FRANZIS EINFÜHRUNG

Helbo

Turbo Pascal

Programmieren lernen mit der
idealen Sprache für Einsteiger

FANZIS EINFÜHRUNG

Lars J. Helbo

Turbo Pascal

Programmieren lernen mit der
idealen Sprache für Einsteiger

CIP-Titelaufnahme der Deutschen Bibliothek
Helbo, Lars J.:
Turbo Pascal: Programmieren lernen mit der idealen Sprache
für Einsteiger / Lars J. Helbo. - München: Franzis, 1990
(Franzis-Einführung)
ISBN 3-7723-6762-3

Umschlaggestaltung: Kaselow-Design, München
© 1990 Franzis-Verlag GmbH, München
Sämtliche Rechte - besonders das Übersetzungsrecht - an Text
und Bildern vorbehalten.
Fotomechanische Vervielfältigungen nur mit Genehmigung des
Verlages. Jeder Nach-
druck, auch auszugsweise, und jede Wiedergabe der
Abbildungen, auch in verändertem
Zustand, sind verboten.
Satz: Franzis - Druck GmbH (DTP), München
Druck: Druckerei Sommer, Feuchtwangen
Printed in Germany • Imprimé en Allemagne
ISBN 3-7723-6762-3

Vorwort

In den letzten Jahren hat Turbo Pascal unter den Benutzern der MS-DOS-kompatiblen Personal-Computer große Verbreitung und Beliebtheit erfahren.
Ziel dieses Buches ist es, dem Programmier-Neuling einen Einstieg in die Pro grammierung mit dieser Sprache zu verschaffen. Dabei soll es aber nicht darum gehen, möglichst viele Befehle auswendig zu lernen. Sie sollen vielmehr verstehen, wie ein Programm aufgebaut ist. In diesem Sinn ist das Buch nicht nur ein Einstieg in Turbo Pascal, sondern auch eine Einführung in die Arbeit des Programmierers schlechthin.
Zum besseren Verständnis enthält das Buch einige größere Beispielprogramme. Diese Programme sollten Sie nicht nur lesen, sondern am Computer durcharbeiten. Nur so können Sie zu einem optimalen Lernerfolg kommen.
Lars J. Helbo

Wichtiger Hinweis:

Die Informationen in diesem Buch werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht. Warennamen sind nicht besonders gekennzeichnet und werden ohne Gewährleistung der freien Verwendbarkeit angegeben.
Bei der Erstellung des Buches wurde mit großer Sorgfalt vorgegangen; trotzdem können Fehler nicht vollständig ausgeschlossen werden. Verlag und Autor können für fehlerhafte Angeben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.
IBM-PC und PC-DOS sind eingetragene Warenzeichen von International Business Machines Corp. (IBM). Turbo Pascal ist eingetragenes Warenzeichen von Borland International.

Inhalt

Einführung 7

1 Installation 10

1.1 Sicherheitskopien 10
1.2 Programme und Dateien 11
1.3 Das Installationsprogramm 12
1.4 Arbeitsdisketten 13
1.5 Einstellungen im Programm 14

2 Bedienung des Turbo-Pascal-Programmiersystems 17

2.1 Das Menü-System 17
2.2 Bedienung des Editors 17
2.3 Bedienung des Compilers 20

3 Programmelemente 23

3.1 Die Sprache Pascal 23
3.2 Programmstruktur 24
3.3 Variablen 25
Numerische Typen 36
String Typen 27
Zusammengesetzte Typen 27
Selbstdefinierte Typen 29
3.4 Konstanten 29
3.5 Prozeduren und Funktionen 30
Globale und lokale Variablen 30
Parameter 32
3.6 Ausdrücke und Befehle 34
Operatoren 35
Zuordnung 36
Aufruf von Prozeduren und Funktionen 37
Programmschleifen 37
Sprünge 39

4 Das erste richtige Programm 41

4.1 Eine Maske für die Daten 41
4.2 Daten eingeben 45
4.3 Daten anzeigen 47
4.4 Ein Menü für das Programm 49
4.5 Kontrollfrage an den Benutzer 53
4.6 Datei öffnen 57
4.7 Datei schließen 59
4.8 Eingabe 59
4.9 Suchfunktionen 64
4.10 Druckerausgabe 76

5 Verbesserungen am Programm 69

5.1 Warnungen 69
5.2 Ein richtiger Editor 70
5.3 Einneues Menü 74
5.4 Verzeichnis wechseln 78
5.5 Dateiauswahl 80
5.6 Weitere Möglichkeiten 83

6 Eigene Units 86

7 Grafikprogrammierung 94

Grafikmodus wählen 94
Ein Beispielprogramm 97
Eingabe von Zahlen 100
Kurvendiagramm 101
Balkendiagramm 104
Tortendiagramm 106

8 Turtlegrafik 108

9 Objektorientierte Programmierung mit Turbo-Pascal 5.5 116

9.1 Nachteile der strukturierten Programmierung 116
9.2 Vererbung 118
9.3 Kapselung 119
9.4 Polymorphie 121

10 Fehlersuche und -behandlung 125

10.1 Programmierfehler 125
Der Debugger 127
10.2 Laufzeitfehler 129

Anhang 132

A: Standard-Prozeduren und Funktionen 132

1.Numerische Operationen 134
2.Typenkonvertierung 135
3. Stringoperationen 135
4.1/O-Operationen 135
5. Speicher-Verwaltung 137
6. Programmablauf 137
7. Systemnahe Programmierung 137
8.Grafik-Grundeinstellungen 138
9. Farben und Muster 139
10.Figuren 139
11. Text im Grafikmodus 140
12. Fehlerbehandlung 140

B: Compilerbefehle 140

1. Globale Einstellungen 140
2. Lokale Einstellungen 141
3. Dateien einbinden 141

C: Tastaturkommandos im Editor 141

D: ASCII-Zeichentabelle 143

E: Erweiterte Tastaturcodes 146

F: Erklärung einiger Begriffe 147

Sachverzeichnis 151

Einführung

Warum eine Programmiersprache?
Ein Computer ist bekanntlich eine Maschine zur Verarbeitung von Daten.
Er kann Daten annehmen, sie nach vorgegebenen Regeln behandeln und die Ergebnisse wieder ausgeben.

Dieser Prozeß ist in hohem Maße durch die Kommunikation zwischen Benutzer und Rechner bestimmt. Diese Kommunikation zwischen Mensch und Maschine wird aber leider dadurch behindert, daß der Computer nicht die menschliche Sprache versteht.

Die Anweisungen des Menschen an den Computer in Form von Programmzeilen müssen also zuvor übersetzt werden, damit sie der Rechner versteht. Dieses "Dolmetschen" geschieht entweder durch einen Interpreter oder durch einen Compiler. Was ist nun der Unterschied?

Ein *Interpreter* übersetzt das Programm Zeile für Zeile während der Laufzeit. Er liest eine Zeile aus dem Quellprogramm, übersetzt sie in Maschinensprache, läßt den Computer den Befehl ausführen, liest die nächste Zeile und so weiter.

Interpreter sind bei Anfängern sehr beliebt, weil man damit sehr schnell zu einem sichtbaren Ergebnis kommt. Es reicht, einige Zeilen in den Interpreter zu schreiben und den Befehl *Run* aufzurufen, schon kann man die Reaktion des Computers erkennen. Der Nachteil der Interpreter liegt vor allem in der niedrigen Geschwindigkeit. Weil der Interpreter ständig Rechenzeit für Übersetzungsarbeit beansprucht, bleibt für die eigentliche Programmausführung nur wenig Zeit übrig, und das Programm läuft entsprechend langsam.

Mit einem *Compiler* wird das gesamte Quellprogramm dagegen vollständig in Maschinensprache übersetzt und abgespeichert. Ein compiliertes Programm ist oft 10 - 20 mal schneller als ein vergleichbares Programm, das mit einem Interpreter läuft. Zusätzlich spart man Speicherplatz, weil der Compiler nicht wie der Interpreter ständig im Speicher sein muß, und man kann das fertige Programm auch an jemanden weitergeben oder verkaufen, der den Compiler nicht besitzt. Alle professionellen Programme, die zum Kauf angeboten werden, sind mit einem Compiler geschrieben.

Ein Compiler-System - wie auch Turbo-Pascal eines ist - besteht im wesentlichen aus drei verschiedenen Programmen: *Editor, Compiler* und *Linker.* Der Editor ist

eine Art Textverarbeitungsprogramm, mit dem das Quellprogramm geschrieben wird. Mit dem Compiler werden die einzelnen Teile des Quellprogramms in Maschinensprache übersetzt, mit dem Linker werden diese Teile dann zum fertigen Programm zusammengebunden.

Zuerst wird also das gesamte Quellprogramm geschrieben und abgespeichert. Dann wird der Compiler gestartet, jede Quellprogramm-Datei wird wieder geladen, compiliert und nochmals abgespeichert.

Manche Compiler brauchen sogar mehrere (bis zu 6) Durchläufe, auch "Passes" genannt, und jedesmal muß der Code geladen und wieder abgespeichert werden. Wenn alle Teile endlich compiliert sind, muß der Linker gestartet werden. Die einzelnen Programmteile werden geladen, zusammengefügt und als fertiges Programm abgespeichert. Danach kann das Programm auch direkt ohne den Compiler gestartet werden, allerdings nur, wenn das Quellprogramm fehlerfrei war. Wer einmal ein Programm geschrieben hat, weiß aber, daß dies wahrscheinlich nicht der Fall ist. Also zurück zum Editor: Fehler finden und korrigieren, wieder compilieren, linken und erproben etc. etc.

Man hat also die Wahl zwischen einem Interpreter mit einfacher Programmierung, aber langsamem Programmablauf, und einem Compiler mit komplizierter Programmierung, aber schnellem Programmablauf. Oder besser gesagt, man hatte die Wahl bis 1982, als ein junger Mann aus Kopenhagen, Anders Hejlsberg, die Idee zu einem ganz besonderen Pascal-Compiler hatte. Der Compiler wurde später an die amerikanische Software-Firma Borland (in der Bundesrepublik Deutschland durch die Firma Heimsoeth vertreten) verkauft und hat seitdem in der Welt der PCs eine Verbreitung und Beliebtheit wie kein anderer Compiler erreicht.

Dieser Erfolg hat mehrere Ursachen. Die wichtigste liegt in den besonderen Eigenschaften von Turbo Pascal. Im Gegensatz zu anderen Compiler-Systemen sind bei Turbo Pascal Editor, Compiler und Linker in einem Programm vereint. Dadurch bleiben alle Teile des Systems ständig im Speicher und müssen nicht immer wieder nachgeladen werden.

Im Speicher bleiben außerdem sowohl das Quellprogramm als auch das fertig compilierte Programm. Dadurch wird es möglich, den gesamten Prozeß von Editierung bis Erprobung (fast) ohne Disketten-Operationen und damit sehr schnell durchzuführen. Turbo Pascal gestattet eine komfortable und schnelle Programm-Entwicklung, die durchaus mit der Nutzung eines Interpreters vergleichbar ist. Darüberhinaus ist Turbo Pascal ein vollwertiges Compiler-System, und die fertigen Programme erreichen beachtliche Geschwindigkeiten.

Im Gegensatz zu den traditionellen Pascal-Versionen ist Turbo Pascal ein integriertes Entwicklungssystem, bei dem ein Texteditor und ein Compiler in eine

homogene Arbeitsumgebung mit einer für den Benutzer attraktiven Benutzeroberfläche eingebunden wurden. Der Editor dient zur Erfassung des Programmtextes (auch Quelltext genannt), der Compiler übersetzt anschließend das Quellprogramm in Maschinensprache und verknüpft es mit den für den Ablauf erforderlichen internen Programmroutinen. Die Entwicklung des Programmtextes, dessen Übersetzung und anschließender Start werden so zu einem Kinderspiel. Die Idee dazu hatte übrigens der bereits erwähnte Anders Heilsberg, der im Jahr 1982 sein Programmsystem an das Software-Haus Borland (hierzulande vertreten durch die Firma Heimsoeth) verkaufte.

Turbo Pascal wurde im Laufe der Jahre mehrfach erweitert und verbessert. Die bei Drucklegung dieses Buches aktuelle Version 5.5 ist mit den Versionen 4.0 und 5.0 weitgehend kompatibel. So können Sie z.B. fast alle Beispielprogramme dieses Buches mit jeder der drei Versionen nachvollziehen.

Die wesentlichen Neuerungen sind ein eingebauter Debugger in den Versionen 5.0 und 5.5 und die sogenannte "objektorientierte Programmierung" (OOP) in Version 5.5. Der Debugger wird in Kapitel 10.1 näher beschrieben, und was OOP ist erfahren Sie in Kapitel 9.

Das Ziel dieses Buches ist es, eine grundlegende Einführung in die Programmierung mit Turbo Pascal zu geben. Außerdem finden Sie am Ende des Buches einige Anhänge mit Informationen, die während des eigenen Programmierens wichtig sind. Hier zunächst eine Übersicht über die einzelnen Kapitel:

Kapitel l zeigt, welche Dateien auf den Originaldisketten vorhanden sind, außerdem wird erklärt, wie der Compiler installiert und gestartet wird.

Kapitel 2 macht Sie mit der Bedienung von Editor und Compiler bekannt. Kapitel 3 beschreibt, aus welchen Teilen ein Programm besteht. Dazu gehören unter anderem Variablen, Konstanten, Operanden und Operatoren.

Kapitel 4 und 5 zeigen anhand eines größeren Beispiels, wie ein Programm aufgebaut wird. Gleichzeitig werden Sie mit einigen der in Turbo Pascal enthaltenen Standard-Prozeduren und -Funktionen bekannt gemacht.

Kapitel 6 erklärt, wie das Turbo Pascal-System durch sogenannte "Units" erweitert werden kann. Ein Unit ist eine Sammlung von Prozeduren und Funktionen. Kapitel 7 und 8 erläutern den Aufbau weiterer Beispielprogramme. Diesmal geht es primär um das Programmieren von Grafik. Dazu ist ein Computer mit eingebauter Grafikkarte erforderlich.

Kapitel 9 beschäftigt sich mit der oben erwähnten "objektorientierten Programmierung" und richtet sich somit speziell an Besitzer der neuen Version 5.5. Kapitel 10 gibt schließlich einige Ratschläge zur Fehlersuche und -beseitigung.

1 Installation

1.1 Sicherheitskopien

Im Vergleich mit anderen Softwareprodukten wird Turbo Pascal zu einem sehr niedrigen Preis verkauft und ohne Kopierschutz geliefert. Für den Benutzer ist das ein großer Vorteil, dadurch kann er nämlich ohne Probleme Sicherheitskopien erstellen.

Über die Wichtigkeit von Datensicherheit und Sicherheitskopien (was letzendlich ein und dasselbe ist!) wurde im Laufe der Zeit sehr viel geschrieben. Trotzdem gibt es immer noch Programmierer, die glauben, es reiche aus, "Qualitätsdisketten" zu benutzen und diese vorsichtig zu behandeln und aufzubewahren.

Die Erfahrung lehrt aber: Trotz vieler Vorsichtsmaßnahmen passiert es immer wieder, daß eine Diskette beschädigt oder gar unbrauchbar geworden ist. Oftmals sind die daraufgespeicherten Daten dann unrettbar verloren. Es sei denn, man hat vorher eine Sicherheitskopie angelegt.

Bevor Sie in diesem Buch fortfahren, sollten Sie also unbedingt Sicherheitskopien von Ihren Turbo Pascal-Originaldisketten anfertigen. Dazu benutzen Sie den DOS-Befehl Diskcopy. Dieser Befehl wird von der Systemdiskette bzw. der Festplatte geladen.

Wenn Sie keine Festplatte besitzen, müssen Sie deshalb als erstes Ihre Systemdiskette in Laufwerk A: legen. Geben Sie dann folgendes ein:

```
diskcopy   a:   b:
```

Bei einem Computer mit zwei Diskettenlaufwerken wird die erste Originaldiskette jetzt in Laufwerk A: und eine leere Diskette in Laufwerk B: gelegt. Hat der Computer dagegen nur ein Laufwerk, wird zunächst nur die erste Originaldiskette eingelegt, der Computer liest diese und zeigt dann auf dem Bildschirm eine Aufforderung, die leere Diskette einzulegen. Nachdem die Diskette kopiert ist, fragt der Computer, ob weitere Disketten kopiert werden sollen. Antworten Sie mit "J" für ja (oder "Y" für yes je nach Betriebssystem-Version) und wiederholen Sie den Vorgang mit der nächsten Diskette. Wenn Sie alle Originaldisketten kopiert haben, legen Sie sie an einen sicheren Ort (am besten in einem anderen Zimmer) und benutzen Sie für das Folgende ausschließlich die Kopien.

1.2 Programme und Dateien

Turbo Pascal wird in verschiedenen Versionen (4.0, 5.0 und 5.5), Sprachen (deutsch und englisch) und auf verschiedenen Datenträgern (3,5 und 5,25 Zoll Disketten) verkauft. Deshalb kann die Anzahl der Dateien und ihre Verteilung auf den einzelnen Disketten unterschiedlich sein. Die Namen der Dateien sind aber immer dieselben.

Unter MS-DOS können Sie die verschiedenen Datentypen an den Endungen der Dateinamen erkennen. Hier ist ein zentraler Programmtyp die sogenannte *Programmdatei,* welche man leicht an den Dateinamen-Ergänzungen .com oder .exe erkennt. Auf den Disketten gibt es je nach Version zwischen 12 und 14 verschiedene Programmdateien, wie z.B. den Compiler und verschiedene Hilfs- und Installations-Programme.

Den Compiler bekommen Sie sogar gleich zweimal: einmal mit dem Namen Turbo.exe und einmal als Tpc.exe. Der erste ist die integrierte Version, bei der Editor, Compiler und Linker in einem Programm zusammengefaßt sind. Diese Version hat auch eine moderne Benutzeroberfläche mit Pull-Down-Menüs und Fenster.

Tpc.exe ist dagegen eine Kommandozeilen-orientierte Version. Das heißt, Sie müssen das Quellprogramm mit einem Editor (der nicht mitgeliefert wird) schreiben und auf Diskette abspeichern. Danach wird der Compiler mit dem Dateinamen des Quellprogramms als Parameter gestartet.

Auf den ersten Blick ist nicht einzusehen, warum man auf den Komfort der integrierten Programmierumgebung verzichten soll. Er ist ja gerade die Stärke von Turbo Pascal. Der einzige erkennbare Grund liegt in dem unterschiedlich hohen Speicherbedarf der beiden Versionen. Für die integrierte Version braucht man einen Computer mit 384 KByte RAM-Speicher, während die Kommandozeilen-orientierte Version auf einem System mit nur 256 KByte laufen kann.

Wer einen Computer mit weniger als 384 KByte RAM hat, sollte aber ernsthaft überlegen, ob es sich nicht lohnen würde, zusätzliche Speicherchips zu kaufen. Es wäre ja auch unwirtschaftlich, die Software nicht voll ausnutzen zu können. Ein größerer Speicher ist darüberhinaus auch in Verbindung mit anderen Programmen von Vorteil.

Die übrigen .com- und .exe-Dateien sind Hilfsprogramme. Wenn Sie mit Turbo.exe arbeiten, sind sie aber zumindest für den Anfang unwichtig. Wir werden später auf einige dieser Hilfsprogramme zurückkommen.

Unentbehrlich sind dagegen die sogenannten *Units.* Das sind Sammlungen von Unterprogrammen, die man in seine eigenen Programme einbinden kann. Turbo Pascal wird zur Zeit mit 7 oder 8 Units geliefert. Die wichtigsten davon befinden

sich in der Datei Turbo.tpl. Andere Units, die seltener benutzt werden, befinden sich in eigenen Dateien. Diese können Sie an der Endung .tpu erkennen.
Die Dateien mit der Endung .pas sind Programmbeispiele. Um eine gute Übersicht zu erreichen, sollten alle Pascal-Quellprogramme diese Endung haben, egal ob sie für Turbo Pascal oder einen anderen Compiler geschrieben sind.
Dann gibt es Dateien mit der Endung .doc. Dazu zählen einige Dateien, die zum Hilfssystem des Compilers gehören. Die meisten sind aber Quellprogramme der eben genannten Units.

Schließlich gibt es noch Dateien mit den Endungen .bgi und .ehr. BGI steht für "Borland Graphics Interface". Diese Dateien brauchen Sie, wenn Sie Programme in Grafik-Modus schreiben wollen. Sie enthalten Treiber-Programme für die verschiedenen Grafik-Karten, mit denen ein PC ausgerüstet sein kann, .ehr steht dagegen für "Character". chr-Dateien enthalten Zeichensätze für den Grafikmodus.

Bei Version 5.x kann es aber auch sein, daß auf den Disketten keine .pas-, .doc-, .bgi- oder .chr-Dateien zu finden sind. Dann gibt es stattdessen auf einer der Turbo Pascal-Disketten einige Dateien mit der Endung .arc. Diese Endung ist die Abkürzung für "Archiv". Die .arc-Dateien enthalten die oben genannten Dateien in komprimierter Form.

Bevor solche Dateien benutzt werden können, müssen sie in das normale Format umgesetzt werden! Das erledigt das Programm Unpack.com. Hierfür muß zuerst ausreichend Platz geschaffen werden; die umgesetzten Dateien brauchen etwa doppelt so viel Platz wie die komprimierten. Mit anderen Worten: Sie müssen zwei Disketten formatieren. Danach wird die Kopie der Originaldiskette in Laufwerk A: und eine leere Diskette in Laufwerk B: gelegt (die Originaldiskette ist ja längst an einem anderen Ort in Sicherheit gebracht!). Dann wird Laufwerk A: zum aktuellen Laufwerk gemacht, und Unpack.com wird mit folgendem Kommando gestartet:

```
unpack Dateiname B:
```

"Dateiname" ist der Name der .arc-Datei, die umgesetzt werden soll, wobei die Endung .arc weggelassen werden darf. Wer eine Festplatte besitzt, könnte statt Laufwerk B: auch ein Verzeichnis auf der Festplatte angeben, mit dem Installationsprogramm geht es aber noch einfacher.

1.3 Das Installationsprogramm

Bei Version 5.x bekommen Sie zwei Installationsprogramme: das Programm Install.exe, das die verschiedenen Dateien zur Festplatte oder Arbeitsdiskette ko-

piert, und Tinst.exe, mit dem einige Einstellungen im Compiler geändert werden können. Bei Version 4.0 ist nur das letztere enthalten.

Haben Sie keine Festplatte, ist es wenig sinnvoll, das Programm Install.exe zu benutzen. Das Programm kopiert lediglich Turbo.exe und Turbo.tpl auf eine leere Diskette. Das können Sie mit dem Befehl *copy* einfacher erledigen.

Haben Sie einen Computer mit Festplatte, ist das Programm Install.exe dagegen sehr hilfreich. Es kopiert dann sämtliche Dateien von den Disketten zur Festplatte und verteilt sie in bis zu fünf verschiedene Unterverzeichnisse. Gleichzeitig werden die .arc-Dateien automatisch umgesetzt.

Wenn das Programm gestartet wird, müssen Sie zuerst angeben, in welchem Laufwerk die Disketten sind und in welche Unterverzeichnisse die Dateien hineinkopiert werden sollen. Das Programm ist weitgehend selbsterklärend; Sie sollten nur darauf achten, daß das erste Unterverzeichnis das Hauptverzeichnis für die vier anderen ist und daß das Programm die Verzeichnisse auch anlegt. Sie müssen also nur dafür sorgen, daß auf der Festplatte genug freier Platz vorhanden ist.

Mit dem Programm Tinst.exe können Sie einige Funktionen im Compiler ändern. Das kann natürlich ganz nützlich sein; es ist aber keineswegs notwendig, eigens eine Installation durchzuführen. Fast alle diese Einstellungen können auch direkt im Compiler (nur bei Turbo.exe) geändert werden.

Ich möchte deshalb empfehlen, das Programm erstmal so zu lassen, wie es ist. Später, wenn Sie Ihre Erfahrungen mit dem Compiler gemacht haben, können Sie jederzeit das Installationsprogramm hervorholen und den Compiler Ihren Wünschen anpassen.

1.4 Arbeitsdisketten

Wenn Sie eine Festplatte besitzen, können Sie ganz einfach alle Dateien in ein oder mehrere Unterverzeichnisse kopieren, bei Version 5.x am einfachsten mit Install.exe. Trotzdem sollten Sie überlegen, welche Dateien Sie wirklich brauchen. Viele sind überflüssig und selbst die größte Festplatte wird irgendwann einmal voll.

Bei einem Computer ohne Festplatte muß eine Arbeitsdiskette mit den notwendigen Dateien zusammengestellt werden. Wie Sie Ihre Arbeitsdiskette zusammenstellen wollen, hängt zum einen Teil davon ab, welche Programme Sie schreiben wollen, und zum anderen, welcher Diskettentyp benutzt wird, und somit wieviel Platz zur Verfügung steht. Ich kann also nur einige Richtlinien geben, anach muß jeder die Größen der Dateien (in der vorliegenden Version) zusammenzählen und seine Entscheidung treffen.

Unbedingt erforderlich sind nur die Dateien Turbo.exe und Turbo.tpl. Diese Dateien sind in Version 4.0 insgesamt 152 KByte, in Version 5.0182 KByte und in Version 5.5 200 KByte groß.

Wünschenswert wäre darüberhinaus die Datei Turbo.hlp. Diese Datei enthält einige Hilfstexte, die während der Arbeit mit dem Compiler über die Funktionstaste **Fl** abgerufen werden können. Die Datei ist aber in Version 4.0 über 80 KByte groß, in Version 5.0 fast 170 KByte und in Version 5.5 über 240 KByte.
Wenn Sie Grafik programmieren wollen, brauchen Sie auch noch die Datei Graph.tpu, mindestens eine .bgi-Datei und eine oder mehrere .chr-Dateien. Schließlich wäre es schön, wenn die Arbeitsdiskette auch als Systemdiskette mit Command.com formatiert wäre.

Spätestens hier muß klar sein, daß man zumindest bei Version 5.x nicht alle Wünsche innerhalb von 360 KByte unterbringen kann. Mein Vorschlag ist, eine Systemdiskette zu formatieren (mit Format B: /S), und dann Turbo.exe, und Turbo.tpl auf diese Diskette zu kopieren. Das reicht für den Anfang und läßt sich natürlich jederzeit ändern.

1.5 Einstellungen im Programm

Jetzt ist es endlich Zeit, Turbo Pascal zu starten. Legen Sie die gerade erstellte Arbeitsdiskette in das aktuelle Laufwerk bzw. machen Sie das Turbo Pascal-Verzeichnis auf der Festplatte zum aktuellen Verzeichnis. Geben Sie dann folgendes ein:

`turbo` **Return**

Nachdem das Programm in den Hauptspeicher geladen ist, meldet es sich mit dem Bild in *Abb. 1.*

In der Mitte des Bildschirms sehen Sie eine Copyright-Meldung der Firma Borland. Darin steht u. a. die Versions-Nummer des Programms. Wenn Sie eine beliebige Taste betätigen, verschwindet die Copyright-Meldung und Sie sehen auf dem Bildschirm oben das Hauptmenü und darunter die beiden Fenster.
Der erste Punkt ("File") im Hauptmenü ist hervorgehoben. Mit den Pfeiltasten rechts/links kann man das helle Feld nach links und rechts bewegen und dabei die anderen Punkte im Hauptmenü hervorheben.

Mit der Taste **Return** können Sie den hervorgehobenen Menüpunkt auswählen, wobei in den meisten Fällen ein Untermenü erscheint. Mit den Tasten **Esc** oder

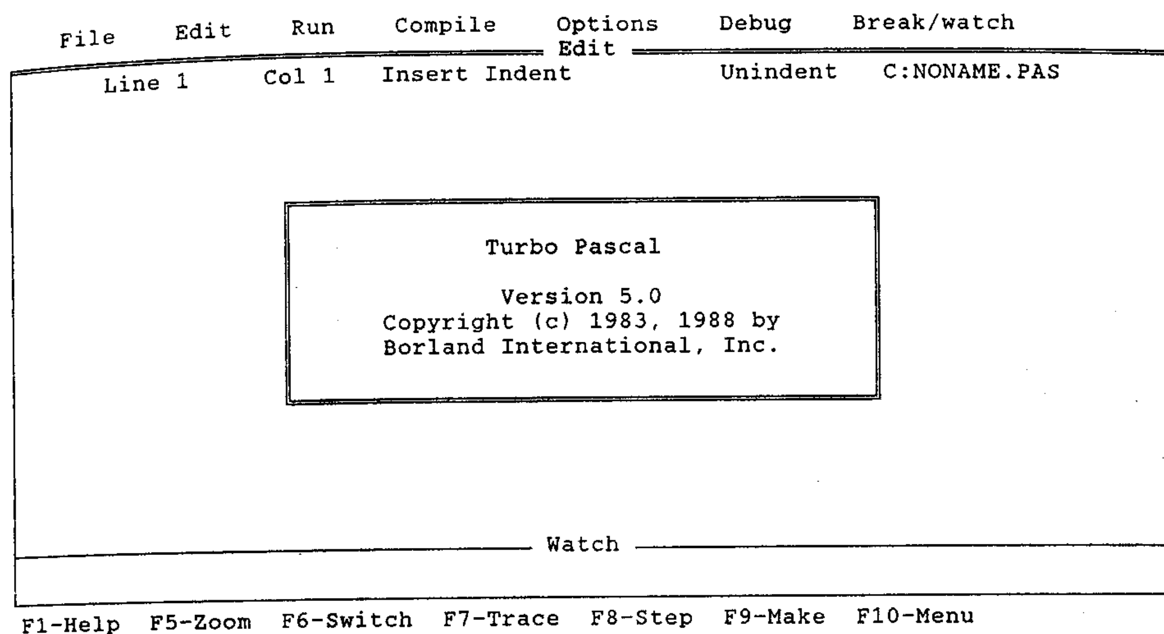

Abb. 1 Turbo Pascal mit der Hauptmenüzeile und den zwei Fenstern "Edit" und "Waten". Bei Version 4.0 fehlen die beiden Menü-Punkte "Debug" und "Break/ Watch", und das untere Fenster trägt den Namen "Output".

F10 kommen Sie zum Hauptmenü zurück. Wenn Sie genau hinsehen, erkennen Sie, daß der erste Buchstabe in jedem Wort des Hauptmenüs etwas heller ist. Mit diesem Buchstaben kann der Menüpunkt direkt ausgewählt werden. Anstatt das hervorgehobene Feld mit den Pfeiltasten beispielsweise zu dem Wort "Compile" zu bewegen und danach **Return** zu drücken, können Sie auch einfach C drücken (oder c).

Wählen Sie j etzt zuerst den Menüpunkt" Options". Entweder drücken Sie so lange die Pfeiltaste, bis das Wort "Options" hervorgehoben ist und danach **Return** oder Sie drücken einfach O.

Mit den verschiedenen Alternativen in dem Untermenü können Sie fast alle Einstellungen ändern wie mit dem Installationsprogramm Tinst.exe. Wie gesagt, können diese Einstellungen zunächst so gelassen werden, wie sie sind. Wer mit einer Festplatte arbeitet, sollte sich aber den Untermenüpunkt "Directories" ein wenig näher ansehen:

Ein Untermenüpunkt wird genauso ausgewählt wie ein Punkt im Hauptmenü, nur müssen Sie hier logischerweise die Pfeiltasten nach oben und nach unten benutzen. Der Untermenüpunkt "Directories" wird also ausgewählt, indem man dreimal die Taste Pfeil nach unten und dann einmal **Return** oder einfacher einmal D drückt. Dabei erscheint noch ein Untermenü. Hier können Sie nun festlegen, in welchen Verzeichnissen Turbo Pascal nach den verschiedenen Dateien suchen

soll. Bei einem Computer mit Diskettenlaufwerken ist dies unwesentlich, weil man dort fast immer im Hauptverzeichnis arbeiten wird. Auf einer Festplatte ist es aber sehr wichtig. Jedes Verzeichnis kann mit komplettem Pfad inklusive Laufwerk angegeben werden.

Im Rahmen des ersten Punktes "Turbo directory" wird das Verzeichnis angegeben, wo sich das Programm Turbo.exe befindet. In diesem Verzeichnis müssen auch die Dateien Turbo.tpl und Turbo.hlp enthalten sein. Der zweite Punkt "EXE & TPU directory" gibt das Verzeichnis an, in dem fertig compilierte Programme abgelegt werden sollen.

Wenn ein Programm compiliert wird, ist es möglich, fremde Quelldateien, Teile eines Units oder Assemblerdateien in das Programm einzubinden. Erhält der Compiler einen solchen Befehl, sucht er zuerst nach den fremden Dateien im aktuellen Verzeichnis. Ist die Suche hier erfolglos, sucht er in den Verzeichnissen, deren Namen in den folgenden drei Menüpunkten angegeben sind.

Unter "Include directories" müssen daher die Verzeichnisse angegeben werden, in denen Pascal-Quellprogramme (mit der Endung .pas) abgespeichert werden. Unter "Unit directories" schreiben Sie die Verzeichnisse, in denen .tpu-Dateien zu finden sind. Wollen Sie auch noch Assembler-Dateien (mit der Endung .obj) in Ihre Programme einsetzen, müssen die dazugehörigen Verzeichnisse unter "Object directories" angegeben werden.

Es wäre natürlich umständlich, wenn man alle diese Eintragungen jedesmal machen müßte. Muß man aber auch nicht. Drücken Sie einmal auf die Taste Esc, womit Sie in das erste Untermenü unter Options zurückgelangen.

Die zwei letzten Punkte in diesem Menü heißen "Save options" und "Retrieve options" oder "Load options". Mit Save Options kann man die Einstellungen abspeichern. Dazu müssen Sie einen Dateinamen angeben. Geben Sie Turbo.tpl an und speichern Sie diese Datei in dasselbe Verzeichnis wie Turbo.exe und Turbo.tpl, werden die Einstellungen jedesmal automatisch geladen.

Sie können auch als Dateinamen Turbo.exe angeben. Dann werden die Einstellungen direkt in die Programm-Datei geschrieben. Diese Möglichkeit sollte mit etwas Vorsicht verwendet werden und niemals auf der Originaldiskette. Schließlich können Sie einen völlig anderen Dateinamen angeben, aber diese Datei muß dann jedesmal mit "Retrieve options" bzw. "Load options" geladen werden.

2 Bedienung des Turbo-Pascal-Programmiersystems

2.1 Das Menü-System

Nachdem Sie den Compiler installiert und angepaßt haben, können sie anfangen, Ihre ersten Programme zu schreiben. Dafür müssen Sie aber wissen, wie Editor und Compiler bedient werden.Die wichtigsten Funktionen werden über verschiedene Menüs aufgerufen. Die Menüfunktionen teilen sich in drei Bereiche auf:
1. Die im letzten Kapitel besprochenen Funktionen für Grundeinstellungen unter dem Menü "Options".
2. Die Funktionen, die in Zusammenhang mit dem integrierten Debugger stehen (nur in Version 5.x). Man findet sie unter den Menüs "Run", "Debug" und "Break/Watch". Diese werden in Kapitel 10 näher besprochen.
3. Die Funktionen, die in direktem Zusammenhang mit der Bedienung von Editor und Compiler stehen. Sie sind in den vier Menüs "File", "Edit", "Run" und "Compile" untergebracht.

2.2 Bedienung des Editors

Jedes neue Programm wird zuerst als Quellcode im Edit-Fenster geschrieben. Man muß also vom Hauptmenü zum Edit-Fenster wechseln. Dazu wählen Sie aus der oberen Menüzeile den Punkt "Edit" aus. Also zuerst mit den Pfeiltasten das hervorgehobene Feld auf "Edit" schieben und **Return** oder die Taste E drücken. Dadurch verschwindet das hervorgehobene Feld im Hauptmenü und stattdessen leuchtet der Cursor im Edit-Fenster auf. Danach kann man gleich anfangen, ein Programm einzutippen. Vom Editor kann man mit der Tastenkombination **Ctrl+K+D** zum Hauptmenü zurückkommen.

Mit den Tasten F10 und **Esc** kann man auch zwischen Hauptmenü und Editor hin- und herwechseln. Die Funktion dieser Tasten ist in den verschiedenen Versionen von Turbo Pascal allerdings nicht einheitlich. Am besten probieren Sie die Möglichkeiten aus.

Wenn der Cursor im Edit-Fenster sichtbar ist, kann man anfangen, den Quellcode zu schreiben. Für den ersten Versuch ist hier ein kleines Pascal-Programm. Wenn Sie das Programm abtippen, sollten Sie darauf achten, daß alles wirklich so bleibt, wie es hier steht. Jeder Doppelpunkt und jedes Semikolon hat eine Bedeutung und darf nicht geändert werden, sonst erhalten Sie eine Fehlermeldung.

```
Program erster_versuch;
Uses Crt;
Const positiv : Char = 'J';
Var antwort : Char;
Begin
WriteLn('Möchten Sie Turbo Pascal lernen? (J/N)');
antwort:=UpCase(ReadKey);
If antwort=positiv Then
WriteLn('Das freut mich')
Else
WriteLn('Das tut mir leid');
End.
```

Wenn der Quellcode geschrieben ist, sollte man ihn erst einmal abspeichern. In dem Untermenü unter "File" gibt es zwei Funktionen: "Save" und "Write to". Der Text kann mit beiden Funktionen abgespeichert werden. Der Unterschied besteht darin, daß man bei "Write to" erst einen neuen Dateinamen angeben muß. Bei "Save" wird die Datei dagegen unter dem Namen abgespeichert, unter dem sie geladen wurde. Dieser Name steht immer in der oberen rechten Ecke des Edit-Fensters.

Ist es aber ein neuer Quellcode, der noch keinen Dateinamen hat, steht an dieser Stelle "NONAME.PAS". In diesem Fall wird auch bei "Save" dem Benutzer die Möglichkeit geben, einen neuen Dateinamen einzugeben. Das sollten Sie dann auch unbedingt tun. Der Dateiname kann frei gewählt werden, nach den normalen Regeln unter MS-DOS. Man sollte aber immer die Endung ".pas" benutzen.

Wenn Sie obenstehendes Beispiel eingegeben haben, können Sie es jetzt mit "Save" unter einem anderen Namen abspeichern. Wie wäre es mit "BEISP_1.PAS"? Eine Datei, die mit "Save" oder "Write to" abgespeichert wurde, kann mit "Load" oder "Pick" wieder in den Editor geladen werden. Wenn "Load" gewählt wird, erscheint ein kleines Fenster mit dem Text "*.PAS". In diesem Fenster kann man den Namen der Datei angeben, die geladen werden soll. Man kann aber auch gleich **Return** drücken. Dann erscheint ein größeres Fenster mit den Namen aller Dateien, die die Endung .pas haben. Wählen Sie jetzt die gewünschte Datei mit den Pfeiltasten aus und bestätigen Sie mit **Return.**

Wer ein größeres Programm schreibt, wird oft mit vielen Quelldateien gleichzeitig arbeiten. Um das ständige Nachladen der verschiedenen Dateien zu vereinfachen,

gibt es die Funktion "Pick". Wenn "Pick" gewählt wird, zeigt sich zuerst ein Fenster, ähnlich einem Untermenü. Als "Menüpunkte" stehen die Namen der 8 Dateien, die zuletzt geladen gewesen sind, und man kann die gewünschte Datei mit Cursortasten und **Return** auswählen. War die gewünschte Datei doch nicht darunter, kann man mit der letzten Zeile in diesem Fenster die Funktion "Load" erreichen, ohne "Pick" verlassen zu müssen.

Sie können jetzt Quelldateien schreiben, abspeichern und wieder laden. Im Menü "File" gibt es noch weitere interessante Funktionen. Mit "New" wird der Arbeitsspeicher des Editors gelöscht. Nachdem eine Quelldatei fertig geschrieben und abgespeichert ist, kann die Datei so vom Editor entfernt werden, bevor die nächste geschrieben wird. Ist die Datei noch nicht abgespeichert, wird darauf aufmerksam gemacht; man riskiert also nicht, unbeabsichtigt Quelltexte zu verlieren.

Im Handbuch ist angegeben, daß man mit der Funktion "Directory" den Inhalt eines Verzeichnisses ansehen könne. Das stimmt auch, ist aber nicht die ganze Wahrheit. In Wirklichkeit arbeitet diese Funktion genau wie "Load", d.h. Programme können ebenfalls über die Funktion "Directory" geladen werden. Die einzige Ausnahme ist, daß die Endung .pas nicht vorgegeben ist.

Die bisher genannten Funktionen arbeiten als Standard mit dem aktuellen Verzeichnis. Mit *cd* (change directory) kann dies aber geändert werden. Man muß einfach den gezeigten Pfad mit dem gewünschten neuen Pfad überschreiben und die Eingabe mit **Return** abschließen.

Stellen Sie sich nun einmal folgendes vor: Sie haben ein besonders schönes (und großes) Programm geschrieben. Gerade als Sie es abspeichern wollen, meldet der Computer: "Diskette voll", und beim Griff zur Diskettenablage entdecken Sie, daß dies ausgerechnet die letzte formatierte Diskette war.

Zugegeben, als umsichtiger Programmierer hält man für solche Notfälle natürlich immer mindestens 10 formatierte leere Disketten bereit. Alle anderen können sich aber mit der Funktion "OS shell" behelfen. Damit kann man DOS-Befehle aufrufen, ohne Turbo Pascal zu verlassen. Wer keine Festplatte besitzt, muß zuerst eine Systemdiskette in Laufwerk A: legen. Dann kann die Funktion gewählt werden. Man befindet sich jetzt in der DOS-Ebene und kann DOS-Funktionen ausführen oder andere Programme starten, z.B. um Disketten zu formatieren oder zu kopieren.

Wenn alles erledigt ist, müssen Sie einfach das Wort "exit" eingeben und mit **Return** abschließen. Schon sind Sie wieder in Turbo Pascal an der Stelle zurück, an der Sie das Programm verlassen haben, und alle geladenen Dateien sind immer noch im Speicher vorhanden.

Es gibt aber zwei Einschränkungen für diese Funktion, die nicht verschwiegen werden sollen. Erstens muß ausreichend Speicher vorhanden sein, es werden ja

gleichzeitig mehrere Programme im Speicher gehalten. Zweitens sollte man vorsichtig sein, welche Programme unter dieser Funktion gestartet werden. Wenn nämlich ein solches Programm einen Systemabsturz verursachen sollte, gehen dabei natürlich auch die nicht gespeicherten Turbo-Pascal-Dateien verloren.

Die letzte Funktion im Menü "File" heißt "Quit". Damit wird Turbo Pascal beendet.

Einige Menüfunktionen können auch mit Tasten oder Tasten-Kombinationen direkt erreicht werden. Wenn das der Fall ist, steht im Menü hinter der Funktion der Name der Taste. Nach "Save" steht z.B. F2. Das bedeutet, daß die Funktion Save auch mit der Taste F2 gerufen werden kann, und zwar unabhängig davon, welches Menü gerade aktiv ist.

Ob man eine Funktion mit dem Menü oder mit einer Taste ruft, ist vor allem Ansichtssache. Die Tasten sind schneller, müssen aber erst auswendig gelernt werden. Am Anfang ist es also einfacher, die Menüs zu benutzen. Wenn Sie aber viel programmieren, werden Sie wahrscheinlich früher oder später anfangen, die Tasten zu benutzen. Es gibt aber auch Funktionen, die nur mit Tasten-Kombinationen gerufen werden können. Im Anhang C finden Sie eine Übersicht über diese Funktionen. Es geht dabei um Funktionen, die nicht unbedingt notwendig sind, die aber das Schreiben von Quelltexten sehr viel einfacher und schneller machen können.

2.3 Bedienung des Compilers

Die Funktionen in den Menüs "File" und "Edit" haben also alle mit dem Editor zu tun. Der Compiler wird dagegen mit den Funktionen unter Run und Compile bedient.

Die erste Funktion im Menü "Compile" heißt auch Compile. Wenn Sie das Beispielprogramm abgeschrieben haben und diese Funktion wählen, wird das Programm compiliert. Dabei erscheint ein Fenster mit verschiedenen Informationen über das Programm.

Wenn der Quellcode fehlerfrei ist, endet die Compilierung mit der Meldung "Success", sonst springt das Programm zurück zum Editor und setzt den Cursor an den ersten gefundenen Fehler.

Besteht ein Programm aus mehreren Quelldateien, müssen diese erst einzeln compiliert und dann zu einem Programm zusammengebunden werden. Dieser Prozeß kann aber auch vereinfacht werden. Erst geben Sie den Namen der Hauptquelldatei unter dem Menüpunkt *Primary file:* an. Dann wählen Sie *Make.*

Die Funktion *Make* lädt erst die Hauptdatei in den Speicher, dann wird untersucht, welche weiteren Quelldateien zum Programm gehören und ob diese seit der letzten Compilierung geändert wurden. Die geänderten Dateien werden neu compiliert und das Ganze zu einem Programm zusammengebunden. Die Funktion *Build* arbeitet ähnlich wie *Make.* Der Unterschied besteht darin, daß hier sämtliche Dateien neu compiliert werden, egal ob es unbedingt notwendig ist oder nicht. Gehen wir kurz zum Menü *Run.* In Version 4.0 ist dies nur ein Menüpunkt wie *Edit.* In Version 5.x hat *Run* dagegen ein eigenes Untermenü bekommen. Die erste Funktion in diesem Untermenü heißt auch *Run* und entspricht dem Menüpunkt in Version 4.0.

Die Funktion *Run* ruft zuerst die Funktion *Make.* Gelingt es dabei, alle Quelldateien zu compilieren, wird das fertige Programm anschließend gestartet und ausgeführt. Dabei bleiben Turbo Pascal und die geladene Quelldatei im Speicher. Wenn das Programm abgeschlossen ist, kehrt man zum Editor zurück. Bevor *Run* gerufen wird, sollte man allerdings trotzdem die Quelldatei abspeichern. Es könnte ja sein, daß das Programm einen Fehler enthält und einen Systemabsturz verursacht.

Wenn Sie immer noch unser kleines Beispielprogramm im Editor haben, können Sie es jetzt mit *Run* zum Laufen bringen. Das Programm soll etwas auf den Bildschirm schreiben. In Version 4.0 geschieht dies in dem Output-Fenster am unteren Teil des Bildschirms. Man kann also gleichzeitig den Quelltext und das Ergebnis des Programms sehen.

In Version 5.x wird dagegen zwischen Editor und Output hin- und hergeschaltet. Wenn Run gerufen wird; verschwinden also Menüs und Edit-Fenster. Stattdessen sehen Sie jetzt den normalen DOS-Bildschirm. Darauf gibt das Programm den Text aus.

Das Beispiel-Programm stellt dem Benutzer als erstes eine Frage: "Möchten Sie Turbo Pascal lernen? (J/N)". Danach wartet das Programm, bis Sie Ihre Antwort eingegeben haben. Dann schreibt es einen Kommentar zu dieser Antwort. Damit ist das Programm zu Ende, und Turbo Pascal schaltet deshalb zurück zum Editor. In Version 4.0 kann man immer noch die Antwort im Output-Fenster lesen, in Version 5.x verschwindet sie aber so schnell, daß man keine Chance hat zu erfahren, was geschrieben wurde.

Das ist aber nicht weiter schlimm. Man kann nämlich jederzeit mit der Tastenkombination **Alt+F5** den letzten Output-Bildschirm zurückrufen. In Version 5.x gibt es im Menü Run einige weitere Funktionen. Sie betreffen aber alle die Bedienung des Debuggers und werden daher erst im Kapitel 10 besprochen.

Gehen wir jetzt zurück zum Compiler-Menü. Wenn Sie das Programm mit *Compile* oder *Run* compiliert haben, geschah es im Hauptspeicher des Rechners.

Irgendwann ist das Programm aber fertig, und alle Fehler sind beseitigt. Dann sollte das fertige Programm ja auch auf Diskette geschrieben werden, damit man es später von DOS aus starten kann, ohne erst Turbo Pascal laden zu müssen. Der vierte Menüpunkt unter "Compile" heißt "Destination". Destination heißt Ziel, und damit entscheiden Sie, wohin das compilierte Programm geschrieben werden soll. Die Standardeinstellung ist "Memory", also Hauptspeicher. Deshalb wurde das Beispielprogramm dorthin geschrieben. Wenn man diesen Menüpunkt wählt, ändert sich die Einstellung zu "Disk".

Versuchen Sie jetzt, das Beispielprogramm noch einmal zu compilieren, und zwar mit *Run, Compile, Make* oder *Build.* Verlassen Sie dann Turbo Pascal mit Quit im File-Menü und geben Sie den DOS-Befehl dir ein. Auf der Diskette sehen Sie jetzt eine neue Datei. Diese Datei hat denselben Namen wie das Beispielprogramm, aber mit der Endung .exe. Es ist also ein fertig ausführbares Programm, das von DOS aus gestartet werden kann.

3 Programmelemente

3.1 Die Sprache Pascal

In der Geschichte der Programmiersprachen gibt es einen Mann, der eine ganz besondere Rolle spielt: Niklaus K. Wirth, Professor an der Technischen Hochschule in Zürich. Seit Anfang der 60er Jahre ist er mit der Entwicklung von Programmiersprachen beschäftigt, so z.B. Algol-68, Modula, Modula-2 und vor allem Pascal.

In seiner gesamten Tätigkeit hat Professor Wirth immer der Idee der "strukturierten Programmierung" sehr große Bedeutung beigemessen. Ursprünglich war Pascal gar nicht als eine Programmiersprache gedacht, vielmehr wollte Professor Wirth ein Lehrmittel schaffen, mit dem er seinen Studenten diese Art des Programmierens beibringen konnte.

Strukturierte Programmierung bedeutet im wesentlichen die Aufteilung eines Programmierproblems in kleine Stücke, wobei jedes Teilstück im Idealfall unabhängig von den anderen geschrieben und verändert werden kann. Diesen Vorgang nennt man auch "modulare Programmierung". Die daraus resultierenden Vorteile sind neben einer übersichtlichen Programmstruktur eine vereinfachte Programmierung, leichte Beseitigung von Fehlern sowie die Möglichkeit, einzelne Programmteile - sogenannte Module - in andere Programme zu übernehmen. Dabei muß der Programmierer nur wissen, wie man in das Modul hineinkommt, welche Wirkung es hat und wie es wieder verlassen wird. Wie die Wirkung erreicht wird, ist dagegen ohne Bedeutung für ihn.

So sinnvoll und selbstverständlich dies alles erscheinen mag - es gibt einige weitverbreitete Programmiersprachen (z.B. zahlreiche Dialekte von Basic und Fortran), die eine Aufteilung in Module gar nicht zulassen. Für die strukturierte Programmierung geeignet sind dagegen u.a. Ada, C, Modula-2 und Pascal.

Professor Wirth hat die erste Version von Pascal 1970 veröffentlicht. Inzwischen hat die Sprache eine sehr große Verbreitung erreicht. Es gibt Pascal-Compiler für die unterschiedlichsten Computer. Die große Verbreitung hat aber auch einen Nachteil. Die verschiedenen Computer haben verschiedene Möglichkeiten, und jeder Programmierer, der einen Compiler schreibt, hat seine Vorstellungen von der idealen Sprache. Deshalb sind im Laufe der Zeit verschiedene Dialekte entstanden.

Das American National Standard Institute hat einen Standard für Pascal geschaffen, man spricht auch von ANSI-Pascal, dem allerdings nur wenige Pascal-Compiler folgen. Auch Turbo Pascal weicht in einigen Punkten von der Norm ab. Die Unterschiede halten sich aber in Grenzen.

3.2 Programmstruktur

Das Programm in Kapitel 2 bestand nur aus einem einzigen Modul. Anhand dieses Beispiels können wir also sehen, wie ein Modul aufgebaut ist. Es wird mit den folgenden Schlüsselwörtern gegliedert: *Program, Uses, Type, Var, Const, Begin* und *End.*

In der ersten Zeile steht das Wort Program, gefolgt von einem Namen. Daraus kann man erkennen, daß dies das Hauptmodul ist. In ANSI-Pascal kann diese Zeile eventuell weggelassen werden, in Turbo Pascal ab Version 4.0 muß sie aber vorhanden sein.

Ein Name ist in Pascal eine beliebig lange Folge von Buchstaben und Zahlen. In Turbo Pascal wird allerdings "nur" zwischen den ersten 63 Zeichen unterschieden. Erlaubte Zeichen sind die Buchstaben a bis *z* (also nicht ä, ö, ü und ß), Zahlen und der Unterstrich (_), wobei das erste Zeichen immer ein Buchstabe sein muß. Merken Sie sich außerdem, daß in Pascal generell nicht zwischen Groß- und Kleinschreibung unterschieden wird. Name ist also dasselbe wie NAME, name oder nAmE. In den Beispielprogrammen in diesem Buch werden alle von Pascal reservierten Wörter groß- und alle Namen kleingeschrieben.

Die zweite Zeile des Beispielprogramms beginnt mit dem Wort Uses. Dies ist eine Besonderheit von Turbo Pascal. Hier werden die Namen der Units angegeben, von denen das Programm Teile benutzt. Ein Unit ist eine Sammlung von Prozeduren und Funktionen, mit anderen Worten: ein fertiges Modul, das in Programme eingesetzt werden kann.. Das Unit-System wird von jedem Programm automatisch geladen. Wollen Sie etwas aus einem anderen Unit benutzen, muß der Name dieses Units hier angegeben werden.

Die Uses-Deklaration darf nur am Anfang eines Programms oder eines selbstgeschriebenen Units stehen.

Im Beispielprogramm folgen jetzt zwei Blöcke, die mit den Schlüsselwörtern *Const* und *Var* gekennzeichnet sind. An dieser Stelle könnte aber auch ein Block mit der Bezeichnung Type stehen.
Nach dem Wort Type folgen Definitionen von Variablentypen, nach *Const* und *Var* Deklarationen von Konstanten bzw. Variablen. In Standard-Pascal darf ein

Modul nur einen Block von jedem Typ enthalten. Turbo Pascal erlaubt dagegen mehrere Blöcke von jedem Typ und sie dürfen beliebig gemischt werden.
Zum Schluß kommt dann das eigentliche Programm, das von den Wörtern Begin und End umschlossen ist. Die Programmausführung beginnt also mit der ersten Zeile nach dem Wort Begin im Hauptmodul.

3.3 Variablen

Bei vielen Berechnung muß das Ergebnis in einer Variablen abgelegt werden. Die Variablen spielen also in jedem Programm eine ganz zentrale Rolle. Bevor eine Variable benutzt werden kann, muß sie deklariert werden. Als Beispiel dient die vierte Zeile in unserem kleinen Programm aus dem letzten Kapitel:

```
Var antwort : Char;
```

Damit wird eine Variable mit dem Namen "antwort" deklariert. Die Variable bekommt den Typ *Char.* Das bedeutet, daß für diese Variable ein Byte im Speicher reserviert und daß das Byte als Textzeichen gelesen werden soll.

Durch die Deklaration bekommt die Variable allerdings noch keinen Wert zugewiesen. Welchen Wert sie hat, hängt allein davon ab, welche Bytes zufällig an dieser Speicherstelle stehen.

Bei der Deklaration werden also der Name und der Typ einer Variablen festgelegt. Der Typ bestimmt dabei den computerinternen Speicherbedarf zur Verwaltung der Variablen.

In Turbo Pascal existieren eine Reihe verschiedener Typen: *String, Char, Integer* usw. Sie lassen sich in numerische, String-, zusammengesetzte und selbstdefinierte Typen klassifizieren.

Vielleicht sieht es so aus, als sei die Deklaration von Variablen etwas sehr Umständliches, besonders weil es andere Programmiersprachen gibt, wo sie gar nicht notwendig ist (z.B. Basic und Logo).

In diesen Sprachen muß aber auch Speicherplatz reserviert werden. Diese Aufgabe wird nur von dem Interpreter erledigt. In Basic wird z.B. nur zwischen Zahlen und Strings (Zeichenketten) unterschieden, wobei normalerweise 8 bzw. 256 Byte reserviert werden. Wenn man aber in diesen Variablen nur ein Datum und den dazugehörigen Wochentag speichern will, würden l bzw. 10 Bytes vollkommen ausreichen. Durch die richtige Deklaration kann also sehr viel Speicherplatz gespart werden.

Damit folgt auch gleich der zweite Vorteil. Bei jeder Berechnung müssen die reservierten Bytes gelesen werden. Wenn Sie so wenige Bytes wie möglich reservieren, wird das Programm also auch schneller.

Schließlich wird auch die Fehlersuche vereinfacht. Wenn Sie bei einem Variablennamen einen Tippfehler machen, wird der Compiler diesen Namen zuerst als den Namen einer ganz neuen Variablen lesen. Da diese neue Variable aber nicht deklariert ist, wird der Compiler eine Fehlermeldung ausgeben.

Numerische Typen

Bei Zahlen wird zwischen Ganzzahlen und Fließpunktzahlen differenziert. Der Unterschied besteht vor allem darin, wie die Zahlen im Speicher abgelegt werden. Bei den Ganzzahlen wird weiter zwischen den sogenannten *Integer*- und *Cardinal*-zahlen unterschieden. Integers sind alle ganzen Zahlen, positive wie negative und Null. Zu den Cardinalzahlen gehören dagegen nur die positiven ganzen Zahlen und Null.

Für Integers gibt es in Turbo Pascal Typen mit l, 2 und 4 Bytes Speicherbedarf. Daraus kann man gleich die möglichen Mindest- und Höchstwerte dieser Typen ableiten:

Typ:	*Speicherbedarf:*	*Bereich:*
Shortint	l Byte	-128 bis+127
Integer	2 Bytes	-32768 bis +32767
Longint	4 Bytes	-2147483648 bis+2147483647

Für Cardinalzahlen gibt es nur zwei Möglichkeiten:

Typ:	*Speicherbedarf:*	*Bereich:*
Byte	l Byte	0 bis 255
Word	2 Bytes	0 bis 65535

Sollen auch Dezimalzahlen in einer Variablen abgelegt werden, müssen Sie einen Fließpunkttyp benutzen. Davon gibt es vier Typen für Dezimalzahlen und einen für Ganzzahlen:

Typ:	*Speicherbedarf:*	*Bereich:*	*Genauigkeit:*
Real	6 Bytes	2.9*10E-39 bis 1.7*10E38	11-12 Stellen
Single	4 Bytes	1.5*10E-45 bis 3.4*10E38	7-8 Stellen
Double	8 Bytes	5.0*10E-324 bis 1.7*10E308	15-16 Stellen
Extended	10 Bytes	1.9*10E-4951 bis 1.1*10E4932	19-20 Stellen
Comp	8 Bytes	-2E+63+1 bis 2E+63-1	19-20 Stellen

Wie Sie sehen, ist der Wertebereich der Fließpunktzahlen erheblich größer als bei den Ganzzahlen. Der Nachteil liegt in der Genauigkeit. Mit einer Variablen vom Typ Extended können zwar Zahlen mit fast 5000 Stellen gespeichert werden, aber nur die ersten 19 oder 20 sind genau, der Rest ist dem Zufall überlassen.

Bei Turbo Pascal Version 4.0 gibt es eine weitere Einschränkung. Wenn Sie hier einen der Typen *Single, Double, Extended* oder *Comp* benutzen, kann das fertige

Programm nur auf einem Computer mit Arithmetik-Hilfsprozessor (80x87) laufen. Dieser Mangel ist in Version 5 ,x behoben. Hier wird das fertige Programm erst untersuchen, ob ein Hilfsprozessor vorhanden ist. Ist dies der Fall, wird er benutzt, andernfalls werden die Berechnungen mit (langsameren) Software-Routinen durchgeführt.

Zu den numerischen Typen gehört auch der Typ Boolean:

Typ:	*Speicherbedarf:*	*Bereich:*
Boolean	l Byte	True - False

Eine Variable vom Typ *Boolean* kann nur die beiden Werte *True* und *False* aufnehmen. Daß es trotzdem ein numerischer Typ ist, liegt daran, daß *True* und *False* vordefinierte Konstanten mit den Werten l bzw. 0 sind.

String-Typen

In unserem Beispielprogramm hatten wir schon den Typ *Char.* Dieser Typ belegt ein Byte im Speicher und kann ein ASCII-Zeichen aufnehmen. In Anhang D gibt es eine Tabelle mit allen ASCII-Zeichen.

Für längere Texte gibt es den Typ *String.* Ein String kann aus einem Text mit bis zu 255 ASCII-Zeichen bestehen. Um Speicherplatz zu sparen, können Sie bei der Deklaration eine maximale Länge des Strings angeben. Diese muß dann hinter dem Wort String in eckigen Klammern stehen, Beispiel:

```
var  zeile : String[80];
```

Ein String hat immer eine maximale und eine aktuelle Länge. Ist die maximale Länge nicht in der Deklaration angegeben, wird sie automatisch auf 255 angesetzt. Die aktuelle Länge ist dagegen die Länge des Strings, der in die Variable abgelegt ist. Diese Länge ist immer kürzer oder gleich der maximalen Länge.

Zu den ASCII-Zeichen gehören auch die Ziffern 0 bis 9. Man darf aber niemals die ASCII-Zeichen 0 bis 9 mit den Zahlen 0 bis 9 verwechseln. Mit Variablen vom Typ *String* oder *Char* ist es nicht möglich, irgendwelche Berechnungen durchzuführen. Auch nicht, wenn in dieser Variablen Ziffern abgelegt sind.

Zusammengesetzte Typen

Manchmal braucht man eine Anzahl von Variablen desselben Typs. Dann kann man einen Array deklarieren. Wollen Sie z.B. einen Array mit 50 Variablen vom Typ *Integer* deklarieren, schreiben Sie folgendes:

```
Var  felt : Array[l..50]Of Integer;
```

Nach dem Wort Array folgt der Index in eckigen Klammern, dann das Wort of und der Typ der Elemente. Der Index besteht aus zwei Zahlen desselben Typs, durch zwei Punkte getrennt. Die Zahlen in dem Index dürfen von jedem Integer-, Cardinal- oder Booleantyp außer *Longint* sein. Daraus ergibt sich, daß ein Array höchstens 65535 Elemente haben kann.

Die einzelnen Elemente eines Arrays werden mit dem Namen des Arrays und einem Index in eckigen Klammern angesprochen und können dann genau wie jede andere Variable dieses Typs behandelt werden:

felt [10] :=50
felt [18] :45

Die Elemente können von jedem beliebigen Typ sein, es ist sogar erlaubt, einen *Array of Array* zu deklarieren, das könnte dann so aussehen:

```
Var felt : Array[l..10]Of Array[l..50]Of Char;
feit[3][45]:='A';
```

Man spricht dann von einem *zwei-dimensionalen* Array. Sie können aber auch *mehrdimensionale Arrays* deklarieren. Theoretisch ist die Anzahl der Dimensionen unbegrenzt, allerdings darf kein Array eine Gesamtgröße von mehr als 65535 Bytes haben.

Die Elemente eines Arrays sind immer von demselben Typ. Manchmal braucht man aber eine Variable, die aus verschiedenen Typen zusammengesetzt ist. Dieser Typ heißt Record. Wollen Sie z.B. ein Programm zum Verwalten einer Büchersammlung schreiben, könnten Sie die folgende Variable deklarieren:

```
Var buch : Record
Verfasser : String[40];
titel : String[30];
Seitenzahl : Word;
schlagwort : Array[l..5]Of
String[20];
End;
```

Die einzelnen Elemente eines Records werden mit dem Namen des Records und dem Namen des Elements von einem Punkt getrennt angesprochen. Im obigen Beispiel könnten Sie z.B. den Titel mit Buch.Titel ansprechen.

Sowohl Arrays als auch Records haben eine feste Größe und eine feste Anzahl von Elementen. In Turbo Pascal gibt es aber auch den Typ *File,* dessen Größe nicht begrenzt ist. Die Daten eines Arrays oder Records werden im Hauptspeicher abgelegt. Die Daten eines Files werden dagegen in einen externen Datei abgelegt, deren Größe nur durch die Größe der Diskette bzw. der Festplatte begrenzt ist.

Die Deklaration eines Files ähnelt der eines Arrays. Allerdings fehlt der Index. Beispiel:

```
Var datei : File Of Integer;
```

Ein besonderer Filetyp ist der Typ *Text.* Diesen Typ kann man als *File of Zeilen* verstehen. Die Elemente sind Strings unterschiedlicher Größe, von Carriage-Return-Zeichen getrennt.

Selbstdefinierte Typen

Wenn alle diese Typen nicht ausreichen, können Sie auch noch Ihre eigenen definieren. Damit können Sie die Lesbarkeit eines Programms erhöhen. Außerdem kann es dadurch manchmal vereinfacht werden. Das gilt z.B., wenn Sie mehrere Variablen von demselben zusammengesetzten Typ deklarieren wollen.

Beispiel:

Erst wird der Typ definiert. Das geschieht nach dem Schlüsselwort Type. Danach können die Variablen ganz normal deklariert werden.
Wir hatten vorher das Beispiel eines Bücherverwaltungsprogramms. In einem solchen Programm würden wir mehrere Variablen vom gezeigten Recordtyp benötigen. Um den ganzen Record nicht jedesmal angeben zu müssen, können wir den Record als neuen Typ definieren. Das würde dann so aussehen:

```
Type buch : Record
Verfasser : String[40];
titel : String[30]';
Seitenzahl : Word;
schlagwort : Array[l..5]Of String[20];
End;
Var buecher : File Of buch;
buch l : buch;
```

3.4 Konstanten

In Turbo Pascal können Konstanten mit oder ohne Typangabe deklariert werden. Im Beispielprogramm im letzten Kapitel gab es auch eine Konstante:

```
Const positiv : Char = 'J';
```

Eine Konstante ist eigentlich eine Variable mit festem Wert. Konstanten dienen vor allem dazu, die Lesbarkeit eines Programms zu verbessern. Die Zahl 12 könnte ja alles Mögliche bedeuten. Die Konstante *Monate_im Jahr* ist dagegen unmittelbar verständlich.

3.5 Prozeduren und Funktionen

Das Beispielprogramm in Kapitel 3 bestand nur aus einem Modul. Die meisten Programme bestehen aber aus einem Hauptmodul und einigen Untermodulen. Die Untermodule können direkt im Hauptmodul vor das Wort *Begin* eingefügt werden. Sie können aber auch in einem selbständigen Unit untergebracht werden, das dann in der Uses-Deklaration aufgenommen wird. Die einzelnen Untermodule sind aber in beiden Fällen gleich.

Es gibt zwei Typen von Untermodulen: *Prozeduren* und *Funktionen.* Der Unterschied besteht darin, daß eine Funktion einen Wert an das aufrufende Modul zurückliefert.

Die Untermodule haben weitgehend dieselbe Form wie das Hauptmodul. Es gibt nur drei Unterschiede: In der ersten Zeile steht nicht "program", sondern entweder "Procedure" oder "Function"; ein Untermodul kann keine eigene Uses-Deklaration haben; nach dem Wort "End" steht kein Punkt, sondern ein Semikolon.

Die Programmausführung beginnt immer nach dem Wort *Begin* im Hauptmodul. Soll ein Untermodul ausgeführt werden, schreiben Sie einfach den Namen dieses Untermoduls im Hauptmodul zwischen *Begin* und *End.* Das Programm springt dann zu dem Wort *Begin* im Untermodul. Wenn das Untermodul ausgeführt ist, springt das Programm zum Hauptmodul zurück und setzt die Programmausführung unmittelbar nach dem Namen des Untermoduls fort.

Genauso können Sie von einem Untermodul aus ein zweites Untermodul rufen und ausführen lassen. Es ist sogar erlaubt, daß ein Untermodul sich selbst ruft, man spricht dann von rekursiver Programmierung.

Globale und lokale Variablen

Jedes Untermodul kann genau wie das Hauptmodul ein oder mehrere Blöcke mit Typendefinitionen und/oder Variablen- oder Konstantendeklarationen enthalten. Eine grundlegende Forderung der strukturierten Programmierung war, daß jedes Modul eine selbständige abgeschlossene Einheit bilden muß.

Das gilt auch für Variablen und Konstanten. Eine Variable, die innerhalb eines Moduls deklariert ist, ist deshalb nur in diesem Modul gültig. Außerhalb des Moduls gilt sie als unbekannt. Zwei Variablen, die in verschiedenen Modulen deklariert sind, dürfen sogar denselben Namen haben; der Compiler ist trotzdem in der Lage, zwischen ihnen zu unterscheiden. So muß es aber auch sein, sonst könnte man nicht ein Modul aus einem Programm herausnehmen und in ein anderes einsetzen, ohne Namenskonflikte zu riskieren.

Um diese Trennung zu demonstrieren, ist hier ein zweites Beispielprogramm:

```
Program zweites_beispiel;
Var zahl : Word;
Procedure probe;
Var zahl : Word;
Begin
Writeln(zahl);
zahl:=100;
Writeln (zahl);
End;
Begin
zahl:=25;
Writeln(zahl);
probe;
Writeln (zahl);
probe;
End.
```

Die Programmausführung beginnt unmittelbar nach dem Wort *Begin* im Hauptprogramm. Die erste Zeile die ausgeführt wird, ist also "zahl:=25;". Dabei bekommt die Variable *zahl* den Wert 25. *Writeln* kennen Sie schon von dem ersten Beispielprogramm; damit wird der Inhalt der Variablen zahl auf den Bildschirm geschrieben.

Dann wird die Prozedur *probe* gerufen. Das Programm springt zur ersten Zeile nach dem Wort *Begin* in der Prozedur. Hier wird wieder die Variable *zahl* auf den Bildschirm geschrieben. Diesmal ist es aber die Variable aus der Prozedur. Sie ist zwar deklariert, hat aber noch keinen Wert. Trotzdem wird irgendetwas geschrieben. Was es ist, hängt davon ab, was zufällig in den reservierten Bytes stand. Die Variable *zahl* bekommt in der nächsten Zeile den Wert 100 und wird anschließend auf dem Bildschirm ausgegeben.

Die Programmausführung ist jetzt bei dem Wort *End* in der Prozedur angekommen. Deshalb springt das Programm zum Hauptmodul zurück. Hier wird die Variable *zahl* wieder geschrieben. Wie Sie sehen, hat sie immer noch den Wert 25. Die Prozedur wird jetzt erneut gerufen. Dabei sehen Sie, daß das Programm inzwischen vergessen hat, daß die Variable schon einen Wert bekommen hatte. Die Variable ist wieder Undefiniert.

Wenn ein Untermodul gerufen wird, springt das Programm zu dem Wort *Begin* im Untermodul. Bei diesem Wort wird die Deklaration durchgeführt, es wird also Speicherplatz für die Variablen des Moduls reserviert. Dann wird das Modul ausgeführt bis zum Wort *End.* Hier wird der reservierte Speicherplatz wieder freigegeben und alle Werte, die in den Variablen gespeichert sind, gehen verloren.

Die Variablen existieren also nur, während das Modul ausgeführt wird. Aber was heißt das nun? Das Hauptmodul wird ja nicht abgeschlossen, wenn die Prozedur gerufen wird, sondern erst bei dem letzten *End.* Während die Prozedur ausgeführt wird, existieren also sowohl die Variablen aus dem Hauptprogramm als auch die aus der Prozedur.

Das können wir nachprüfen. Entfernen Sie im Beispielprogramm die Variablen-Deklaration aus der Prozedur. Wenn das Programm jetzt wieder gestartet wird, gibt es in der Prozedur keine Variable mit dem Namen *zahl.* In dem Hauptprogramm gibt es aber eine. Deshalb greift die Prozedur auf diese zurück. Umgekehrt wäre es allerdings nicht möglich. Eine Variable in der Prozedur existiert ja nur, während die Prozedur ausgeführt wird. Das Hauptprogramm hat auf diese Variable keinen Zugriff.

Ein Untermodul kann auf die Variablen des Hauptprogramms zugreifen, kann sie lesen und sogar ändern. Es tut es aber nur, wenn keine lokale Variable desselben Namens existiert.

Am Anfang des Kapitels wurde erwähnt, daß ein Untermodul weitgehend so aussieht wie das Hauptmodul. Das bedeutet, daß ein Untermodul auch Untermodule enthalten kann. Das Programm bekommt eine Baumstruktur.

Die Begriffe globale und lokale Variable sind deshalb relativ. Eine Variable ist im eigenen Modul lokal, in den unterstehenden Modulen global und in den höherstehenden Modulen unbekannt.

Dabei kann es natürlich passieren, daß zwei höherstehende Module globale Variablen mit demselben Namen enthalten. Wenn das so ist, wird ein Untermodul immer auf die Variable im nächsthöheren Modul zugreifen.

Parameter

Sie können Werte von einem Modul in ein anderes übertragen, indem Sie sie in Variablen ablegen. Sie müssen nur dafür sorgen, daß die Variablen in dem höherstehenden Modul deklariert sind.

Das wäre aber nicht ganz in Übereinstimmung mit den Prinzipien des strukturierten Programmierens. Wenn eine Prozedur bestimmte globale Variablen erwartet und benutzt, kann sie ja nicht beliebig in andere Programme eingefügt werden. Deshalb sollten Sie lieber die Übergabe von Werten mit Hilfe von Parametern erledigen. Das haben Sie schon gesehen. Die Prozedur *Writeln,* die als Standardprozedur mit dem Compiler geliefert wird, wird im Regelfall mit einem Parameter aufgerufen. Der Parameter ist das, was auf dem Bildschirm geschrieben werden soll.

Parameter können Sie auch in selbstgeschriebenen Prozeduren benutzen. Nach dem Wort "Procedure" und dem Namen der Prozedur müssen Sie einfach den Namen und Typ des Parameters angeben, beide in Klammern und durch einen Doppelpunkt getrennt. Das könnte dann so aussehen:

```
Program drittes_beispiel;
Var zahl : Word;
Procedure probe(zahl_a : Word);
Var zahl : Word;
Begin
Writeln(zahl_a);
zahl:=100;
Writeln(zahl);
End;
Begin
zahl:=25;
probe(zahl);
End.
```

Die Prozedur *probe* erwartet jetzt einen Parameter vom Typ *Word.* Innerhalb der Prozedur ist der Parameter eine lokale Variable. Allerdings bekommt sie ihren Anfangswert beim Aufruf der Prozedur, also schon im Hauptprogramm.
Der Vorteil ist, daß die Prozedur jetzt ohne Änderungen in jedes andere Programm eingefügt werden kann.

Am Anfang des Abschnittes wurde erwähnt, daß der Unterschied zwischen Prozeduren und Funktionen darin besteht, daß eine Funktion einen Rückgabewert liefert. Das hatten wir auch schon. In dem ersten Beispielprogramm gab es die Zeile:

```
antwort:=UpCase(ReadKey);
```

Sie enthält gleich zwei Standardfunktionen: *ReadKey* und *UpCase.* Die erste hält die Programmausführung an, bis der Benutzer eine Taste betätigt, und liefert als Rückgabewert den ASCII-Wert dieser Taste. Die zweite erwartet einen Parameter vom Typ *Char* und liefert als Rückgabewert den entsprechenden Großbuchstaben.

In dem Beispiel wurde die Funktion *ReadKey* als Parameter an die Funktion *UpCase* übergeben. Dadurch wird das Programm etwas kürzer, und man spart eine Variable. Sie können aber auch so schreiben:

```
buchstabe:=ReadKey;
antwort:=UpCase(buchstabe);
```

wo sowohl *buchstabe* als auch *antwort* Variablen vom Typ *Char* sind.

Eine Funktion kann genau wie eine Prozedur mit einem Parameter gerufen werden. Auch bei den selbstgeschriebenen Funktionen gibt es in diesem Punkt keinen Unterschied.

Der Rückgabewert muß bei den selbstgeschriebenen Funktionen an zwei Stellen berücksichtigt werden. Erstens muß hinter dem Funktionsnamen und eventuellen Parametern der Typ des Rückgabewertes angegeben werden. Zweitens muß unmittelbar vor dem Wort End dem Rückgabewert einen Wert zugewiesen werden. Hier ein Beispiel:

```
Program viertes_beispiel;
Var ergebnis : Word;
Function quadrat(seite : Word):Word;
Var zahl : Word;
Begin
zahl:=seite*seite;
quadrat:=zahl ;
End;
Begin
ergebnis:=quadrat(25);
Writeln(ergebnis)
End.
```

3.6 Ausdrücke und Befehle

Jetzt fehlt nur noch das, was zwischen *Begin* und *End* stehen soll. Man könnte sagen, dies sei das eigentliche Programm, also die Berechnungen. Auch hier können wir auf die schon gezeigten Beispiele zurückgreifen. Im vierten Beispiel gab es die folgende Zeile:

zahl:=seite*seite;

Hier wird der Inhalt der Variablen *seite* mit sich selbst multipliziert und das Ergebnis in der Variablen *zahl* abgelegt.

Die Zeile ist ein Befehl. Es wird befohlen, daß das Quadrat von *seite* in *zahl* abgelegt werden soll. Vielleicht haben Sie es schon bemerkt: Jeder Befehl wird mit einem Semikolon abgeschlossen. Das hat einen entscheidenden Vorteil. Dadurch ist es nämlich möglich, einen Befehl über mehrere Zeilen gehen zu lassen oder mehrere Befehle in einer Zeile zu schreiben. Der Compiler wird alles, was zwischen zwei Semikolons steht, als einen Befehl lesen.

Operatoren

Der rechte Teil der Zeile "seite * seite" ist ein Ausdruck. Ein Ausdruck besteht aus *Operanden* und *Operatoren. seite* ist der Operand und der Stern ist der Operator. Das was passiert, also daß *seite* mit sich selbst multipliziert wird, ist eine Operation. Das Quadrat von *seite* ist das Ergebnis der Operation.

Operanden können Variablen und Konstanten sein. Dabei gilt, daß die Typen der Operanden in einem Ausdruck *kompatibel* sein müssen. Das bedeutet, daß die Operanden zueinander passen müssen. Es ist z.B. einleuchtend, daß man nicht einen Buchstaben zu einer Zahl addieren kann.

Die Regeln für die Kompatibilität sind leider etwas kompliziert. Der Compiler wird aber immer auf einen Regelverstoß aufmerksam machen. Deshalb reicht es, die folgenden, etwas vereinfachten Regeln zu kennen:

1. Zwei identische Typen sind immer kompatibel.
2. Alle Fließpunkttypen sind kompatibel.
3. Die Typen String und Char sind kompatibel.
4. Integer- und Cardinaltypen sind kompatibel.

Ein Ausdruck kann aus vielen Operanden und Operatoren bestehen. Dabei geschieht die Auswertung des Ausdruckes in einer bestimmten Reihenfolge. Jeder Operator hat eine Priorität. Operatoren mit höherer Priorität werden zuerst ausgewertet. Sonst geschieht die Auswertung von links nach rechts. Unabhängig davon werden Teilausdrücke in runden Klammern allerdings zuerst bearbeitet. Damit kann man die Reihenfolge der Auswertung beeinflussen.

Höchste Priorität hat der Operator *Not.* Dieser Operator braucht als einziger nur einen Operanden. Ist der Operand ein Ganzzahlentyp, wird dieser bitweise negiert; ist der Operand dagegen vom Typ *Boolean,* wird er einfach negiert. Andere Operanden sind nicht erlaubt.

Zweite Priorität haben Operatoren für Multiplikation und Division. Sie haben immer zwei Operanden von kompatiblen Typen. Bei dem Operator "/" ist das Ergebnis immer ein Fließpunkttyp, sonst muß das Ergebnis mit den Operanden kompatibel sein.

Operator	*Operanden*	*Operation*
*	Ganzzahl,Fließpunkt	Multiplikation
/	Ganzzahl,Fließpunkt	Division
Div	Ganzzahl	Integerdivision
Mod	Ganzzahl	Moduladivision
And	Ganzzahl,Boolean	Verknüpfung und
Shl	Ganzzahl	shift links
Shr	Ganzzahl	shift rechts

Dritte Priorität haben Operatoren für Addition und Subtraktion. Auch sie brauchen zwei Operanden, und hier müssen beide Operanden und das Ergebnis immer kompatibel sein.

Operator	*Operanden*	*Operation*
+	Ganzzahl,Fließpunkt	Addition
	String	Verkettung
-	Ganzzahl,Fließpunkt	Subtraktion
Or	Ganzzahl,Boolean	Verknüpfung oder
Xor	Ganzzahl,Boolean	Verknüpfung entweder oder

Vierte Priorität haben die Vergleichsoperatoren. Hier werden zwei kompatible Operanden verglichen. Das Ergebnis ist immer vom Typ Boolean, also entweder True oder False (wahr oder falsch):

Operator	*Operation*
=	gleich
<>	ungleich
<	kleiner als
>	größer als
<=	kleiner oder gleich
>=	größer oder gleich

Zuordnung

Gehen wir zurück zu der Zeile "zahl:=seite*seite;". Der rechte Teil der Zeile war ein Ausdruck, dessen Ergebnis in der Variablen *zahl* abgelegt werden soll. Das wird mit dem Befehlswort ":=" (Doppelpunkt Gleichheitszeichen) erledigt. Dieses Zeichen bedeutet also gleich.

Das alltägliche Wort gleich hat aber zwei ganz verschiedene Bedeutungen. Es kann entweder eine Zuordnung oder eine Aussage sein.
Bei der *Zuordnung* "A gleich B" bekommt A den Wert von B. A ändert seinen Wert, und nachher sind beide gleich.

Bei der *Aussage* "A gleich B" ändern weder A noch B ihren Wert. Es handelt sich lediglich um eine Behauptung, die entweder wahr oder falsch sein kann.

Um diese beiden Fälle zu unterscheiden, benutzt man in Pascal zwei verschiedene Zeichen. Bei der Zuordnung schreibt man ":=" , bei der Aussage nur das Gleichheitszeichen. Das Zeichen ":=" ist also ein Befehlswort, das nichts mit dem Operator "=" zu tun hat.

Der Zuordnungsbefehl legt einen Wert in einer Variablen ab. Der Wert kann dabei eine Konstante, der Inhalt einer Variablen oder das Ergebnis eines Aus-

druckes sein. Sie müssen aber immer darauf achten, daß die möglichen Werte innerhalb des Wertebereichs der Variablen fallen. Sie dürfen z.B. nicht einen negativen Wert einer Variablen des Typs Word zuordnen. Das Ergebnis wäre ein Bereichsfehler.
Wie das Programm auf einen Bereichsfehler reagiert, hängt von der Einstellung im Menü "Options/Compiler/Range checking" ab. Steht hier *Off,* wird das Programm mit einen falschen Wert weiterarbeiten. Ändert man aber die Einstellung zu *On,* wird das Programm mit einer Fehlermeldung abgebrochen.

Aufruf von Prozeduren und Funktionen

Es gibt auch andere Befehle, die schon in den Beispielprogrammen vorgekommen sind. Der Aufruf einer Prozedur oder Funktion ist nämlich auch ein Befehl. Solche Befehle bestehen immer aus dem Namen der Prozedur oder Funktion, die gerufen werden soll, gefolgt von eventuellen Parametern in runden Klammern.

Wie schon erwähnt, liefert eine Funktion immer einen Rückgabewert. Das muß schon beim Aufruf berücksichtigt werden. Der Rückgabewert kann entweder in einer Variablen abgelegt, als Operand in einem Ausdruck benutzt oder als Parameter für eine Prozedur oder eine zweite Funktion verwendet werden. Als Beispiel soll nochmals auf das erste Beispielprogramm verwiesen werden.

Programmschleifen

Oft sollen in einem Programm einige Befehle mehrmals wiederholt werden. Dazu benutzt man eine Schleife. In Turbo Pascal gibt es drei Typen von Schleifen; die erste ist die For-To-Do-Schleife:

```
For variable:=ausdruck1 To ausdruck2 Do befahl;
```

Beim ersten Schleifentyp wird die Variable zuerst einem Wert zugeordnet, und zwar dem Ergebnis von *ausdruck1.* Dann wird untersucht, ob die Variable größer als das Ergebnis von *ausdruck2* ist. Wenn dies nicht der Fall ist, wird der Befehl ausgeführt. Danach wird die Variable um eins vergrößert, der neue Wert wird wieder mit *ausdruck2* verglichen und so weiter. Beispiel:

```
For i:=l To 10 Do Writeln('=Probe');
```

damit wird das Wort Probe 10 mal auf dem Bildschirm geschrieben.
Es wäre auch erlaubt, statt *To* das Wort *Downto* zu schreiben. Dann wird die Variable in jedem Durchlauf um eins verringert und nicht vergrößert.

Sie können in jedem Durchlauf nur einen Befehl ausführen lassen. Das ist aber kein Problem; wenn mehrere Befehle zwischen den Wörtern *Begin* und *End* stehen, wird der Compiler sie als einen Befehl verstehen. Beispiel:

```
For i:=l To 10 Do
Begin
Writeln ( 'erster Befehl');
Writeln (' zweiter Befehl');
End;
```

Manchmal ist es nicht möglich zu entscheiden, wie oft ein Befehl ausgeführt werden soll. Vielmehr soll der Befehl so lange ausgeführt werden, wie ein bestimmter Zustand besteht oder bis ein bestimmter Zustand eintritt. Das können Sie mit den folgenden beiden Schelfen erreichen:

```
While ausdruck Do befehl;
Repeat befehl Until ausdruck;
```

Der Ausdruck muß in beiden Fällen ein Ergebnis vom Typ *Boolean* liefern, es muß also ein Vergleich zweier Operanden sein. In der While-Do-Schleife wird der Befehl so lange wiederholt, wie *ausdruck* als Ergebnis den Wert *True* liefert. In der *Repeat-Until-Schleife* wird der Befehl dagegen wiederholt, bis *ausdruck* den Wert *True* liefert.

In vielen Fällen können Sie dasselbe Ergebnis mit mehreren verschiedenen Schleifen erreichen. Die zwei folgenden Schleifen haben z.B. genau dieselbe Wirkung wie obenstehende For-To-Do-Schleife:

```
While i<11 Do
Begin
Writeln ( 'erster Befehl');
Writeln (' zweiter Befehl');
End;
Repeat
Writeln ( 'erster Befehl');
Writeln (' zweiter Befehl'11);
Until  i=11;
```

Wie Sie sehen, muß man innerhalb der Schleife einen der Operanden in dem Ausdruck ändern, sonst würde die Schleife endlos laufen. Allerdings wird diese Änderung oft ein Nebenergebnis der anderen Befehle sein.

Bei der *While-Do-Schleife* wird der Ausdruck am Anfang der Schleife überprüft. Wenn der Ausdruck schon von Anfang an den Wert *False* liefert, werden die Befehle innerhalb der Schleife überhaupt nicht ausgeführt.

Bei der *Repeat-Until-Schleife* gelangen die Befehle dagegen immer mindestens einmal zur Ausführung.

Wenn Sie das Aussehen des Ausdruckes festlegen, müssen Sie sorgfältig überlegen, welche Werte dieser Ausdruck unter verschiedenen Umständen annehmen könnte. In der folgenden Schleife wird die Variable i z.B. niemals den Wert 11 annehmen. Der Ausdruck i=ll kann also niemals das Ergebnis *True* liefern. Damit ist die Schleife endlos oder genauer gesagt, die Schleife wird so oft wiederholt, bis ein Bereichsfehler auftritt.

```
i:=0;
Repeat
Writeln(i);
i:=i+2;
Until   i=ll;
```

Sprünge

Wer schon einmal in Basic programmiert hat, kennt mit Sicherheit den Befehl *Goto.* Einen solchen Befehl gibt es auch in Pascal. Mit *Goto* wird zu einer anderen Stelle im Programm gesprungen, von wo aus die Programmausführung fortgesetzt wird.

In Pascal darf das Ziel des Sprunges allerdings nur innerhalb desselben Moduls liegen. Alles andere wäre ja auch ein Verstoß gegen die Regeln der strukturierten Programmierung, die besagen, daß ein Modul nur einen Eingang und einen Ausgang haben darf.

Ziel eines Sprunges ist ein Label. Ein Label ist entweder eine ganze Zahl im Bereich 0 bis 9999 oder ein Name, von einem Doppelpunkt gefolgt. Wie andere Elemente müssen auch Labels deklariert werden. Das erfolgt in einem eigenen Block hinter dem Schlüsselwort *Label.* Hier ein Beispiel:

```
Program beispiel_fuenf;
Label sprungziel;
Begin
Goto sprungziel;
Writeln ( ^Diese Zeile wird nicht ausgeführt');
sprungziel:
End.
```

Sie sollten aber mit der Verwendung von *Goto* sehr vorsichtig sein. Ein Modul kann hiermit sehr unübersichtlich werden. Außerdem gibt es immer elegantere Lösungen. In diesem Buch wird der Befehl auf jeden Fall nicht wieder vorkommen.

Der Befehl *Goto* ist ein unbedingter Sprungbefehl. Viel öfter braucht man aber einen bedingten Sprung, d.h. er wird nur unter bestimmten Umständen ausgeführt. Hierzu gibt es den Befehl *If-Then-Else:*

```
If ausdruck Then befehl1 [Eise befeh!2];
```

Wie bei den Schleifen muß *ausdruck* ein Ergebnis vom Typ *Boolean* liefern. Liefert *ausdruck* den Wert *True,* wird *befeh1l* ausgeführt, sonst wird *befehl2* ausgeführt. *Eise* und *befehl2* können je nach Bedarf weggelassen werden.
Wenn in einem der beiden Fälle mehrere Befehle ausgeführt werden sollen, müssen sie auch hier von den Wörtern *Begin* und *End* umschlossen sein. Schließlich sollten Sie bedenken, daß *If-Then-Else* ein Befehl ist. Das bedeutet, daß Sie vor dem Wort *Eise* kein Semikolon schreiben dürfen.
Soll zwischen mehr als zwei Möglichkeiten unterschieden werden, können Sie natürlich mehrere If-Then-Else-Befehle benutzen. Eleganter ist aber der Befehl *Case.* Die Syntax für diesen Befehl lautet:

```
Case ausdruck Of
fall_l : befehl_l;
fall 2 : befehl 2;
......
fall_n   :   befehl_n;
[Else       befehl_sonst;]
End.
```

ausdruck muß ein Ergebnis vom Typ *Integer, Shortint* oder *Byte* liefern. *fall_l* bis *fall_n* sind Konstanten, die mögliche Ergebnisse des Ausdruckes repräsentieren, und *befehl_l* bis *befehl_n* sind die Befehle, die jeweils ausgeführt werden sollen.

Sollte *ausdruck* einen Wert liefern, der nicht in der Liste aufgeführt ist, wird der Befehl nach dem Wort Eise ausgeführt. Auch hier können das Wort Eise und der letzte Befehl eventuell weggelassen werden.

In Verbindung mit den Sprungbefehlen gibt es zwei Standardprozeduren, die erwähnt werden sollen: *Exit* und *Halt.*

Die Prozedur *Exit* macht einen unbedingten Sprung zu dem Wort End im aktuellen Modul. Dadurch wird eine Prozedur oder Funktion vorzeitig beendet. Wird die Prozedur dagegen im Hauptmodul gerufen, wird das Programm unterbrochen.

Die Prozedur *Halt* bricht dagegen immer das Programm ab, egal wo die Prozedur gerufen wird. Das kann z.B. notwendig sein, wenn ein ernsthafter Fehler aufgetreten ist. Beide Prozeduren sind nur sinnvoll in Verbindung mit einem If-Then-Else-Befehl.

4 Das erste richtige Programm

Nachdem wir jetzt alle Elemente eines Pascal-Programms kennengelernt haben, können wir sofort anfangen, das erste größere Programm zu schreiben.

In diesem Kapitel soll deshalb ein etwas umfassenderes Programm zur Verwaltung von Büchern erstellt werden. Mit dem Programm soll es möglich sein, die Daten von Büchern einzugeben, zu speichern und wieder anzeigen zu lassen. Auf diese Weise können Sie ein Buch nach Titel, Verfasser oder Schlagwort suchen und Listen mit Büchern ausdrucken.

Klingt das zu schwierig? Dann sollten Sie bedenken, daß Turbo Pascal eine strukturierte Programmiersprache ist. Das Programm wird in kleine Module aufgeteilt. Die meisten Module werden weniger als 24 Programmzeilen haben; Sie können also das gesamte Modul auf dem Bildschirm sehen. Der Compiler übernimmt die Aufgabe, die Module zu einem Programm zusammenzufügen.

Es gibt dabei einige Regeln, die Sie berücksichtigen sollten, um den Überblick nicht zu verlieren. Benutzen Sie immer aussagekräftige Namen für Variablen, Prozeduren etc. Bei der Compilierung werden auch die Namen in Maschinensprache übersetzt. Deshalb können Sie ruhig die erlaubten 63 Zeichen ausnutzen, das Programm wird dadurch weder größer noch langsamer.

Wie früher erwähnt, sollten Sie auch Konstanten benutzen sowie überall Kommentare einsetzen. Kommentare müssen in geschweiften Klammern oder zwischen runden Klammern und Sternen stehen. { Dies ist ein Kommentar } und (* dies ist auch ein Kommentar *). Damit können Sie die Bedeutung und Wirkung einzelner Programmzeilen erklären und so die Lesbarkeit wesentlich erhöhen. Die Kommentare werden vom Compiler vollkommen ignoriert. Sie machen also das übersetzteProgramm weder größer noch langsamer.

4.1 Eine Maske für die Daten

In jedem Datenverwaltungsprogramm ist es sehr wichtig, ein Schema für die Daten zu haben. Wenn die Daten eingegeben oder wieder gesucht werden, müssen sie formatiert auf dem Bildschirm gezeigt werden.

Man könnte das Schema für die "Karteikarte" mit den Grafikprozeduren von Turbo Pascal zeichnen. Dann würde das Programm aber nur auf einem Computer mit Grafikkarte laufen. Es mag sein, daß die meisten Computer inzwischen eine solche Karte haben. Trotzdem sollte man immer versuchen, die kleinstmögliche Anforderungen an die Hardware zu stellen. Nur dann kann man erreichen, daß das Programm später auf vielen verschiedenen mehr oder weniger kompatiblen MS-DOS-Rechnern laufen kann.

In diesem Fall gibt es außerdem eine einfachere Möglichkeit. In Anhang D finden Sie eine Übersicht über alle Textzeichen, die unter MS-DOS darstellbar sind. Dazu gehören auch die sogenannten Grafikzeichen, das sind die Zeichen mit den ASCII-Nummern 176 bis 223; mit ihnen kann man auch ein Schema "zeichnen".

Mit der Standardprozedur *WriteLn* können sämtliche ASCII-Zeichen auf dem Bildschirm ausgegeben werden. Das, was geschrieben werden soll, muß der Prozedur als Parameter übergeben werden. Der Syntax der Prozedur lautet:

```
WriteLn([Var   datei:Text,]v1[,v2,...,vn]);
```

wobei Parameter in eckigen Klammern weggelassen werden können.
Die Prozedur schreibt die Parameter *v1* bis *vn* in eine Datei vom Typ *Text* oder, wenn keine Datei angegeben wird, auf den Bildschirm an die augenblickliche Cursor-Position. Danach wird der Cursor an den Anfang der nächsten Zeile gesetzt.

Die Parameter *v1* bis *vn* können Variablen oder Konstanten jedes einfachen Typs sein. Zusammengesetzte Typen wie Array oder Record sind dagegen nicht erlaubt. Werden mehrere Parameter übergeben, müssen sie durch Kommas getrennt sein.

Die Grafikzeichen für die senkrechten Linien und die Ecken gibt es nicht auf der Tastatur. Sie können sie aber mit der Alternate-Taste und dem Zahlenblock an der rechten Seite der Tastatur eingeben. Dazu müssen Sie die Alternate-Taste festhalten und gleichzeitig den ASCII-Wert auf dem Zahlenblock eingeben. Wenn die Alternate-Taste wieder losgelassen wird, erscheint das Zeichen. Um die obere rechte Ecke des Schemas auf den Bildschirm zu bringen, müssen Sie also die Alternate-Taste festhalten und dann die Zahl 191 auf dem Zahlenblock eingeben.Für die linke Ecke geben Sie 218 ein, und den senkrechten Strich erzeugen Sie mit 179.

Die Prozedur wird dann so aussehen:

```
Procedure schema;

Begin
WriteLn('┌────────────────────────────────────────────┐');
WriteLn('│       Buch - Verwaltungs - Programm        │');
WriteLn('├────────────────────────────────────────────┤');
WriteLn('│ Verfasser.: ...........................    │');
WriteLn('│ Titel.....: ...........................    │');
WriteLn('│ Verlag....: ...........................    │');
WriteLn('│ ISBN......: ...........................    │');
WriteLn('│ Erschienen: ...........................    │');
WriteLn('│ Sprache...: ...........................    │');
WriteLn('│ Seiten....: ...........................    │');
WriteLn('│ Schlagwort: ...........................    │');
WriteLn('│             ...........................    │');
WriteLn('│             ...........................    │');
WriteLn('│             ...........................    │');
WriteLn('│             ...........................    │');
WriteLn('└────────────────────────────────────────────┘');
End;
```

Als Platzhalter für die noch fehlenden Daten dienen normale Punkte.
Sie können natürlich auch andere Daten wählen. Wenn Sie ausschließlich deutschsprachige Bücher besitzen, können Sie natürlich das Feld Sprache weglassen. Genauso können Sie ein Feld mit Angaben über Illustrationen hinzufügen.

Wenn Sie die Prozedur im Editor eingegeben haben und sie dann mit *Run* starten, wird der Compiler sofort mit einer Fehlermeldung abbrechen. Eine Prozedur muß immer aus einem Hauptprogramm gerufen werden. Das Hauptprogramm ist hier ganz klein, es besteht nur aus dem Programmnamen, den beiden Wörtern Begin und End und dem Aufruf der Prozedur.

An dieser Stelle sollten Sie überlegen, ob Sie das gesamte Programm in einer oder mehreren Dateien abspeichern wollen. Sie können die Prozedur im Hauptprogramm unmittelbar vor dem Wort Begin einfügen. Im weiteren Verlauf dieses Kapitels kommen aber viele andere Prozeduren hinzu. Diese müssen alle in das Hauptprogramm eingefügt werden. Das Programm wird also sehr groß und damit weniger übersichtlich.

Eine andere Möglichkeit wäre, jede Prozedur in einer eigenen Datei abzuspeichern und den Compiler die Dateien zusammenfügen zu lassen. Das können Sie mit dem Compilerbefehl $I erreichen.

Compilerbefehle stehen wie Kommentare in geschweiften Klammern, aber unmittelbar nach der ersten Klammer folgen das Dollarzeichen "$" und ein Befehl. In Anhang B finden Sie eine Übersicht über diese Compilerbefehle.
Die Prozedur muß dann in einer eigenen Datei abgespeichert werden, z.B. mit dem Namen Schema.pas. Dieser Name muß hinter dem $I-Befehl angegeben werden. Die Datei muß entweder in dem aktuellen Verzeichnis gespeichert werden oder in dem Verzeichnis, das im Menüpunkt "Options/Directories/Include directories" angegeben ist.

Der $I-Befehl veranlaßt den Compiler, die Datei mit der Prozedur zu laden und sie im Hauptprogramm an Stelle des Compilerbefehls einzufügen. Das geschieht, bevor das Programm compiliert wird. Für das fertige Programm ist es also ohne Bedeutung, ob Sie die Prozedur im Hauptprogramm einfügen oder ob Sie den $1-Befehl benutzen. Indem Sie den Befehl benutzen, können Sie aber die einzelnen Dateien klein halten. Andererseits wird die Compilierung etwas länger dauern, weil der Compiler ständig Quelldateien nachladen muß.

Jetzt können wir das Hauptprogramm schreiben:

```
Program katalog;                   { Programmname }
{$1 SCHEMA.PAS}                    { Datei mit Schema-Prozedur einfügen }
Begin                              { Anfang des Hauptprogrammes }
Schema;                            { Prozedur wird gerufen }
End.                               { Ende des Hauptprogrammes }
```

Es bietet sich an, das Hauptprogramm unter dem Namen Katalog.pas abzuspeichern. Dieser Name wird im Menüpunkt "Compile/Primary file" angegeben. Jetzt können Sie das Programm mit *Run* starten.

Leider ist das Ergebnis nicht ganz so, wie man es sich wünscht. Das Schema wird auf dem DOS-Bildschirm gezeigt. Deshalb stehen darüber noch einige DOS-Befehle. Bevor das Schema gezeigt wird, muß der Bildschirm also gelöscht werden. Das erledigt die Prozedur *ClrScr.* Dafür wird in der Prozedur unmittelbar nach dem Wort Begin folgende Zeile eingefügt:

ClrScr;

Diese Prozedur löscht den gesamten Bildschirm und setzt den Cursor in die linke obere Ecke. Die Prozedur stammt aber aus dem Unit *Crt,* deshalb muß im Hauptprogramm ein Uses-Block eingefügt werden. Sie müssen hierzu das Hauptprogramm erneut in den Editor laden und die folgende Zeile unmittelbar hinter dem Programmnamen einsetzen:

Uses Crt;

Wenn das Programm jetzt wieder gestartet wird, erscheint das Schema in der

oberen linken Ecke eines leeren Bildschirms. Es würde aber schöner aussehen, wenn es in der Mitte des Bildschirms stünde. Das kann mit Hilfe der Prozedur *Window* erreicht werden. Mit dieser Prozedur wird die Ausgabe auf dem Bildschirm auf ein bestimmtes Fenster begrenzt. Die Syntax lautet:

```
Window(x1,y1,x2,y2 <byte>);
```

Die vier Parameter bezeichnen die Koordinaten der linken oberen und rechten unteren Ecke des Fensters. Nachdem diese Prozedur ausgeführt ist, arbeiten *WriteLn* und viele andere Prozeduren nur innerhalb des Fensters. In das Hauptprogramm wird deshalb unmittelbar nach *ClrScr* die folgende Zeile eingefügt:

Window(l7,l,63,25) ;

Damit sieht das Hauptprogramm so aus:

```
Program katalog;
Uses Crt;
{$1 SCHEMA.PAS}
Begin
ClrScr;
{ Programmname }
{ wegen ClrScr }
{ Datei mit Schema-Prozedur einfügen }
{ Anfang des Hauptprogrammes }
{ Bildschirm löschen }
Window(l7,l,63,25);
schema;
End.
{ Fenster definieren }
{ Prozedur wird gerufen }
{ Ende des Hauptprogrammes }
```

Wenn Sie jetzt wieder das Programm starten, erscheint das Schema in der oberen Mitte des Bildschirms; alles ist, wie es sein sollte.

4.2 Daten eingeben

Als nächstes brauchen wir eine Prozedur, mit der die Daten für ein Buch eingegeben werden können. In dieser Prozedur kann die Standardprozedur *ReadLn* verwendet werden. *ReadLn* liest eine oder mehrere Zeilen aus einer Datei oder von der Tastatur und verbindet sie mit Variablen. Die Syntax lautet:

```
ReadLn([Var f <Text>,] Var v1[,v2,...,vn]);
```

Die Syntax ist also der von WriteLn sehr ähnlich, allerdings sind hier nur Variablen als Parameter zugelassen.
Bevor die Prozedur geschrieben werden kann, müssen deshalb die notwendigen Variablen deklariert werden. Die Variablen werden später in anderen Prozeduren gebraucht, deshalb werden sie global deklariert. Die Deklaration soll darum im Hauptpogramm nach der Uses-Deklaration eingefügt werden.
Erst wird ein Record definiert. Dieser soll alle Daten eines Buches aufnehmen können und wird so aussehen:

```
Type buch = Record
Verfasser,titel,verlag,erschienen : String[30];
isbn,spräche: String[15];
selten : String[5];
schlagwort : Array[l..5]Of String[30];
End;
```

Alle Elemente sind als *Strings* definiert. Man könnte natürlich auch die Seitenzahl als *Word* definieren. Wenn *ReadLn* mit einer numerischen Variablen als Parameter gerufen wird und der Benutzer etwas eingibt, was nicht als eine Zahl interpretiert werden kann, ist das Ergebnis allerdings eine Fehlermeldung. Bei Strings sind dagegen alle Eingaben erlaubt. Man kann also einen möglichen Programmabbruch vermeiden. Später im Programm kann man dann immer noch die Eingaben überprüfen und eventuell korrigieren.

Nachdem der Typ definiert ist, kann die Variable deklariert werden:

```
Var buch1 : buch;
```

Auch diese Zeile muß im Hauptprogramm stehen, nach der Typendefinition aber vor dem $I-Befehl.

Wenn *ReadLn* Zeichen von der Tastatur empfängt, werden die Zeichen gleichzeitig auf dem Bildschirm angezeigt. Das ist hier auch erwünscht, allerdings sollten die Zeichen an den richtigen Stellen in dem Schema gezeigt werden. Um das zu erreichen, muß der Cursor erst in dem Schema plaziert werden. Dazu gibt es die Standardprozedur *GotoXY.* Die Syntax hierfür lautet:

```
GotoXY(x,y <Byte>);
```

Die Prozedur plaziert den Cursor in Spalte x und Zeile y. Beide Parameter sind relativ zu einem mit Window definierten Fenster. Damit kann die Eingabeprozedur geschrieben werden.

```
Procedure eingabe;                            { Name der Prozedur }
Var i : ShortInt;                             { Für For-To-Do-Schleife }
Begin
     With buch1 Do Begin
      GotoXY(14,4);       ReadLn(Verfasser);
      GotoXY(14,5);       ReadLn(titel);
      GotoXY(14,6);       ReadLn(verlag);
      GotoXY(14,7);       ReadLn(isbn);
      GotoXY(l 4,8);      ReadLn(erschienen)
      GotoXY(14,9);       ReadLn(spräche);
      GotoXY(14,10);      ReadLn(selten);
      For i:=l To 5 Do Begin
        GotoXY(14,i+10);     ReadLn(schlagwort[i]);
```

```
            End;                    { For To Do }
        End;                        { With Buch1 o }
End;                                { Prozedure Eingabe }
```

Hier ist es sehr sinnvoll, zwei Befehle in eine Zeile zu schreiben. Sowohl *GotoXY* als auch *ReadLn* sind ganz kurz und gehören eng zusammen.

Neu ist auch der Befehl *With-Do.* Wenn mehrere Elemente desselben Records angesprochen werden sollen, kann das Programm mit diesem Befehl vereinfacht werden. Normalerweise werden die Elemente eines Records mit dem Namen des Records und dem Namen des Elements angesprochen. Es müßte also heißen:
`ReadLn(Buch1.Verfasser);`

Der Befehl *With-Do* veranlaßt aber den Compiler, jede Variable zunächst als Element des angegebenen Records zu betrachten. Damit kann der Name des Records ohne Risiko weggelassen werden. Nur wenn der Record kein Element mit diesem Namen enthält, wird der Compiler woanders suchen. Das gilt hier für die lokale Variable i.

In der Prozedur gibt es insgesamt drei verschachtelte Begin-End-Blöcke. Hier ist es sehr wichtig zu beachten, wo welcher Block anfängt und wo er wieder aufhört. Um das zu erleichtern, werden die Befehle innerhalb eines Blockes normalerweise um drei Plätze nach rechts verschoben. Außerdem sollten Sie bei jedem End immer als Kommentar angeben, was zu Ende geht. Wenn die Module etwas größer werden, kann man nämlich sehr schnell die Übersicht verlieren.

4.3 Daten anzeigen

Daten, die einmal eingegeben sind, müssen natürlich auch später in dem Schema wieder angezeigt werden können. Diese Prozedur wird der Prozedur *Eingabe* sehr ähnlich. Einzige Änderungen: Statt *ReadLn* muß überall *WriteLn* stehen, und die Prozedur muß einen anderen Namen haben.

Damit können Sie die neue Prozedur sehr einfach erstellen. Erst laden Sie die Eingabeprozedur, (Datei Eingabe.pas). Wählen Sie dann den Menüpunkt "File/ Write to" und speichern Sie die Prozedur unter dem Namen Ausgabe.pas ab. Jetzt müssen Sie den Namen der Prozedur von Eingabe in Ausgabe ändern.

Schließlich müssen überall *ReadLn* in *WriteLn* geändert werden; auch dafür gibt es eine einfache Möglichkeit: Halten Sie die Ctrl-Taste fest, betätigen Sie dann erst die Taste Q und dann A. In der oberen linken Bildschirmecke erscheint das Wort "Find:" (suchen), hier schreiben Sie jetzt *ReadLn.* Wenn Sie **Return** drücken, erscheint an derselben Stelle "Replace with:" (ersetze durch). Hier schreiben Sie *WriteLn.* Danach erscheint das Wort "Options:", und hier schreiben Sie *gn.*. Dabei

steht g für "global", das heißt, es soll überall im Text gesucht und ersetzt werden, und n bedeutet, daß die gefundenen Wörter ohne Rückfrage ersetzt werden sollen.

Danach ist die neue Prozedur fertig. Sie sieht so aus:

```
Procedure ausgabe;                             { Name der Prozedur }
Var i : ShortInt;                         { Für For-To-Do-Schleife }
Begin
   With buch1 Do Begin
      GotoXY(14,4);    WriteLn(verfasser);
      GotoXY(14,5);    WriteLn(titel);
      GotoXY(14,6);    WriteLn(verlag);
      GotoXY(14,7);    WriteLn(isbn);
      GotoXY(14,8);    WriteLn(erschienen);
      GotoXY(14,9);    WriteLn(sprache);
      GotoXY(14,10);   WriteLn(seiten);
      For i:=1 To 5 Do Begin
        GotoXY(14,i+10);    WriteLn(schlagwort[i]);
      End;                                          { For To Do }
   End;                                         { With Buch1 Do }
End;                                         { Prozedur Eingabe }
```

Um diese und die vorige Prozedur ausprobieren zu können, muß das Hauptprogramm folgendermaßen geändert werden:

```
Program katalog;                                { Programmname }
Uses Crt;                             { wegen Window und GotoXY }
Type buch = Record
              verfasser,titel,verlag,erschienen : String[30];
              isbn,sprache: String[15];
              seiten : String[5];
              schlagwort : Array[1..5]Of String[30];
           End;
Var buch1 : buch;
{$I SCHEMA.PAS}        { Datei mit Schema-Prozedure einfügen }
{$I EINGABE.PAS}      { Datei mit Eingabe-Prozedure einfügen }
{$I AUSGABE.PAS}      { Datei mit Ausgabe-Prozedure einfügen }
Begin                            { Anfang des Hauptprogrammes }
   schema;                          { Prozeduren werden gerufen }
   eingabe;
   schema;
   ausgabe;
End.                              { Ende des Hauptprogrammes }
```

Zuerst wird das Schema gezeichnet. Dann wird die Prozedur Eingabe gerufen. Wenn alle Eingaben gemacht sind, wird das Schema neu gezeichnet und die gemachten Eingaben werden angezeigt.

4.4 Ein Menü für das Programm

In dem obenstehenden Hauptprogramm werden die verschiedenen Untermodule in einer festen Reihenfolge gerufen. Sinnvoller ist es, wenn der Benutzer entscheidet, wann Daten eingegeben, gezeigt, gespeichert oder ausgedruckt werden sollen. Der Benutzer muß auch bestimmen können, wann das Programm abgebrochen wird.

Das Programm soll deshalb in einer großen Schleife laufen, bis der Benutzer eine Unterbrechung vornimmt. Innerhalb der Schleife soll der Benutzer dann zu den verschiedenen Untermodulen verzweigen können.

Damit ist es möglich, ein Ablaufdiagramm über das Programm zu zeichnen. *Abb. 2* zeigt das Diagramm. Die einzelnen Symbole sind in der DIN-Norm 66001 definiert.

Der Programmablauf beginnt bei dem Wort Begin, läuft dann nach unten in die Schleife an der linken Seite des Diagramms. Die Rhomben in der Mitte sind Abzweigungen, in Turbo Pascal wären dies z.B. If-Then-Else-Befehle. Rechts sind die verschiedenen Untermodule und der Programmabschluß, zu dem der Benutzer verzweigen kann.

Die meisten Programme sind nach diesem Prinzip aufgebaut. Ein Programm kann natürlich mehr oder weniger Untermodule haben. Das ändert aber nichts an dem prinzipiellen Aufbau.

Das nächste Problem ist, das Diagramm in Programmbefehle umzusetzen. In Turbo Pascal ist dies relativ einfach. Das Programm soll laufen, bis es vom Benutzer unterbrochen wird. Hierzu eignet sich eine Repeat-Untü-Schleife. Innerhalb der Schleife soll zu den verschiedenen Untermodulen abgezweigt werden. Das könnte mit einer Reihe von If-Then-Else-Befehlen geschehen. Einfacher ist es aber mit einem Case-Befehl.

Was fehlt, ist dann nur noch der Ausdruck im Case-Befehl. Der einzige Weg, durch den der Benutzer dem Programm seine Wünsche mitteilen kann, ist über die Tastatur. Das Programm kann seinerseits mit der Funktion *ReadKey* ermitteln, welche Taste gedrückt wurde. Damit muß das Hauptprogramm über den folgenden Rahmen aufgebaut werden.

BEGIN

END

Abb. 2: Ablaufdiagramm

```
Program katalog;
Uses Crt;                                          { wegen ReadKey }
Var taste : Char;
   stop : Boolean;
Begin
   stop:=False;
   Repeat                                        { Schleifenanfang }
      taste:=Readkey;
      Case taste Of
         ......
         ......
      End;                                         { Case Taste of }
   Until stop;
End.                                                    { Program }
```

Als nächstes müssen wir überlegen, wie wir dem Benutzer mitteilen können, welche Taste er drücken soll, um eine bestimmte Funktion auszulösen. Dazu gibt es prinzipiell zwei Möglichkeiten. Die eine ist, ein Handbuch zu schreiben, die andere und bessere, ein Menü in das Programm einzubauen.

Die einfachste Form eines Menüs ist eine Liste mit den verschiedenen Funktionen und den dazugehörigen Tasten. Wenn Sie die Menüs von Turbo Pascal betrachten, werden Sie sehen, daß ein Buchstabe etwas heller als die anderen ist. Mit diesem Buchstaben kann der Benutzer die Funktion wählen.

Etwas Ähnliches kann man auch mit Turbo Pascal machen. Dazu gibt es die beiden folgenden Standardprozeduren:

```
LowVideo;
HighVideo;
```

Beide werden ohne Parameter gerufen. Der Ergebnis ist, daß Zeichen, die danach geschrieben werden, etwas dunkler bzw. heller auf dem Bildschirm erscheinen. Auf Zeichen, die schon auf dem Bildschirm zu sehen sind, haben die Prozeduren allerdings keinen Einfluß.

Um den ersten Buchstaben eines Wortes heller erscheinen zu lassen, müssen Sie also erst *HighVideo* rufen, dann den ersten Buchstaben schreiben, *LowVideo* rufen und den Rest des Wortes schreiben. In diesem Falle ist es aber nicht sehr sinnvoll, *WriteLn* zu benutzen. Entweder würde das Wort auf zwei Zeilen verteilt oder Sie müßten *GotoXY* zweimal rufen.

Die Prozedur *WriteLn* ist allerdings nur eine Erweiterung der allgemeineren Prozedur *Write.* Die beiden Prozeduren haben dieselbe Syntax und verlangen denselben Parameter. Auch die Wirkung ist fast gleich. Der einzige Unterschied ist, daß der Cursor bei *Write* nicht in die nächste Zeile gesetzt wird, sondern unmittelbar nach dem zuletzt geschriebenen Zeichen stehen bleibt.

Um das Wort "Beispiel" in der gewünschten Weise zu schreiben, müssen Sie also die folgenden Befehle benutzen:

```
HighVideo;
Write ( 'B');
LowVideo;
WriteLn('eispiel');
```

Damit kann ein Menü erstellt werden. Erst muß das Schema wieder ein wenig nachlinks verschoben werden, um Platz auf dem Bildschirm zu schaffen. Deshalb müssen die Parameter der Prozedur Window geändert werden. Laden Sie dazu die Prozedur Schema und ändern Sie den ersten Parameter von Window von 17 auf 7.

Jetzt kann das Menü auf der rechten Seite des Bildschirms untergebracht werden. Die Befehle für das Menü werden selbstverständlich in einer eigenen Prozedur untergebracht, die so aussieht:

```
Procedure menue;
Begin
   Window(60,1,79,12);
   WriteLn ('┌──────────────┐');
   Write   ('│'); HighVideo; Write('L'); LowVideo; WriteLn('aden      │');
   Write   ('│'); HighVideo; Write('S'); LowVideo; WriteLn('peichern  │');
   Write   ('│'); HighVideo; Write('E'); LowVideo; WriteLn('ingeben   │');
   Write   ('│e'); HighVideo; Write('R'); LowVideo; WriteLn('ste      │');
   Write   ('│'); HighVideo; Write('N'); LowVideo; WriteLn('ächste    │');
   Write   ('│s'); HighVideo; Write('U'); LowVideo; WriteLn('chen     │');
   Write   ('│'); HighVideo; Write('A'); LowVideo; WriteLn('usdrucken │');
   Write   ('│'); HighVideo; Write('Q'); LowVideo; WriteLn('uit       │');
   WriteLn ('└──────────────┘');
   Window(7,1,53,20);
End;
```

Um besonders deutlich zu machen, mit welcher Taste die verschiedenen Funktionen gewählt werden können, sind diese Buchstaben nicht nur heller, sondern werden auch großgeschrieben. Mit einer Taste kann man natürlich nur eine Funktion wählen. Deshalb müssen die hervorgehobenen Buchstaben alle unterschiedlich sein. Um das zu erreichen, war es hier notwendig, in zwei Fällen nicht den ersten, sondern den zweiten Buchstaben des Wortes zu wählen.

Bevor die Prozedur abgeschlossen wird, muß Window noch einmal gerufen werden. Sonst würden die eingegebenen Daten nicht in dem Schema, sondern in dem Menü erscheinen.

Nachdem jetzt feststeht, welche Funktionen mit welchen Tasten gerufen werden sollen, kann auch das Hauptmodul fertig geschrieben werden, oder doch zumindest fast fertig.

```
Program katalog;                                  { Programmname }
Uses Crt;                              { wegen Window und GotoXY }
Type buch = Record
               verfasser,titel,verlag,erschienen : String[30];
               isbn,sprache: String[15];
               seiten : String[5];
               schlagwort : Array[1..5]Of String[30];
            End;
```

```
Var taste : Char;
    stop  : Boolean;
    buch1 : buch;
{$I SCHEMA.PAS}                             { Datei mit Schema-Prozedur einfügen }
{$I MENU.PAS}                                 { Datei mit Menü-Prozedur einfügen }
Begin                                             { Anfang des Hauptprogrammes }
    schema;                                          { Schema wird gezeichnet }
    menue;                                             { Menü wird gezeichnet }
    stop:=False;
    Repeat
        taste:=UpCase(Readkey);
        Case taste Of
            'L' : ;                                            ( Datei laden }
            'S' : ;                                        { Datei speichern }
            'E' : ;                                         { Daten eingeben }
            'R' : ;                                     { erstes Buch zeigen }
            'N' : ;                                   { nächstes Buch zeigen }
            'U' : ;                                            { Buch suchen }
            'A' : ;                                       { Liste ausdrucken }
            'Q' : Stop:=True;                            { Programm beenden }
        End;
    Until stop;
End.                                             { Ende des Hauptprogramms }
```

Damit kann das Programm gestartet werden. Da die einzelnen Funktionen noch nicht fertig sind, können sie natürlich auch nicht gerufen werden. Sie können aber ganz einfach wie hier die Plätze innerhalb des Case-Befehls frei lassen und die Befehle dann nach und nach einfügen, wenn die einzelnen Prozeduren fertig sind. Allerdings sollte man unbedingt wie hier die Abbruch-Funktion einfügen. Die Repeat-Until-Schleife ist sonst endlos.

4.5 Kontrollfrage an den Benutzer

Bleiben wir einen Moment bei der Abbruchfunktion des Programmes. Wirklich gute Programme unterscheiden sich unter anderem durch die Benutzerfreundlichkeit. Dazu gehört, daß es dem Benutzer erlaubt ist, Fehler zu machen, ohne daß dies schwerwiegende Folgen hat.
Wenn ein Programm beendet wird, gehen alle nicht gespeicherten Daten verloren. Deshalb sollte es niemals möglich sein, ein Programm durch einen einfachen Tippfehler zu unterbrechen. Bevor das Programm unterbrochen wird, sollte man daher den Benutzer nochmals fragen, ob er wirklich das Programm beenden will.

Auch in anderen Fällen kann es notwendig sein, den Benutzer um eine Bestätigung zu bitten. Deshalb sollte diese Nachfrage in einem selbständigen Modul untergebracht werden.

Das Modul soll generell verwendbar sein. Um das zu erreichen, wird es als Funktion geschrieben. Die Frage, die gestellt werden soll, kann dann als Parameter übergeben und die Antwort des Benutzers als Rückgabewert geliefert werden. Der Parameter muß vom Typ *String* sein. Für den Rückgabewert könnten wir verschiedene Typen wählen. Hier wird der Typ *Boolean* benutzt. Das hat den Vorteil, daß die Funktion als Ausdruck direkt in einen If-Then-Else-Befehl eingebunden werden kann.

Wenn die Funktion den Namen *Akzept* hat, sieht die letzte Zeile innerhalb des Case-Befehls im Hauptprogramm so aus:

```
'Q' : If akzept ('Wirklich das Programm verlassen' ) Then stop:=True;
```

Daraus können Sie auch erkennen, was innerhalb der Funktion passieren soll. An den übergebenen Stringparameter muß der Text " (J/N)?" angehängt werden. Dann wird der String auf dem Bildschirm gezeigt, das Programm wird angehalten, bis der Benutzer die Taste J oder N betätigt. War es die Taste J, wird als Rückgabewert *True* geliefert. Schließlich muß die Frage wieder vom Bildschirm entfernt werden.

Die Frage soll in einem kleinen Fenster erscheinen. Dieses Fenster wird auch später in anderen Prozeduren gebraucht und soll deshalb in eine eigene Prozedur ausgegliedert werden. Da diese Prozedur innerhalb der Funktion gerufen wird, muß sie im Programm vor der Funktion stehen.

```
Procedure fenster;
   Window(10,18,69,21);
   WriteLn('┌──────────────────────────────────────────────────────┐');
   WriteLn('│                                                      │');
   WriteLn('└──────────────────────────────────────────────────────┘');
End;
```

Und hier ist die Funktion:

```
Function akzept(frage : String):Boolean;
Var zeile : String[56];
   taste : Char;
Begin
   zeile:=Copy(frage,1,50);
   zeile:=Concat(zeile,' (J/N)');
```

```
    fenster;
    GotoXY(30-Length(zeile) Div 2,2);
    Write(zeile);
    Repeat
       taste:=UpCase(ReadKey);
    Until (taste='J') Or (taste='N');
    ClrScr;
    Window(7,1,53,20);
    If taste='J' Then akzept:=True
    Else akzept:=False
End;
```

Die Funktion enthält drei neue Standardfunktionen, die nachfolgend besprochen werden.

In das kleine Fenster paßt nur eine Zeile mit höchstens 56 Zeichen. Wenn " (J/N) ? " hinzukommt, darf die Frage nicht mehr als 50 Zeichen enthalten. Damit kein Fehler entstehen kann, muß die Frage demnach zuerst auf diese Länge begrenzt werden.

Das wird hier mit der Standardfunktion *Copy* gemacht. Diese Funktion kopiert einige Zeichen aus einem String in einen anderen String. Die Syntax lautet:

```
S1<String>:=Copy(s2<String>,anfang,anzahl<Integer>);
```

Aus dem String s2 werden anzahl Zeichen beginnend bei anfang in den String sl kopiert. Mit dem Befehl "zeile:=Copy(frage,l,50);" werden also die ersten 50 Zeichen aus dem String frage in den String zeile kopiert.

Danach soll der Text" (J/N)?" an diesen String angehängt werden. Dazu wird die Standardfunktion Concat benutzt. Die Syntax hierfür lautet:

```
s<String>:=Concat(s1, s2, ...,sn<String>);
```

Damit werden die Strings s1 bis sn zu einem String s verbunden. Ein String kann in Turbo Pascal niemals mehr als 255 Zeichen enthalten. Wenn der Gesamtstring s mehr Zeichen enthält, wird er deshalb auf diese Länge abgeschnitten.

Nachdem die beiden Strings verbunden sind, wird am unteren Teil des Bildschirms ein neues Fenster definiert. Darin wird ein Rahmen gezeichnet.

Jetzt soll die Frage innerhalb des Rahmens zentriert geschrieben werden. Dazu muß man zuerst die Länge des Strings feststellen. Das geschieht mit der Standardfunktion *Length.*

Wie in Abschnitt 3.3 erklärt, hat ein String immer eine maximale und eine dynamische Länge. Die Funktion *Length* liefert die dynamische Länge eines String. Die Syntax ist:

länge_von_s<Integer>:=Length(s<String>);

Alle anderen Befehle in der Funktion *akzept* sind schon bekannt. Die Funktion muß dann nur noch in eine eigene Datei abgespeichert werden. Diese wird mit einem $I-Befehl in das Hauptprogramm eingebunden.

4.6 Datei öffnen

Nachdem das Hauptprogramm zumindest als Rahmen fertig ist, müssen die einzelnen Prozeduren geschrieben werden. Die erste Prozedur soll Daten von Diskette laden.

In Verbindung mit der Eingabeprozedur wurde ein Record *buch* definiert. Dieser Record kann die Daten eines Buches aufnehmen. Das Programm soll aber viele Bücher verwalten. Deshalb muß eine Variable vom Typ *Array Ofbuch* oder *File Of buch* deklariert werden.

Der Vorteil eines Arrays ist, daß alle Daten im Hauptspeicher sind. Dadurch arbeiten besonders Such- und Sortierfunktionen sehr schnell. Auf der anderen Seite ist die mögliche Datenmenge begrenzt. Ein Array darf nämlich höchstens 65520 Bytes enthalten.

Die Daten in einem File werden dagegen in einer Datei abgelegt. Damit wird zwar die Handhabung wesentlich langsamer, es lassen sich aber auch wesentlich größere Datenmengen aufnehmen.

Sie müssen also überlegen, welche Datenmengen verarbeitet werden sollen. Der Record buch hat eine Gesamtgröße von 305 Bytes. Ein Array könnte somit nur 214 Elemente vom Typ buch umfassen.

Deshalb werden wir hier eine Variable vom Typ *File* deklarieren. Dazu muß im Hauptprogramm die folgende Variablendeklaration eingefügt werden:

```
buecher  :  File  Of  buch;
```

Der Inhalt einer Variablen vom Typ *File* wird in einer externen Datei abgelegt. Der Name dieser externen Datei wird dem Compiler mit der Prozedur *Assign* mitgeteilt. Die Syntax ist:

```
Assign(datei<File>,dateiname<String>);
```

datei ist eine deklarierte Filevariable, *dateiname* ist der Name der externen Datei, eventuell mit komplettem Pfad.
Zum Öffnen dieser Datei gibt es zwei Standard-Prozeduren:

Reset (Var datei<File>, [record__groesse<Word>]) ;
Rewrite(Var datei<File>,[record_groesse<Word>]);

datei ist eine deklarierte Filevariable, die mit Hilfe von *Assign* mit einer externen Datei verbunden wurde. Ist die Variable nur als *File* deklariert, muß die Größe der einzelnen Elemente mit dem Parameter *record_groesse* angegeben werden. Die Variable *buecher wurde* aber als *File Ofbuch* deklariert, ein Element ist also ein Record vom Typ *buch.* Deshalb kann der zweite Parameter hier weggelassen werden.

Der Unterschied zwischen den beiden Prozeduren liegt in der Wirkung. *Reset* öffnet eine vorhandene externe Datei und setzt den Dateicursor an ihren Anfang. Ist keine Datei mit dem angegebenen Namen vorhanden, entsteht ein lO-Fehler. *Rewrite* dagegen schafft eine neue Datei mit dem angegebenen Namen. Ist eine Datei mit diesem Namen schon vorhanden, wird sie gelöscht und eine neue, leere Datei mit demselben Namen wird erstellt.

Wir müssen also eine Prozedur schreiben, mit der der Benutzer erst einen Dateinamen eingeben kann. Dieser Dateiname wird mit der Variablen *buecher* verbunden und mit *Reset* geöffnet. Entsteht dabei ein Fehler, können wir davon ausgehen, daß die Datei nicht vorhanden ist. Deshalb soll der Benutzer gefragt werden, ob eine neue Datei erstellt werden soll. Wenn er dies bestätigt, wird die Datei mit *Rewrite* geöffnet. Die einzige offene Frage ist, wie wir einen Fehler bei *Reset* abfangen und feststellen können.

Der Fehler, der bei *Reset* entstehen kann, ist ein lO-Fehler. Solche Fehler führen normalerweise zu einem Programmabbruch und einer entsprechenden Fehlermeldung. Mit dem Compilerbefehl {$!-} kann man einen Programmabbruch verhindern. Der Compilerbefehl muß vor der möglichen Fehlerstelle stehen. Mit der Standardfunktion IOResult kann man dann kontrollieren, ob ein Fehler entstanden ist. Die Syntax für diese Funktion lautet:

```
i_o_ergebnis<Word>:=IOResult;
```

IOResult liefert das Ergebnis der letzten IQ-Operation. Ist das Ergebnis 0, ist alles korrekt abgelaufen. Ist dagegen ein Fehler entstanden, liefert die Funktion die Nummer des Fehlers. In diesem Falle reicht es uns aus festzustellen, daß ein Fehler entstanden ist. Näheres zu Fehlern und ihrer Behandlung finden Sie in Kapitel 10.

Wenn die Prozedur in das Programm eingefügt wird, entsteht noch ein Problem. Versucht der Benutzer z.B. Daten einzugeben, bevor er eine Datei geöffnet hat, entsteht ein Fehler. Bei allen Ein- und Ausgabeprozeduren muß deshalb erst überprüft werden, ob eine Datei geöffnet ist. Ähnlich ist es, wenn der Benutzer erst mit einer Datei arbeitet und dann versucht, eine neue zu öffnen, ohne vorher die alte zu schließen. Die Prozedur *Assign* darf nämlich niemals bei einer offenen Datei verwendet werden.

Diese Probleme lassen sich aber leicht lösen, indem man eine globale Variable vom Typ *Boolean* deklariert. Darin wird der Zustand von *buecher* festgehalten. Ist die Variable *buecher* mit einer offenen Datei verbunden, bekommt die Boolean-Variable den Wert *True.* Dieser Wert kann dann überall nach Bedarf überprüft werden.

Im Hauptprogramm fügen wir entsprechend eine Variable ein:

```
offen : Boolean;
```

Diese Variable muß am Anfang des Hauptprogramms den Wert *False* zugewiesen bekommen. Jetzt kann die Prozedur zum Öffnen einer Datei geschrieben werden:

```
Procedure oeffnen;                                  { Name der Prozedur }
Var dateiName : String[60];                   { Variable für Dateinamen }
Begin
   fenster;      GotoXY(3,2);                  { Dateinamen eingeben }
   Write('Dateiname: '); ReadLn(DateiName);                      ClrScr;
   {$I-}                           { IO-Fehlerkontrolle ausschalten }
   If offen Then Close(buecher);      { Datei schließen wenn offen }
   Assign(buecher,dateiName);       { Variable und Datei verbinden }
   Reset(buecher);                                    { Datei öffnen }
   If IOResult=0 Then offen:=True
   Else If akzept('Neue Datei anlegen') Then Begin
           Rewrite(buecher);                          { Datei anlegen }
           If IOResult=0 Then offen:=True
           Else offen:=False;
        End;
   {$I+}                           { IO-Fehlerkontrolle einschalten }
   Window(7,1,53,20);
End;                                               { Ende der Prozedur }
```

Nur die Prozedur *Close* ist noch nicht besprochen. Damit wird die mit *datei* verbundene externe Datei aktualisiert und dann geschlossen. Die Syntax lautet:

```
Close(datei <File>);
```

datei ist eine Filevariable beliebigen Typs, die mit einer externen Datei verbunden und geöffnet ist. Ist *datei* nicht geöffnet, entsteht ein Laufzeitfehler.
Die Prozedur *oeffnen* muß wie alle anderen Prozeduren in einer Datei abgespeichert werden. Dann müssen Sie die Hauptdatei wieder laden. Hier wird die neue Datei mit einem Compilerbefehl eingefügt. Nun können Sie in den Case-Befehl den Name der Prozedur als Befehl einsetzen.

4.7 Datei schließen

Die letzte Prozedur war vielleicht ein wenig kompliziert. Dafür ist die folgende um so einfacher. Ist die Filevariable *buecher* geöffnet, wird sie wieder geschlossen, und die Variable *offen* bekommt den Wert *False.*

```
Procedure schliessen;
Begin
If offen Then Close(buecher) ;:
offen:=False;
End;
```

Eigentlich könnte man auf diese Prozedur verzichten, denn das Wort End im Hauptmodul schließt automatisch alle offenen Dateien.

4.8 Eingabe

Nachdem die Datei geöffnet ist, können die ersten Daten eingegeben werden. Wir hatten bereits eine Eingabeprozedur geschrieben. Sie enthält allerdings nur die Eingabe über die Tastatur. Für unsere Zwecke müssen die neuen Daten zudem in eine Datei eingefügt werden.

Das ist relativ einfach. Am Anfang dieses Kapitels wurde erwähnt, daß die Prozeduren *Write* und *WriteLn* keine Arrays oder Records schreiben können. Das gilt aber nur dann, wenn auf den Bildschirm oder in eine Filevariable vom Typ *Text* geschrieben wird.

Mit *Write* kann man aber auch ein Element in eine andere Filevariable schreiben. Das geht auch, wenn die Elemente des Files zusammengesetzte Typen sind. Sie können also mit *Write* ohne weiteres ein Element vom Typ *Buch* in die Filevariable *buecher* schreiben.

Wie auf dem Bildschirm wird an der augenblicklichen Cursorposition geschrieben. Um kein vorhandenes Element zu überschreiben, sollte das neue Element an das Ende der Datei angefügt werden. Dazu muß erst die Größe der Datei mit Hilfe der Funktion *FileSize* gefunden werden:

elemente<Longint>:=FileSize(datei<File>);

Der Funktion liefert die Anzahl der Elemente in *datei.*
Mit der Standardprozedur *Seek* ist es möglich, den Cursor innerhalb der Datei auf ein bestimmtes Element zu setzen. Die Syntax lautet:

```
Seek(Var datei<File>,position<Longint>);
```

Position ist ein *Longint*-Ausdruck und gibt die Nummer des Elements an, auf das der Cursor gesetzt werden soll. Dabei hat das erste Element die Nummer 0. Die Elemente der Filevariablen *datei* sind also von 0 bis *FileSize(datei)-!* numeriert. Soll ein neues Element an die Datei angehängt werden, muß es dementsprechend bei der Position *FileSize(datei)* geschrieben werden.

Die Bücher sollen aber nicht in zufälliger Reihenfolge in die Datei geschrieben, sondern alphabetisch nach Verfasser und Titel gespeichert werden. Wenn ein neues Buch zunächst an das Ende der Datei hinzugefügt wurde, muß es deshalb nachträglich an die richtige Stelle nach unten sortiert werden, d.h. das neue Buch muß jeweils mit der vorhergehenden Eintragung so lange verglichen und umgetauscht werden, bis es an der richtigen Stelle in der Datei angelangt ist.

Mit den Vergleichsoperatoren >,<,=,=£ etc. können Sie nicht nur Zahlen, sondern auch Strings vergleichen. Dabei werden diese aber nicht nach dem Alphabet, sondern nach der ASCII-Tabelle verglichen.

Das bedeutet, daß Kleinbuchstaben generell vor Großbuchstaben, die deutschen Sonderzeichen erst nach allen anderen Buchstaben, und zwar in der Reihenfolge: ü, ä, Ä, ö, Ö, Ü, ß, einsortiert werden. Insgesamt keine befriedigende Lösung. Mit der Funktion *UpCase* werden die Buchstaben a bis z in die entsprechenden Großbuchstaben umgesetzt. Damit könnte der Unterschied zwischen Groß- und Kleinschrift überwunden werden. Die Funktion hat aber keine Wirkung auf die deutschen Sonderzeichen. Außerdem wird nur ein Zeichen und kein String umgesetzt.

Wir lösen dieses Problem dadurch, daß wir uns eine eigene Funktion schreiben. Die Syntax dieser Funktion soll wie folgt aussehen:

```
zeile2<String>:=gcase(zeile1<String>) ;
```

Die Funktion muß die Strings so verändern, daß ein Vergleich von zwei umgesetzten Strings eine korrekte alphabetische Reihenfolge ergibt. D.h. Kleinbuchstaben sollen in Großbuchstaben und die Buchstaben ä, ö und ü in a, o und u umgesetzt werden, ß wird in ss aufgelöst.

Das Listing der programmierten Funktion sieht so aus:

```
Function gcase(zeile:String):String;            { Name der Funktion }
Var i,j     : Integer;
    ergebnis: String;
Begin
    j:=0;
    For i:=1 To Length(zeile) Do Begin         { Jedes Zeichen in zeile }
        Inc(j);
        Case zeile[i] of
```

```
          'ä','Ä' : ergebnis[j]:='A';
          'ö','Ö' : ergebnis[j]:='O';
          'ü','Ü' : ergebnis[j]:='U';
          'ß' : Begin
                   ergebnis[j]:='S';
                   Inc(j);
                   ergebnis[j]:='S';
                End;                                      { 'ß' : Begin }
      Else ergebnis[j]:=Upcase(zeile[i]);                     { a bis z }
      End;                                         { Case zeile[i]... }
    End;                                             { For i:=1 To.....}
    ergebnis[0]:=Chr(j);
    gcase:=ergebnis;                            { Rückgabewert liefern }
End;                                              { Ende der Funktion }
```

In Turbo Pascal ist der Typ String als *Array ofChar* definiert. Ein Zeichen in einem String kann deshalb als Element eines Arrays gelesen werden.

In dieser Funktion muß jedes Zeichen des Strings gelesen und umgesetzt werden. Das passiert in einer For-To-Do-Schleife. Die Schleife muß vom ersten bis zum letzten Zeichen laufen. Das erste Zeichen in einem String hat immer den Index 1.

Der Index des letzten Zeichens muß der dynamischen Länge des Strings entsprechen, oder einfacher gesagt, enthält der String 8 Zeichen und hat das erste Zeichen die Nummer l, dann muß das letzte Zeichen Nummer 8 sein. Die dynamische Länge eines Strings kann mit der Standardprozedur *Length* gefunden werden.

Innerhalb der For-To-Do-Schleife wird jedes Zeichen aus dem Parameter *zeile* untersucht, umgesetzt und in die lokale Stringvariable ergebnis eingefügt.

Der Buchstabe ß wird in ss aufgelöst. Deshalb kann es passieren, daß die beiden Strings nachher nicht dieselbe Länge haben. Daraus folgt, daß man nicht denselben Index für beide Strings benutzen kann. Hier wird für zeile der Index i und für ergebnis der Index j benutzt. Die Variable i ist gleichzeitig Index für die Schleife und wird dadurch automatisch bei jedem Durchlauf um den Wert l erhöht. Dagegen müssen wir, immer wenn ein Zeichen in ergebnis eingefügt wird, auch die Variablej erhöhen. Das könnte mit dem Befehl"j:=j+l" geschehen. Hier benutzen wir statt dessen die Standardprozedur *Inc.* Sie hat die folgende Syntax:

```
Inc(a<num>[,b<Longint>]);
```

a ist eine Variable eines beliebigen numerischen Typs. Durch die Prozedur wird a um den Wert b erhöht oder, wenn b weggelassen wird, um den Wert 1. Dementsprechend gibt es auch die Prozedur *Dec:*

```
Dec(a<num>[,b<Longint>]);
```

Dadurch wird a um den Wert b bzw. um l vermindert.

Neu ist auch die Zeile: "ergebnis[0]:=Chr(j);". Um diesen Befehl zu verstehen, muß man erst sehen, wie eine Stringvariable intern aufgebaut ist. Wie gesagt, ist dieser Typ als Array of Char definiert. Ein Array hat eine untere und eine obere Begrenzung. Bei einem String ist die untere Begrenzung immer 0. Die obere muß zwischen l und 255 liegen. Die Deklaration "String[80]" ist also eigentlich dasselbe wie "Array[0..80]Of Char".

In das erste Element eines Strings, also das Element mit dem Index 0, wird die dynamische Länge des Strings geschrieben. Danach folgen die einzelnen Zeichen. Das ist auch der Grund dafür, weshalb ein String höchstens 255 Zeichen enthalten kann. Jedes Element eines Strings ist nämlich ein Byte groß, und 255 ist die größte Zahl, die durch ein Byte dargestellt werden kann.

Nachdem alle Zeichen der Variablen *zeile* umgesetzt sind, enthält der String *ergebnis* insgesamt j Zeichen. Diese Zahl muß dann in *ergebnis[0]* geschrieben werden.

Da die Variable j vom Typ *Integer* ist und die Elemente eines Strings vom Typ *Cha* sind, würde der Befehl "ergebnis[0]:=j;" daher einen Compilerfehler hervorrufen.

Um das zu vermeiden, muß j erst mit der Standardfunktion *Chr* umgesetzt werden. *Chr* hat die Syntax:

```
c<Char>:=Chr(ascii_wert<Byte>) ;
```

Chr erwartet als Parameter eine Zahl im Bereich 0 bis 255 und liefert als Ergebnis das Zeichen, dessen ASCII-Wert dem Parameter entspricht. In Turbo Pascal gibt es auch die umgekehrte Funktion. Sie heißt *Ord:*

```
ascii_wert<Longint>:=Ord(c <Char>) ;
```

Diese Fuktion erwartet einen Parameter vom Typ *Char* und liefert als Ergebnis dessen ASCII-Wert. Ist c eine Variable vom Typ *Char* und *zeile* eine Variable vom Typ *String,* folgen aus dieser Definitionen:

```
c = Chr(Ord(C))
Length(zeile) = Ord(zeile[0])
```

Gehen wir jetzt zur Sortierprozedurzurück. Das einzig Neue dabeiist die Prozedur *Read.* Genau wie *Write* eine erweiterte Form von *WriteLn* ist, ist *Read* eine erweiterte Form von *ReadLn.* Die Syntax von *Read* und *ReadLn* ist gleich. Damit dürfte es wohl niemanden überraschen, daß man mit *Read* ein Element aus einer Filevariablen lesen, genauso wie man mit *Write* ein Element in eine Filevariable schreiben kann.

Und hier ist jetzt die Sortierprozedur:

```
Procedure sortieren;                              { Name der Prozedur }
Var ofset : Longint;
  buch1,buch2 : buch;
Begin
  If FileSize(buecher)<2 Then Exit;
  ofset:=FileSize(buecher)-2;
  Repeat
      Seek(buecher,ofset);
      Read(buecher,buch1);
      Read(buecher,buch2);
      If gcase(buch1.verfasser)<gcase(buch2.verfasser) Then Exit;
      If gcase(buch1.verfasser)=gcase(buch2.verfasser) Then
          If gcase(buch1.titel)<gcase(buch2.titel) Then Exit;
      Seek(buecher,ofset);
      Write(buecher,buch2);
      Write(buecher,buch1);
      Dec(ofset);
      Until ofset<0;
End;
```

Erst wird die Größe der Datei untersucht. Enthält sie weniger als zwei Elemente, ist es natürlich nicht möglich, die Datei zu sortieren, und die Prozedur wird in diesem Fall mit Exit verlassen.

Danach werden die zwei letzten Elemente der Datei gelesen und verglichen. Ist die Reihenfolge korrekt, so ist das neue Buch schon an der richtigen Stelle und die Prozedur kann verlassen werden. Sonst werden die zwei Bücher umgetauscht, die Variable ofset wird um eins vermindert und das neue Buch wird mit der nächsten Eintragung verglichen.

Jetzt fehlt nur noch eine kleine Prozedur, mit der das Schema vor der Eingabe geleert werden kann.

```
Procedure schema_leeren;                  { Name der Prozedur }
Var i : Integer;
Begin
   For i:=4 To 15 Do Begin
      GotoXY(14,i);  Write('........................................');
   End;                                  { For i:=4 To 15 ... }
End;                                                { Prozedur }
```

Zum Schluß muß die eigentliche Eingabeprozedur geschrieben werden. Nach dieser umfangreichen Vorarbeit ist dies aber kein Problem. Hier ist sie:

```
Procedure eingeben;                     { Name der Prozedur }
Begin
   If offen Then Begin
      schema_leeren;
      eingabe;                    { Daten werden eingegeben }
      Seek(buecher,FileSize(buecher));
      Write(buecher,buch1);    { Daten an Datei anhängen }
      Sortieren;             { Neue Element einsortieren }
   End;                                { If Offen Then ... }
End;                                    { Ende der Prozedur }
```

4.9 Suchfunktionen

Die Daten aus der Datei sollen auf dem Bildschirm angezeigt werden können. Dazu soll das Programm drei verschiedene Funktionen enthalten.

Mit der ersten Suchfunktion wird der erste Record aus einer Datei gezeigt. Die zweite zeigt den jeweils nächsten Record. Damit ist es dann möglich, von Anfang bis Ende durch die Datei zu blättern. Die dritte ist eine richtige Suchfunktion. Hier soll der Benutzer ein Schlagwort eingeben. Die Funktion zeigt dann alle dazu passenden Bücher.

Mit der Prozedur aus Abschnitt 4.3 kann man die Daten eines Buches auf dem Bildschirm zeigen. Die erste Suchfunktion ist damit relativ einfach. Der Cursor wird auf das erste Element der Datei gesetzt. Dann wird das Element mit Read gelesen und mit *ausgäbe* gezeigt.

In der zweiten Suchfunktion wird der Cursor auf das nächste Element der Datei gesetzt, das dann ebenfalls gelesen und gezeigt wird. Das Programm muß also immer festhalten, welches Element als letztes gezeigt wurde. Um das korrekt bewerkstelligen zu können, muß die Position des Cursors als globale Variable deklariert werden. Im Hauptprogramm muß darum die folgende Zeile eingefügt werden:

```
offset  :  Longint;
```

In den beiden Prozeduren wird ein Element aus der Datei gelesen und gezeigt. Dabei können leicht Fehler entstehen. Ist die Datei nicht geöffnet oder enthält sie noch keine Daten, kann natürlich auch nichts gelesen werden. Außerdem darf die Variable *offset* keinen negativen Wert annehmen, und sie darf auch nicht größer als die Größe der Datei werden. Wenn alle diese Probleme berücksichtigt wurden, sehen die beiden Prozeduren so aus:

```
Procedure erste;                          { Name der Prozedur }
Begin
   If offen And (FileSize(buecher)>0) Then Begin
      offset:=0;                           { erster Datensatz}
      Seek(buecher,offset);               { Cursor plazieren }
      Read(buecher,buch1);                     { Daten lesen }
      schema_leeren;
      ausgabe;                                { Daten zeigen }
   End;                                   { If Offen and ... }
End;                                              { Prozedur }
Procedure naechste;                       { Name der Prozedur }
Begin
   If offen And (FileSize(buecher)>0) Then Begin
      Inc(offset);                            { Nächstes Buch }
      If offset<0 Then ofset:=0;
      If offset>=FileSize(buecher) Then offset:=0;
      Seek(buecher,offset);               { Cursor plazieren }
      Read(buecher,buch1);                     { Daten lesen }
      schema_leeren;
      ausgabe;                                { Daten zeigen }
   End;                                   { If Offen and ... }
End;                                              { Prozedur }
```

Interessant ist auch die Zeile:

```
If offen And (FileSize(buecher)>0) Then Begin
```

Die Funktion *FileSize* darf nur auf eine offene Datei angewendet werden. Ist die Datei nicht geöffnet, müßte die Auswertung des Ausdrucks also zu einem Fehler führen. Daß dies nicht passiert, liegt an der sogenannten "Short circuit boolean evaluation".

In dem Menü "Options/Compiler" gibt es die Funktion "Boolean evaluation". Hier steht als Standard "Short Circuit", das aber in "Complete" geändert werden kann.

Bei "Short Circuit" wird ein zusammengesetzter Booleanausdruck nur so weit ausgewertet, bis das Gesamtergebnis eindeutig feststeht, bzw. bei Evaluierung weiterer Booeanausdrücke sich dieses nicht mehr ändern kann. Bei "Complete" wird ein solcher Ausdruck immer vollkommen ausgewertet.

Im obenstehenden Beispiel hat dies weitgehende Folgen. Ist die Datei buecher nicht geöffnet, hat die Variable offen den Wert *Fähe.* Das bedeutet, daß der gesamte Ausdruck niemals den Wert *True* haben kann. Der zweite Teil des Ausdrucks (FileSize(buecher)>0) hat somit keine Bedeutung und wird nicht untersucht. Die Funktion *FileSize* wird also nur gerufen, wenn die Datei offen ist.

Haben Sie dagegen die Einstellung im Menü zu "Complete" geändert, wird der gesamte Ausdruck untersucht und das Ergebnis ist ein Fehler.
Ein Fehler könnte auch entstehen, wenn die beiden Teilausdrücke umgetauscht würden. Die folgende Zeile ist somit falsch:

```
If (FileSize(buecher)>0) And offen Then Begin
```

In der dritten Suchfunktion soll nach Übereinstimmung mit einem eingegebenen Schlagwort gesucht werden. Der Benutzer soll erst ein Schlagwort eingeben. Dazu können Sie die Prozeduren *Fenster* und *ReadLn* benutzen. Dann soll die gesamte Datei durchsucht werden, offset bekommt hier zunächst den Wert 0 und wird dann nach und nach erhöht, bis die Größe der Datei erreicht ist. Die Schlagwörter der Bücher sollen dann mit dem eingegebenen Schlagwort verglichen werden. Bei Übereinstimmung sollen die Daten gezeigt werden.

Wir könnten die beiden Strings mit dem Vergleichoperator (=) vergleichen. Dann müßte der Benutzer aber das gesamte Schlagwort eingeben und nicht nur einen Teil davon. Deshalb sollte stattdessen die Funktion *Pos* benutzt werden. Die Syntax der Funktion ist:

```
position<Byte>:=Pos(sl,s2<String>);
```

Die Funktion untersucht, ob der String sl in String s2 enthalten ist. Ist dies der Fall, wird als Ergebnis die Position des ersten Zeichens aus sl in s2 geliefert. Ist sl dagegen nicht in s2 enthalten, wird die Zahl 0 geliefert.

Damit kann auch die dritte Suchfunktion geschrieben werden:

```
Procedure suchen;                                 { Name der Prozedur }
Var schlagWort : String[60];
   i          : Integer;
   taste      : Char;
Begin
   If (Not offen) Or (FileSize(buecher)=0) Then Exit;
   fenster;                 GotoXY(3,2);
   Write('Schlagwort: ');   ReadLn(schlagWort);
   ClrScr;                  Window(7,1,53,20);
   SchlagWort:=gcase(schlagWort);
   ofset:=0;
   While ofset<FileSize(buecher) Do Begin
      Seek(buecher,ofset);                      { Cursor plazieren }
      Read(buecher,buch1);                           { Daten lesen }
      For i:=1 To 5 Do
         If Pos(schlagWort,gcase(buch1.schlagwort[i]))>0 Then Begin
            schema_leeren;
```

```
                ausgabe;                           { Daten zeigen }
               taste:=ReadKey;                           { Pause }
        End;                                { If Pos(Schlagw.... }
        Inc(ofset);                               { Nächstes Buch }
      End;                                    { While ofset.... }
End;                                                 { Prozedur }
```

Bevor die beiden Strings verglichen werden, werden sie mit der Funktion gcase behandelt. Damit werden eventuelle Unterschiede in Groß- und Kleinschreibung umgangen.

4.10 Druckerausgabe

Unter Turbo Pascal werden die parallelen und seriellen Schnittstellen grundsätzlich wie Dateien behandelt. Die Programmierung ist dieselbe, egal ob mit einer Diskettendatei, einem Modem oder einem Drucker gearbeitet wird.

Jede Schnittstelle hat einen reservierten Dateinamen. Die drei möglichen parallelen Schnittstellen heißen LPT1, LPT2 und LPT3. Die zwei möglichen seriellen Schnittstellen COM1 und COM2. Ist der Computer mit weniger Schnittstellen ausgerüstet, sind es immer die größeren Nummern, die fehlen.

Ein Drucker ist normalerweise mit LPT1 verbunden. Wollen Sie etwas auf dem Drucker schreiben, müssen Sie erst eine Filevariable vom Typ *Text* deklarieren. Die Filevariable und LPT1 werden dann mit Hilfe von *Assign* verbunden und mit *Rewrite* geöffnet. Danach kann die Druckerausgabe mit *Write* oder *WriteLn* durchgeführt werden.

```
Procedure drucken;
Var drucker      : Text;
   xpos,ypos,i : Integer;
Const seitenLaenge = 66;                       { 66 Zeilen/Seite }
Begin
   If (Not offen) Or (FileSize(buecher)=0) Then Exit;
   Assign(drucker,'LPT1');
   Rewrite(drucker);                            { Drucker öffnen }
   ofset:=0;   ypos:=1;                  { Erstes Buch, erste Zeile }
   While ofset<FileSize(buecher) Do Begin     { Die ganze Datei}
      Seek(buecher,ofset);
      Read(buecher,buch1);
      Write(drucker,buch1.verfasser);
      xpos:=Length(buch1.verfasser);
      While xpos<32 Do Begin              { Punkte bis Spalte 32 }
```

```
          Write(drucker,'.');
          Inc(xpos);
        End;                                    { While XPos<32 ... }
        WriteLn(drucker,buch1.titel);
        Inc(ofset); Inc(ypos);     { Nächstes Buch, nächste Zeile }
        Ifypos>seitenLaenge Then Begin               { Seite voll ? }
          For i:=1 To 6 Do WriteLn(drucker);
          ypos:=1;                { Erste Zeile auf nächste Seite }
        End;                                   { If YPos>=Seiten... }
End;                                              { While Ofset ... }
Close(drucker);                                 { Drucker schließen }
End;                                            { Ende der Prozedur }
```

Mit dieser Prozedur wird eine Liste mit sämtlichen Büchern der Datei auf dem Drucker ausgegeben. Von jedem Buch werden Verfasser und Titel geschrieben. Der Name des Verfassers beginnt in Spalte l, der Titel in Spalte 32. Der Zwischenraum zwischen Verfasser und Titel wird mit Punkten aufgefüllt. Auf jede Seite werden 66 Titel geschrieben. Danach folgen 6 Leerzeilen als Perforationssprung. In der Standardunit *Printer* wird eine Datei mit dem Namen "Lst", deklariert, mit LPT1 verbunden und geöffnet. Damit können Sie unmittelbar etwas auf dem Drucker ausgeben. Der Drucker muß dann an die erste parallele Schnittstelle angeschlossen sein. Unit *Printer* wird in die Uses-Deklaration aufgenommen. Als Dateiname bei *Write* und *WriteLn* muß "Lst" benutzt werden. Allerdings ist die Deklaration einer eigenen Filevariablen kein großer Aufwand und hat den Vorteil, daß man jede Schnittstelle benutzen kann.

5 Verbesserungen am Programm

Das Programm ist jetzt fertig. Es erfüllt die gestellten Anforderungen. Man kann Daten über Bücher speichern, sortieren, zeigen und ausdrucken lassen. Wenn Sie so weit gekommen sind, sollten Sie allerdings das Programm einer kritischen Prüfung unterziehen. Das kann sehr schwierig sein. Wenn man gerade ein Programm fertiggestellt hat und sich darüber freut, daß es gelungen ist, ist es nicht leicht, kritisch zu sein.

In diesem Kapitel sollen einige Schwachstellen des Programms gezeigt und Verbesserungsvorschläge gemacht werden.

5.1 Warnungen

Wenn der Benutzer versucht, Daten zu zeigen, ohne vorher eine Datei geöffnet zu haben, passiert nichts. Es entsteht kein Fehler. Für den Benutzer ist aber nicht erkennbar, warum keine Daten gezeigt werden. Das Programm müßte dem Benutzer mitteilen, warum keine Daten gezeigt werden können.

Ein ähnliches Problem gibt es an vielen anderen Stellen des Programms. Gebraucht wird also eine Prozedur, mit der eine Mitteilung an den Benutzer gegeben werden kann.

Diese Prozedur wird der Funktion *akzept* sehr ähnlich sein. Allerdings soll kein Rückgabewert geliefert werden und es ist egal, welche Taste der Benutzer als Bestätigung betätigt.

Deshalb wird hier nicht die Funktion *ReadKey,* sondern die Funktion *KeyPressed* benutzt. Die Syntax lautet:

```
ergebnis<boolean>:=KeyPressed;
```

Ist eine Taste gedrückt, wenn *KeyPressed* gerufen wird, liefert die Funktion den Wert *True.* Sonst wird der Wert *False* geliefert. Und hier ist jetzt die Prozedur Warnung:

```
Procedure warnung(zeile : String);   { Name der Prozedur }
Begin
   fenster;
   GotoXY(30-Length(zeile) DIV 2,2);
   Write(zeile);                             { Text schreiben }
   Repeat Until KeyPressed;                           { Pause }
   ClrScr;    Window(7,1,53,20);                  { Aufräumen }
End;                                               { Prozedur }
```

Die Prozedur kann an vielen verschiedenen Stellen in das Programm eingesetzt werden. Hier als Beispiel die Prozedur *naechste* in abgeänderter Form:

```
Procedure naechste;                          { Name der Prozedur }
Begin
  If Not offen Then warnung('Datei ist nicht geöffnet')
  Else
    If FileSize(buecher)=0 Then warnung('Datei enthält keine Daten')
    Else Begin
      Inc(ofset);                              { Nächstes Buch }
      If ofset<0 Then Ofset:=0;
      If ofset>=FileSize(buecher) Then Begin
        warnung('Datei enthält keine weiteren Daten');
        Exit;
      End;
      Seek(buecher,ofset);                  { Cursor plazieren }
      Read(buecher,buch1);                      { Daten lesen }
      ausgabe;                                 { Daten zeigen }
    End                                    { If FileSize(b... }
End;                                        { If offen and ... }
End;                                              { Prozedur }
```

Die Prozedur wird hiermit leider unübersichtlicher. Andererseits steigt die Benutzerfreundlichkeit des Programms. Das Programm wird damit deutlich professioneller.

5.2 Ein richtiger Editor

Das nächste Problem, das hier aufgegriffen werden soll, liegt in der Eingabeprozedur. In dem Turbo-Pascal-Editor können Sie den Cursor überall hin- und herbewegen und überall Zeichen einfügen oder löschen. Verglichen damit ist die Eingabeprozedur nicht besonders komfortabel. Sie erfüllt ihren Zweck, es könnte aber besser sein.

Das Problem liegt in der benutzten Standardprozedur *ReadLn.* Diese Prozedur empfängt die Zeichen von der Tastatur, verbindet sie mit der Variablen und zeigt sie auf dem Bildschirm. Mit einer solchen Prozedur ist es einfach, Programme zu schreiben. Man muß sich aber den Regeln der Prozedur unterordnen.

Wollen Sie dagegen eine größere Flexibilität erreichen, wird das Programm etwas komplizierter. Wenn Sie statt *ReadLn* die Funktion *ReadKey* benutzen, können Sie individuell auf jede Taste reagieren. Sie können dann auch Cursor- oder Funktionstasten eine Funktion geben.

ReadKey liefert aber nur den Wert der gedrückten Taste. Der Programmierer muß dann selber dafür sorgen, daß die einzelnen Werte zu Strings verbunden und auf dem Bildschirm gezeigt werden.

Wenn der Cursor überall frei herumbewegt werden kann, ist es natürlich sehr wichtig zu beachten, daß er innerhalb des Schemas bleibt. Deshalb sollten Sie als erstes die Rahmen des Schemas als Konstante definieren. So ist es auch später relativ einfach, das Schema zu erweitern.

Innerhalb der Prozedur werden die Daten in einem lokalen Array gesammelt. Erst wenn alle Daten eingegeben sind, werden sie in den globalen Record *buch1* übertragen. Dadurch müssen Sie sich nicht darum kümmern, in welchen String das aktuelle Zeichen eingefügt werden soll. Heißt das Array "zeilen", wird nämlich immer in den String *zeilen[ypos]* geschrieben, wobei *ypos* ein Ausdruck für die aktuelle Zeile ist.

Die Prozedur läuft in einer großen Repeat-Until-Schleife. Innerhalb der Schleife wird der aktuelle String erst an der Cursorposition geteilt. Das ist notwendig, weil es auch möglich sein soll, Zeichen in der Mitte des Strings einzufügen. Dann wird *ReadKey* gerufen. In einem Case-Befehl wird auf der jeweiligen Taste reagiert.

Auf der Tastatur gibt es normale Tasten und Sondertasten. Hat der Benutzer eine normale Taste betätigt, liefert *ReadKey* ein ASCII-Zeichen. Das sind alle Tasten mit Buchstaben, Zahlen oder anderen Zeichen, aber auch Druckersteuerzeichen wie Backspace (ASCII 8), Return (ASCII 13), Escape (ASCII 27) und Space (ASCII 32).

Hat der Benutzer dagegen eine Sondertaste betätigt, liefert *ReadKey* den Wert 0. Wird *ReadKey* daraufhin sofort wieder gerufen, liefert die Funktion den sogenannten "erweiterten Tastaturcode". Die Sondertasten sind die Funktionstasten, Cursortasten etc. In Anhang E gibt es eine Übersicht über die erweiterten Tastaturcodes.

Und hier ist jetzt die neue Eingabeprozedur:

```
Procedure eingabe;                                   { Name der Prozedur }
Const zeile_min = 4;                                       { Erste Zeile }
      zeile_max = 16;                                     { Letzte Zeile }
      spalte_min = 14;                                    { Linke Spalte }
      spalte_max = 44;                                   { Rechte Spalte }
Var zeilen          : Array[zeile_min..zeile_max]Of
                            String[spalte_max-spalte_min];
   links,rechts   : String[spalte_max-spalte_min];
   xpos,ypos,i    : Byte;
   taste          : Char;
   ende           : Boolean;
Begin
   ende:=False;
   xpos:=spalte_min;                            { Cursor in erste Spalte }
   ypos:=zeile_min;                              { Cursor in erste Zeile }
   For i:=zeile_min To zeile_max Do zeilen[i]:='';
   Repeat
      GotoXY(xpos,ypos);                              { Cursor plazieren }
      links:=Copy(zeilen[ypos],1,xpos-spalte_min);
      rechts:=Copy(zeilen[ypos],xpos-spalte_min 1,spalte_max-xpos);
      taste:=ReadKey;                                 { Taste ermitteln }
      Case taste Of
       #0 : Case ReadKey Of                               { Sondertaste }
          #71 : xpos:=spalte_min;                                { Home }
          #72 : If ypos>zeile_min Then Begin               { Pfeil hoch }
            Dec(ypos);
            xpos:=spalte_min+Length(zeilen[ypos]);
          End;                                                    { #72 }
          #73 : ypos:=zeile_min;                              { Page up }
          #75 : If xpos>spalte_min Then Dec(xpos);       { Pfeil links
          #77 : If xpos<spalte_max Then Inc(xpos);      { Pfeil rechts
          #79  : xpos:=spalte_min+Length(zeilen[ypos]);        { End }
          #80 : If YPos<ZeileMax Then Begin               { Pfeil unten }
            Inc(ypos);
            xpos:=spalte_min+Length(zeilen[ypos]);
          End;                                                    { #80 }
          #81 : ypos:=zeile_max;                            { Page down }
          #83 : Begin                                          { Delete }
            rechts:=Copy(rechts,2,Length(rechts)-1);
            zeilen[ypos]:=links+rechts;
      End;                                                        { #83 }
     End;                                        { #0 : Case ReadK ... }
     #8 : Begin                                         { Backspace }
      If xpos>spalte_min Then Dec(xpos);
```

```
      links:=Copy(links,1,Length(links)-1);
      zeilen[ypos]:=links+rechts;
    End;                                                          { #8 }
     #13 : Begin                                             { Return }
       xpos:=spalte_min;
       If  ypos<spalte_max  Then  Inc(ypos);
     End;                                             { #13 }
     #27  : If akzept('Sind alle Daten korrekt')          { Escape }
              Then  ende:=True;
    Else                                   { Buchstaben, Zahlen etc. }
     If  xpos<spalte_max  Then  Inc(xpos);
     zeilen[ypos]:=links+taste+rechts;
    End;                                            { Case Taste Of }
    GotoXY(spalte_min,ypos);
    Write(zeilen[ypos],#250);                     { Zeile schreiben }
  Until  Ende;
  With buch1 Do Begin  { Daten von Array in Record übertragen }
     verfasser:=zeilen[4];
     titel:=zeilen[5];
     verlag:=zeilen[6];
     isbn:=zeilen[7];
     erschienen:=zeilen[8];
     sprache:=zeilen[9];
     seiten:=zeilen[10];
     For i:=1 To 5 Do Begin
       schlagwort[i]:=zeilen[i+10];
     End;                                     { For i:=1 To 5 ... }
  End;                                               { With buch1 Do }
End;                                                       { Prozedur }
```

In Verbindung mit dieser Prozedur müssen einige Bemerkungen über Konstanten gemacht werden. Normalerweise werden Konstanten mit einem Typ deklariert. Eine Konstante mit angegebenem Typ wird von Turbo Pascal intern als Variable mit festem Wert angesehen. Konstanten, die mit Typangabe deklariert sind, dürfen deshalb nicht bei der Deklaration von Variablen oder weiteren Konstanten benutzt werden. Ein Array darf z.B. nicht mit Variable *Index* deklariert werden. Der Compiler würde dann bei der Variablendeklaration mit einer Fehlermeldung abbrechen.

Wird eine Konstante dagegen ohne Typ deklariert, ersetzt der Compiler schon bei der Compilierung überall im Programm den Namen durch den angegebenen Wert. Eine solche Konstante ist also lediglich eine andere Schreibweise im Quellcode. Deshalb gibt es hier bei der Variablendeklaration kein Problem.

Innerhalb des Case-Befehls wird das Doppelkreuz (#) benutzt. Dieses Zeichen veranlaßt den Compiler, die folgende Zahl als ASCII-Wert zu lesen. Damit hat das Zeichen fast dieselbe Wirkung wie die Funktion *Chr.* Die Betonung liegt aber auf "fast". Ein Funktionsergebnis ist prinzipiell eine Variable. Das Doppelkreuz von einer Zahl gefolgt ist dagegen eine Konstante.

Arbeiten Sie mit Version 5.x, hat das hier keine Bedeutung. Hier ist es durchaus erlaubt, "Chr(71)" statt "#71" zu schreiben. In Version 4.0 ist es aber nicht erlaubt, Variablen als Verzweigungen innerhalb eines Case-Befehls zu benutzen. Hier ist es notwendig, das Doppelkreuz zu benutzen.

Bei dieser Prozedur zeigt sich übrigens ein großer Vorteil der strukturierten Programmierung: Sie können die Prozedur in das Programm einfügen, ohne andere Teile des Programms zu ändern. Sie müssen nur darauf achten, daß die Prozedur unter demselben Namen wie die alte Eingabeprozedur abgespeichert wird. Wenn Sie das Programm dann neu compilieren, wird die neue Prozedur anstelle der alten in das Programm eingebunden, und das Programm ist sofort wieder lauffähig.

Wenn Sie jetzt Daten eingeben, kann der Cursor überall in dem Schema hin- und herbewegt werden. Wenn alle Daten eingegeben sind, wird die Funktion mit der Taste **Escape** beendet.

5.3 Ein neues Menü

In einem Menü in Turbo Pascal können Sie eine Funktion immer mit dem ersten Buchstaben des Wortes auswählen. Sie können aber auch das hervorgehobene Feld auf das Wort setzen und **Return** drücken. Schließlich können viele Funktionen mit den Funktionstasten ausgewählt werden. Etwas Ähnliches können Sie auch in eigenen Programmen machen. Dazu müssen Sie wissen, wie ein Text invertiert auf dem Bildschirm gezeigt werden kann.

In der Unit *Crt* gibt es eine globale Variable mit dem Namen *TextAttr.* Die Darstellungsweise der Buchstaben auf dem Bildschirm wird von dem Wert dieser Variablen entschieden.

TextAttr ist vom Typ *Byte.* Die Variable besteht also aus acht Bits. Die Bedeutung der einzelnen Bits ist in dem folgenden Schema dargestellt:

0 1 2	3	4 5 6	7
F F F	V	H H H	B

Mit den Bits 0 bis 2 wird die Vordergrundfarbe, mit den Bits 4 bis 6 die Hintergrundfarbe festgelegt. Ist Bit 3 gesetzt, wird der Text heller dargestellt. Dieses Bit wird von der Prozedur HighVideo gesetzt und von LowVideo gelöscht. Wenn Bit 7 gesetzt ist, wird der Text blinken.

Vorder- und Hintergrundfarbe werden jeweils mit 3 Bits bestimmt. Damit können insgesamt 8 verschiedene Farben dargestellt werden. Die möglichen Farben sind:

0 Schwarz
1 Blau
2 Grün
3 Türkis
4 Rot
5 Violett
6 Braun
7 Hellgrau

Mit den Prozeduren *TextBackground* und *TextColor* können Sie die Farben des Hinter- bzw. Vordergrunds ändern. Die Syntaxen der beiden Prozeduren lauten:

```
TextBackground(farbe <Byte>);
TextColor(farbe <Byte>);
```

Bei *TextBackground ist färbe* eine Zahl zwischen 0 und 7, entsprechend obenstehender Liste. Bei *TextColor* können Sie gleichzeitig das Videobit ändern. Deshalb *ist färbe* hier eine Zahl zwischen 0 und 15, wobei die Zahlen 0 bis 7 die 8 Farben in Kombination mit LowVideo bezeichnen und 8 bis 15 dieselben Farben mit High Video, also etwas heller.

Mit den vier Prozeduren *TextColor, TextBackground, LowVideo und HighVideo* können Sie den Wert der Variable *TextAttr* ändern. Sie können aber direkt der Variablen einen Wert zuweisen. Wie der Text bei verschiedenen Werten von *TextAttr* dargestellt wird, zeigt das folgende kleine Programm.

```
Program farb_test;                        { Name des Programms }
Uses Crt;                          { wegen GotoXY und TextAttr }
Var i,j : Byte;
Begin
   ClrScr;                                { Bildschirm löschen }
   For i:=0 To 15 Do
     For j:=0 To 15 Do Begin
       GotoXY(j*4+1,i+1);                  { Cursor plazieren }
       TextAttr:=i*16+j;           { TextAttr gleich 0 bis 255 }
       Write(TextAttr);                      { Zahl schreiben }
     End;                                 { For j:=0 To 15 ... }
End;                                                { Prozedur }
```

Mit einem Monochrom-System ist es natürlich nicht möglich, irgendwelche Farben darzustellen. Das obenstehende Programm zeigt aber, daß *TextAttr* auch hier nicht ohne Bedeutung ist.

Ist sowohl Vorder- als auch Hintergrundfarbe schwarz, wird kein Text dargestellt. Das ist wohl nicht überraschend; es gilt aber nur bei dieser Farbe. Ist die Vordergrundfarbe blau, wird der Text unterstrichen. Ist die Vordergrundfarbe schwarz und die Hintergrundfarbe hellgrau, wird der Text invertiert. Das ist z.B. der Fall, wenn *TextAttr* den Wert 112 hat.Damit können Sie ein Turbo Pascal-ähnliches Menü machen. Sie müssen nur den aktuellen Menüpunkt mit *TextAttr 112* schreiben und alle anderen mit den umgekehrten Farben, also *TextAttr 7.*

Das bedeutet, daß die Texte des Menüs immer wieder neu geschrieben werden müssen. Daher reicht es nicht aus, die Texte auf dem Bildschirm zu schreiben, sie müssen in einem Array gespeichert werden.

Es wäre hier ebenfalls möglich, einen Buchstaben des Wortes heller zu schreiben. Es würde aber das Programm etwas komplizierter und auch langsamer machen. Deshalb sollten hier stattdessen die Funktionstasten benutzt werden.

Zuerst muß die Prozedur *menue* neu programmiert werden. Hierbei sind die verschiedenen Menütexte in ein Array zu speichern und auf dem Bildschirm auszugeben. Die Prozedur sieht jetzt so aus:

```
Procedure menue;
Var i : Integer;
Begin
   menue_text[1]:=' Laden      F1 ';
   menue_text[2]:=' Speichern  F2 ';
   menue_text[3]:=' Eingeben   F3 ';
   menue_text[4]:=' Erste      F4 ';
   menue_text[5]:=' Nächste    F5 ';
   menue_text[6]:=' Suchen     F6 ';
   menue_text[7]:=' Ausdrucken F7 ';
   menue_text[8]:=' Quit       F8 ';
   Window(60,1,79,12);
   WriteLn(' ┌───────────────┐');
   For i:=1 To 8 Do WriteLn('│',menue_text[i],'│ ')
   WriteLn(' └───────────────┘');
End;
```

Danach muß das Hauptprogramm geändert werden. Innerhalb der großen Schleife muß jetzt auch der aktuelle Menüpunkt immer wieder neu geschrieben werden. Außerdem gilt es, mit dem Case-Befehl auch auf die Cursor- und Funktionstasten zu reagieren. Wie das geht, wissen Sie aber schon von der Eingabeprozedur.

```
Program katalog;                                      { Programmname }
Uses Crt;                                    { wegen Window und GotoXY }
Type buch = Record
           verfasser,titel,verlag,erschienen  :  String[30];
           isbn,sprache:  String[15];
           seiten  :  String[5];
           schlagwort  :  Array[1..5]of  String[30];
         End;
Var taste : Char;
   stop,offen  :  Boolean;
   buch1  :  Buch;
   buecher  :  File of Buch;
   ofset  :  Longint;
   menue_text  :  Array[1..8]Of  String[15];
   menue_punkt  :  Integer;
{$I  Dialog.pas}
{$I  Schema.pas}
{$I  Sortiere.pas}
{$I  Eingabe.pas}
{$I  Ausgabe.pas}
{$I  Menue.pas}
{$I  Oeffnen.Pas}
{$I  Suchen.Pas}
{$I  Drucker.pas}
Begin                                    { Anfang des Hauptprogramms }
   schema;                                  { Schema wird gezeichnet }
   menue;                                     { Menü wird gezeichnet }
   stop:=False;     offen:=False;    menue_punkt:=1;
   Repeat
      Window(60,1,79,12);
      TextAttr:=112;  GotoXY(2,menue_punkt+1);
      Write(menue_text[menue_punkt]);
      taste:=Readkey;                               { Auf Taste warten }
      TextAttr:=7;      GotoXY(2,menue_punkt+1);
      Write(menue_text[menue_punkt]);
      Window(7,1,53,20);
      Case taste Of
        #0 : Case ReadKey Of
          #59 : oeffnen;                                        { F1 }
          #60 : schliessen;                                     { F2 }
          #61 : eingeben;                                       { F3 }
          #62 : erste;                                          { F4 }
          #63 : naechste;                                       { F5 }
```

```
          #64 : suchen;                                              { F6 }
          #65 : drucken;                                             { F7 }
          #66 : Begin                                                { F8 }
            If akzept('Wirklich das Programm verlassen') Then
              stop:=True;
          #72 : If menue_punkt>1 Then Dec(menue_punkt);{Pfeil oben }
          #73 : menue_punkt:=1                                  { Page up }
          #80 : If menue_punkt<8 Then Inc(menue_punkt);{Pfeil unten}
          #81 : menue_punkt:=8;                               { Page down }
        End;                                        {#0 : Case ReadKey ... }
        #13 : Case menue_punkt Of                                { Return }
          1 : oeffnen;
          2 : schliessen;
          3 : eingeben;
          4 : erste;
          5 : naechste;
          6 : suchen;
          7 : drucken;
          8 : If Akzept('Wirklich das Programm verlassen') Then
              stop:=True;
        End;                                  { #13 : Case menue_punkt ... }
      End;                                                { Case Taste of }
    Until Stop;
End.                                              { Ende des Hauptprogramms }
```

5.4 Verzeichnis wechseln

Mit der besseren Bedienung bekommt das Programm so langsam einen professionellen Ansatz. Bei der Handhabung der Dateien gibt es dagegen noch einiges zu tun. Als erstes soll die Möglichkeit geschaffen werden, das aktuelle Verzeichnis zu wechseln. Die Dateien mit unseren Buchdaten werden immer im aktuellen Verzeichnis gespeichert. Deshalb müßte es möglich sein, dieses Verzeichnis zu ändern. Dann könnte man z.B. die Programmdiskette in Laufwerk A haben und die Dateien auf einer Datendiskette in Laufwerk B speichern.
Für diese Prozedur brauchen Sie zwei neue Standardprozeduren:

```
GetDir(laufwerk <Integer>,var pfad <String>);
ChDir(pfad <String>);
```

Die Prozedur *GetDir* liefert in der Variablen *pfad* den aktuellen Pfad des Laufwerks. Welches Laufwerk untersucht werden soll, wird mit dem Parameter *laufwerk* festgelegt, der einen Wert zwischen 0 und 26 annehmen kann. Dabei steht 0 für das aktuelle Laufwerk und l bis 26 für die Laufwerke A bis Z.

Mit *ChDir* können das aktuelle Verzeichnis und Laufwerk festgelegt werden. Bei {$!-} kann nach beiden Prozeduren ein Ergebnis mit *lOResult* ausgelesen werden. Mit diesen beiden Standardprozeduren ist es relativ einfach, ein kleines Programm zum Laufwerkwechsel zu schreiben. Erst müssen Sie *GetDir* rufen, dann bekommt der Benutzer die Gelegenheit, den gefundenen Pfad zu ändern. Schließlich wird mit *ChDir* der geänderte Pfad zum aktuellen erklärt.

```
Procedure verzeichnis;                          { Name der Prozedur }
Var pfad    : String;
   zeichen : Char;
Begin
   GetDir(0,pfad);                      { aktuellen Pfad ermitteln }
   fenster;
   Repeat
     GotoXY(3,2);   Write(pfad);                    { Pfad zeigen }
     zeichen:=ReadKey;           { Zeichen von Tastatur holen }
     Case zeichen Of
       #8 : If Length(pfad)>0 Then Begin
         Write(#8,#32);              { letztes Zeichen löschen }
         pfad:=Copy(pfad,1,Length(pfad)-1);
       End;                                  { #8 : If Length ... }
       #33,#35..#41,#45..#58,#64..#90,#92,#95..#123,#125,#12(
           pfad:=pfad+Upcase(zeichen);
     End;                                     { Case Zeichen Of }
   Until zeichen=#13;                               { bis Return }
   {$I-}                              { IO-Kontrolle ausschalten }
   ChDir(pfad);                           { neuen Pfad festlegen }
   If IOResult<>0 Then warnung('Neuen Pfad nicht gefunden');
   {$I+}                        { IO-Kontrolle wieder einschalten }
   ClrScr;    Window(7,1,53,20);                      { aufräumen }
End;                                                   { Prozedur }
```

Zwischen *GetDir* und *ChDir* enthält die Prozedur einen etwas vereinfachten Editor. Es ist hier nicht möglich, Zeichen in der Mitte des Strings einzufügen. Das können Sie natürlich auch programmieren, aber der Aufwand wäre unverhältnismäßig hoch.

Beachtenswert ist, daß nur ganz bestimmte Zeichen angenommen werden. Unter MS-DOS sind in einem Pfad nur die folgenden Zeichen zugelassen:

die Buchstaben: a bis z und A bis Z
die Ziffern: 0 bis 9
die Sonderzeichen: _, $, &, !, %, #,', - @, {,}, ~, `, /, \

Deshalb werden nur diese Zeichen in der Prozedur überhaupt angenommen. Dadurch ist eine wesentliche Fehlerquelle gleich ausgeschaltet.

5.5 Dateiauswahl

Auch das Laden einer Datei ist in dem Programm etwas umständlich. Das Problem ist, daß man sich an die Namen der Dateien erinnern muß. Im Turbo-Pascal-Editor kann man dagegen alle vorhandenen Dateien in einem Fenster zeigen lassen und seine Auswahl dort treffen. Das kann auch programmiert werden.

Damit dieses Modul überall einsetzbar wird, wird es als Funktion geschrieben. Der Rückgabewert der Funktion ist dann der Name der ausgewählten Datei.

Wenn die Funktion *dateiwahl* heißt, muß in der Prozedur *oeffnen* nur der Befehl "ReadLn(dateiname);" durch "dateiname:=dateiwahl;" ersetzt werden.

Innerhalb der Funktion soll zuerst dem Benutzer ermöglicht werden, einen Dateinamen einzugeben. Dann wird dieser Name auf Wildcards (* und ?) untersucht. Enthält der Name solche Wildcards, werden alle passenden Dateien gefunden und in einem Fenster angezeigt. Danach soll der Benutzer unter diesen Namen den richtigen auswählen können. So vollzieht sich die Auswahl wie in dem Menü.

Das Einlesen des ersten Dateinamens geschieht mit *ReadLn.* Ob der Dateiname Wildcards enthält, kann mit *Pos* überprüft werden. Um die passenden Dateien zu finden, brauchen wir aber zwei neue Standardprozeduren: *FindFirst* und *FindNext.Die* beiden Prozeduren befinden sich im Unit *Dos.* Dieses Unit muß deshalb in der Uses-Deklaration des Hauptprogramms aufgenommen werden.
Die Syntaxen der beiden Prozeduren lauten:

```
FindFirst(pfad<String>,attr<Word>,Var r<SearchRec>) ;
FindNext(Var r <SearchRec>);
```

Mit *FindFirst* können Sie die erste Datei finden, die zu einem Auswahlkriterium paßt. Danach können Sie mit *FindNext* alle anderen Dateien finden. Die zwei Prozeduren werden in der Regel zusammen verwendet.

Der Variablen *pfad* wird der mit *ReadLn* eingegebene Dateiname zugewiesen. Geben Sie nur einen Dateinamen an, wird in dem aktuellen Verzeichnis auf dem aktuellen Laufwerk gesucht.

Mit *Attr* wird angegeben, welche Arten von Dateien gesucht werden sollen. Normale Dateien werden immer gesucht. Darüber hinaus gibt es aber 6 weitere Arten von Dateien. Für jede Dateiart, die gesucht werden soll, muß in die Variable *Attr* ein Bit gesetzt werden. In der folgenden Tabelle sehen Sie die möglichen Dateiarten und die jeweiligen Zahlen, die addiert werden müssen, um den richtigen Wert von *Attr* zu finden.

Normale Dateien	0
Read-only-Dateien	1
Versteckte Dateien	2
Systemdateien	4
Diskettennamen	8
Verzeichnisse	16
Archiv-Dateien	32

Wenn *Attr* den Wert 17 bekommt, wird z.B. nach normalen Dateien, read-only-Dateien und Verzeichnissen gesucht.

Wird eine passende Datei gefunden, werden die Daten dieser Datei in dem letzten Parameter r abgelegt, r ist eine Variable vom Typ *SearchRec.* Dieser Typ ist im Unit *Dos* definiert:

```
SearchRec = Record
Fill: Array[1..21]Of Byte;
Attr: Byte
Time: Longint;
Size: Longint;
Name: String[12];
End;
```

Fill wird nur intern gebraucht. *Attr* zeigt an, welcher Dateityp gefunden wurde. *Time* und *Size* enthalten Zeit und Größe der gefundenen Datei. Name enthält den Dateinamen.

Haben Sie in der Variablen *pfad* ein ungültiges Verzeichnis angegeben oder konnte zu dem Suchkriterium keine passende Datei gefunden werden, entsteht ein DOS-Fehler. Ein solcher Fehler führt nicht zu einem Programmabbruch. Die Fehlernummer wird aber in der globalen Variablen *DosError* abgelegt. *DosError* ist eine Variable vom *Typ Integer,* die im Unit *Dos* deklariert ist.

Wenn eine passende Datei gefunden wurde, bekommt *DosError* den Wert 0. Wurde ein ungültiger Pfad angegeben, bekommt die Variable den Wert 2. Wenn keine weiteren Dateien gefunden werden konnten, bekommt *DosError* den Wert 18.

Mit diesen Prozeduren können die passenden Dateien gefunden werden. Hierbei muß zuerst ein ausreichend großes Array für die Dateinamen deklariert werden.

Hier wird das Array mit 200 Elementen deklariert. Wer mehr als 200 Dateien in einem Verzeichnis gespeichert hat, muß entweder ein Auswahlkriterium benutzen oder noch seine Festplatte umorganisieren.

Nun wird die erste Datei mit *FindFirst* gefunden und im Array abgelegt. Danach werden alle weiteren Dateien mit *FindNext*gefunden. Das heißt, *FindNext-wird* so lange aufgerufen, bis *DosErwr* einen Wert ungleich 0 bekommt oder bis das Array voll ist, was unter normalen Umständen nicht passieren sollte.

Wenn alle Dateien gefunden sind, müssen sie dem Benutzer zur Auswahl gezeigt werden. In das kleine Fenster passen nur 4 Dateinamen. Deshalb muß es mit den Pfeiltasten möglich sein, durch das Array zu blättern, bis die richtige Datei gefunden ist. Hier die fertige Funktion:

```
Function  dateiwahl  :  String;
Var  dateiname     :  String[12];
    dateiinfo     :  SearchRec;
    i,ofs1,ofs2  :  Byte;
    taste        :  Char;
    namen         :  Array[0..200]Of  String[12];
Begin
    fenster;     GotoXY(3,2);     Write('Datei:  ');
    ReadLn(dateiname);                          { Dateinamen  eingeben }
    FindFirst(dateiname,0,dateiInfo);          {  erste  Datei  finden  }
If  DosError=0  Then  Begin
    If  (Pos('*',dateiname)>0)  Or  (Pos('?',dateiname)>0)  Then  Begin
        i:=0;
        While  (DosError=0)  And  (i<201)  Do  Begin
          namen[i]:=dateiinfo.Name;       {  Namen  in  Array  einfügen  }
          Inc(i);
          FindNext(dateiinfo);                  {  Nächste  Datei  suchen  }
        End;                                   {  While  (DosError=0)  ...  }
        While  i<201  Do  Begin         {  Übrige  Arrayelemente  löschen  }
          namen[i]:='';
          Inc(i);
        End;                                            {  While  i<201  ...  }
        ofs1:=0;     ofs2:=0;
        Repeat;
          fenster;
            For  i:=0  To  3  Do  Begin              {  4  Dateinamen  zeigen  }
              GotoXY(3+i*14,2);
              Write(namen[ofs1*4+i]);
          End;                                            {  For  i:=0  To  3  }
          TextAttr:=112;     GotoXY(3+ofs2*14,2);
```

```
        Write(namen[ofs1*4+ofs2]);
        taste:=ReadKey;
        TextAttr:=7;      GotoXY(3+ofs2*14,2);
        Write(namen[ofs1*4+ofs2]);
        Case taste Of
        #0 : Case ReadKey Of                        { Sondertaste }
          #72 : If ofs1<0 Then Dec(ofs1);           { Pfeil oben }
          #75 : If ofs2>0 Then Dec(ofs2);           { Pfeil links }
          #77 : If ofs2<3 Then Inc(ofs2);           { Pfeil rechts }
          #80 : If ofs1<50 Then Inc(ofs1);          { Pfeil unten }
        End;                                        { Case ReadKey Of }
      End;                                          { Case Taste Of }
      Until taste=#13;
      dateiname:=namen[ofs1*4+ofs2];
    End;                             { If (Pos('*',DateiName ... }
  End;                                   { If DosError=0 ... }
  ClrScr;   Window(7,1,53,20);                      { Aufräumen }
  DateiWahl:=dateiname;                             { Rückgabewert }
End;                                                { Funktion }
```

5.6 Weitere Möglichkeiten

Es steht uns nun offen, das Programm weiter zu vergrößern und verbessern. Tatsächlich werden die wenigsten Programme jemals "fertig", sondern die Programmierer schreiben immer neue, verbesserte Versionen, bis irgendwann das Programm als veraltet aufgegeben wird.

Trotzdem müssen wir irgendwo aufhören. Außerdem soll dieses Buch ja keine endgültigen Programme liefern, sondern zum eigenen Programmieren verführen. Folgende Anregungen mögen dazu dienen, wie das Programm sich weiterentwickeln könnte.

Es soll ja Menschen geben, die Bücher wegwerfen. Wer zu dieser Gruppe gehört, braucht die Möglichkeit, ein Buch aus der Datei zu löschen. Eine solche Prozedur muß mit einer Suchfunktion anfangen. Der Benutzer muß das Buch, das gelöscht werden soll, schnell finden können.

Dann muß das gefundene Buch bis ans Ende der Datei sortiert werden. Zum Schluß muß ein Element am Dateiende abgeschnitten werden.

Die Sortierfunktion ist sehr einfach, es muß nämlich nichts verglichen werden. Es reicht aus, alle die Bücher, die weiter hinten in der Datei stehen, um einen Platz nach vorne zu holen.

Innerhalb einer Schleife wird jedes Element mit *Read* gelesen, dann wird das Offset um eins vermindert und das Element wird wieder mit *Write* geschrieben.

Wenn diese Schleife am Dateiende angekommen ist, muß das letzte Element der Datei abgeschnitten werden. Das können Sie mit der Standardprozedur *Truncate* erledigen:

```
Truncate(datei <File>);
```

Die Prozedur schneidet die Datei an der aktuellen Cursorposition ab. Um das letzte Element der Datei *buecher* abzuschneiden, brauchen Sie also die folgenden zwei Befehle:

```
Seek (buecher,Size(buecher)-1) ;
Truncate(buecher) ;
```

Die Prozedur müssen Sie aber selber schreiben.

Der zweite Verbesserungsvorschlag betrifft die Suchfunktion. Hier muß der Benutzer das gesuchte Schlagwort eingeben. Man sollte aber mit der Vergabe von Schlagwörtern sehr vorsichtig sein. Wenn alle Computerbücher mit dem Schlagwort "EDV" katalogisiert sind, hilft es nicht, das Wort "Computer" einzugeben.

Deshalb wäre es sinnvoll, alle Schlagwörter in einer Datei getrennt zu sammeln. Dadurch kann man bei der Eingabe kontrollieren, welche Wörter bei ähnlichen Büchern benutzt wurden.

Wenn Sie eine solche Datei einrichten, könnten Sie auch die Suchbegriffe direkt aus der Datei auswählen. Die Auswahl müßte dann ähnlich der Dateiauswahl sein. Auch beim Ausdrucken könnten Sie einen Suchbegriff einbauen. Damit wäre es möglich, Listen zu bestimmten Themen auszudrucken.

In Turbo Pascal gibt es die Funktion "OS Shell", mit der DOS-Betriebssystem-Funktionen ausgeführt werden können, ohne das Programm zu verlassen. Mein letzter Vorschlag ist, eine solche Funktion in das eigene Programm einzubauen. Bei dieser Funktion wird erst der größtmögliche Speicherblock freigegeben. In diesem Bereich wird dann das Programm "Command.com" von der Systemdiskette gestartet. Das ist alles.

Wenn ein Turbo Pascal-Programm gestartet wird, werden Programmcode, Variablen und Stapel im unteren Teil des Speichers abgelegt. Der obere Teil des Speichers wird für den sogenannten "Heap" reserviert. Mit den Standardprozeduren *New* und *GetMem* können aus diesem Bereich Speicherblöcke für das Programm in Anspruch genommen werden.

Im Menü "Options/Compiler/Memory Sizes" können Sie festlegen, wieviel Speicher das fertige Programm benutzen darf. Mit dem ersten Untermenüpunkt kann

der Stabel bis auf 64 KByte vergrößert werden. Mit "Low heap limit" und "High heap limit" kann die kleinste bzw. größte akzeptable Heap-Größe festgelegt werden.

Bei "Low heap limit" steht als Standard 0. Damit wird das Programm auch lauffähig, wenn kein Speicher als Heap zur Verfügung steht. Der Standard bei "High heap limit" ist 655360 (640 KByte). Dadurch werden bis zu 640 KByte oder mit anderen Worten der gesamte freie Speicher als Heap reserviert.

Dadurch gibt es aber keinen freien Speicher, in dem andere Programme gestartet werden können. Um einen freien Speicherblock zu schaffen, muß der Wert bei "High heap limit" deshalb deutlich heruntergesetzt werden, z.B. auf 50000.
Jetzt kann Command.com gestartet werden. Das geschieht mit der Standardprozedur *Exec:*

```
Exec(pfad,kommando_zeile <String>);
```

pfad ist der Name des Programms, das gestartet werden soll. *kommando_zeile* ist ein String mit Parameter, der an das Programm übergeben werden soll. In diesem Fall lautet der Befehl:

```
Exec ('\Command.com', ' ' ) ;
```

Bevor das Programm gestartet wird, sollten Sie den Bildschirm löschen. Sie sollten auch eine Anleitung für den Benutzer schreiben wie in Turbo Pascal. Wenn der Benutzer Command.com wieder mit *Exil* beendet, müssen Schema und Menü wieder aufgebaut werden, aber das dürfte ja jetzt kein Problem sein.

6 Eigene Units

Zusammen mit dem Turbo-Pascal-System wird eine Anzahl von Units geliefert. In den früheren Kapiteln ist schon beschrieben, wie diese Units in den Programmen eingesetzt werden können. Durch das Programmieren eigener Units ist das System erweiterbar. Es gibt zwei verschiedene Gründe, Units zu schreiben.

Das Hauptprogramm oder ein Unit wird im Speicher als ein *Programmsegment* abgelegt. Ein solches *Segment* darf aber niemals größer als 64 KByte sein. Sehr große Programme müssen deshalb immer in Units aufgeteilt werden.

Sie können auch Units schreiben, um damit eine Bibliothek von Programm-Modulen aufzubauen. In den beiden letzten Kapiteln gab es z.B. die Funktion *Akzept* und die Prozedur *Warnung.* Diese beiden Module könnten in ganz anderen Programmen Verwendung finden. Wenn Sie sie häufig benutzen wollen, ist es sinnvoll, sie in einem Unit unterzubringen. Dann können die Module unmittelbar von einem neuen Programm gerufen werden. Deshalb soll hier gezeigt werden, wie Sie aus diesen zwei Modulen ein Unit machen.

Bevor Sie das Unit schreiben, sollten Sie gründlich überprüfen, ob die Module tatsächlich ohne weiteres in andere Programme eingefügt werden können. Dabei stellen sich uns zwei Probleme. Es geht um die letzte Zeile der Module:

```
ClrScr;          Window(7,1,53,20);        { Aufräumen }
```

Erst wird das kleine Fenster gelöscht, dann wird ein Fenster um das Schema definiert. In unserem Buchverwaltungs-Programm ist dies korrekt. In einem anderen Programm wäre es aber wahrscheinlich falsch.

Es könnte sein, daß das andere Programm an dieser Stelle des Bildschirms etwas zeigt, was nicht gelöscht werden darf. Es ist auch unwahrscheinlich, daß der Window-Befehl in einem anderen Programm genauso aussehen soll. Wenn das Modul gerufen wird, gibt es auf dem Bildschirm einen bestimmten Stand. Dieser muß am Ende des Moduls wiederhergestellt werden. In dem Buchverwaltungs-Programm ist dies einfach, weil der Stand bekannt ist. Soll das Modul dagegen in jedem beliebigen Programm eingesetzt werden können, muß der Stand des Bildschirms erst registriert und gespeichert werden.

Am einfachsten zu speichern ist die Fenstergröße. Im Unit Crt gibt es zwei globale Variablen vom Typ *Word: WindMin* und *WindMax.* Wenn der Befehl *Window* gerufen wird, werden die obere linke und die untere rechte Ecke des Fensters in diesen beiden Variablen gespeichert. Der Inhalt dieser Variablen wird aber von der Prozedur *fenster* geändert. Am Anfang der beiden Module müssen die Werte deshalb gelesen und gesichert werden. Dann kann das Fenster am Ende des jeweiligen Moduls wiederhergestellt werden.

Dazu müssen erst zwei neue lokale Variablen des selben Typs deklariert werden:

```
Var  w_min,w_max  :  Word;
```

In diesen Variablen können dann die Begrenzungen des Fensters gesichert werden:

```
w_min:=WindMin;       w_max:=WindMax;
```

WindMin und *WindMax* sind Variable vom Typ *Word;* sie enthalten also jeweils zwei Bytes. Im niedrigen Byte wird die X-Koordinate gespeichert, im höheren die Y-Koordinate. Die Bytes können mit den Standardfunktionen *Hi* und *Lo* gelesen werden. Die Syntaxen dieser Funktionen sind:

```
a <Byte>:=Hi(b <Integer|Word>);
a <Byte>:=Lo(b <Integer|Word>);
```

Die Funktion Hi liefert als Ergebnis das höhere Byte aus dem Parameter b, *Lo* entsprechend das niedere Byte.

Die Werte, die in *WindMin* und *WindMax* gespeichert werden, haben allerdings einen anderen Wertebereich als die Parameter für *Window* und *GotoXY.* Bei einem Bildschirm mit 25 Zeilen zu je 80 Zeichen geht der Wertebereich für *WindMin* und *WindMax* von (0,0) bis (79,24). Die Parameter für *Window* reichen dagegen von (1,1) bis (80,25). Um das frühere Fenster am Ende des Moduls wiederherzustellen, muß deshalb der folgende Befehl eingefügt werden:

```
Window (Lo (w_min) +l,Hi (w_min) +1, Lo (w_max) +l,Hi (w__max) +1) ;
```

Schwieriger ist es, den Bildschirminhalt wiederherzustellen. Turbo Pascal hat keine Standardprozedur, mit der ein Teil des Bildschirms gelesen werden könnte. Der Inhalt des Bildschirms wird in einem bestimmten Speicherbereich, dem sogenannten Videospeicher, abgelegt. Für jedes Zeichen auf dem Bildschirm werden 2 Bytes benötigt. Das erste enthält das Zeichen als ASCII-Wert, das zweite die Darstellungsweise, d.h. den Wert, den die Variable *TextAttr* hatte, als das Zeichen geschrieben wurde.

Die Bytes der einzelnen Zeichen folgen unmittelbar hintereinander. Die Zeichen sind zeilenweise abgespeichert von oben bis unten und innerhalb der Zeilen von links nach rechts.

Das kleine Fenster umfaßt drei Zeilen auf dem Bildschirm. Für diese drei Zeilen enthält der Videospeicher insgesamt 480 Bytes. Man müßte also eine Kopie dieser Bytes machen und sie dann später zurückkopieren.

Damit dies möglich ist, müssen Sie allerdings erst wissen, wo der Videospeicher sich befindet. Hier stellt sich das nächste Problem. Die Adresse des Videospeichers ist nämlich bei einem Färb- und einem Monochromsystem unterschiedlich.

Unter MS-DOS besteht eine Speicheradresse aus einer Segmentadresse und einem Offset. Die zwei Teile der Adresse werden normalerweise als vierstellige Hexadezimalzahlen durch einen Doppelpunkt getrennt dargestellt. Auf einem Farbsystem beginnt der Videospeicher an der Adresse $B800:$0000, auf einem Monochromsystem an der Adresse $BQOO:$0000. Das Dollarzeichen ($) veranlaßt den Compiler, die darauffolgende Zahl als Hexadezimalzahl zu lesen.

Wenn das Programm auf jedem Computer lauf fähig sein soll, muß es also erst untersuchen, wie der Computer ausgerüstet ist. Das ist leicht mit einem BIOS-Interrupt zu überprüfen.

BIOS steht für "Basic Input Output System". Es ist ein Teil von MS-DOS und enthält einige Prozeduren, die sogenannten *BlOS-Interrupts.* Sie sind nicht durch einen Namen, sondern mit einer Nummer gekennzeichnet. Vergleichbar den Prozeduren in Turbo Pascal sind BlOS-Interrupts teils mit Parametern aufzurufen. Diese werden im Register des Zentralprozessors abgelegt, von wo man auch etwaige Ergebnisse des Interrupts wieder ausliest.

Der Aufruf eines Interrupts ist von Turbo Pascal aus sehr einfach. Sie müssen nämlich in den Registern weder lesen noch schreiben, das erledigt die Standardprozedur *Intr.* Die Syntax lautet:

```
Intr(intNr <Byte>; Var regs <Registers>);
```

IntNr ist die Nummer des Interrupts, der ausgeführt werden soll, *regs* ist eine Variable vom Typ *Registers,* dieser Record ist in Unit *Dos* so definiert:

```
Registers  =  Record
                   Case  Integer  Of
                    0:   (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags  :  Word);
                    1:   (AL,AH,BL,BH,CL,CH,DL,DH  :  Byte);
                  End.
```

Die Prozedur *Intr* kopiert den Inhalt des Records in die Register des Zentralprozessors, ruft den Interrupt und kopiert den Inhalt der Register zurück in den Record. Die Namen der Elemente in dem Record entsprechen den Namen der jeweiligen Register. Man muß also nur die Parameter in den Record schreiben, die Prozedur *Intr* aufrufen und die Ergebnisse aus dem Record lesen.

Die Beschreibung der einzelnen Interrupts und deren Parameter und Ergebnisse würde hier zu weit führen. Man findet sie aber in vielen Büchern über MS-DOS oder Assemblerprogrammierung.

Mit dem Interrupt *nr 17* können Sie herausfinden, wie der Computer ausgerüstet ist. Dieser Interrupt wird ohne Parameter gerufen. Trotzdem muß eine Variable vom Typ Registers deklariert werden, um das Ergebnis aufnehmen zu können.

Das Ergebnis ist vom Typ *Word* und wird in dem Register AX geliefert. Die einzelnen Bits in der Variablen Word geben Aufschluß über die Hardwarekonfiguration des Rechners. In Bit 14 und 15 wird z.B. die Anzahl der parallelen Schnittstellen angegeben. Ist Bit 12 gesetzt, hat der Rechner ein Joystickanschluß. Bit 6 und 7 enthalten die Anzahl der angeschlossenen Diskettenlaufwerke.

In unserem Fall sind nur Bit 4 und 5 wichtig. Wenn diese beiden Bits gesetzt sind, d.h. sie haben jeweils den Wert l, hat der Computer eine Monochromkarte. Ist dagegen nur eins der beiden Bits gesetzt, handelt es sich um ein Farbsystem.

Erst müssen Sie also eine Variable vom Typ *Registers* deklarieren. Außerdem brauchen Sie eine Variable vom Typ *Word,* in die die Segmentadresse des Videospeichers geschrieben werden kann:

```
Var regs : Registers;
bildschirm : Word;
```

In das Programm müssen dann nur noch die folgenden drei Zeilen eingefügt werden:

```
Intr(17,regs) ;
If (regs.AX And 48)=48 Then bildschirm:=$BOOO
Eise bildschirm:=$B800;
```

In der Zahl 48 sind nur das vierte und fünfte Bit gesetzt. Wenn die *And*-Verknüpfung dieser Zahl mit *Regs.AX als* Ergebnis 48 liefert, müssen Bit 4 und 5 in *Regs.AX auch* gesetzt sein. Das kann aber nur geschehen, wenn der Computer eine Monochromkarte hat.

Die Adresse des Videospeichers kann jetzt als (Bildschirm:0000) angegeben werden. Das ist gleichzeitig die Adresse der ersten Zeile auf dem Bildschirm. Jede dieser Zeilen ist mit 160 Bytes vertreten. Das linke Zeichen einer Zeile entspricht somit der Adresse (Bildschirm:160*(Zeile-l). Das kleine Fenster befindet sich in den Zeilen 18 bis 20. Um diesen Bereich zu sichern, müssen Sie deshalb ab der Adresse (Bildschirm:160*17) 480 Bytes kopieren.

Es gibt keine Prozedur, mit der Bytes aus dem Speicher kopiert werden können. Turbo Pascal definiert aber drei ganz besondere Arrays: *Mem, MemWund MemL.* Die drei Arrays sind als *Array Of Byte, Array Of Word* bzw. *Array Of Longint* definiert. Sie umfassen den gesamten Hauptspeicher des Rechners und erwarten

als Index eine Adresse. Damit können Speicherinhalte mit einer einfachen Zuweisung kopiert werden.

Es liegt jetzt nahe, zwei neue Prozeduren in das Unit aufzunehmen. Mit der einen soll eine Zeile von dem Bildschirm in eine Variable kopiert, mit der anderen soll sie zurückkopiert werden.

Die Variable, die die Zeile aufnehmen soll, muß 160 Bytes umfassen. Sie sollte als neuer Typ definiert werden:

```
Type zeile_speicher = Array[0..159]Of Byte
```

Die beiden Prozeduren müssen dann in einer Schleife einfach 160 Bytes aus dem Array *Mem* in eine Variable vom Typ *zeüe_speicher* kopieren bzw. zurückkopieren.

Das neue Unit soll die Funktion *akzept* und die 4 Prozeduren *warnung, fenster, retten* und *wiederherstellen* enthalten. Ein Unit wird weitgehend wie ein Programm aufgebaut. Es zerfällt aber in zwei Teile: *Interface* und *Implementation.* Alles, was im Interface-Teil steht, ist überall im Programm bekannt. Der Inhalt des Implementation-Teils ist dagegen nur innerhalb des Units bekannt. Auf den Turbo-Pascal-Disketten gibt es einige Dateien mit der Endung doc. Diese Dateien enthalten den Interface-Teil der Standardunits.

Hier ist als Beispiel das Unit mit den 5 Prozeduren:

```
Unit meinunit;                                   { Name des Units }

Interface                              { Anfang des Interface-Teils }
Uses Crt,Dos;                      { Uses-Deklaration für das Unit }
Type  frage_string   =  String[50];
      zeile_speicher  = Array[0..159]Of Byte;
Function  akzept(frage  :  frage_string):Boolean;
Procedure  warnung(zeile  :  frage_string);
Procedure  retten(zeile:Integer;Var  speicher:zeile_speicher);
Procedure  wiederherstellen(zeile:Integer;speicher:zeile_speicher);

Implementation                    { Anfang des Implementation-Teils }
Var bildschirm  : Word;          { Segment-Adresse des Videospeichers }
    regs        : Registers;
    i           : Integer;
Procedure  retten(zeile:Integer;  Var  speicher:zeile_speicher);
Begin
    For  i:=0  To  159  Do  speicher[i]:=Mem[bildschirm:i+160*(zeile-1)];
End;
```

```
Procedure  wiederherstellen(zeile:Integer;speicher:zeile_speicher);
Begin
   For  i:=0  To  159  Do  Mem[bildschirm:i+160*(Zeile-1)]:=speicher[i];
End;

Procedure  fenster;
   Window(10,18,69,20);
   WriteLn('                                                    ');
   WriteLn('                                                    ');
   Write  ('                                                    ');
End;

Function  akzept(frage  :  frage_string):Boolean;
Var zeile          : String[56];
     speicher      : Array[1..3]Of  zeile_speicher;
     taste         : Char;
     w_min,w_max   : Word;
Begin
    zeile:=Concat(zeile,'  (J/N)');          {  (J/N)  an  Frage  anhängen  }
    For  i:=1  To  3  Do  retten(i+17,speicher[i]);
    w_min:=WindMin;   w_max:=WindMax;           { Fensterdaten  retten  }
    fenster;                                  { kleines Fenster zeichnen }
    GotoXY(30-Length(zeile)  DIV  2,2);            {  Cursor  plazieren  }
    Write(zeile);                                  {  Frage  schreiben  }
    Repeat                                     { auf Tastendruck warten }
        taste:=UpCase(ReadKey);
    Until  (taste='J')  or  (taste='N');
    For  i:=1  To  3  Do  wiederherstellen(i+17,speicher[i]);
    Window(Lo(w_min),Hi(w_min),Lo(w_max),Hi(w_max));      {  aufräumen  }
    If  taste='J'  Then  akzept:=True                 {  Rückgabewert  }
    Else  akzept:=False
End;                                                   {  Funktion  }

Procedure  warnung(zeile  :  frage_string);          {Name  der  Prozedur}
Var  speicher    :  Array[1..3]Of  zeile_speicher;
    w_min,w_max  :  Word;
Begin
    For  i:=1  To  3  Do  retten(i+17,speicher[i]);
    w_min:=WindMin;     w_max:=WindMax;          {  Fensterdaten  retten  }
    fenster;                                  {  kleines  Fenster  zeichnen  }
    GotoXY(30-Length(zeile)  DIV  2,2);            {  Cursor  plazieren  }
    Write(zeile);                                     {  Text  schreiben  }
    Repeat  Until  KeyPressed;                  {  auf  Tastendruck  warten  }
```

```
    For i:=1 To 3 Do wiederherstellen(i+17,speicher[i]);
    Window(Lo(w_min),Hi(w_min),Lo(w_max),Hi(w_max));     { aufräumen }
End;                                                       { Prozedur }

Begin                                      { Hauptprogramm des Units }
   Intr(17,regs);                        { BIOS-Interrupt Nr. 17 rufen }
    If (regs.AX And 48)=48 Then bildschirm:=$B000;
    Else bildschirm:=$B800;
End;                                       { Hauptprogramm des Units }
```

In der ersten Zeile des Units steht das Wort "Unit" und der Name, unter dem es abgespeichert wird.

Dann folgt der Interface-Teil; er wird mit dem Schlüsselwort "Interface" eingeleitet. Als erste Zeile in diesem Teil kann eine Uses-Deklaration enthalten sein. Damit ist es auch innerhalb eines Units möglich, Teile anderer Units zu verwenden.

Danach können Definitionen bzw. Deklarationen globaler Typen, Konstanten und Variablen folgen. In dem Beispiel stehen nur zwei Typendefinitionen. Wenn das Unit in die Uses-Deklaration eines Programms aufgenommen wird, können diese Typen überall im Programm benutzt werden.

Der Implementationsteil, der mit dem Schlüsselwort "Implementation" eingeleitet wird, kann auch Definitionen und Deklarationen enthalten. In dem Beispiel werden die drei Variablen *bildschirm, regs* und *i* deklariert. Diese Variablen sind aber nur innerhalb des Units bekannt.

Die Prozeduren und Funktionen stehen im Implementations-Teil. Damit können sie aber nur innerhalb des Units gerufen werden. Um dieses Problem zu umgehen, müssen sie außerdem im Definitions-Teil deklariert werden. Die Deklaration einer Prozedur oder Funktion besteht ganz einfach aus der ersten Zeile des jeweiligen Moduls.

Im Listing sind nur die Funktion *akzept* und die Prozeduren *Warnung, retten* und *wiederherstellen* im Interface-Teil deklariert. Diese vier Module sind damit global und können von anderen Teilen des Programms gerufen werden. Die Prozedur *fenster* wird dagegen nicht deklariert. Sie ist eine lokale Prozedur, die nur innerhalb des Units gerufen werden kann.

Am Ende des Units steht ein kleines "Hauptprogramm". Dieses Programm wird immer sofort ausgeführt, wenn das Unit in die Uses-Deklaration eines Programms aufgenommen wird. Damit können z.B. Variablen einen Anfangswert zugewiesen bekommen. In unserem Beispiel geht es um die Variable *bildschirm.* In diese Variable wird die Segmentadresse des Videospeichers abgelegt.

Wenn das Unit fertig programmiert ist, muß es compiliert werden. Hierzu wird der Menüpunkt "Compile/Destination" auf "Disk" eingestellt. Dann compilieren Sie das Unit mit "Compile", "Make" oder "Build". Auf der Diskette befindet sich daraufhin eine neue Datei mit dem Namen "Meinunit.tpu". Diese Datei müssen Sie entweder im aktuellen Verzeichnis abspeichern oder in dem Verzeichnis, das im Menüpunkt "Options/Directories/Unit directories" angegeben ist.
Sie können allerdings auch das neue Unit mit dem Hilfsprogramm Tpumover.exe in die Datei Turbo.tpl einfügen. Diese Datei enthält die wichtigsten Standard-Units.
Wenn das Programm von DOS gestartet wird, erscheinen zwei Fenster auf dem Bildschirm. Im linken Fenster werden die Units aus Turbo.tpl gezeigt. Soll Meinunit.tpu in Turbo.tpl eingefügt werden, müssen Sie zuerst das rechte Fenster mit F6 aktivieren. Dann laden Sie Meinunit.tpu mit **F3.** Der Name des Units erscheint jetzt im rechten Fenster. Mit der Plus-Taste wird Meinunit markiert, dabei erscheint an der linken Seite des Namens ein kleiner Pfeil. Jetzt können Sie mit der Taste **Ins** (0 im Zahlenblock) das Unit in das andere Fenster kopieren.
Zum Schluß müssen Sie noch mit F6 in das linke Fenster zurückwechseln und Turbo.tpl mit F2 abspeichern. Das Programm wird mit der Taste **Esc** beendet.

7 Grafikprogrammierung

Die einzige Grafikmöglichkeit, die im Betriebssystem MS-DOS vorgesehen ist, sind ASCII-Grafikzeichen. Das bedeutet aber nicht, daß man auf einem PC keine Grafik programmieren kann. Es heißt nur, daß weitere Grafikfähigkeiten erst durch eine Systemerweiterung in Form von Grafikkarten und zugehörigen Treibern geschaffen werden müssen.

Inzwischen sind fast alle Computer mit Grafikkarten ausgestattet. Trotzdem ergeben sich aus diesem Mangel des Betriebssystems zwei Probleme für den Programmierer. Erstens gibt es viele verschiedene Typen von Grafikkarten. Damit wird es sehr schwierig, ein Grafikprogramm zu schreiben, das ohne Probleme auf jedem Computer läuft. Zweitens muß immer streng zwischen Text- und Grafikmodus unterschieden werden.

7.1 Grafikmodus wählen

Der Unterschied zwischen Text- und Grafikmodus besteht in der Art, wie der Bildschirminhalt gespeichert wird. Im letzten Kapitel wurde beschrieben, wie der Bildschirminhalt in Textmodus gespeichert wird. Für jedes Zeichen auf dem Bildschirm stehen im Speicher zwei Bytes, eins mit dem Zeichen als ASCII-Wert und eins mit der Darstellungsweise.

Im Grafikmodus wird der Bildschirminhalt dagegen Punkt für Punkt (auch Pixel genannt) gespeichert. Für jeden Punkt können ein oder mehrere Bits im Speicher reserviert sein. Ist für jeden Punkt nur ein Bit reserviert, kann der Punkt entweder hell oder dunkel sein. Das Bild ist damit monochrom.

Sollen dagegen auch Farben dargestellt werden, müssen für jedes Pixel mehrere Bits reserviert werden. Auch die Anzahl der Punkte ist für den Speicherbedarf entscheidend. Deshalb können die meisten Grafikkarten in mehreren Modi betrieben werden, entweder mit hoher Auflösung mit wenigen Farben oder in niedriger Auflösung mit vielen Farben.

Ab Version 4.0 enthält Turbo Pascal Treiber für alle gängigen Grafikkarten. Sie- können die Dateien an der Endung .bgi (Borland Graphics Interface) erkennen. Der Treiber wird mit der Standardprozedur *InitGraph* installiert.

Diese Prozedur befindet, sich wie alle Prozeduren und Funktionen, die mit Grafik zu tun haben, im Unit *Graph.* Dieses Unit ist nicht in Turbo.tpl enthalten, sondern in der Datei Graph.tpu. Die Syntax für *InitGraph* lautet:

```
InitGraph(Var treiber <Integer>,
Var modus <Integer>,
pfad : String);
```

Die Prozedur lädt den Treiber von Diskette und aktiviert den Grafikmodus. Der Treiber wird in dem Verzeichnis, das mit dem Parameter *pfad* angegeben ist, gesucht.

Mit dem Parameter *treiber* geben Sie an, welche Grafikkarte der Computer hat und somit, welcher Treiber geladen werden soll. Mit *modus* geben Sie die gewünschte Auflösung an.

Der Treiber muß eine Zahl im Bereich 0 bis 10 sein bzw. eine der im Unit *Graph* definierten Konstanten:

Konstante	*Zahl*	*Bemerkung*
Detect	0	
CGA	1	
MCGA	2	
EGA	3	EGA mit 256 KBytes
EGA64	4	EGA mit 64 KBytes
EGAMono	5	EGA monochrom
IBM8514	6	Nur bei Version 5.x
HercMono	7	Hercules und Kompatible
ATT400	8	
VGA	9	
PC3270	10	

Bei einigen Grafikkarten können Sie, wie erwähnt, zwischen mehreren Modi wählen. Das wird mit dem Parameter *modus* angegeben, modus ist eine Zahl im Bereich zwischen 0 und 5. Für diesen Parameter sind im Unit *Graph* auch verschiedene Konstanten definiert:

Konstante	*Modus*	*Auflösung*	*Farben*
CGAC0	0	320*200	Farben 0
CGAC1	1	320*200	Farben 1
CGAC2	2	320*200	Farben 2
CGAC3	3	320*200	Farben 3
CGAC4	4	640*200	Mono
MCGAC0	0	320*200	Farben 0
MCGAC1	1	320*200	Farben 1
MCGAC2	2	320*200	Farben 2

MCGAC3	3	320*200	Farben 3
MCGACMed	4	640*200	Mono
MCGACHi	5	640*480	Mono
EGALo	0	640*200	16 Farben
EGAHi	1	640*350	16 Farben
EGA64Lo	0	640*200	16 Farben
EGA64Hi	1	640*350	4 Farben
EGAMonoHi	3	640*350	Mono
HercMonoHi	0	720*348	Mono
ATT400C0	0	320*200	Farben 0
ATT400C1	1	320*200	Farben 1
ATT400C2	2	320*200	Farben 2
ATT400C3	3	320*200	Farben 3
ATT400CMed	4	640*200	Mono
ATT400CHi	5	640*400	Mono
VGALo	0	640*200	16 Farben
VGAMed	1	640*350	16 Farben
VGAHi	2	640*480	16 Farben
PC3270Hi	0	720*350	Mono
IBM8514Lo	0	640*480	256 (nur Version 5.x)
IBM8514Hi	1	1024*768	256 (nur Version 5.x)

Bei den Karten CGA, MCGA und ATT400 können in der niedrigen Auflösung drei Farben und die Hintergrundfarbe gezeigt werden. Sie können dabei zwischen vier verschiedenen Farbkombinationen wählen. Die gezeigten Farben sind:

Farben 0: hellgrün, hellrot, gelb
Farben 1: helltürkis, hellviolett, weiß
Farben 2: grün, rot, braun
Farben 3: türkis, violett, hellgrau

Insgesamt gibt es also sehr viele Möglichkeiten. Sie sollten hierbei immer bedenken, daß das Programm auf möglichst vielen verschiedenen Computern lauffähig sein sollte. Um das zu erreichen, sollten sie den Parameter *treiber* bei *InitGraph* den Wert 0 geben. Dann wird *InitGraph* zuerst mit Hilfe der Prozedur *DetectGraph* feststellen, welche Grafikkarte der Computer hat und den entsprechenden Treiber installieren.

InitGraph wird dann auch einen passenden Modus wählen. Welcher Modus gewählt wird, kann mit der Funktion *GetGraphMode* ermittelt werden. Die Syntax der Funktion lautet:

```
modus<Integer>:=GetGraphMode;
```

In der Variablen *modus* wird der aktuelle Grafikmodus abgelegt. Die Werte für *modus* sind dieselben wie bei *InitGraph.*

Manchmal müssen Sie vielleicht zwischen Text- und Grafikmodus hin- und herschalten. Das können Sie mit den beiden Prozeduren *RestoreCrtMode* und *SetGraphMode* erledigen. Die Syntaxen der beiden Prozeduren lauten:

```
RestoreCrtMode;
SetGraphMode (modus <Integer>)-;
```

modus ist der Wert, der mit *GetGraphMode* ermittelt wurde. *RestoreCrtMode* schaltet in den Textmodus zurück. Dabei bleibt der Grafiktreiber allerdings im Speicher. *SetGraphMode* schaltet zurück in den gewählten Grafikmodus. Beachten Sie, daß *GetGraphMode* gerufen werden muß, während der Computer noch im Grafik-Modus ist, also vor *RestoreCrtMode.*

Wenn das Programm abgeschlossen wird, muß auch in den Textmodus zurückgeschaltet werden. Danach muß der Grafiktreiber aus dem Speicher entfernt und der dafür reservierte Speicherbereich freigegeben werden. Dazu dient die Prozedur *CloseGraph,* die ohne Parameter gerufen wird.

Ist das Turbo-Pascal-System auf einer Festplatte im Verzeichnis \Sprachen\Turbo_P installiert, könnte der Rahmen um ein Grafikprogramm beispielsweise so aussehen:

```
Program grafik_beispiel;              { Name des Programms }
Uses Graph;                         { Unit Graph einschließen }
Var treiber,modus : Integer;
Begin
   treiber:=0;
   InitGraph(treiber,modus,'C:\Sprachen\Turbo_P\Graf\');
   modus:=GetGraphMode;              { Grafikmodus ermitteln }
   RestoreCrtMode;                     { Zum Textmodus zurück }
   SetGraphMode(modus);              { Zum Grafikmodus zurück }
   CloseGraph;                          { Grafikmodus beenden }
End;                                      { Ende des Programms }
```

7.2 Ein Beispielprogramm

Grafik kann in vielen Zusammenhängen genutzt werden. Ein wichtiger Einsatzbereich ist die grafische Darstellung statistischer Werte, aus denen wir unser Beispiel entnehmen wollen. Das Programm zeichnet zehn Werte in Form eines Kurven-, Balken und Tortendiagramms.

Zuerst brauchen wir ein Hauptprogramm. Der Aufbau des Programms ist dem des Buchverwaltungsprogramms sehr ähnlich. Allerdings muß für unsere Zwecke das Grafiksystem eingeschaltet werden.

Das Hauptprogramm sieht so aus:

```
Program grafik;                            { Name des Programms }
Uses Crt,Dos,Graph,MeinUnit;
Var stop                    : Boolean;
   taste                    : Char;
   menue_text               : Array[1..6]Of String[15];
   treiber,modus,menue_p    : Integer;
   zahlen                   : Array[1..10]Of Real;
{$I Menue.pas}
{$I Eingabe.pas}
{$I Kurve.pas }
{$I Torte.pas}
{$I Balken2D.pas}
{$I Balken3D.pas}
Begin
   treiber:=0;    { InitGraph soll Grafikkarte ermitteln }
InitGraph(treiber,modus,'C:\Sprachen\Turbo_P\Graf\');
If GraphResult<>0 Then Halt;      { Abbruch falls Fehler }
modus:=GetGraphMode;             { Grafikmodus ermitteln }
stop:=False;
menue_p:=1;
Repeat
   RestoreCrtMode;                    { In Textmodus zurück }
   menue;                              { Menü neu zeichnen }
   TextAttr:=112; GotoXY(2,menue_p+1);
   Write(menue_text[menue_p]);
   taste:=ReadKey;                { Auf Tastendruck warten }
   TextAttr:=7;   GotoXY(2,menue_p+1);
   Write(menue_text[menue-p]);
   Case Taste Of
     #0 : Case ReadKey Of                 { wenn Sondertaste }
       #59 : eingeben;
       #60 : kurve;
       #61 : torte;
       #62 : balken2d;
       #63 : balken3d;
       #64 : If akzept('Wirklich das Programm verlassen')
         Then Begin stop:=True;
         CloseGraph;              { Grafiksystem schließen }
       End;                                 { If Akzept(... }
       #72 : If menue_p>1 Then Dec(menue_p);
       #73 : menue_p:=1;
       #80 : If menue p<6 Then Inc(menue_p);
```

```
        #81 : menue_p:=6;
      End;                              { #0 : Case ReadKey... }
      #13 : Case menue_p Of                          { Return }
        1 : eingeben;
        2 : kurve;
        3 : torte;
        4 : balken2d;
        5 : balken3d;
        6 : If akzept('Wirklich das Programm verlassen')
          Then Begin stop:=True;
          CloseGraph;                 { Grafiksystem schließen }
        End;                               { 6 : If Akzept(... }
      End;                              { #13 : Case MenueP... }
    End;                                      { Case Taste Of }
  Until stop;
End.                                                { Program }
```

Bei dem Versuch, das Grafiksystem mit *InitGraph* zu installieren, können viele Fehler entstehen. Vielleicht hat der Computer gar keine Grafikkarte, vielleicht konnte der Grafiktreiber nicht in dem angegebenen Verzeichnis gefunden werden oder der Speicher reicht nicht aus, um den Treiber zu laden.

Konnte das Grafiksystem nicht korrekt installiert werden, hat es aber wenig Sinn, das Programm weiterlaufen zu lassen. Deshalb sollten Sie immer das Ergebnis von *InitGraph* überprüfen. Das können Sie mit der Funktion *GraphResult* erledigen. Die Syntax der Funktion lautet:

```
ergebnis<Integer>:=GraphResult;
```

Liefert die Funktion das Ergebnis Null, ist die letzte Grafikoperation fehlerfrei gelaufen. Ist dagegen ein Fehler entstanden, liefert die Funktion ein Ergebnis zwischen -l und -15.

Die meisten Funktionen des Programms arbeiten im Grafikmodus. Das Menü muß dagegen im Textmodus gezeigt werden. Deshalb wird innerhalb der Repeat-Until-Schleife in den Textmodus zurückgeschaltet. Wenn eine Funktion im Grafikmodus arbeiten soll, muß nur am Anfang der Funktion in diesen Modus umgeschaltet werden.

Das Programm arbeitet also wechselweise mit dem Text- und Grafikbildschirm. Während der Umschaltung wird der jeweils nicht benutzte Bildschirm gelöscht. Wenn eine Funktimn in Grafikmodus ausgeführt wird, geht folglich das Menü auf dem Textbildschirm verloren. Deswegen muß die Prozedur *menue* unmittelbar nach *RestoreCrtMode* gerufen werden.

Die Prozedur *menue* ist der des Buchverwaltungsprogramms sehr ähnlich. Nur die Anzahl der Menüpunkte und ihre Namen sind geändert. Es lohnt sich deshalb, die alte Prozedur zu laden, mit "Write to" unter einem anderen Namen oder in einem anderen Verzeichnis abzuspeichern und dann entsprechend zu ändern.

```
Procedure menue;                         { Name der Prozedur }
Var i : Integer;
Begin
   menue_text[1]:=' Eingeben   F1 ';
   menue_text[2]:=' Kurve      F2 ';
   menue_text[3]:=' Torte      F3 ';
   menue_text[4]:=' Balken 2D  F4 ';
   menue_text[5]:=' Balken 3D  F5 ';
   menue_text[6]:=' Quit       F6 ';
   Window(60,1,79,12);
   WriteLn(' ┌─────────────────┐');
   For i:=1 To 6 Do WriteLn(' │',menue_text[i],'│')
   WriteLn(' └─────────────────┘');
End;                                             { Prozedur }
```

7.3 Eingabe von Zahlen

Für die verschiedenen Diagramme benötigen wir einige Zahlen. Dazu können Sie natürlich einen richtigen Editor schreiben. Die Eingabe von Zahlen ist aber hier nicht das Hauptanliegen. Deshalb ist die Prozedur so einfach wie möglich gehalten:

```
Procedure Eingeben;                      { Name der Prozedur }
Var i : ShortInt;
Begin
   fenster;
   Window(10,18,65,21);
   For i:= 1 To 10 Do Begin
      GotoXY(3,2);
      ClrEol;
      Write('Zahl Nr',i,':');
      ReadLn(zahlen[i]);
   End;
   Window(10,18,69,21);
   ClrScr;                               { Fenster entfernen }
End;                                             { Prozedur }
```

Erst wird das kleine Fenster gezeichnet, dann kann der Benutzer innerhalb einer Schleife zehn Zahlen eingeben. Für die Eingabe wird die Prozedur *ReadLn* benutzt.

Die Prozedur enthält eine neue Standardprozedur *ClrEol.* Diese Prozedur löscht den Inhalt der aktuellen Textzeile rechts vom Cursor. Hier wird die Prozedur benutzt, um die alte Zahl zu löschen, bevor der Benutzer die nächste Zahl eingibt. Sie dürfen dabei nicht vergessen, daß der Rahmen des Fensters auch aus Textzeichen besteht. *ClrEol* würde deshalb die rechte Seite des Fensters löschen. Die Prozedur arbeitet aber nur innerhalb des mit Window definierten Fensters. Deshalb kann dieser nicht gewünschte Nebeneffekt der Prozedur mit einem neuen Window-Befehl unterbunden werden.

7.4 Kurvendiagramm

Jetzt können die einzelnen grafischen Prozeduren geschrieben werden. Das Array mit den 10 Zahlen wurde im Hauptprogramm als *Array ofReal* deklariert. Damit kann der Benutzer Zahlen aus vielen verschiedenen Bereichen eingeben.
In der ersten Prozedur sollen die 10 Zahlen in Form einer Kurve dargestellt werden. Der Wertebereich der Zahlen kann allerdings sehr unterschiedlich sein, so daß die Skala der Y-Achse variabel sein muß. Sonst könnten einige Zahlen oberhalb des Bildschirmrandes dargestellt werden.

Die Höhe des Bildschirms sollte voll ausgenutzt werden, um die Unterschiede der Zahlen bestmöglich zu zeigen. Auf der anderen Seite darf kein Punkt der Kurve oberhalb des Bildschirms sein. Das können Sie erreichen, wenn Sie zuerst die größte Zahl ermitteln und diese dann an dem oberen Bildschirmrand darstellen lassen.

Ein zweites Problem ergibt sich aus der unterschiedlichen Auflösung der Grafikkarten. Auf dem Grafikbildschirm gibt es ein Koordinatensystem. Wie auf dem Textbildschirm befindet sich der Punkt mit der Koordinate (0,0) an der oberen linken Ecke des Bildschirms. Auf dem Textbildschirmwerden mit den Koordinaten die Zeichen gezählt, auf dem Grafikbildschirm dagegen die Pixel. Deshalb kann die rechte untere Ecke des Bildschirms je nach Auflösung ganz unterschiedliche Koordinaten aufweisen, von (320,200) bei einer CGA-Karte bis (1024,768) bei einem IBM 8514-System.

Mit den beiden Standardfunktionen *GetMaxX* und *GetMaxY* können Sie die Koordinaten des rechten unteren Bildschirmpunktes ermitteln. Die Syntaxen der Funktionen lauten:

```
x_pos<Integer:=GetMaxX;
y_xos<Integer:=GetMaxY;
```

Nachdem wir die Größe bzw. die Auflösung unseres Grafikbildschirmes kennen, setzen wir die zehn Zahlen in Bildschirmkoordinaten um und verbinden diese mit Linien. Zur Orientierung, an welcher Stelle ein Punkt gesetzt oder gelöscht wird, existiert ein Grafikcursor, der sich anfänglich an der Bildschirmkoordinate (0,0) befindet.

Er kann aber mit der Standardprozedur *MoveTo* bewegt werden. Die Syntax der Prozedur ist:

```
MoveTo(x,y <Integer>);
MoveTo(x,y <Integer>;
```

Damit wird der Cursor an den Punkt (x,y) gesetzt. Auf dem Bildschirm ist er allerdings nicht zu sehen. In der Prozedur *kurve* wird *MoveTo* benutzt, um den Cursor an den ersten Punkt der Kurve zu bewegen. Sonst würde die Kurve immer an der oberen linken Bildschirmecke anfangen.

Danach wird die Kurve mit der Prozedur *LineTo* gezeichnet. Die Syntex lautet:

```
LineTo(x,y <Integer>);
```

Mit dieser Prozedur wird eine Linie auf dem Bildschirm von der aktuellen Cursorposition bis zu dem Punkt (x,y) gezogen. Gleichzeitig wird der Grafikcursor an den Punkt (x,y) gesetzt.

Es gibt zwei weitere Prozeduren, mit denen Linien gezeichnet werden können: *LineRel* und *Line.* Die Syntaxen lauten:

```
Line(xl,yl,x2,y2 <Integer>);
LineRel(x_rel,y_rel <Integer>);
```

Die Prozedur *Line* zeichnet eine Linie von (xl,yl) bis (x2,y2). Der Cursor wird dabei allerdings nicht bewegt. *LineRel* zieht eine Linie von der aktuellen Cursorposition bis zu dem Punkt (x,y), wobei die beiden Parameter (x_rel und y_rel) die Entfernung von der Cursorposition bis (x,y) angeben. Bei *LineRelv/ird* der Cursor zu dem Punkt (x,y) bewegt.

Die Kurve könnte mit allen drei Prozeduren gezeichnet werden. Hier wird der Cursor mit *MoveTo* zum ersten Punkt der Kurve bewegt. Dann wird die Kurve innerhalb einer Schleife mit *LineTo* gezeichnet.

Wie aus den Syntaxen hervorgeht, erwarten sowohl *MoveTo* als auch *LineTo* Parameter vom Typ *Integer.* Die Zahlen im Array sind dagegen vom Typ *Real.* Sie müssen deshalb erst umgesetzt werden. Dabei sollen alle Stellen hinter dem

Komma entfernt und der Typ geändert werden. Beides erledigt die Funktion *Round:*

```
ergebnis<Long!nt>:=Round(zahl<Real>);
```

Die Kurve allein hat wenig Aussagekraft. Deshalb sollten Sie die beiden Achsen mit Zahlen versehen. Die Prozeduren *Write* und *WriteLn* können allerdings nur imTextmodus benutzt werden. Im Grafikmodus müssen Sie dagegen *OutText* oder *OutTextXY* benutzen. Die Syntaxen lauten:

```
OutText(text_string <String>);
OutTextXY(x,y <Integer>,text_string <String>);
```

Beide Prozeduren schreiben den String *text_string auf* dem Grafikbildschirm. Der Unterschied besteht darin, das *OutText* an der aktuellen Cursorposition schreibt. *OutTextXY* schreibt dagegen an der Position (x,y).

Die Zahlen, die an den beiden Achsen geschrieben werden, müssen relativ zur Bildschirmauflösung und zum Zahlenbereich berechnet und plaziert werden. Die Ergebnisse dieser Berechnungen sind numerische Typen. *OutTextXY* erwartet aber einen Parameter vom Typ *String.* Deshalb müssen die Zahlen erst mit der Prozedur *Str* umgesetzt werden. Diese Prozedur hat die folgende Syntax:

```
Str(zahl[:breite[:dezimalsteilen]],Var text <String>);
```

zahl ist ein numerischer Ausdruck, *breite* und dezimalsteilen sind Ganzzahlenausdrücke. Der Parameter *zahl* wird als Stringausdruck in *text* geschrieben. Mit *breite* kann angegeben werden, wie viele Zeichen geschrieben werden sollen. Ist *zahl* eine Dezimalzahl, kann mit dem Parameter *dezimalsteilen* angegeben werden, wie viele Stellen hinter dem Komma gezeigt werden sollen. Ist *zahl* dagegen ein *Integertyp,* darf der Parameter dezimalstellen nicht angegeben werden.

Jetzt können Sie die Prozedur schreiben:

```
Procedure Kurve;                                    { Name der Prozedur }
Var i        : Integer;
    maxzahl  : Real;
    a        : Char;
    t        : String[10];
Begin
    SetGraphMode(modus);                         { in Grafikmodus schalten }
    maxzahl:=0;
    For i:=1 To 10 Do If zahlen[i]>maxzahl Then maxzahl:=zahlen[i];
     MoveTo(GetMaxX Div 10,                            { Cursor bewegen }
       GetMaxY-GetMaxY*Round(zahlen[1])Div Round(maxzahl));
    For i:=2 To 10 Do                                  { Kurve zeichnen }
       LineTo(GetMaxX Div 10*i,
```

```
      GetMaxY-GetMaxY*Round(zahlen[i])Div  Round(maxzahl));
    For i:=1 To 10 Do Begin                         { Beschriftung }
      Str(i:2,T);
      OutTextXY(GetMaxX  Div  10*i,GetMaxY-10,T);
      Str(maxzahl/10*i:6:1,T);
      OutTextXY(0,GetMaxY-GetMaxY  Div  10*i,T);
    End;
    a:=ReadKey;                                            { Pause }
End;                                                   { Prozedur }
```

Beachten Sie, daß *OutTextXY* auch den Grafikcursor bewegt. Deshalb dürfen Kurve und Beschriftung nicht innerhalb derselben Schleife ausgeführt werden.

7.5 Balkendiagramme

Die Prozedur, mit der ein Balkendiagramm gezeichnet wird, ist der vorhergehenden sehr ähnlich. Auch hier müssen dieselben Überlegungen und Berechnungen zur Bildschirmauflösung und Zahlenbereich durchgeführt werden. Die Beschriftung der Grafik ist ebenfalls dieselbe.

Statt einer Linie müssen hier Rechtecke mit der Prozedur *Bar* gezeichnet werden. Die Syntax dieser Prozedur ist:

```
Bar(xl,yl,x2,y2 <Integer>);
```

Die Prozedur zeichnet ein Rechteck, wobei (xl,yl) und (x2,y2) die linke obere bzw. die rechte untere Ecke definieren. Das Rechteck wird mit der aktuellen Farbe und dem aktuellen Muster bzw. einer Schraffierung ausgefüllt.

Die Schraffierung wird mit *SetFillStyle* festgelegt. Ein selbstdefiniertes Muster dagegen mit *SetFillPattern.* Die Syntaxen der beiden Prozeduren lauten:

```
SetFillStyle(muster<Word>, farbe<Word>) ;
SetFillPattern(muster<FillPatternType>,farbe<Word>);
```

Bei *SetFillStyle* ist *muster* eine Zahl zwischen 0 und 11. Bei 0 und l wird die Fläche mit der Hinter- bzw. Vordergrundfarbe ausgefüllt, bei allen anderen Werten mit verschiedenen vordefinierten Mustern oder Schraffierungen.

FillPatternType ist im Unit *Graph* als Array[1..8]Of Byte definiert. Die 8 Bytes werden als Bitmuster gelesen, jedes Byte bestimmt dadurch das Muster für eine Zeile. Das Muster wiederholt sich für alle 8 Zeilen. Mit *SetFillPattern* können Sie sehr viele verschiedene Muster erzeugen. Die Berechnung der Muster ist etwas

umständlich, Sie können aber auch ganz einfach das Array mit Zufallszahlen füllen und dann die Ergebnisse betrachten.

Im Beispielprogramm werden nur zehn verschiedene Muster gebraucht. Deshalb benutzen wir hier die Prozedur *SetFillStyle.* Unser Programmlisting sieht so aus:

```
Procedure balken2D;                                      { Name der Prozedur }
Var i        : Integer;
    maxzahl  : Real;
    a        : Char;
    t        : String[10];
Begin
    SetGraphMode(modus);                            { in Grafikmodus schalten }
    maxzahl:=0;
    For i:=1 To 10 Do If zahlen[i]>maxzahl Then maxzahl:=zahlen[i];
    For i:=1 To 10 Do Begin
      SetFillStyle(i,1);                                  { Muster festlegen }
      Bar(GetMaxX Div 10*i-10,
        GetMaxY-12,
        GetMaxX Div 10*i+10,
        GetMaxY-GetMaxY*Round(zahlen[i])Div Round(maxzahl));
    End;
    For i:=1 To 10 Do Begin                                   { Beschriftung }
      Str(i:2,t);
      OutTextXY(GetMaxX Div 10*i,GetMaxY-10,t);
      Str(MaxZahl/10*i:6:1,t);
      OutTextXY(0,GetMaxY-GetMaxY Div 10*i,t);
    End;
    a:=ReadKey;                                                      { Pause }
End;                                                              { Prozedur }
```

Für eine pseudo-dreidimensionale Darstellung verwenden Sie die Prozedur *BarSD* :

```
Bar3D(xl, yl,x2,y2<Integer>,tiefe<Word>,spitze<Boolean>);
```

Die vier ersten Parameter haben dieselbe Bedeutung wie bei *Bar.* Der Parameter *tiefe* gibt an, wie viele Pixel die perspektivische Tiefe ausmachen soll. Hat *spitze* den Wert *True,* bekommt die obere Seite auch eine perspektivische Tiefe.

Um die Prozedur *balkenSd* zu schreiben, müssen Sie in der Prozedur *balken2d* nur diesen einen Befehl ersetzen. *BarBD* muß mit den folgenden Parametern gerufen werden:

```
Bar3D(GetMaxX Div 10*i-10,
      GetMaxY-12,
      GetMaxX Div 10*i+10,
      GetMaxY-GetMaxY*Round(zahlen[i])Div Round(MaxZahl),
      5,True);
```

7.6 Tortendiagramm

Das letzte Diagramm, das gezeichnet werden soll, ist das Tortendiagramm. Hierfür gibt es in Turbo Pascal die Prozedur *PieSlice.* Die Syntax ist:

```
PieSlice(x,y<Integer>,winkeil,winkel2,radius<Word>);
```

Die Prozedur zeichnet einen Ausschnitt aus einem ausgefüllten Kreis. Die Parameter x und y bezeichnen den Mittelpunkt des Kreises und damit die Spitze des Ausschnitts, *winkeil* und *winkeü* bezeichnen Anfangs- und Endwinkel auf dem Kreisbogen. Beide Parameter müssen in Grad angegeben werden. Der Wert 0 steht für einen Winkel direkt rechts vom Zentrum. Bei steigenden Werten wächst der Winkel auf dem Kreisbogen entgegengesetzt dem Uhrzeigersinn an.

Das Diagramm sollte auch beschriftet werden. Bei jedem Tortenausschnitt sollte zumindest die Nummer des Wertes stehen. Die Position dieses Textes ließe sich natürlich mit Hilfe von trigonometrischer Funktionen berechnen.

Einfacher ist es aber mit der Standardprozedur *GetArcCoords:*

GetArcCoords (var bogen_k<ArcCoordsType>) ;

Der Typ *ArcCoordType* ist im Unit *Graph* so definiert:

```
ArcCoordsType = Record
X,Y : integer;
Xstart,Ystart : integer;
Xend,Yend     : integer;
End;
```

Die Prozedur *GetArcCoords* liefert die Koordinaten für den zuletzt gezeichneten Tortenausschnitt. In die Elemente X und Y werden die Koordinaten des Kreismittelpunkts geschrieben. Die vier übrigen Parameter liefern die Koordinaten des Anfangs- und Endpunktes auf dem Kreisbogen. Ausgehend von diesen Zahlen können Sie die Koordinaten des Textes berechnen.

So sieht die letzte Prozedur des Programms aus:

```
Procedure Torte;                                    { Name der Prozedur }
Var i            : ShortInt;
    total,summe  : Real
    a            : Char;
    bogen_k      : ArcCoordsType;
    t            : String[2];
Begin
    SetGraphMode(modus);                      { in Grafikmodus schalten }
    total:=0;
    summe:=0;
    For i:=1 To 10 Do total:=total+zahlen[i];
    For i:=1 To 10 Do Begin
      SetFillStyle(i,1);                              { Muster festlegen }
      PieSlice(GetMaxX Div 2,                { X-Koordinate von Zentrum }
          GetMaxY Div 2,                     { Y-Koordinate von Zentrum }
          Round(summe),                                 { Anfangswinkel }
          Round(summe+360*zahlen[i]/total,                  { Endwinkel }
          GetMaxY Div 2-10);                                   { Radius }
      summe:=summe+360*zahlen[i]/total;
      GetArcCoords(bogen_k);               { Bogenkoordinaten ermitteln }
      Str(i:2,t);                             { i in String übertragen }
      OutTextXY(bogen_k.Xstart-(bogen_k.X-bogen_k.Xstart)Div  10-5,
          bogen_k.Ystart-(bogen_k.Y-bogen_k.Xstart)Div  10,t);
    End;                                   { For i:= 1 To 10 Do Begin }
    a:=ReadKey;                                                 { Pause }
End;                                                         { Prozedur }
```

8 Turtlegrafik

Haben Sie schon einmal von der Programmiersprache Logo gehört? Diese Sprache ist vor allem durch die Turtlegrafik (zu deutsch Schildkrötengrafik) bekanntgeworden. In der Computergrafik, wie wir sie bis jetzt kennengelernt haben, werden Linien und Figuren durch Koordiaten definiert.

In der Turtlegrafik ist dies alles ganz anders. Hier gibt es einen Grafikcursor, die sogenannte Turtle. Sie wird mit verschiedenen Befehlen auf dem Bildschirm dirigiert, wobei sie eine Spur in Form einer Linie hinterläßt. Die Bewegung der Turtle ist hierbei durch eine Richtungs-und Entfernungsangabe bestimmt.Wenn Sie dies geometrisch betrachten, geht es darum, daß eine Linie nicht durch die Koordinaten der Endpunkte, sondern durch die Koordinaten eines Endpunkts sowie Richtung und Länge der Linie definiert wird.

Und was hat das mit Turbo Pascal zu tun? Etwas schon; in Turbo Pascal Version 3.0 gab es nämlich diese Art von Grafik. Ab Version 4.0 wurde zwar das gesamte Grafiksystem wesentlich erweitert und verbessert, aber die Turtlegrafik wurde aufgegeben. Den neueren Turbo-Pascal-Systemen liegen aber zwei Units, *TurboS* und *Graph.3* bei. Diese beiden Units enthalten alle Prozeduren und Funktionen aus Version 3.0. Wenn Sie *GraphS* in die Uses-Deklaration aufnehmen, können Sie auch weiterhin problemlos im CGA-Farbgrafikmodus Turtlegrafik programmieren.

In der Turtlegrafik wird mit einem ganz speziellen Koordinatensystem gearbeitet. Hier liegt der Anfangspunkt nicht in der oberen linken Ecke, sondern in der Mitte des Bildschirms. Die X-Koordinaten steigen von links nach rechts und in der linken Hälfte des Bildschirms negativ. Die Y-Koordinaten steigen von unten nach oben - im Gegensatz zur normalen Grafik, wo die Koordinatenlinie 0 ganz oben ist.

Auch in der Turtlegrafik ist es möglich, ein Fenster zu definieren. Die Prozedur heißt *TurtleWindow* und lautet:

```
TurtleWindow(x_mitte,y_mitte,breite, hoehe <Integer>);
```

Damit wird ein Fenster definiert, wobei das Koordinatensystem so verschoben wird, daß die Koordinate (0,0) in die Mitte des Fensters gelegt wird. Am Anfang steht die Turtle an der Koordinate (0,0) mit der Richtung 0, das heißt nach oben. Mit der Prozedur *Home,* die ohne Parameter gerufen wird, können Sie jederzeit

die Turtle in diese Position zurückbringen. Die vier wichtigsten Prozeduren sind *Forwd, Back, TurnRight* und *TurnLeft,* sie lauten:

```
Forwd(entfernung <Integer>);
Back(entfernung <Integer>);
TurnRight(winkel <Integer>);
TurnLeft(winkel <Integer>);
```

Mit diesen Prozeduren wird die Turtle bewegt bzw. ihre Richtung geändert. *entfernung* ist die Entfernung, wie weit die Turtle bewegt werden soll, in Pixel gemessen; *winkel* wird in Grad gemessen. Sowohl entfernung als auch *winkel* können positiv oder negativ sein.

Wenn die Turtle mit *Forwd* oder *Back* dirigiert wird, wird normalerweise eine Linie auf dem Bildschirm gezogen. Wenn Sie das nicht wollen, müssen Sie vorher die Prozedur *PenUp* rufen. Damit wird - bildlich gesprochen - der Stift der Turtle hochgezogen. Wenn Sie danach *Forwd* oder *Back* rufen, bewegt sich die Turtle zwar, es wird aber keine Linie gezeichnet. Wenn Sie wieder zeichnen wollen, müssen Sie den Stift mit *PenDown* wieder senken. Sowohl *PenUp* als auch *PenDown* werden ohne Parameter gerufen.

Jetzt können Sie das erste Turtlegrafikprogramm schreiben. Bei dieser Grafik kann man durch rekursive Prozeduraufrufe besonders beeindruckende Ergebnisse erzielen, das heißt, mit Prozeduren, die sich selber rufen. Das folgende Programm zeigt ein solches Beispiel:

```
Programm turtleGrafik;                       { Name des Programms }
Uses Graph3,Crt;
Procedure baum(zweig : Integer);
Begin
   If zweig<=0 Then Exit;
   TurnLeft(20);                              { links drehen 20°  }
   Forwd(zweig*4);                                    { vorwärts }
   Baum(zweig-1);                       { rekursiver Prozedurruf }
   Back(zweig*4);                                       { zurück }
   TurnRight(40);                            { rechts drehen 40°  }
   Forwd(zweig*4);                                    { vorwärts }
   Baum(zweig-1);                       { rekursiver Prozedurruf }
   Back(zweig*4);                                       { zurück }
   TurnLeft(20);                              { links drehen 20°  }
End;                                         { Ende der Prozedur }
Begin
   GraphMode;                          { Grafikmodus einschalten }
   PenUp;                                         { Stift heben }
   SetPosition(0,-80);{ Cursor plazieren, ohne zu zeichnen }
```

```
   PenDown;                                  { Stift senken }
   baum(8);                                 { Prozedur rufen }
   Repeat Until KeyPressed;                          { Pause }
   TextMode(1);                       { zum Textmodus zurück }
End.                                   { Ende des Programms }
```

Mit der Prozedur *SetPosition* wird die Turtle an die angegebenen Koordinaten plaziert. Dementsprechend gibt es *SetHeading,* womit die Richtung der Turtle festgelegt werden kann. *SetHeading* wird mit einem Parameter vom Typ *Integer* gerufen, einer Zahl zwischen 0 und 359, womit die Richtung, in Grad gemessen, festgelegt wird.

Im obenstehenden Programm gibt es auch die Prozedur *GraphMode.* Hiermit wurde in Version 3.0 in den Grafikmodus geschaltet. Genau hier liegt aber der Haken. *GraphMode* funktioniert nämlich nur auf einem Computer mit CGA-Farb-Grafikkarte.

In Turbo Pascal wurde erst ab Version 4.0 die Möglichkeit geschaffen, Grafik für Hercules-, EGA- oder VGA-Systeme zu programmieren. Wenn Sie eine dieser Grafikkarten besitzen und nicht auf die Turtlegrafik verzichten wollen, gibt es deshalb nur eine Möglichkeit: Sie müssen die Prozeduren und Funktionen neu schreiben. Wie das gemacht wird, zeigt das folgende Beispiel:

```
Unit kroete;                                          { Name des Units }
Interface
Uses  Graph,Crt;
Procedure penup;                                    { hebt den Stift hoch }
Procedure pendown;           { senkt den Stift auf die Zeichenfläche herab }
Procedure setposition(x,y : Integer); { plaziert die Turtle auf (X,Y) }
Procedure setpencolor(farbe : Word);        { legt die Zeichenfarbe fest }
Function xcor : Integer;             { liefert die X-Koordinate die Turtle }
Function ycor : Integer;             { liefert die Y-Koordinate die Turtle }
Procedure setheading(winkel : Integer);          { legt die Richtung fest }
Function heading : Integer;              { liefert die aktuelle Richtung }
Procedure turnleft(winkel : Integer);      { dreht die Turtle nach links }
Procedure turnright(winkel : Integer); { dreht die Turtle nach rechts }
Procedure turtledelay(zeit : Integer);          { Pause (Millisekunden) }
Procedure home;                    { setzt die Turtle in die Ausgangslage }
Procedure clearscreen;                      { löscht das aktuelle Fenster }
Procedure turtlewindow(xm,ym,b,h : Integer);       { definiert ein Fenster }
Procedure forwd(entfernung : Integer); { bewegt die Turtle nach vorne }
Procedure back(entfernung : Integer);        { bewegt die Turtle zurück }
Procedure graphmode;                           { schaltet in Grafikmodus }
Implementation
```

```
Var  treiber,modus  :  Integer;
       richtung,xpos,ypos,xrel,yrel  :  Real;
      stift  : Boolean;
Procedure  penup;                                    { hebt den Stift hoch }
Begin
    stift:=False;
End;
Procedure  pendown;          { senkt den Stift auf die Zeichenfläche herab }
Begin     stift:=True;
End;
Procedure  setposition(x,y  :  Integer);  { plaziert die Turtle auf (X,Y) }
Begin
     If  stift  Then  Line(Round(xpos),Round(ypos),
                           Round(xrel)+x,Round(yrel)-y);
    xpos:=xrel+x;
    ypos:=yrel-y;
End;
Procedure  setpencolor(farbe  :  Word);         { legt die Zeichenfarbe fest }
Begin
     If  farbe<=GetMaxColor  Then  SetColor(farbe);
End;
Function  xcor  :  Integer;            { liefert die X-Koordinate der Turtle }
Begin
     xcor:=Round(xpos);
End;
Function  ycor  :  Integer;            { liefert die Y-Koordinate der Turtle }
Begin
     ycor:=Round(ypos);
End;
Procedure  setheading(winkel  :  Integer);         { legt die Richtung fest }
Begin
    richtung:=winkel  Mod  360;
End;
Function  heading  :  Integer;               { liefert die aktuelle Richtung }
Begin
     heading:=Round(richtung);
End;
Procedure  turnleft(winkel  :  Integer);     { dreht die Turtle nach links }
Begin
     richtung:=richtung-winkel;
     If  richtung<0  Then  richtung:=richtung+360;
End;
Procedure  turnright(winkel  :  Integer);  { dreht die Turtle nach rechts }
```

```
Begin
     richtung:=richtung+winkel;
     If richtung>360 Then richtung:=richtung-360;
End;
Procedure turtledelay(zeit : Integer);          { Pause (Millisekunden) }
Begin
    Delay(zeit);
End;
Procedure home;                          { setzt die Turtle in die Ausgangslage }
Begin
     If stift Then Line(Round(xpos),Round(ypos),Round(xrel),Round(yrel));
    xpos:=xrel;
    ypos:=yrel;
    richtung:=0;
    pendown;
End;
Procedure clearscreen;                        { löscht das aktuelle Fenster }
Begin
    home;
    ClearViewPort;
End;
Procedure turtlewindow(xm,ym,b,h : Integer);   { definiert ein Fenster }
Begin
    SetViewPort(GetMaxX Div 2+xm-b Div 2,
                GetMaxY Div 2-ym-h Div 2,
                GetMaxX Div 2+xm+b Div 2,
                GetMaxY Div 2-ym+h Div 2,True);
    xrel:=xrel+xm;
    yrel:=yrel-ym;
    PenUp;
    home;
End;
Procedure forwd(entfernung : Integer); { bewegt die Turtle nach vorne }
Var xneu,yneu,winkel : Real;
Begin
     winkel:=Abs(Pi/2-Abs(Pi-(richtung/360*2*Pi)));
     If richtung<180 Then xneu:=xpos+Cos(winkel)*entfernung
     Else xneu:=xpos-Cos(winkel)*entfernung;
     If(richtung>90) And (richtung<270) Then
     yneu:=ypos+Sin(winkel)*entfernung
     Else yneu:=ypos-Sin(winkel)*entfernung;
     If stift Then Line(Round(xpos),Round(ypos),
                        Round(xneu),Round(yneu));
```

```
     xpos:=xneu;
     ypos:=yneu;
End;
Procedure back(entfernung : Integer);        { bewegt die Turtle zurück }
Begin
     forwd(-entfernung);
End;
Procedure graphmode;                          { schaltet in Grafikmodus }
Begin
     treiber:=0;
     InitGraph(treiber,modus,'C:\Sprachen\Turbo_P\Graf\');
     If GraphResult<>0 Then Halt;
     xrel:=GetMaxX Div 2;
     yrel:=GetMaxY Div 2;
     xpos:=xrel;
     ypos:=yrel;
End;
Begin                                         { Hauptprogramm des Units }
     richtung:=0;
     stift:=True;
End.                                                 { Ende des Units }
```

Diese Variablen sind nur innerhalb des Units bekannt. Aus dem Programm heraus können sie nur indirekt, mit Hilfe der Prozeduren des Units, geändert werden.

Die Variable *richtung* enthält zu jeder Zeit die Richtung der Turtle. Der Wert dieser Variablen wird unter anderem von *TurnLeft* und *TurnRight* geändert. In den beiden Variablen *xpos* und *ypos* steht die Position der Turtle, xpos und ypos sind Koordinaten des normalen Grafik-Koordinatsystems, es sind also keine Turtlekoordinaten.

Mit *xrel* und *yrel* wird der Unterschied zwischen den beiden Koordinatensystemen festgehalten. Das Turtlekoordinatensystem hat wie erwähnt seinen Anfangspunkt in der Mitte des Bildschirms bzw. in der Mitte des Fensters. Die Werte *vonxrel* und *yrel* sind somit von der Auflösung des Bildschirms und von der Größe und Position eines eventuellen Fensters abhängig.

Mit der Variablen *stift* kann schließlich entschieden werden, ob bei einer Bewegung der Turtle eine Linie gezeichnet werden soll. Hat diese Variable den Wert *True,* wird gezeichnet, sonst nicht.

Die Variablen *richtung, xpos, ypos, xrel* und *yrel* sind vom Typ *Real.* Die Parameter und Ergebnisse der verschiedenen Prozeduren und Funktionen sollen aber vom Typ *Integer* oder *Word* sein. Deshalb muß in vielen Fällen eine Typenänderung (mit *Round)* vorgenommen werden. Man könnte dafür auch intern mit Integervariablen arbeiten. Dann würden sich aber bei den Berechnungen Rundungsfehler ergeben. Diese würden sich bei der Position der Turtle aufsummieren und zu ungenauen Zeichnungen führen.

Innerhalb des Units werden einige Prozeduren und Funktionen benutzt, die noch nicht besprochen wurden.

In der Prozedur *setpencolor* werden die Prozedur *SetColor* und die Funktion *GetMaxColor* benutzt, sie lauten:

```
SetColor(färbe <Word>);
färbe <Word> := GetMaxColor;
```

Mit *SetColor* wird die Farbe für alle nachfolgenden Zeichenoperationen festgelegt, *färbe* ist die Nummer der Farbe, die benutzt werden soll. Die verschiedenen Grafikkarten erlauben unterschiedlich viele Farben. Damit variieren die möglichen Werte *im färbe.*

Der kleinstmögliche Wert *für färbe* ist immer 0. Der größte kann mit *GetMaxColor* gefunden werden. Auf einem Computer mit Hercules-Karte liefert die Funktion den Wert l, auf einem VGA-System dagegen 15.

In der Prozedur *TurtleDelay* wird die Prozedur *Delay* benutzt:

```
Delay(millisekunden <Word>);
```

Die Prozedur hält das Programm für einige Millisekunden an. Die Prozedur *ClearScreen* enthält die Standardprozedur *ClearViewPort:*

```
ClearViewPort;
```

Damit wird der Inhalt des aktuellen Grafikfensters gelöscht und der Grafikcursor wird in die obere linke Ecke des Fensters (Koordinaten 0,0) positioniert.

Ein Grafikfenster wird mit *SetViewPort* definiert:

```
SetViewPort(xl,yl,x2,y2 <Integer>,clipping <Boolean>);
```

Die Parameter xl ,y l ,x2 und y2 bezeichnen die Koordinaten der oberen linken und unteren rechten Ecke des Fensters. Hat *clipping* den Wert *True,* wird jede künftige Grafikausgabe auf das Fenster begrenzt.

Die Parameter der Prozedur *TurtleWindow* bezeichnen dagegen die Mitte, Breite und Höhe des Fensters und nicht die Ecken. Innerhalb der Prozedur müssen die Parameter deshalb unter Berücksichtigung der jeweiligen Auflösung umgesetzt werden.

In der Prozedur *Forwd* müssen die Parameter auch umgesetzt werden. Ausgehend von dem einen Endpunkt, der Richtung und der Länge der Linie muß der zweite Endpunkt gefunden werden. Dazu werden die folgenden vier Funktionen benutzt:

```
A <Num>  := Abs(b <Num>);
A <Real> := Cos(b <Real>)
A <Real> := Sin(b <Real>)
A <Real> := Pi;
```

Abs liefert den absoluten Wert der Parameter, also den Wert ohne Vorzeichen. *Cos* und *Sin* liefern den Cosinus bzw. den Sinus des Arguments. Dabei müssen Sie beachten, daß die beiden Funktionen die Einheit Radianten und nicht Grad zugrundelegen. *Pi* liefert schließlich den Wert von π
Das Unit muß unter dem Namen "Kroete.pas" abgespeichert und compiliert werden. Danach kann es in Programmen wie dem obenstehenden benutzt werden. In diesem Programm ändern Sie lediglich die Uses-Deklaration. Statt *GraphS* geben Sie *Kroete* an. Schon kann das Programm wieder gestartet werden. Es ist dann vielleicht nicht ganz so schnell, dafür läuft es aber auf jedem Computer mit beliebiger Grafikkarte.

9 Objektorientierte Programmierung mit Turbo Pascal 5.5

9.1 Nachteile der strukturierten Programmierung

Wie Sie gesehen haben, wird in der strukturierten Programmierung streng zwischen Daten und Programmcode unterschieden. Jedes Modul beginnt mit der Definition und Deklaration von Daten und endet mit dem eigentlichen Programmcode. Durch diese Trennung werden die Programme übersichtlicher.

Auf der anderen Seite ist diese Trennung nicht ganz realistisch. Während des Programmablaufs sind Daten und Programmcode eng miteinander verbunden. Die Daten können von dem Programmcode verändert werden. Gleichzeitig entscheiden die Daten über den Ablauf des Programms, also welcher Programmcode ausgeführt werden soll. Dazu ein einfaches Beispiel:

```
Program beispiel;
Var i : Integer;
Begin
i:=-10;
While i<10 Do Inc(i); /
End.
```

Die Variable i bekommt durch die Zuweisung "i:=-10;" und durch den Befehl “Inc(i);“ neue Werte. Andererseits wird durch den Wert von i entschieden, wie oft die Schleife durchlaufen wird.

Wenn diese enge Verbindung zwischen Daten und Programmcode nicht ausreichend beachtet wird, können sehr leicht Fehler entstehen und zwar dann, wenn die richtigen Daten mit dem falschen Programmcode zusammengebracht werden oder umgekehrt. Solche Fehler sind oft schwierig zu finden, weil sowohl Daten als auch Programmcode in sich fehlerfrei sind.

Wenn Sie im obenstehenden Beispiel die Variable i nicht als *Integer,* sondern als *Cardinal* deklarieren, wäre dies an sich kein Fehler. Bei der Zuweisung "i:=-10;" würde aber ein Bereichsfehler entstehen. Eine Variable vom Typ *Cardinal* darf niemals einen negativen Wert bekommen.

Ein ganz anderes Problem liegt in der Definition der Variablentypen. Ein zentraler Punkt in unserem Buchverwaltungsprogramm war die Definition eines Records, der die Daten eines Buches aufnehmen konnte. Dabei haben wir nur solche Daten berücksichtigt, die für fast jedes Buch sinnvoll sind.

Wollen Sie das Programm ausbauen, um weitere Daten über jedes Buch erfassen zu können, müssen Sie wahrscheinlich zwischen verschiedenen Gruppen von Büchern unterscheiden. Es gibt einige Daten, die für jedes Buch erfaßt werden müssen, wie z.B. der Titel. Andere Daten kommen dagegen nur bei bestimmten Büchern in Frage. Bei einem Lexikon müssen andere Daten erfaßt werden als bei einem Roman. Dasselbe gilt, wenn das Programm gleichzeitig zur Erfassung von Zeitschriften benutzt werden soll.

Wenn Sie nur einen Record für alle Möglichkeiten definieren, müßte dieser sehr groß sein. Bei der Erfassung von Buchdaten blieben viele Felder des Records unberücksichtigt, was zwangsläufig zur Verschwendung von Speicherplatz führt.

Stattdessen können Sie für jede Buchkategorie einen eigenen Record definieren. Die verschiedenen Records sind sich dann alle mehr oder weniger ähnlich. Jeder Record hat einige Elemente, die vermutlich auch in mehreren anderen Records enthalten sind. Um die Definition zu vereinfachen, können Sie dann die gemeinsamen Elemente in einem besonderen Record sammeln und diesen dann als Element in alle anderen Records einsetzen. Das könnte dann so aussehen:

```
Type medium = Record
                titel,verlag,erschienen : String[30];
              End;
       buch = Record
                basisdaten : medium;
                verfasser : String[30];
                isbn,sprache: String[15];
                seiten : String[5];
                schlagwort : Array[1..5]Of String[30];
              End;
zeitschrift = Record
                basisdaten : medium;
                isbn : String[15];
                seiten : String[3];
                schlagwort : Array[1..10]Of String[30];
              End;
```

Den Titel eines Buches müssen Sie hierbei beispielsweise als "buchl.basisdaten.titel" ansprechen. Das ist umständlich.

Für jeden Record sind außerdem die gleichen oder ähnliche Prozeduren zu schreiben. Sie brauchen z.B. für jeden Record eine Prozedur, die den Inhalt dieses Records auf dem Bildschirm zeigt oder von Diskette lädt.

Wenn das Programm viele ähnliche Records und Prozeduren enthält, erhöht sich aber das Risiko, daß diese miteinander falsch verbunden werden.

Auf Grund solcher Überlegungen entstand die Idee von der *objektorientierten Programmierung.* Es gibt neben Turbo Pascal 5.5 nur wenige Interpreter bzw. Compiler, die diese Art des Programmierens erlauben.

Die objektorientierte Programmierung beruht im wesentlichen auf drei Prinzipien: *Vererbung, Kapselung und Polymorphie.*

9.2 Vererbung

Bei der objektorientierten Programmierung sind Objekte und Records sehr ähnlich. Objekte können Elemente anderer Objekte übernehmen oder "erben". Dadurch kann die obenstehende Definition vereinfacht werden:

```
Type medium = Object
              titel,verlag,erschienen : String[30];
          End;
     buch = Object(medium)
             verfasser : String[30];
             isbn,sprache: String[15];
             seiten : String[5];
             schlagwort : Array[1..5]Of String[30];
          End;
     zeitschrift = Objekt(medium)
                   isbn: String[15];
                   seiten : String[3];
                   schlagwort : Array[1..10]Of String[30];
          End;
```

Die Objekte *buch* und *Zeitschrift* sind als Objekte vom Typ *medium* definiert. Man sagt auch, daß *buch* und *Zeitschrift* Nachkommen von *medium* sind und *medium* der Vorfahr der beiden anderen Objekte ist. Jedes Objekt kann nur einen direkten Vorfahren, aber beliebig viele Nachkommen haben. Allerdings vererbt ein Objekt alle Elemente des Vorfahrs, auch dessen geerbte Elemente. Ein neues Objekt *Fachzeitschrift* vom Typ *Zeitschrift* würde also auch die Elemente von *medium* übernehmen.

Beim Aufruf der Elemente eines Objektes ergibt sich ein zweiter Vorteil gegenüber den Records. Die vererbten Elemente sind nicht Elemente eines Unterrecords, sondern richtige Elemente des Objekts. In einer Variablen *buchl* vom Typ *buch* kann der Titel deshalb direkt als "buchl .titel" angesprochen werden.

Die Vorteile der Vererbung werden besonders bei der Programmierung grafischer Figuren sichtbar. Dazu ein kleines Beispiel:

```
Type punkt = Object
        x,y : Integer;
        End;
        linie = Object(punkt)
            richtung,laenge : Integer;
        End;
        rechteck = Object(punkt)
            breite,hoehe : Integer;
        End;
        kreis = Object(punkt)
            radius : Integer;
        End;
        ellipse = Object(kreis)
            radius2 : Integer;
        End;
        tortenausschnitt = Object(kreis)
            anfangswinkel,endwinkel : Integer;
        End;
```

Hier wird deutlich, wie man immer neue Obj ekte durch Hinzufügen einzelner Elemente schaffen kann.

9.3 Kapselung

Das zweite Prinzip der objektorientierten Programmierung ist die *Kapselung von Daten und Programmcode.* Das bedeutet, daß ein Objekt nicht nur Variablen enthalten kann, sondern auch Prozeduren und Funktionen. Solche Elemente heißen *Methoden des Objekts* und werden genau wie alle anderen Elemente vererbt.

In der Definition des Objekts müssen die Methoden definiert werden. Das heißt, sie müssen als Elemente in das Objekt eingefügt werden, und zwar auf dieselbe Art und Weise wie Prozeduren und Funktionen im Interface-Teil eines Units:

```
Type punkt = Object
          x,y : Integer;
          Procedure zeigen;
          Procedure verstecken;
          Procedure gehezu(neux,neuy : Integer);
        End;
```

Da die Prozeduren vererbt werden, stehen sie gleich in allen sechs Objekten (und in sämtlichen Nachkommen) zur Verfügung.

Es genügt leider nicht, die Prozeduren zu definiere; Sie müssen auch festlegen, was sie tun sollen. Dazu müssen Sie für jede Prozedur eine Implementierung schreiben. Auch das kennen Sie schon von den Units:

```
Procedure punkt.zeigen;
Begin
PutPixel(x,y,1);
End;
```

Da die Prozedur ein Element des Objekts *punkt* ist, muß der Name als "punkt.zeigen" angegeben werden. Bei den Variablen x und y sehen Sie wieder einen Vorteil der Objekte. Die Prozedur zeigen ist innerhalb des Objekts punkt definiert und somit in demselben Gültigkeitsbereich wie die Variablen x und y. Deshalb kann die Prozedur ohne weiteres auf diese Variable zugreifen. Sie müssen also nicht:

PutPixel(punkt.x,punkt.y,1);

schreiben.

Bei den Objekten *linie, rechteck* oder *kreis* muß die Prozedur anders arbeiten. Das erreichen Sie, indem Sie für jedes Objekt eine neue Implementierung schreiben:

```
Procedure rechteck.zeigen;
Begin
     SetFillStyle (1,1);
     Bar(x,y,x+laenge,y+hoehe);
End;
Procedure kreis.zeigen;
Begin
     SetColor (1);
     Circle(x,y,radius);
End;
```

Dabei sollten Sie beachten, daß jede Prozedur die Variablen aus dem eigenen Objekt benutzt. Die Prozedur *rechteck. zeigen* wird automatisch auf die Variablen x und y aus dem Objekt *rechteck* zugreifen. Die Verknüpfung von den richtigen Daten mit dem falschen Programmcode (oder umgekehrt) ist somit ausgeschlossen.

Bemerkenswert dabei ist, daß Sie nicht unbedingt für jedes Objekt eine neue Implementierung schreiben müssen. Bei Prozeduren und Funktionen werden sowohl die Definition als auch die Implementierung vererbt. Eine neue Implementierung müssen Sie nur dann schreiben, wenn die vererbte nicht ausreichend ist.

9.4 Polymorphie

Betrachten wir jetzt die Prozedur *gehezu:*

```
Procedure punkt.gehezu(neux,neuy : Integer);
Begin
     verstecken;
     x:=neux;
     y:=neuy;
     zeigen;
End;
```

Diese Prozedur zeigt, daß es nicht immer notwendig ist, für jedes Objekt eine neue Implementierung zu schreiben. Hier müßte man lediglich dafür sorgen, daß die Prozedur immer auf die richtigen Versionen von *verstecken, zeigen,* x und y zugreift, dann kann dieselbe Implementierung für alle Figuren gelten.

Bei den Variablen wird dies automatisch erledigt. Bei den Unterprozeduren ist es allerdings nicht ganz so einfach. Normalerweise werden die Adressen der Prozeduren direkt in den Programcode geschrieben und es gibt dann keine Möglichkeit, dies während der Laufzeit zu ändern.

Hierbei kommt das dritte Prinzip des objektorientierten Programmierens, die *Polymorphie* oder "das späte Binden", zum Tragen. Dabei werden die Unterprozeduren als sogenante "virtuelle Methoden" deklariert. Die Adressen dieser Methoden werden nicht in den Programmcode, sondern in eine sogenannte "Virtuellen Methoden Tabelle" (VMT) geschrieben. Jedesmal, wenn die Methode ausgeführt werden soll, wird die Adresse aus der Tabelle geholt.

Als Programmierer müssen Sie sich um die VMT gar nicht kümmern. Bei der Definition der Methode geben Sie lediglich das Wort "Virtual" hinter dem Namen der Methode an. Außerdem müssen Sie beachten, daß solche Methoden in jedem Nachkommen des Objekts mit dem Zusatz *Virtual* definiert werden muß. Einmal Virtual, immer Virtual.
Die VMT muß außerdem von einer besonderen Prozedur, einem sogenannten *Constructor,* initialisiert werden. Jedes Objekt, das virtuelle Methoden enthält,

muß auch einen Constructor enthalten, und der Constructor muß immer vor der ersten virtuellen Methode gerufen werden.

Die Initialisierung der VMT geschieht automatisch. Man kann aber (als Nebeneffekt) den Constructor für die Initialisierung von Variablen nutzen, wie in dem nachfolgenden Beispielprogramm gezeigt wird .
Umgekehrt gibt es auch den Prozedur-Typ *Destructor,* der vor Programmende gerufen werden sollte, um verschiedene Aufräumarbeiten zu erledigen.

Und hier ist ein fertiges Programmbeispiel:

```
Program objekt_beispiel;                          { Name des Programms }
Uses  Graph;
Type punkt = Object                                { Objekt-Definition }
        x,y  :  Integer;
        Constructor  anfang(xi,yi  :  Integer);
        Destructor  ende;
        Procedure  zeigen;  Virtual;
        Procedure  verstecken;  Virtual;
        Procedure  gehezu(neux,neuy  :  Integer);
      End;
      rechteck  =  Object(punkt)
        laenge,breite  :  Integer;
        Constructor  anfang(xi,yi,li,bi:  Integer);
        Procedure  zeigen;  Virtual;
        Procedure  verstecken;  Virtual;
      End;
Var  treiber,modus,i  :  Integer;               { Variablen-Deklaration }
      pixel  :  punkt;
      kvadrat  :  rechteck;

Constructor  punkt.anfang(xi,yi  :  Integer);  { Methoden-Implementierungen }
Begin
   x:=xi;
   y:=yi;
End;
Destructor  punkt.ende;
Begin
   verstecken;
End;
Procedure  punkt.zeigen;
Begin
   PutPixel(x,y,1);
End;
Procedure  punkt.verstecken;
```

```
Begin
   PutPixel(x,y,0);
End;
Procedure  punkt.gehezu(neux,neuy  :  Integer);
Begin
   verstecken;
   x:=neux;
   y:=neuy;
   zeigen;
End;
Constructor  rechteck.anfang(xi,yi,li,bi  :  Integer);
Begin
   x:=xi;
   y:=yi;
   laenge:=li;
   breite:=bi
End;
Procedure  rechteck.zeigen;
Begin
   SetColor(1);
   Bar(x,y,x+laenge,y+breite);
End;
Procedure  rechteck.verstecken;
Begin
   SetColor(0);
   Bar(x,y,x+laenge,y+breite);
End;
Begin              { Hauptprogramm }
   treiber:=0;
   InitGraph(treiber,modus,'C:\Sprache\Pas\Graf\');
   pixel.anfang(2,2);
   kvadrat.anfang(20,20,10,10);
   For  i:=1  To  GetMaxY-10  Do
     Begin
       pixel.gehezu(i*2,i);
       kvadrat.gehezu(i*2+20,i+20);
   End;
   pixel.ende;
   kvadrat.ende;
End.                                          { Ende des Programms }
```

Zuerst werden zwei Objekttypen *punkt* und *rechteck* definiert. Von jedem Typ wird eine Varaible deklariert *pixel* und *kvadrat.* Dann folgen die Implementierun-

gen der verschiedenen Methoden, Constructors und Destructors, und zum Schluß ein kleines Hauptprogramm.

Das Programm schaltet in den Grafikmodus und zeigt dann einen Punkt und ein Rechteck, die sich diagonal über den Bildschirm bewegen. Das ist an sich nichts Bemerkenswertes. Sie sollten aber beachten, wieviel Platz die Definition und Deklaration der Objekte innerhalb des Programmes einnehmen. Im Prinzip ist die Implementierung der Methoden ja auch ein Teil der Definition.

Sie können das Programm mit weiteren Objekten wie Kreis, Ellipse, Linie etc. erweitern. Das Programm wird dadurch schnell sehr umfangreich, andererseits aber auch sehr leistungsfähig. Haben Sie erst einmal die richtigen Objekte mit den richtigen Methoden definiert, können Sie mit wenigen Befehlen im Hauptprogramm richtige Animationen erstellen.

10 Fehlersuche und -behandlung

Jedes Programm muß seinen Zweck erfüllen. Es soll bestimmte Aufgaben erledigen, und zwar möglichst einwandfrei. Andererseits nimmt mit wachsender Größe des Programmes die Schwierigkeit zu, alle möglichen Fehler zu vermeiden. Die Fehlersuche und Fehlerbeseitigung ist deswegen bei der Softwareerstellung ein ganz zentrales Thema. Es gibt zwei verschiedene Arten von Fehlern: *Programmierfehler* und *Laufzeitfehler.*

10.1 Programmierfehler

Diese Gruppe unterteilt sich in syntaktische und logische Fehler. Die syntaktischen Fehler sind Verstöße gegen die Regeln der Programmiersprache. Dazu gehören unter anderem Variablen, die nicht deklariert sind, Parameter mit falschen Typen und vergessene Sonderzeichen.

```
Program SyntaxFehler;
Var A : Char;
Begin
   For i:=1 To 10 Do Begin      { i ist nicht deklariert }
     WriteLn(Cos(A));    { A sollte numerischer Typ sein }
   End                                 { Semikolon fehlt }
End.
```

Solche Fehler sind nicht weiter schlimm. Sobald man versucht, das Programm zu compilieren, wird der Compiler die Fehler finden, seine Arbeit unterbrechen und mit einer Fehlermeldung auf die Fehlerstelle hinweisen.

Im Vergleich zu anderen Programmiersprachen sind die Regeln in Pascal relativ streng. Manchmal scheint es, als ob die Programmierung dadurch unnötig umständlich wird. Andererseits bilden diese Regeln die Grundlage für die Fehlersuche des Compilers. Deshalb fängt der Compiler bei der Compilierung auch wesentlich mehr Fehler ab als z.B. ein C-Compiler. Gerade dadurch ist die Arbeit mit Pascal im Endeffekt einfacher.

Die logischen Fehler werden dagegen nicht vom Compiler entdeckt und sind deshalb viel schwieriger zu finden. Man könnte auch sagen, die logischen Fehler seien gar keine Fehler, man hat nur nicht das programmiert, was man eigentlich wollte. Ein typischer logischer Fehler ist die Endlos-Schleife.

```
Program LogikFehler;
Var i : Integer;
Begin
    i:=1;
    While i<10 Do Begin          { die Schleife ist endlos }
      WriteLn('Text');
    End;
End.
```

In diesem Beispiel wurde einfach vergessen, daß die Zählvariable einer While-Do-Schleife innerhalb der Schleife geändert werden muß. Der Compiler hat keine Möglichkeit zu erkennen, daß ein Fehler vorliegt. Es könnte ja sein, daß der Programmierer eine Endlos-Schleife schreiben möchte.

Wenn das Programm mit *Run* gestartet wird, erhält man also keine Fehlermeldung. Das Programm wird compiliert und ist meistens auch lauffähig. Allerdings arbeitet es nicht so wie erwartet. Vielleicht erscheinen Zeichen ohne jeden Zusammenhang auf dem Bildschirm, berechnete Ergebnisse sind völlig falsch oder (und das ist das Schlimmste) es passiert gar nichts.

Es kann sehr schwierig sein, solche Fehler zu finden. Hat man dagegen erkannt, wo der Fehler liegt, ist es meistens sehr einfach, ihn zu beseitigen. Das gilt für das obenstehende Beispiel. Hat man erkannt, daß die Variable i innerhalb der Schleife nicht erhöht wird, ist es sehr einfach, den Befehl "Inc(i)" einzufügen.

Es geht also zunächst darum, die Fehlerstelle zu finden. Dazu sollten Sie erst versuchen, das Programm einige Male laufen zu lassen. Dabei können Sie dann die verschiedenen Funktionen ausprobieren. So läßt sich vielleicht feststellen, daß der Fehler nur bei bestimmten Funktionen auftritt. Haben Sie die Regeln der strukturierten Programmierung eingehalten, wissen Sie dann oft gleich, in welchem Modul der Fehler zu finden ist.

Als nächstes können Sie versuchen, Teile des Programms auszuklammern. Wird ein Teil des Programms in geschweifte Klammern gesetzt, ignoriert der Compiler ihn. Sie können das Programm wiederholt laufen lassen und dann beobachten, ob der Fehler immer noch auftritt. Damit können Sie Schritt für Schritt die Fehlerstelle einkreisen.

Ein besonderes Problem liegt darin, daß die Werte der Variablen nicht sichtbar sind. Auf dem Bildschirm erscheint nur die Ergebnisse des Programms. Daraus müssen Sie dann auf die Werte der Variablen schließen.

Es kann deshalb hilfreich sein, die Werte einzelner Variablen an verschiedenen Stellen im Programm auf dem Bildschirm ausgeben zu lassen. In dem obigen Beispiel könnten Sie den Befehl "Write(i)" in die Schleife einfügen. Dann würden Sie sofort sehen, daß diese Variable ihren Wert nicht ändert. Anhand dieser Information könnten Sie schnell den Fehler finden.

Der Debugger

In Version 5.x wird die Fehlersuche von dem integrierten Debugger unterstützt. Es handelt sich dabei um einen sogenannten *Run-Time-Debugger.* Damit ist es möglich, ein Programm Zeile für Zeile ausführen zu lassen und dabei die Wertveränderungen der Variablen zu beobachten.

Der Debugger wird mit den Funktionen in den Menüs *Run, Debug* und *Break/ Watch* bedient. Erst müssen Sie entscheiden, welche Variablen beobachtet werden sollen. Die erste Funktion in dem Menü *Break/watch* heißt *Add watch.* Wenn Sie diese Funktion wählen, erscheint ein kleines Fenster. Dort kann ein Variablenname eingegeben werden. Wenn mit **Return** abgeschlossen wird, erscheint im Watch-Fenster im unteren Teil des Bildschirms der Name der Variablen, gefolgt von einem Doppelpunkt und dem Text "Unknown identifier".

Werden *mit Add watch* mehrere Variablen hinzugefügt, erweitert sich das Watch-Fenster auf Kosten des Edit-Fensters nach oben. Vor der zuletzt hinzugefügten Variablen erscheint im Watch-Fenster ein kleiner Punkt. Er bezeichnet die aktuelle Variable. Mit der Taste *F6* können Sie zwischen Edit- und Watch-Fenster hin- und herschalten. Wenn das Watch-Fenster das aktuelle Fenster ist, steht der Cursor an der aktuellen Variablen.

Mit den beiden nächsten Funktionen im Break/watch-Menü kann die aktuelle Variable im Watch-Fenster gelöscht bzw. editiert werden. Mit der vierten Funktion "Remove all watches" werden sämtliche Variablen aus dem Watch-Fenster gelöscht.

Mit den Funktionen in dem unteren Teil des Break/watch-Menüs können sogenannte "Breakpoints" in das Programm eingefügt werden. Wenn das Programm mit *Run* gestartet wird, läuft es nur bis zum ersten Breakpoint. Dadurch ist es möglich, bestimmte Programmteile ausführen zu lassen, um den Ablauf des Programms besser zu kontrollieren.

Mit der Funktion "Toggle breakpoint" wird im Edit-Fenster in der Zeile, in der der Cursor steht, ein Breakpoint eingefügt. Ist in dieser Zeile schon ein Breakpoint vorhanden, wird er gelöscht. Mit "Clear all Breakpoints" werden alle Breakpoints aus dem Programm wieder gelöscht, und mit "View next breakpoint" wird die Zeile mit dem nächsten Breakpoint im Edit-Fenster gezeigt.

Soll ein Fehler in einem Programm mit Hilfe des Debuggers gefunden werden, wird zuerst der Programmteil geladen, in dem der Fehler vermutet wird. Der Cursor wird etwas vor die mögliche Fehlerstelle gesetzt, und mit der Menüfunktion "Break/watch/Toggle breakpoint" wird ein Breakpoint eingefügt. Der Breakpoint sollte lieber etwas zu früh als etwas zu spät im Quellcode stehen.

Dann fügen Sie mit "Break/watch/Add watch" alle kritischen Variablen in das Watch-Fenster ein. Sie sollten vor allem alle Zählvariablen in Schleifen berücksichtigen. Solche Variablen verändern ihren Wert sehr oft, und der Verlauf des Programms ist von ihnen direkt abhängig. Danach starten Sie das Programm mit *Run.* Es wird aber nur bis zum eingefügten Breakpoint ausgeführt. Dort wird die Programmausführung automatisch unterbrochen. Im Edit-Fenster erscheint der Quelltext mit dem Breakpoint, wobei die Zeile mit dem Breakpoint invertiert dargestellt wird.

Im Watch-Fenster hat sich dabei auch etwas geändert. Hinter dem Namen der Variablen steht jetzt nicht mehr "Unknown identifier", sondern ein Wert. Es ist der Wert, den die Variable hat, wenn die Zeile mit dem Breakpoint ausgeführt wird.

Mit den Funktionen "Run/Trace into" oder "Run/Step over" können Sie das Programm jetzt Zeile für Zeile ausführen lassen. Die beiden Funktionen können auch mit den Funktionstasten F7 bzw. F8 gerufen werden, was in der Regel einfacher ist. Der Unterschied der beiden Funktionen zeigt sich, wenn im Programm eine selbstgeschriebene Funktion oder Prozedur gerufen wird. Mit *Trace into* wird auch das Untermodul Zeile für Zeile ausgeführt. Mit *Step over* werden Untermodule dagegen normal ausgeführt, als ob das Programm einfach mit *Run* gestartet wäre. Bei der schrittweisen Programmausführung wird im Edit-Fenster immer die aktuelle Zeile invertiert gezeigt. Im Watch-Fenster erscheinen die gewählten Variablen, gefolgt von dem Wert, den sie in der aktuellen Zeile haben.

Wenn das Programm etwas auf den Bildschirm schreiben soll, wird kurz zum Ausgabebildschirm geschaltet. Anschließend wird gleich zum Edit-Fenster zurückgeschaltet; Sie können aber jederzeit mit der Tastenkombination **Alt-FS** den Ausgabebildschirm zeigen lassen.

Auf diese Art können Sie wirklich verfolgen, was im Programm passiert. Sie können die Ausgabe, den Programmablauf und die Werte der Variablen vergleichen und sehen, was unter verschiedenen Umständen geschieht. Damit ist es meistens kein Problem, den Fehler zu finden.

Reicht es immer noch nicht aus, gibt es eine zusätzliche Möglichkeit. Im Menü *Debug* gibt es die Funktion *Evaluate.* Wenn Sie diese Funktion wählen, erscheint ein Fenster mit drei Rubriken: "Evaluate", "Result" und "New value". Damit kann der Wert einer Variablen beliebig geändert werden. Erst müssen Sie in das Feld "Evaluate" den Namen der Variablen eingeben. Dabei erscheint im Feld Result der augenblickliche Wert dieser Variablen. Mit der Cursortaste (unten)

bewegen Sie dann den Cursor zum Feld "New Value". Hier können Sie dann einen neuen Wert für die Variable eingeben. Wenn Sie jetzt die Funktion mit **Esc** beenden, arbeitet das Programm mit dem neuen Wert weiter. Dadurch können Sie ausprobieren, wie sich das Programm bei ganz bestimmten Variablenwerten verhält.

10.2 Laufzeitfehler

Laufzeitfehler entstehen erst während der Programmausführung. Sie sind meist Ergebnisse von Fehlbedienungen durch den Benutzer. Trotzdem sollte man bereits bei der Programmierung auf solche Fehler eingehen. Man kann zwar nicht verhindern, daß der Benutzer Fehler macht. Mit einer durchdachten Benutzerführung kann man die Häufigkeit solcher Fehler wesentlich verringern und außerdem in vielen Fällen Auswirkungen begrenzen.

Eine mögliche Benutzerführung wurde schon besprochen. Dazu gehört das in Kapitel 4 vorgestellte Menüsystem sowie die in Kapitel 5.1 erwähnten Warnungen.

Ein Laufzeitfehler führt in der Regel zu einem Programmabbruch. Wurde das Programm aus dem Editor mit *Run* gestartet, wird die Fehlerstelle im Quellcode automatisch gefunden und im Editfenster gezeigt. Gleichzeitig sehen Sie eine Meldung mit der Fehlerursache. Entsteht der Fehler dagegen in einem fertig compilierten Programm, das von DOS gestartet wurde, ist dies natürlich nicht möglich. Dann wird das Programm abgebrochen und eine Fehlermeldung nach folgendem Muster gezeigt:

```
Runtime error n at xxxx:yyyy
```

Mit n wird die Nummer des Fehlers angegeben. Das ist eine Zahl im Bereich zwischen l und 255. Mit xxxx:yyyy wird die Adresse des Fehlers im Hauptspeicher angegeben.

Mit diesen Angaben können Sie nachträglich die Fehlerstelle im Quellcode finden. Dazu müssen Sie den Quellcode ohne irgendwelche Änderungen in den Editor laden. Im Menü *Compile* gibt es die Funktion *Find error.* Wenn Sie diese Funktion wählen, wird ein kleines Fenster mit dem Titel "Error Address" sichtbar. Hier geben Sie die Adresse aus der Fehlermeldung ein. Die Eingabe wird mit der Taste **Return** abgeschlossen. Dann wird das Programm automatisch neu compiliert, die eingegebene Adresse gesucht und die Stelle im Quellcode im Editfenster gezeigt.

Diese Methode ist während der Programmentwicklung sehr hilfreich. Ist das Programm dagegen fertig, kann sie sehr störend sein. Ein fertiges Programm sollte nicht durch einen Fehler unterbrochen werden. Stattdessen sollte der Benutzer

auf den Fehler aufmerksam gemacht werden und die Möglichkeit bekommen, den Fehler zu korrigieren.

Um die Konsequenzen der Laufzeitfehler zu mindern, müssen Sie bei der Programmierung ständig auf mögliche Fehlerstellen achten. Haben Sie dann eine solche Stelle ausgemacht, müssen Sie versuchen, den Fehler aufzufangen und die Programmausführung in alternative Bahnen zu lenken. In dem Beispiel in Kapitel 7.2 gab es die folgenden Zeilen:

```
InitGraph(treiber,modus,'C:\Sprachen\Turbo_P\Graf\');
If GraphResultoO Then Halt;
```

Bei der Prozedur *InitGraph* kann sehr viel schiefgehen. Vielleicht ist der Grafiktreiber nicht vorhanden oder er befindet sich in einem anderen Verzeichnis. Es könnte auch sein, daß der Computer gar keine Grafikkarte besitzt. Gelingt es nicht, das Grafiksystem zu installieren, hat es aber keinen Sinn, das Programm weiterlaufen zu lassen. Deshalb wird es in diesem Falle mit *Halt* abgebrochen.

Es gibt fünf Arten von Laufzeitfehlern: *DOS-Fehler, I/O-Fehler, kritische und fatale* Fehler sowie die Fehler in dem Grafik-System.

DOS-Fehler entstehen in Verbindung mit der Handhabung von externen Dateien, z.B. wenn das Programm versucht, eine nicht vorhandene Datei zu öffnen. Die Fehlernummern der DOS-Fehler liegen im Bereich zwischen l und 99.
I/O-Fehler entstehen bei dem Versuch, in eine Datei zu schreiben oder daraus zu lesen, z.B. wenn die Datei nicht geöffnet ist. Die Fehlernummern der I/O-Fehler liegen im Bereich von 100 bis 149.

DOS-Fehler und I/O-Fehler entstehen oft an denselben Stellen und können zum Teil dieselben Ursachen haben. Gelingt es nicht, eine Datei zu offen, ist es ein DOS-Fehler. Wenn aber das Programm anschließend versucht, aus der nicht geöffneten Datei zu lesen, ist dies ein I/O-Fehler.

DOS-Fehler führen nicht immer zu einem Programmabbruch. Auf jeden Fall kann man aber mit dem Compilerbefehl {$!-} verhindern, daß DOS-Fehler und auch I/O-Fehler das Programm unterbrechen. Benutzen Sie diesen Befehl, müssen Sie hinter allen kritischen Stellen untersuchen, ob ein Fehler entstanden ist. Darauf muß entsprechend reagiert werden.

Bei einem DOS-Fehler wird die Fehlernummer in die globale Variable *DosError* geschrieben. Hat diese Variable den Wert 0, ist kein Fehler entstanden. Sonst enthält die Variable die Nummer des Fehlers. Entsprechend kann mit der Funktion *I/OResult* untersucht werden, ob ein I/O-Fehler entstanden ist. Die Funktion liefert 0, wenn alles korrekt abgelaufen ist, bzw. die Nummer des letzten I/O-Fehlers. In den Kapiteln 4 und 5 gibt es einige Beispiele, die zeigen, wie DOS- bzw. I/O-Fehler aufgefangen werden.

Kritische Fehler haben Fehlernummern im Bereich zwischen 150 und 199. Dazu gehören vor allem Fehler in Verbindung mit externer Hardware wie Laufwerke und Drucker. Solche Fehler entstehen z.B. beim Versuch, auf eine schreibgeschützte Diskette zu schreiben, wenn man vergessen hat, das Laufwerk zu schließen, wenn der Drucker kein Papier hat oder wenn er nicht angeschlossen ist.

In einem Programm, das mit Turbo Pascal geschrieben ist, führen kritische Fehler immer zu einem Programmabbruch und es gibt keine Möglichkeit, solche Fehler abzufangen. Unter MS-DOS und in den meisten anderen Programmen verursachen solche Fehler eine Meldung vom Typ:

```
Critical error xxx
Abort, Retry, Ignore
```

Dadurch hat man z.B. die Chance, das Laufwerk zu schließen oder neues Papier in den Drucker einzulegen. Das Laufzeitsystem von Turbo Pascal fängt aber diese Meldungen ab und ersetzt sie durch einen unbedingten Programmabbruch.

Auch die relativ seltenen fatalen Fehler führen zu einem Programmabbruch. Die Nummern der fatalen Fehler liegen im Bereich von 200 bis 255. Zu dieser Gruppe gehören Stack-Overflow, Fließpunkt-Overflow und Division durch Null.

Die Fehler im Zusammenhang mit den grafischen Systemen haben negative Fehlernummern im Bereich zwischen -l und -15. Sie entstehen fast ausschließlich bei dem Versuch, einen Grafiktreiber oder einen Grafikfont zu laden, z.B. wenn die Datei nicht gefunden wurde oder der freie Speicher zu klein ist. Wie im Kapitel 7 gezeigt, können solche Fehler mit der Funktion *GraphErrorMsg* ermittelt und aufgefangen werden.

Anhang

A: Standard-Prozeduren und –Funktionen

Wenn Sie Ihre eigenen Programme schreiben, werden Sie oft die Syntax einer Prozedur oder Funktion nachschlagen müssen. Dazu dient dieser Anhang. Er enthält alle Standard-Prozeduren und -Funktionen außer denjenigen, die aus dem Unit *TurboS* oder *GraphS* stammen. Um die Suche zu erleichtern, ist der Anhang thematisch gegliedert.

Es wird hier lediglich die Syntax angegeben, die Variablennamen sind aber so gewählt, daß die Wirkung einer Prozedur oder Funktion oft erkennbar ist. Wenn es z.B. heißt:

```
quadratwurzel_von__a<Real>:=Sqrt(a<Real>);
```

kann man daraus erkennen, daß die Funktion *Sqrt* die Quadratwurzel einer Zahl liefert.

Alle Prozeduren und Funktionen in Abschnitt 8 bis 12 stammen aus Unit *Graph.* Bei den übrigen ist angegeben, aus welchem Unit sie stammen.

Die Namen der Prozeduren und Funktionen sind mit Fettschrift gekennzeichnet und innerhalb des einzelnen Abschnitts alphabetisch sortiert.

Der Typ eines Parameters oder Ergebnisses ist hinter dem Namen in spitzen Klammern angegeben. Sind mehrere Parameter von demselben Typ, wird der Typ nur hinter dem letzten Parameter angegeben.

Sind dagegen Parameter von verschiedenen Typen erlaubt, werden diese durch den geraden Strich (I) getrennt angegeben. Außer den bekannten Typnamen werden zwei Abkürzungen benutzt: Darf ein Parameter von jedem Variablentyp sein, wird dies mit <var> angegeben. In gleicher Weise bezeichnet <num> einen beliebigen numerischen Typ, wobei Parameter und Ergebnis einer Funktion in diesem Fall immer derselbe numerische Typ sein müssen.

Schließlich werden die folgenden vordefinierten Typen aus dem Unit *Dos* und *Graph* benutzt:

```
Unit  Dos:
     DateTime = Record
                    Year,Month,Day,Hour,Min,Sec : Word;
                End;

     Registers = Record
                    Case Integer Of
                        0: (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags : Word);
                        1: (AL,AH,BL,BH,CL,CH,DL,DH : Byte);
                 End;

     SearchRec = Record
                    Fill: Array 1..21 Of Byte;
                    Attr: Byte;
                    Time: Longint;
                    Size: Longint;
                    Name: String[12];
                 End;
```

In Version 5.x gibt es zusätzlich die Typen:
  PathStr = String[79];
  DirStr = String[67];
  NameStr = String[8];
  ExtStr = String[4];

```
Unit  Graph
     ArcCoordsType  =  Record
                          X, Y           : Integer;
                          Xstart,Ystart  : Integer;
                          Xend,Yend      : Integer;
                       End;

     FillPatternType  =  Array 1..8 Of Byte;

     FillSettingsType  =  Record
                             Pattern : Word;
                             Color   : Word;
                          End;

     LineSettingsType  =  Record
                             LineStyle : Word;
                             Pattern   : Word;
                             Thickness : Word;
                          End;
```

```
PaletteType = Record
                Size   : Byte;
                 Colors : Array 0..MaxColors Of Shortint;
              End;

TextSettingsType = Record
                     Font      : Word;
                     Direction : Word;
                     CharSize  : Word;
                     Horiz     : Word;
                     Vert      : Word;
                   End;

ViewPortType = Record
                 X1,Y1,X2,Y2 : Integer;
                 Clip        : Boolean;
               End;
```

1. Numerische Operationen

a_ohne_vorzeichen<num>:=**Abs**(a<num>);
arcustangens_von_a<Real>:=**ArcTan**(a<Real>);
cosinus_von_a<Real>:=**Cos(**a<Real>);
Dec(a<num>,b<Longint>);
a<Real>:=**Exp**(b<Real>);
nachkommateil_von_a<Real>:=**Frac**(a<Real>);
oberes_byte_von_a<Byte>:=**Hi**(a<Integer|Word>);
Inc(a<num>,b<Longint>);
vorkommateil<Real>:=**Int**(a<Real>);
a<Real>:=**Ln**(b<Real>);
niederes_byte_von_a<Byte>:=**Lo**(a<Integer|Word>);
true_wenn_a_ungerade<Boolean>:=**Odd**(a<Longint>);
a<Real>:=**Pi**;
vorgänger_von_a<num>:=**Pred**(a<num>);
zufallszahl<Word|Real>:=**Random** (a<Word>);
Randomize;
abgerundete_wert_von_a<Longint>:=**Round**(a<Real>);
sinus_von_a<Real>:=**Sin**(a<Real>);
quadrat_von_a<num>:=**Sqr**(a<num>);
quadratwurzel_von_a<Real>:=**Sqrt**(a<Real>);
nachfolger_von_a<num>:=**Succ**(a<num>);
a<Integer|Word>:=**Swap**(b<Integer|Word>);

2. Typenkonvertierung

c<Char>:=**Chr**(ascii_wert<Byte>);
ascii_wert<Longint>:=**Ord**(c<Char|set-element>);
str(a<num>:b:c<Integer>,Var a_als_string<String>);
vorkommateil_von_a<Longint>:=**Trunc**(a<Real>);
Val(s<String>,Var s_als_zahl<num>,Var fehler<Integer>);

3. Stringoperationen

s1_s2_sn<String>:=**Concat**(s1,s2, ...,sn<String>);
s1<String>:=**Copy**(s2<String>,anfang,anzahl_zeichen<Integer>);
Delete(s<String>,anfang,anzahl_zeichen<Integer>);
Insert(s1,Var s2<String>,anfang<Integer>);
dynamische_länge_von_s<Integer>:=**Length**(s<String>);
position_von_s1_in_s2<Byte>:=**Pos**(s1,s2<String>);
grossbuchstabe<Char>:=**UpCase**(c<Char>);

4. I/O-Operationen

Append(datei<Text>);
Assign(datei<File Of..|Text>,dos_dateinamen<String>);
AssignCrt(Var datei<text>); Crt
BlockRead(Var datei<File Of ...>,Var puffer<Array Of Byte|Char>,
anzahl,Var ergebnis <Word>);
BlockWrite(Var datei<File Of ...>,Var Puffer<Array Of Byte|Char>,
anzahl,Var ergebnis <Word>);
ChDir(pfad<String>);
Close(datei<File Of..|Text>);
ClrEol; Crt
ClrScr; Crt
DelLine; Crt
freie_bytes<Longint>:=**DiskFree**(laufwerk<Byte>); Dos
gesamt_bytes<Longint>:=**DiskSize**(laufwerk<Byte>); Dos
a<Boolean>:=**Eof** (datei<File Of ...|Text>) ;
a<Boolean>:=**Eoln**(datei<Text>);
Erase(datei<File Of ...|Text>);
absolut<PathStr>:=**FExpand**(relativ<PathStr>); **(nur 5.x)**
cursorposition<Longint>:=**FilePos**(datei<File Of ...|Text>);
elemente<Longint>:=**FileSize**(datei<File Of..|untyped>);
FindFirst(pfad<String>,attr<Word>,Var r<SearchRec>); Dos
FindNext(Var r<SearchRec>); Dos

Flush(Var datei<Text>);

pfad<PathStr>:=**FSearch**(datei<PathStr>,
verzeichnisse<String>); — Dos (nur 5.x)

FSplit(pfad<PathStr>,Var verzeichnis<DirStr>,
Var name<NameStr>,Var endung<ExtStr>); — Dos (nur 5.x)

GetCBreak(Var a<Boolean>); — Dos (nur 5.x)

Getdir(laufwerk<Byte>,Var pfad<String>);

GetFattr(Var datei<File>,Var attr<Word>); — Dos

GetFTime(Var datei<File>,Var zeit<Longint>); — Dos

GetVerify(Var verify_flag<Boolean>); — Dos

GotoXY(spalte,zeile<Byte>); — Crt

HighVideo; — Crt

InsLine; — Crt

i_o_ergebnis<Word>:=**IOResult**;

true_wenn_taste_gedrueckt<Boolean>:=**KeyPressed**; — Crt

LowVideo; — Crt

MkDir(pfad<String>);

NormVideo; — Crt

NoSound; — Crt

PackTime(Var time<DateTime>,Var zeit<Longint>); — Dos

Read(Var datei<File Of ...|Text>,Var w1,w2,...,wn);

ascii_wert<Char>:=**ReadKey**; — Crt

Readln(Var datei<Text>,Var w1,w2,...,wn);

Rename(Var datei<File>,name<String>);

Reset(Var datei<File>,record_groesse<Word>);

Rewrite(Var datei<File Of ...|Text>,record_groesse<Word>);

RmDir(pfad<String>);

Seek(Var datei<File Of ...>,position<Longint>);

a<Boolean>:=**SeekEof**(Var datei<Text>);

a<Boolean>:=**SeekEoln**(Var datei<Text>);

SetCBreak(a<Boolean>); — Dos (nur 5.x)

SetFattr(Var datei<File>,attr<Word>); — Dos

SetFTime(Var datei<File>,zeit<Longint>); — Dos

SetTextBuf(Var datei<Text>,Var puffer<Var>,groesse<Word>);

SetVerify(verify<Boolean>); — Dos (nur 5.x)

Sound(frekvenz: Word); — Crt

TextBackground(farbe<Byte>); — Crt

TextColor(farbe<Byte>); — Crt

TextMode(modus<Word>); — Crt

Truncate(Var datei<File Of..|Text>);

UnpackTime(zeit<Longint>,Var time<DateTime>); — Dos

text_spalte<Byte>:=**WhereX**; — Crt

text_zeile<Byte>:=**WhereY**; — Crt

Window(x1,y1,x2,y2<Byte>); Crt
Write(Var datei<File Of...|Text>,w1,w2,...,wn);
Writeln(Var datei<Text>,w1,w2,...,wn);

5. Speicher-Verwaltung

adresse<Pointer>:=**Addr**(x<Var|Procedure|Function>);
Dispose(Var p<Pointer>);
FillChar(Var anfang<Var>,anzahl<Word>,zeichen<Char>);
FreeMem(Var adresse<Pointer>,bytes<Word>);
GetMem(Var adresse<Pointer>,bytes<Word>);
Mark(Var point<Pointer>);
groesster_verfuegbarer_ram_block<Longint>:=**MaxAvail**;
freier_ram_speicher<Longint>:=**MemAvail**;
Move(Var quelle,ziel<Var>,anzahl_bytes<Word>);
New(Var p<Pointer>);
offset_adresse<Word>:=**Ofs**(x<Var|Procedure|Function>);
p<Pointer>:=**Ptr**(segment,offset<Word>);
Release(Var point<Pointer>);
segment_adresse<Word>:=**Seg**(x<Var|Procedure|Function>);
groesse_in_bytes<Word>:=**SizeOf**(x);

6. Programmablauf

Delay(millisekunden<Word>);	Crt
exitcode<Word>:=**DosExitCode**;	Dos
Exec(pfad,commanline<String>);	Dos
Exit;	
Halt(exitcode<Word>);	
Keep(exitcode<Word>);	Dos
OvrClearBuf;	Overlay (nur 5.x)
puffer_groesse<Longint>:=**OvrGetBuf**;	Overlay (nur 5.x)
OvrInit(overlay_datei_name<String>);	Overlay (nur 5.x)
OvrInitEMS;	Overlay (nur 5.x)
OvrSetBuf(puffer_groesse<LongInt>);	Overlay (nur 5.x)
anzahl_parameter<Word>:=**ParamCount;**	
Parameter<String>:=**Paramstr**(nummer<Word>);	

7. Systemnahe Programmierung

register_inhalt<Word>:=**CSeg**;	
version_nummer<Word>:=**DosVersion**;	(nur 5.x)
Register_Inhalt<Word>:=**DSeg**;	
anzahl_DOS_Strings<Integer>:=**EnvCount**;	Dos (nur 5.x)

dos_string<String>:=**EnvStr**(nummer<Integer>); Dos (nur 5.x)
GetDate(Var jahr,monat,tag,wochentag<Word>); Dos
s<String>:=**GetEnv**(envvar<String>); Dos (nur 5.x)
GetIntVec(interrupt_nr<Byte>,Var vektor<Pointer>); Dos
GetTime(Var stunde,minute,sekunde,sek_100<Word>); Dos
Intr(interrupt_nr<Byte>,Var b<registers>); Dos
MsDos(Var b<registers>); Dos
SetDate(jahr,monat,tag<Word>); Dos
SetIntVec(interrupt_nr<Byte>,vektor<Pointer>); Dos
SetTime(stunde,minute,sekunde,sek_100<Word>); Dos
stack_pointer<Word>:=**SPtr**;
register_inhalt<Word>:=**SSeg**;
SwapVectors; (nur 5.x)

8. Grafik-Grundeinstellungen

ClearDevice;
ClearViewPort;
CloseGraph;
DetectGraph(Var graf_driver,graf_modus<Integer>);
GetaspectRatio(Var x_aspect,y_aspect<Word>);
name<String>:=**GetDriverName**; (nur 5.x)
modus<Integer>:=**GetGraphMode**;
maximaler_modus<Integer>:=**GetMaxMode**; (nur 5.x)
rechte_spalte<Integer>:=**GetMaxX**;
untere_zeile<Integer>:=**GetMaxY**;
name<String>:=**GetModeName**; (nur 5.x)
GetModeRange(treiber<Integer>,Var min_modus,max_modus<Integer>);
GetViewSettings(Var v<ViewPortType>);
GraphDefaults;
InitGraph(Var driver<Integer>,Var modus<Integer>,pfad<String>);
a<Integer>:=**InstallUserDriver**(datei<String>,p<Pointer>); (nur 5.x)
a<Integer>:=**RegisterBGIdriver**(driver<Pointer>);
RestoreCrtMode;
SetactivePage(seite<Word>);
SetaspectRatio(x_aspect,x_aspect<Word>); (nur 5.x)
SetGraphBufSize(puffer_groesse<Word>);
SetGraphMode(modus<Integer>);
SetViewPort(x1,y1,x2,y2<Integer>,clipping<Boolean>);
SetVisualPage(seite<Word>);#

9. Farben und Muster

hintergrundfarbe<Word>:=**GetBkColor**;
zeichenfarbe<Word>:=**GetColor**;
GetDefaultPalette(standardpalette<Palettetype>); (nur 5.x)
GetFillPattern(Var muster<FillPatternType>);
GetFillSettings(Var s<FillSettingsType>);
GetLineSettings(Var s<LineSettingsType>);
groesste_farbnummer<Word>:=**GetMaxColor**;
GetPalette(Var aktuelle_palette<PaletteType>);
farb_anzahl<Word>:=**GetPaletteSize**; (nur 5.x)
farbe<Word>:=**GetPixel**(x,y<Integer>);
SetallPalette(Var palette);
SetBkColor(hintergrundfarbe<Word>);
SetColor(zeichenfarbe<Word>);
SetFillPattern(muster<FillPatternType>,farbe<Word>);
SetFillStyle(muster<Word>,farbe<Word>);
SetLineStyle(stil,muster,breite<Word>);
SetPalette(nummer<Word>,farbe<Shortint>);
SetRGBPalette(farbNr,rot,gruen,blau<Integer>); (nur 5.x)
SetWriteMode(modus<Integer>); (nur 5.x)

10. Figuren

arc(x,y<Integer>,anfang,ende,radius<Word>);
Bar(x1,y1,x2,y2<Integer>);
Bar3D(x1,y1,x2,y2<Integer>,tiefe<Word>,spitze <Boolean>);
Circle(x,y<Integer>,radius<Word>);
DrawPoly(anzahl<Word>,Var polypoints);
Ellipse(x,y<Integer>,anfang,ende,x_radius,y_radius<Word>);
FillEllipse(x,y<Integer>,x_radius,y_radius<Word>); (nur 5.x)
FillPoly(anzahl_punkte<Word>,Var polypoints);
FloodFill(x,y<Integer>,rand<Word>);
bogen<ArcCoordsType>:=**GetarcCoords**;
GetImage(x1,y1,x2,y2<Integer>,Var bitmap);
pixel_spalte<Integer>:=**Getx**;
pixel_zeile<Integer>:=**Gety**;
groesse_in_bytes<Word>:=**ImageSize**(x1,y1,x2,y2<Integer>);
Line(x1,y1,x2,y2<Integer>);
LineRel(relativ_x,relativ_y<Integer>);
LineTo(x,y<Integer>);
MoveRel(relativ_x,relativ_y<Integer>);
MoveTo(x,y<Integer>);

PieSlice(x,y<Integer>,anfang,ende,radius<Word>);
PutImage(x,y<Integer>,Var bitmap,modus<Word>);
PutPixel(x,y<Integer> farbe<Word>);
Rectangle(x1,y1,x2,y2<Integer>);
Sector(x,y<Integer>,anfang,ende,x_radius,y_radius<Word>); (nur 5.x)

11. Text im Grafikmodus

GetTextSettings(Var setting<TextSettingsType>);
font_nr<Integer>:=**InstallUserFont**(font_name<String>); (nur 5.x)
OutText(text<String>);
OutTextxy(x,y<Integer>,text<String>);
font_nr<Integer>:=**RegisterBGIfont**(font<Pointer>);
SetTextJustify(horizontal,vertikal<Word>);
SetTextStyle(font,richtung,vergroesserung<Word>);
SetUserCharSize(mult_x,div_x,mult_y,div_y<Word>);
hoehe_in_pixel<Word>:=**TextHeight**(text<String>);
breite_in_pixel<Word>:=**TextWidth**(text<String>);

12. Fehlerbehandlung

fehlermeldung<String>:=**GraphErrorMsg**(fehler_nr<Integer>);
fehler_br<Integer>:=**GraphResult**;

B: Compilerbefehle

Mit Compilerbefehlen kann entschieden werden, wie ein Programmtext compiliert werden soll. Viele dieser Befehle können aber auch vor der Compilierung im Menü Options/Compiler festgelegt werden. Im folgenden sind die meisten dieser Befehle zusammen mit den entsprechenden Menüs angegeben, wobei die Standardeinstellung als erstes steht.

1. Globale Einstellungen

Die globalen Compilerbefehle dürfen nur einmal am Anfang des Programms stehen. Diese Einstellungen können einfacher mit den ensprechenden Punkten im Menü eingestellt werden.

{$A+}, {$A-}	O/C/Align data	(nur 5.x)
{$D+}, {$D-}	O/C/Debug information{$E+},	
{$E-}	O/C/Emulation	(nur 5.x)
{$L+}, {$L-}	O/C/Link buffer{	
$M S,Min,Max}	O/C/Memory Sizes	
{$N-}, {$N+}	O/C/Numeric Processing	
{$O-}, {$O+}	O/C/Overlays allowed	(nur 5.x)
{$T-}, {$T+}	O/C/Turbo pascal map file	(nur 4.0)

2. Lokale Einstellungen

Die lokalen Befehle haben nur bis zum nächsten Befehl von demselben Typ Gültigkeit. Sie können überall dort eingefügt werden, wo Kommentare auch erlaubt sind.

{$B-}, {$B+}	O/C/Boolean evaluation
{$F-}, {$F+}	O/C/Force far calls
{$I+}, {$I-}	O/C/I/O checking
{$R-}, {$R+}	O/C/Range checking
{$S+}, {$S-}	O/C/Stack checking
{$V+}, {$V-}	O/C/Var-string checking

3. Dateien einbinden

Mit diesen Befehlen können fremde Dateien in ein Programm eingebunden werden. Die Dateien werden in den unter *Options/Directories* angegebenen Verzeichnissen gesucht. Quelldateien werden statt des Befehls in den Programmtext eingefügt und mitcompiliert. Objektdateien werden vom Linker eingebunden.

{$I Quelldateiname}	O/D/Include directories
{$L Objektdateiname}	O/D/Objekt directories

C: Tastaturbelegung im Editor

Die sekundäre Tastaturbelegung kann mit dem Installationsprogramm nach Wunsch geändert oder ergänzt werden. Außer mit den hier angegebenen Tastenkombinationen können viele Funktionen mit den Funktionstasten gewählt werden. Diese sind in den entsprechenden Menüs oder in der unteren Bildschirmzeile angegeben.

	Primär:	Sekundär:
Cursor bewegen zu:		
Neue Zeile	Ctrl+M	Ctrl+M
Zeichen links	Ctrl+S	Pfeil links
Zeichen rechts	Ctrl+D	Pfeil rechts
Wort links	Ctrl+A	Ctrl+Pfeil links
Wort rechts	Ctrl+F	Ctrl+Pfeil rechts
Zeile oben	Ctrl+E	Pfeil oben
Zeile unten	Ctrl+X	Pfeil unten
Zeilenanfang	Ctrl+Q+S	Home
Zeilenende	Ctrl+Q+D	End
Erste Zeile	Ctrl+Q+E	Ctrl+Home
Letzte Zeile	Ctrl+Q+X	Ctrl+End
Textanfang	Ctrl+Q+R	Ctrl+PgUp
Textende	Ctrl+Q+C	Ctrl+PgDn
Nächster Fehler	Ctrl+Q+W	
Blockanfang	Ctrl+Q+B	
Blockende	Ctrl+Q+K	
Letzte Position	Ctrl+Q+P	
Marke 0	Ctrl+Q+0	
Marke 1	Ctrl+Q+1	
Marke 2	Ctrl+Q+2	
Marke 3	Ctrl+Q+3	
Klammeranfang	Crtl+Q+[	
Klammerende	Crtl+Q+]	
Tabulatormarke	Ctrl+I	
Scrollfunktionen:		

Zeile nach oben	Ctrl+W	
Zeile nach unten	Ctrl+Z	
Seite nach oben	Ctrl+R	PgUp
Seite nach unten	Ctrl+C	PgDn

Blockfunktionen:

Blockanfang setzen	Ctrl+K+B
Blockende setzen	Ctrl+K+K
Wort markieren	Ctrl+K+T
Blockmarken entfernen	Ctrl+K+H
Block kopieren	Ctrl+K+C
Block verschieben	Ctrl+K+V
Block löschen	Ctrl+K+Y
Block von Diskette lesen	Ctrl+K+R
Block auf Diskette schreiben	Ctrl+K+W

Block ausdrucken	Ctrl+K+P	
Einfügen/Löschen:		
Einfügemodus wählen	Ctrl+V	Ins
Zeile einfügen	Ctrl+N	
Zeile löschen	Ctrl+Y	
Zeilenrest löschen	Ctrl+Q+Y	
Wort löschen	Ctrl+T	
Zeichen löschen	Ctrl+G	Del
Zeichen links löschen	Ctrl+BS	Ctrl+H
Optionen einfügen	Ctrl+O+O	
Kontrollzeichen schreiben	Ctrl+P	
Zeile wiederherstellen	Ctrl+Q+L	
Übrige Funktionen:		
Marke 0 setzen	Ctrl+K+0	
Marke 1 setzen	Ctrl+K+1	
Marke 2 setzen	Ctrl+K+2	
Marke 3 setzen	Ctrl+K+3	
Editor verlassen	Ctrl+K+D	Ctrl+K+Q
Einrücken wählen	Ctrl+O+I	Ctrl+Q+I
Tabulator wählen	Ctrl+O+T	Ctrl+Q+T
String suchen	Ctrl+Q+F	
String austauschen	Ctrl+Q+A	
Suchvorgang wiederholen	Ctrl+L	
Hilfe zu Sprachelement	Ctrl+F1	
Datei speichern	Ctrl+K+S	

D: ASCII-Zeichentabelle

Der ASCII-Wert eines Zeichens kann mit der Funktion *Ord* gefunden werden. Umgekehrt kann das Zeichen mit der Funktion *Chr* gefunden werden bzw. mit dem Operator #.

Beispiel:
a = Chr(65)
a = #65
65 = Ord(a)

Die ersten 32 Zeichen sind Sonderzeichen, die nicht auf dem Bildschirm wiedergegeben werden können. Sie werden u.a. für die Steuerung eines Druckers benutzt. Interessant sind dabei vor allem die Zeichen mit den Nummern 7,8,9,13 und 27. Mit dem ersten ist es möglich, einen Ton im Lautsprecher zu erzeugen. Die vier anderen können direkt über die Tastatur eingegeben und abgefragt werden.

Nr.	*Zeichen*	*Nr.*	*Zeichen*	*Nr.*	*Zeichen*	*Nr.*	*Zeichen*
000	NUL	034	"	068	D	102	f
001	SOH	035	#	069	E	103	g
002	STX	036	$	070	F	104	h
003	ETX	037	%	071	G	105	i
004	EOT	038	&	072	H	106	j
005	ENQ	039	'	073	I	107	k
006	ACK	040	(	074	J	108	l
007	Glocke	041	)	075	K	109	m
008	Backspace	042	*	076	L	110	n
009	Tabulator	043	+	077	M	111	o
010	LF	044	,	078	N	112	p
011	HT	045	-	079	O	113	q
012	FF	046	.	080	P	114	r
013	Return	047	/	081	Q	115	s
014	SO	048	0	082	R	116	t
015	SI	049	1	083	S	117	u
016	DLE	050	2	084	T	118	v
017	DC1	051	3	085	U	119	w
018	DC2	052	4	086	V	120	x
019	DC3	053	5	087	W	121	y
020	DC4	054	6	088	X	122	z
021	NAK	055	7	089	Y	123	{
022	SYN	056	8	090	Z	124	|
023	ETB	057	9	091	[	125	}
024	CAN	058	:	092	\	126	~
025	EM	059	;	093	]	127	Delete
026	SUB	060	<	094	^	128	Ç
027	Escape	061	=	095	_	129	ü
028	FS	062	>	096	`	130	é
029	GS	063	?	097	a	131	â
030	RS	064	@	098	b	132	ä
031	US	065	A	099	c	133	à
032	Space	066	B	100	d	134	å
033	!	067	C	101	e	135	ç

Nr.	*Zeichen*	*Nr.*	*Zeichen*	*Nr.*	*Zeichen*	*Nr.*	*Zeichen*
136	ê	166	ª	196	─	226	Γ
137	ë	167	º	197	┼	227	π
138	è	168	¿	198	╞	228	Σ
139	ï	169	⌐	199	╟	229	σ
140	î	170	¬	200	╚	230	µ
141	ì	171	½	201	╔	231	τ
142	Ä	172	¼	202	╩	232	Φ
143	Å	173	¡	203	╦	233	Θ
144	É	174	«	204	╠	234	Ω
145	æ	175	»	205	═	235	δ
146	Æ	176	░	206	╬	236	∞
147	ô	177	▒	207	╧	237	∅
148	ö	178	▓	208	╨	238	∈
149	ò	179	│	209	╤	239	∩
150	û	180	┤	210	╥	240	≡
151	ù	181	╡	211	╙	241	±
152	ÿ	182	╢	212	╘	242	≤
153	Ö	183	╖	213	╒	243	≥
154	Ü	184	╕	214	╓	244	⌠
155	¢	185	╣	215	╫	245	⌡
156	£	186	║	216	╪	246	÷
157	¥	187	╗	217	┘	247	≈
158	₧	188	╝	218	┌	248	°
159	ƒ	189	╜	219	█	249	•
160	á	190	╛	220	▄	250	·
161	í	191	┐	221	▌	251	√
162	ó	192	└	222	▐	252	ⁿ
163	ú	193	┴	223	▀	253	²
164	ñ	194	┬	224	α	254	■
165	Ñ	195	├	225	β	255	Space

E: Erweiterte Tastaturcodes

Die Standardfunktion *Readkey* hält das Programm so lange an, bis der Benutzer eine Taste betätigt. Die Funktion liefert dann als Ergebnis den ASCII-Wert dieser Taste (siehe Anhang D).

Auf der Tastatur gibt es zusätzlich einige Sondertasten (oder Tasten-Kombinationen), die keinem ASCII-Wert zugeordnet werden können. Wenn der Benutzer eine dieser Tasten betätigt, liefert *Readkey* den Wert 0. Wird *Readkey* gleich darauf noch einmal aufgerufen, liefert di£ Funktion den sogenannten erweiterten Tastaturcode. Dieser Code kann mit dem folgenden Programm ermittelt werden:

```
Program sondertasten;
Uses Crt;                       (* wegen Readkey *)
Var a,b : Char;
Begin
   Repeat
   a:=Readkey;
   If a=Chr(0) Then             (* falls Sondertaste *)
     Begin
      b:=Readkey;
      Writeln(Ord(b));
   End;
   Until a=Chr(27);             (* bis Esc *)
End.
```

	Taste allein	mit Shift	mit Ctrl	mit Alt
F1 bis F10	59-68	84-93	94-103	104-113
F11	133	135	137	139
F12	134	136	138	140
Home	71		119	
Pfeil hoch	72			
Page up	73		132	
Print screen			114	
Pfeil links	75		115	
5 (Zahlenblock)	76			
Pfeil rechts	77		116	
End	79		117	
Pfeil runter	80			
Page down	81		118	

	Taste allein	mit Shift	mit Ctrl	mit Alt
Insert	82			
Delete	83		255	
1 bis 0				120-129
Q bis P				16-25
A bis L				30-38
Z bis M				44-50
Tabulator	15			

F: Erklärung einiger Begriffe .

Assembler bezeichnet sowohl eine maschinennahe Programmiersprache als auch das Programm, mit dem diese Sprache in reine Maschinensprache übersetzt wird. Die Assemblersprache wird auch als Sprache der zweiten Generation bezeichnet. Der Unterschied zwischen der Assembler- und Maschinensprache liegt lediglich in der Schreibweise. Die Zahlen, die in der Maschinensprache Befehle, Register u.a. bezeichnen, sind in der Assemblersprache durch sinnvolle Abkürzungen ersetzt.

Binärsystem. Zahlensystem mit der Basis 2. Als Zahlen werden 0 und l verwendet. Bit. Die kleinstmögliche Informationseinheit. Ein Bit kann maximal zwei verschiedene Werte annehmen. Die Werte können unterschiedlich interpretiert werden, z.B. als True/False, 1/0, Ja/Nein oder +/-.

Byte. Die kleinste Informationseinheit, die von einem Computer direkt verarbeitet werden kann. Ein Byte besteht aus 8 Bits und kann dementsprechend 256 verschiedene Werte annehmen (2 hoch 8). Die Werte können unterschiedlich interpretiert werden, z.B. eine Zahl im Bereich zwischen 0 und 255 oder zwischen -128 und +127, aber auch als ein Zeichen im ASCII-Zeichensatz.

Compiler. Programm, mit dem eine Quelldatei in Objektcode übersetzt werden kann. Die Compilierung kann in einer oder mehreren Stufen, auch Passes genannt, durchgeführt werden. Der Objektcode ist nicht direkt ausführbar, sondern er muß erst mit einem Linker gebunden werden. Dabei können mehrere Objektdateien, die einzeln compiliert sind, zu einem fertigen, ausführbaren Programm zusammengefügt werden. Bei Turbo Pascal sind Compiler und Linker zu einem Programm vereint.

Datensegment s. Segment.

Editor. Ein Programm, mit dem Quelldateien geschrieben werden können. Ein Editor ist im Prinzip ein einfaches Textverarbeitungsprogramm. Deshalb können die meisten Textverarbeitungsprogramme auch als Editoren benutzt werden. Die Texte werden als einfache, unformatierte ASCII-Dateien gespeichert.

Hauptquelldatei. In Turbo Pascal ist das die Quelldatei mit dem Hauptmodul. Die ersten Zeilen in dieser Datei müssen den Programmnamen und eine eventuelle Uses-Deklaration enthalten.

Hexadezimalsystem. Zahlensystem mit der Basis 16. Als Zahlen werden die Ziffern 0 bis 9 sowie die Buchstaben A bis F verwendet, wobei die Buchstaben für die dezimalen Zahlen 10 bis 15 stehen. Eine zweistelligen Hexadezimalzahl (00 bis FF) hat einen Wertebereich zwischen 0 und 255 (dezimal) und entspricht damit einem Byte.

Integerdivision unterscheidet sich von der normalen Division dadurch, daß sowohl beide Operanden als auch das Ergebnis Ganzzahlen sind. Ein eventueller Divisionsrest wird nicht beachtet. Siehe auch Modula-Division.
Beispiel: 11 Div 4 = 2

Linker. Programm, mit dem eine oder mehrere Objektdateien zusammengefügt und in ein ausführbares Programm umgesetzt werden können. Bei Turbo Pascal sind Compiler und Linker zu einem Programm vereint.

Maschinensprache. Die Programmiersprache, die direkt vom Computer verstanden wird. Wird auch als Programmiersprache der ersten Generation bezeichnet. Sie besteht ausschließlich aus Bytes.

Modula-Division. Division, in der sowohl beide Operanden als auch das Ergebnis Ganzzahlen sind. Eine Modula-Division liefert als Ergebnis den Divisionsrest.
Beispiel: 11 Mod 4 = 3

Programmsegment s. Segment.

Quellprogramm. Ein Programm in Assembler oder in einer Hochsprache wie Turbo Pascal geschrieben. Das Programm ist nicht ausführbar, sondern muß zuerst mit einem Assembler bzw. einem Compiler in Maschinensprache übersetzt werden.

Segment. Unter MS-DOS wird der Hauptspeicher in Segmente aufgeteilt. Ein Segment kann höchstens 64 KByte groß sein. Ein Programm besteht aus einem oder mehreren Programmsegmenten mit Programmcode und einem Datensegment mit Variablen und typenbestimmten Konstanten. In Turbo Pascal wird jedes Unit in einem eigenen Programmsegment abgelegt.

Shift links, rechts. Rechenoperation, mit der alle Bits in einem Operanden um eine bestimmte Anzahl von Plätzen nach links oder rechts verschoben werden. Damit

gehen an der einen Seite einige Bits verloren. An der anderen Seite wird mit Null-Bits aufgefüllt. Die Ergebnisse von *Shift Links* und *Shift Rechts* entsprechen einer Multiplikation bzw. einer Integerdivision mit einer Potenz von 2.
Beispiel: 5 Shl 3 = 40   (entspricht: 5*(2 ^ 3)=40)

Verknüpfung. Rechenoperation, in der zwei Operanden miteinander verbunden werden. Man unterscheidet zwischen der logischen und der bitweisen Verknüpfung. Außerdem gibt es drei Arten der Verknüpfung: And, Or und Xor. Bei der logischen Verknüpfung werden zwei Boolean-Operanden nach dem folgenden Schema verbunden:

	And	Or	XOr
True, True	True	True	False
True, False	False	True	True
False, True	False	True	True
False, False	True	False	False

Bei der bitweisen Verknüpfung werden alle Bits in den beiden Operanden paarweise nach demselben Schema verbunden.

Helbo

Turbo Pascal

Dieses Buch vermittelt einen Einstieg in die Arbeit mit Turbo Pascal. Es enthält einige größere Beispielprogramme, z.B. ein Programm zur Verwaltung von Büchern. Daran zeigt der Autor schrittweise, wie die Programmierung in Turbo Pascal vonstatten geht.
Aus einfachen Problemen heraus führt er an die zur Lösung notwendigen Prozeduren und Befehle heran. Die konkrete Umsetzung des Kennengelernten geschieht in zahlreichen Beispielen zum Nachvollziehen. So erfährt der Leser, wie man mit Dateien umgeht, eigene Units schreibt, Arrays definiert, Programme in Prozeduren und Funktionen organisiert usw. Die Kapitel sind weitgehend unabnhängig voneinander aufgebaut, so daß sich nicht nur der Neuling, sondern auch der erfahrenere Programmierer gezielt z. B. über Turtlegrafik oder objektorienterte Programmierung (OOP) informieren kann. Der Anhang listet überblicksartig alle zentralen Pascal-Routinen mit ihrer Syntax auf. Damit dient das Buch zusätzlich als praktische Referenz beim Experimentieren mit eigenen Programmen.
Das Buch wurde auf Grundlage der Version 5.0 geschrieben. Da aber die Versionen 4.0 und 5.0 weitgehend kompatibel sind, können auch Besitzer älterer Versionen damit arbeiten.

03200
9 783772 367625
ISB N 3-7723-6762-3 DM +032.00