehen zum angezeigten Zettel (sie sind von null an numeriert). Der angezeigte Zettel ist offensichtlich STOP, und BBB sollte vorher kommen, also haben wir die richtige Position für BBB gefunden: unmittelbar nach START und vor STOP.

Jetzt kommt das Entscheidende. Verschieben Sie nicht die beiden vorhandenen Zettel, sondern legen Sie den Zettel BBB in der dritten Position ab (Position 2, wenn von null an gezählt wird). Streichen Sie die '1' in der oberen Hälfte von START und ersetzen Sie sie durch '2'. START zeigt nun auf den neuen Zettel BBB. Schreiben Sie '1' oben auf den neuen Zettel, denn der nächste Zettel in alphabetischer Reihenfolge ist STOP. Wenn Sie jetzt den Zeigern oben auf den Zetteln von START an folgen, erhalten Sie die Reihenfolge START, BBB, STOP.

Nehmen Sie nun einen weiteren Zettel und tragen unten 'DDD' ein. Folgen Sie den Zeigern so weit, bis ein Zettel kleiner als DDD ist, aber der nächste größer. In unserem Fall sind Sie dann hinter BBB und vor STOP. Legen Sie DDD in die vierte Position. Streichen Sie die '1' auf BBB und ersetzen sie durch '3'. Schreiben Sie '1' auf DDD, um anzuzeigen, daß der nächste Zettel STOP ist. Die Zeiger ergeben nun die Reihenfolge START, BBB, DDD, STOP.

Mit den übrigen beiden Zetteln müßten Sie jetzt selbst zurechtkommen. Einer wird 'AAA', der andere 'CCC'. Am Ende des Verfahrens sollten die Zeiger die Reihenfolge START, AAA, BBB, CCC, DDD, STOP ergeben. Beachten Sie, daß dies nichts mit der Reihenfolge der Zettel auf dem Tisch zu tun hat: Die Reihenfolge kommt nur über die Zeiger zustande, und zwar ohne je etwas verschieben zu müssen, das schon abgelegt ist. Wenn Sie das verstanden haben, können Sie nachvollziehen, wie dieses Verfahren in BASIC durchgeführt wird. Dabei arbeiten wir mit Strings in einem Feld und verwenden ein paralleles Integerfeld, um die Zeiger abzulegen.

Die nächste Routine erzeugt eine verkettete Liste in alphabetischer Reihenfolge:

```
1000 REM******************
1001 REM VERKETTETE LISTE
1002 REM******************
1100 DIM A$(499), A%(499):IT=2:HO=0
1110 A$(0)=CHR$(0)
1120 A$(1)=CHR$(255)
1130 A%(0)=1
1140 A%(1)=32767
1150 INPUT"WORT EINGEBEN";IN$:IFIN$="END
E"THENSTOP
1160 ADD=0
1170 FORI=1TOIT-1
1180 TT=ADD
1190 ADD=A%(ADD)
1200 IFA$(ADD)<IN$THENNEXTI
1210 A%(TT)=IT
1220 A$(IT)=IN$:A%(IT)=ADD
1230 IT=IT+1
1240 GOTO1150
```

Da zu dieser Routine einige Erklärungen nötig sind, gehen wir sie Zeile für Zeile durch:

1100 A$ ist das Hauptfeld zum Speichern der Daten, in A% werden die Zeiger abgelegt, und IT bedeutet die Anzahl der schon gespeicherten Elemente. IT wird zunächst auf 2 gesetzt, weil das Feld zwei Hilfselemente enthält, die Anfang und Ende der verketteten Liste markieren. Die Variable HO wird später erklärt.

1100–1140 Das sind die beiden Hilfselemente, also START und STOP in unserem Beispiel. Der ersten Position (Adresse 0) des Feldes wird ein Element zugewiesen, dessen Zeiger auf die zweite Position (Adresse 1) zeigt und das den Inhalt CHR$(0) hat. Das bedeutet, bei allen gewöhnlichen Strings ist dieses Element immer das alphabetisch erste des Feldes. In der zweiten Position des Feldes steht ein Element, dessen Zeiger-Komponente auf 32767 zeigt (dieser Wert wird nicht gebraucht, da das letzte Element auf nichts zu zeigen hat), und dessen Daten-Komponente ein einzelnes Byte mit dem Wert 255 ist. Dieses Element wird immer das letzte der Datei in alphabetischer Reihenfolge sein. Diese beiden Eingaben haben den Grund, daß es wie bei vielen anderen Datenstrukturen einfacher ist, mit einer funktionsfähigen Datei anzufangen, als mit Hilfe besonderer Abfragen in der Routine zu testen, ob ein neues Element an den Anfang oder das Ende der Liste gehört. Das erste und

das letzte Element der Datei müssen anders behandelt werden, weil auf das erste von keinem Element aus gezeigt wird und das letzte auf kein Element zeigt. Bei vielen Methoden der Datenspeicherung gibt es Probleme mit ersten und letzten Elementen, und häufig ist es einfacher, das Problem zu umgehen und eine künstliche erste und letzte Zeile einzufügen, wenn das Feld eingerichtet ist.

1160 In der Variablen ADD wird die Position des gerade untersuchten Elements im Feld gespeichert; sie zählt von null an.

1170 Die maximale Anzahl an Vergleichen zwischen dem eingegebenen Element und den vorhandenen Elementen ist gleich der Anzahl der Elemente im Feld.

1180 TT ist normalerweise die Position des bei der alphabetischen Suche zuletzt geprüften Elements. Wir müssen sie uns für den Fall merken, daß das neue Element zwischen dieses und das nächste eingefügt werden muß, weil dann der Zeiger des Elements in TT geändert werden muß.

1190 In ADD wird nun die Position abgelegt, auf die der Zeiger des Elements in TT zeigt, nämlich auf das nächste Element in der verketteten Liste.

1200 Falls das aktuelle Element in ADD nicht größer als das neue Element ist, haben wir nicht die richtige Stelle gefunden, also bringt uns die Schleife den Zeigern folgend zum nächsten Element.

1210 Wenn wir hier angekommen sind, ist die korrekte Position für das neue Element erreicht. Der Zeiger des Elements in TT muß jetzt so geändert werden, daß er auf die Position zeigt, in die ein neues Element eingegeben wird, d. h. Position IT.

1220 Das neue Element wird eingegeben. Sein Zeiger zeigt auf das Element, auf das vorher vom Element in TT gezeigt wurde.

1230 IT wird um eins erhöht, um festzuhalten, daß jetzt ein weiteres Element vorhanden ist.

Um zu überprüfen, ob die Routine ordnungsgemäß funktioniert, geben Sie folgende Zeilen ein, die die Liste in alphabetischer Reihenfolge ausdrucken:

```
1152 IFIN$="ANZEIGE"THENGOSUB2000:GOTO11
50
2000 REM***************
2001 REM ANZEIGE LISTE
2002 REM***************
2010 ADD=0:FORI=1TO IT-HO-2
2020 PRINTA$(A%(ADD)):ADD=A%(ADD)
2030 NEXTI:RETURN
```

Hierbei dient das Zeigerbyte jedes Elements zum Auffinden der Adresse des nächsten. Wieder wird die Variable HO verwendet – sie wird nach der nächsten Routine erklärt.
Gelöscht wird nach einem ähnlichen System. Diese Routine löscht ein numerisches Element aus der Liste – genauso einfach wäre es, den String anzugeben und vor dem Löschen in der Liste danach zu suchen.

```
1154 IFIN$="LOESCHEN"THENGOSUB3000:GOTO1
150
3000 REM***************
3001 REM LOESCHEN WORT
3002 REM***************
3010 INPUT"ZAHL DES ZU LOESCHENDEN WORTE
S";NN
3020 P2=0:FORI=0TONN-1
3030 P1=P2:P2=A%(P2)
3040 NEXTI
3050 A%(P1)=A%(P2):HO=HO+1:RETURN
```

Beachten Sie, daß nichts wirklich gelöscht wird; es wird nur der Zeiger des Elements vor dem zu löschenden gleich dem Zeiger des zu löschenden Elements gesetzt. Das Element vor dem zu löschenden zeigt dann auf das Element nach dem zu löschenden, und das Programm weiß nichts mehr vom gelöschten Element, obwohl es noch immer vorhanden ist. Wenn Sie es wirklich löschen wollen, müssen Sie eine Zeile hinzufügen:

```
3045 A$(A%(P1))=""
```

In dieser Routine wird die Variable HO jedesmal um eins erhöht, wenn eine Element gelöscht ist. Das mag seltsam erscheinen, weil es naheliegender wäre, einfach den Wert von IT um eins zu vermindern. Das Problem dabei ist aber, daß neue Elemente stets am Ende der Liste eingefügt werden und IT das Ende der Liste anzeigt. Weil wir nicht wirklich beim Löschen oder Einfügen die Adresse irgendeines Elements ändern, würde ein Verkleinern von IT bedeuten, daß ein neues Element ein schon vorhandenes überschreibt. Weiter unten in diesem Kapitel werden wir untersuchen, was dagegen zu tun ist, daß auf diese Weise platzverschwendende 'Löcher' im Feld zurückbleiben.
Verkettete Listen lassen sich mit Gewinn in Programmen einsetzen, in denen regelmäßig Listen erzeugt werden müssen. Sie haben jedoch den Nachteil, daß nur von den Enden her auf sie zugegriffen werden kann, in diesem Fall nur von einem Ende her, dem Listenkopf. Man kann auch Listen erzeugen, die vorwärts und rückwärts verkettet sind, indem man zwei Zeiger verwendet, von denen einer auf das vorhergehende Element und der andere auf das dahinter zeigt. Aber auch dann kann man nur am Anfang und am Ende vernünftig starten. Springt man in eine verkettete Liste, ist es unmöglich zu entscheiden, an welcher Stelle innerhalb der Liste man sich befindet. Man kennt nur die Adresse eines benachbarten Elements. Anders gesagt: Wenn man das 30. Element einer verketteten Liste erreichen will, kann man nicht unmittelbar darauf zugreifen, sondern muß dem beschwerlichen Weg der 29 vorausgehenden Zeiger folgen. Im nächsten Abschnitt untersuchen wir einen Ausweg aus diesem Dilemma.

ZEIGERSTRINGS

Wie wir im letzten Abschnitt gesehen haben, kann man eine geordnete Liste erzeugen, ohne ständig Elemente in einem Feld umherschieben zu müssen. Elemente können einfach an das Ende eines Feldes angehängt werden, wobei einem getrennten Zeigerfeld die Aufgabe überlassen bleibt, ihre richtige Position zu bestimmen. Dabei ergab sich das Problem, daß man nicht in die Liste springen konnte, wenn man z. B. die Position des 30. Elements bestimmen wollte. Wir haben jedoch im vorigen Kapitel und in Kapitel 3 eine Art der Datenstruktur kennengelernt, die Einfügungen von Elementen in ihre richtige Position erlaubt, ohne die vorhandenen Elemente zu verschieben: den String. Wir wissen auch, daß man Zahlen in Strings speichern kann, und haben mit Verfahren experimentiert, die genau das ermöglichen, was mit einer verketteten Liste nicht geht, nämlich auf jede beliebige Position im String zuzugreifen. Wenn wir dazu berücksichtigen, was wir über Zeiger gelernt haben, sind wir beim Zeigerstring – einer Methode, mit der wir mitten in ein Feld springen können und dennoch den Vorteil behalten, neue Elemente einfügen

zu können, ohne erst Platz durch Verschieben des bestehenden Listeninhalts schaffen zu müssen.
Dazu brauchen wir nur eine neue Technik, um Zahlen im Bereich 0–65535 in ein Feld von Strings einzusetzen, deren Gesamtlänge variabel ist. Es ähnelt einem Verfahren aus Kapitel 3 und läßt sich z. B. mit der nachstehenden Routine realisieren:

```
4000 REM*****************
4001 REM INITIALISIERUNG
4002 REM*****************
4010 DIM PTR$(19):IT=0:HO=0
4020 FORI=0TO1:FORJ=1TO125:PTR$(I)=PTR$(
I)+"01":NEXTJ,I:IT=250
5000 REM**************
5001 REM EINGABE TERM
5002 REM**************
5010 INPUT"ZAHL (1-32767) ";NN
5020 NN$=CHR$(NN/256)+CHR$(NN-256*INT(NN
/256))
5030 GOSUB8000
5040 PTR$(LL)=LEFT$(PTR$(LL),2*LP-2)+NN$
+MID$(PTR$(LL),2*LP-1)
5050 IT=IT+1
5060 GOSUB9000
5070 GOTO5010
8000 REM***********************
8001 REM POSITIONS BESTIMMUNG
8002 REM***********************
8010 PRINT"POSITION (1 BIS";IT+1+(NN<=0)
;") ";:INPUTPP
8020 LL=INT((PP-1)/125):LP=PP-125*LL
8030 RETURN
9000 REM*********************
9001 REM SETZEN DES STRINGS
9002 REM*********************
9010 FORI=0TO18
9020 IFLEN(PTR$(I))<=250THEN9050
9030 PTR$(I+1)=RIGHT$(PTR$(I),LEN(PTR$(I
))-2)
```

```
9040 PTR$(I)=LEFT$(PTR$(I),LEN(PTR$(I))-
2)
9050 NEXTI
9060 RETURN
```

Beachten Sie, daß Zeile 4020 vorläufig ist und nur die Routine testen soll.
Lassen Sie die Routine durchlaufen, und geben Sie auf Abfrage folgende Werte als Zahl und Position ein:

```
16705
251

16962
252

17219
253

16962
1

16705
1

16962
126

16705
126
```

Wenn Sie nun das Programm mit STOP/RESTORE stoppen, können Sie die Funktionsfähigkeit der Routine prüfen, indem Sie direkt, d. h. ohne Zeilennummer

```
?PTR$(0)
```

eingeben, worauf ein 01-String ausgegeben werden müßte, dem AABB vorausgeht, das heißt die als Zwei-Byte-Zeichenpaare dargestellten Zahlen 16705 und 16962. Die Eingabe

```
?PTR$(1)
```

sollte dasselbe liefern.
Der eigentliche Test besteht jetzt in der Eingabe

```
?PTR$(2)
```

die den String 01010101AABBCC ergeben müßte. Ein Abzählen der Zeichen in den ersten beiden Strings müßte sechs volle Zeilen plus 10 weitere Zeichen auf dem Bildschirm erkennen lassen. Wenn all das zutrifft, kann die Routine Zwei-Byte-Zahlen über die volle Länge der Daten aufnehmen und die übrigen Daten automatisch so verschieben, daß kein einziger String länger ist als 250 Bytes.

```
6000 REM**********************
6001 REM LOESCHEN VOM STRING
6002 REM**********************
6010 PRINT"LOESCHEN";:GOSUB8000
6020 PTR$(LL)=LEFT$(PTR$(LL),2*LP-2)+MID
$(PTR$(LL),2*LP+1)
6030 GOSUB9500
6040 RETURN
9500 REM********************************
9501 REM SETZEN DES STRING ZUM LOESCHEN
9502 REM********************************
9510 FORI=0TO18
9520 IF LEN(PTR$(I))>=250OR LEN(PTR$(I+1
))=0 THEN 9550
9530 PTR$(I)=PTR$(I)+LEFT$(PTR$(I+1),2)
9540 PTR$(I+1)=MID$(PTR$(I+1),3)
9550 NEXTI
9560 RETURN
```

Zum Löschen von Elementen brauchen wir nur noch eine leicht überarbeitete Version der Einfüge-Routine anzubinden, sowie eine weitere Zeile, mit der zu diesem zweiten Programmteil gesprungen wird:

```
5015 IF NNK=0THENGOSUB6000:GOTO 5010
```

Lassen Sie jetzt die ganze Routine durchlaufen und geben Sie ein:

```
16705
1
```

```
16705
126

16705
251
```

Das setzt AA an den Anfang der ersten drei Elemente des Stringfelds. Auf die nächste Abfrage nach einer Zahl geben Sie 0 ein, und wenn Sie nach der zu löschenden Zahl gefragt werden, antworten Sie mit '1'. Beim Stoppen des Programms mit STOP/RESTORE sollten Sie feststellen, daß PTR$(0) und PTR$(1) einen 01-String mit AA am Ende enthalten und PTR$(2) den String 0101. Demnach kann sowohl eingefügt als auch gelöscht werden, wobei jeweils die Länge des Strings angepaßt wird. Zeile 4020 kann jetzt gestrichen werden.
Wir wissen jetzt genug, um mit einem Zeigerstring umgehen zu können: Wir brauchen ihm jetzt nur noch etwas zu geben, auf das er zeigen kann. In der nächsten Routine werden Strings in alphabetischer Reihenfolge in ein Feld eingefügt. Der Trick dabei besteht darin, die Strings nicht einfach von Position Null bis zum letzten Element durchzugehen, sondern sie bei jeder neuen Eingabe in der durch den Zeigerstring vorgeschriebenen Reihenfolge zu untersuchen. Hier ist die Routine:

```
4000 REM*****************
4001 REM INITIALISIERUNG
4002 REM*****************
4010 DIM PTR$(19),A$(499):IT=2:HO=0
4020 A$(0)=CHR$(0)
4030 A$(1)=CHR$(255)
4040 PTR$(0)=CHR$(0)+CHR$(0)+CHR$(0)+CHR
$(1)
5000 REM**************
5001 REM EINGABE WORT
5002 REM**************
5010 INPUT"BITTE WORT EINGEBEN ";IN$
5014 IF IN$="ENDE"THENSTOP
5016 IF IN$="ANZEIGE"THENGOSUB7000:GOTO5
010
5020 GOSUB8000
5030 NN$=CHR$(IT/256)+CHR$(IT-256*INT(IT
/256))
5040 PTR$(LL)=LEFT$(PTR$(LL),2*LP-2)+NN$
```

```
+MID$(PTR$(LL),2*LP-1)
5050 A$(IT)=IN$
5060 GOSUB9000
5070 IT=IT+1
5080 GOTO5010
7000 REM****************
7001 REM LISTE ANZEIGEN
7002 REM****************
7005 IFIT-HO=2THENRETURN
7010 FORPP=2TOIT-HO-1
7020 LL=INT((PP-1)/125):LP=PP-125*LL
7030 PA=256*ASC(MID$(PTR$(LL),2*LP-1))+A
SC(MID$(PTR$(LL),2*LP))
7040 PRINTA$(PA)
7050 NEXTPP
7060 RETURN
8000 REM******************
8001 REM POSITION SETZEN
8002 REM******************
8010 FORPP=1TOIT-HO
8020 LL=INT((PP-1)/125):LP=PP-125*LL
8030 PA=256*ASC(MID$(PTR$(LL),2*LP-1))+A
SC(MID$(PTR$(LL),2*LP))
8040 IFA$(PA)<IN$THENNEXTPP
8050 RETURN
9000 REM********************
9001 REM SETZEN DES STRING
9002 REM********************
9010 FORI=0TO18
9020 IF LEN(PTR$(I))<=250THEN9050
9030 PTR$(I+1)=RIGHT$(PTR$(I),2)+PTR$(I+
1)
9040 PTR$(I)=LEFT$(PTR$(I),LEN(PTR$(I))-
2)
9050 NEXTI
9060 RETURN
```

Ab 8000 liest die Routine das Stringfeld in der vom Zeigerfeld vorgeschriebenen Reihenfolge, wobei die richtige Adresse jedes Elements im Feld A$ in der Variablen PA abgelegt wird. Wenn die richtige Position endeckt ist, ist sie schon in den

Variablen PP, LL und LP gespeichert. Diese werden an die Routine bei 5000 übergeben, die schon darauf eingerichtet ist, die Position zu übernehmen, die durch diese drei Werte definiert wird. Bei 5000 hat die Routine eine einzige wichtige Änderung: NN$ – also die beiden Zeichen, in denen der neue Zeiger bis zu seinem Einsatz abgelegt wird – ist eine Übersetzung von IT, das sowohl die Anzahl der bisher gespeicherten Elemente als auch die Nummer des ersten freien Elements im Feld A$ darstellt.
Wie die beiden Routinen zusammen arbeiten, kann man testen, indem man auf Abfrage PRINT eingibt. Daraufhin wird die Routine ab Zeile 7000 aufgerufen, die ganze Liste in der Reihenfolge der in PTR$ enthaltenen Werte auszudrucken.

LÖSCHEN MIT ZEIGERN

Nachdem wir die bei Einfügungen entstehenden Probleme bewältigt haben, können wir uns dem Löschen unter Verwendung eines Zeigerfelds zuwenden. Wir haben schon eine Routine zum Löschen einzelner Elemente aus einem Zeigerfeld. Diese wollen wir so modifizieren, daß der Benutzer nicht mehr aufgefordert wird, die Nummer des Elements anzugeben, sondern das Element selbst eingeben kann, worauf das Feld danach durchsucht wird und das Element und der Zeiger gelöscht werden. Fügen Sie folgende Zeilen an die oben abgedruckte Routine an:

```
5012 IFIN$="LOESCHEN"THENGOSUB6000:GOTO5
010
6000 REM***************
6001 REM WORT LOESCHEN
6002 REM***************
6010 INPUT"ZU LOESCHENDES WORT ";IN$:GOS
UB8000
6020 IFIN$<>A$(PA)THENRETURN
6030 PTR$(LL)=LEFT$(PTR$(LL),2*LP-2)+MID
$(PTR$(LL),2*LP+1):GOSUB9500
6040 HO=HO+1:A$(PA)="":RETURN
9500 REM*********************************
9501 REM SETZEN D. STRING ZUM LOESCHEN
9502 REM*********************************
9510 FORI=0TO18
9520 IFLEN(PTR$(I))>=250ORLEN(PTR$(I+1))
=0THEN9540
9530 PTR$(I)=PTR$(I)+LEFT$(PTR$(I+1),2):
```

```
PTR$(I+1)=MID$(PTR$(I+1),3)
9540 NEXTI:RETURN
```

Zum Auffinden der richtigen Stelle wird dasselbe Verfahren wie beim Einfügen eines Elements angewendet: Die Liste wird in der durch den Zeigerstring vorgeschriebenen Reihenfolge nach dem ersten Element durchsucht, das alphabetisch größer als das neue Element ist. Nach dem Rücksprung aus dieser Suchroutine wird das gefundene Element mit dem zu löschenden verglichen und nur dann gelöscht, wenn beide gleich sind. Wenn das vorgegebene Element gefunden ist, ist seine Position schon in der Variablen PA und seine Zeigerposition in LL und LP gespeichert.
Sie sollten jetzt in der Lage sein, eingegebene Elemente zu löschen.

SCHWARZE LÖCHER

Es bleibt ein letztes Problem zu lösen, nämlich was mit den beim Löschen von Elementen im Feld zurückbleibenden Lücken zu tun ist. Wenn wir nichts dagegen tun, wird das Feld allmählich mit solchen Lücken durchsiebt, bis sie so überhandnehmen, daß kein Platz mehr für Daten bleibt, obwohl das Feld womöglich fast leer ist. Dem kann man mit einer schon angedeuteten Methode abhelfen, und zwar mit dem Einsatz eines Stacks. In unsere Zeigerfeld-Routine können wir eine neue Zeile einfügen:

```
6065 ES$=CHR$(PA/256)+CHR$(PA-256*INT(PA
/256))+ES$
```

Auf diese Weise wird jede erzeugte Lücke in ES$ aufgezeichnet, und wir müssen dann nur noch die Einfüge-Routine überreden, davon Kenntnis zu nehmen. Dazu werden einige Zeilen der obigen Routine geändert:

```
5022 TT=IT:IFLEN(ES$)=0THEN5030
5024 TT=256*ASC(ES$)+ASC(MID$(ES$,2))
5026 ES$=MID$(ES$,3)
5028 IT=IT-1:HO=HO-1
5030 NN$=CHR$(TT/256)+CHR$(TT-256*INT(TT
/256))
5050 A$(TT)=IN$
```

Nachdem diese Zeilen eingegeben sind, wird jedes neu eingegebene Element immer in die erste freie Stelle des Feldes abgelegt. Erst wenn keine Lücken mehr da sind, wird das Element hinten angehängt.

SCHLUSS

Wenn Sie dieses und das vorige Kapitel durchgearbeitet haben, ist Ihnen sicher klar, daß Sie sich nicht etwa morgen hinsetzen und alle diese Methoden in Ihrem nächsten Programm anwenden sollen. Die hier beschriebenen Datenstrukturen sollen Sie an die vielen Möglichkeiten erinnern, den Speicher Ihres C 64 zu nutzen: Alle haben ihre besonderen Stärken und Schwächen, und alle können bei der einen oder anderen Gelegenheit zweckmäßig sein. Die Beschäftigung mit ernsthaftem Programmieren wird es wahrscheinlich mehr als einmal mit sich bringen, daß ein bestimmter Datensatz – vielleicht nur 10 oder 20 irgendwo in einem Programm benutzte Elemente – Sie in Verlegenheit bringt. Dann ist es an der Zeit, sich dieses Kapitel noch einmal vorzunehmen und hoffentlich in einem der vielen hier dargestellten Verfahren die Antwort zu finden, die Sie die ganze Zeit über gesucht haben.

KAPITEL 11

EINFÜGEN VON DATEN

Um nicht zuviel auf einmal zu erklären, haben wir bisher die Frage ignoriert, auf welche Weise man neue Daten am schnellsten in ein Feld einfügt und gezielt die richtige Stelle dafür findet. In diesem Kapitel untersuchen wir, wie große Datenfelder verschoben werden, wie schnelle Recherchen durchgeführt werden, und wie man diese mit den Zeigerfeld-Techniken aus dem vorigen Kapitel verbindet.

NORMALES SUCHEN UND VERSCHIEBEN

Die programmiertechnisch weitaus einfachste Methode, ein Element in ein geordnetes Feld einzufügen, besteht darin, das Feld Element für Element zu durchsuchen und alle Elemente im Anschluß an den gesuchten Ort eine Stelle weiter zu schieben. Nach Ansicht vieler ist es am günstigsten, vom Ende des Feldes her zu suchen, jedes Element mit dem einzufügenden zu vergleichen und es zu verschieben, wenn es sich als eines von denen herausstellt, die verschoben werden müßten, um weiter vorn im Feld Platz für das neue Element zu schaffen. Ob das richtig ist oder nicht, hängt davon ab, ob die zur Suche nach der richtigen Position eines Elements benötigten Zeilen mit denen kombiniert werden können, die die Daten verschieben, damit das neue Element aufgenommen werden kann.
Das nächste Listung gehört zu einer einfachen Routine, die erst ein geordnetes Zahlenfeld initialisiert, dann das Feld nach der richtigen Stelle zum Einfügen eines neuen Elements durchsucht und dann durch Verschieben von Daten dafür Platz macht:

```
500 REM******************
501 REM KONTROLL ROUTINE
502 REM******************
510 GOSUB 750
520 TI$="000000"
530 GOSUB1000
540 GOSUB1500
550 PRINTTI$
560 STOP
750 REM******************
751 REM INITIALISIERUNG
752 REM******************
```

```
760 DIM A%(9999):IT=9950
770 FORI=0TOIT-1:A%(I)=I:NEXT
780 INPUT"NEUER WERT ";NI
790 RETURN
1000 REM*******
1001 REM SUCHE
1002 REM*******
1030 IF IT=0 THEN1070
1040 FORI=0TOIT-1
1050 IFA%(I)>=NI THEN SP=I:I=IT-1
1060 NEXTI
1070 RETURN
1500 REM***********
1501 REM EINSETZEN
1502 REM***********
1505 IF IT=0 THEN1540
1510 FORI=ITTOSP+1 STEP-1
1520 A%(I)=A%(I-1)
1530 NEXTI
1540 A%(SP)=NI
1550 RETURN
```

Diese allgemein gehaltene Routine kann auch auf andere Datensätze angewandt werden. Aus diesem Grund benutzen wir IT, um festzuhalten, wie viele Elemente vorhanden sind, wenn neue eingefügt werden. Es sei ausdrücklich darauf hingewiesen, daß alle derartigen Routinen den Sonderfall eines Felds berücksichtigen müssen, das keine Elemente enthält. Leider wird die Schleife 'FOR I=0 TO IT−1' selbst dann einmal ausgeführt, wenn sie tatsächlich 'FOR I=0 TO −1' bedeutet und deshalb logischerweise nicht ausgeführt werden sollte. Weil das Abarbeiten der Schleife mit diesen Werten einen ILLEGAL QUANTITY ERROR zur Folge hat, muß sie übersprungen werden.
Um die Routine zu testen, geben Sie auf Abfrage nach einem neuen Element 5000 ein. Die Ausgabe von TI$ soll zeigen, daß diese Routine in 81 Sekunden abgearbeitet wird (TI$="000121"). Durchsucht man jedoch das Feld vom anderen Ende her, so kann man auf eine der Schleifen verzichten, wie in der nächsten Routine, die in die vorige eingefügt werden sollte:

```
560 TI$="000000"
570 GOSUB2000
580 PRINTTI$
```

```
590 STOP
2000 REM**********************
2001 REM SUCHEN UND EINSETZEN
2002 REM**********************
2010 IFIT=0THENSP=0:GOTO2060
2020 FORI=IT-1TO0STEP-1
2030 IFNI>A%(I)THENSP=I:I=0:GOTO2050
2040 A%(I+1)=A%(I)
2050 NEXTI
2060 A%(SP+1)=NI
2070 IT=IT+1
2080 RETURN
```

Diese Routine läuft nur 73 Sekunden, wenn sie durch Einfügen von 5000 getestet wird (TI$="000113") – eine Ersparnis von ungefähr 10% gegenüber der letzten, wie Sie selbst feststellen können, wenn Sie beide zusammen durchlaufen lassen. Sie bekämen ganz andere Ergebnisse, wenn das einzufügende Element nahe am Anfang oder Ende des Felds wäre. Durch Einfügen eines Elements in der Mitte erhalten wir jedoch die durchschnittliche Zeit für die Eingabe einer Reihe von Elementen, die nicht zum Anfang oder Ende des Feldes hin gewichtet sind.
Die zweite Methode hat den Nachteil, daß zwei verschiedene Programmfunktionen zusammen laufen: Suchen und Einfügen. Im vorliegenden Fall macht das überhaupt nichts, aber wenn wir ein schnelleres Einfüge- oder Suchverfahren finden, können wir es nicht einfach in das Programm einflicken. Wir haben schon im Kapitel über den Gebrauch von Integerfeldern schnellere Einfügemethoden kennengelernt. Es gibt aber auch eine viel schnellere Suchmethode.

BINÄRES SUCHEN

Offensichtlich sind bei den letzten beiden Routinen ungefähr 5000 Vergleiche zur Bestimmung der richtigen Position erforderlich. Natürlich wird das Feld nicht immer auf sämtlichen Stellen sinnvolle Daten enthalten. Dennoch ist die Anzahl der Vergleiche durchschnittlich immer gleich der Hälfte der Anzahl der Feldelemente. Tatsächlich ist das völlig unnötig, denn zum Auffinden der richtigen Stellen im o. g. Feld sind höchstens 13 Vergleiche erforderlich. Betrachten Sie das folgende Beispiel:

1. Wir wollen ein Element in das obige Feld einfügen, und zwar die Zahl 6172.

2. Wir beginnen die Suche nach der richtigen Position, indem wir diejenige Position

im Feld untersuchen, die die höchste in die Gesamtzahl der Feldelemente passende Potenz von 2 darstellt. In diesem Fall haben wir 10 000 Elemente, und die höchste in diese Zahl passende Potenz von 2 ist 8192. Wir nennen diese Zahl den 'Schrittwert' oder SV (step value). Den ersten Vergleich stellen wir zwischen dem neuen Element und dem schon in Position SV des Felds befindlichen Element an. Die Suchposition nennen wir SP, und zu Beginn bekommt sie den Wert 8191, um zu berücksichtigen, daß das Feld bei Null, nicht bei 1 anfängt.

3. Der Wert in Position 8191 (SP) ist größer als das neue Element 6172, also nehmen wir den ursprünglichen Schrittwert 8192, dividieren ihn durch 2 und erhalten so ein neues SV mit dem Wert 4096. Dieser wird von 8191 (SP) subtrahiert und ergibt das neue SP 4095.

4. Nun wird ein neuer Vergleich bei 4095 (SP) durchgeführt. Die Zahl in dieser Position des Feldes ist kleiner als das neue, also dividieren wir SV durch 2 (=2048), addieren es diesmal zu SP und erhalten das neue SP 6143.

5. Wieder ist das Element in 6143 kleiner als das neue Element, also dividieren wir SV durch 2 (=1024). Das wird zu SP addiert und liefert 7167.

6. Das Element in 7161 ist größer als 6172, also wird SV durch 2 dividiert (=512) und von SP subtrahiert. Das Ergebnis ist 6655.

7. Die Suche wird an folgenden Stellen und mit folgenden Sprüngen fortgesetzt:

6655 (−256)
6399 (−128)
6271 (−64)
6207 (−32)
6175 (−16)
6159 (+8)
6167 (+4)
6171 (+2)
6173 (−1)
6172, die richtige Position.

Wie Sie sehen, haben wir nichts weiter getan, als abnehmende Potenzen von Zwei zu addieren oder zu subtrahieren, je nachdem ob die Zielposition weiter oben oder weiter unten im Feld war. Der Erfolg hängt nicht davon ab, ob das Feld wie in diesem Fall aus jeweils um 1 wachsenden Zahlen besteht. Man braucht nur ein beliebiges String- oder Zahlenfeld, das regelmäßig aufsteigend oder absteigend geordnet ist.

Zum Einfügen eines neuen Elements in ein Feld ist die Anzahl der notwendigen Vergleiche im allgemeinen:

```
INT(LOG(IT)/LOG(2))+1
```

Diese Formel gibt an, wie viele Stellen die binär dargestellte Anzahl IT der Elemente hat, plus 1, und das ist immer die maximale Anzahl der erforderlichen Vergleiche. Also:

```
INT(LOG(500)/LOG(2))+1  =9
```

und 500 ist binär:

111110100 – neun Stellen lang.

Dieses Suchverfahren bietet offensichtlich eine deutliche Einsparung gegenüber dem im vorigen Abschnitt verwendeten: Es werden 4087 Vergleiche eingespart. Wenn wir nach einem Element in einem Feld statt nach einer Position zum Einfügen suchen, kann die Einsparung sogar noch größer sein. Beim Suchen nach einem nicht vorhandenen Element müßte man sämtliche 10 000 Elemente durchgehen – beim binären Suchen würden auch dann 13 Vergleiche reichen, und wenn das gesuchte Element auch dann nicht gefunden wäre, wüßten wir, daß es nicht im Feld ist. Im nächsten Listing wird mit dem binären Suchverfahren wie bei den vorigen Listings ein Element bei 5000 eingefügt:

```
590 TI$="000000"
600 GOSUB3000
610 GOSUB4000
620 PRINTTI$
630 STOP
3000 REM**************
3001 REM BINAER SUCHE
3002 REM**************
3010 IF IT=0THENSP=0:GOTO3090
3020 POWER=INT(LOG(IT)/LOG(2))
3030 SP=2↑POWER-1
3040 FORSV=POWER-1TO0STEP-1
3050 SP=SP+2↑SV*((NI(A%(SP))-(NI>A%(SP))
)
3060 IFSP>IT-1THENSP=IT-1
3070 NEXTSV
```

```
3080 IFA%(SP)<NITHENSP=SP+1
3090 RETURN
4000 REM**********
4001 REM EINSETZEN
4002 REM**********
4010 IFIT=0THEN4050
4020 FORI=ITTOSP+1STEP-1
4030 A%(I)=A%(I-1)
4040 NEXTI
4050 A%(SP)=NI
4060 IT=IT+1
4070 RETURN
```

Es gibt hier zwei kleine Abweichungen von der vorausgehenden Beschreibung des Verfahrens. Erstens kann der Suchzeiger (SP) über die Anzahl IT−1 der Feldelemente springen. Wenn das geschieht, setzt Zeile 3060 den Suchzeiger auf das letzte Feldelement zurück. Diese Wertänderung beeinflußt die Suchoperation nicht. Zweitem: Wenn das eingegebene Element sich nicht schon im Feld befindet, ist die gefundene Position einer der beiden Werte, die über oder unter dem Element lägen, falls es vorhanden wäre. Liegt der gefundene Wert darüber, kann durch Verschieben aller Daten von dieser Position an Platz geschaffen werden. Liegt der gefundene Wert darunter, muß SP erst auf das nächste Feldelement gesetzt werden, damit es die richtige Position angibt, bis zu der die Daten verschoben werden sollen. Das leistet Zeile 3080.
Wenn Sie die Routine wieder mit dem Wert 5000 laufen lassen, ergibt sich eine Laufzeit von nur 49 Sekunden, die hauptsächlich auf das Verschieben der Elemente verwendet wird. Die Verzögerung infolge der großen Anzahl von Vergleichen ist fast vollständig behoben.

REINES SUCHEN

In den oben genannten Beispielen hatte das Suchen den Zweck, ein neues Element einzusetzen. Das muß natürlich nicht so sein. Mit der ersten und dritten Methode kann man auch einfach die Position eines Elements bestimmen, vorausgesetzt, es ist im Feld vorhanden. Ändern Sie die Kontrollroutine zum Ausprobieren so ab:

```
500 REM******************
501 REM KONTROLL ROUTINE
502 REM******************
510 GOSUB750
```

```
520 TI$="000000"
530 GOSUB1000
540 PRINTSP,A%(SP)
550 PRINTTI$
590 TI$="000000"
600 GOSUB3000
610 PRINTSP,A%(SP)
620 PRINTTI$
630 STOP
```

Wenn Sie jetzt einen Wert eingeben, werden er und seine Position zusammen mit der Zeit für das normale Suchen und der Zeit für das binäre Suchen ausgegeben. Falls Sie 5000, d. h. den Wert in der Mitte des Feldes eingeben, sollte das normale Suchen 34 Sekunden und das binäre Suchen eine Sekunde dauern. Geben Sie versuchsweise eine nicht-ganze Zahl ein – daraufhin wird die ganze Zahl über oder unter diesem Wert ausgedruckt. Durch Ergänzung eines Vergleichs wie

```
615 IFA%(SP)<>N1THENPRINT"NICHT VORHANDE
N" :STOP
```

wird automatisch geprüft, ob ein Element vorhanden ist.

BINÄRES SUCHEN MIT ZEIGERFELDERN

Wie wir gesehen haben, reduziert binäres Suchen die Zeit für Recherchen in einem Feld auf einen Bruchteil, so daß nur die Zeit für das Verschieben der Daten bei Einfügungen übrigbleibt. Diese Verzögerung können wir wiederum größtenteils ausschalten, indem wir binäres Suchen mit den Zeigerfeld-Techniken kombinieren, die wir in Kapitel 10 kennengelernt haben. An dieser Vorgehensweise wird eine starke Seite des modularen Programmierens deutlich; denn um das Verfahren beim Suchen nach der richtigen Position zu ändern, brauchen wir nur ein Modul auszuwechseln und sicherzustellen, daß es weiterhin die richtige Variable an das Hauptprogramm übergibt. Für das übrige Programm ist es ohne Belang, was im Suchmodul passiert, solange es nur die richtige Variable übergibt. Das Modul bei 8000 im Zeigerfeld-Programm am Ende des letzten Kapitels wird nun:

```
10 IT=5:HO=3
8000 REM**********************
8001 REM POSITIONS BESTIMMUNG
8002 REM**********************
```

```
8010 TT=IT-HO
8020 POWER=INT(LOG(TT)/LOG(2))
8030 PP=2↑POWER
8040 FORSV=POWER-1TO0STEP-1
8050 LL=INT((PP-1)/125):LP=PP-125*LL
8060 PA=256*ASC(MID$(PTR$(LL),2*LP-1))+A
SC(MID$(PTR$(LL),2*LP))
8070 PP=PP+2↑SV*((IN$<A$(PA))-(IN$>A$(PA
)))
8080 IFPP>TT-1THENPP=TT-1
8090 NEXTSV
8100 LL=INT((PP-1)/125):LP=PP-125*LL
8110 PA=256*ASC(MID$(PTR$(LL),2*LP-1))+A
SC(MID$(PTR$(LL),2*LP))
8120 IFA$(PA)<IN$THENPP=PP+1
8130 LL=INT((PP-1)/125):LP=PP-125*LL
8140 PA=256*ASC(MID$(PTR$(LL),2*LP-1))+A
SC(MID$(PTR$(LL),2*LP))
8150 RETURN
```

Eine solche Routine würden Sie sicher nicht in einem kurzen Programm unterbringen, das nur geringe Datenmengen speichert. Bei großen Datenmengen ergibt sich jedoch eine verblüffende Geschwindigkeitsdifferenz, wenn Sie mit einem Zeigerfeld das Verschieben von Feldern vermeiden und durch binäres Suchen die Suchzeit verringern.

KAPITEL 12

VERMISCHTES

Dieses Kapitel enthält eine Reihe nicht miteinander verwandter Techniken, die einzeln kein eigenes Kapitel rechtfertigen, aber dennoch beim Entwickeln eines Programms von Wert sein können. Die vorgestellten Methoden sind: DEF-Anweisungen, DATA-Anweisungen, FOR-Schleifen, Zeitmessung mit TI/TI$ und Runden mit INT.

VOM BENUTZER DEFINIERTE FUNKTIONEN

In BASIC ist eine Funktion eine Anweisung, die eine vollständige Operation eines bestimmten Typs mit einer Zahl oder einem String ausführt, um das gewünschte Ergebnis zu erhalten. Die meisten Funktionen auf dem C 64 sind mathematische Funktionen: Sie wenden mathematische Operationen auf eine Zahl an, berechnen z. B. den Sinus einer Zahl. Ohne diese eingebauten Funktionen müßte der Benutzer jedesmal, wenn er einen Sinus, Cosinus, Tangens oder ähnliches benötigt, die gesamten Anweisungen für ihre Berechnung in BASIC formulieren.
Soweit es ihre spezifischen Operationen betrifft, sind Funktionen überaus nützlich, um Komplexität und Umfang eines Programms zu reduzieren. Leider können sie nicht für alle Berechnungen eingesetzt werden, und es kommt oft vor, daß in einem Programm eine bestimmte Rechnung mehrfach auszuführen ist. Manchmal läßt sich so etwas mit einem Unterprogramm bewältigen, vielleicht sogar mit einem einzeiligen. Ein einzelnes Unterprogramm kann jedoch nur einen einzigen Variablensatz verarbeiten:

```
1800 X=Y↑2 + Y↑3 + 5*Y
```

verarbeitet z. B. immer nur die Variablen Y und X. Soll die Variable Z verarbeitet und das Ergebnis in NN abgelegt werden, müßten Sie das Unterprogramm mit dieser Zeile aufrufen:

```
200 Y=Z:GOSUB1800:NN=X
```

Eine flexiblere Methode besteht in der Definition einer neuen Funktion, wie etwa in der nächsten Zeile:

```
100 DEF FNA(Y)=Y↑2 + Y↑3 + 5*Y
```

Diese Zeile definiert eine Funktion mit dem Namen FNA (FN darf von einem beliebigen gültigen Variablennamen gefolgt werden), wobei die Variable Y im Laufe des Programms durch jede andere ersetzt werden darf:

```
300 NN=FNA(Z)
```

leistet dasselbe wie Zeile 200 vorhin.
Dadurch kann die Übersichtlichkeit eines Programms wesentlich verbessert werden. In Kapitel 11, bei der binären Suche in Verbindung mit einem Zeigerstring, ist z. B. die Zeile

```
FA=256*ASC(MID$(PTR$(LL),2*LP-1))+ASC(MI
D$(PTR$(LL),2*LP))
```

dreimal verwendet worden. Diese aufwendige und für Eingabefehler anfällige Wiederholung hätte vermieden werden können, wenn bei der Programm-Initialisierung eine Funktion definiert worden wäre, zum Beispiel:

```
100 DEFFNA(LL)=256*ASC(MID$(PTR$(LL),2*L
P-1))+ASC(MID$(PTR$(LL),2*LP))
```

Im binären Suchmodul hätte man dann nur noch nach diesem Muster Zeilen einfügen müssen:

```
8060 PA=FNA(LL)
```

Da hierbei keine Variable ersetzt worden ist, arbeitet die Funktion genauso wie die ursprüngliche Zeile. Viele Routinen in diesem Buch könnten durch die Verwendung definierter Funktionen gekürzt werden.

BEENDEN VON FOR-SCHLEIFEN

Vielleicht haben Sie bemerkt, daß in diesem Buch FOR-Streifen häufig benutzt werden, um Daten zu durchsuchen und die Position eines Elements in einem Feld oder String zu bestimmen. Wenn man eine Schleife so verwendet, taucht die Frage auf, wie man sie beenden soll. Dies muß genau durchdacht werden; denn wie in Kapitel 2 erwähnt, benötigt jede Schleife Platz im begrenzt aufnahmefähigen 'Stack' des C 64, der nur dann wieder freigemacht wird, wenn die Schleife korrekt beendet ist – d. h. wenn die Schleifenvariable die in der FOR-Anweisung gegebene Obergrenze erreicht hat. All das wird mit der nächsten Schleife erreicht:

```
100 FORI=0TOIT-1
110 IFA$(I)=IN$THENPP=I:I=IT-1
120 NEXTI
```

Sobald im Feld A$ das richtige Element gefunden ist, wird die Schleife beendet, indem I den höchsten Wert bekommt, den die Schleife verarbeiten soll. Betrachten Sie dagegen ein anderes Beispiel:

```
100 FORI=0TOIT-1
110 IFA$(I)=IN$THENGOTO140
120 NEXTI
130 PRINT"WORT NICHT GEFUNDEN":FORI=1TO5
000:NEXT
```

Wenn IN$ im Feld nicht vorhanden ist, druckt die Schleife 'ITEM NOT FOUND', anderenfalls überspringt sie das Schleifenende und die Fehlermeldung. Damit bliebe die in Zeile 100 begonnene Schleife unbeendet, was einigen Platz im Stack kostet. Trotz dieses Nachteils verwende ich diese Technik gelegentlich, und zwar aus einem Grund, der beim Durchlaufen der nächsten kurzen Routine klar wird:

```
100 FORI=1TO1000
110 FORJ=1TO10
120 NEXTI
```

Hier wird die I-Schleife ganze 1000 mal ohne schlimme Folgen begonnen und nicht beendet. Das ist möglich, weil beim Einsprung in die Schleife mit derselben Schleifenvariablen kein zusätzlicher Platz im Stack belegt wird, nachdem die Anfangsschleife einmal definiert ist.
Das können Sie ausnutzen, indem Sie durchgängig dieselben Variblennamen verwenden. Ich nehme fast immer solche, die mit I anfangen, und falls mehrere Schleifen gleichzeitig offen sein müssen, die nächsten Buchstaben im Alphabet. Wenn ich bei denselben Namen bleibe – vor allem bei Schleifen, die möglicherweise nicht korrekt verlassen werden – kann ich die Vorteile unbeendeter Schleifen nutzen – ohne rätselhafte OUT OF MEMORY-Fehlermeldungen, weil der Stack wegen zu vieler unbeendeter Schleifen übergelaufen ist.

DATA-ANWEISUNGEN

Bei einem Programm, das mit einer mehr oder weniger festen Datenmenge arbeitet, lohnt es sich oft nicht, ein besonderes Feld zu definieren und die Daten darin abzulegen. DATA-Anweisungen reichen dazu aus. Sie können im selben Programm für viele verschiedene Zwecke eingesetzt werden. Die einzige Schwierigkeit liegt darin, daß viele Leute ziemlich verunsichert zu reagieren scheinen, wenn sie mit mehreren Datenbereichen zu verschiedenen Zeiten arbeiten müssen und nicht wissen, wie sie bei Bedarf an den richtigen Datenabschnitt kommen sollen.
Dieses Problem ist in der historischen Entstehung der DATA-Anweisungen begründet, die lange vor der Geburt des Mikrocomputers bei den ersten Großrechnern anfängt. In Kapitel 4 sind schon die Probleme erwähnt worden, die den Programmierern von Großrechnern vor der Entwicklung des Dialogbetriebs zu schaffen machten, bei dem der Benutzer ein Programm während der Ausführung überwachen kann. Wie alle anderen brauchten auch die Programme für solche Geräte Daten zum Verarbeiten. Diese standen in Form von Lochkarten zur Verfügung. Die Karten wurden als Stapel in einen Kartenleser gelegt, und wenn das Programm eine Anweisung wie 'READ NN' vorfand, übersetzte der Kartenleser die gestanzten Löcher der ersten Karte in eine Zahl, übernahm diese Zahl als Wert der Variablen NN und nahm dann die zweite Karte auf. Der Lochkartenstapel konnte nur in zwei Richtungen gelesen werden: indem man die nächste Karte untersuchte (egal, ob sie gebraucht wurde oder nicht), oder indem man zum Anfang des Stapels zurückkehrte. Bei den ersten Mikrocomputern erkannte man die Fähigkeit, in einem Programm Daten zu bestimmen, zwar als nützlich an. Diese Aufgabe wurde jedoch so formuliert, als bezöge sie sich auf das Lesen von Karten. Ein BASIC-Programm enthielt Anweisungen wie diese:

```
100 DATA 12,45827,67,123,76593,212,1065
```

und die Datenelemente solcher Zeilen wurden mit der Anweisung gelesen:

```
200 READ X
```

Dabei gab es jedoch wie zuvor nur zwei Wege durch die vorhandenen Daten. Zur ersten Ausführung von Zeile 200 wurde der Variablen X der Wert 12 zugeordnet, d. h. das erste Datenelement; die zweite READ-Anweisung übergab 45 827 an eine Variable etc. Diese Abfolge konnte nur mit einem anderen Befehl unterbrochen werden:

```
300 RESTORE
```

setzte den 'Datenzeiger' auf den Anfang der ersten Datazeile des Programms zurück.
Viele moderne Mikrocomputer haben diese Einschränkung mit der Einführung des Befehls 'RESTORE LN' überwunden. LN gibt eine Zeilennummer an, und der Datenzeiger wird auf das erste Element der ersten DATA-Anweisung in oder hinter Zeile LN des Programms gesetzt. Dadurch können verschiedene Daten-Teilbereiche einbezogen werden, die jeweils verschiedenen Zwecken im Programm dienen, und bei Bedarf kann auf Daten aus jedem Bereich leicht zugegriffen werden. Da der C 64 diese zusätzliche Fähigkeit leider nicht besitzt, müssen die vorhandenen Beschränkungen auf andere Art umgangen werden.
Ein unkompliziertes Beispiel für die Verwendung von DATA-Anweisungen könnte so aussehen:

```
1000 DIMMO$(11),AA%(9)
1010 DATA JANUAR,FEBRUAR,MAERZ,APRIL,MAI
,JUNI
1020 DATA JULI,AUGUST,SEPTEMBER,OKTOBER,
NOVEMBER,DEZEMBER
1030 DATA 12,125,23,64,17,176,38,78,169,
5
1040 DATA INITIALISIERUNG,EINFUEGEN,LOES
CHEN,ANZEIGEN,ENDE
1050 FORI=0TO11:READ MO$(I):NEXT
1060 FORI=0TO9:READ AA%(I):NEXT
1070 FORI=1TO5:READ T$:PRINTT$:NEXT
```

Hier versorgen die DATA-Zeilen ein Feld mit Monatsnamen und ein anderes mit nötigen Daten. Der dritte DATA-Abschnitt druckt die Optionen als Menü, um einzelne PRINT-Befehle für jede Option zu erübrigen. Beim ersten Lauf funktioniert alles einwandfrei, dann aber könnte es erforderlich werden, während des Programmablaufs das Menü später nochmals auszudrucken. Wie die DATA-Anweisungen stehen, kann man in diesem Fall nur den Zeiger auf den Datenanfang zurücksetzen und alle Daten noch einmal lesen, wobei nicht benötigte ausgelassen werden: Die Monatsnamen müssen nicht noch einmal eingelesen werden. Diesen Zweck erfüllen die Zeilen:

```
2000 FORI=1TO22:READT$:NEXT
2010 FORI=1TO5:READ T$:PRINT T$:NEXT
```

Anders gesagt, Daten können gelesen und abgelegt werden, wenn die Anfangsposition des gewünschten Elements genau bekannt ist. Überflüssige Daten werden als Strings gelesen, denn unser Beispiel akzeptiert sowohl Zahlen als auch Stringdaten. Wo viele DATA-Anweisungen vorkommen, können jedoch bei der Positionsberechnung bestimmter Datengruppen leicht Fehler gemacht werden, so daß das Programm auf READ die falschen Daten übergibt. Darüber hinaus stört jede Veränderung an der Anzahl der Elemente die Ausgewogenheit des ganzen Prozesses, so daß im gesamten Programm die Werte aller Schleifen, die Daten lesen und ablegen, verändert werden müssen. Im günstigsten Fall ist das nur lästig, aber bei Programmen, die für die Arbeit mit häufig geänderten DATAs bestimmt sind, ist es völlig undurchführbar.
Hier bietet sich der Ausweg, DATA-Anweisungen mit Markierungen zu versehen, die dem Programm zeigen, wo es sich innerhalb der DATAs befindet und wo die Datei zu Ende ist. In Verbindung mit der oben abgedruckten Routine erhalten wir so:

```
1000 DIMMO$(11),AA%(9)
1010 DATA#1,JANUAR,FEBRUAR,MAERZ,APRIL,M
AI,JUNI
1020 DATA JULI,AUGUST,SEPTEMBER,OKTOBER,
NOVEMBER,DEZEMBER
1030 DATA#2,12,125,23,64,17,176,38,78,16
9,5
1040 DATA#3,INITIALISIERUNG,EINGABE,LOES
CHEN,ANZEIGE,ENDE,#-1
1045 FORI=1TO1000:READT$:IFT$="#1"THENI=
1000
1046 IFT$="#-1"THENRESTORE
1047 NEXTI
1050 FORI=0TO11:READMO$(I):NEXT
1055 FORI=1TO1000:READT$:IFT$="#2"THENI=
1000
1057 NEXTI
1060 FORI=0TO9:READAA%(I):NEXT
1065 FORI=1TO1000:READT$:IFT$="#3"THENI=
1000
1066 IFT$="#-1"THENRESTORE
1067 NEXTI
1070 FORI=1TO5:READ T$:PRINTT$:NEXTI
```

Das Programm erhält hier den Befehl, in den DATA-Anweisungen nach der Markierung des richtigen Datenabschnitts zu suchen und erst dann Informationen zu übernehmen. Die Zahl 1000 in den neuen Schleifen ist einfach eine beliebige Zahl, die größer als die Anzahl der Datenelemente in den DATA-Anweisungen sein muß. Wenn das Programm die Datei bis zum Ende liest, erzeugt es keinen 'OUT OF DATA ERROR', sondern entdeckt die Markierung '#−1' und weiß, daß es bei der ersten DATA-Anweisung mit dem Lesen (READ) neu beginnen muß. Auf die wenigen Daten in unserem Beispiel würden Sie diese Methode selbstverständlich nicht anwenden, aber in Programmen, die große Datenmengen dieser Art verarbeiten, können strukturierte DATA-Anweisungen ein Segen sein.

ZEITSTEUERUNG MIT TI UND TI$

Es ist Ihnen sicher nicht entgangen, daß in den bisher angegebenen Routinen zwei ganz verschiedene Methoden verwendet wurden, um Timings während des Programmablaufs vorzuschreiben und zu erzeugen. Um eine Zeitspanne vorzuschreiben, etwa für den Ausdruck einer bestimmten Meldung auf dem Bildschirm, werden fast immer FOR. . .TO-Schleifen benutzt, z. B.:

```
100 PRINT"DIES IST EINE NACHRICHT"
110 FORI=1TO2000:NEXT
120 PRINT"■"
```

Dieser einfachen Technik kann sich jeder bedienen, wobei die erforderliche Schrittgröße durch Probieren ermittelt wird. Zur Steuerung komplizierterer Vorgänge ist die Methode jedoch ungeeignet. Einen einzelnen Vorgang kann man zwar gut ausdrucken und mit Hilfe einer Schleife eine Weile stehen lassen, aber das geht nicht, wenn Sie eine *Reihe* von Vorgängen ausführen und nach Ablauf der angegebenen Zeit beenden möchten. In Einzelfällen kann das mit einer Schleife durchgeführt werden, aber es ist viel einfacher, die eingebauten Zeitgeberfunktionen des C 64 einzusetzen: TI und TI$.
Diese beiden sind eingebaute 'Systemvariablen' mit eigenen Variablenwerten, die aber normalerweise nicht vom Anwender, sondern vom C 64 selbst gesetzt werden. TI ist eine numerische Variable, die beim Einschalten des C 64 auf Null gesetzt wird und sich dann jede Sechzigstelsekunde um eins erhöht. Die Anzahl der seit dem Einschalten verstrichenen Sekunden ist demnach durch Eingeben von

```
?TI/60
```

einfach zu erfahren.

Für den täglichen Gebrauch ist TI$ wahrscheinlich nützlicher. Es leitet sich von TI ab, bietet aber ein übersichtlicheres Format *und* kann vom Benutzer auf jeden beliebigen Wert gesetzt werden. TI$ initialisiert beim Einschalten des C 64 mit "000000" und erhöht sich im Sekundentakt. Es wird jedoch nicht einfach die Sekundenzahl ausgegeben, sondern STUNDEN/MINUTEN/SEKUNDEN; 021537 bedeutet also zwei Stunden, 15 Minuten und 37 Sekunden. Um beide Variablen zu vergleichen, gibt man ein:

```
?TI/60,TI$
```

TI$ kann wie jeder andere String einen beliebigen Wert zugewiesen bekommen, und TI wird aus diesem Wert errechnet, obwohl es nicht direkt verändert werden kann. Eingeben von

```
TI$="120000"
```

weist TI$ 12 Stunden zu. Die wesentlichen Einschränkungen dabei sind: Wenn TI$ gleich einer Zahl gesetzt wird, die nicht genau sechs Stellen hat, wird ein Fehler erzeugt, und bei einem Wert über "240000" wird TI$ auf Null zurück gesetzt.
Diese Eigenschaft von TI$ haben wir gelegentlich schon beim Vergleich verschiedener Sortier- und Suchverfahren benutzt, indem wir am Anfang TI$ "000000" gesetzt und nach Beendigung des betreffenden Prozesses ausgedruckt haben, um die abgelaufene Zeit festzustellen. Selbst diese Technik kann auf vielfältige Weise eingesetzt werden, z. B. um Reaktionszeiten in Tests und Spielen zu kontrollieren, oder als professioneller Touch bei Anwendungsprogrammen, die dem Benutzer mitteilen, wie lange das Programm am Schluß einer bestimmten Arbeitssitzung in Betrieb ist.
Des weiteren kann man mit TI und TI$ nicht nur Zeitspannen ausdrucken, sondern auch leicht *vorschreiben.* Lesen Sie folgende Zeilen:

```
1000 TT=TI
2000 IF TI-TT>18000 THEN RETURN
```

Stände die erste Zeile am Anfang einer Routine und wäre die zweite irgendwo darin eingebunden, wo sie regelmäßig abgearbeitet würde, so würde die betreffende Routine nach fünf Minuten beendet (18 000/60 = 300 Sekunden = 5 Minuten). Wenn der Wert von TI nicht erhalten werden muß, kann sogar die Berechnung der gewünschten Anzahl von Sechzigstelsekunden entfallen. Die nächsten beiden Zeilen hätten dieselbe Funktion:

```
1000 TI$="0000001000 TI$="000000"
2000 IF TI$>"000000" THEN RETURN
```

Das Haar in der Suppe ist dabei der Kassettenrekorder: Solange mit ihm kommuniziert wird, zählen TI und TI$ nicht weiter. Man kann ein Programm um die 'Echtzeituhr'-Funktion ergänzen, indem man Ti$ aufspaltet und etwas übersichtlicher darstellt:

```
1000 TT$=TI$
1010 PRINT LEFT$(TT$,2);":";MID$(TT$,3,2
)":";MID$(TT$,5)
```

Der Gebrauch eines Hilfsstrings TT$ ist der direkten Arbeit mit TI$ vorzuziehen, da TI$ sich während des Aufspaltens möglicherweise ändert. Wenn also TI$ beim Ausdrucken der Zeit "005959" gewesen und auf "010000" gesprungen wäre, bevor Minuten und Sekunden errechnet waren, ergäbe sich die Zeit '00:00:00'.

RUNDEN MIT INT

Die Funktion INT, die Nachkommastellen von einer Zahl abschneidet, ist in den vorhergehenden Routinen oft gebraucht worden. Wie wir wissen, kann INT auch die maximale erlaubte Anzahl von Dezimalstellen einer Zahl bestimmen. Eine Manipulation wie

```
100 NN=INT(1000*NN)/1000
```

ergäbe drei Dezimalstellen.
Wir sollten nicht außer acht lassen, daß INT auch Zahlen auf die *nächstgrößere* ganze Zahl aufrunden kann, nicht nur auf die nächstkleinere abrunden. Dazu addiert man einfach 0.5 zu einer Zahl und läßt sie dann mit INT verarbeiten. Probieren Sie die nächste Routine durch Eingeben einer Reihe nicht-ganzer Zahlen aus:

```
100 INPUT NN
110 PRINT INT(NN+0.5)
120 GOTO100
```

KAPITEL 13

FORMATIEREN

Die Fähigkeit des C 64, Informationen programmgesteuert auf den Bildschirm zu bringen, ist gekennzeichnet durch eine eigenartige Mischung aus großer Flexibilität einerseits und einem Defizit an Standardfunktionen andererseits, über die fast alle Mikrocomputer der heutigen Generation verfügen. Die Flexibilität ergibt sich aus der Freiheit des Programmierers, mit direkten Cursor-Steuerzeichen die Printposition zu ändern und mit anderen Steuerzeichen die Farbeigenschaften zu kontrollieren, und aus der Leichtigkeit, mit der Zeichen in den Bildschirmspeicher und von da auf den Bildschirm gepoket werden können. Ein Nachteil liegt im Fehlen von Anweisungen wie PRINT AT, das die Bestimmung einer einzelnen Bildschirmposition durch einen einzigen BASIC-Befehl erlaubt, oder PRINT USING, das in derselben Weise die Bestimmung des Formats von Zahlen oder Strings ermöglicht, um z. B. beim Ausdrucken eine einheitliche Anzahl von Dezimalzahlen zu garantieren. In diesem Kapitel werden wir untersuchen, wie man beim C 64 das beste aus seinen starken Seiten macht und manche seiner schwachen Seiten umgeht.

CURSORSTEUERUNG

Es gibt so viele Arten, Cursor-Steuerzeichen zu benutzen, wie es verschiedene Verwendungsmöglichkeiten des Bildschirms zur Darstellung von Informationen gibt. Programme, die auf dem Bildschirm komplexe Strukturen erzeugen, sind auf dem C 64 wahrscheinlich viel leichter zu realisieren als auf jedem anderen vergleichbaren Heimcomputer. Informationen können seitenweise auf den Bildschirm geschrieben werden, ohne ständig die Position neu zu berechnen, wie es bei PRINT AT nötig ist. Zeichen können ganz bequem nach oben, unten, rechts oder links (bezogen auf das letzte Zeichen) verschoben werden. Betrachten Sie das folgende Beispiel, das eine Reihe von Strings mit vier Zeichen von rechts unten nach links oben diagonal auf dem Bildschirm schreibt:

```
100 A$="****"
110 PRINT"
                                        ";
120 FORI=1TO24
130 PRINTA$;"
140 NEXT
```

Aus diesem kleinen Beispiel lassen sich zwei Schlüsse ziehen. Erstens: Die Bewegung eines Elements in bezug auf das vorhergehende, also die 'relative Bewegung', ist einfach und logisch. Sie bestimmen lediglich die Anzahl der Stellen, um die Sie in einer beliebigen Richtung weiterrücken möchten, und bedienen die entsprechende Cursorsteuerung. Der zweite Schluß ergibt sich aus Zeile 110: Die Cursorsteuerung kann mühsam werden, wenn man Printpositionen bestimmt, die nicht in der Nähe der linken oberen Bildschirmecke liegen. Diese Einschränkung kann durch Zuweisung von zwei Strings beim Initialisieren des Programms in gewisser Weise überwunden werden, z. B.:

```
100 HO$="███████████████████████████████████████
        ":REM 39 * CURSOR RECHTS
110 VE$="████████████████████████"        :
REM 24 * CURSOR RUNTER
```

Diese beiden Strings enthalten genügend Cursor-Steuerzeichen in VErtikaler und HOrizontaler Richtung, um die Printposition von links oben an jede beliebige Stelle auf dem Bildschirm zu bringen. Um die Printposition in die letzte Zeile zu fahren, müßte man etwa folgende Zeile eingeben:

```
120 PRINT"♥";LEFT$(VE$,24);LEFT$(HO$,34)
;
```

Auf der Buchseite sieht das länger aus als im Original, aber die Angabe der Strings in voller Länge hätte natürlich eineinhalb Zeilen ausgefüllt. Wenn Sie die Funktion, die im Grunde eine Version von PRINT AT ist, regelmäßig benutzen wollen, können Sie den Vorgang durch ein speziell für diese Aufgabe geschriebenes Unterprogramm etwas verkürzen:

```
1710 PRINT"♥";LEFT$(VE$,VE);LEFT$(HO$,HO
);:RETURN
```

Nach der Eingabe kann die Printposition mit einer Zeile wie

```
120 VE=12:HO=12:GOSUB1710:PRINT"HALLO"
```

beliebig über den Bildschirm bewegt werden. Das ist etwas kürzer als die vorige Version, die jedesmal LEFT$ verwendete, und auch leichter lesbar. Man sollte dabei nicht vergessen, daß die Bildschirmpositionen nicht von eins sondern von null an numeriert sind, d. h. die Vertikale zählt von 0 bis 24 und die Horizontale von 0 bis

39. Falls Sie jeweils von eins an zählen möchten, muß das Unterprogramm modifiziert werden, damit es mit VE−1 und HO−1 läuft.
Damit können wir noch einmal die diagonale Stringreihe ausprobieren (HO$ und VE$ werden als definiert vorausgesetzt):

```
120 A$="****"
130 FOR VE=24TO0 STEP-1
140 HO=VE+10
150 GOSUB1710:PRINTA$;
160 NEXT VE
170 END
```

Auch wenn dies nur eine Nachahmung ist, zeigt sich darin die Stärke von PRINT AT, nämlich die Möglichkeit, den Bildschirm mit Hilfe von Variablen zu formatieren. Auf diese Weise können ziemlich komplexe Strukturen erzeugt werden. Die nächste Routine bildet eine dreieckige Tabelle aus Zeichenpaaren. Es handelt sich in dem Beispiel um einen Pseudo-String, aber die Daten könnten genauso aus einem Feld mit sinnvollen Daten geholt werden.

```
120 A$="**"
130 FORVE=0TO12
140 FORHO=0TO3*VE STEP3
150 GOSUB1710:PRINTA$
160 NEXTHO,VE
170 END
```

Mit PRINT AT kann der Benutzer auch die Bildschirmformatierung steuern, und diese Funktion ist oft sehr nützlich. Stellen Sie sich ein 12 mal 6 großes Feld mit Informationen vor, die auf dem Bildschirm in Tabellenform dargestellt werden. PRINT AT erlaubt dem Benutzer, neue Elemente einzufügen und ausgedruckt zu sehen, ohne die gesamte Bildschirmseite vom Feld aus neu schreiben zu müssen. Nachdem die Formatierstrings VE$ und HO$ definiert sind und das Unterprogramm eingegeben ist, sähe ein typisches Programm so aus:

```
500 DIMA%(11,5):PRINT"■"
510 FORVE=0TO11:FORHO=0TO30 STEP6:GOSUB1
710
520 PRINTA%(VE,HO/6):NEXTHO,VE
530 VE=21:HO=0:GOSUB1710
```

```
540 INPUT"REIHE ";VE
550 INPUT"SPALTE ";HO
560 INPUT"WERT (0-32767) ";NN
580 GOSUB1710:PRINT"      ":REM FUENF SPA
CES
590 GOSUB1710:PRINTNN
600 GOTO530
```

Wenn Sie das Programm laufen lassen, werden Sie auf die Schwierigkeit stoßen, daß die INPUTs immer ihre Vorgänger überschreiben. Das kann verwirrend sein und bringt uns auf einen weiteren zweckmäßigen Formatierstring, den ich in eigenen Programmen meist O$ nenne. O$ enthält nichts weiter als 39 Leerstellen und wird wie üblich beim Programmstart definiert. Danach kann es in Verbindung mit PRINT AT zum Löschen häufig benutzter Zeilen dienen. Man könnte in das obige Programm eine Schleife einfügen:

```
525 HO=0:FORVE=21TO23:GOSUB1710:PRINT O$
:NEXT
```

In den benötigten Zeilen würden damit alle Zeichen gelöscht.

VERWENDUNG VON CURSOR-STEUERZEICHEN

Die Nachahmung von PRINT AT ist zwar sinnvoll, aber auf die direkte Verwendung von Cursor-Steuerzeichen in PRINT-Anweisungen sollte deshalb nicht verzichtet werden. Die meisten Bildschirmseiten haben keine regelmäßige Struktur, sondern sind auf Übersichtlichkeit für den Benutzer angelegt. PRINTs und INPUTs werden oftmals ausgerückt, als Blickfang oder zwecks Unterteilung von Informationen in logische Abschnitte auf dem Bildschirm. Kleine relative Bewegungen der Printpositionen werden am besten durch Cursor-Steuerzeichen veranlaßt (vgl. Seite 7):

```
100 INPUT"WERT 1 ";A
110 INPUT"■WERT 2 ";B
120 INPUT"■WERT 3 ";C
130 INPUT"■■■WERT 4 ";D
140 INPUT"■WERT 5 ";E
```

Hier wäre es völlig überflüssig, die Inputs mit einem PRINT AT-Unterprogramm auszudrücken. Es gibt sogar regelmäßige Strukturen, die man günstiger mit Steuer-

zeichen direkt gestaltet. Ein Beispiel dafür ist ein Wort, das vertikal statt horizontal auf den Bildschirm geschrieben werden soll. Die Verwendung von zwei Cursor-Steuerzeichen erspart hier die mühsame Arbeit mit Variablen:

```
100 A$="HALLO"
110 FORI=1TOLEN(A$)
120 PRINTMID$(A$,I,1);"■■";
130 NEXTI
```

Auch beim Überschreiben einer Textzeile kann man die Taktik zu weit treiben, wo ein einfaches 'Cursor hoch' denselben Zweck erfüllt:

```
100 PRINT"DIES IST EIN TEXT"
110 FORI=1TO2000:NEXTI
120 PRINT"□UND DIES UEBERSCHREIBT IHN"
```

Wie wir im Kapitel über Eingaben gesehen haben, kann dieselbe Technik bei INPUTs verwendet werden, indem die Abfrage mit 'Cursor hoch' eingeleitet wird. So kann bei einem Eingabefehler der INPUT neu geschrieben und automatisch über die vorhergehenden Zeichen gedruckt werden.

TAB

Die Funktion TAB ist eine weitere Formatierhilfe, mit der man Daten in einzelne Kolonnen aufteilen kann. Die nächsten Zeilen geben Zeichengruppen in gleichen Abständen auf den Bildschirm aus:

```
100 PRINT"■";:FORI=0TO36STEP3
110 PRINTTAB(I);"**";
120 NEXTI
```

Selbstverständlich kann damit mühelos eine Tabelle auf dem Bildschirm formatiert werden, allerdings sind dazu einige Hinweise nötig: Erstens hat TAB das Format TAB(X) *ohne* Lücke zwischen TAB und der ersten Klammer. Wenn Sie dazwischen eine Leerstelle lassen, was einleuchtend wäre, werden die Daten nicht formatiert, und vor jedem Element erscheint eine '0'. Zweitens arbeitet TAB immer vom Anfang der gerade beschriebenen Zeile an, und nur von links nach rechts. TAB(10) heißt also: "Drucke ab Zeichen 10, falls die Zeichenspalte 10 nicht bereits überschritten

wurde." Beim Versuch, mit TAB an eine Position *vor* der aktuellen Zeichenposition zu springen, wird TAB völlig ignoriert, und das Zeichen wird in die nächste verfügbare Spalte geschrieben. Sie können das überprüfen, indem Sie die oben angeführt Routine abändern:

```
100 PRINT"■":FORI=36TO0STEP-3
110 PRINTTAB(I);"■**"
115 GETT$:IFT$=""THEN115
120 NEXTI
```

Als Ergebnis dieser veränderten Routine bewegt sich die Printposition von '**' immer nach rechts, als ob TAB nicht vorhanden wäre. Wenn Sie das Semikolon am Schluß von Zeile 110 entfernen, wird jedesmal eine neue Zeile begonnen und das Problem stellt sich nicht.
TAB löscht nicht die Zeichen, über die es sich bewegt, sondern nur die bedruckten Zeichenpositionen. Aus diesem Grund kann man es – ähnlich wie bei der o. g. Nachahmung von PRINT AT – zur Änderung von Tabellen verwenden:

```
100 FORI=1TO10:PRINT"*** ";:NEXT:REM EIN
 SPACE NACH DEN STERNEN
110 PRINT"■";
120 FORI=0TO32STEP8:PRINT TAB(I);"***";:
NEXT
```

SPC

SPC hat dieselbe Funktion wie TAB, d. h. es bewegt die Printposition von der aktuellen Spalte um mehrere Spalten nach rechts. In den Handbüchern zum C 64 wird 'Space' fälschlich als eine Funktion erklärt, die Leerstellen druckt. Genau wie TAB druckt SPC nichts, es bewegt einfach die Printposition, ohne die Zeichen im übersprungenen Bereich zu beeinträchtigen. Geben Sie zur Probe diese kurze Routine ein:

```
100 PRINT"■XXXXX"
110 PRINT"■"SPC(7);"*"
```

Die von SPC übersprungene, mit X beschriebene Zeile wird nicht gelöscht.
Da SPC immer in bezug auf die aktuelle Printposition arbeitet, eignet es sich weniger als TAB zum Formatieren einer regelmäßigen Struktur auf dem Bildschirm, es sei denn, die Länge der gedruckten Elemente ist einheitlich. Wird SPC zum

Trennen von Elementen in mehreren Zeilen eingesetzt, so werden die Elemente zwar in gleichen Abständen, aber nicht unbedingt in Kolonnen gedruckt. SPC ist jedoch in einer Hinsicht überlegen: Viele Drucker haben Schwierigkeiten mit TAB, arbeiten jedoch reibungslos mit SPC. Falls Sie derartige Probleme mit dem Ausdrucken von Tabellen haben, definieren Sie zunächst die Breite der gewünschten Kolonne einschließlich Leerspalten und gleichen Sie dann mit SPC mögliche Unterschiede zwischen der Länge des zu druckenden Strings und der Breite der Kolonne aus:

```
100 A$(0)="*":A$(1)="**":A$(2)="***":A$(
3)="****"
110 OPEN1,4:CMD1
120 FORI=1TO10
130 FORJ=0TO3
140 PRINTA$(J);SPC(6-LEN(A$(J)));
150 NEXTJ:PRINT:NEXTI
160 PRINT#1:CLOSE1
```

Zeilen 100 und 160 der Routine öffnen eine Datei auf den Drucker, übertragen vorübergehend alle gedruckten Ausgaben in diese Datei und schließen die Datei nach Beendigung des Druckens. Beachten Sie, daß am Ende jeder Zeile von Elementen ein eigenes PRINT stehen muß. Wäre das nicht der Fall, würde SPC über das Zeilenende hinaus weitere sechs Zeichenkolonnen drucken und so das Format zerstören, im Unterschied zu TAB, das bei Erreichen des Zeilenendes das Semikolon hinter der PRINT-Anweisung ignoriert.

NACHAHMUNG VON PRINT USING

Tabellen lassen sich oft viel leichter formulieren, wenn man Inhalte einheitlicher Länge verarbeiten kann. Zur Veranschaulichung können Sie folgende Routine eingeben:

```
100 PRINT"■";
110 FORI=1TO20
120 PRINT(RND(0)*15)↑2
130 NEXT
```

Wie Sie sehen, ergibt sich daraus eine recht verworrene Kolonne. Es wäre wenig zweckmäßig, mit diesem Format eine Tabelle zu erzeugen, auch wenn hier alle Zeilenanfänge genau untereinanderliegen. Die sinnvolle Darstellung numerischer

Daten setzt voraus, daß die Formatierung den Vergleich verschiedener Zahlen auf einen Blick ermöglicht.
Viele Mikrocomputer lösen die Aufgabe mit dem Befehl 'PRINT USING', der dem Programmierer erlaubt, das Format einer Zahl oder eines Strings vorzuschreiben: wie viele Stellen, wie viele Dezimalstellen, Ausfüllen mit Blanks bis auf Standardlänge. Leider besitzt der C 64 diese Funktion nicht, jedoch sind (wie PRINT AT) die meisten notwendigen Funktionen leicht zu simulieren.
Um eine Zahl zu formatieren, übersetzen wir sie als erstes in einen String. Der ursprüngliche Wert bleibt dabei unverändert; es handelt sich nur um eine Hilfsmaßnahme zum Drucken, durch die wir uns über die Länge des Elements informieren und das Format verändern können. Mit Hilfe einer einfachen Eigenschaft des Logarithmus ermitteln wir die Anzahl der Stellen vor dem Dezimalpunkt und standardisieren dann das Format mit einfachen Stringoperationen. Man schreibt dafür am besten ein Unterprogramm, da die Formatierfunktion vermutlich mehrmals im Programm gebraucht wird:

```
100 SS$="                                   ":REM SOV
IEL SPACES WIE MOEGLICH
1810 NN$=LEFT$(SS$,8-INT(LOG(ABS(NN))/LO
G(10)+1))+STR$(NN)
1820 RETURN
```

Zeile 100 weist lediglich darauf hin, daß die Routine nur funktioniert, wenn vorher ein Leerstring SS$ definiert wurde. Der Ausdruck LOG(NN)/LOG(10) ergibt den Wert von NN, wenn er in Logarithmen zur Basis 10 ausgedrückt ist; auf dieser Basis arbeitet das Programm. Das ist einfach deswegen sinnvoll, weil der ganzzahlige Teil eines Logarithmus zur Basis 10 um eins kleiner ist als die Anzahl der Vorkommastellen. Also ist LOG(120)/LOG(10)=2.0918125, und durch Addieren von 1 zum ganzzahligen Teil ('2') erfahren wir, daß die Zahl drei Vorkommastellen hat. Wie Sie sehen, haben wir für unsere Berechnung den LOG von ABS(NN) genommen, denn der LOG einer negativen Zahl wäre Unsinn. Ausgerüstet mit diesen Informationen ist es nun ein leichtes, für eine einheitliche Anzahl von Vorkommastellen zu sorgen, indem man der Zahl ein ausreichend langes SS$ voranstellt. Dazu dienen LEFT$ und STR$; letzteres wandelt eine Zahl in einen String desselben Formats wie die ursprüngliche Zahl um. Bei der vorigen Routine haben die Zahlen in der Regel neun Vorkommastellen oder -ziffern. Es sind neun statt acht, weil STR$ an den Anfang einer positiven Zahl – dort, wo bei einer negativen Zahl das Minuszeichen stände – eine Leerstelle einsetzt.
Sie sind nun in der Lage, anhand einer modifizierten Version der weiter oben erklärten Zufallszahlenerzeugung die Arbeitsweise des Unterprogramms auszuprobieren:

```
100 PRINT"␣";
110 FORI=1TO20
120 NN=(RND(0)*15)↑2
130 GOSUB1810
140 PRINTNN$
150 NEXT
160 END
```

Das Problem der Ausrichtung von Dezimalpunkten ist damit gelöst, aber bevor wir Daten streng tabellarisch gestalten können, bleibt noch eine andere Frage zu klären: Die Anzahl der *Nach*kommastellen. Dabei müssen wir etwas anders vorgehen. Bei Zahlen mit zu vielen Dezimalstellen kann man einfach den String aufgliedern, bei zu wenigen ist das jedoch nicht ganz so leicht. Wir können nicht einfach den formatierten String bis zur gewünschten Länge mit Nullen auffüllen; denn es könnten auch Zahlen ohne Dezimalpunkt vorkommen, bei denen angehängte Nullen zu einem sinnlosen Ergebnis führten. Zum Glück gibt es ein einfaches Mittel, das uns die Überprüfung der Nachkommastellen erspart: Wir addieren zu jeder Zahl einen Dezimalbruch, *der mit mindestens so vielen Nullen beginnt, wie unser Standardformat erfordert.* Um zwei Dezimalstellen zu erhalten, brauchen wir demnach nur .001 zu der ursprünglichen Zahl zu addieren. Bei der Umwandlung der Zahl in einen String wird die '1' am Ende dann einfach abgeschnitten. Zahlen, die schon zwei Dezimalstellen haben, ändern ihren Wert dadurch nicht, sondern werden nur um einen Anhang von Nullen ergänzt, und Zahlen ohne Dezimalstellen erhalten bei Bedarf außerdem einen Dezimalpunkt. Die einzige Komplikation betrifft negative Zahlen, die besondere Maßnahmen erfordern. Die Formatierung nach diesem Muster geschieht ebenfalls mit einer kurzen Routine:

```
1900 REM****************
1901 REM DEZIMAL FORMAT
1902 REM****************
1910 NN$=STR$(INT(100*NN)/100+(.001*SGN(
NN)))
1920 NN$=LEFT$(SS$,8-INT(LOG(ABS(NN))/LO
G(10)+1))+LEFT$(NN$,LEN(NN$)-1)
1930 RETURN
```

Hier ist nur eine Stelle unklar, nämlich die Verwendung der Funktion SGN in Zeile 1910. Auf eine Zahl angewendet, bildet SGN das Ergebnis 1 oder −1, je nachdem, ob die Zahl positiv oder negativ ist. Folglich wird in Zeile 1910 das Ergebnis der ersten Zeilenhälfte um .001 erhöht oder vermindert, in Abhängigkeit davon, ob die

ursprüngliche Zahl positiv oder negativ war. Das Ergebnis von 123 + .001 ist z. B. sinnvoll (123.001) und nach Entfernen der '1' am Schluß zu verwenden, während −123 + .001 uns ein völlig sinnloses Ergebnis (−122.999) liefert, mit dem wir nichts anfangen können. Wenn NN −123 gewesen wäre, hätte SGN dafür gesorgt, daß .001 mit −1 multipliziert und daher subtrahiert statt addiert worden wäre; −123.001 *wäre* das gewünschte Resultat.
Wir haben jetzt fast alles, was wir brauchen, um das Format einer Zahl innerhalb der Grenzen zu bestimmen, die eine ordentliche Datentabelle auf dem Bildschirm verlangt. Es bleibt nur noch eine einzige Frage offen: mögliche Minuszeichen. Wie bereits angedeutet, wird einer Zahl bei der Umwandlung in einen String eine Leerstelle vorangestellt, wenn sie positiv ist, bzw. ein Minuszeichen, wenn sie negativ ist. Um möglichst viele Daten in einer Tabelle unterzubringen, kämen wir besser ohne Minuszeichen hin; z. B. könnten wir bei vier verfügbaren Stellen pro Zahl Werte bis 9999 aufnehmen, statt uns auf 999 plus Leerstelle für ein eventuell vorkommendes Minuszeichen zu beschränken. Die Lösung besteht einfach darin, das ohnehin unauffällige Minuszeichen abzuschaffen und zur Kennzeichnung negativer Zahlen die Farbfunktionen des C 64 einzusetzen:

```
2000 REM*******************************
2001 REM KENNZEICHNEN NEGATIVER WERTE
2002 REM*******************************
2010 NN$=STR$(INT(100*NN)/100+(.001*SGN(
NN)))
2020 NN$=LEFT$(SS$,8-INT(LOG(ABS(NN))/LO
G(10)+1))+LEFT$(NN$,LEN(NN$)-1)
2030 IF NN<0 THEN NN$="[REVERS EIN]"+NN$
+"[REVERS AUS]"
2040 RETURN
```

Gegenüber der letzten Routine bewirkt die zusätzliche Zeile 2030, daß negative Zahlen nicht nur formatiert, sondern auch auf dem Bildschirm invers dargestellt, d. h. hervorgehoben werden. Wenn Sie einen Farbmonitor verwenden, können Sie die Zeichenkette auch rot drucken, indem Sie das RVS-Zeichen am Anfang durch das Steuerzeichen für Rot ersetzen.
Bei entsprechender Bearbeitung dieser Technik können Sie nun komplexe Tabellen in übersichtlichem Format auf den Bildschirm schreiben, die alle gewünschten Informationen viel wirkungsvoller vermitteln als die formlosen Zeichenkolonnen, die man auch nach zwei- oder dreimaligem Lesen noch nicht ganz versteht.

JUSTIEREN

Werden Strings innerhalb einer Tabelle dargestellt, ist es oft angebracht, sie in einheitlicher Länge zu gestalten, damit mehrere Kolonnen mit Wörtern rechts- oder linksbündig in aufeinanderfolgenden Zeilen gedruckt werden können. Hier kann man mit der Printposition taktieren, aber häufig gibt es auch die einfache Möglichkeit, jeden String bis auf Standardlänge aufzufüllen. Mit der nächsten Zeile wird beispielsweise ein String am Ende bis zu einer Gesamtlänge von zehn Zeichen mit Leerstellen aufgefüllt:

```
100 A$=LEFT$(A$+SS$,10)
```

wobei SS$ der schon bekannte Leerstring ist. Zum Auffüllen des Anfangs (damit jedes Wort an derselben Stelle endet) gibt man ein:

```
100 A$=RIGHT$(SS$+A$,10)
```

Beide Zeilen basieren darauf, daß Strings innerhalb der Klammer einer Stringfunktion addiert werden können, und erübrigen die mühsame Konstruktion einer Zeile wie:

```
100 A$=A$ + LEFT$(SS$,10-LEN(A$))
```

Die beiden kürzeren Versionen führen jedoch zur Verstümmelung von A$, wenn es den zugewiesenen Platz überschreitet. Diese Wirkung können Sie gegebenenfalls auch absichtlich einsetzen, um das Format einer Tabelle zu bereinigen.

EINFACHE LOGOS UND GRAFIKEN

Die Auflockerung eines guten Programms mit einfachen Grafiken, einer Titelseite für den ersten Programmstart und einzelnen Illustrationen kann ausschlaggebend dafür sein, wie der spätere Benutzer damit zurechtkommt. Die Benutzer eines C 64 haben insofern Glück, als er sich im Vergleich zu anderen Heimcomputern wahrscheinlich am leichtesten für Programmkosmetik benutzen läßt, obwohl die Gestaltung vieler Programme auf diesbezügliche Unkenntnis schließen läßt. Bei anderen Heimcomputern läßt sich eine gute Grafik oft erst nach komplizierter Planung auf den Bildschirm bringen: Entwurf auf Millimeterpapier und mühsame Übertragung der Zeichnung in Print-Anweisungen. Beim C 64 dagegen kann man den Bildschirm einfach wie einen Zeichenblock behandeln, auf dem man mit Hilfe des hervorragenden Zeichensatzes eine beliebige Grafik entwirft und diese *danach* einfach in Programmzeilen überträgt.

Zur Herstellung unkomplizierter Grafiken, die nicht mehr als 35 Zeichenspalten horizontal ausfüllen, löschen Sie einfach den Bildschirm, bewegen den Cursor und tippen oder löschen Zeichen von der Tastatur aus, ganz gleich ob Text oder Grafik. Die einzige Regel dabei schreibt einen mindestens fünf Zeichen breiten Rand an der linken Bildschirmseite und mindestens eine Leerzeile rechts vor. Wenn Sie die Grafik wie beschrieben auf dem Bildschirm ausgeführt haben, kehren Sie an den linken Rand des Bildschirms zurück und setzen Sie von oben nach unten Zeilennummern mit nachfolgendem Fragezeichen und Anführungszeichen. (Um das 'Abführungszeichen' brauchen Sie sich nicht zu kümmern.) Natürlich sollten Sie jeweils RETURN-Taste drücken.
Sie sind nach der Numerierung jeder Zeichenzeile am unteren Rand angelangt und haben aus Ihrer Grafik praktisch ein Programm gemacht. Auch wenn Sie mit der Anordnung der Zeichen auf dem Bildschirm experimentieren wollen, müssen Sie nicht mehr tun. Auf diese Weise kann alles eingegeben werden, was sich mit dem normalen Zeichensatz aufbauen läßt: ein Bild, eine Titelseite aus großformatigen Buchstaben, sogar der Plan für eine komplexe Tabelle.
Die wichtigste Einschränkung betrifft die Breite der so eingegebenen Grafiken. Bei Eingaben, die so lang sind, daß kein Platz mehr für Zeilennummern und die abgekürzte Print-Anweisung ('?') bleibt, müssen Sie den Bildschirm zur Hälfte beschreiben und dann mit SHIFT/INST an den Anfang einer Zeile Leerstellen einfügen. Dabei läuft jede einzelne Zeile in die nächste über und zerstört so scheinbar die Grafik. Das soll Sie nicht beirren. Setzen Sie einfach eine Zeilennummer vor jede volle Zeile (nicht vor die Zeilenreste); beim Programmdurchlauf merken Sie dann, daß die Grafik korrekt ausgedruckt wird. Sie dürfen erst einmal nur die halbe Grafik eingeben, weil sie sich beim Zeilenüberlauf auf dem Bildschirm nach unten ausdehnt und alle Zeilen verlorengehen, die dabei über den unteren Rand hinausgeschoben werden. Nach Eingabe der halben Grafik können Sie sie vollständig oder teilweise auf die obere Bildschirmhälfte drucken und nach derselben Methode ergänzen.
Die eventuell einzufügenden Zeilennummern sind der Grund dafür, daß die letzte Spalte der Zeile freibleiben sollte. Reicht eine Grafik in zwei aufeinanderfolgenden Zeilen bis zum Bildschirmende, so interpretiert der C 64 sie als 80-Zeichen-Zeile und verhindert das Einfügen weiterer Zeichen. Falls Sie gezwungen sind, bis zum Ende des Bildschirms zu gehen, dürfen Sie das erst nach der Übertragung der Grafik in Programmzeilen tun. Nun können Sie am Ende jeder Zeile das notwendige zusätzliche Zeichen einfügen, dahinter ein abschließendes Fragezeichen und ein Semikolon, sonst wird die Grafik im Wechsel mit Leerstellen gedruckt.

SCHLUSS

Ein potentiell nützliches Programm, das wegen der undurchschaubaren Datendarstellung im Grunde für keinen praktischen Zweck eingesetzt werden kann, gehört für den Computer-Benutzer zu den ärgerlichsten Dingen der Welt. Die Anwendung der in diesem Kapitel beschriebenen Techniken macht kaum Mühe, ihre Wirkung hingegen ist ganz ohne Mühe zu erkennen. Probieren Sie es.

NACHWORT

Wenn Sie das Buch durchgearbeitet und dabei alle Programmbeispiele eingegeben haben, sind Sie geduldiger als ich. Selbstverständlich habe auch ich sie eingegeben, aber ich werde schließlich auch dafür bezahlt.
Viel wahrscheinlicher haben Sie große Teile des Buches einfach überflogen und nur angehalten, um besonders interessante Programme und Methoden auszuprobieren. Stellen Sie das Buch danach aber nicht gleich als eines von den Nachschlagwerken, die man ab und zu konsultiert, zu den anderen ins Regal. Sollten Sie nicht gerade vorhaben, sich sofort auf ein neues Projekt einzulassen, können Sie das Buch am besten nutzen, indem Sie Ihre eigenen Programme starten und versuchen, das Gelernte darauf anzuwenden. Es scheint zwar merkwürdig, kompliziertere Techniken auf eher einfache und sogar zufriedenstellende Programme anzuwenden, aber darum geht es nicht. Datenverarbeitungstechniken kann man sich nur aneignen, wenn man sie ein- oder zweimal benutzt, selbst wenn der Nutzen nichts nützt (falls Sie mir folgen können).
Ein weiterer Vorschlag wäre, aus den abgedruckten Routinen inklusive Einzeilern ein Verzeichnis nützlicher Techniken auf Band oder Diskette anzulegen. Wie Sie bemerkt haben, sind die komplexeren Techniken schon so dargestellt, daß sie leicht eingegeben und ausprobiert werden können. Einzeilige Routinen könnten zwecks Veranschaulichung ihrer Wirkung um eine Input- und eine Print-Anweisung ergänzt werden. Wenn Sie die Routinen in dieser Form abspeichern, können Sie sie zu einem späteren Zeitpunkt daraufhin untersuchen, ob sie eine bestimmte Aufgabe bewältigen; in diesem Fall können sie, wie in Kapitel 1 vorgeschlagen, in das Programm eingebunden werden. Weitere geeignete Routinen können aus Zeitschriften und Büchern entnommen werden, auch wenn Sie an den zugehörigen Programmen nicht interessiert sind. Ich selbst besitze eine Sammlung von Routinen, die zur Zeit nicht viel leisten, aber bei jeder neuen schwierigen Aufgabe bin ich im nachhinein froh darüber, sie angelegt zu haben.
Vergessen Sie also nicht einfach diejenigen Techniken, für die Sie im Moment keine Verwendungsmöglichkeiten sehen. Früher oder später sind sie es vielleicht, die über Erfolg oder Mißerfolg entscheiden.

FOLGENDE BÜCHER SIND BISHER IN DER COMMODORE SACHBUCHREIHE ERSCHIENEN:

Band 1
ALLES ÜBER DEN COMMODORE 64
ISBN 3-89 133-000-6
(Artikel-Nr. 55 64 20)
1984, 480 S., Fe. Ebd.

DM 59,–

Band 2:
ALLES ÜBER DEN VC 20
ISBN 3-89 133-004-9
(Artikel-Nr. 58 00 20)
1984, 200 S., Br.

☐ **DM 9,80**

Band 3:
LOGO FÜR COMMODORE
mit 2 Disketten
ISBN 3-89 133-001-4
(Artikel-Nr. 64 10 50)
1984, 364 S., A4, Fe. Ebd.

DM 159,–

Band 4:
DAS COMMODORE 64
SPIELE-BUCH
ISBN 3-89 133-002-2
(Artikel-Nr. 55 64 15)
1984, 160 S., Br.

DM 29,80

Band 5:
DAS VC 20 SPIELE-BUCH
ISBN 3-89 133-003-0
(Artikel-Nr. 58 00 15)
1984, 160 S., Br.

DM 29,80

Band 6:
PLUS/4 ROM-LISTING
ISBN 3-89 133-006-5
(Artikel-Nr. 58 40 00)
1984, 280 S., Br.

DM 59,–

Band 7:
AUTOMATEN UND SENSOREN
ZUM SELBERBAUEN FÜR
COMMODORE COMPUTER
von John Billingsley
ISBN 3-89 133-007-3
(Artikel-Nr. 58 00 05)
1984, 128 S., Br.

DM 24,80

Band 8
MATHEMATIK MIT DEM
COMMODORE 64
von Czes Kosniowski
mit 1 Diskette
ISBN 3-89 133-008-1
(Artikel-Nr. 55 64 30)
1984, 166 S., Br.

DM 34,80

Band 9:
PROGRAMMIERTECHNIKEN FÜR
FORTGESCHRITTENE AUF DEM
COMMODORE 64
von David Lawrence
ISBN 3-89 133-009-X
(Artikel-Nr. 55 64 35)
1985, 184 S., Br.

DM 29,80

Band 10:
DER COMMODORE 64 ALS GRAFIK-KÜNSTLER
von Boris Allan
ISBN 3-89 133-010-3
(Artikel-Nr. 55 64 38)
1985, 144 S., Br.
DM 24,80

Band 11:
DER COMMODORE 64 IN DER PRAXIS
von David Lawrence
ISBN 3-89 133-011-1
(Artikel-Nr. 55 64 23)
1985 in Druck

Band 12:
PROGRAMMIERUNG IN MASCHINENSPRACHE AUF DEM COMMODORE 64: DAS WERKZEUG
von Mark England und David Lawrence
ISBN 3-89 133-012-X
(Artikel-Nr. 55 64 25)
1985 in Druck

Band 13:
PROGRAMMIERUNG IN MASCHINENSPRACHE AUF DEM COMMODORE 64: GRAFIK UND MUSIK
von Mark England und David Lawrence
ISBN 3-89 133-013-8
(Artikel-Nr. 55 64 26)
1985 in Druck

Band 14:
PROGRAMMIERUNG IN MASCHINENSPRACHE AUF DEM COMMODORE 64: SPIELE
von Paul Roper
ISBN 3-89 133-014-6
(Artikel-Nr. 55 64 27)
1985 in Druck

Band 15:
KÜNSTLICHE INTELLIGENZ AUF DEM COMMODORE 64
von Keith und Steven Brain
ISBN 3-89 133-015-4
(Artikel-Nr. 55 64 31)
1985 in Druck

Band 16:
FLOPPY-PROGRAMMIERUNG MIT DEM COMMODORE 64
von David Lawrence und Mark England
ISBN 3-89 133-016-2
(Artikel-Nr. 55 64 28)
in Vorbereitung

Band 17:
STRATEGIESPIELE AUF DEM COMMODORE 64
EINE PROGRAMMIERANLEITUNG
ISBN 3-89 133-017-0
(Artikel-Nr. 55 64 32)
in Vorbereitung

Band 18:
DER COMMODORE 16 IN DER PRAXIS
ISBN 3-89 133-018-9
(Artikel-Nr. 55 50 00)
in Vorbereitung

Preisänderungen vorbehalten
☐ Unverbindliche Preisempfehlung

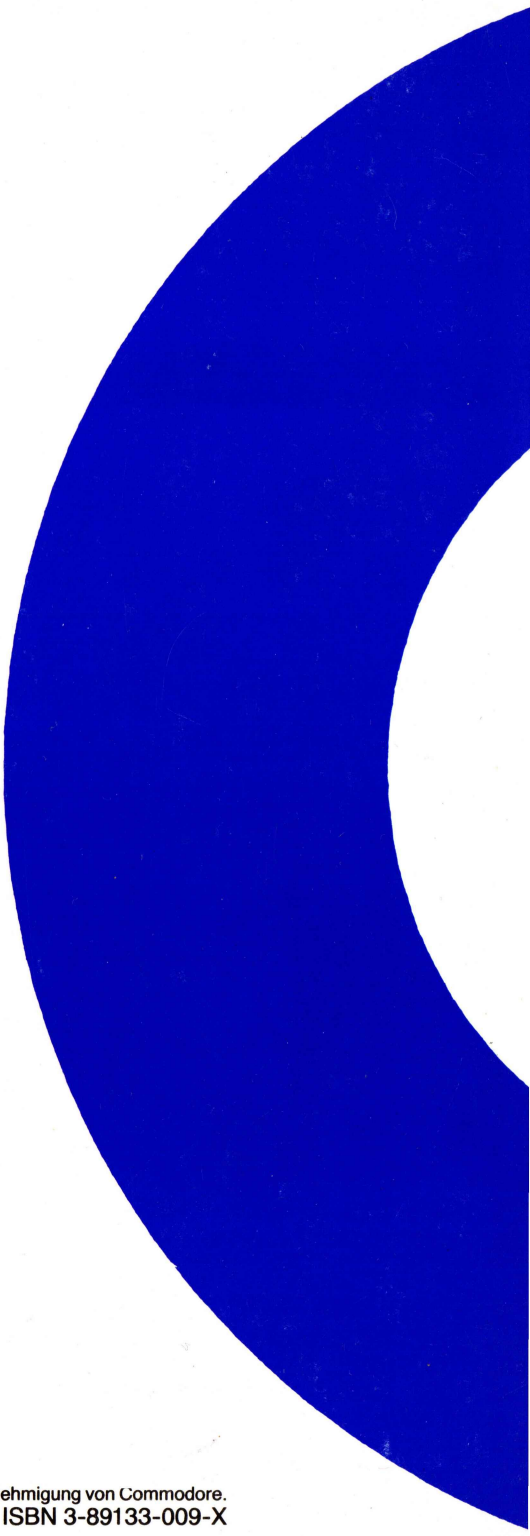
Commodore
EINE GUTE IDEE NACH DER ANDEREN

Dies ist ein Buch für alle, die mit dem Commodore 64 (und dem damit voll compatiblen Commodore 128) endlich ernsthaft programmieren wollen. Es analysiert einige der Techniken, die man zum Schreiben erfolgreicher Anwendungsprogramme braucht.

Das Buch ist randvoll mit Ratschlägen und Programmbeispielen von Einzeilern bis hin zu komplexeren Routinen, die eine wesentlich effizientere Programmiertechnik ermöglichen.

Wenn Sie die hier vorgestellten Methoden anwenden, werden Sie bald Programme schreiben können, die besser, schneller und klarer sind und sich ökonomischer und sicherer speichern lassen.

David Lawrence ist einer der erfolgreichsten Autoren zum Thema Mikrocomputer in Großbritannien. Sein Name steht für eine Reihe von Bestsellern. Neben seiner Tätigkeit als Buchautor schreibt er auch kommerziell Software und verfaßt regelmäßig Artikel für die Zeitschrift 'Popular Computing Weekly'.



Commodore

Commodore GmbH
Lyoner Straße 38
D-6000 Frankfurt/M. 71

Commodore AG
Aeschenvorstadt 57
CH-4010 Basel

Commodore GmbH
Kinskygasse 40–44
A-1232 Wien

Artikel-Nr. 556435/4.85 Änderungen vorbehalten ISBN 3-89133-009-X